



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

SPLC 2006 and Beyond

The Software Engineering Institute was the proud sponsor of the 2006 Software Product Line Conference. As the tenth official gathering of the software product line community, SPLC 2006 provided a venue for 190 + practitioners, researchers, and educators from 22 countries to reflect on the achievements made during the past decade, assess the current state of the field, and identify key challenges still facing researchers and practitioners. The conference featured 2 keynote presentations, 16 research and 7 experience papers, 3 panels, 14 tutorials, 4 workshops, 5 demonstrations, birds-of-a-feather discussions, and the 2006 hall of fame induction. The [Proceedings of SPLC 2006](#) are available from IEEE Computer Society (ISBN 0-7695-2599-7).

We would like to thank all those who contributed to the success of this conference and look forward to seeing you at the [11th International Software Product Line Conference](#) (SPLC 2007), 10-14 September 2007 in Kyoto, Japan.

[John D. McGregor](#), Clemson University (Conference Chair)

Frank van der Linden, Philips Medical Systems (Program Chair)

Robert L. Nord, Software Engineering Institute (Program Chair)

Post-Conference Information

- [Welcome Message](#) - John D. McGregor

- [Software Product Lines Automate Development](#), Darryl K. Taft, eWeek, 25 August 2006.

- Conference Presentations

- Keynotes

- [The Option Value of Software Product Lines](#)

- Carliss Baldwin

- [Aspect-Oriented Programming Radical Research in Modularity](#)

- Gregor Kiczales

- Workshops

- [APLE - 1st International Workshop on Agile Product Line Engineering](#)

- [Managing Variability for Software Product Lines: Working With Variability Mechanisms](#)

- [Variability Management Working with Variability Mechanisms](#)

- [SPLiT'06: 3rd Workshop on Software Product Line Testing](#)

- [OSSPL - First International Workshop on Open Source Software and Product Lines](#)

- Open source strengths for defining software product line practices ([paper](#), [presentation](#))

- Feature-Oriented Determination of Product Line Asset Types: In-House, COTS, or Open Source? ([paper](#), [presentation](#))

- Open Source in the Software Product Line: An Inevitable Trajectory? ([paper](#), [presentation](#))

- OSS Product Family Engineering ([paper](#), [presentation](#))

- [Software Product Lines Doctoral Symposium](#)

- Panels
 - Product Derivation Approaches
 - [Building Interactive TV Applications with pure::variants](#)
Danilo Beuche
 - [BigLever Software Gears Solution](#)
Charles Krueger
 - [Product Derivation Panel - Domain-Specific Modeling](#)
Juha-Pekka Tolvanen
 - [Product Derivation Approaches: The Digital TV case and Koala](#)
Rob van Ommering
 - [Testing in a Software Product Line](#)
 - Product Line Research
 - [Introduction](#)
Liam O'Brien
 - [Product Line Research - Panel Statement](#)
Dirk Muthig
 - [Software Product Line Research Topics](#)
Kyo Kang
 - [Lessons Learned from the last 10 years and Directions for the next 10 years](#)
Klaus Pohl
 - [Lessons Learned from the Last Ten Years and Directions for the Next Ten](#)
Paul Clements
- Paper Presentations
 - [Product Line Adoption: A Vice President's View & Lessons Learned](#)
Salah Jarrad
 - [New Methods in Software Product Line Development](#)
Charles W. Krueger
- 2006 Software Product Line Hall of Fame
 - Inducted from SPLC-Europe 2005
 - [RAID controller firmware product line, LSI Logic - Engenio Storage Group](#)
 - Nominated for induction at SPLC 2007
 - [Bosch Gasoline Systems: Engine Control Software Product Line](#)
 - Philips Low-end Television Product Line

Pre-Conference Information

- | | |
|---|---|
| ● Keynote Speakers | ● Registration |
| ● Technical Program | ● Important Dates |
| ● Workshops | ● Location/Hotel Reservation |
| ● Tutorials | ● Past Conferences |
| ● Panels | ● Conference & Program Committees |
| ● Demonstrations | ● Corporate Supporters |
| ● Software Product Lines Doctoral Symposium | ● News Items about SPLC |
| ● Software Product Line Hall of Fame | ● Help Publicize SPLC 2006 |
| ● Birds-of-a-Feather Sessions | |



Contact Information:

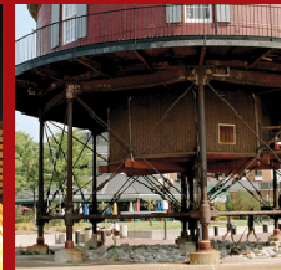
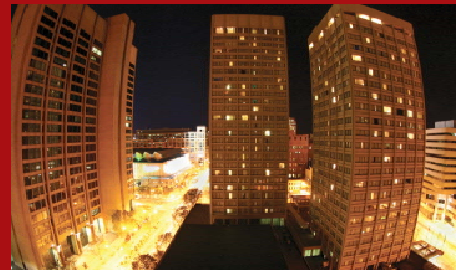
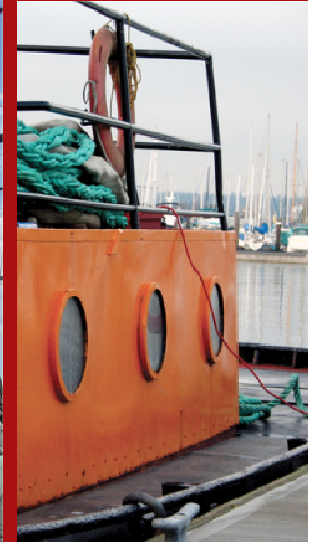
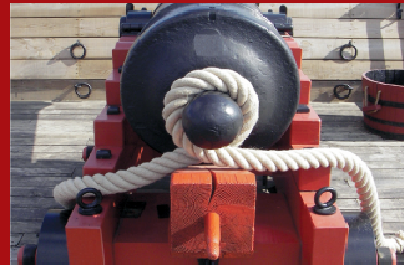
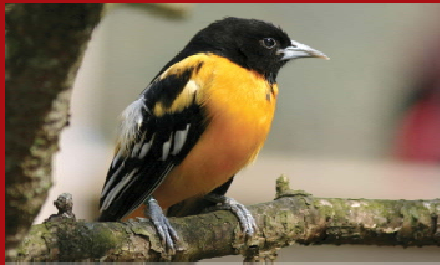
For general information, contact [John D. McGregor](#).
For web site information, contact [Bob Krut](#).

Location:

[Baltimore Marriott Waterfront](#)
[Baltimore](#), Maryland, USA

SPLC 2006

10th International Software Product Line Conference
August 21-24, 2006 | Inner Harbor | Baltimore, Maryland



Welcome to SPLC 2006!!

John D. McGregor
August 23, 2006



Software Engineering Institute

Carnegie Mellon

© 2006 Carnegie Mellon University



Demographics

Here is where we come from and who we are:

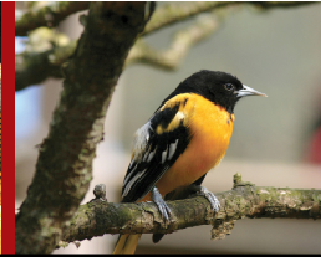
22 countries; five with most representation:

United States	97	Commercial	66
Germany	22	Academic	51
South Korea	15	Defense industry	20
Japan	10	Defense agency	4
Finland	6	Civil agency	4

Total at start of Wednesday - 194



SPLC 2006



The story so far

We have already had:

food

14 tutorials

food

4 research workshops &
the Doctoral Symposium

food

Conference Reception



© 2006 Carnegie Mellon University



Software Engineering Institute

Carnegie Mellon

SPLC2006 Conference | Welcome

Page 3



**SPLC
2006**

Conference HighLights

Today is a long day with lots to do, you can rest when you get home

We kept the Europeans' 1.5 hour lunches from SPLC 2005 but Americanized them and we will have working lunches – demos will be conducted during lunch and you will be able to take your lunch in to see the demos in which you are interested.

Don't forget the SEI Reception and BoFs tonight.

SEI's DoD BoF

BigLever's BoF

Research BoF

© 2006 Carnegie Mellon University



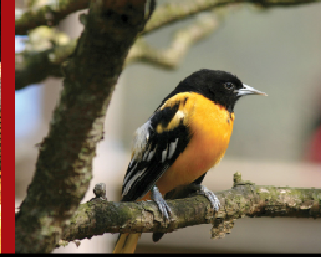
Software Engineering Institute

CarnegieMellon

SPLC2006 Conference | Welcome

Page 4

**SPLC
2006**



Conference HighLights

Don't miss

the Hall of Fame

Ice Cream Social

© 2006 Carnegie Mellon University



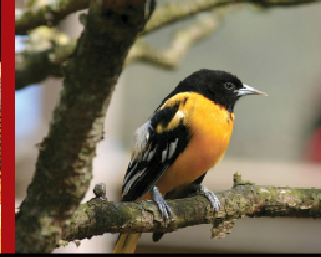
Software Engineering Institute

Carnegie Mellon

SPLC2006 Conference | Welcome

Page 5

SPLC 2006



www.SPLC.net

We expect the website to become a community resource

Any panel, research workshop, or other non-proceedings material may be added to the site to make it widely available

Check the site periodically as the SPLC2007 crew builds their program

© 2006 Carnegie Mellon University



Software Engineering Institute

Carnegie Mellon

SPLC2006 Conference | Welcome

Page 6



Thanks to

Frank van der Linden – *Program Chair Philips Medical Systems*

Robert L. Nord – *Program Chair Software Engineering Institute*

Daniel J. Paulish – *Tutorials Chair Siemens Corporate Research*

Birgit (Mom) Geppert – *Workshop Chair - Avaya Labs*

Isabel John – *Doctoral Symposium Chair - Fraunhofer Institute for Experimental Software Engineering*

Dave M. Weiss – *Hall of Fame Chair - Avaya Labs*

Patrick Donohoe – *Public Relations Chair - SEI*

Liam O'Brien – *Proceedings Editor - Lero, The Irish Software Engineering Research Centre*

Melissa L. Russ – *Local Publicity and Arrangements - Space Telescope Science Institute*

Bob Krut – *Web Chair - SEI*

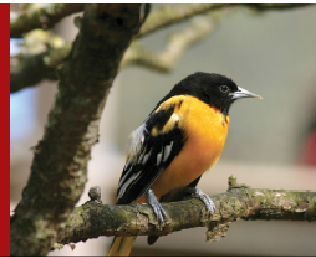
Ruth Lynn Gregg – *Registration and Logistics - SEI*

Pennie Walters, Daniel Pipitone, Bob Fantazier, David Gregg – *Printed Materials - SEI*

Carole Mann – *Registration Registration Systems Lab*



**SPLC
2006**



Welcome Alexander



© 2006 Carnegie Mellon University



Software Engineering Institute

Carnegie Mellon

SPLC2006 Conference | Welcome

Page 8

**SPLC
2006**



Thanks to the SPLC Steering Committee

Linda Northrop

*Software Engineering Institute,
Chair*

Len Bass

Software Engineering Institute

Paul Clements

Software Engineering Institute

Kyo C. Kang

POSTECH

John McGregor

Clemson University

Henk Obbink

Philips

Frank van der Linden

Philips Medical Systems

David M. Weiss

Avaya Labs

© 2006 Carnegie Mellon University



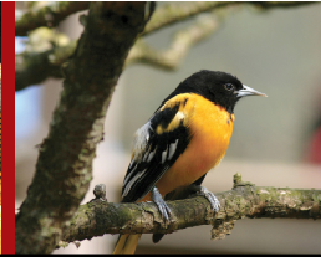
Software Engineering Institute

Carnegie Mellon

SPLC2006 Conference | Welcome

Page 9

SPLC 2006



SPLC 2006 THANKS OUR GENEROUS SPONSORS

SPLC 2006 is sponsored by



Software Engineering Institute

CarnegieMellon

Gold-Level Corporate Supporters



Microsoft®

PHILIPS

Silver-Level Corporate Supporters

AVAYA



NOKIA



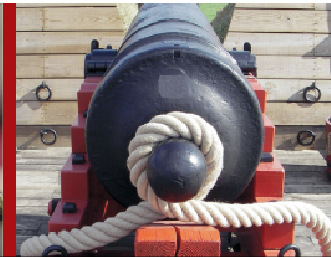
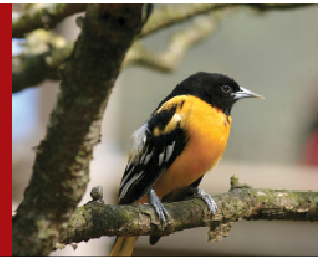
Software Engineering Institute

CarnegieMellon

SPLC2006 Conference | Welcome

Page 10

**SPLC
2006**



Program HighLights

Rod Nord, program co-chair

© 2006 Carnegie Mellon University



Software Engineering Institute

CarnegieMellon

SPLC2006 Conference | Welcome

Page 11

SPLC 2006



Here's Linda

Linda Northrop, chair of the SPLC steering committee, will add a welcome from the SPLC steering committee and introduce today's keynote speaker.

© 2006 Carnegie Mellon University

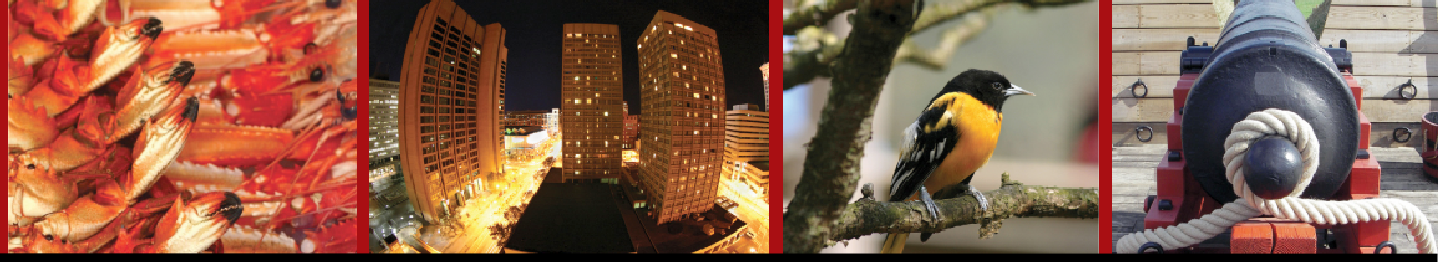


Software Engineering Institute

Carnegie Mellon

SPLC2006 Conference | Welcome

Page 12



Text Format

Regular text format is 20 pt, SEI Blue, single line-spaced (1) and 0.5 line space after. Text highlight example *is here. Or here.*

- First level indent is 20 pt, black, single line-spaced (1) and 0.5 line space after.
 - Second level indent is 20 pt, black, single line-spaced (1) and 0.5 line space after. Dash leader is used.
 - Third level indent is 20 pt, black, single line-spaced (1) and 0.5 line space after. Open Dot leader is used.
 - Second level indent is 20 pt, black, single line-spaced (1) and 0.5 line space after. Dash leader is used.



The Option Value of Software Product Lines

Carliss Y. Baldwin
Harvard Business School

SPLC 06
Baltimore, MD
August 23, 2006

Unmanageable Designs—What They Are and their Financial Consequences

Carliss Y. Baldwin
Harvard Business School

SPLC 06
Baltimore, MD
August 23, 2006

Three Points to begin

- u Large, complex, evolving designs
 - Are a fact of modern life
 - Need *design architectures*—
 - » “Description of the entities in a system and their relationships”
 - » Way of assigning work (Parnas)
- u Designs create option value
 - Value operates like a force in the economy
 - We fight to create it and to keep it—using *strategy, including product line strategy*
- u Design Architecture, Option Value and Strategy
 - *How can you create and capture value in a large, complex evolving set of designs?*
 - Subject of this talk

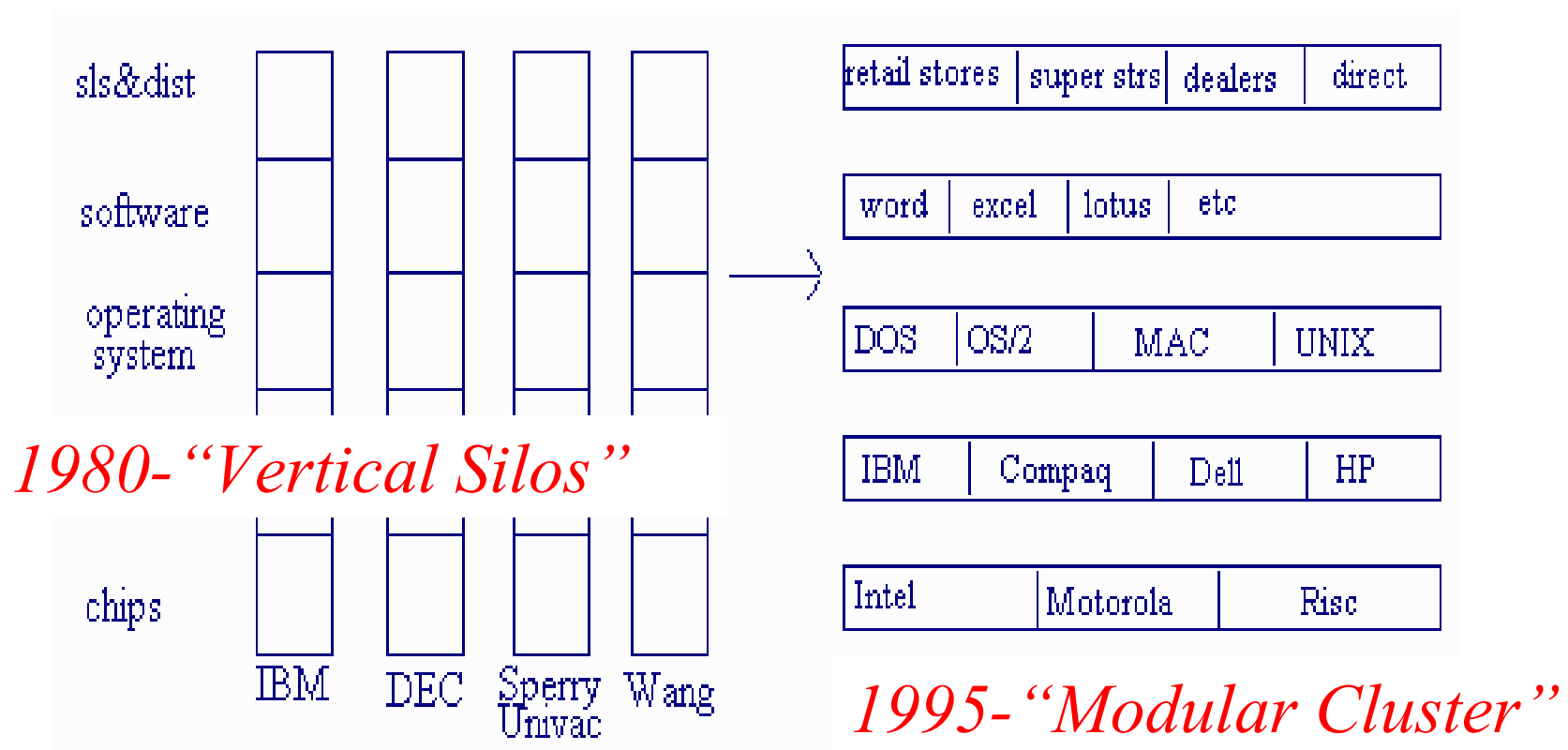
In the economy, value acts
like a force

Value = money or the promise of money

Consider the computer industry...

The changing structure of the computer industry

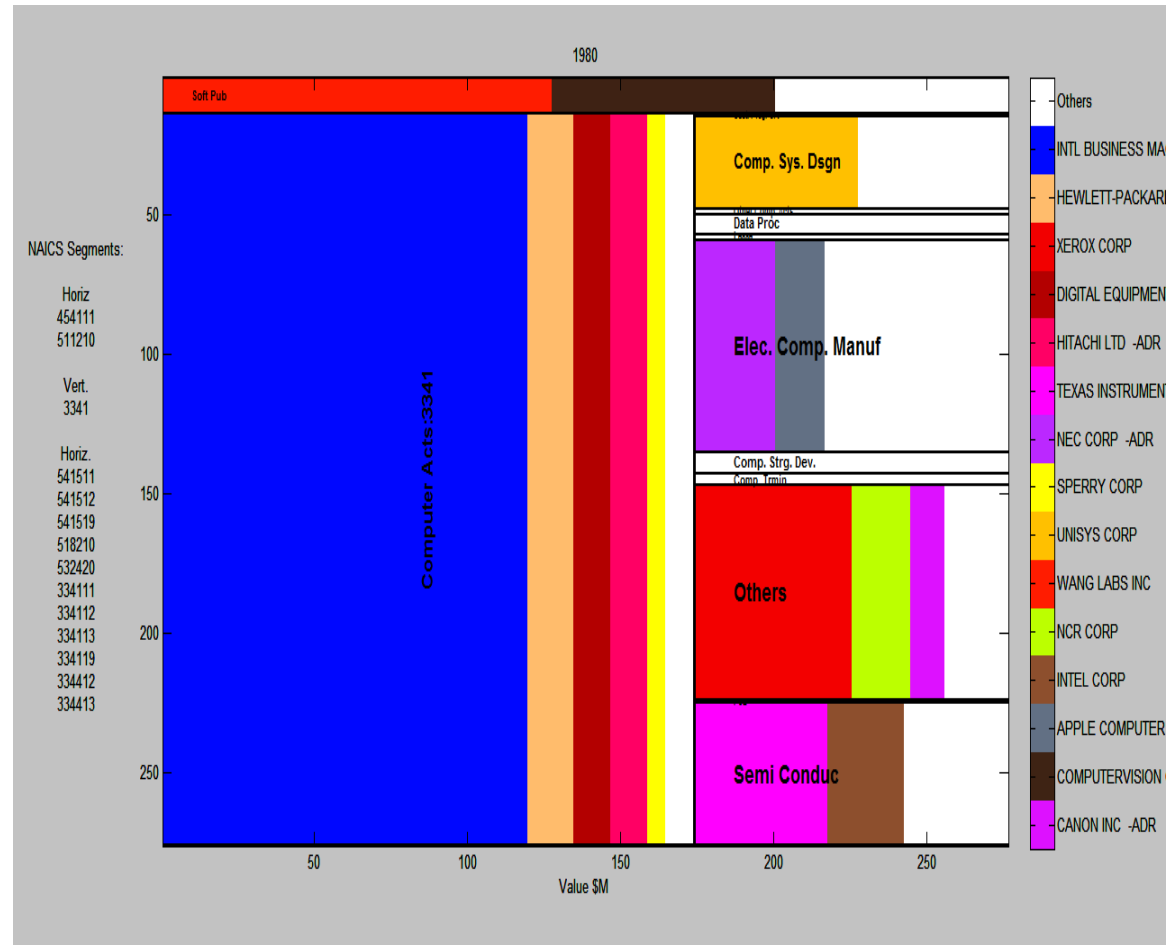
- u Andy Grove described a **vertical-to-horizontal transition** in the computer industry:



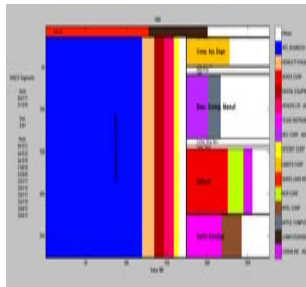
Andy's Movie Stack View in 1980

Top 15 Public Companies in US Computer Industry

Area reflects
Market Value
in Constant
US \$

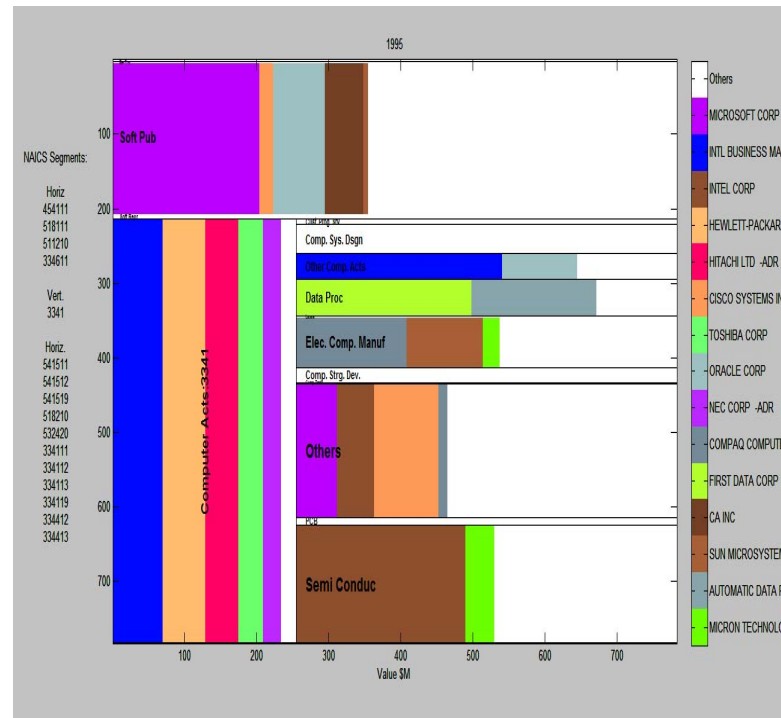


Andy's Movie Stack View in 1995

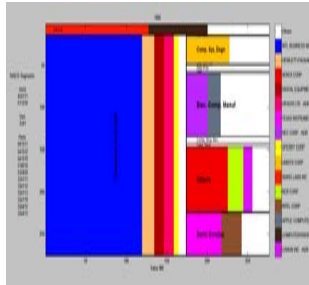


**Top 15 Public
Companies in
US Computer
Industry**

**Area reflects
Market Value
in Constant
US \$**

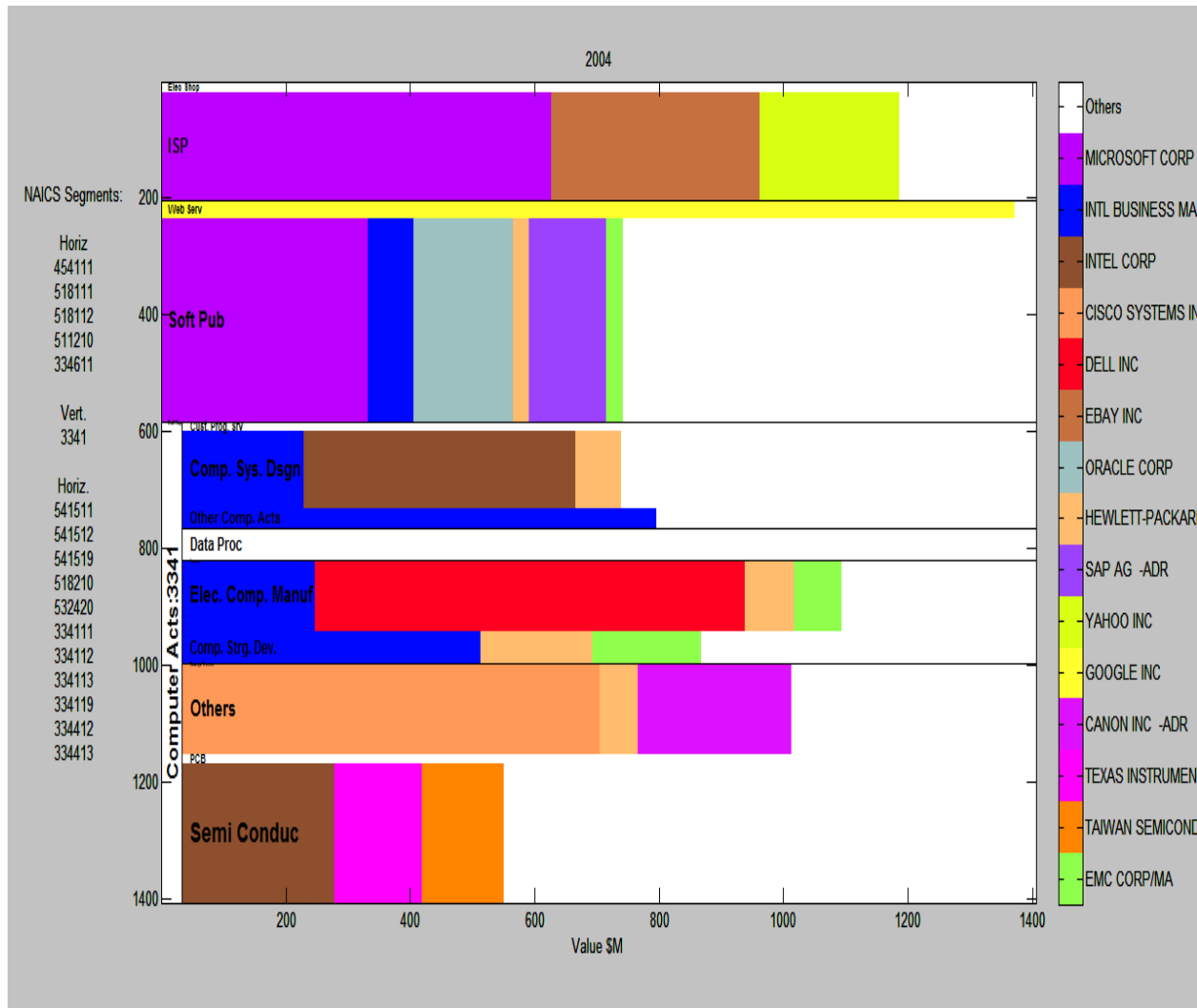


Andy's Movie Stack View in 2004



**Top 15 Public
Companies in
US Computer
Industry**

**Area reflects
Market Value
in Constant
US \$**



Turbulence in the Stack

Departures from Top 15:

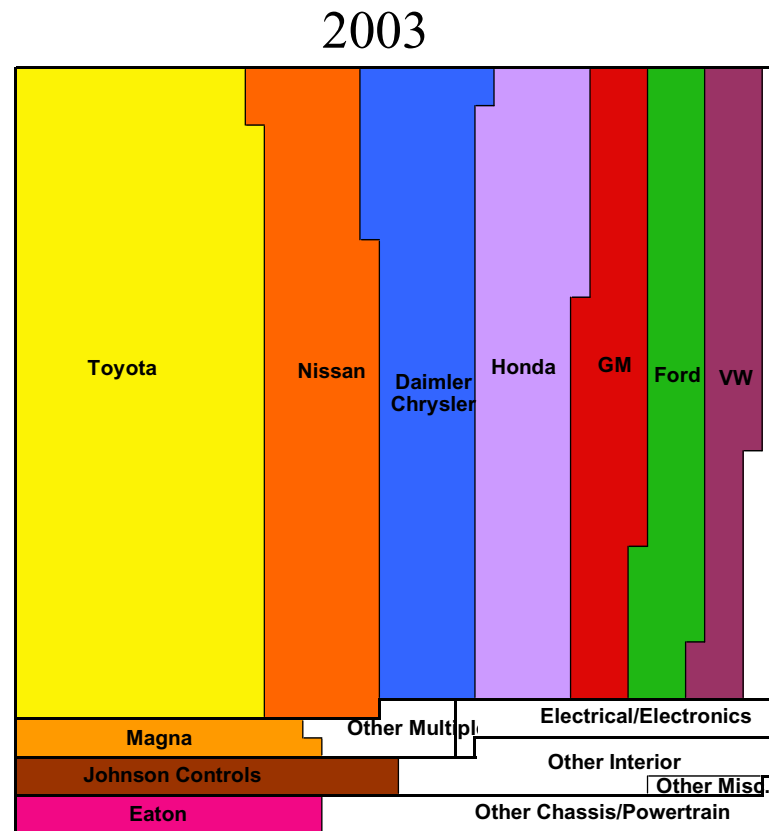
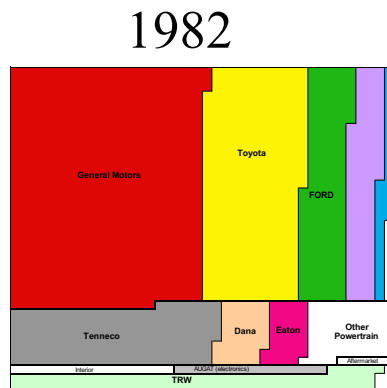
- u Xerox (~ bankrupt)
- u DEC (bought)
- u Hitachi
- u NEC
- u Sperry (bought)
- u Unisys (marginal)
- u Wang (bankrupt)
- u NCR (bought)
- u Computervision (LBO)

Arrivals to Top 15:

- u Microsoft
- u Cisco
- u Google
- u Dell
- u EBay
- u Yahoo
- u SAP
- u Taiwan Semiconductor
- u First Data

Sic Transit Gloria Mundi ... Sic Transit

Contrast to the Auto Industry



**Top 10 Public
Companies in
US Auto
Industry**

**Area reflects
Market Value
in Constant
US \$**

Value stayed in one layer!

Two patterns

- u “Manageable” designs = auto industry
- u “Unmanageable” designs = computer industry

What makes computer designs so unmanageable?

This was the question Kim Clark and I set out to answer in 1987.

After studying the history of computer designs and correlating their changes with value changes

We concluded that *modularity* was part of the answer...

Modularity is

- u The degree to which a set of designs (or tasks) is partitioned into components, called modules, that are
- u highly dependent within a module, nearly independent across modules
- u A property of architecture
- u Somewhat under the architect's control

Modular Architecture?

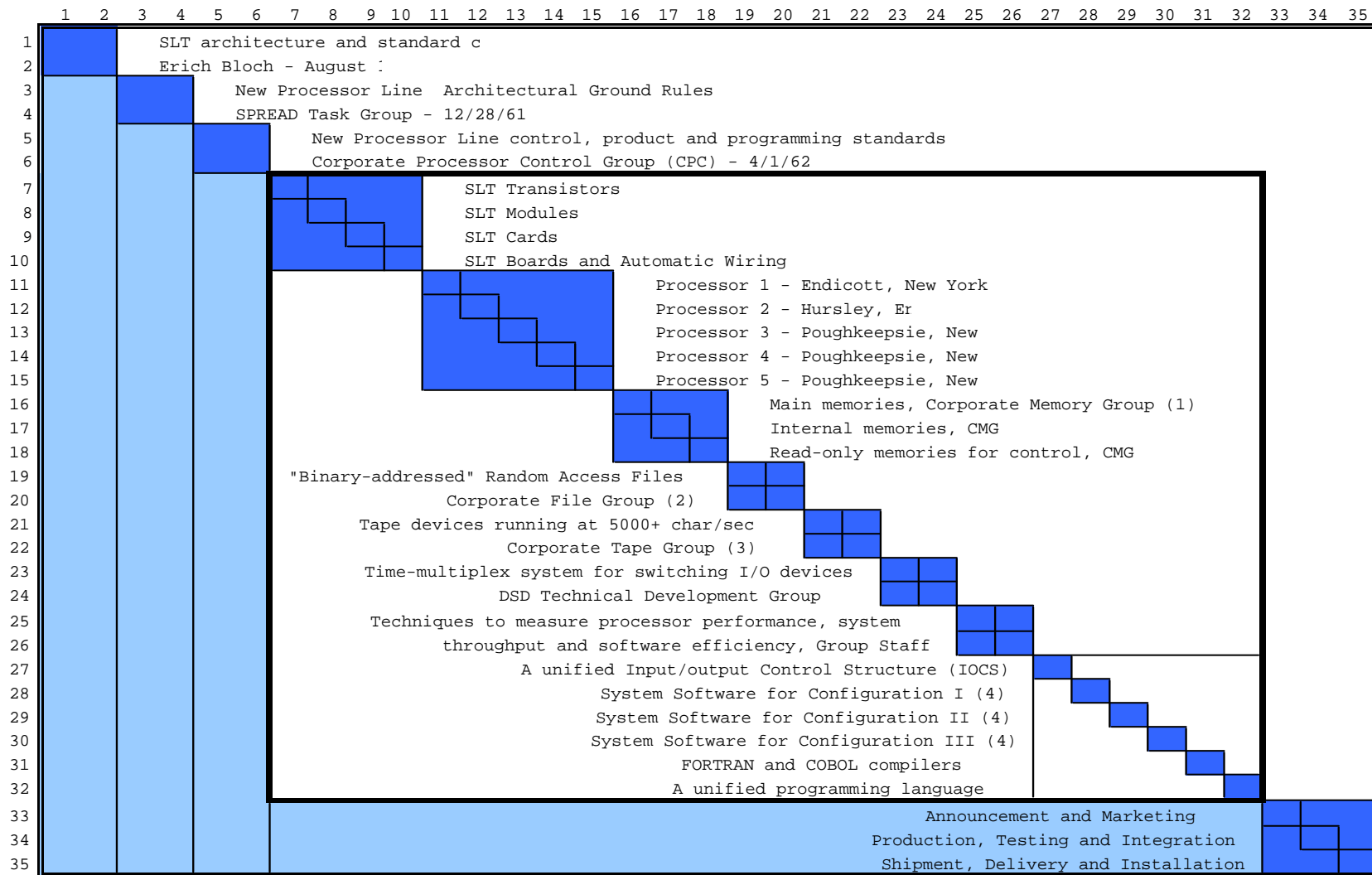
- u Modules = Entities with few relationships
 - “Near-decomposable”
 - “Loosely coupled”
- u *Modular architectural view* describes modules, ie, components and links
 - Less detailed than “full” architecture
 - Locates, does not describe interfaces
 - Strategist’s perspective, not engineer’s

Modular Architectural Views

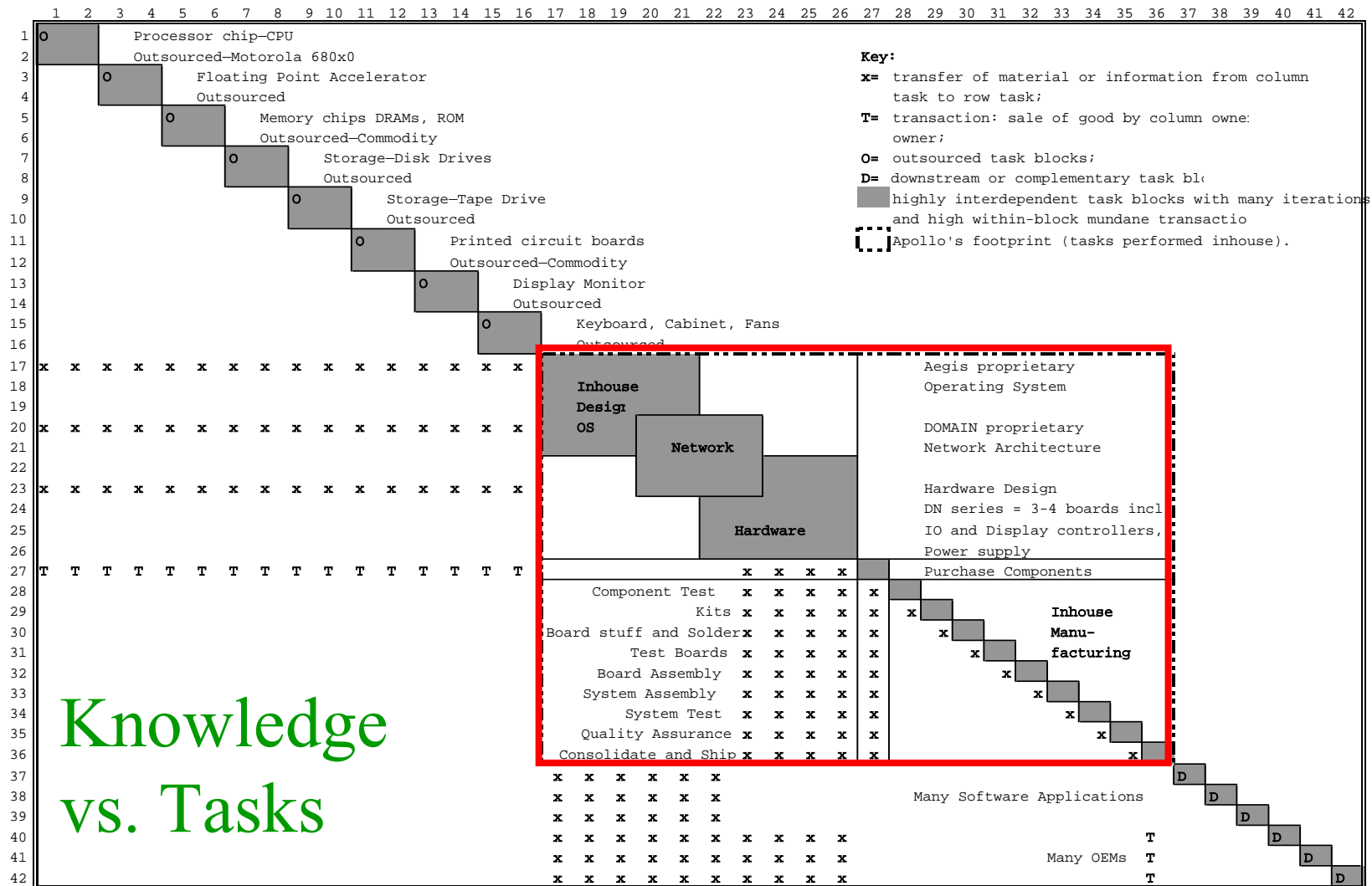
Pictures of entities and links...

But not interfaces

Design and Production Architecture of IBM System/360

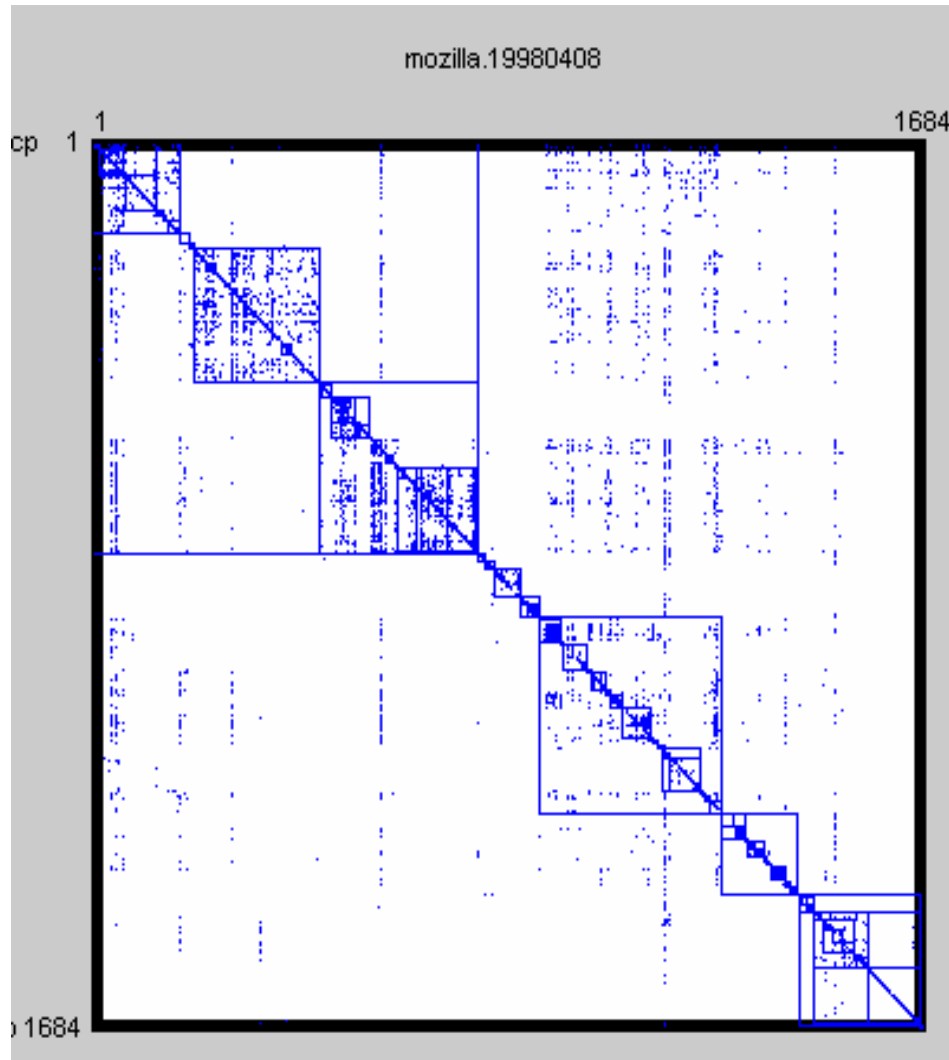


Design and Production Architecture of Engineering Workstation (Apollo)

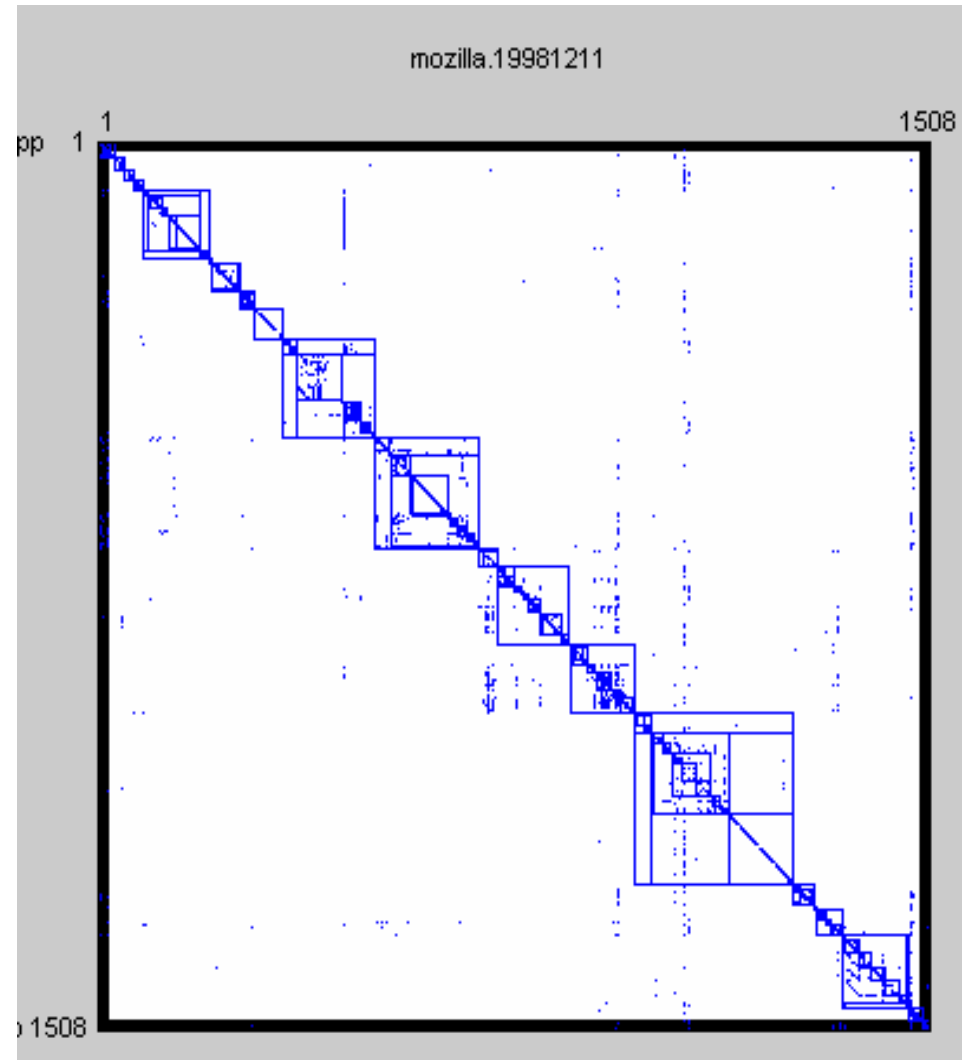


Two Browser Architectures: Entities = Files; Links = Function Calls

Mozilla Before Redesign



Mozilla After Redesign



The system's *Modular Architecture* determines its options hence option value

- u Modules = Units of DESIGN
- u Design are options
 - Can always stay with the old design

So...

- u Modules are Carriers of Option Value

This is the “*Power of Modularity*”

Modularity, Options and Product Lines

User Group	A	B	C	D	E
Potential Revenue	\$1,000	\$500	\$1,500	\$500	\$1,000

A multi-component software product addresses the needs of each group

Design Cost = \$100 per component

Total Cost = \$10

An Option is

- u The *right but not the obligation* to take an action
 - Action = Use a new design
 - If new is better than old, use new;
 - Otherwise, keep the old.
- u Each group (with potential revenue) is an option
 - To design a codebase or not

With an integral architecture, which groups are “in the money”?

User Group	A	B	C	D	E
Potential Revenue	\$1,000	\$500	\$1,500	\$500	\$1,000

A multi-component software product addresses the needs of each group

Design Cost = \$100 per component

Total Cost = \$10

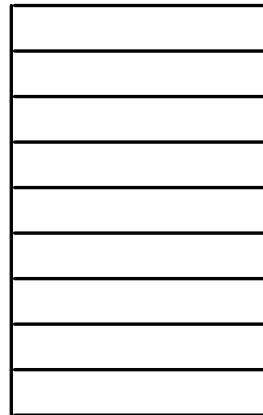
Only Group C (“Center”) is in the money.

Total Profit = \$1,500 - \$1,000 = \$500

A Change of Architecture

User Group	A	B	C	D	E
Potential Revenue	\$1,000	\$500	\$1,500	\$500	\$1,000

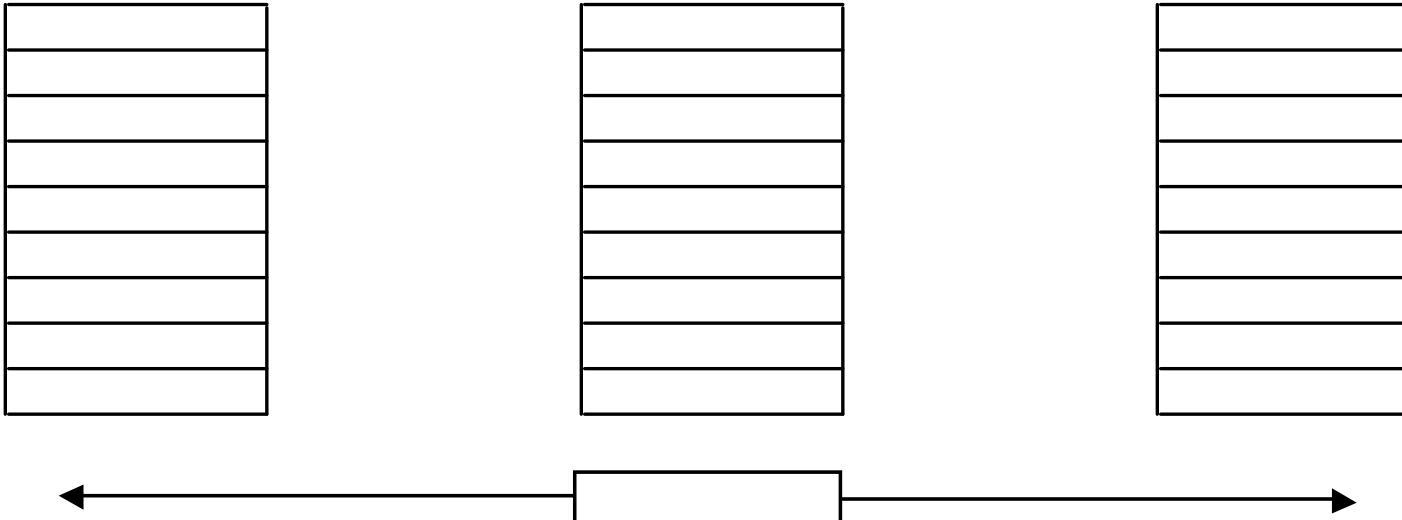
9 Custom
Components
and 1 Shared
Component



Which groups are “in the money”?

With one shared component, which groups are “in the money”?

User Group	A	B	C	D	E
Potential Revenue	\$1,000	\$500	\$1,500	\$500	\$1,000
	<div></div>		<div></div>		<div></div>



Groups A and E are now “in the money”!

Total Profit = \$500 + \$

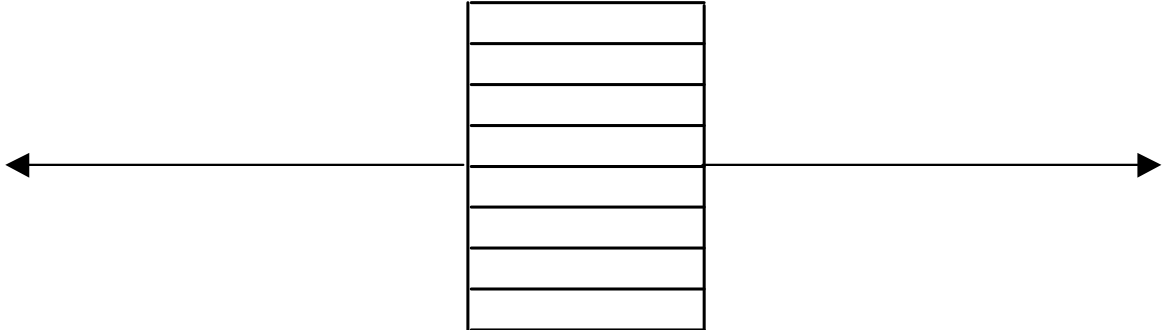
Extra Option Value

Option Terms

- u Value of “underlying” asset, V
 - Revenue potential of each group
- u Exercise (or “strike”) price, E
 - Cost of customized components
- u Option Value
 - $\text{Max}(V-E, 0)$
 - If $V-E < 0$, don't exercise, get 0.
- u Cost of Option, C
 - Cost of shared component
- u Net Option Value
 - Sum of Option Values minus Option Cost
 - $\$600 + \$100 + \$100 - \$100 = \$700$

With an 8-component Platform and 5 Modules...

User Group	A	B	C	D	E
Potential Revenue	\$1,000	\$500	\$1,500	\$500	\$1,000
	Module 1	Module 1	Module 1	Module 1	Module 1



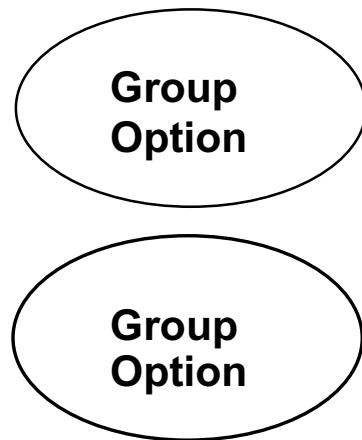
Exercise Price (per group) = \$200 —ALL groups in the money!

Option Values: A=\$800; B=300; C=1300; D=300; E=800.

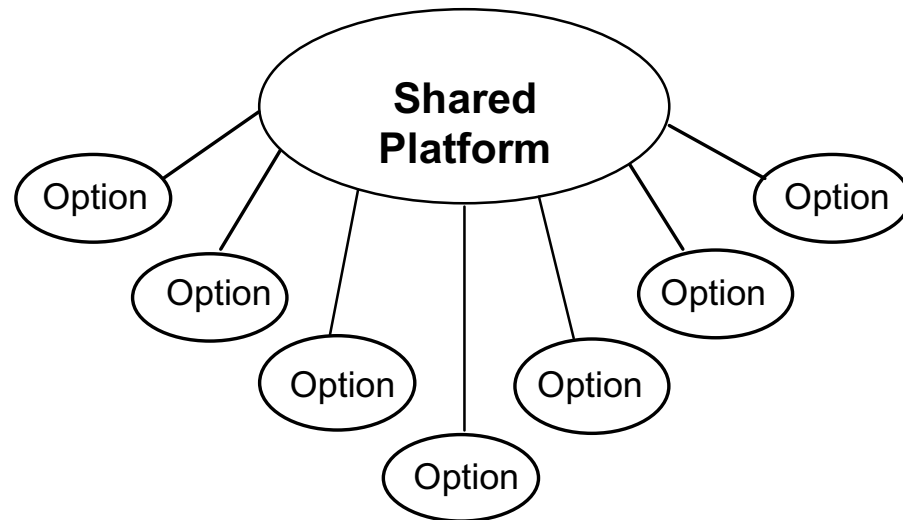
**Net Option Value: \$800 + 300 + 1300 + 300 + 800 – 800
= \$2700!**

Conclusion: Shared platforms reduce exercise cost per group

System Before Platform



System after Platform



Result: (1) More group options are in the money; (2) More profit

So far, nothing seems
“unmanageable”...

BUT, what has been designed can
be re-designed

“Unmanageable” Design Architectures...

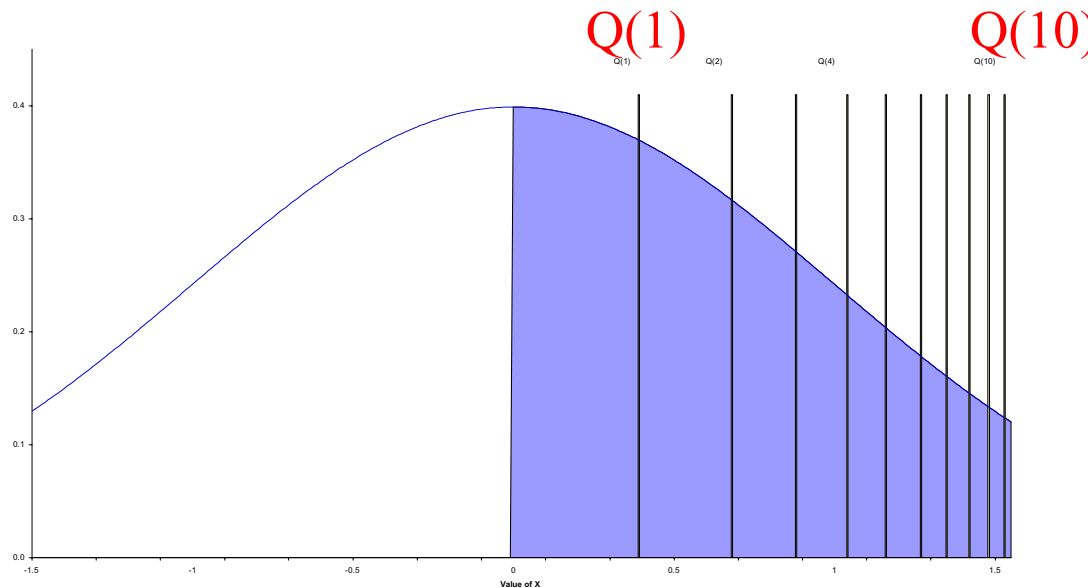
Modular, i.e, have lots of modules

+

Very High Option Potential, i.e. **High**
option value of Redesign

Option Potential

- u Probabilistic concept—value in the “right tail” of a distribution of outcomes



$Q(k) =$
Expected value
of the best of k
trials

The higher is
 $Q(k)$, the
higher the
module's
option
potential

High Option Potential + Low Strike Price

- u Pays for LOTS of experiments
- u Lots of experiments => lots of turnover of designs
- u Lots of entry =>

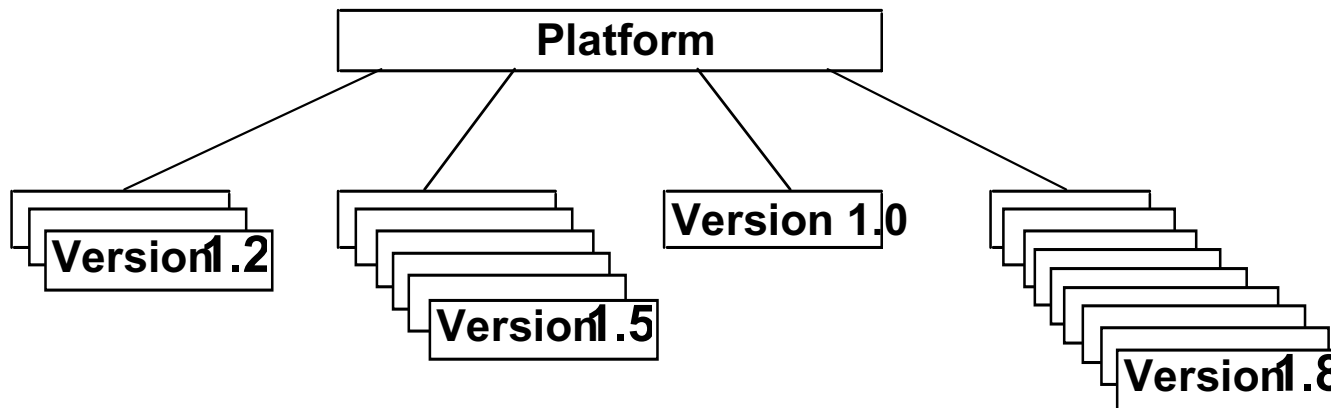
Market Turbulence

The dark side of a platform is ...

Competition in modules!

Measuring Option Potential

- u Successive, improving versions are evidence of option potential being realized over time—after the fact
- u Designers see option potential before the fact
- u What do they see?



OP = Low

Medium

Zero

High

Major challenge in research and practice right now

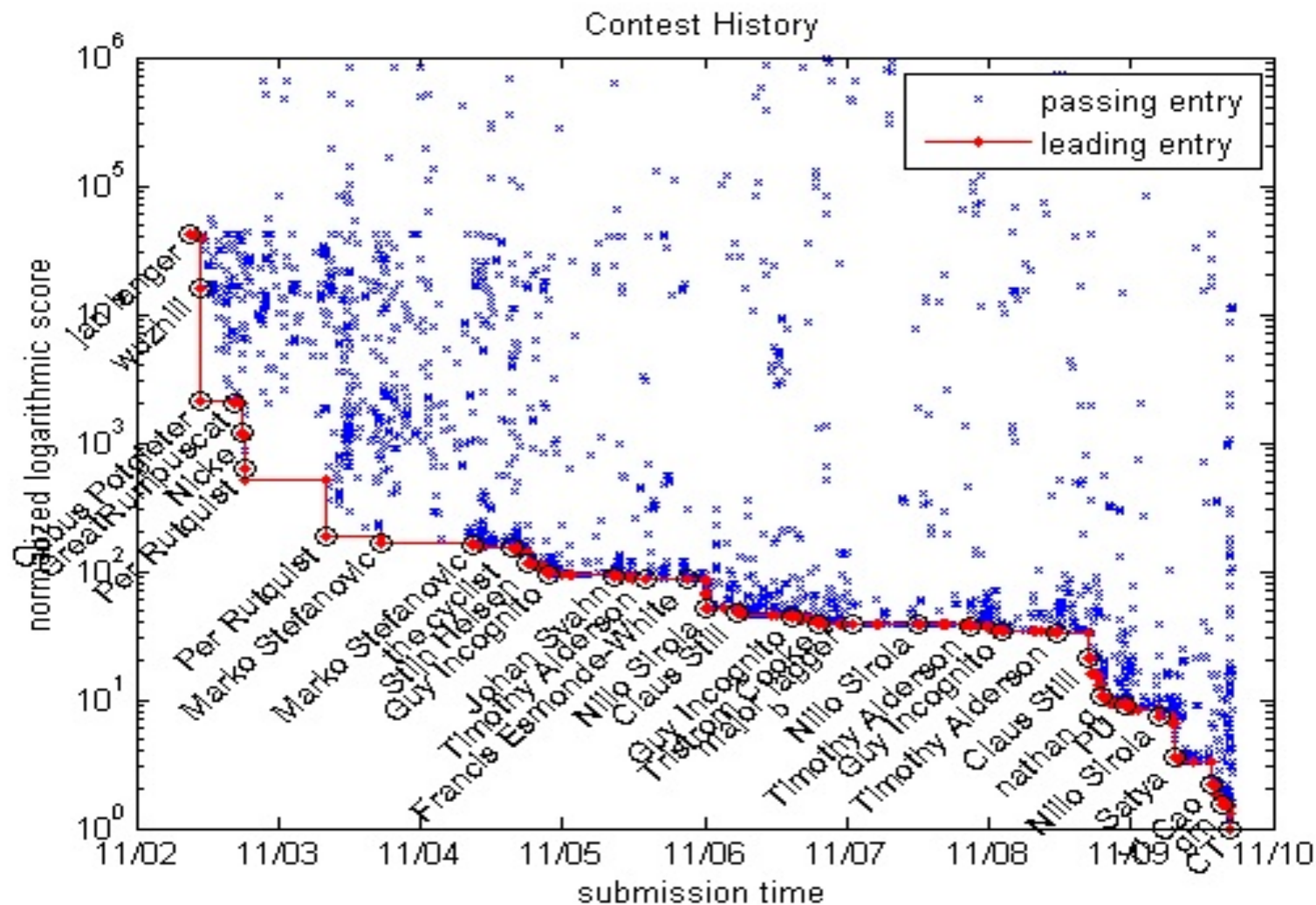
Science may not be able to deliver tools to measure *ex ante* option potential reliably

But *ex ante* estimates are what's needed

Option potential is like dark matter in the universe

- u Scientists can measure its effects but we can't measure "it"
- u "Wizards" can perceive option potential
 - But wizards don't talk to scientists!
- u Thus we lack ways to measure option value scientifically
 - It is a "research frontier"

Option potential at work— Matlab programming contest



Sources of option potential

u Physics—

- Moore's Law (dynamics of miniaturization) applies to MOSFET circuits and systems (Mead and Conway)
- Power and heat systems vs. logic systems (Dan Whitney)

u User innovation

- Users' discovery of their own needs
- “Killer apps”

u Architecture

- Experimenting with different relationships among components

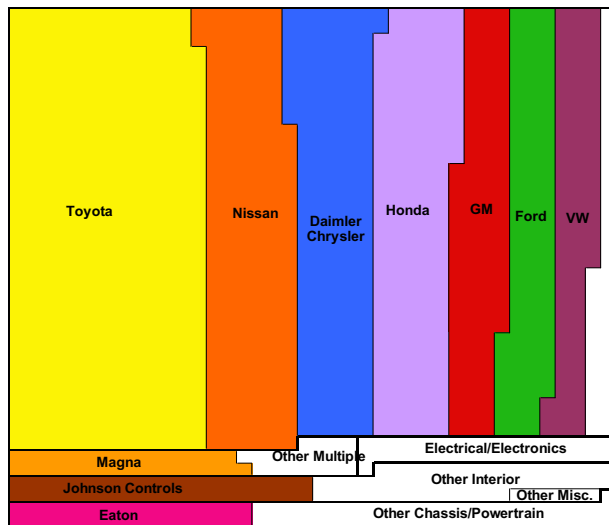
Product Lines!

Baldwin-Clark Conjecture

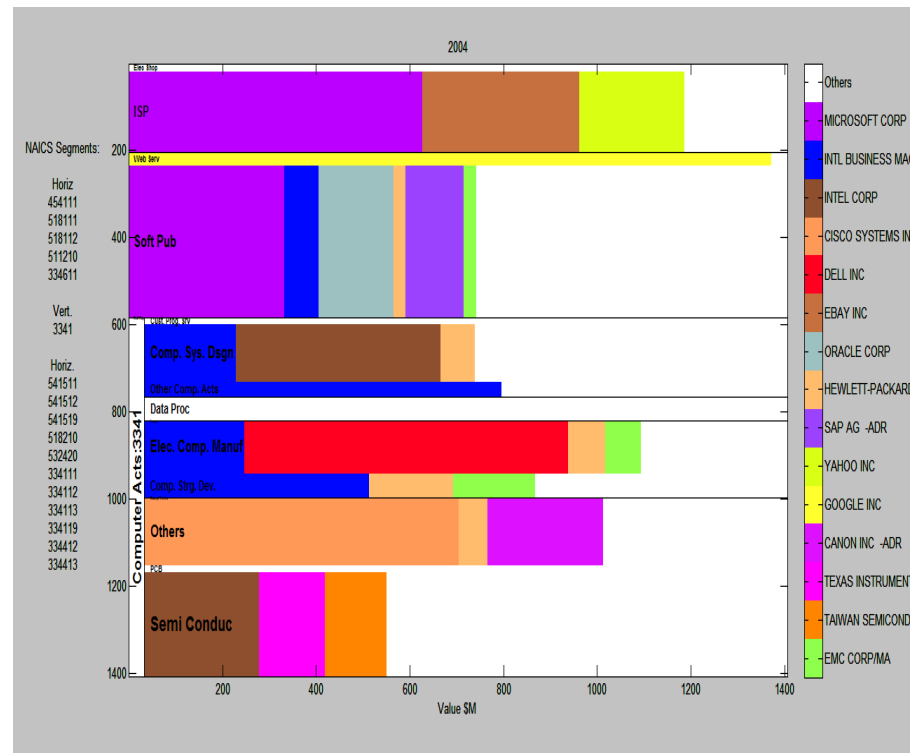
- u Need BOTH modularity AND option value to get rapid design evolution (“unmanageable designs”);
- u With rapid design evolution comes industry instability and turbulence;
- u Unmanageable designs are “mad, bad, and dangerous to know.”

High option potential induces entry => industry structure change

Autos



Computers



Recapping the argument

- u Designs create value

- Value operates like a force in the economy
- Changes the structure of industries

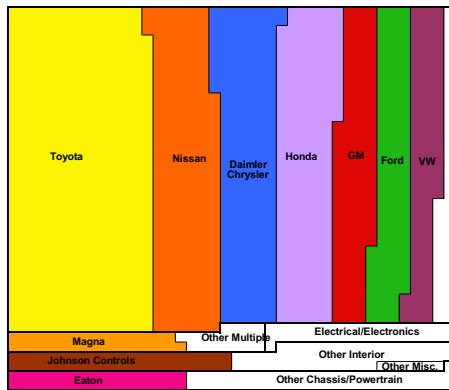
- u Designs have architectures

- *Modularity* and *Option Value* are the key economic properties of an architecture
- *Option Value* =
Number of Customers x **Option Potential** of the Design
- *Modularity* + *High Option Potential* => **Unmanageable**

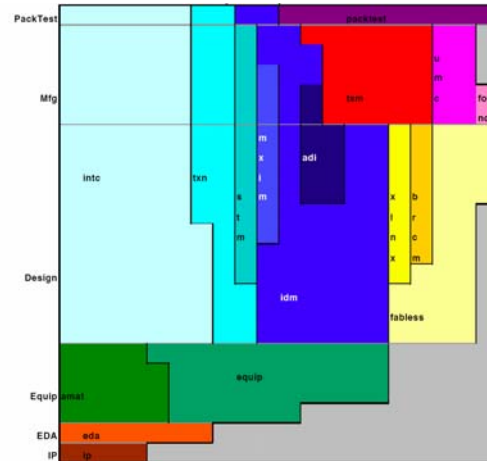
- u *We have not yet asked:*

How do you capture value in a complex designed system?

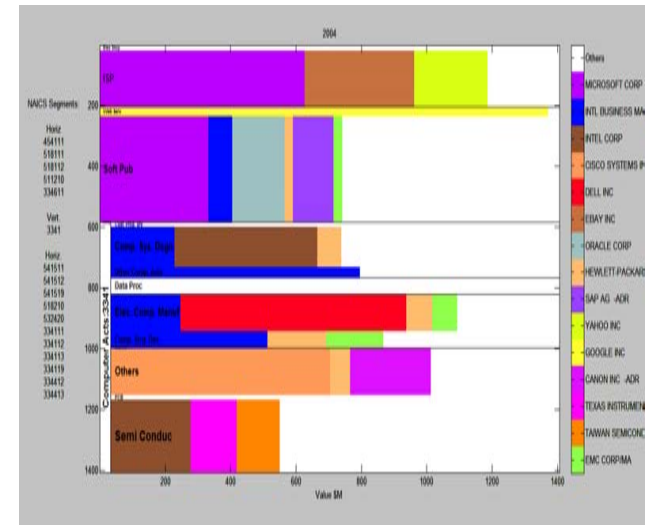
Autos?



Semiconductors?



Computers?

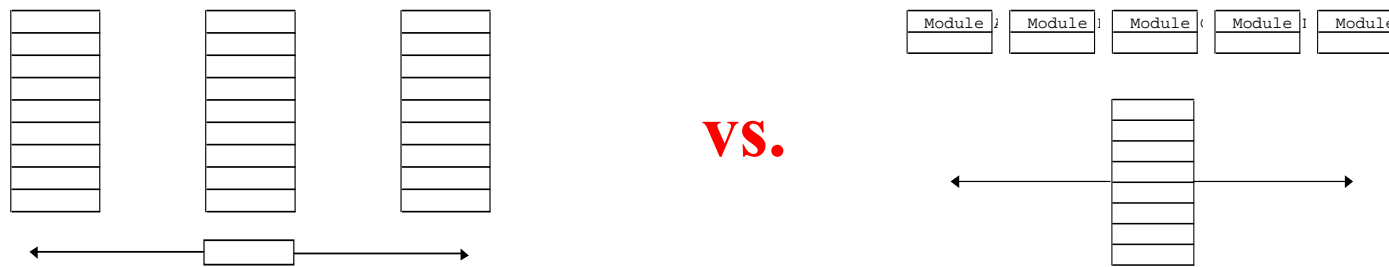


If your world is like...

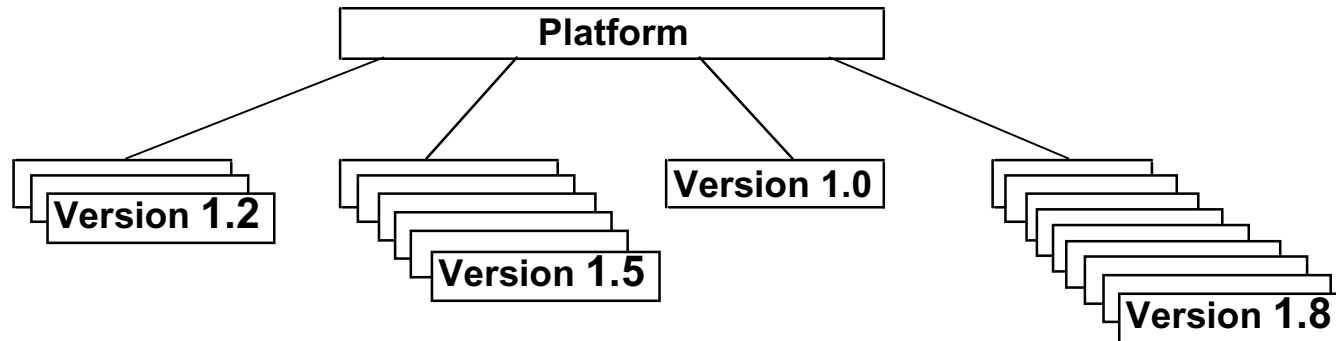
- u **Autos**—product lines will stay within integrated firms (which may have internal platforms and modules)
- u **Semiconductors**—integrated firms and platform-module combinations will coexist
- u **Computers**—platform-module combinations will drive out integrated firms

Which world? Depends on

u Modularity potential of product line



u Option potential of the modules



I can't make it simpler...

Sorry!

And how do you make money?

Ask me after!

Remember

- u *Splitting* a complex system into *platform and modules* decreases the exercise price of “group options”
 - Split and customize
- u After splitting, some of the customized modules may have *high option potential*
 - value in the right tail of probability distribution
- u Modules with high option potential are *unmanageable*

Unmanageable Designs are...

Mad, bad, and dangerous to know

Embrace them, but carefully!

Thank you!

How do you make money in a modular, high-option-potential world?

- u Old paradigm
 - “Plunge in”
 - “Get lucky”
 - “Watch out for Microsoft”
 - “Get bought by HP”

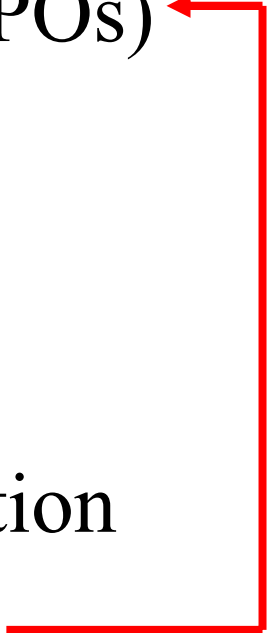
The new paradigm

<u>Company</u>	<u>Rank in 2005</u>
Microsoft	1
Intel	3
Cisco	4
Dell	5
Ebay	6
Oracle	7
SAP	9
Yahoo	10
Google	13
Taiwan Semiconductor	14
EMC	15

Our research strategy—Look for

- u Stable patterns of behavior involving several actors operating within a consistent framework of *ex ante* incentives and *ex post* rewards
- u \implies Equilibria of linked games with self-confirming beliefs (Game theory)

How a “stable pattern” works

- u Anticipation of \$\$\$ (visions of IPOs)
 - u Lots of investment
 - u Lots of design searches
 - u Best designs “win”
 - u Fast design evolution => innovation
 - u Lots of real \$\$\$ (an actual IPO)
- 

“Rational expectations equilibrium”

Not one stable pattern, but several

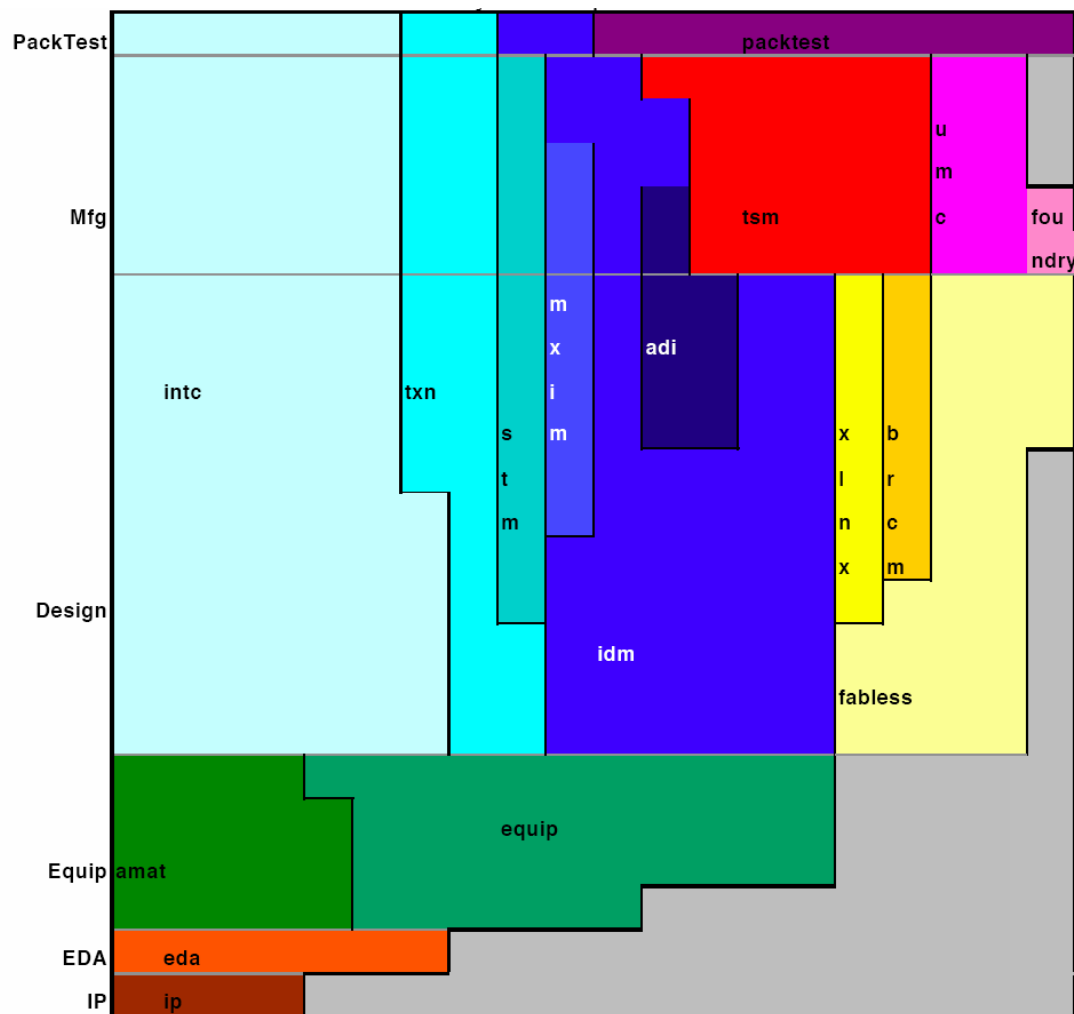
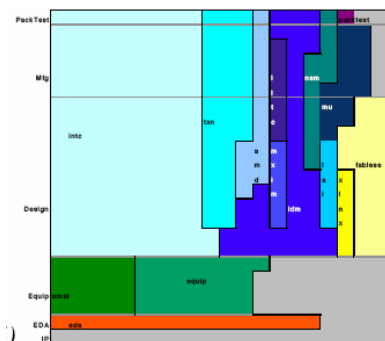
- u Blind competition (everyone)
- u Own the platform, NOT the modules
 - MSFT, Intel, Ebay, TSMC
- u Use M&A to be the “lead firm” in some slice of the stack
 - Cisco, Ebay, Oracle, SAP, Yahoo, Google
- u Use design architecture to reduce your “footprint”
=> high ROIC
 - Dell, Google
- u Use the open source process to replace platforms that you don’t own
 - IBM and Linux

The Evolution of Beliefs

- u “Blind” competitors
 - don’t know others exist
- u “Footprint” competitors
 - Don’t expect to influence others—just compete
- u “Lead firms”
 - Must influence the beliefs of their competitors
 - FUD — “Fear, uncertainty and doubt”
 - Others cannot be blind!

Modular platforms don't always win!

Strojwas (2005)
Semiconductor Industry
Top 10 Firms:
1994 and 2004



Option potential of
semiconductor modules is not
high enough to drive integral
architectures completely out.

These are *technical judgments*
that affect corporate strategy and
industry structure

Huge penalties to getting this wrong!
Xerox, DEC, Sperry, Unisys, Wang,
NCR, Computervision

Three Stable Patterns

- u “Blind” Competition

- PCs in the early 1980s
- Internet in the mid-1990s

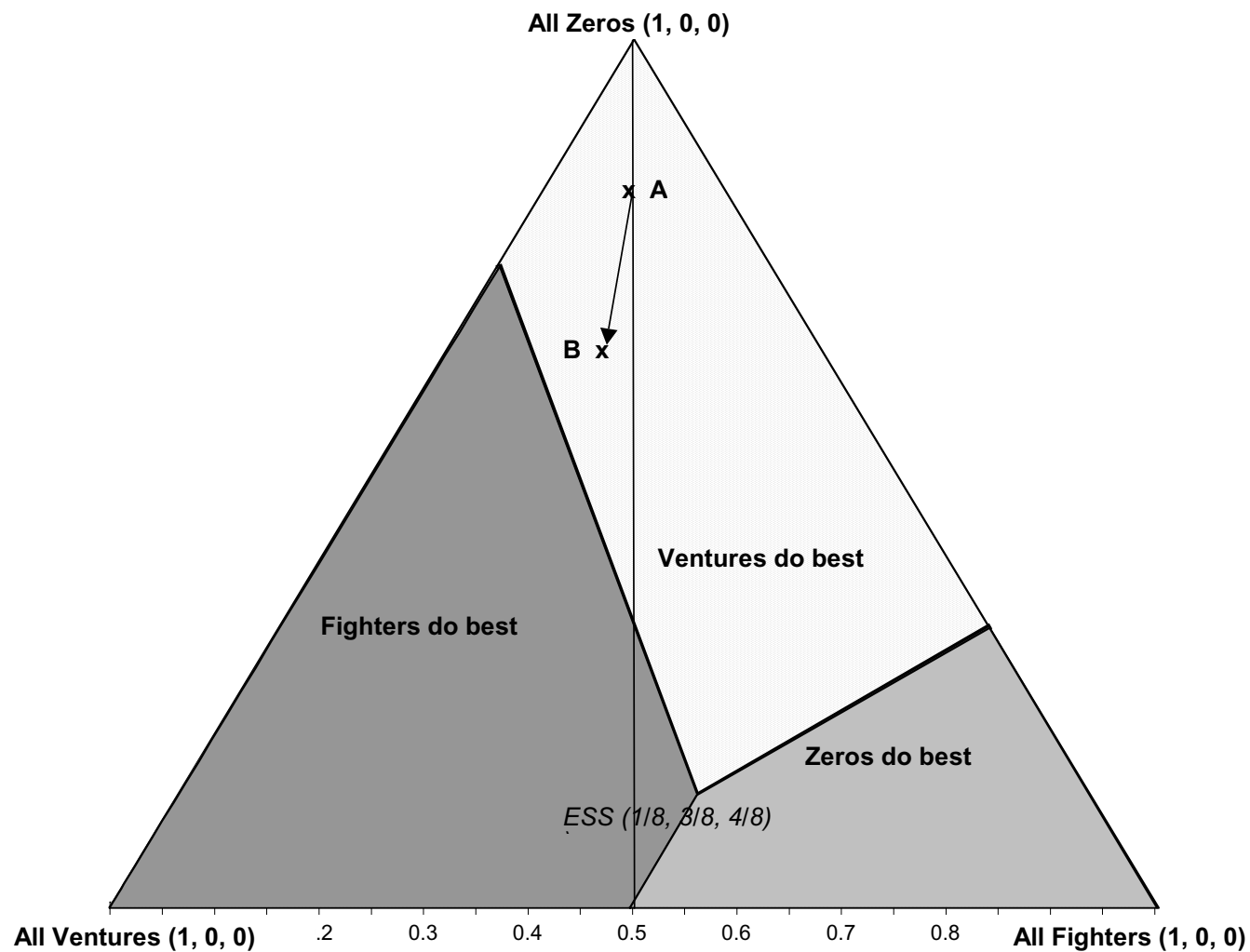
- u “Footprint” Competition

- Sun vs. Apollo
- Dell vs. Compaq (and HP and ...)

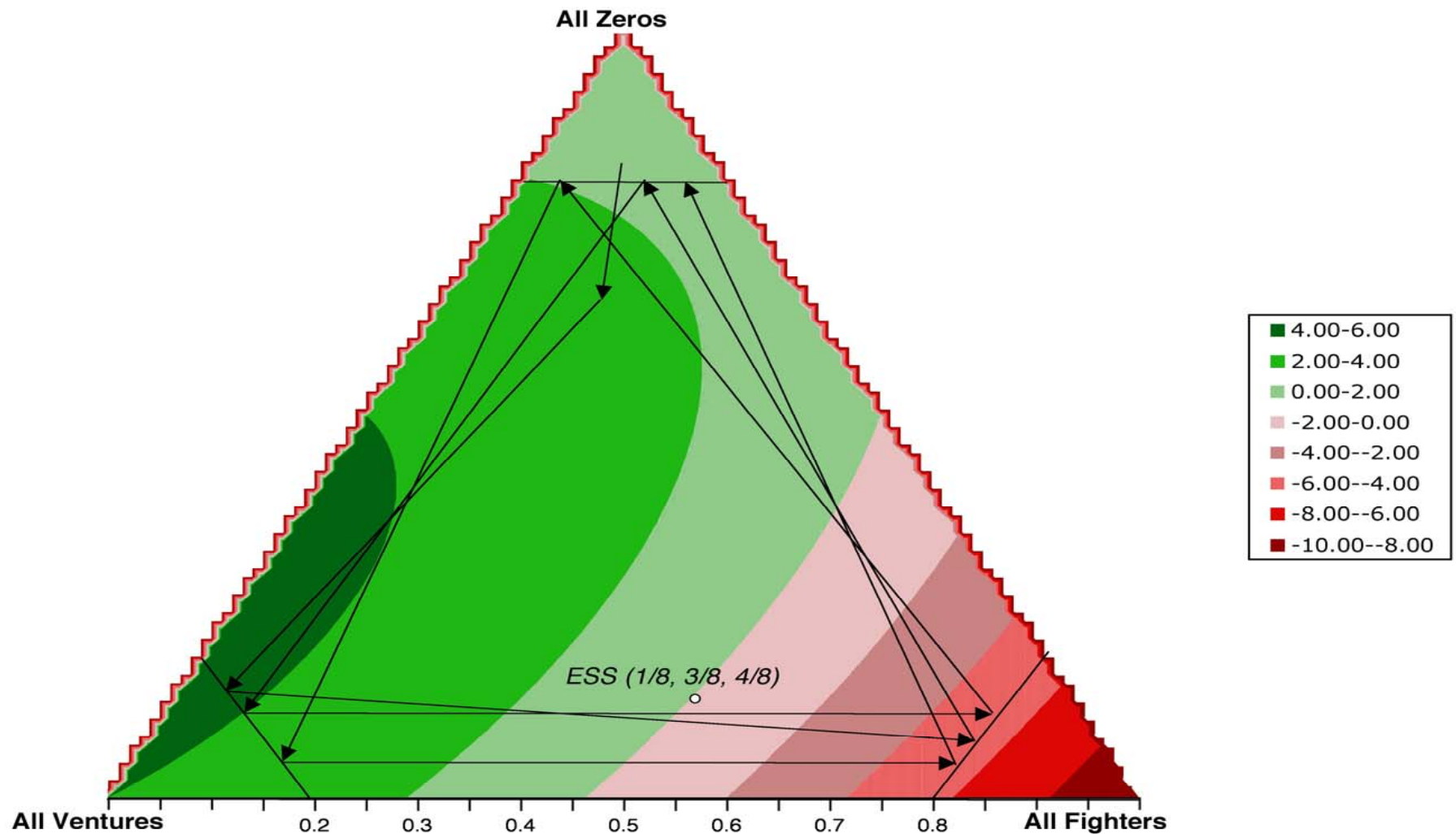
- u “Lead Firm” Competition

- Monopoly—MSFT
- Mergers & Acquisitions—Cisco, Oracle, SAP, ...

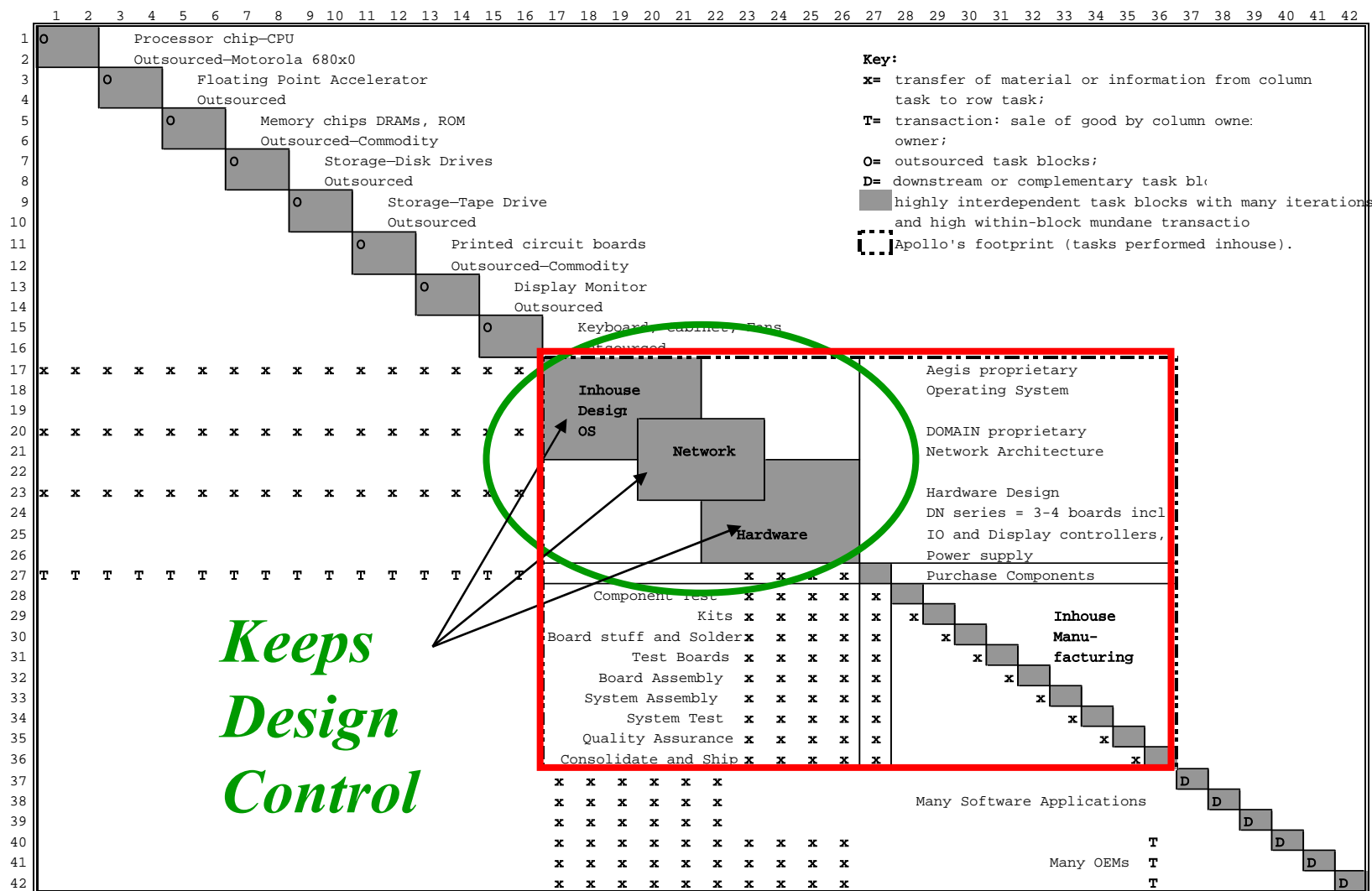
“Blind” Competition



“Blind” Competition

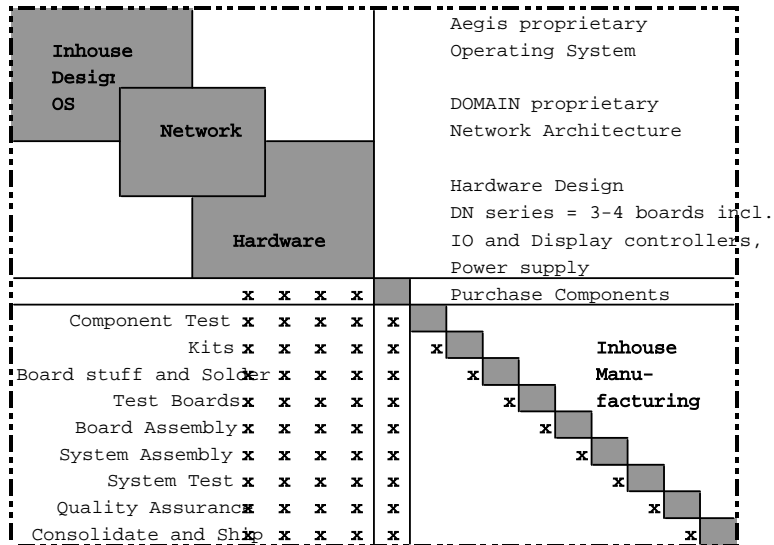


“Footprint” Competition—Apollo



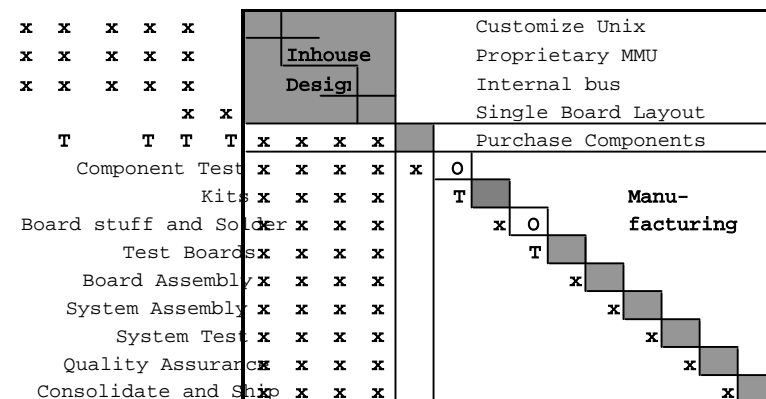
Then Sun came along...

Apollo Computer



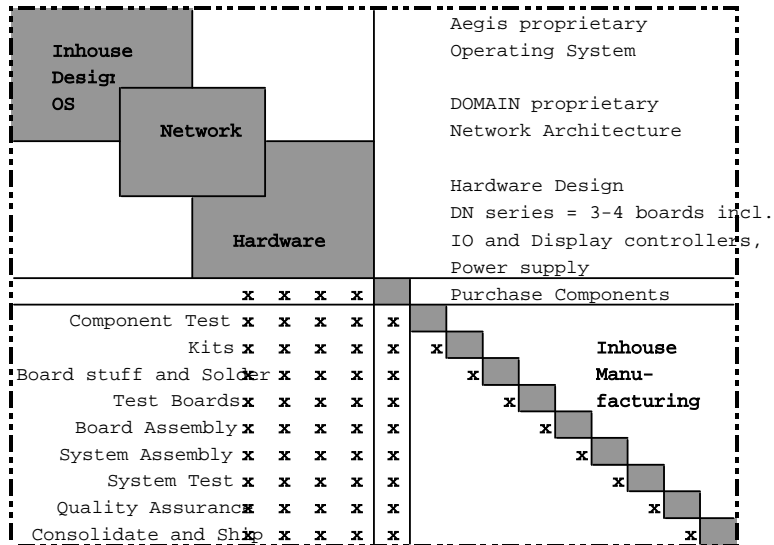
And did even less!

How?



Then Sun came along...

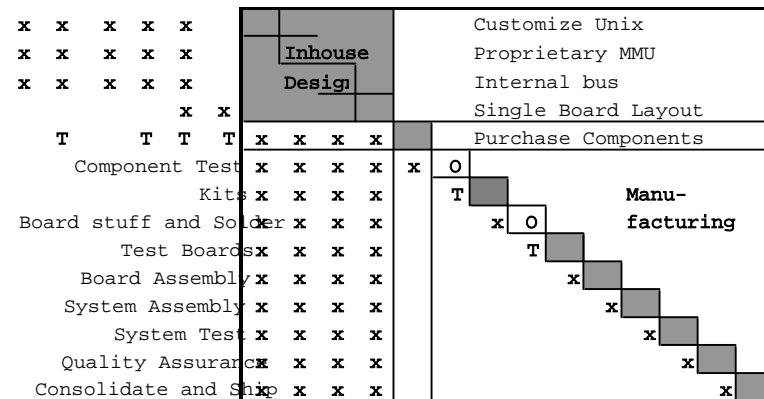
Apollo Computer



Design Architecture for high performance with a small footprint

Public Standards for outsourcing

*And did even less!
How?*



Result: ROIC advantage to Sun

Average over 16 Quarters:

	Apollo Computer	Sun Microsystems	
Invested Capital Ratios (Annualized)			
Net Working Capital/ Sales (%)	29%	15%	Low is good
Ending Net PPE / Sales (%)	24%	13%	Low is good
Invested Capital/Sales (%)	57%	31%	Low is good
Profitability			
Net Income/Sales	0%	6%	High is good
ROIC			
ROIC (excl Cash, Annualized)	2%	20%	High is good

Sun used its ROIC advantage to drive Apollo out of the market

Apollo was acquired by HP in 1989

Compaq vs. Dell

- u Dell did to Compaq what Sun did to Apollo ...
- u Dell created an equally good machine, and
- u Used *design architecture* to reduce its footprint in production, logistics and distribution costs
 - Negative Net Working Capital
 - Direct sales, no dealers
- u Result = Higher ROIC

Higher ROIC always wins!

1997	Compaq Computer	Dell Computer	
Invested Capital Ratios (Annualized)			
Net Working Capital/ Sales (%)	-2%	-5%	Low is good
Ending Net PPE / Sales (%)	8%	3%	Low is good
Invested Capital/Sales (%)	8%	-2%	Low is good
Profitability			
Net Income/Sales	8%	7%	High is good
ROIC			
ROIC (excl Cash, Annualized)	101%	-287%	!!

Dell started cutting prices; Compaq struggled, but in the end had to exit.

Compaq was acquired by HP in 2002

“Lead Firm” Competition

- u Monopolist needs to deter all potential entrants with threats of price war
 - Very fragile equilibrium
 - Potentially expensive to create “enough” FUD
- u M&A Lead Firm does not try to deter all entry in the design space
 - Expects to *buy* most successful entrants *ex post*
 - More robust equilibrium
 - Maybe more advantageous, when you count the cost of FUD

Aspect-Oriented Programming

Radical Research in Modularity

Gregor Kiczales

University of British Columbia
Software Practices Lab



Expressiveness

- The code looks like the design
- “What’s going on” is clear
- The programmer can say what they want to

Programs must be written for people to read, and only incidentally for machines to execute.

[SICP, Abelson, Sussman w/Sussman]



Modularity and Abstraction

- Working definitions:

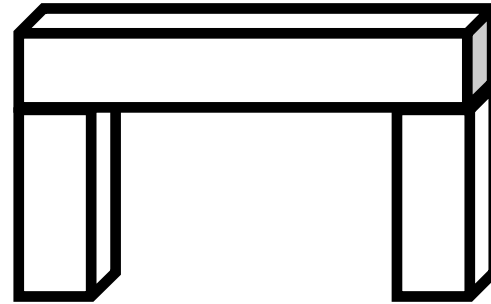
A module is a localized unit of source code with a well-defined interface.

Abstraction means hiding irrelevant details (behind an interface).

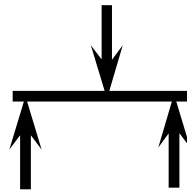


Our Work is Like?

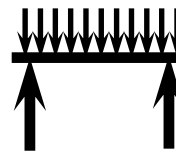
**assembling
wooden
blocks**



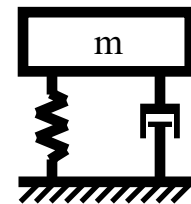
**modeling
and
designing**



simple
statics



more
detailed
statics



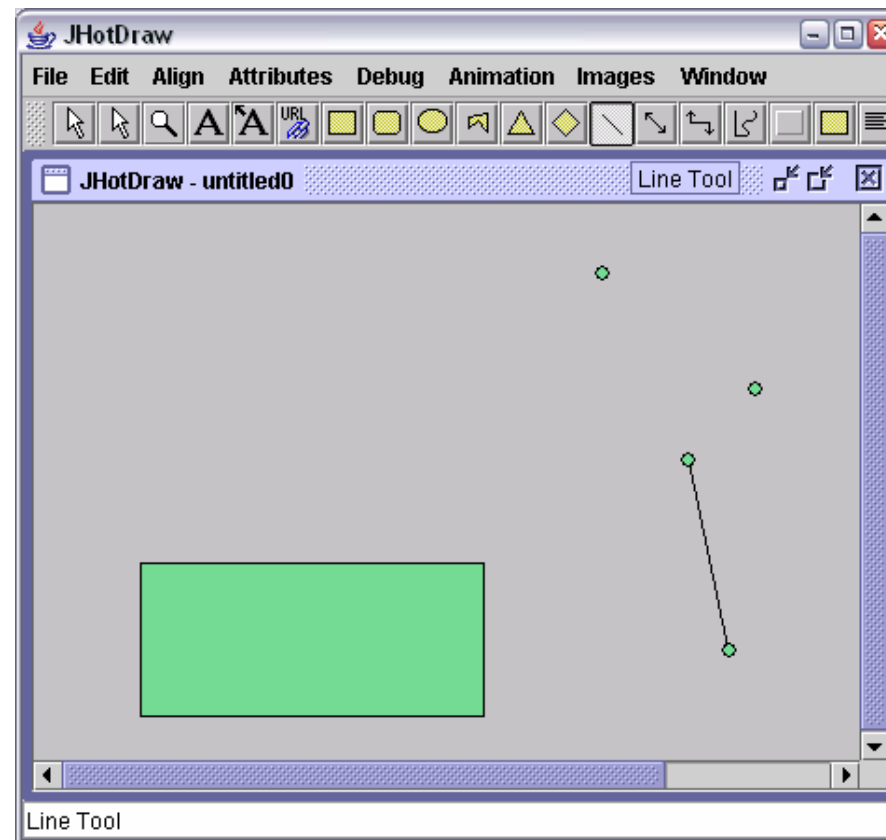
simple
dynamics

Outline

- Introduction
- Intro to AOP
 - OOP/AOP Example
 - Example with AspectJ
 - Other Examples
- Modularity and Abstraction
 - Is AspectJ Code Modular, Abstract
 - Explore Several Critiques
 - Join Point Models
 - Future Possibilities

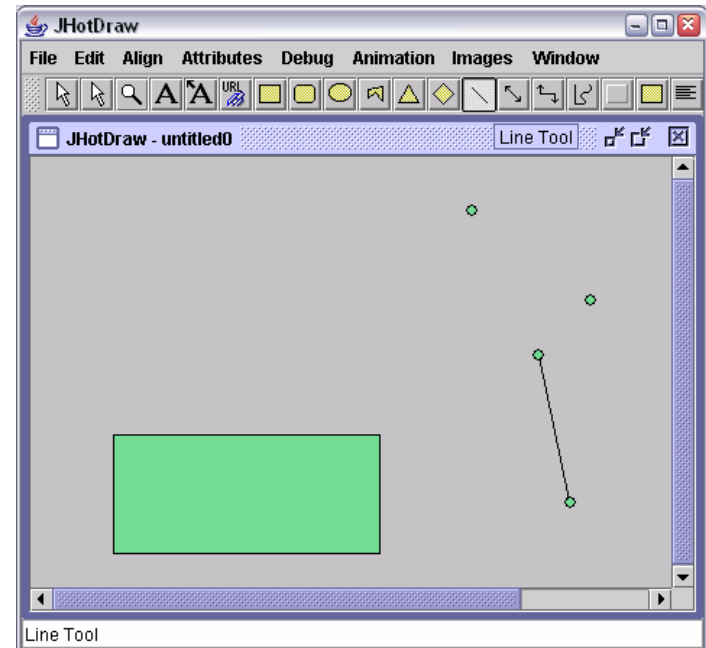


Simple Drawing tool (i.e. JHotDraw)



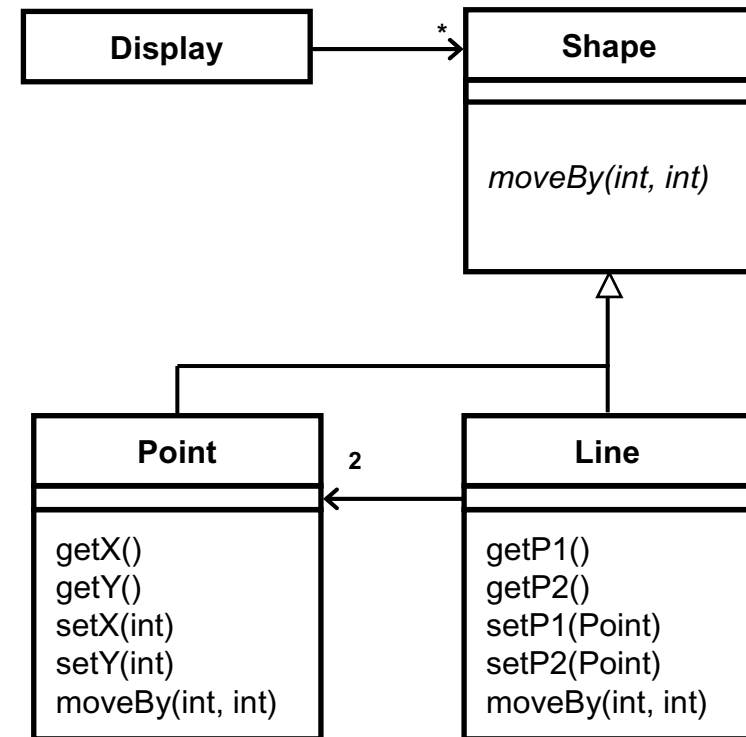
Key Design Elements

- Shapes
 - simple (Point)
 - compound (Line...)
 - display state
 - displayed form
- Display
- ...
- Display update signaling
 - when shapes change
 - update display
 - aka Observer Pattern



Using Objects

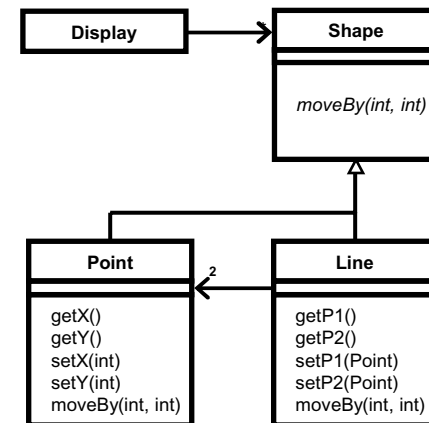
- Shapes
- Display
- Update signaling



Using Objects

- Shapes
- Display
- Update signaling
 - Expressive
 - code looks like the design
 - “what’s going on” is clear
 - Modular
 - localized units
 - well defined interfaces
 - Abstract
 - focus on more or less detail

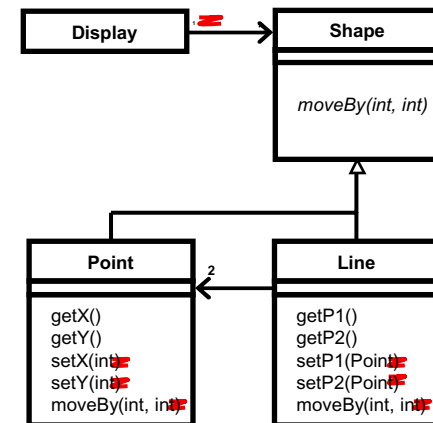
```
class Point extends Shape {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void moveBy(int dx, int dy) {  
        x = x + dx; y = y + dy;  
    }  
  
    void setX(int x) {  
        this.x = x;  
    }  
  
    void setY(int y) {  
        this.y = y;  
    }  
}
```



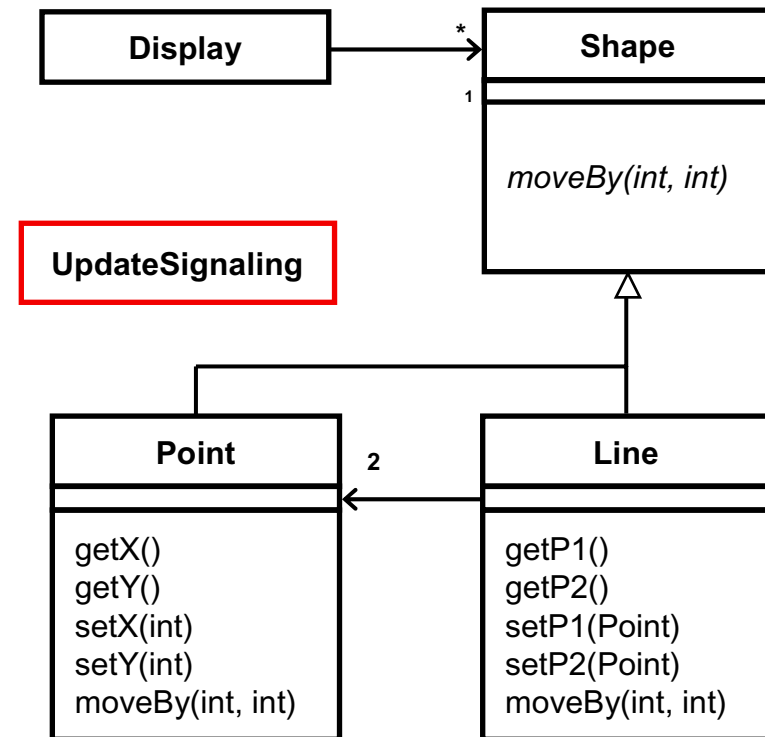
Using Objects

- Shapes
- Display
- Update signaling
 - Expressive
 - Point, Line harder to read
 - structure of signaling
 - not localized, clear, declarative
 - Modular? Abstract?
 - signaling clearly not localized
 - Point, Line polluted
 - revisit this later

```
class Point extends Shape {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void moveBy(int dx, int dy) {  
        x = x + dx; y = y + dy;  
        display.update(this);  
    }  
    void setX(int x) {  
        this.x = x;  
        display.update(this);  
    }  
    void setY(int y) {  
        this.y = y;  
        display.update(this);  
    }  
}
```

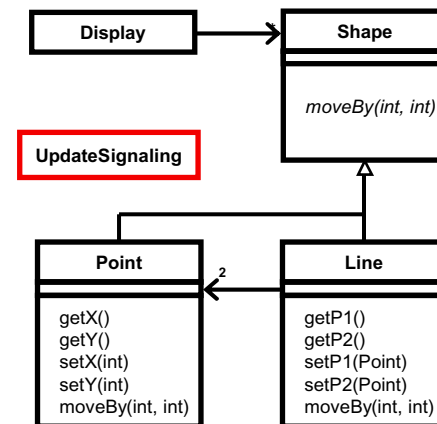


Using Aspect-Oriented Programming



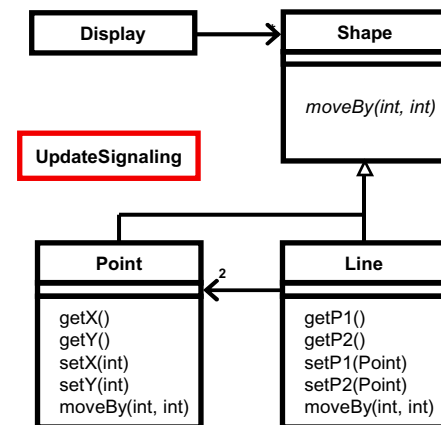
Using Aspect-Oriented Programming

```
aspect UpdateSignaling {  
  
    private Display Shape.display;  
  
    pointcut change():  
        call(void Point.setX(int))  
        || call(void Point.setY(int))  
        || call(void Line.setP1(Point))  
        || call(void Line.setP2(Point))  
        || call(void Shape.moveBy(int, int));  
  
    after(Shape s) returning: change()  
        && target(s) {  
        s.display.update();  
    }  
}
```



Using Aspect-Oriented Programming

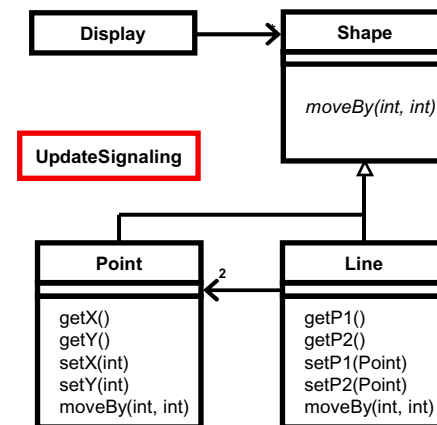
```
aspect UpdateSignaling {  
  
    private Display Shape.display;  
  
    pointcut change():  
        call(void Shape.moveBy(int, int))  
        || call(void Shape+.set*(...));  
  
    after(Shape s) returning: change()  
        && target(s) {  
        s.display.update();  
    }  
}
```



Using Aspect-Oriented Programming

- Shapes
- Display
- Update signaling
 - Expressive
 - “what’s going on” is clear
 - Modular
 - localized units
 - well defined interfaces
 - Abstract
 - focus on more or less detail

```
aspect UpdateSignaling {  
  
    private Display Shape.display;  
  
    pointcut change():  
        call(void Shape.moveBy(int, int))  
        || call(void Shape+.set*(...));  
  
    after(Shape s) returning: change()  
        && target(s) {  
        s.display.update();  
    }  
}
```



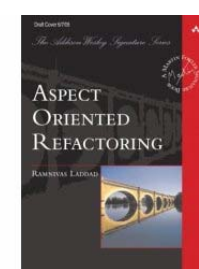
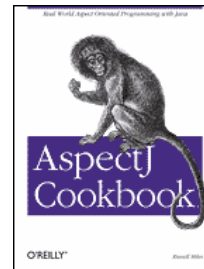
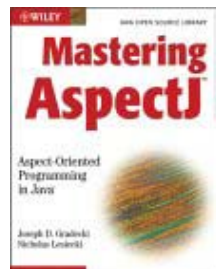
Outline

- Introduction
- Intro to AOP
 - OOP/AOP Example
 - Example with AspectJ
 - Other Examples
- Modularity and Abstraction
 - Is AspectJ Code Modular, Abstract
 - Explore Several Critiques
 - Join Point Models
 - Future Possibilities



AOP w/AspectJ

- AspectJ is
 - seamless extension to Java
 - Eclipse open source project
 - de-facto standard on Java platform
 - model for other AOP tools
 - supported by IBM, Interface 21, BEA



MIT Technology Review 10
Leading technologies 2000

2002 World Technology
Network Finalist

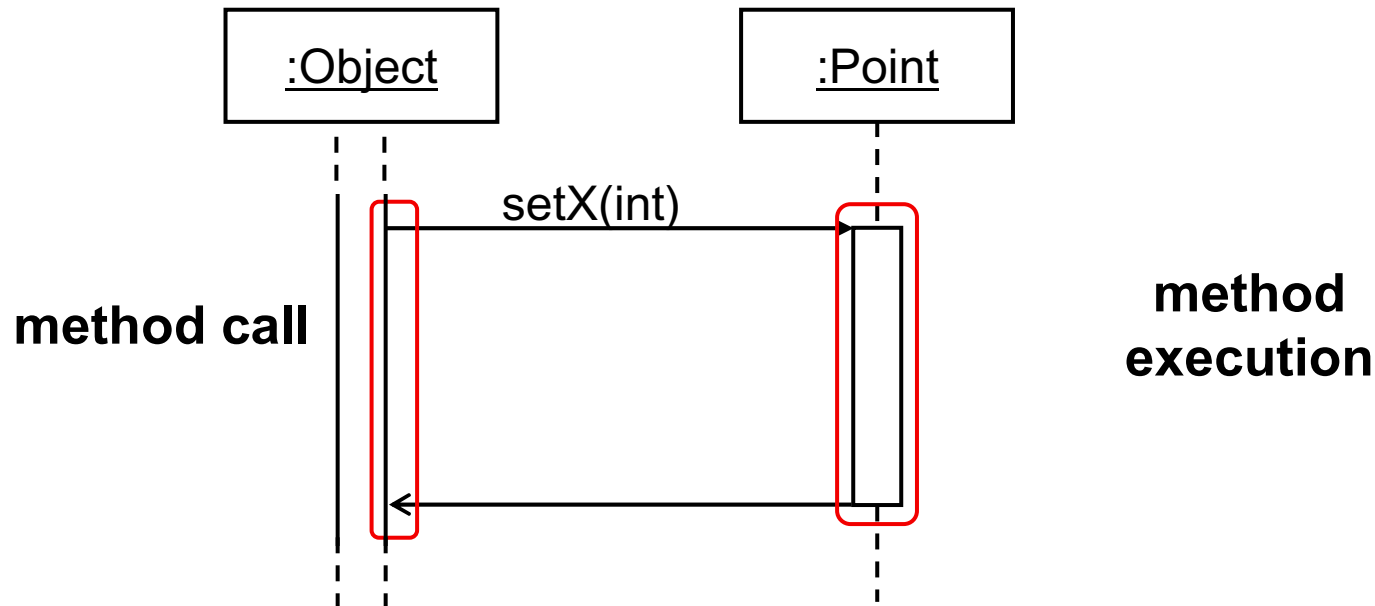


MIT Technology Review
TR100 2004



Dynamic Join Points

points of aspect
correspondence



- 11 kinds of dynamic join point
 - well defined points in flow of execution
 - method, constructor, and advice execution
 - method & constructor call
 - field get & set
 - exception handler execution
 - static, object pre- and object initialization

Pointcuts

means of identifying
dynamic join points

a pointcut is a predicate on dynamic join points that:

- can match or not match any given join point
- says “what is true” when the pointcut matches
- can optionally expose some of the values at that join point

```
execution(void Line.setP1(Point))
```

matches method execution join points with this signature



Pointcut Composition

pointcuts compose like predicates, using **&&**, **||** and **!**

```
execution(void Line.setP1(Point)) ||  
execution(void Line.setP2(Point));
```

whenever a Line executes a
“**void** setP1(Point)” or “**void** setP2(Point)” method



Primitive Pointcuts

<ul style="list-style-type: none">- call, execution, adviceexecution- get, set- handler- initialization, staticinitialization	<p>kinded</p> <p>match one kind of DJP using signature</p>
<ul style="list-style-type: none">- within, withincode- this, target, args- cflow, cflowbelow	<p>non-kinded</p> <p>match all kinds of DJP using variety of properties</p>




User-Defined Pointcuts

user-defined (aka named) pointcuts

- defined with pointcut declaration
- can be used in the same way as primitive pointcuts

name **parameters**



```
pointcut change():  
    execution(void Line.setP1(Point)) ||  
    execution(void Line.setP2(Point));
```

Every powerful language has three mechanisms for [combining simple ideas to form more complex ideas]:

- * primitive expressions, which represent the simplest entities the language is concerned with,
- * means of combination, by which compound elements are built from simpler ones, and
- * means of abstraction, by which compound elements can be named and manipulated as units.

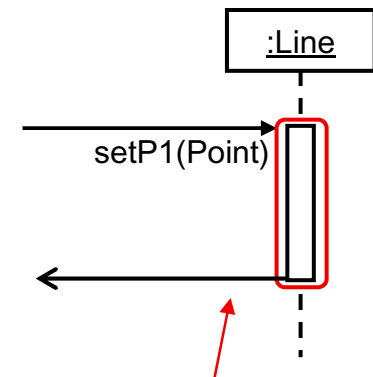
[SICP, Abelson, Sussman w/ Sussman]

After Advice

means of semantic effect
at dynamic join points

```
pointcut change():  
    execution(void Line.setP1(Point)) ||  
    execution(void Line.setP2(Point));
```

```
after() returning: change()  
{  
    <code here runs after each change>  
}
```



after advice
runs on the
way back out

A Simple Aspect

UpdateSignaling v1

```
aspect UpdateSignaling {  
  
    pointcut change():  
        execution(void Line.setP1(Point)) ||  
        execution(void Line.setP2(Point));  
  
    after() returning: change()  
    {  
        Display.update();  
    }  
}
```

box means complete running code

How to Read This Code

UpdateSignaling v1

Here is the UpdateSignaling aspect of the system.

```
aspect UpdateSignaling {  
  
    pointcut change():  
        execution(void Line.setP1(Point)) ||  
        execution(void Line.setP2(Point));  
  
    after() returning: change()  
    {  
        Display.update();  
    }  
}
```

Some points in the system's execution are a "change".

Specifically, these method executions.

After returning from change points-
update the display.



Without AspectJ

UpdateSignaling v1

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
        Display.update();  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
        Display.update();  
    }  
}
```

what you would write if you didn't
have AspectJ;
NOT what AspectJ produces
OR meaning of AspectJ code

- what you would expect
 - update calls are scattered and tangled
 - “what is going on” is less explicit

How Do You Think About Objects?

- Objects
 - Define their own behavior
 - Have fields and methods
 - Clear interface
- A datastructure w/
 - Vector of fields
 - Pointer to method table
- Dispatch code
 - Method call table entry
- Macrology to
 - Make fields look like vars
 - Method calls look nice



Abstraction

- Objects
 - Define their own behavior
 - Have fields and methods
 - Clear interface

Helps to

- do OO *design*
- scale use of objects to large systems

- A datastructure w/
 - Vector of fields
 - Pointer to method table
- Dispatch code
 - Method call table entry
- Macrology to
 - Make fields look like vars
 - Method calls look nice

Helps understand

- *one way* to implement OOP
- potential performance costs
- language semantics issues



Abstraction

Helps to

- do *AO design*
- scale use of aspects to large systems

Helps understand

- *one way* to implement AOP
- potential performance costs
- language semantics issues

- Aspects
 - Define their own behavior
 - Have pointcuts, advice ...
 - Clear interface
- A datastructure w/
 - Vector of fields
 - Pointer to method table
- Code transformations
 - Find join point shadows
 - Insert interceptor calls



Abstraction

- **Objects**
 - Define their own behavior
 - Have fields and methods
 - Clear interface
- **Aspects**
 - Define their own behavior
 - Have pointcuts, advice ...
 - Clear interface
- A datastructure w/
 - Vector of fields
 - Pointer to method table
- A datastructure w/
 - Vector of fields
 - Pointer to method table
- Dispatch code
 - Method call table entry
- Code transformations
 - Find join point shadows
 - Insert interceptor calls
- Macrology to
 - Make fields look like vars
 - Method calls look nice



A Multi-Class Aspect

UpdateSignaling v2

```
aspect UpdateSignaling {  
  
    pointcut change():  
        execution(void Shape.moveBy(int, int)) ||  
        execution(void Line.setP1(Point)) ||  
        execution(void Line.setP2(Point)) ||  
        execution(void Point.setX(int)) ||  
        execution(void Point.setY(int));  
  
    after() returning: change() {  
        Display.update();  
    }  
}
```



Using Naming Convention

UpdateSignaling v2b

```
aspect UpdateSignaling {  
  
    pointcut change():  
        execution(void Shape.moveBy(int, int)) ||  
        execution(void Shape+.set*(*) );  
  
    after() returning: change() {  
        Display.update();  
    }  
}
```



Using Attributes

UpdateSignaling v2c

```
aspect UpdateSignaling {  
  
    pointcut change():  
        execution(@Change * *(..));  
  
    after() returning: change() {  
        Display.update();  
    }  
}
```

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    @Change  
    void moveBy(int dx, int dy) {  
        p1.moveBy(dx, dy);  
        p2.moveBy(dx, dy);  
    }  
  
    @Change  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
  
    @Change  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}
```



Values at Join Points

UpdateSignaling v3

- pointcut can explicitly expose certain values
- advice can use explicitly exposed values

```
aspect UpdateSignaling {  
  
    pointcut change(Shape shape):  
        this(shape) &&  
        (execution(void Shape.moveBy(int, int)) ||  
         execution(void Shape+.set*(*)) );  
  
    after(Shape s) returning: change(s) {  
        Display.update(s);  
    }  
}
```



Crosscutting Structure

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void moveBy(int dx, int dy) {  
        p1.moveBy(dx, dy);  
        p2.moveBy(dx, dy);  
    }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}
```

```
aspect UpdateSignaling {  
  
    pointcut change(Shape shape):  
        this(shape) &&  
        (execution(void Shape.moveBy(int, int) ||  
         execution(void Shape+.set*(*))));  
  
    after(Shape s) returning: change(s) {  
        Display.update(s);  
    }  
}
```

```
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void moveBy(int dx, int dy) {  
        x = x + dx; y = y + dy;  
    }  
    void setX(int x) {  
        this.x = x;  
    }  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

- Aspect and classes crosscut
- Pointcut cuts interface
 - through Point and Line
 - advice programs against interface
 - interface structure is declarative

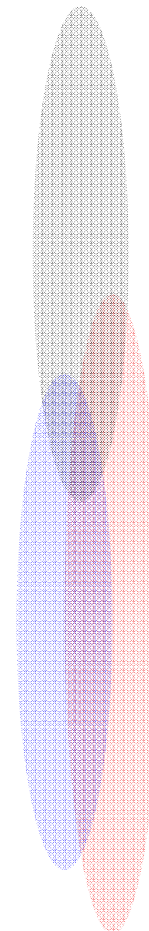
Crosscutting

c1 and c2 crosscut wrt a common representation iff projections overlap, but do not contain [Masuhara, ECOOP03]

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
}
```

```
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void setX(int x) {  
        this.x = x;  
    }  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

```
aspect UpdateSignaling {  
  
    pointcut change(Shape shape):  
        this(shape) &&  
        (execution(void Shape.moveBy(int, int)) ||  
         execution(void Shape+.set*(*) ));  
  
    after(Shape s) returning: change(s) {  
        Display.update(s);  
    }  
}
```



```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
        Display.update();  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
        Display.update();  
    }  
}
```

```
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void setX(int x) {  
        this.x = x;  
        Display.update();  
    }  
    void setY(int y) {  
        this.y = y;  
        Display.update();  
    }  
}
```



Scattering and Tangling

```
class Shape {
    private Display display;

    abstract void moveBy(int, int);
}

class Line extends Shape {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void moveBy(int dx, int dy) {
        p1.moveBy(dx, dy);
        p2.moveBy(dx, dy);
        display.update(this);
    }

    void setP1(Point p1) {
        this.p1 = p1;
        display.update(this);
    }
    void setP2(Point p2) {
        this.p2 = p2;
        display.update(this);
    }
}

class Point extends Shape {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

    void moveBy(int dx, int dy) {
        x = x + dx;
        y = y + dy;
        display.update(this);
    }

    void setX(int x) {
        this.x = x;
        display.update(this);
    }
    void setY(int y) {
        this.y = y;
        display.update(this);
    }
}
```

Observer pattern is

scattered –
spread around

tangled –
mixed in with other concerns



IDE support

- AJDT (AspectJ Development Tool)
- An Eclipse Project
- Goal is JDT-quality AspectJ support
 - highlighting, completion, wizards...
 - structure browser
 - immediate
 - outline
 - overview



Only Top-Level Changes

UpdateSignaling v4

```
aspect UpdateSignaling {  
  
    pointcut change(Shape shape):  
        this(shape) &&  
        (execution(void Shape.moveBy(int, int)) ||  
         execution(void Shape+.set*(*)) );  
  
    pointcut topLevelChange(Shape shape):  
        change(shape) && !cflowbelow(change(Shape));  
  
    after(Shape s) returning: topLevelChange(s) {  
        Display.update(s);  
    }  
}
```

Compositional Crosscutting

```
class Line {
    private Point p1, p2;

    Point getP1() { return p1; }
    Point getP2() { return p2; }

    void moveBy(int dx, int dy) {
        p1.moveBy(dx, dy);
        p2.moveBy(dx, dy);
    }

    void setP1(Point p1) {
        this.p1 = p1;
    }
    void setP2(Point p2) {
        this.p2 = p2;
    }
}
```

```
class Point {
    private int x = 0, y = 0;

    int getX() { return x; }
    int getY() { return y; }

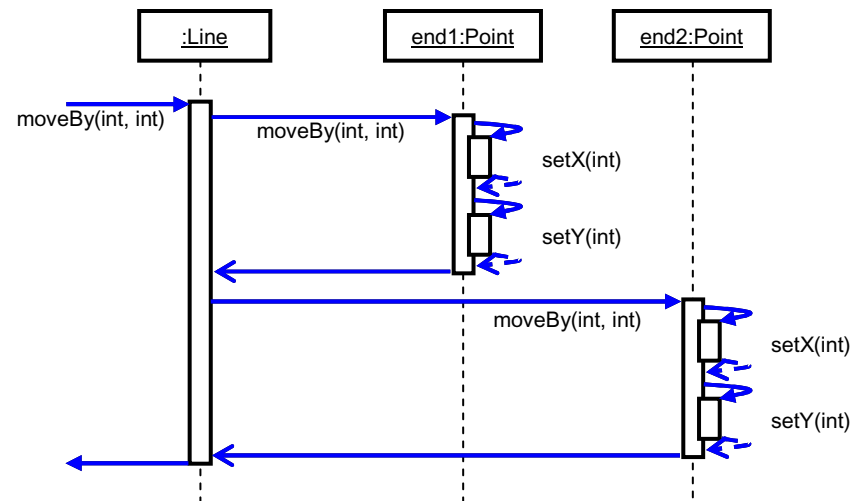
    void moveBy(int dx, int dy) {
        x = x + dx; y = y + dy;
    }
    void setX(int x) {
        this.x = x;
    }
    void setY(int y) {
        this.y = y;
    }
}
```

```
aspect UpdateSignaling {

    pointcut change(Shape shape):
        this(shape) &&
        (execution(void Shape.moveBy(int, int)) ||
         execution(void Shape+.set*(*)));

    pointcut topLevelChange(Shape shape):
        change(shape) && !cflowbelow(change(Shape));

    after(Shape s) returning: topLevelChange(s) {
        Display.update(s);
    }
}
```



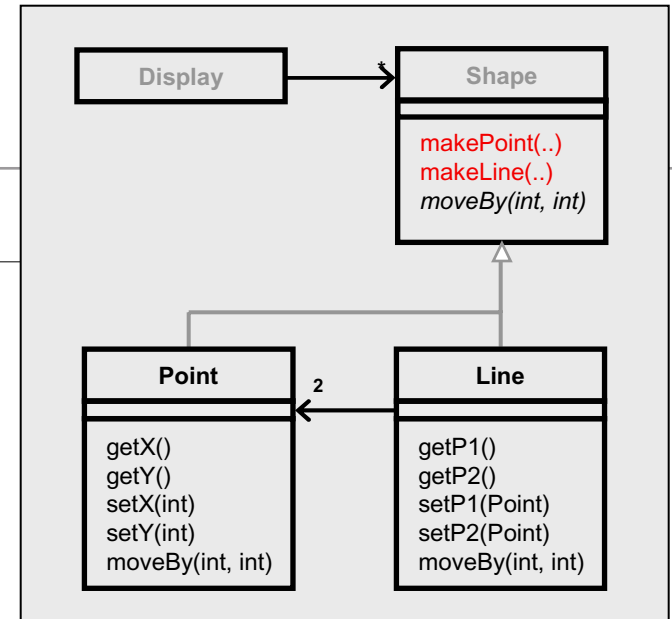
Outline

- Introduction
- Intro to AOP
 - OOP/AOP Example
 - Example with AspectJ
 - Other Examples
- Modularity and Abstraction
 - Is AspectJ Code Modular, Abstract
 - Explore Several Critiques
 - Join Point Models
 - Future Possibilities



Design Invariants

```
aspect FactoryEnforcement {  
  
    pointcut newShape():  
        call(Shape+.new(..));  
  
    pointcut inFactory():  
        withincode(Shape+ Shape.make*(..));  
  
    pointcut illegalNewShape():  
        newShape() && !inFactory();  
  
    before(): illegalNewShape() {  
        throw new RuntimeException("Must call factory method...");  
    }  
}
```



Design Invariants

```
aspect FactoryEnforcement {
```

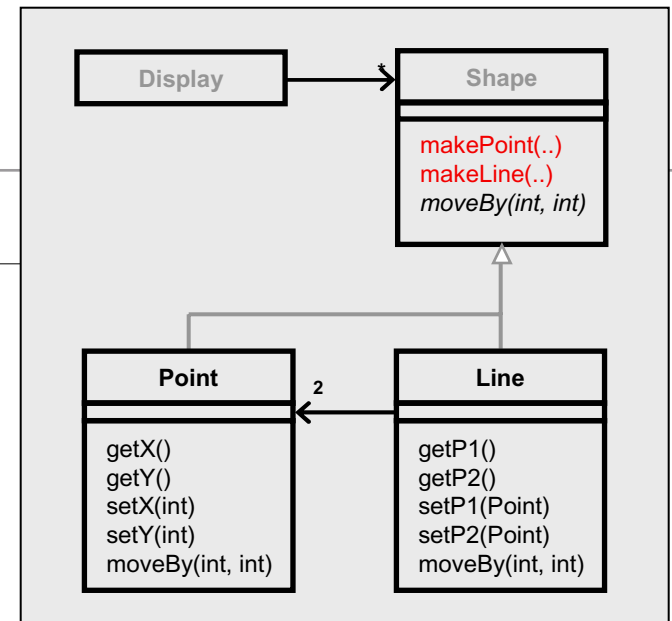
```
    pointcut newShape():  
        call(Shape+.new(..));
```

```
    pointcut inFactory():  
        withincode(Shape+ Shape.make*(..));
```

```
    pointcut illegalNewShape():  
        newShape() && !inFactory();
```

```
    declare error: illegalNewShape():  
        "Must call factory method to create figure elements.";
```

```
}
```



(Simple) Authentication State FSM

```
public aspect AccessibilityFSM {  
  
    private enum State { INIT, AUTHENTICATED, REJECTED };  
  
    private State curr = State.INIT; // global state  
  
    pointcut authenticate(): ...;  
  
    pointcut access(): ...;  
  
    after() returning: authenticate() { curr = State.AUTHENTICATED; }  
    after() throwing:  authenticate() { curr = State.REJECTED; }  
  
    before(): access() {  
        if( curr != State.AUTHENTICATED )  
            throw new AccessException();  
    }  
}
```

FFDC [Colyer et. al. AOSD 2004]

```
public aspect FFDC {  
  
    private Log log = <appropriate global log>;  
  
    after() throwing (Error e):  
        execution(* com.ibm..*(...)) {  
        log.log(e);  
    }  
}
```

- Logs every error as soon as its thrown
- Consistent policy makes logs meaningful
- Real FFDC implementations are more complex

From a Spacewar Game

```
class Ship {
    ...
    public void fire() { ... }
    public void rotate(int direction) { ... }
    public void fire() { ... }
    ...
    static aspect EnsureShipIsAlive {

        pointcut helmCommand(Ship ship):
            this(ship) &&
            ( execution(void Ship.rotate(int)) ||
              execution(void Ship.thrust(boolean)) ||
              execution(void Ship.fire()) );

        void around(Ship ship): helmCommand(ship) {
            if ( ship.isAlive() ) {
                proceed(ship);
            }
        }
    }
}
```



dfLOW pointcut [Masuhara et. al.]

```
aspect Sanitizing {  
    String around (String s):  
        call(void print(String))  
        && args(s)  
        && dfLOW[s, userInput]  
            (call(String Request.get()) && returns(userInput))  
  
    proceed(quote(s));  
}  
}
```

- Quotes strings passed to out.print
- when generating responses
- and the string is based on user input

One Display per Shape

UpdateSignaling v5

```
aspect UpdateSignaling {  
  
    private Display Shape.display;  
  
    static void setDisplay(Shape s, Display d) {  
        s.display = d;  
    }  
  
    pointcut change(Shape  
        this(shape) &&  
        (execution(void Sh  
            execution(void Sh  
  
    after(Shape s) return  
        s.display.update(s  
  
}
```

private with respect to aspect

- inter-type declarations
- aka open classes [Cannon 78]
- declares members of other types
 - fields, methods
- display field
 - is in objects of type Shape
 - but belongs to UpdateSignaling aspect



From a Compiler

```
/**
 * Implements the crosscutting relationships concerning the different kinds of
 * labels that different kinds of statements (and one expr) have. The declare
 * parents block can be read as table of what ASTs have what labels.
 */
aspect HasLabel {

    private interface Label      {} //enclosing loop's label
    private interface TopLabel   {}
    private interface DoneLabel  {}
    private interface IncrLabel  {}
    private interface TrueLabel  {}
    private interface FalseLabel {}
```

declare parents:

WhileStat	implements	TopLabel,	DoneLabel;
ForStat	implements	TopLabel, IncrLabel,	DoneLabel;
BreakStat	implements	Label	;
ContinueStat	implements	Label	;
BinaryExpr	implements	TrueLabel,	DoneLabel;
IfStat	implements	TrueLabel, FalseLabel,	DoneLabel;

```
...
}
```



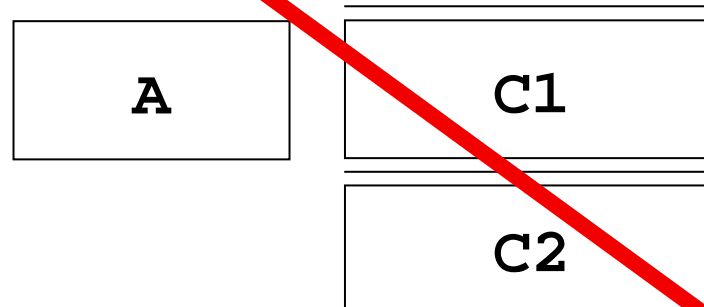
Outline

- Introduction
- Intro to AOP
 - OOP/AOP Example
 - Example with AspectJ
 - Other Examples
- Modularity and Abstraction
 - Is AspectJ Code Modular, Abstract?
 - Explore Several Answers
 - Join Point Models
 - Future Possibilities



~~"AOP is Anti-Modular"~~

- “it changes the behavior of my code”
 - A can affect behavior visible at interface to C1



- But
 - C2 can do that also
 - That's the nature of modularity:
 - A module implements its behavior in terms of other well-defined behaviors

Real Programmers Use VI

- In non-AOP programmers can easily chase module references
 - to know what has to be consulted
 - to determine complete behavior of C1
 - we don't want to have to use tool support
- But
 - include files are 'easy' to chase down?
 - write enterprise code w/o tools?



Put the Genie Back in the Bottle

- Propose 'improvement' to AOP so that
 - methods, classes, files etc.
that want advice, say so explicitly
- But
 - this just reduces AOP back to procedure calls
 - whole point was to go beyond that

Without AspectJ

UpdateSignaling v5

```
class Shape {  
    private Display display;  
    abstract void moveBy(int, int);  
}
```

```
class Line extends Shape {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void moveBy(int dx, int dy) {  
        p1.moveBy(dx, dy);  
        p2.moveBy(dx, dy);  
        display.update(this);  
    }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
        display.update(this);  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
        display.update(this);  
    }  
}
```

```
class Point extends Shape {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void moveBy(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
        display.update(this);  
    }  
  
    void setX(int x) {  
        this.x = x;  
        display.update(this);  
    }  
    void setY(int y) {  
        this.y = y;  
        display.update(this);  
    }  
}
```

- Replaying the same evolution
- Through 4 versions
- In plain OO (Java)

“display updating” is not modular

- evolution is cumbersome
- changes are scattered
- have to track & change all callers
- it is harder to think about

With AspectJ

UpdateSignaling v5

```
class Shape {  
  
    abstract void moveBy(int, int);  
}
```

```
class Line extends Shape {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void moveBy(int dx, int dy) {  
        p1.moveBy(dx, dy);  
        p2.moveBy(dx, dy);  
    }  
  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}
```

```
class Point extends Shape {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void moveBy(int dx, int dy) {  
        x = x + dx;  
        y = y + dy;  
    }  
  
    void setX(int x) {  
        this.x = x;  
    }  
  
    void setY(int y) {  
        this.y = y;  
    }  
}
```

```
aspect UpdateSignaling {  
  
    private Display Shape.display;  
  
    static void setDisplay(Shape s, Display d) {  
        s.display = d;  
    }  
  
    pointcut change(Shape shape):  
        this(shape) &&  
        (execution(void Shape.moveBy(int, int)) ||  
         execution(void Shape+.set*(*)) );  
  
    after(Shape s) returning: change(s) {  
        shape.display.update(s);  
    }  
}
```

UpdateSignaling is modular

- all changes in single aspect
- evolution is modular
- it is easier to think about

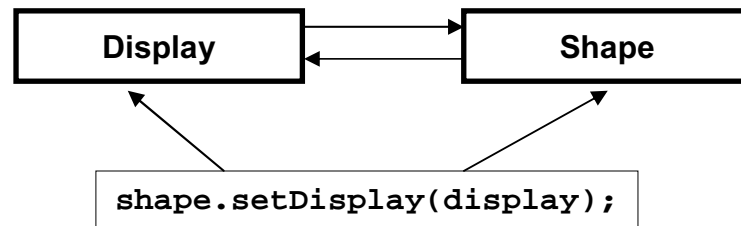


Selling Different Service Aspects

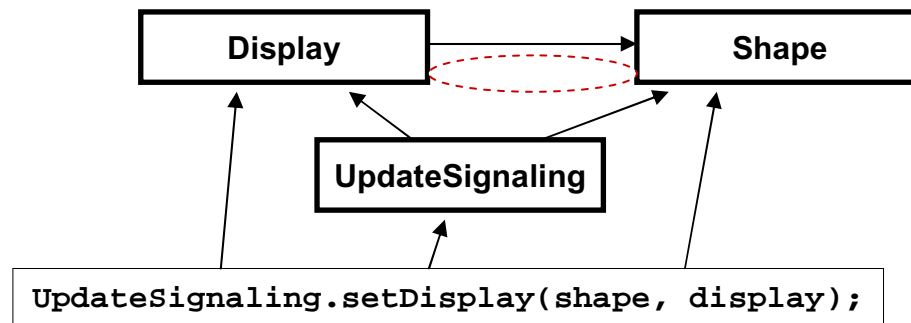
- During internal exploration of AspectJ @ IBM
 - key point
 - “So we could sell different logging policies?”
- Product-line potential of
 - FFDC and related serviceability aspects



Comparing *refers* to relations



Plain Java



w/ AspectJ 1

What's Going On

- The nays make a subtle assumption:
*A module is a **statically** localized unit of source code with a well-defined **static** interface.*

*Abstraction means hiding **permanently** irrelevant details behind an interface.*
- But...
 - crosscutting concerns (ccc)
 - from perspective of ccc, system modularity is different
 - it is decomposed along entirely different lines
 - for ccc to be modular, modularization can't be static



[Kiczales Mezini, ICSE 05]

- Starts w/ AspectJ style AOP
- Presents more flexible definition of module
 - modules are statically localized
 - but interfaces are somewhat more dynamic
 - constructed based on complete system configuration
 - complete module interface not known until system configuration is known!
- Shows that modular reasoning
 - is possible
 - works better than without AOP if there are crosscutting concerns



Crosscutting Concerns are Real

- Crosscutting concerns are a fact of life
- They are a 'root problem' for modularity
- Even simple UpdateSignalling
 - cannot be implemented modularly w/o AOP

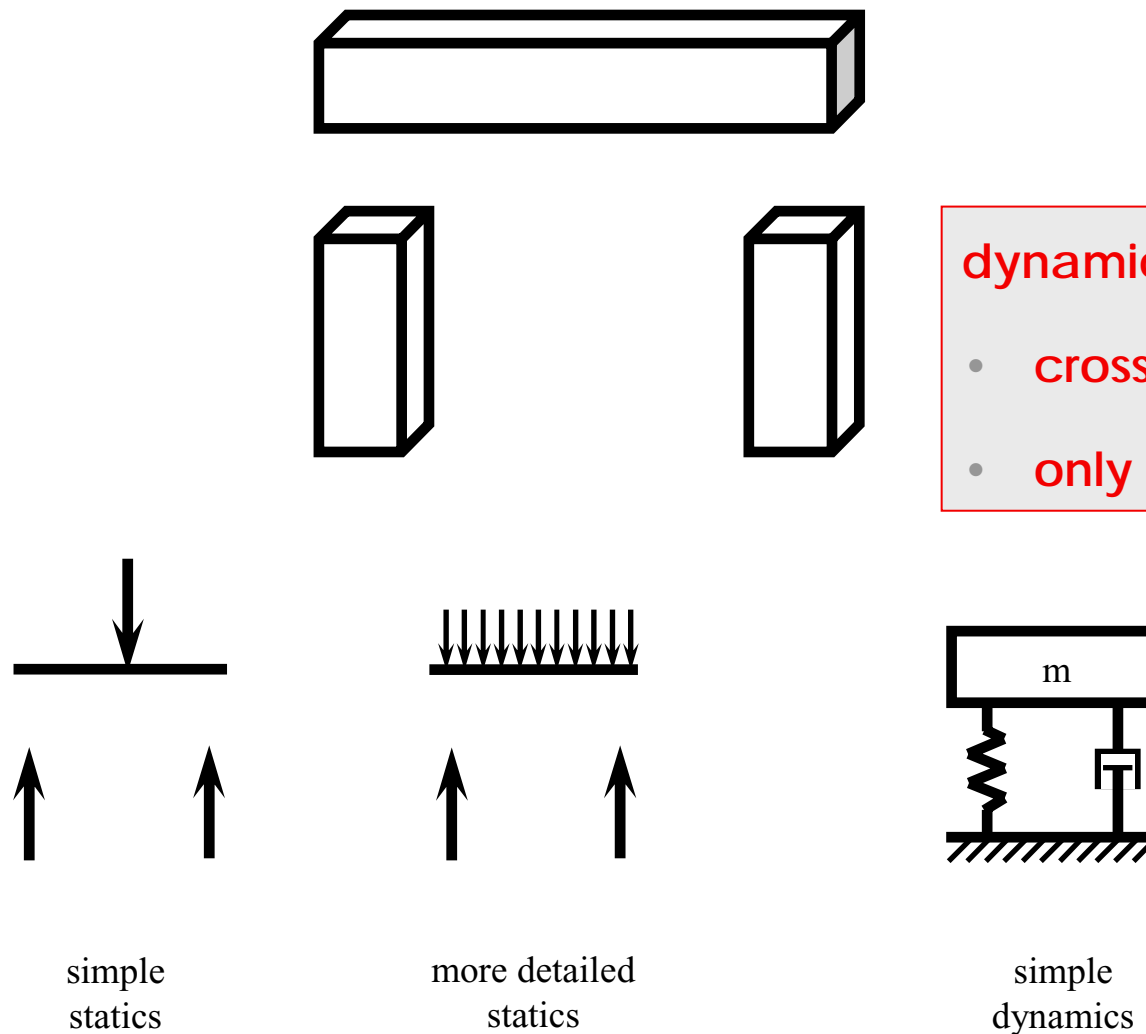
SSP modularity not enough

A module is a statically localized unit of source code with a well-defined static interface.

Abstraction means hiding permanently irrelevant details behind an interface.



Crosscutting In Other Domains



[Whitney – Physical Limits to Modularity]

- Compares VLSI and complex electro-mechanical systems
- Claims that modular design difficult in high-power systems
- In our terms
 - the high-power engenders many crosscutting concerns

“in CEMO thousands of distinct parts must be designed to create a product with a similar total number of parts, and many must be verified first individually and again in assemblies by simulation and/or prototype testing; a modular approach works sometimes, but not in systems subjected to severe weight, space, or energy constraints; in constrained systems, parts must be designed to share functions or do multiple jobs; design and performance of these parts are therefore highly coupled”



[Smith, On the Origin of Objects¹]

- How is it that we can see the world in different ways?
- Registration is
 - process of ‘parsing’ objects out of fog of undifferentiated stuff
 - constantly registering and re-registering the world
 - mediates different perspectives on a changing world
 - enables moving in and out of connection with the world
- Critical properties of registration
 - multiple routes to reference
 - morning star, evening star
 - ability to exceed causal reach
 - person closest to average height in Gorbachev's office now
 - indexical reference
 - the one in front of him

1. On this slide, object means in the real-world.



Traditional Mechanisms

```
class Line {  
    private Point p1, p2;  
  
    Point getP1() { return p1; }  
    Point getP2() { return p2; }  
  
    void moveBy(int dx, int dy) {  
        p1.moveBy(dx, dy);  
        p2.moveBy(dx, dy);  
    }  
    void setP1(Point p1) {  
        this.p1 = p1;  
    }  
    void setP2(Point p2) {  
        this.p2 = p2;  
    }  
}
```

```
class Point {  
    private int x = 0, y = 0;  
  
    int getX() { return x; }  
    int getY() { return y; }  
  
    void moveBy(int dx, int dy) {  
        x = x + dx; y = y + dy;  
    }  
    void setX(int x) {  
        this.x = x;  
    }  
    void setY(int y) {  
        this.y = y;  
    }  
}
```



stream of instructions

- Modular program structures
- Give rise to execution stream
- Only one place has static direct causal access to given point in stream
 - via single module that gives rise to it
 - equivalent to ‘3 static assumptions’

Join Point Models

```
class Line {  
    private Point p1, p2;  
}
```

```
aspect UpdateSignaling {  
  
    pointcut change(Shape shape):  
        this(shape) &&  
        (execution(void Shape.moveBy(int, int)) ||  
         execution(void Shape.set*(*) ));  
  
    pointcut topLevelChange(Shape shape):  
        change(shape) && !cflowbelow(change(Shape));  
  
    after(Shape s) returning: topLevelChange(s) {  
        Display.update(s);  
    }  
}
```

```
int getX() { return x; }  
int getY() { return y; }  
  
void moveBy(int dx, int dy) {  
    x = x + dx; y = y + dy;  
}  
void setX(int x) {  
    this.x = x;  
}  
void setY(int y) {  
    this.y = y;  
}  
}
```



stream of instructions

- Pointcuts
 - pick out dynamic join points in stream
 - unconstrained by original program modularity
 - ‘register’ instructions in own form
 - create a crosscutting modularity

Join Point Models

- (De)compose software in different ways
- Register aspects out of fog of undifferentiated points
 - means of identifying JPs (aka pointcut) registers
 - aspects/slices/concerns... group over that
- Connect and have effect through that registration
 - means of semantic effect (aka advice)
- Critical properties of registration
 - multiple routes to reference
 - `void setX(int nx) { ... }, call(void setX(int)), cflow(...)`
 - exceed causal reach
 - `within(com.sun..*)`, `!within(com.mycompany.mysystem)`
 - indexical reference
 - `cflow(...)`



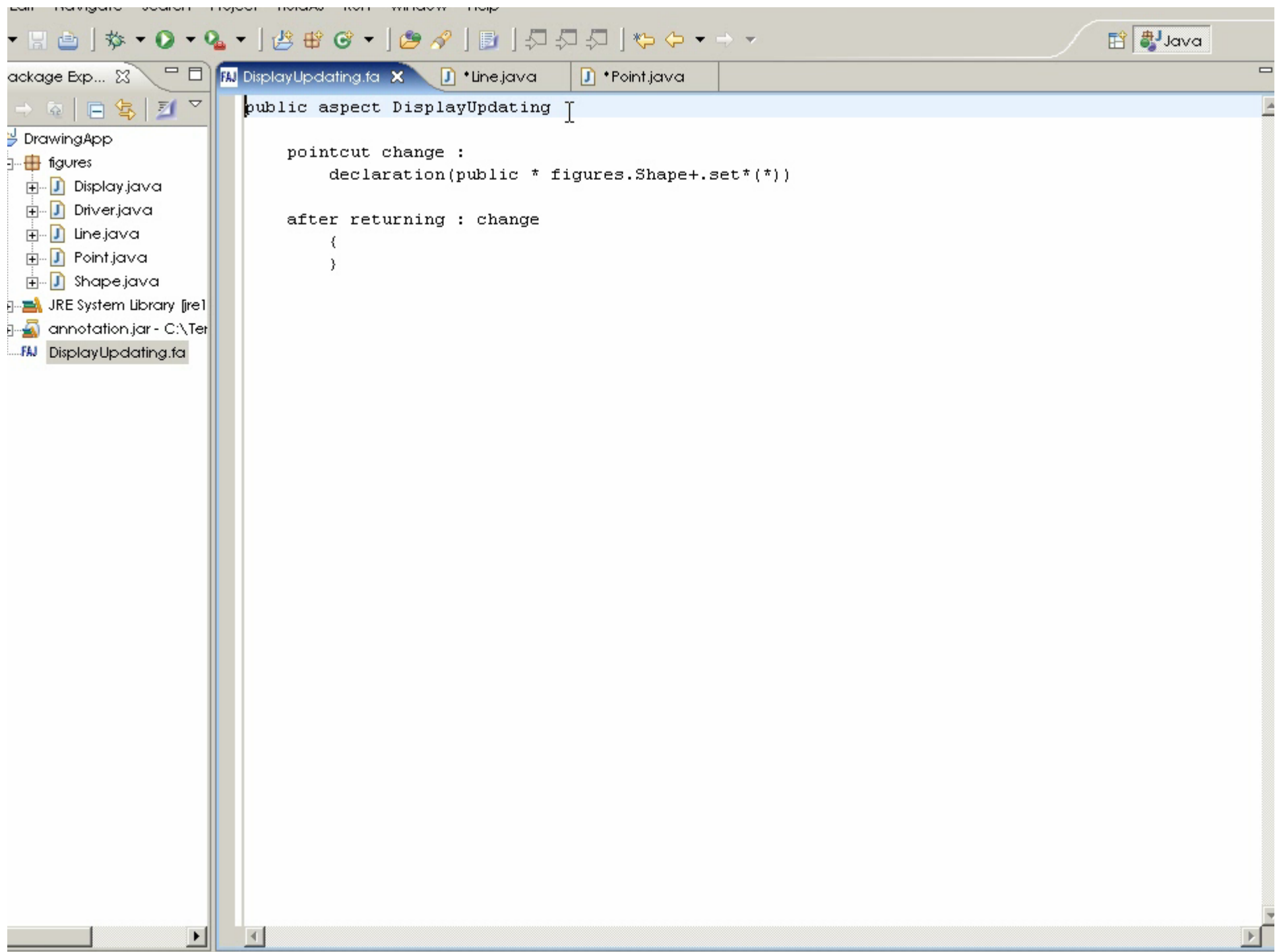
Variation in Modularity

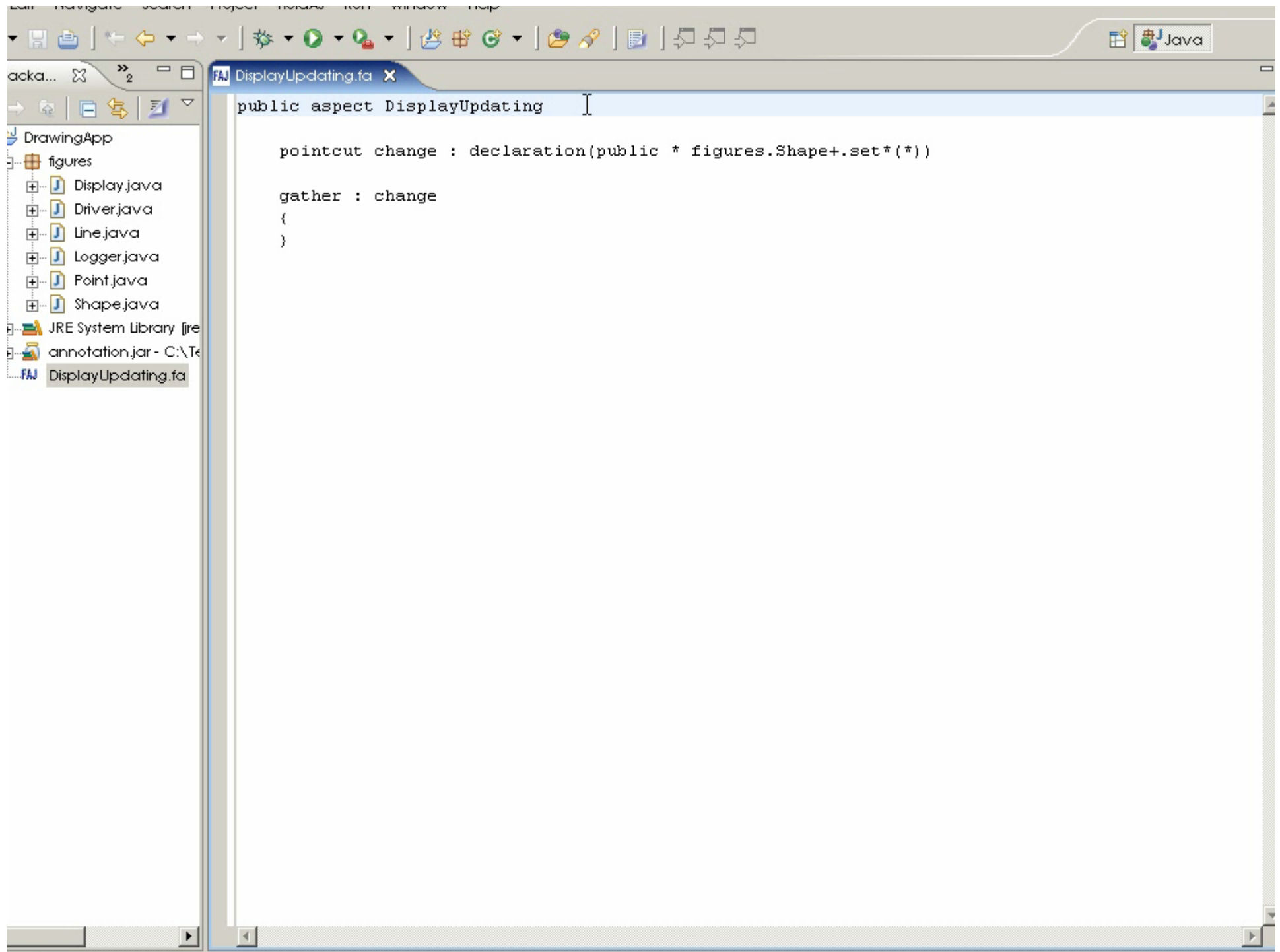
A module is a ~~statically~~ localized unit of source code with a well-defined ~~static~~ interface.

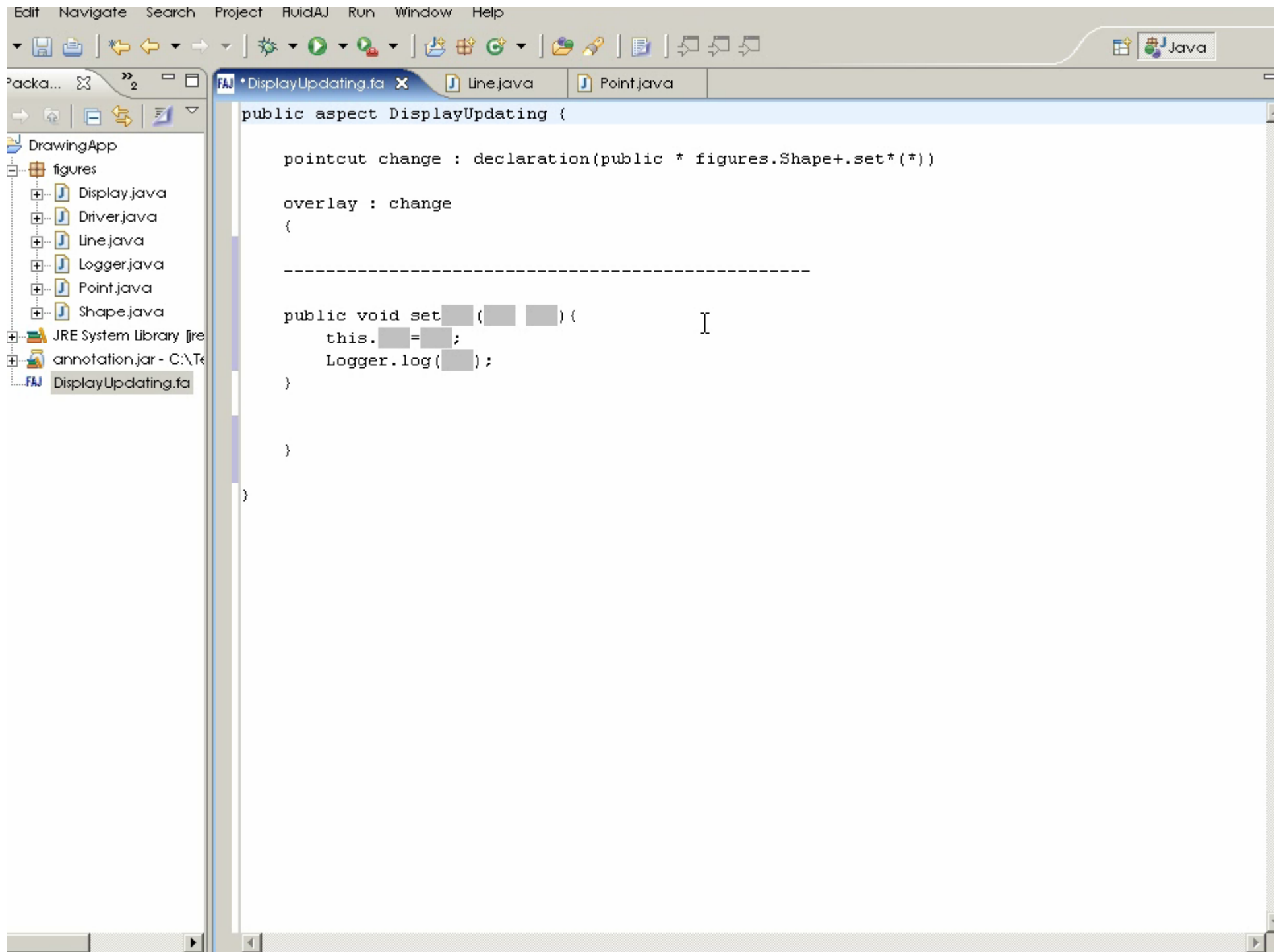
Abstraction means hiding ~~permanently~~ irrelevant details behind an interface.

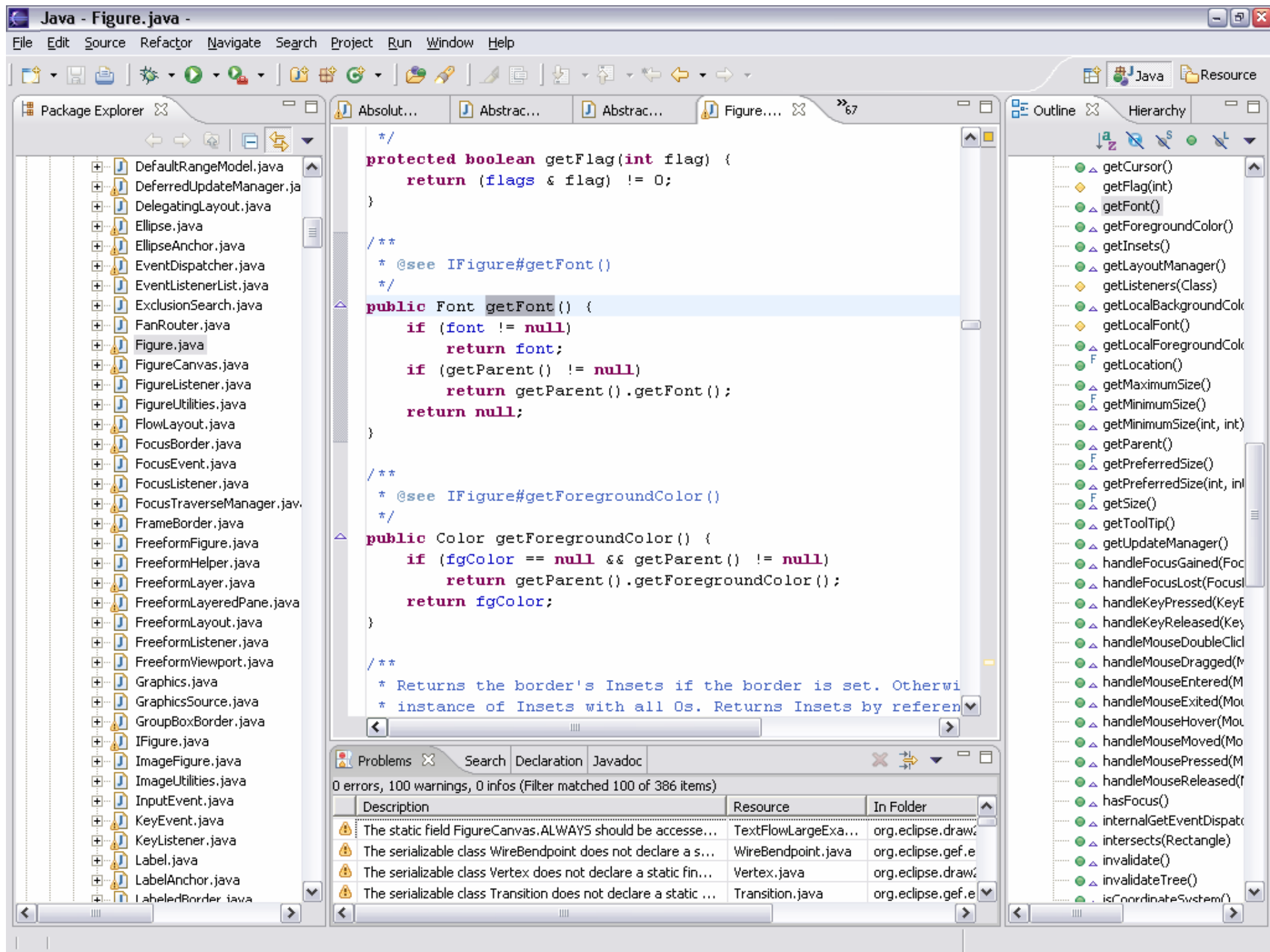
- Remove static restriction
- Consider what could go in 3 holes
- Be more 'radical' rather than conservative

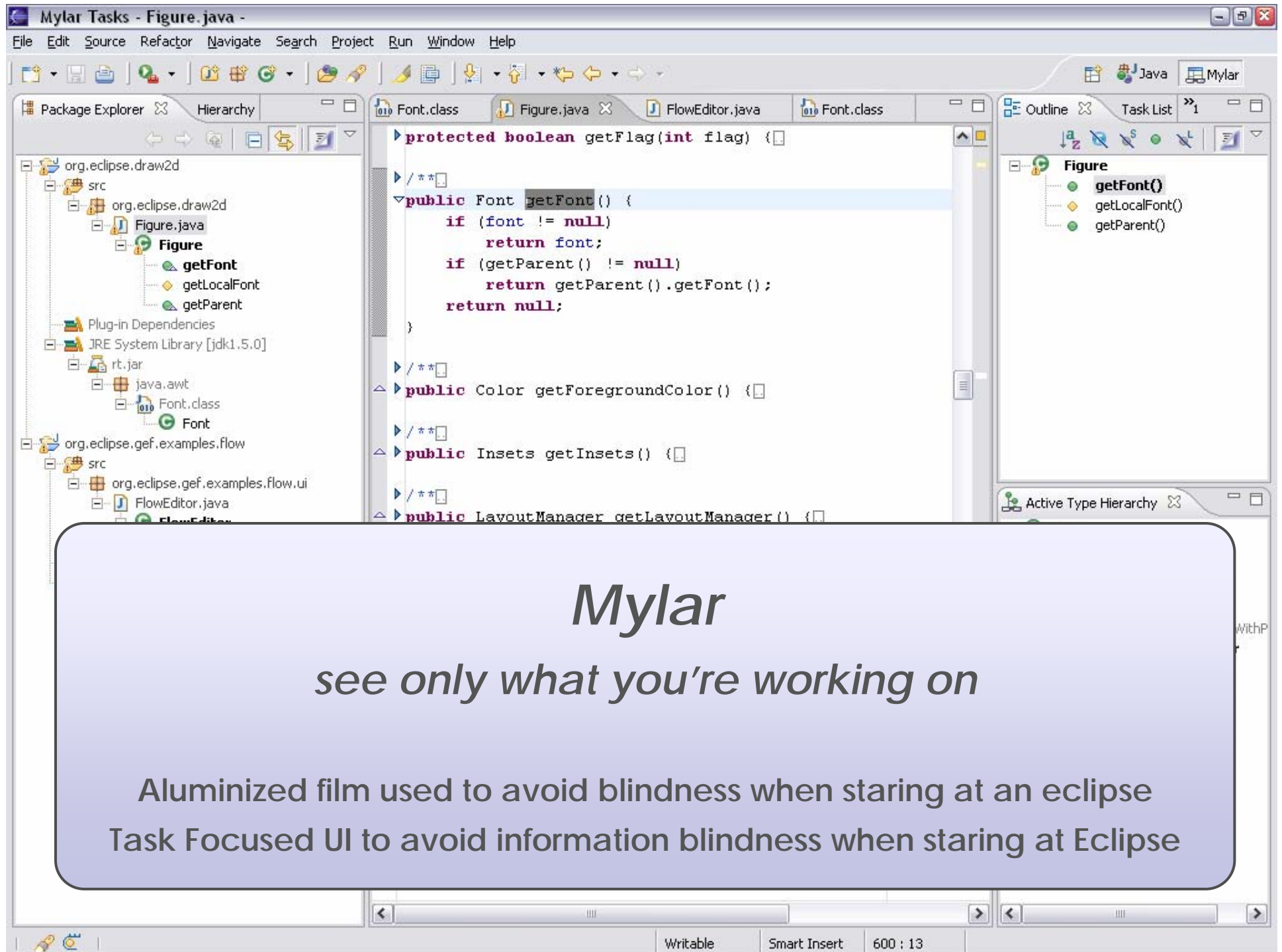












Radical Research in Modularity

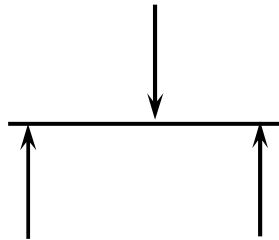
A module is a ~~statically~~ localized unit of source code with a well-defined ~~static~~ interface.

Abstraction means hiding ~~permanently~~ irrelevant details behind an interface.

- AspectJ: static, per-configuration, per aspect
- Hyper/J: re-arrangeable, static, re-arrangeable
- Fluid AOP: fluid, fluid, fluid
- Mylar: dynamic filter, static, dynamic filter
- <please add here>

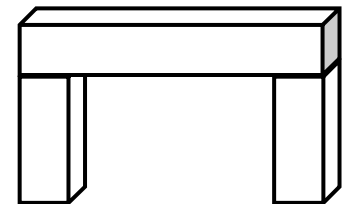


Models, Programs and Systems

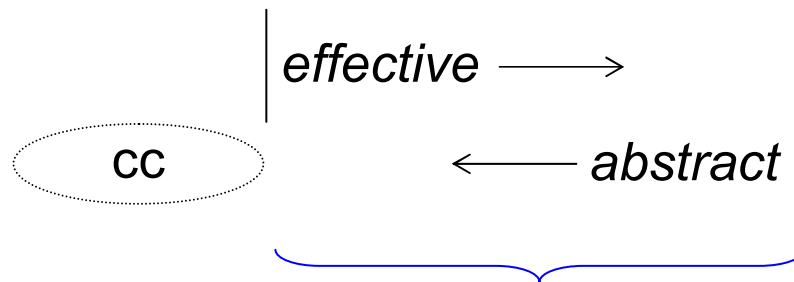


model

```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```



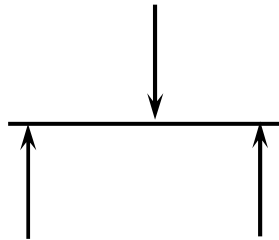
system



*programs live in
this magic space*

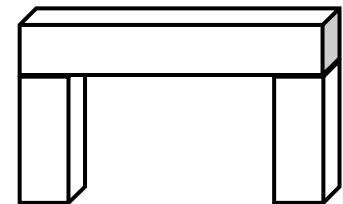
*Brian's account talks
(in part) about this space*

Models, Programs and Systems

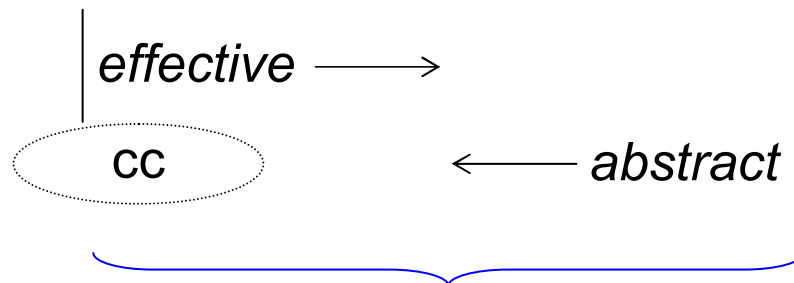


model

```
i = 1
while (i < 4) {
  print(i)
  i = i + 1
}
```



system



*programs live in
this magic space*

*Brian's account talks
(in part) about this space*



SPLC 2006

Summary for the 1st International Workshop on Agile Product Line Engineering

APPLE 2006

**August 21, 2006 Baltimore, MD
(W2)**

www.lsi.upc.edu/events/apple

Hosts

Pere Botella and Kendra Cooper

Organizers

Xavier Franch and Kendra Cooper



- **Purpose of the Workshop**

Bring together a diverse group of participants interested in integrating agile methods and software product line engineering methods to enable the rapid development of high quality software with reduced cost

- **Workshop Participants**

- 14 international participants from academia and industry
- Our participants:
 - had expertise in agile methods and/or software product line development
 - had expertise in diverse domains
 - were very enthusiastic

- **Overall Organization of the Full Day Workshop**

Two morning sessions were for presenting and discussing an interesting collection of seven research and experience papers

- Papers have been posted on the workshop website
- Presentations will be posted soon



SPLC 2006

- **Overall Organization of the Workshop (cont.)**

Two afternoon sessions for identifying and discussing topics of interest to both researchers and practitioners

Four major topics were identified:

- 1. Experiences using “pure” agile, “pure” product line engineering, and combinations of agile and product line engineering methods**
 - strengths, limitations of “pure” approaches**
 - experiences using combined approaches**
- 2. Issues in defining an Agile Software Product line method**
 - start with an agile approach and tailor it?**
 - start with a product line approach and tailor it?**
- 3. Empirical design/assessment of approaches in industry and academia**
 - How to provide evidence that an approach is useful**
- 4. How to share of theory/empirical results between academia and industry**

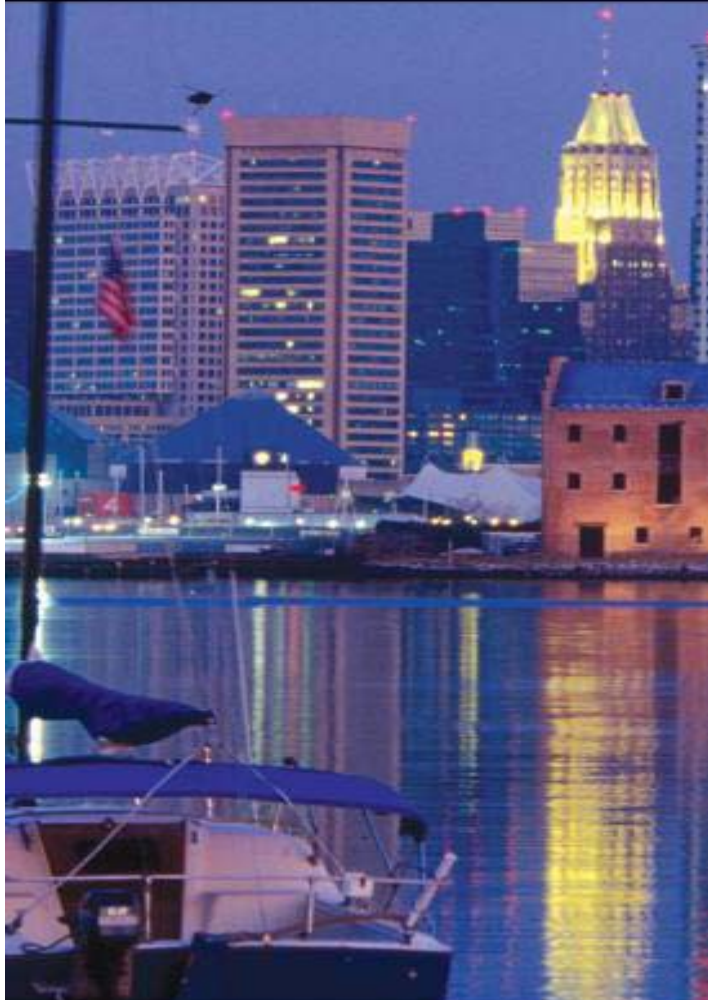
**The first two topics were discussed in a lively atmosphere
We ran out of time to discuss the other two topics**



• **Future communication and events**

- **Presentations will be posted on the website**
- **Summary of our discussion will be distributed for comments, revised, and then posted on the website**
- **Contact information for participants will be distributed for future discussion and collaboration**
- **Special issue of Journal of Systems and Software (JSS) has been arranged, call for papers will be announced shortly**
- **APPLE 2007 looks promising!**

SPLC 2006



Managing Variability for Software Product Lines: Working With Variability Mechanisms

**SPLC
21 August 2006
Baltimore, Maryland, USA**



Workshop capsule

GOAL: As a community, we wish to produce guidance for use of variation mechanisms of sufficient detail to be useful to a software product line engineer.

ATTENDANCE: ~30

FORMAT: Five short paper presentations before lunch (out of 13 submitted). Four **working groups** formed to work 3 hours after lunch.

WORKING GROUPS:

- Criteria influencing selection of variation mechanisms
- Criteria focused on cost
- Criteria focused on performance
- Variation mechanisms and evolution



Workshop results

A comprehensive cookbook of the form

Situation	Variation mechanism
Situation	Variation mechanism
...	...

...is too complex a task for any one workshop.

To make progress, the workshop community proposed a Wiki site be set up, where contributors can submit *patterns* of variation mechanism usage.

Pattern: (problem, context) solution



Variation mechanism Wiki

The workshop focused on how to describe the conditions under which one would choose a variation mechanism – the criteria for selection. For example

- Required run-time quality attributes of the products
- Required non-run-time quality attributes of the products
- Required binding time(s) of the mechanism
- Domain of application; stability of domain
- Cost of building the mechanism (over time)
- Cost of exercising the mechanism (over time)
- People skills required to build and to exercise
- Tools and automation compatibility requirements
- Legacy asset compatibility



Wiki

Discussed preliminary design of the wiki for

- A pattern contributor
 - “In what domains has this mechanism been successfully used?”
 - “With which languages/environments has this mechanism been successful used?”
 - “What quality attribute effects does this mechanism have?”
 - Etc.
- A pattern consumer



Next workshop

There was group sentiment to have another workshop at the next SPLC.

“Price of admission” might be contributing a pattern.

In the interim we will try to set up the wiki and seed it with some patterns. This will involve settling on a pattern language to use, which will evolve over time as we gain experience.



Workshop execution

Five short paper presentations (out of 13 submitted):

- “Implementing a Variation Point: A Pattern Language,”
John M. Hunt, John McGregor
- “Using Costing Information as Decision Support in Variability
Management,” Holger Schackmann, Horst Lichter
- “Coherent Integration of Variability Mechanisms at the
Requirements Elicitation and Analysis Levels,”
Nicolas Guelfi, Gilles Perrouin
- “Using Dependencies to Select Variability Realization
Techniques in Software Product Lines,”
Roberto Silveira Silva Filho, David F. Redmiles
- “Good Practice Guidelines for Code Generation in Software
Product Line Engineering,”
Neil Loughran Iris Groher Awais Rashid

Variability Management – Working with Variability Mechanisms

Proceedings of the Workshop held in conjunction with the
10th Software Product Line Conference (SPLC-2006)



Editors:
Paul Clements
Dirk Muthig

IESE-Report No 152.06/E
Version 1.0
October 15, 2006

A publication by Fraunhofer IESE

Fraunhofer IESE is an institute of the Fraunhofer Gesellschaft.

The institute transfers innovative software development techniques, methods and tools into industrial practice, assists companies in building software competencies customized to their needs, and helps them to establish a competitive market position.

Fraunhofer IESE is directed by
Prof. Dr. Dieter Rombach (Executive Director)
Prof. Dr. Peter Liggesmeyer (Director)
Fraunhofer-Platz 1
67663 Kaiserslautern

Table of Contents

1	Introduction	1
2	Towards the Use of Dependencies to Manage Variability in Software Product Lines Roberto Silveira Silva Filho and David F. Redmiles	4
3	Using Costing Information as Decision Support in Variability Management Holger Schackmann, Horst Lichter	16
4	Module Structures and SPL Variability Philipp Schneider, Phonak Group	27
5	A first step towards a framework for the automated analysis of feature models David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés	39
6	Version management tools as a basis for integrating Product Derivation and Software Product Families Jilles van Gorp, Christian Prehofer	48
7	Coherent Integration of Variability Mechanisms at the Requirements Elicitation and Analysis Levels Nicolas Guelfi, Gilles Perrouin	58
8	Product Line Architecture Variability Mechanisms Steve Livengood	72
9	Implementing a Variation Point: A Pattern Language John M. Hunt and John D. McGregor	83
10	On the Architectural Relevance of Variability Mechanisms in Product Family Engineering Arnd Schnieders	97
11	Good Practice Guidelines for Code Generation in Software Product Line Engineering Neil Loughran, Iris Groher and Awais Rashid	108
12	Beyond Code: Handling Variability in Art Artifacts in Mobile Game Product Lines Vander Alves, Gustavo Santos, Fernando Calheiros, Vilmar Nepomuceno, Davi Pires, Alberto Costa Neto, Paulo Borba	124

1 Introduction

1.1 Overview

Managing variability is the essence of software product line practice. Variability enters the product line picture through the need for different features, deployment on different platforms, the desire for different quality attributes, and the accommodation of different deployment scenarios. Eventually, every need for variability manifests itself in one way or another in the actual artifacts that populate a product line's core asset base.

"Variation mechanisms" is the name we give to the constructs that achieve variation at the artifact level. Catalogs of these mechanisms have been published, and they come in a wide variety. They may be

- requirements-level (such as the use of feature models, use case extensions, etc.)
- application-level (such as the use of configurators or program generators)
- architectural (such as plug-ins, or component replacement or replication)
- design-level (such as aspects), or
- implementation-level constructs (such as inheritance or parameterization)
- runtime variation (such as reflective programming or conditionals)

Selecting the correct variation mechanism(s) can have a dramatic effect on the cost to deploy new products, react to evolutionary pressures, and in general maintain and grow the product line. But selection remains an ad hoc process in nearly all product line organizations.

A one-day workshop entitled "Managing Variability for Software Product Lines: Working With Variability Mechanisms" was held in conjunction with the 2006 Software Product Line Conference on August 21, 2006 in Baltimore, USA. Its goal was to begin to fill the void between variability requirements visible to those who deal with features and other product-level concerns, and the variation mechanisms visible to creators and consumers of a product line's core assets. The goal of the workshop was to begin to codify a body of knowledge for the informed and purposeful selection of variation mechanisms to use in a software product line's core assets.

Advertised topics of interest included

- Reasoning frameworks for variability selection
- Factors that affect the selection of variability mechanisms

- Cost models to enable reasoned selection of variability mechanisms
- Variability mechanisms especially suited for non-software artifacts
- Binding time issues from an strategic or economic viewpoint

1.2 Program Committee

The program committee for the workshop comprised (in alphabetical order):

- Michalis Anastasopoulos, Fraunhofer IESE
- Martin Becker, Fraunhofer IESE
- Jan Bosch, Nokia
- Stan Jarzabek, National University of Singapore
- Charles Krueger, BigLever Software, Inc.
- Juha Kuusela, Bosch
- Klaus Schmid, University of Hildesheim
- Rob van Ommering, Philips Research

1.3 Workshop Execution

Approximately thirty people attended the workshop, making it the workshop with the highest attendance at the conference. Thirteen position papers were submitted, which are included in this report. The workshop was highly interactive and focused on making tangible progress towards answering specific questions relating to best practices in variability management.

During the morning session there were short presentations of five selected papers. The bulk of the workshop, however, was reserved for discussions and overall conclusions. Participants were assigned to groups reflecting specific topics. Then, the discussions were carried out by raising and debating relevant questions related to every topic. Finally, a member of each group presented that group's conclusions.

1.4 Workshop Conclusions

A comprehensive cookbook of the form ...

- Situation → Variation mechanism
- Situation → Variation mechanism
- ...

...is too complex a task for any one workshop. To make progress, the workshop community proposed a Wiki site be set up, where contributors can submit patterns of variation mechanism usage.

Pattern: (problem, context) → solution

The workshop focused on how to describe the conditions under which one would choose a variation mechanism – the criteria for selection.

For example

- Required run-time quality attributes of the products
- Required non-run-time quality attributes of the products
- Required binding time(s) of the mechanism
- Domain of application; stability of domain
- Cost of building the mechanism (over time)
- Cost of exercising the mechanism (over time)
- People skills required to build and to exercise
- Tools and automation compatibility requirements
- Legacy asset compatibility

There was widespread group sentiment to have another workshop at the next SPLC, wherein the “price of admission” might be contributing a pattern. In the interim the organizers intend to the wiki and seed it with some patterns, as well as solicit patterns from the product line community at large. This will involve settling on a pattern language to use, which will evolve over time as we gain experience.

1.5 Outline

The remaining chapters present the ten papers that were submitted to the workshop and have been accepted by the program committee.

2 Towards the Use of Dependencies to Manage Variability in Software Product Lines

Roberto Silveira Silva Filho and David F. Redmiles
Donald Bren School of Information and Computer Sciences
Department of Informatics, University of California, Irvine, CA 92697-3430, USA
{rsilvafi, redmiles}@ics.uci.edu

Dependencies have been used in feature-oriented product line to manage feature incompatibilities alternatives, activation requirements, and to support the built of different software configurations. Few studies, however, have been devoted to study the role of dependencies in limiting the variability of product lines and as important criteria for selecting variability realization techniques. Understanding those variability implications, allow us to better understand the design trade-offs of a particular product line, to bound its variability dimensions, and to decide, as early as in the design phase, where and which variability realization techniques to apply. This position paper proposes the use of dependencies as one of the main criteria to be used in bounding variability and choosing the appropriate variability realization techniques. We motivate and exemplify our approach with a publish/subscribe product line.

2.1 Introduction

The goal of software product lines is “to capitalize on commonality and manage variability in order to reduce the time, effort, cost and complexity of creating and maintaining a product line of similar software systems”¹ Whereas in a software product line, reuse allows the reduction of the costs of producing similar software systems, variability permits the customization of a software family to the different needs of the product line members. It also facilitates the incorporation of new software products in a product family, thus adding value to the product line (Baldwin and Clark 2000). Variability, however, comes with a cost. The more variability a product line supports, the more complex its implementation becomes. Dependencies from the problem domain and from the variable feature set pose a limit in the solution variability by hindering the reuse of existing features, increasing the solution complexity and posing extra burden in the configuration strategies.

In the design of software product lines, feature-oriented approaches have been successfully used in the industry, where there seems to be a consensus on the use of feature-oriented design models. In this approach, the variability among software products is modeled in terms of features. Features represent units of variation in different versions of the software (Svahnberg, Gorp et al. 2005).

¹ Extracted from www.softwareproductlines.com/introduction/concepts.html

Ideally, features must be implemented (or realized) as independent (or modular) pieces of code that can be specified and built in isolation from one another. In practice, however, features are not completely independent; those units of variability usually require different services from other features either through hierarchical decomposition or by all sorts of “use” dependencies. According to the complexity of the system, this hierarchy can have an arbitrary number of levels. Moreover, due to variability and domain constraints, incompatibilities among features may also exist. Hence, in a feature-oriented model, the representation of features dependencies is crucial. Features may be incompatible (exclude dependency), may require additional or complementary features (usage dependency), may modify the behavior of other features (modify dependency), and may also have special activation dependencies (multiple, exclusive, subordinate, concurrent or sequential) (Lee and Kang 2004). Additionally, due to reuse and the variability realization technique employed, different features may share common sub-features. Those dependencies can impact the design and behavior of software in different ways. For example, they can originate unforeseeable behavior in software as the case of feature interference problem (Cameron and Velthuisen 1993) where the combination of two or more features that share common resources can interfere with one another in unpredictable ways.

With such a variety of dependencies and relations between features, the choice of the appropriate variability realization technique (such as whether to employ component or aspectual decomposition; or to apply compile time or runtime variability for example) may be the difference between a tangled and a modular implementation, guaranteeing important characteristics to the software product line such as maintainability, modularity, comprehensibility and the extensibility. Moreover, the understanding of those dependencies and their impact in the system properties may support variability bounding and design simplifications, allowing designers to better assess the costs of varying or fixing certain aspects of the product line.

2.2 Background and Motivation

The study of the role of dependencies in software product lines has gained recent attention from the research community. The focus, however, has been more on the use of those dependencies to prevent architecture configuration mismatch, and less on the study of the impact of those dependencies on the system design complexity and their impact on the variability of software. For example, in the FODA (Kang, Cohen et al. 1990), FORM (Kang 1998), RSEB (Griss, Favaro et al. 1998) methods and in the generative approach in (Czarnecki and Eisenecker 2000), dependencies are used to model usage interactions (alternative, multiple, optional and mandatory) as well as incompatibility relations (exclusive or excludes), with the focus on configuration management and conflict resolution. Recently, (Ferber, Haag et al. 2002) stresses the importance of dependency analysis in feature diagrams, and proposes a separate fea-

ture-dependency model that complements the existing feature tree. Additionally, it characterizes different interactions between features such as intentional, environmental, and usage dependencies. Finally, in a more recent work, (Lee and Kang 2004) studied the role of dependencies on modeling runtime feature interactions, introducing the notion of activation and modification dependencies in feature diagrams.

In the implementation domain, feature dependencies usually manifest themselves in the form of coupling between the components that implement those features, in special data and control coupling occur as a consequence of activation and usage dependencies. Those dependencies have different impacts in the variability of the final solution. Whereas control coupling usually limits the activation order of the different pieces of software, data coupling can limit the variability and reuse of those components. (Parnas 1978; Stevens, Myers et al. 1999)

On the light of those problems, different variability realization approaches have been used. For example, (Lee and Kang 2004) propose a set of object-oriented realization strategies to address activation dependencies. Those strategies are presented in the form of design patterns derived from existing Factory, Proxy and Builder patterns (Gamma, Helm et al. 1995). In essence, those patterns focus on managing and enforcing activation dependencies by promoting the late-binding of the components that implement the many software features. Whereas useful in many contexts, this modular (object-oriented) decomposition is not always sufficient to address other kinds of dependencies, especially cross-cutting variability dimensions or aspects, originated from more fundamental problem dependencies. This motivated recent work such as (Garcia, Sant'Anna et al. 2005), where Aspects are used to modularize design patterns.

In this position paper, we argue towards a more deep understanding of the role in dependencies in software product lines. Not only as important information for configuration management support, but as main factors to be considering in the design, bounding and variability realization selection phases. We exemplify the role of dependencies with the following case study.

2.2.1 Case Study: Publish/Subscribe Product Line

Pub/sub infrastructures provides an asynchronous message service where information providers (publishers) generate information in the form of events (or messages); whereas information consumers (or subscribers) express interest in those events by means of subscriptions. Based on the subscription (an expression or query that can include the event content, order or time restrictions), the events are routed from the publishers to the appropriate subscribers. The events are then delivered according to a notification policy. Different publish/subscribe systems have been built from scratch in the last years, being tai-

lored to different application domains. This observation motivated our research in the development of YANCEES (Silva Filho and Redmiles 2005), a flexible infrastructure, that can be tailored to the needs of different publish/subscribe application domains.

One important step in the design of a product line is the problem domain analysis and the identification of the main units of variability (the features). In our design, we adopted the framework proposed by (Rosenblum and Wolf 1997). In this model, routing, event, notification, observation (or subscription), timing and resource are the main design concerns of a publish/subscribe infrastructure. They represent the main variability dimensions in our model. The event model defines how the event is represented (for example: tuples, record, object or plain text). The routing model defines the strategy used to match subscriptions to events (whether by the content, by a specific field (topic), or by a dedicated channel where all events produced are delivered to the subscriber). The notification model defines how to deliver events to the subscribers once they are matched with the subscription (push or pull). The subscription model defines the query language, and all the commands that can be part of it. Those commands may operate over the content or over the order of the events. The timing model defines guarantees with respect to the total or partial order of the events. The resource model specifies how the infrastructure is implemented (whether distributed in a peer-to-peer or hierarchical fashion, or whether it is centralized). Note that the abstract features (routing, event, notification, subscription, timing and resource), define a set of variability dimensions that are further specialized in the following hierarchy level by different optional features.

A possible feature diagram representing such product line variability is depicted in Figure 1. The Diagram uses a UML notation. Stereotypes are used to express optionally (OR relation) and exclusivity (XOR relation). An optional feature can be selected together with other optional features in the same level, for the same super feature. Abstract features appear as the first level under the pub/sub infrastructure concept, and are not marked with stereotypes. Aggregation indicates containment and composition implies a part-role relation of the pub/sub concept. When no stereotype is used, the features or concepts are mandatory.

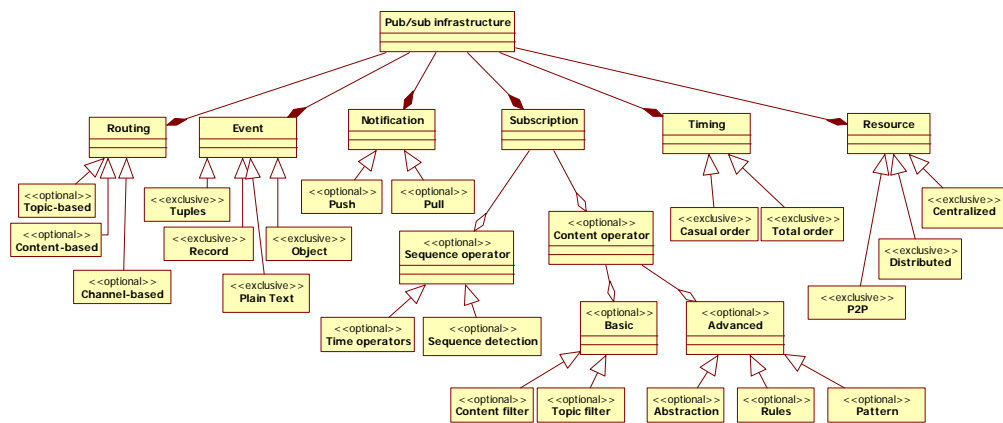


Figure 1: Feature Diagram of Pub/sub systems

2.2.2 The Role Of Fundamental And Configuration-Specific Dependencies

Feature diagrams, as the one presented in Figure 1, express the basic model of feature-oriented product line design. A feature diagram represents a tree of features where the root represents a concept. A concept is usually represented by a set of mandatory, optional and abstract features. The first level of a feature diagram usually represents a set of abstract features that implement a concept, in our example, a pub/sub infrastructure. Those abstract features are parents of more specific optional, alternative, exclusive and mandatory features. For example, routing, subscription, notification, event, timing and resource features, which variability is further defined by means of concrete features such as record-based for event, total order timing, content-based filtering for subscription and so on.

In a feature-based approach, we define as **fundamental dependencies**, those relations between abstract features that are imposed by the problem domain. In other words, they comprise the interactions between generic abstract features that define the problem domain. In our example, events, routing, subscriptions and notifications are inter-related by the very publish/subscribe process itself: events are routed following subscriptions, generating notifications. This process is common to all publish/subscribe infrastructures and provides a conceptual model where configuration-specific features can be “plugged in”; moreover, they define our variability dimensions in the product line. In contrast to fundamental dependencies, **configuration-specific dependencies** are those dependencies that exist between optional features and/or the components that implement them. For example, some notification servers may use pull notification approach, which require event persistency and user authentication support, sub-features present only in certain members of the product family, specializations of the generic notification feature.

In our case study, as seen in Figure 1, variability exists in every design dimension of the system. A problem then surfaces when those variability dimensions (or abstract features) are further refined and implemented. First, the dependencies between features are not easily visualized in the diagram; they in fact are not represented. Second, the abstract features are usually implemented as part of the base or common code, whereas the optional features are provided on specific configurations. As a consequence, the abstract pub/sub fundamental dependencies implicitly impact the implementation of each feature of the model. For example, the implementation of a subscription command in our case study must account for the way events are represented (records, objects, plain text, attribute-value pairs) and the routing guarantees that the infrastructure provides (total or partial order of events). Those implicit dependency usually become encoded in the base implementation of the product line, and will manifest themselves later in the implementation of the infrastructure. As a consequence, every time the event format or the routing policies vary, the subscription commands that depend on those parameters will need to vary, either by providing alternative implementations, or by accounting for this variability in the feature implementation itself.

This observation has important consequences: (1) **the combinatorial explosion of features**. Feature dependencies work as an important variability limitation factor for the product line: the more dependencies exist between features, the harder it is to manage all possible combinations of features in a product line. (2) **Reduced generality of features**: dependencies also limit the reuse of existing features and their implementations since changes in one feature can impact features that depend on it. This fact prevents the unbounded generalization of product lines, and places a theoretical limit in how one can leverage reuse in such systems. (3) **Limited extensibility**: Since new features need to cope with the existing dependencies in the model their implementation tends to be more complex and prone to errors. Using Parnas terminology (Parnas 1976), the dependencies from the incomplete program (or base code), impact the variability of the product family as a whole since implicit domain relations (or dependencies) are usually inherited from the incomplete code.

As a consequence, in the design of a product line, a balance between variability and reuse needs to be achieved. In this position paper, we argue that the dependencies represent the key to understand and tackle this problem. With such information, designers can either limit or fix the variability of a certain dimension or choose a variability realization technique that minimizes such coupling. In spite of its importance, both fundamental and configuration-specific dependencies are usually not explicitly represented in feature diagrams. Instead of augmenting existing diagrams with such information, we propose the use of alternative diagrams and models to elucidate those dependencies (Kruchten 1995).

2.3 Approach

In our work, we propose the study of dependencies in a more fundamental way, understanding the implications of those feature relationships in the choice of the most adequate variability realization techniques. Our approach comes from the observation that abstract problem features, originated in the problem domain are usually chosen as main variability dimensions in the problem. Those features, however, have implicit fundamental dependencies (or coupling) that becomes part of the common code of the architecture and ends up constraining the variability of the product line as a whole, and the features specifically. Hence, the analysis of those fundamental dependencies can help us select the proper combination of variability dimensions and feature realization approaches in order to reduce their impact in the software that will ultimately implement the model. Also, depending on the target software and hardware platforms or different environmental constraints and limitations, the dependency model can provide a basis to decide which variability dimensions to fix, and/or which ones to make variable. The approach can be summarized as:

1. Perform initial feature domain analysis
2. Identify the abstract features in the domain that define an abstract system commonality
3. Perform a dependency analysis with respect to the abstract feature dependencies
4. Consider the target platform hardware and software constraints
5. Bound the mandatory features variability in order to minimize the impact of dependencies
6. Choose appropriate variability realization technique for the base code
7. Choose the variability realization technique for the optional code

The variability realization approach used to model a feature will impact the variability of the system in different ways. For example, (Czarnecki and Eisenacker 2000) defines two main decomposition approaches: Modular decomposition and Aspectual decompositions. When used in conjunction, those approaches can complement one another and reduce the impact of dependencies (or coupling) in the code. Those approaches are discussed in more detail in the next sections.

2.3.1 Making Dependencies Explicit

The feature diagram in Figure 1 does not allow the visualization of all the dependencies between the problem features, depicting only optionally, aggregation, specialization and exclusion. One way to visualize dependencies is proposed by (Ferber, Haag et al. 2002). The use of graphs to represent such infor-

mation, however, does not scale. Instead, we propose the use the DSM (Dependency Structure Matrix) notation as that used by (Baldwin and Clark 2000). In this representation, dependencies between parameters, or in our case features, are represented in the form of a square matrix. In our approach, instead of representing only the number of dependencies or their simple existence, with an X for example, we label the dependencies with the kind of coupling they provide. A 'D' is placed in the matrix to represent a data dependency if the i^{th} column depends on the dimension in the j^{th} row; similarly, a 'C' is placed to represent a control dependency. The DSM of our case study is presented at Figure 2.

Figure 2 reveals some interesting dependencies between the main variability dimensions of the product line. Note that the event model and its representation directly impacts the subscription and routing models. A change in the event format requires a change in the subscription language and routing algorithms due to a strong data coupling between those two features or dimensions (the event content and format itself). Timing is another crucial feature in the model. A change in the routing algorithm may impact the timing guarantees of the product line (guaranteed delivery and total order of events), which will impact the subscription language semantics. A change in the resource model may also affect the timing model. For example, in a hierarchical distributed system, the total order of events may not be available. Finally, the notification model is orthogonal to the other features. Since it manages only events, it can vary independently from the other features. Hence, this simple analysis allows us to draw two lessons: first, by analyzing the dependencies between the abstract features, as in Figure 2, a system architect can identify relations that are not initially obvious in the original feature model of the system; and second, as a consequence, she can adopt some strategies to minimize the impact of dependencies in the final variability of the product line. For example, use different decompositions such as aspects, or even fix some variability dimensions, such as the event model. In doing so, the design of a product line can be optimized and pitfalls such as hidden dependencies can be assessed, managed or limited.

		A	A1	A2	B	B1	B1.1	B1.2	B2	B2.1	B2.2	B2.3	B2.4	B2.5	C	C1	C2	C3	D	D1	D2	D3	D4	E	E1	E2	F	F1	F2	F3
A	notification	.																												
A1	_push		.																											
A2	_pull			.																										
B	subscription				.																									
B1	_sequence					.																								
B1.1	_order						.																							
B1.2	_interval							.																						
B2	_content								.																					
B2.1	_filterContent									.																				
B2.2	_filterTopic										.																			
B2.3	_abstraction											.																		
B2.4	_rules												.																	
B2.5	_pattern													.																
C	routing					C									.															
C1	_topic					C										.														
C2	_content					C											.													
C3	_channel					C												.												
D	event					D			D	D	D	D	D	D	D	D	D													
D1	_tuple					D			D	D	D	D	D	D	D	D	D													
D2	_record					D			D	D	D	D	D	D	D	D	D													
D3	_text					D			D	D	D	D	D	D	D	D	D													
D4	_object					D			D	D	D	D	D	D	D	D	D													
E	timing					C	C	C	C							C	C	C												
E1	_casualOrder					C	C	C								C	C	C												
E2	_totalOrder					C	C	C								C	C	C												
F	resource																								C	C	C			
F1	_federated																								C	C	C			
F2	_centralized																								C	C	C			
F3	_P2P																								C	C	C			

Figure 2: DMS showing the dependencies between the features.

2.3.2 Variability realization techniques

Modular decomposition aims at decomposing the system into a hierarchy of modules (components, objects, methods and so on) in such a way that cohesion in the modules are maximized, while coupling is minimized. Many recurring solutions exist to help in the design of such systems, including design patterns (Lee and Kang 2004), conditional compilation, templates and other approaches (Svahnberg, Gurf et al. 2005) that are usually designed for functional or object-oriented programming.

Modular decomposition is not always possible to accomplish in object-oriented languages due to what is called crosscutting concerns, that can be a result of non-functional requirements for example. Also, due to the lack of modularity in many object-oriented design patterns (Hannemann and Kiczales 2002), their use throughout the system, and especially in the base code, makes it hard for the software to support changes and support evolution. For each design pattern applied, new dependencies are introduced to the product line architecture, as well as additional configuration costs to manage those dependencies.

Moreover, it is usually the case that a feature is mapped not to a single component but to a set of components installed in different parts of the base code. In our example, this approach can be applied in the case of Notification and Resource models in Figure 2. Those models do not depend on any other model.

Aspectual decomposition aims at decomposing a system into a set of perspectives. Each one of those perspectives comprises concerns that refer to a common model. Another way of describing this approach is that aspects encapsulate the coupling that might exist between components that implement a single feature, that otherwise would usually become hard coded in the many components that implement a feature. In the example of Figure 2, control dependencies can be modularized as aspects, that weave to the base code, the appropriate algorithm, according to the selected timing constraints. Examples of modularizations using aspects are discussed at (Garcia, Sant'Anna et al. 2005) and (Hannemann and Kiczales 2002).

Other decomposition and strategies are also possible. In the pub/sub example, data coupling can be addressed by using reflection as described in (Eugster and Guerraoui 2001).

2.3.3 Bounding and Restrictions

In order to reduce the coupling between dimensions, one simple alternative is the bounding of variability dimensions. After a dependency analysis, designers can opt for limiting the variability of some abstract features. For example, the event dimension in Figure 1 can be restricted to support only attribute-value representation. This representation is generic enough to be used as topic and object-based representations by using some conventions. The textual event representation, however, is not trivially supported in this model, and must be adapted to conform to attribute-value representations. A trade-off, therefore, between variability and bounding exists and needs to be considered, according to the applications the system will support.

Environmental restrictions also play an important role. Variability techniques such as aspectual decomposition or reflection may not be available in a given platform. In this case, whenever bounding is not an option, more traditional approaches such as design patterns (Lee and Kang 2004) may be used.

2.4 Final Considerations

Feature dependencies are important in product line design since they can limit the variability of the software. There are two major categories of dependencies: fundamental problem dependencies, coming from the common problem features, that originate the common base code; and feature-specific dependencies that are originate from optional and alternative features in the product line. The nature of the dependencies between the product line features, especially those that are part of the problem domain, have a deep impact in the resulting variability of the software solution, a consequence of the coupling of the components in the final code implementation. The understandings of those variability

dimensions, allow us to analyze the design trade-offs of a particular product line, and must be considered in the design of product lines. The use of dependency analysis tools such as DSMs can help designers identify, as early as in the design phase, the dependencies between the main variability dimensions of the problem. This information is crucial to support designers in choosing the appropriate variability realization techniques that can better suit the implementation (or realization) of the features, or to select which variability dimension to fix in order to simplify the architecture design.

Whereas dependencies alone are not the only criteria to be used in selecting a variability realization approach, their understanding provides an important input to practitioners in understanding the variability restrictions imposed by the problem domain, helping them in the choice of the adequate technique for their case.

2.5 Acknowledgements

This research was supported by the U.S. National Science Foundation under grants 0534775, 0326105, and 0205724 and by the Intel Corporation.

2.6 References

- Baldwin, C. Y. and K. B. Clark (2000). Design Rules, Vol. 1: The Power of Modularity. Cambridge, MA, MIT Press.
- Cameron, E. J. and H. Velthuisen (1993). "Feature interactions in telecommunications systems." IEEE Communications Magazine 31(8): 18-23.
- Czarnecki, K. and U. W. Eisenecker (2000). Generative Programming - Methods, Tools, and Applications, Addison-Wesley.
- Eugster, P. T. and R. Guerraoui (2001). Content-Based Publish/Subscribe with Structural Reflection. 6th USENIX Conference on Object-Oriented Technologies and Systems, COOTS'01, San Antonio, TX.
- Ferber, S., J. Haag, et al. (2002). "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line." Lecture Notes in Computer Science. Second International Conference on Software Product Lines, SPLC'02 2379: 235-256.
- Gamma, E., R. Helm, et al. (1995). Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Publishing Company.

Garcia, A., C. Sant'Anna, et al. (2005). Modularizing design patterns with aspects: a quantitative study. Aspect-oriented software development, Chicago, Illinois, ACM Press.

Griss, M. L., J. Favaro, et al. (1998). Integrating Feature Modeling with RSEB. Fifth International Conference on Software Reuse.

Hannemann, J. and G. Kiczales (2002). Design Pattern Implementation in Java and AspectJ. 17th Annual ACM conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02), Seattle, Washington.

Kang, K. C. (1998). "FORM: A Feature-Oriented Reuse Method with Domain Specific Architectures." Annals of Software Engineering 5: 345-355.

Kang, K. C., S. G. Cohen, et al. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study - CMU/SEI-90-TR-021. Pittsburgh, PA, Carnegie Mellon Software Engineering Institute.

Kruchten, P. B. (1995). "The 4+1 View Model of architecture." IEEE Software 12(6): 42-50.

Lee, K. and K. C. Kang (2004). "Feature Dependency Analysis for Product Line Component Design." Lecture Notes in Computer Science - 8th International Conference on Software Reuse, ICSR'04 3107: 69-85.

Parnas, D. L. (1976). "On the Design and Development of Program Families." IEEE Transactions on Software Engineering SE-2(1): 1-9.

Parnas, D. L. (1978). Designing software for ease of extension and contraction. 3rd international conference on Software engineering, Atlanta, Georgia, USA, IEEE Press.

Rosenblum, D. S. and A. L. Wolf (1997). A Design Framework for Internet-Scale Event Observation and Notification. 6th European Software Engineering Conference/5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Zurich, Switzerland, Springer-Verlag.

Silva Filho, R. S. and D. Redmiles (2005). Striving for Versatility in Publish/Subscribe Infrastructures. 5th International Workshop on Software Engineering and Middleware (SEM'2005), Lisbon, Portugal., ACM Press.

Stevens, W. P., G. J. Myers, et al. (1999). "Structured design." IBM Systems Journal 38(2-3): 231 - 256.

Svahnberg, M., J. v. Gorp, et al. (2005). "A Taxonomy of Variability Realization Techniques." Software Practice and Experience 35(8): 705-754.

3 Using Costing Information as Decision Support in Variability Management

Holger Schackmann, Horst Lichter

Decisions on the scope of a software product line and the selection of variability realization techniques have a large impact on costs and therefore substantially affect the economic viability of a product line approach. But there is a lack of information to guide these decisions, since there is no clarity on the cost implications of variability. To close this gap, we propose an approach to costing that enables a more accurate allocation of incurred costs to the features within the software product line. More accurate costing information provides guidance for future scoping decisions and enables an assessment of the cost implications of different variability realization techniques as a first step towards a cost model for the reasoned selection of appropriate techniques.

3.1 Introduction

The principal decisions in managing variability within a software product line (SPL) are made on two levels. On the requirements level the scope of the SPL must be defined, i.e. it must be identified which features should be variable within the SPL and decided which variants should be supported. Secondly, it has to be decided how to implement the variability by selecting appropriate variability realization techniques [1]. These decisions have to be taken during the initial development as well as during the further evolution of an SPL.

Decisions on the two levels are rather intertwined. The planned variability within the SPL influences the choice of variability realization techniques. These can in turn limit or facilitate adjustments of the scope.

The remainder of this paper is organized as follows. Section 2 depicts the challenges of managing the variability during evolution of a SPL. Section 3 discusses influencing factors for the selection of variability realization techniques. Section 4 depicts deficiencies of current costing approaches in gathering the costs implications of variability and proposes a costing approach based on features as cost objects. Section 5 then describes how this information can be used to guide variability management decisions. Section 6 provides a brief summary.

3.2 Scoping during SPL evolution

There exist several approaches for the initial definition of the scope [3]. But the problem of scoping during maintenance and evolution of an SPL is rather unexplored. In practice we encounter that SPLs are introduced gradually by exploiting commonalities between products that had been developed in customer

specific projects. Thus an exhaustive scoping was not performed and scoping decisions may not be documented or lost during further development.

Typical risks associated with initial scoping are a too big or too small definition of the scope [4]. During evolution of a product line we rather face the problem of an increasing variability within the scope. Several reasons may lead to the introduction of new variants of features. Due to deadline pressure it may often be faster to produce a customer specific solution instead of spending additional effort on investigating chances for reuse and coordinating necessary changes with other product developments. Fulfilling new customer requests with new customer specific features may be motivated by the concept of customer orientation and disregard of the follow-up costs. A large amount of variability is accompanied by an increasing technical and cognitive complexity. So the offering of features can not completely be communicated to the customer. Similarities of customer requests to existing features will be difficult to re-cognize, so similar features will unconsciously be developed several times.

Summarized, variability becomes a major cost driver, since each customer specific feature must be maintained, integrated in subsequent releases, tested separately and possibly considered during deployment, user training and customer support. A problem is that these costs are not transparent. Therefore it is not possible to balance the follow-up costs against the value that a specific feature offers to the customers.

3.3 Selection of variability realization techniques

Since the scoping activity analyses potential products and possible future extensions, this information is a basic guidance for the selection of variability realization techniques. For example it should be considered to how many products a specific feature will be relevant. If it is necessary in most of the products a variability mechanism should be selected that allows easy configuration of this feature during application engineering. If the feature is only needed in a few products it may suffice to provide a realization technique that requires product specific development, e.g. specializing particular classes.

Besides the scoping decisions there are some other influencing factors that will be described in the following.

3.4 Requirements of the application domain

There typically exist non-functional requirements of the application domain which discourage the use of certain variability realization techniques. As an example performance requirements can rule out variability realization techniques

with runtime binding. Memory constraints may require a binding before start-up time.

Furthermore there may be the requirement to allow easy reconfiguration or upgrade of the system, possibly even at runtime, which necessitates a runtime binding of the variability.

Fritsch et al. describe an approach to evaluate variability realization techniques according to qualities, like extensibility or performance [5]. But an evaluation in general is difficult since the fulfillment of a quality depends on other influencing factors like the architectural context and the sound application of the variability realization technique.

3.4.1 Existing architecture

The variability realization techniques that are already used in the existing product line architecture have to be taken into account. Relying on established techniques reduces the complexity within the architecture, compared to the usage of a many different realization techniques. Furthermore those techniques and their implications may be better understood by the developers.

Also the presence of legacy components within the SPL can rule out the usage of novel variability realization techniques like aspect oriented programming.

3.4.2 Organization

Different organizational models can be applied to SPL development, dependent on factors like the size of the product line or the physical location of the staff [6]. Choosing appropriate variability realization techniques can substantially reduce the necessary coordination efforts between different development units. For example a domain engineering unit can support a certain variant feature by providing alternative architectural components. During application engineering only one variant has to be selected. Thus the complexity visible to application engineering units is reduced and the interfaces between the development units are kept small.

Nevertheless application engineering units may need some flexibility for product specific adaptations. Choosing variability realization techniques that are open for adding variants during application engineering can help to define the points where product specific adaptations are allowed, while other variant features can be kept under control of the domain engineering unit. This can help to limit the growth of variability within the product line, which may be caused by the independent development of similar features for different products within the SPL.

3.4.3 Follow-up costs

The selection of variability realization techniques has a large influence on the costs of subsequent activities. Some techniques may have higher initial development costs in domain engineering, but therefore allow a fast development of new variants during application engineering.

Costs for product derivation and testing are recurring costs in each release cycle of a product. Using variability realization techniques that allow a configuration-based product derivation, these costs will probably be lower compared to variability realization techniques that require the development and maintenance of product specific assets.

Costs for deployment of a product are recurring costs for each customer site and for each product release or update. The installation at the customer site may be more or less complex, partly influenced by the applied variability realization techniques. If complex configuration steps are necessary, an expert might be needed for installation on customer site, while in other cases an automatic installation is possible.

Furthermore maintenance costs are affected, since the selection of variability realization techniques influences testability and comprehensibility. For example using preprocessor directives can lead to a scattering of variation points, which makes debugging and changing the software more difficult.

Summarized, there are many influencing factors on the follow-up costs of variability decisions, such as the number of customers, the number of products and their release cycles, as well as the amount of change requests during evolution.

3.4.4 Current practice

In current practice selection of variability realization techniques is rather based on ad hoc decisions or primarily influenced by the techniques already used in the architecture as well as by non-functional requirements from the application domain, as far as they have been identified. Other factors remain unconsidered due to insufficient scoping during evolution and no transparency on the follow-up costs of variability decisions. The multitude of influencing factors complicates the construction of a cost model that would enable a reasoned selection of variability realization techniques. Therefore sound decision criteria for the selection of variability realization techniques, as well as for scoping during evolution are lacking. We believe that gaining more clarity on the cost implications of variability will help to overcome this situation.

Existing cost models for SPL development rather follow a top down approach and support SPL investment decisions on a high level (see [2] for an overview).

Gathering more accurate information on the costs of variability can provide complementary input for these models as a first step towards the development of a cost model for the reasoned selection of appropriate variability realization techniques.

3.5 Costing approaches

3.5.1 Deficiencies in traditional costing methods

Traditional costing methods in software development usually allocate costs either to customer projects, maintenance projects or internal development projects. This may suffice for accounting and budget control. But with multiple reuse the relationship between the direct labor hours that went into development and the costs of software breaks down[7]. Significant costs are incurred by the reuse infrastructure. Neither the benefits of reusing assets within application engineering, nor the costs of introducing additional variabilities can therefore be gathered with these costing methods.

3.5.2 Activity based costing

Activity based costing was originally developed in the context of manufacturing industry, motivated by problems in managing an increasing variety within the product portfolio[8][9]. It was recognized that traditional costing systems failed to allocate the overhead costs caused by additional product variants in a reasonable way, since these costs are allocated to products on a per-unit basis. This leads to suboptimal decisions in portfolio definition and product construction, due to an unconsciously cross-subsidization of rather exotic products.

Rather than viewing work products as direct consumers of overhead resources, activity based costing introduces the activities of the production process to stand in between work products and consumed resource costs. This allows a more accurate allocation to the cost objects consuming the activities, e.g. products, services or customers. The improved cost transparency can guide decisions to establish profitable customer relationships, e.g. by changing the construction of products, changing the prices, reconfiguring or replacing products, improving production processes, or eventually abandon a product completely.

3.5.3 Activity based costing in software development

Analogously to the cost anomalies in manufacturing the costs of developing and maintaining an SPL are for the greater part indirect or overhead costs, which can not easily be allocated to a certain product or customer.

Can an activity based costing approach be applied to software development? Fichman and Kemerer have proposed the adoption of activity based costing to component based software development, to address the need for economic incentives for investments into reusability [7].

While the relevant activities can be identified based on defined development processes, it is not evident what should be the primary cost objects. The allocation of activity costs to customers or software components respectively raises many difficulties. It is not clear how development and maintenance costs of core assets can be distributed to customers. The use of software components as cost objects is problematic when there is no direct relation of an activity to a certain component, e.g. activities like requirements engineering or system testing.

3.5.4 Features as cost objects

As described before, the number of customer specific features has a large impact on costs. Therefore one can analyze which features are standard features that are relevant for most customers and which features can be seen as more exotic features, since they are only included in products for one or a few customers. Gathering the costs caused by those exotic features more accurately, would help to attain more clarity on the influence of variability on costs. Therefore we propose the use of features as the primary cost object for activity based costing in SPL development. Feature models describe the commonalities and variabilities within an SPL [10][11]. So this approach can capture the costs associated to those variabilities. Figure 3 gives an overview of the cost assignments that are described in the following.

Personnel costs are dominating the costs of software development. Other costs, like costs for tools or hardware, can be apportioned to the personnel costs. Therefore it may suffice to concentrate on personnel costs and differentiate between hourly wages according to the qualification level required for an activity. Existing change request management systems, task management systems or time registration systems provide a basis for the resource costs assignment to activities, since they basically enable a detailed gathering of labor hours for most activities.

Since features are visible in all phases of the development life cycle, most development assets, like requirements, software components, test cases, documentation, change requests and even support calls can be linked to one or more features. Thus the cost for most activities can be distributed to features as cost objects.

A catalogue of relevant activities can be developed based on a defined development process, including for example activities like requirements engineering,

design, implementation and test during the initial development of a new feature, as well as activities during evolution of a feature, like the handling of change requests and bug reports and the development of product-specific adaptations of features. To enable an analysis of the cost structures it should be differentiated between activities in domain engineering and application engineering and furthermore between activities associated to the initial development of a feature and further maintenance activities.

It will not in all cases be possible to allocate the costs of each activity to a single feature, e.g. initial requirements engineering activities which can be associated to several features. But since a feature model typically contains concept features to allow a hierarchic structuring of the model, such costs may be allocated to a concept feature that structures a set of features associated with the activity. In other cases it can be reasonable to distribute the costs of an activity to more than one feature, e.g. if the interaction of features is the subject of a maintenance activity. So this should allow the allocation of most of the former indirect costs, like maintenance costs of core assets, to features.

There will nevertheless be activities that can not be allocated to features. Either an association to a feature or a set of features can not be identified, e.g. for management activities, or gathering the related costs would be impractical or too expensive, e.g. the assignment of the costs of documentation activities to features. These remaining costs must be allocated to projects as before.

To enable a further allocation of the cost from features to customers, an assessment of the importance or value of the provided features to the different customers is necessary. It might not be possible to express this value in monetary units, but it suffices to assess the contribution of a feature to user satisfaction on a simple scale of values. Techniques like the Kano method [12] or Quality Function Deployment [13][14] may be applied to this purpose. The assignment of the costs of a feature to the customers can then be based on the different importance of the feature to its customers.

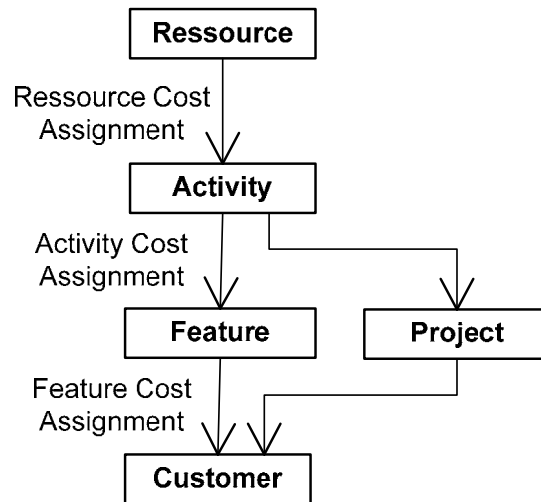


Figure 3: Features as cost objects

3.6 Using costing information in variability management

The depicted approach to costs enables an improved cost transparency. A better understanding of the relation between initial development costs of a feature and its follow-up costs can improve future cost estimation and guide the pricing of development and maintenance contracts. The costing information can also guide variability management decisions in several ways.

3.6.1 Guidance of scoping decisions

The allocation of activity costs during development and evolution to features will give a more realistic view on the costs associated to a feature. If costly features are identified, it must be analyzed which importance or business value these features provide to the customers, and which profits or competitive advantage is gained for the development organization by offering these features.

Features which are not that important for the customers but generate high additional costs can be identified. They can either be replaced by existing similar features, modified in a manner that allows a better fit to the SPL architecture, or can possibly be abandoned completely. If a feature is important for certain customers, it must be examined up to which extent the customers can be charged with the real costs.

In all cases the importance of a feature or customer for the market strategy has to be considered. It can be a reasonable decision to take a loss in order to open

or develop a market. But to decide on the strategy, there must be some estimation where and how much money is lost.

3.6.2 Support of selection of variability realization techniques

More accurate information on costs will enable to analyze the cost structures associated to a feature and its variants. It can be gathered which costs are spent for the initial development of a feature, for the development of new variants of this feature, for maintenance of this feature and its variants, and eventually for testing in subsequent releases. Furthermore it can be gathered how all of these costs are distributed between domain engineering and application engineering. Moreover the influence of change requests on these costs can be retrieved.

An investigation of the cost implications of different variability realization techniques can be based on comparisons of the cost structures for different features and its variants and searching for regularities or recurring patterns. For example an analysis of the distribution of maintenance costs between domain engineering and application engineering may help to identify which realization techniques require high maintenance efforts within application engineering and which techniques leave most of the maintenance work within domain engineering thereby possibly reducing overall maintenance costs. It can also be investigated, if there are variability realization techniques which facilitate the development of new variants.

Furthermore these observations may enable better estimations of future costs, e.g. the costs for developing a new variant of a feature, the costs of product derivation, or the maintenance costs of existing features.

All together, the empirical observations enabled by the described costing approach facilitate a better understanding of the cost implications of variability. Therefore this can be a first step towards the development of cost model for the reasoned selection of variability mechanisms.

3.7 Further work

In our further work we aim at developing a detailed model for a costing approach based on features as cost objects. Within the context of an ongoing industry cooperation project we will conduct a case study on gathering cost information in an SPL development process. To provide appropriate tool support, the collection of costing information should be integrated with our existing tool set for feature modeling [15].

3.8 Summary

Scoping decisions during the evolution of a software product line, as well as the selection of variability realization techniques are in current practice rather based on ad hoc decisions. The difficulties in steering the variability and selecting appropriate realization techniques lead to increasing maintenance costs. We believe that one cause of this problem is a lack of sound information on the costs implications of variability.

This resembles a problem suffered in manufacturing industry. Given an increased number of product variants, traditional costing approaches were unable to provide a sound cost allocation and therefore lead to wrong decisions in product portfolio definition and product construction. This problem was addressed with the development of new costing approaches like activity based costing.

In this paper we propose the application of activity based costing to software product line development using features as primary cost objects. Features provide a natural basis for allocating costs and are anchored in existing techniques for variability modeling.

Bringing together cost information based on features as cost objects and the assessment of the customer value of features provide a sound guidance for scoping decisions. Furthermore the cost implications of the selection of variability realization techniques can be investigated to obtain an empirical basis for the development of a cost model for the reasoned selection of variability mechanisms.

3.9 References

- [1] Svahnberg, M., J. van Gurp, J. Bosch (2004): A taxonomy of variability realization techniques. Software – Practice and Experience. Wiley Interscience, Chichester.
- [2] Clements, P.C., J.D. McGregor, S.G. Cohen (2005): The Structured Intuitive Model for Product Line Economics (SIMPLE). CMU/SEI-2005-TR-003, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [3] Schmid, K. (2003): Planning Software Reuse – A Disciplined Scoping Approach for Software Product Lines. PhD Theses in Experimental Software Engineering, vol 12, Fraunhofer IRB Verlag, Stuttgart.
- [4] Clements P., L. Northrop (2001): Software Product Lines: Practices and Patterns, Addison Wesley, Boston, Massachusetts.

- [5] Fritsch, C., A. Lehn, T. Strohm (2002): Evaluating Variability Implementation Mechanisms. In Schmid K., Geppert B. (Eds.) Proceedings of the Second International Workshop on Product Line Engineering PLEES'02, IESE-Report No. 056.02/E. Fraunhofer IESE, Kaiserslautern.
- [6] Bosch, J. (2000): Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education, Harlow.
- [7] Fichman, R., C. Kemerer (2002): Activity Based Costing for Component-based Software Development. Information Technology and Management, vol 3 (1/2), 137-160. Springer, Netherlands.
- [8] Schuh, G. (2005): Produktkomplexität managen – Methoden, Strategien, Tools. (in German) Hanser Verlag, München.
- [9] Kaplan, R.S., R. Cooper (1997): Cost and Effect. Harvard Business School Press, Boston, Massachusetts.
- [10] Kang K., S. Cohen, J. Hess, W. Novak, A. Peterson (1990): Feature-Oriented Domain Analysis (FODA) Feasibility Study. CMU/SEI-90-TR-021, ADA235785, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.
- [11] Lee, K., K.C. Kang, J. Lee (2002): Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In Gacek C. (Ed.) Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools, ICSR-7, Austin, Texas. 62-77. LNCS 2319, Springer, Berlin.
- [12] Kano, N., N. Seraku, F. Takahashi, S. Tsuji (1996). Attractive quality and must be quality. In J. D. Hromi (Ed.) The best on quality (Vol. 7). 165-186. ASQ Quality Press, Milwaukee, Wisconsin.
- [13] Akao, Y. (1988): Quality Function Deployment QFD: Integrating Customer Requirements into Product Design. Productivity Press, Portland, Oregon.
- [14] Helferich, A., G. Herzwurm, S. Schockert (2005): QFD-PPP: Product Line Portfolio Planning Using Quality Function Deployment. In Obbink, H., Pohl, K. (Eds.) Proceedings of the 9th International Software Product Line Conference, Rennes. 162-173. LNCS 3714, Springer, Berlin.
- [15] von der Maßen, T., H. Lichter (2004): RequiLine: A Requirements Engineering Tool for Software Product Lines. In van der Linden, F. (Ed.) Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena. 168-180. LNCS 3014, Springer, Berlin.

4 Module Structures and SPL Variability

Philipp Schneider, Phonak Group

During the last two years I participated in the development and application of a SPL for hearing instrument fitting software. One core aspect of its architecture is the module view, because most of the required variability was realized by applying well known structural patterns. This paper examines how effective these patterns could be applied to release half a dozen fitting software variants.

4.1 Context

When we started with the SPL the main focus was on the “right” module structure which would allow us to build future fitting applications. We were convinced that this, what I call “Lego approach to variability”, would help us to realize most of our variability requirements. Like many other teams we put a lot of emphasis on defining the software modules and their interfaces. How to define module structures is well known and documented in the pattern community. The resulting architecture helped us to deliver a set of highly successful fitting software applications. The coarse grained module structure gave us some important level of variability, but in many cases we had to augment the structural patterns with other variability mechanisms, to support finer grained variability requirements.

In the following paper I would like to present the structural patterns of our SPL architecture and share our experience on how effective these patterns could be applied to solve our variability requirements.

4.2 Separation of Presentation and Business Logic

4.2.1 Forces and Structure

The presentation is often the only thing which needs to be changed to create a new brand or to address a different market segment. It is crucial to separate the presentation aspects from functional business logic aspects. It is important to consider all presentation aspects, including error messages from lower layers and data items which are stored in a database and should be presented differently according to the chosen locale. To be able to release a new product with a different user interface, presentation and the business logic should be placed in separate software modules.



Figure 4: Separation of presentation and business logic

4.2.2 Experience

- It really pays off to separate the presentation of your application into a separate module. In most cases it might make sense to exclude the presentation from the SPL, because it usually needs to be tailored to a specific application. But even if the presentation is excluded from the SPL, at least a common user interface framework will be required, to serve as a skeleton for the presentation layer.
- Key is how the presentation is bound to the business. What happens very quickly is that different application developers start to share “glue” code and small utilities, which encapsulate what they consider “ugly” module interfaces. In this case it is important to respond to the demand from the SPL users and either refactor the “ugly” interfaces or to put these code elements into shared libraries, which can then become part of the SPL.
- Internationalization and localisation is an important aspect of a presentation layer. We strongly suggest to use existing variability mechanisms already present in your programming environment (like .NET satellite assemblies), instead of rewriting a custom solution. This will allow you to use existing 3rd party tools.

4.3 Encapsulate Knowledge about 3rd Party Modules

4.3.1 Forces and Structure

There are two possible approaches to treat 3rd party modules. You can consider them integral part of your technological choice and use them everywhere, or you can try to encapsulate them in one module. If you do not encapsulate a 3rd party module, it must be available on all your target systems. If you are afraid this might limit the scope of your core assets too much, you can choose to write an adapter. This is expensive, but allows you to exchange that technology by simply rewriting the adapter.

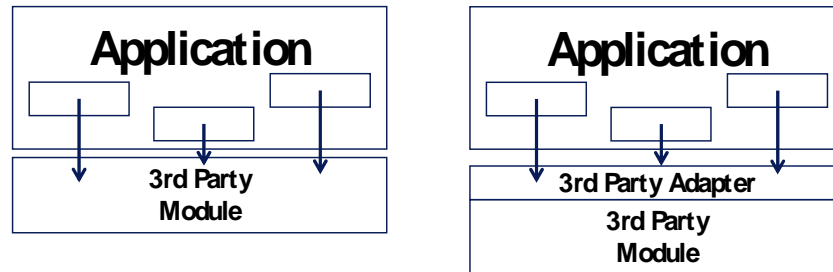


Figure 5: Usage of an adapter to encapsulate 3rd party modules

4.3.2 Experience

- Some modules are encapsulated completely behind adapters. This would make it easier to port the SPL to a target platform, where these encapsulated modules are not available. A side effect of using adapters is that the developers of the SPL are not exposed to the 3rd party module at all. This has a psychological effect: developers tend to ignore the underlying 3rd party modules and forget to look for bugs in those modules.
- In the future I would be more careful to use an adapter. The adapter certainly locates all the code which is dependent on the 3rd party module in one place, but usually the interfaces just take care about the data mapping and neglect behavioural aspects. I would recommend using adapters only if your SPL needs to support several different 3rd party module configurations. Otherwise the effort to develop an adapter is too high.

4.4 Composition

4.4.1 Forces and Structure

The “Lego” approach towards a SPL is to provide individual modules, which are then integrated to create different applications. By choosing different modules, different variants can be created. The Lego approach works wonderfully for electronic circuits or any hardware components.

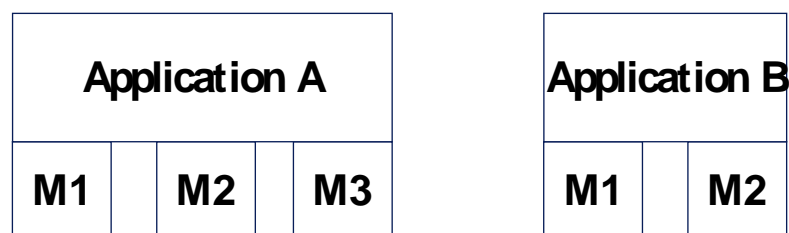


Figure 6: Composition of software modules

4.4.2 Experience

- We really thought that SPL users would be able to choose the right modules and integrate them into different applications. After three different applications this really never happened. The users of the SPL explained that they prefer to start with a working application. Therefore they copy an existing application and then strip it slowly until they have the requested behaviour. This is much easier than starting with individual modules which first need to be assembled to do something useful.
- Through composition the users of the SPL can choose if they want a certain feature or not. In our experience composition is too clumsy, to allow to refine the overall behaviour otherwise, then to turn on or off certain features.
- Another big challenge for variability through composition is the definition of the module interfaces. How are you going to exchange information between the modules? If you decide to use typed, synchronous interfaces, the types in the interfaces will constrain you on how you will be able to plug the modules together.

4.5 Layering

4.5.1 Forces and Structure

Most systems use layering, to group modules together which operate on the same abstraction layer. If a layer deals with a certain aspect of the system, it can be replaced by a different implementation to create a variant of the product.

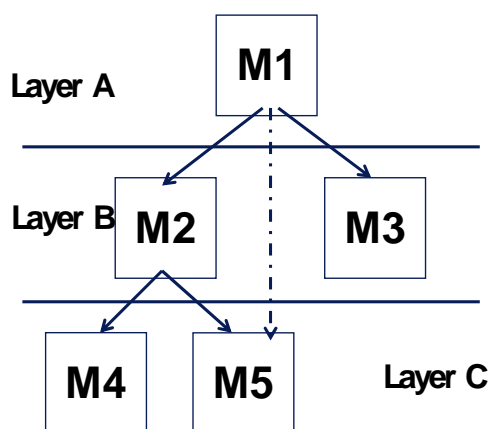


Figure 7:

Layering of modules

4.5.2 Experience

If you plan to replace a layer in the SPL, we urge you to use a strict layering! In our SPL the presentation layer is allowed to communicate with the two top layers below it. This reduced the number of data transformations required, but coupled the presentation tightly to the business logic layers. For the future we plan to reduce this coupling by following a strict layering for the presentation layer.

4.6 Runtime configuration

4.6.1 Forces and Structure

To allow fine grained configuration of a module it can offer a configuration interface. Such an interface is usually realized using XML to describe the variability. In the case of runtime configuration, the module will load the configuration at execution time, and adapt its behaviour accordingly. Depending on how much needs to be configurable, the module configuration might only contain key/value definitions, or even implement part of an algorithm, which is then interpreted (or precompiled) at runtime. To allow backwards and forward compatibility of applications, the configuration has to be carefully versioned and managed.

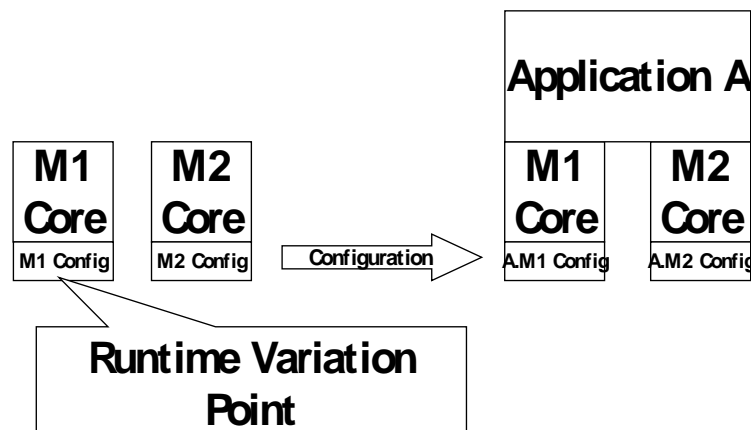


Figure 8: Runtime configuration of modules

4.6.2 Experience

- A well defined module structure, together with runtime configuration will allow you to satisfy most of your variability requirements. But be careful to plan beforehand, where and how you will store your configurations. Make

the configurability of a module part of its specification and only offer what is requested as a variation point.

- Make sure to not break the module encapsulation with the configuration. Do not create configurations which cut across modules! If you really require a configuration file which cuts across modules, make sure to only include configuration items which are required by all modules.
- Implement a consistent error handling, for all possible configurations errors. Make sure that each module includes some self test, to check its configuration.
- In large systems additional configuration tools will be required to maintain the configuration (we wrote more code for such tools than for the actual core assets). Such tools will do static checking of values, provide versioning for configuration items and help SPL users to “fine-tune” their core assets during system integration.

4.7 Build time configuration

4.7.1 Forces and Structure

Instead of resolving variation points at runtime you can use your build environment to create module and products variants at build time. Usually the build environment will require a build command, which fully qualifies the requested module or product variant. Many different approaches to qualify a module or product are possible (feature lists, scripts, variant models).

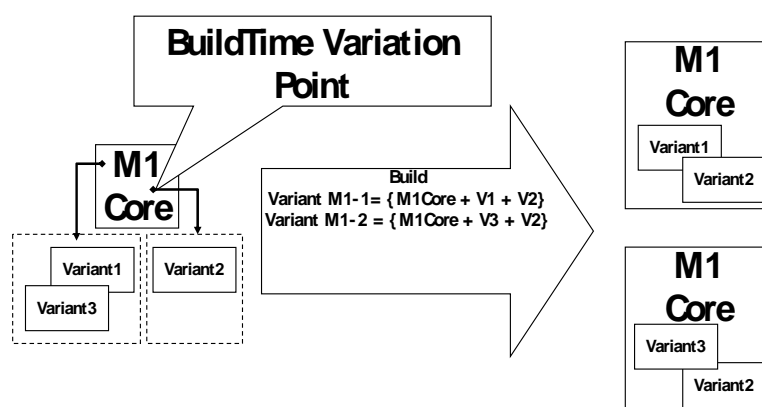


Figure 9: Creation of module variants at build time

4.7.2 Experience

- Build time variation points proved very powerful to create automatically a range of different installation CDs. But we also learned that the build scripts tend to get very complex. I strongly suggest you look out for tool (a variant configurator) before you start developing your own build scripts.
- Make sure that your variant configurator spits out the final package which you will install on the clients machine - automated everything! Maybe you would like to allow clients of the build environment to choose the variant, to reduce build times during development.
- Next time we start a project, we will start with the variant configurator. We hope that this will help us to reduce runtime configuration to a minimum.

4.8 Interface, implementation and factory

4.8.1 Forces and Structure

If you would like to allow users of the SPL to change the behaviour by offering alternative implementations you can supply a configurable factory, which will choose the configured implementation at runtime.

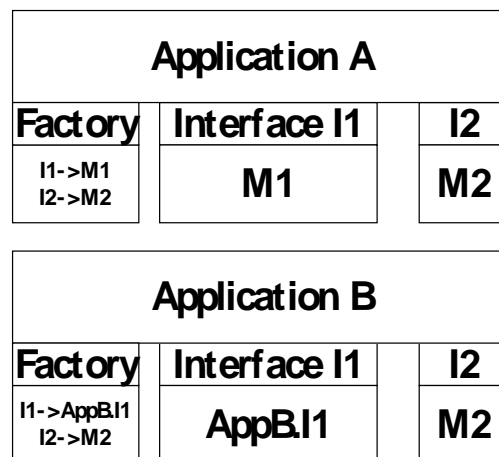


Figure 10: Usage of factory, to choose implementation at runtime

4.8.2 Experience

- Be careful with typed interfaces, because these will form a common module which has to be updated, every time an interface changes.

- Although all our modules are created through a configurable factory, we never used this functionality to create a product variant. We only used this variability to try out new algorithms or to allow researchers, to try out new features.

4.9 Configurable module factory

4.9.1 Forces and Structure

In many cases it might make sense to create a module with a different interface at runtime. This allows clients, through reflection, to code dynamically against the available features in an interface. In such a situation a configurable factory can create a module, by reading both the interface and the behavioural configuration.

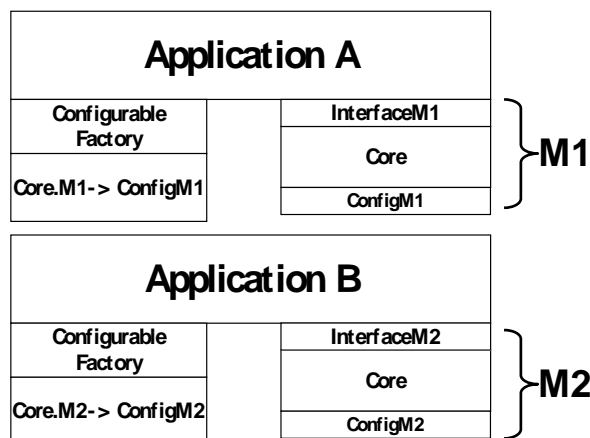


Figure 11: Using a configurable factory to change interface and behaviour

4.9.2 Experience

- In our context this proved to be a very powerful pattern to create modules which represents different hardware devices. The clients of these modules use reflection to query which features are available, and can create like that reflective presentations.
- On the other hand this pattern has a huge impact on memory consumption and start-up time, because the whole representation of the module needs to be created at runtime.
- Another nice side-effect is that some updates of the hardware devices can be represented purely as a new configuration.

4.10 Façade and session state

4.10.1 Forces and Structure

In most systems the complexity lies in the interaction between the different modules. To be able to make these interactions variable, they need to be localized in one module, so that this module can be replaced by a different implementation. In more advanced implementations such a module might delegate some responsibility to a configurable workflow engine.

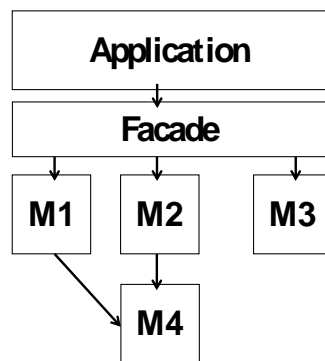


Figure 12: Usage of façade to simply interface to application

4.10.2 Experience

Originally, the façade was not considered part of the SPL because it is tied closely to the workflow and functionality of the application. But when we implemented the second application based on the SPL, it was decided to reuse the façade. This was mainly due to the complexity of the façade (usually all the code which does not belong to either the presentation or a specific module tends to end up in the façade). But this now makes it very difficult, to offer radically different workflows for both applications.

4.11 Framework

4.11.1 Forces and Structure

If the SPL defines a template architecture which is the same for all variants, it makes a lot of sense to encode this knowledge in a framework, which will call the modules in the right order.

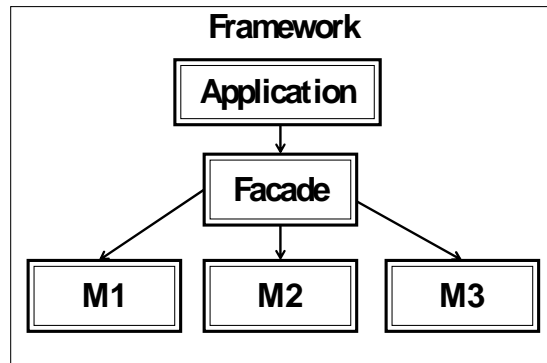


Figure 13: Framework which encapsulates the component interactions

4.11.2 Experience

- Today we can reuse 85-90% of the lines of code between products. This is mainly due to a large and powerful framework, which controls all the module interactions and provides the base abstractions.
- But the framework also ties all the modules together (at least in the runtime view). In our case this is not so bad because in most products we need all the modules anyway. We now started to split the framework apart, to allow clients which only would like to use one or two modules, to be able to re-use only part of the framework.

4.12 Summary

The following table summarizes our experience with the variability of the presented module structures, in the context of our fitting software product line.

Pattern	Benefit (High/Medium/Low)	Comment
Separation of presentation and business logic	High	Crucial if SPL needs to support different presentation layers.
Encapsulate knowledge about 3rd party modules	Medium	Allows locating code dependent on 3rd party in one module. This mainly improves portability of SPL.
Composition	Low	Composition of modules does not allow the level of granularity we required.
Layering	Low	Replacing a complete layer will require major refactoring of layers above.

Runtime configuration	High	Key pattern to realize module variants. Costs performance, but allows maintaining one code base. Requires major investment in powerful toolset to allow system integration to maintain configurations.
Build time configuration	High	Important pattern to create different deployments. Not used heavily, because we did not have major resource constraints in our platform.
Interface, Implementation, Factory	Low	This pattern was only used to allow try out new algorithms. Not used to generate new product variants, because we wanted to maintain one code base.
Configurable module factory	High	This pattern basically combines the power of runtime configuration with a factory. Allows to implement reflective interfaces.
Façade and session state	Medium	Important to decouple presentation layer from business logic. Represents a placeholder for code which does not belong to the presentation or to one module. Requires continuous refactoring!
Framework	Low	The framework basically represents the architecture. Sometimes we feel, that the framework constrains us to much.

4.13 References

- [Jacobson] Jacobson, I.; Griss, M.; & Jonsson, P. Software Reuse: Architecture, Process, and Organization for Business Success. Reading, MA: Addison-Wesley Longman, 1997.
- [Gamma] Gamma, E.; Helms, R.; Johnson, R.; & Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.
- [Anasta] Anastasopoulos, M. & Gacek, C. Implementing Product Line Variabilities (IESE-Report No. 089.00/E, V1.0). Kaiserslautern, Germany: Fraunhofer Institut Experimentelles Software Engineering, 2000.

- [Bosch 00] Jan Bosch, Design & Use of Software Architectures, Addison Wesley, 2000.
- [Bach] Felix Bachmann, Len Bass, Managing Variability in Software Architectures
- [Hillside] Gerard Meszaros, Jim Doble, A Pattern Language for Pattern Writing <http://hillside.net/patterns/writing/patternwritingpaper.htm>

5 A first step towards a framework for the automated analysis of feature models

David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés
University of Seville
{benavides, sergio, trinidad, aruiz}@tdg.lsi.us.es

Feature modelling is a common mechanism for variability management in the context of software product lines. After years of progress, the number of proposals to automatically analyse feature models is still modest and the data about the performance of the different solvers and logic representations used in such area are practically non-existent. Three of the most promising proposals for the automated analysis of feature models are based on the mapping of feature models into CSP, SAT and BDD solvers. In this paper we present a performance test between three off-the-shelf Java CSP, SAT and BDD solvers to analyse feature models which is a novel contribution. In addition, we conclude that the integration of such proposals in a framework will be a key challenge in the future.

5.1 Introduction

Feature Models (FMs) are one of the most common variability mechanisms. Good tool support is needed to debug, extract information and in summary analyze FMs in order to select them as a variability mechanism in a Software Product Line (SPL) approach. A FM represents all possible products of a SPL in a single model using features. FMs can be used in different stages of development such as requirements engineering [10], [11], architecture definition or code generation [1], [3]. A FM is a tree-like structure and consists of: i) relations between a parent feature and its child features. ii) cross-tree constraints that are typically inclusion or exclusion statements of the form “if feature F is included, then feature X must also be included (or excluded)”.

Automated analysis of FMs is an important challenge in SPL research [1], [2]. It can be performed using off-the-shelf solvers to automatically extract useful information of the SPL such as the number of possible combinations of features, all the configurations following a criteria, finding the minimum cost configuration, etc. Although there have been some promising proposals based in the representation of FMs as a Constraint Satisfaction Problem (CSP), boolean SAT-ifiability problem (SAT) and Binary Decision Diagrams (BDD) the performance of the solvers working with such representations is unknown for the SPL community.

In a previous work, we presented a performance comparison of two CSP java solvers analysing FMs [8]. In this paper we go further integrating different

solvers and logic representations. First we give a complete mapping for the three solvers (BDD, SAT and CSP) and then we present a performance comparison of them. To the best of our knowledge, this is the first test that measures the performance of solvers dealing with different logic representations of FMs.

The remainder of the paper is structured as follows: in Section 1.2 the automated analysis of FMs is outlined and details on how to translate a FM into a CSP, BDD and SAT are presented. Section 1.3 focuses on the results of the experiment. Finally we summarize our conclusions and describe our future work in Section 1.4.

5.2 Automated analysis of Feature Models

Once a FM is translated into a suitable representation it is possible to use off-the-shelf solvers to automatically perform a great variety of operations such as calculating the number of possible combinations of features, retrieving configurations following a criteria, finding the minimum cost configuration, etc [6].

There is a great variety of techniques and tools that can be used in the automated analysis of FMs. This paper focus on three well known problems in the area of automated reasoning: Constraint Satisfaction Problems (CSP), Boolean Satisfiability Problems (SAT) and Binary Decision Diagrams (BDD). All those representations have not been yet fully adopted in the automated analysis of FMs. In the next sections we will give a brief overview of each of them and finally we will introduce how translating a FM into a CSP, SAT and BDD.

5.2.1 Constraint Satisfaction Problem

Constraint Programming can be defined as the set of techniques such as algorithms or heuristics that deal with CSPs. A CSP consists on a set of variables, finite domains for those variables and a set of constraints restricting the values of the variables. A CSP is solved by finding states (values for variables) in which all constraints are satisfied. CSP solvers can deal with numerical values such as integer domains. The main ideas concerning the use of constraint programming on FM analysis were stated in [6], [7].

5.2.2 Boolean Satisfiability Problem (SAT)

A propositional formula is an expression consisting on a set of boolean variables (literals) connected by logic operators ($\neg, \vee, \wedge, \rightarrow, \leftrightarrow$). The propositional satisfiability problem (SAT) consists on deciding whether a given propositional formula is satisfiable, that is, if logical values can be assigned to its variables in a way that makes the formula true.

The problem is restricted by using the propositional formulas in conjunctive normal form (CNF), that is, propositional formulas composed by a conjunction of clauses in which each clause is a disjunction of literals (e.g. $((L1 \vee L2) \wedge (L3 \vee L4) \wedge (L5 \vee L6))$). Every propositional formula can be converted into an equivalent formula in CNF by using logical equivalences. The basic concepts about the using of SAT in the automated analysis of FMs were introduced in [1].

5.2.3 Binary Decision Diagrams (BDD)

A Binary Decision Diagram (BDD) is a data structure used to represent a boolean function. A BDD is a rooted, directed, acyclic graph composed by a group of decision nodes and two terminal nodes called 0-terminal and 1-terminal. Each node in the graph represents a variable in a boolean function and has two child nodes representing an assignment of the variable to 0 and 1. All paths from the root to the 1-terminal represents the variable assignments for which the represented boolean function is true meanwhile all paths to the 0-terminal represents the variable assignments for which the represented boolean function is false.

Although the size of BDDs can be reduced according to some established rules, the weakness of this kind of representation is the size of the data structure which may vary between a linear to an exponential range depending upon the ordering of the variables. Calculating the best variable ordering is an NP-hard problem. In the context of the automated analysis of FMs there are some tools that claim the internal use of BDDs [9].

5.2.4 Mapping

Rules for translating FMs to constraints are listed in Figure 1. In all cases the notation more common in the bibliography has been used. The final representation of the FM is the conjunction of the translated relations following the rules of Figure 14 plus an additional constraint selecting the root which is included in all products.

5.3 Experimental Results

The experiments focused on a performance comparison of three off-the-shelf Java solvers working with CSP, SAT and BDD in order to test how these representations can influence in the automatic analysis of FMs. The comparison results were obtained from the execution of a number of FMs mapped as CSP,

BDD and SAT in three of the most popular Java solvers within the research community: JaCoP² (CSP), JavaBDD³ (BDD) and Sat4j⁴ (SAT).



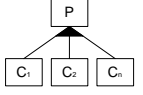
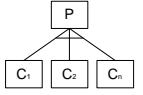
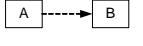
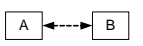
RELATION		CSP	BDD	SAT
MANDATORY		$P = C$	$P \leftrightarrow C$	$(\neg P \vee C) \wedge (\neg C \vee P)$
		$if (P = 0)$ $C = 0$	$C \rightarrow P$	$\neg C \vee P$
		$if (P > 0)$ $sum(C_1, C_2, \dots, C_n) in \{1..n\}$ $else$ $C_1 = 0, C_2 = 0, \dots, C_n = 0$	$P \leftrightarrow (C_1 \vee C_2 \vee \dots \vee C_n)$	$(\neg P \vee C_1 \vee C_2 \vee \dots \vee C_n) \wedge (\neg C_1 \vee P) \wedge$ $(\neg C_2 \vee P) \wedge \dots \wedge (\neg C_n \vee P)$
		$if (P > 0)$ $sum(C_1, C_2, \dots, C_n) in \{1..1\}$ $else$ $C_1 = 0, C_2 = 0, \dots, C_n = 0$	$(C_1 \leftrightarrow (\neg C_2 \wedge \dots \wedge \neg C_n \wedge P)) \wedge$ $(C_2 \leftrightarrow (\neg C_1 \wedge \dots \wedge \neg C_n \wedge P)) \wedge$ $(C_n \leftrightarrow (\neg C_1 \wedge \neg C_2 \wedge \dots \wedge \neg C_{n-1} \wedge P))$	$(C_1 \vee C_2 \vee \dots \vee C_n \vee \neg P) \wedge$ $(\neg C_1 \vee \neg C_2) \wedge \dots \wedge (\neg C_1 \vee \neg C_n) \wedge (\neg C_1 \vee P) \wedge$ $(\neg C_2 \vee \neg C_3) \wedge \dots \wedge (\neg C_2 \vee \neg C_n) \wedge (\neg C_2 \vee P) \wedge$ $(\neg C_{n-1} \vee \neg C_n) \wedge (\neg C_{n-1} \vee P) \wedge (\neg C_n \vee P)$
		$if (A > 0)$ $B > 0$	$A \rightarrow B$	$\neg A \vee B$
		$if (A > 0)$ $B = 0$	$\neg(A \wedge B)$	$\neg A \vee \neg B$

Figure 14: Mapping

5.3.1 The Experiment

We used four groups of 50 randomly generated FMs. Each group included FMs with a number of features in an specific range ([50-100],[100-150],[150-200] and [200-300]) with a double aim: test the performance of small, medium and large instances and working out averages from the results in order to avoid as much exogenous interferences as possible. After formulating each one as a CSP, BDD and SAT, we proceeded with the execution. Each FM was executed several times increasing the number of cross-tree constraints from one until the 25% of the number of the features in the FM in order to find out how dependencies influence in the performance. The dependencies were added randomly as well, but checking that the same feature can not appear in more than

² http://www.cs.lth.se/home/Radoslaw_Szymanek/

³ <http://javabdd.sourceforge.net>

⁴ <http://www.sat4j.org>

one cross-tree constraint and that a feature can not have a cross-tree constraint with any of its ancestors. Averages were obtained from all the FMs in each range with the same percentage of cross-tree constraints. Table 1 summarizes the characteristics of the experiments.

N. of Features	N. of instances	Dependencies
[50-100)	50	[0%-25%]
[100-150)	50	[0%-25%]
[150-200)	50	[0%-25%]
[200-300]	50	[0%-25%]

Table 1: Experiments

As exposed in [6], [7], there are some operations that can be performed. For our experiments we performed two operations: i) finding out if a model is satisfiable, that is, if it has at least one solution and ii) finding the total number of configurations of a given FM. The first one is the simplest operation while the second is the hardest one in terms of performance because it is necessary to work out the total number of possible combinations. The data extracted from the tests were:

- Average memory used by the logic representation of the FM (measured in Kilobytes).
- Average execution time to find one solution (measured in milliseconds).
- Total number of solutions, that is, the potential number of products represented in the FM.
- Average execution time to obtain the number of solutions (measured in milliseconds).

In order to evaluate the implementation, we measured its performance and effectiveness. We implemented the solution using Java 1.5.0_04. We ran our tests on a WINDOWS XP PROFESSIONAL SP2 machine equipped with a 3Ghz Intel Pentium IV microprocessor and 512 MB of DDR RAM memory.

5.3.2 The Results

The experimental comparison revealed some interesting results. The first evidence was that JavaBDD is on average 96% faster than JaCoP and 75% faster than Sat4j finding one solution. However, JavaBDD revealed a memory usage on average 928% higher than JaCoP and 1672% higher than Sat4j. On the other hand, although JaCoP and Sat4j showed a similar memory usage, SAT representation showed better results in both aspects, memory and especially in time. The performance of the solvers was similar in the four groups of experiments. Figure 15 and 16 presents the results for the group of FMs with a number of features between 100 and 150.

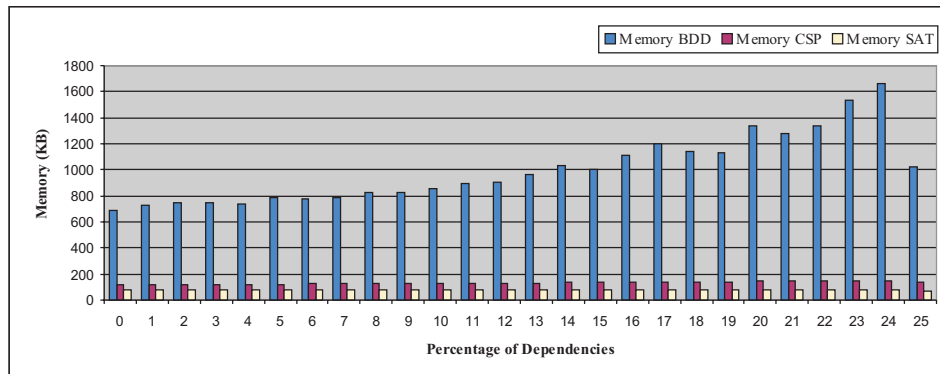


Figure 15: Memory Usage

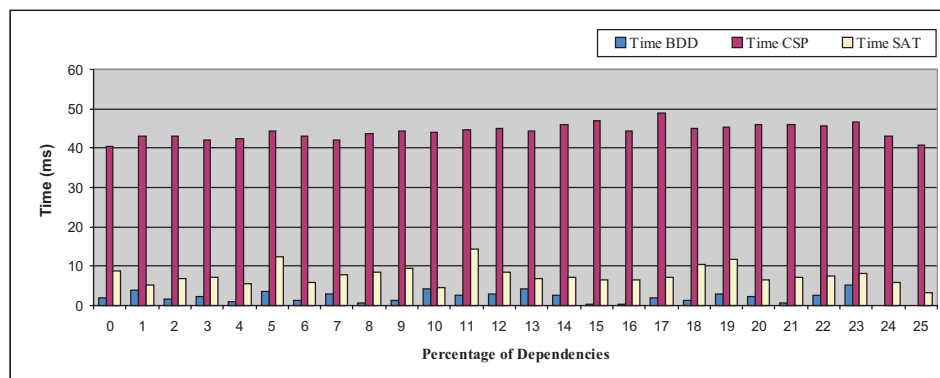


Figure 16: Average time to get one solution

In fact, Figure 15 can be confusing in the sense that in the worst case the memory usage is insignificant (in the order of 2 Mb) but this behavior seems to be exponential with the number of features and dependencies. For instance, in the range of (200-300) features, we found some cases where the memory used by the solver was around 300 Mb. We think that in bigger FMs (e.g. 1000 features) this can be even a bigger problem. What Figure 15 try to stress is the difference in the use of memory of the three solvers.

The results obtained from finding the total number of configurations of a given FM showed a great superiority of JavaBDD. While JaCoP and Sat4j were computationally incapable of performing that operation in a reasonable time in most of the cases, JavaBDD lasted 5312 ms to work out the 7.7×10^{34} solutions of the worst case.

Finally, we found some unexpected results or outliers in the data obtained from the experiments with JaCoP and JavaBDD. On the one hand, JaCoP showed in a

few consecutive executions a huge number of backtracks and consequently a great time penalty. On the other hand, JavaBDD revealed in a few experiments a huge memory usage which seemed to increase exponentially with the number of dependencies. We are investigating the possible causes of these behaviors [5].

5.3.3 Discussion

The great superiority of JavaBDD on finding the total number of solutions is because for calculating the number of solutions, in general, CSP and SAT solvers have to retrieve all the solutions (which is a #P-complete problem [12]) meanwhile BDD solvers use efficient graph algorithms to calculate the total number of solutions without the need of calculating all the solutions. The huge memory usage of BDD solvers depends on the variable ordering for representing the BDD. As stated earlier, the size of BDDs can be reduced with a good variable ordering, however, calculating the best variable ordering is a NP-hard problem.

To the best of our knowledge, there is only a proposal to include feature attributes in the automated analysis of feature models [6]. This proposal uses CSP solvers for that aim and we are not aware of any result where BDD or SAT solvers could be used to deal with feature attributes in order to maximize or minimize values.

As a result of the test, we claim that there is not an optimum representation for all the possible operations that can be performed on FMs. Therefore, we think that a framework for the automated analysis of feature models is needed. The framework will be designed to be open to other solvers where formal semantics of feature models will play a fundamental role [13]. The development of such a framework is the seed of our ongoing research [4].

5.4 Conclusion and Future Work

In this paper we integrated the use of different solvers in the automated analysis of FMs. We presented how to translate a feature model into a CSP, SAT and BDD and we performed a comparative test between three off-the-shelf Java solvers managing with each representation. The results showed that using BDDs for determining satisfiability in a FM is much faster than using SAT or CSP. On the other hand, FMs mapped as BDDs required a bigger memory usage in comparison with CSP and SAT. The test also revealed that while FMs mapped as SAT and CSP are computationally incapable of finding the total number of configurations in most of medium and large size FMs, FMs mapped as BDDs can get it in a very low time.

We think that there is not an optimum representation for all the possible operations that can be performed on FMs, therefore a framework for the automated analysis of feature models is needed.

5.5 Acknowledgements

We thank Don Batory and Jean-Christophe Trigaux for their helpful comments on an earlier draft of this paper.

5.6 References

- [1] D. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines Conference, LNCS 3714*, pages 7–20, 2005.
- [2] D. Batory, D. Benavides, and A. Ruiz-Cortés. Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, Conditionally accepted, 2006.
- [3] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Software Eng.*, 30(6):355–371, 2004.
- [4] D. Benavides, A. Durán, A. Ruiz-Cortés, P. Trinidad, and S. Segura. A framework for the automated manipulation of software product lines. *In preparation*.
- [5] D. Benavides, A. Ruiz-Cortés, B. Smith, Barry O’Sullivan, and P. Trinidad. Computational issues on the automated analyses of feature models using constraint programming. *International Journal of Software Engineering and Knowledge Engineering*, in preparation, 2006.
- [6] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005.
- [7] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Using constraint programming to reason on feature models. In *The Seventeenth International Conference on Software Engineering and Knowledge Engineering, SEKE 2005*, 2005.
- [8] D. Benavides, S. Segura, P. Trinidad, and A. Ruiz-Cortés. Using java csp solvers in the automated analyses of feature models. *LNCS*, to be assigned:to be assigned, 2006.

- [9] K. Czarnecki and P. Kim. Cardinality-based feature modeling and constraints: A progress report. In *Proceedings of the International Workshop on Software Factories At OOPSLA 2005*, 2005.
- [10] S. Jarzabek, Wai Chun Ong, and Hongyu Zhang. Handling variant requirements in domain modeling. *The Journal of Systems and Software*, 68(3):171–182, 2003.
- [11] M. Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the Second Software Product Line Conference (SPLC2)*, LNCS 2379, pages 176–187, San Diego, CA, 2002. Springer.
- [12] G. Pesant. Counting solutions of csps: A structural approach. In *IJCAI*, pages 260–265, 2005.
- [13] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, Minneapolis, Minnesota, USA, September 2006.

6 Version management tools as a basis for integrating Product Derivation and Software Product Families

Jilles van Gurp, Christian Prehofer
Nokia Research Center, Software and Application Technology Lab
{jilles.vangurp|christian.prehofer}@nokia.com

This paper considers tool support for variability management, with a particular focus on product derivation, as common in industrial software development. We show that tools for software product lines and product derivation have quite different approaches and data models. We argue that combining both can be very valuable for integration and consistency of data. In our approach, we illustrate how product derivation and variability management can be based on existing version management tools.

6.1 Introduction

The last ten years have seen an increasing interest in software product lines [1]. This has started from different approaches such as generative programming [2], feature-oriented design [4], and programming [6]. By now, there is a large amount of research on software product lines and variability management, which has led to several tools and various industrial experience, e.g. for mobile phones [3].

This paper considers tool support for variability management, with a particular focus on product derivation, as common in industrial software development. The purpose of product derivation in a product family is to construct a software product from a base platform consisting of architecture, design and reusable code. The product derivation process consists of selecting, pruning, extending and sometimes even modifying the product family assets. Additionally, in many companies which practice product family development, this is not a one time activity but a process that has an iterative nature:

- Usually, both product family and derived products evolve independently. They each have their own roadmaps and deadlines. However, it may be desirable to propagate changes from the product family to already derived products (e.g. bug fixes or new features) at certain points in time.
- As outlined in [9], repeated iterations are often required as understanding of the product requirements progresses.

We discuss in the following tool support for software product lines and product variation. For instance, common tools for product derivation are version management systems, such as Subversion [11][12]. We show that both areas have quite different approaches and data models. We argue that combining both

can be very valuable for integration and consistency of data. In our approach, we sketch how product derivation and variability management can be based on existing version management tools.

6.1.1 Tool support and product derivation

To support product derivation, various research papers have proposed using variability models and tools. Some of these approaches have been applied successfully in practice as well. The problem of describing and representing variability in these models is well covered in the research field. Essentially the tool support for this can be broken down into two categories (though some tools arguably fit in both categories):

- **Build oriented tools.** Tools that integrate into the build process. Examples of such tools include KOALA [10] and PROTEUS [7]. These tools enable the selection and configuration of components and generating glue code.
- **Documentation oriented tools.** Tools that focus on documenting the provided variability and that provide traceability of requirements and variation points to code. A good example of this is e.g. COVAMOF [9] and VARMOD [8]. These tools are primarily used to guide the (manual) process of product derivation.

Both categories of tools are very useful in their own right. Most of the above research has mostly been centered on the requirements and variability aspects. However, the process of product derivation (i.e. exploiting the provided variability in the software product family to create product variants) is only partially supported by tools.

Product derivation is about more than this. It includes:

- Selecting components. Reusing components provided by the software product family.
- Overriding components. Replacing provided components with alternative implementations (provided by the software product family or product specific).
- Modifying provided components. Sometimes product requirements conflict with product family requirements. Adapting such code in the derived product is a solution that despite its disadvantages is preferred in many companies.
- Providing new variants for existing variation points (e.g. implementing component interfaces)
- Adding product specific components and architecture.
- Configuring reused, modified and product specific components.

We observe that none of the variability management tools fully supports all of these activities. Secondly, we observe that the product derivation process, like the rest of the development process, is iterative. In other words, it is not a one time activity but a recurring activity during the evolution of a product.

6.2 Supporting product derivation with version management

The solution we propose involves exploiting functionality provided in common version management tools. The advantages of doing this are:

- Version management tools are used anyway in development organizations so it is a relatively easy transition for development teams to start using these tools for product derivation as well.
- Version management tools are the place to keep track of relations between artifacts (typically components) both in space (branches) as in time (revisions). Derived components can be seen as product specific branches of product family component.

Notice that version management works on any development artifacts (directory, a file, or an entire subsystem). We identify files or directories with components for simplicity.

In version management terms, product derivation is equivalent to creating a branch (or branches) of the main product platform and then committing product specific changes on these branches. However, this is not how version management tools are currently used in many product family using organizations. Instead products are usually created by copying (or generating configurations) artifacts from the product family and then adding product specific artifacts. This is especially problematic when product specific changes need to be made to the copied artifacts.

This is very similar to the notion of having multiple branches of the same code base in a version repository. A version management system supports this type of functionality by:

- Keeping track of the changes
- Allowing for changes to be merged from one branch to another

6.2.1 Subversion

There are roughly three generations of version management systems:

- Individual file based systems like RCS. Manage individual files. These systems are rarely used these days.
- Systems that can version groups of files (CVS and many commercial version management systems). While still popular, these systems lack many of the features that would enable using them for product derivation.
- Change-set oriented version management systems. These include systems like Subversion and GIT. The key difference is that rather than versioning individual files (like CVS) changes to the complete system are versioned with all its aspects including file system manipulation, symbolic link creation, meta-information modification and file changes etc. The delta between two revisions of the system in the version management system is called a change set.

Subversion [11][12] is a good example of a third-generation version management system. For the remainder of this paper we will assume Subversion or similarly capable, change set oriented version management system when we refer to version management. Our approach requires many of the features common in this new generation of version management tools. A few essential features are:

- **File system based rather than file based.** It can version all file system activities, including deletion, moving, linking and copying. For example, the history of a renamed file includes all commits before it was renamed; the rename; and all commits after it was renamed.
- **Copy by reference.** Copies are always by reference. The consequence of this is that a copy preserves version history and that making copies in the version repository is both fast and cheap in using server-side storage (unlike CVS where version history is not preserved and copying actually results in a full copy by value on the server).
- **Flexible repository layout.** Branching and tags are implemented as copies (by reference). Unlike many second generation versioning systems, branching and tagging are not special operations. For example, creating a new branch from trunk amounts to making a copy of a specific revision of the trunk directory to the branches directory. Subversion repositories (by convention, not by rule) contain directories with the names trunk, branches and tags. However, whether these directories are located directly under the root or deeper in the directory structure is up to the repository maintainer. In fact, using the subversion move operation it is trivial to change the directory layout if needed.
- **Properties.** Subversion supports annotations by associating properties (name value pairs) with any artifacts under version management. Of course changes to properties are also properly versioned (so they property manipulation is part of the version history).

6.2.2 Information models

The information model used by most variability tools is very different from that used by version management tools such as subversion. In the context of product derivation, both models are relevant. Therefore, we provide a brief outline of both in this section.

As outlined in the introduction the purpose of most variability tooling is to support product derivation either by documenting the variability in the software and/or by automating parts of the derivation process (e.g. component configurations; build configurations). Most approaches are centered on requirements, features and development artifacts. The information used by such tools consists of:

- Feature models. A common way to model variability is to construct feature diagrams with variant features. These models may be textual or graphical.
- Mapping of features to requirements.
- Mapping of variant features to design and implementation level artifacts. Especially the build oriented tools require this information in order to support the derivation process.

The information model of version management systems on the other hand is concerned with managing the changes of files and directories. The information it manages consists of

- A tree of directories, files and associated meta data properties. Usually the tree structure is derived from the logical architecture. For example, each directory represents a particular subsystem or module.
- In subversion, the meta data properties mentioned under 1 are used to represent various properties related to versioning (revision number; commit message; data and time) the file content (character used for new lines in text files; the mime-type of the file content; etc). Additionally files and directories may be annotated using custom properties. Subversion does not do anything with these properties (except for tracking changes to this data) but they may be used in scripts or custom applications that integrate the subversion programming API (bindings for C, python and Java exist).
- Storing change sets between revisions of the versioned tree. In subversion, each commit to the version repository is stored internally as a delta to the previous revision of the repository (and unlike CVS, the revision always refers to the entire contents of the repository instead of artifacts in the repository).

6.2.3 Integrating both information models

As can be seen from the description above, both information models have different purposes. Although there may be little overlap in both, consistency can be an issue. Furthermore, product derivation support that goes beyond the regular branching and merging functionality, requires integration of these information models.

There are two strategies for doing this:

- Integrate version management information in existing variability tooling (e.g. by storing subversion URLs and revision numbers of relevant artifacts in the repository).
- Store variability model information and mappings into the version management repository.

The latter strategy may be supported using subversions annotation feature. Since version management systems store development artifacts this concerns mostly storing the mapping of features and variability models to development artifacts. Using, for example, a "mandatory" property component directories corresponding to non optional features in the feature model could be marked as such. Similarly, a "depends-on" property might be used to indicate feature dependencies on other components in the repository. In this way, the information is directly stored with the corresponding code, which can help in avoiding potential inconsistencies when working with different data bases.

6.2.4 Using the information model

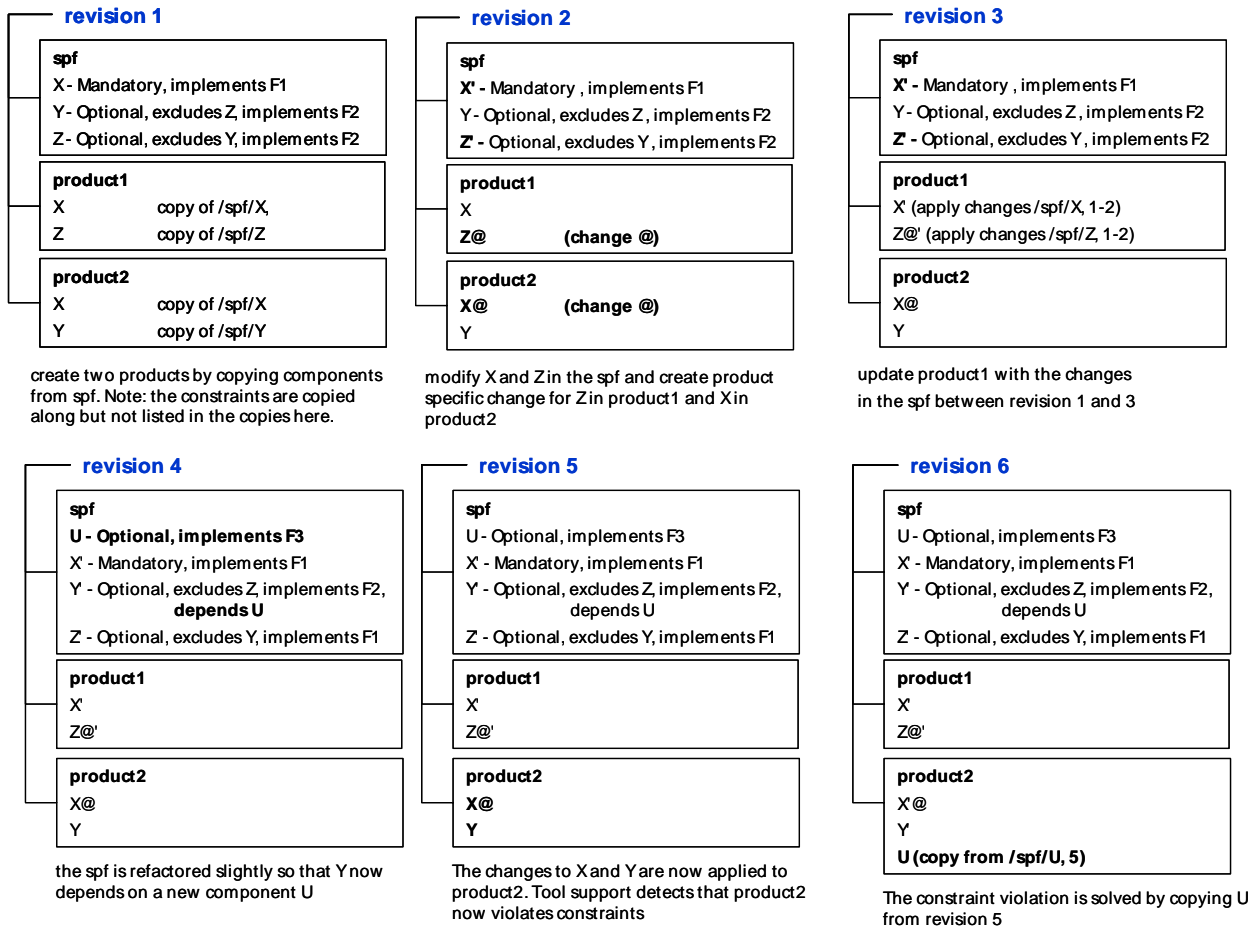


Figure 17: Example version management for product derivation and families

To illustrate how product derivation would work with such an integrated information model, we run through a small example scenario based on an imaginary SPF that we follow through a few revisions, as shown in Figure 17 above.

In revision 1, an SPF (software product family) directory is created and a few components (X, Y and Z) are added. Using subversion properties it is specified that X is a mandatory component and implements feature F1. Similarly, Y and Z are optional and are variants of the same feature F2. Since a product can use only one of the implementations of F2, both implementations Y and Z exclude each other.

Furthermore, revision 1 includes two product derivations in the form of two directories in the repository (product1 and product2). Copies from SPF have been made of X and Y for product1 and X and Z for product2. Although we do not show this explicitly, the properties on the SPF components are copied as well. This allows us to use a tool to validate the constraints (in this case there are no violations).

In revision 2, we do some maintenance on the SPF. This results in changes to /SPF/X and /SPF/Z. We indicate these changes using a '. Additionally product engineers make a product specific change to /product1/Z and /product2/X. These changes are indicated with a '@'.

In revision 3, product1 is updated with the changes made to the SPF in revision 2. /product1/Z now has both the product specific changes and the SPF changes. It might be possible that these changes are conflicting in which case the conflict would have to be resolved. It is worthwhile to point out that this conflict could have been identified (using a so-called dry-run for the merge of the changes on all the derived product components) already in revision 2 when the change was made to SPF/Z. In a real product family, the ability to analyze the impact of important changes on derived products is of course a very important feature any potential conflicts might result in these changes to be reconsidered or in some kind of upgrade strategy for the affected products.

In revision 4, some more refactoring is done on the SPF. A component U is added and some changes to Y result in a dependency between Y and U.

In revision 5, product2, which is still based on the revision1 of the SPF, is updated with the changes to X and Y. This results in a situation where constraints are violated.

Revision 6 resolves the constraint violation by copying U to product2.

The scenario above can be enhanced with tool support based on the information in the version repository. For example, constraints validation could be automated and run before each commit; as part of a nightly build or even integrated into the IDE. Similarly, impact analysis of changes on the SPF could be supported by trying to merge the changes to each of the derived products. These are just two simple but extremely useful ways to provide tool support using subversion.

6.3 Conclusions and future work

In this article we have outlined first ideas for complementing existing tools for product derivation based on software variability modeling with version management functionality in order to better support the derivation of software

products. Our approach is especially appropriate in situations where it may be expected that:

- Derived products may include modifications to the components that they are derived from.
- Changes made to the product family after the initial derivation takes place need to be propagated to derived products.

The main purpose of this position paper is to shape our ideas with respect to future work:

- Provide a more formal definition of the information models.
- Explore additional opportunities for automating product derivation steps.
- Build layer of tools on top of subversion and existing variability tools and validate concepts using a case study.
- Explore advantages of using distributed version management systems where change sets are pulled rather than pushed, a notion that shifts control from product family developers to product developers.

6.4 References

- [1] J Bosch, Design and use of software architectures: adopting and evolving a product-line approach, - 2000 - ACM Press/Addison-Wesley Publishing Co., New York, NY K.
- [2] Czarnecki and U. Eisenecker. Generative Programming: Methods, Tools, and Applications. Addison-Wesley, 2000.
- [3] Savolainen, J. Oliver, I. Mannion, M. Hailang Zuo, Transitioning from product line requirements to product line architecture, Computer Software and Applications Conference, COMPSAC 2005.
- [4] Kang, C. K., Lee, J., Donohoe, P., Feature-Oriented Software product line Engineering, 2002, IEEE Software, 19, 4, 58-65.
- [5] P. Sochos, I. Philippow, and M. Riebisch. Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In Object-Oriented and Internet-Based Technologies. 2004.
- [6] C. Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In ECOOP'97, 1997.

- [7] E. Tryggeseth, B. Gulla, R. Conradi, Modelling Systems with Variability using the PROTEUS Configuration Language, Lecture Notes In Computer Science Vol. 1005, Springer-Verlag, pp 216 - 240, 1995.
- [8] Pohl, K.; Böckle, G.; van der Linden, F.: Software Product Line Engineering – Foundations, Principles, and Techniques. Springer, Heidelberg 2005.
- [9] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, Jan Bosch: Modeling Dependencies in Product Families with COVAMOF. ECBS 2006: 299-307.
- [10] R. van Ommering, Building product populations with software components, proceedings of the 24rd International Conference on Software Engineering, pp. 255-265, 2002.
- [11] C. Michael Pilato, Ben Collins-Sussman, Brian W. Fitzpatrick, Version Control with Subversion, O'Reilly Media, 2004, available at svnbook.red-bean.com
- [12] The Subversion Project Home, <http://subversion.tigris.org>

7 Coherent Integration of Variability Mechanisms at the Requirements Elicitation and Analysis Levels

Nicolas Guelfi *, Gilles Perrouin**

*Laboratory for Advanced Software Systems, University of Luxembourg,
Luxembourg, nicolas.guelfi@uni.lu

**Computer Science Department, University of Namur,
Belgium, gilles.perrouin@uni.lu

Early phases of product line development can be separated in requirement elicitation and analysis. The former provides an abstract and informal description of the product line, while the latter provides a technical specification as precise and as complete as possible. The major problems we face are to define the content of each phase for optimal development cycle and to provide consistency between those phases. This paper aims at integrating product line variability mechanisms between requirements elicitation and analysis levels. First, we present a requirements elicitation template based on use case variants. Then, product analysis phase is done using a generative mechanism starting from the core analysis assets and specifying the variation covered by the use case variants. These mechanisms are coherently related by means of consistency rules and the same approach is employed to integrate feature models with the analysis phase. Finally elements for variability integration reasoning are derived on the basis of these rules.

7.1 Introduction

Variability [1] is a key notion for product line (PL) engineering and, as a result, various mechanisms have been proposed to cope with variability at all stages of product line development [2,3]. However, industrial research [4,5] has shown that variability mechanisms selection, although having a great impact in terms of flexibility and performance (at implementation stage), is the result of an often arbitrary choice. Furthermore, the same research states that the interaction between such mechanisms at various levels has not been sufficiently studied. The contribution of this paper addresses this lack at the requirements elicitation and analysis levels. This is done by examining the integration relationships between elicited requirements described in terms of a simple template we defined or depicted using a generic type [6] of feature diagrams [7] and the analysis model provided by the FIDI methodology [8]. This methodology addresses the development of distributed JAVA applications in a product line context. New PL members are derived by transformation from an existing structure, called architectural framework [8] (AF) which includes the core assets of the product line organized in a reference architecture.

Section 7.2 presents our requirements elicitation template defined to describe informally product lines and illustrates its usage on an example. Section 7.3 describes how these requirements can be precisely defined with the help of an analysis model and model transformation operations. Section 7.4 presents consistency rules integrating requirements elicitation and analysis variation mechanisms. It then discusses factors affecting variability mechanisms selection discovered while elaborating these rules. Section 7.5 presents some related work while Section 7.6 concludes this paper and outlines some future directions.

7.2 Requirements Elicitation Template

In this section, we present an informal yet structured template called REquirements Elicitation Template (REQET) that is fully described in [9]. Its main purpose is to describe PLs requirements at the very early stages of their development. The template comprises two sections: the *Domain Elicitation Table (DOMET)* and the *Use Case Elicitation Template (UCET)*. We illustrate this template with a “Hello World” PL whose members display a welcome message in several languages according to user choice and in some cases quote local writers.

7.2.1 Domain Elicitation Table

The role of the DOMET is to provide the necessary information to understand all the possible variants concerning data (domain concepts) amongst PL members. It takes the form of a data dictionary depicted using a tabular notation as shown on Table 1 below:

Concept Name	Var Type	Description	Dependencies
LuMessage	<i>Alt</i>	Contains the Luxembourgish welcome message: “Moien”.	Exclusive with respect to FrMessage
FrMessage	<i>Alt</i>	Contains the French welcome message: “Bonjour”	Exclusive with respect to LuMessage
LuxQuotes	<i>Opt</i>	List of quotes from famous Lxembourgish writers.	This concept requires LuMessage.
LangKey	<i>Mand</i>	Contains unique language key defined for each language, here “lu” or “fr”	Depends on the language defined in the PL, here “LuMessage” and “FrMessage”

Table 1: Domain Elicitation Table (DOMET)

Concept Name labels the concept via a unique identifier. **Var Type** column is filled with the following keywords:

- *Mand*: means that the concept is mandatory in the product line and hence must be present in all PL members,
- *Alt*: represents one of the alternative concepts that has to be chosen for a given PL member,
- *Opt*: represents an optional concept that may be omitted.

Description is an informal explanation of the concept purpose, while **Dependencies** column exposes any kind of relationship with other concept(s) such as generalization/specialization, related alternative or optional concepts etc. The nature and meaning of these dependencies is intentionally not specified to allow a flexible description. For example, in our "Hello World" PL, according to Table 1, we have the three following possibilities; one product in which only the Luxembourgish message is present, one in which only the French message is present and finally one in which, in addition to the Luxembourgish message, citations from Luxembourgish writers are available. It has to be noted that the domain table is partial; mandatory concept(s) that are not related to any variation description are not part of the DOMET and are simply written as plain text in the use case descriptions. Concepts of the DOMET are not represented in any graphical notation (UML class diagram...) for two reasons. The first one is, as these concepts form a subset of the whole PL ones they do not give the "big picture" of the PL. The last one is that there is no codification of the relationships expressed in the dependencies column so that syntax and semantics of the relationships within the diagram would have to be defined differently for each PL. In fact, we believe that the objective of the DOMET is to prepare a precise and complete analysis of the PL concepts by providing incomplete and informal information mainly dedicated to variations understanding.

7.2.2 Use Case Template

The second and last section of our template describes the behavior of PL members by means of use cases. We chose the widely accepted template given by Cockburn [10] and adapted it to support PL specificities according to [11,12]. The UCET is defined as follows:

- **UID**: An unique identifier for the use case, such as 'ucxx' where X is a digit e.g. 'uc01',
- **Use Case Name**: A short active verb phrase summarizing use case goal, e.g. "Display localized welcome Messages",
- **Var Type**: One of {'Mand', 'Alt', 'Opt'}. *Mand* represents mandatory behavior that all products should support, *Alt* represents a behavioral choice (other alternative use case are mentioned within brackets) and finally *Opt* represents a facultative behavior. In our PL, UC01 is mandatory,
- **Description**: A longer statement of use case goal if needed,

- **Actors:** Participating actors of the use case, e.g. "User"
- **Dependency:** Dependencies with other use cases (inclusion, extension...),
- **Preconditions:** Conditions that must hold prior to use case execution, e.g. "Application has been launched",
- **Postconditions:** Conditions that must hold after use case execution, e.g. "A message has been displayed to the user",
- **Main Scenario:** Standard and most frequent behavior decomposed in steps, e.g. "1. User enters (lu)[V1] language key, 2. Application displays welcome message (Moien)[V2] and may be a citation ("Wou d'Uelzecht durech d'Wisen zéit, Duerch d'Fielsen d'Sauer brécht Wou d'Rief laanscht d'Musel dofteg bléit, Den Himmel Wäin ons mëcht.", Michel Lentz, 1871⁵)[V3] to the user", parentheses here denote the scope of the variant and the value given represents the "default" value for data or behavior the variant is related to,
- **Alternative Scenario:** Alternative or less frequent behavior⁶, e.g. "2a. Application displays the national flag when the current date is the one of the selected country national celebration". Due to space reasons, additional concepts required by this alternative are not shown here
- **Non Functional:** Quality attributes required if any, e.g. "The message should be displayed in less than 1 second "
- **Variation Points Description:** Explain here the variation points (variants) introduced in the use case text via labels of the form: V1...Vn. Variants may concern data or behavior:
 - V1: Type:Alt, Concerns:Data, values={lu,fr},
 - V2: Type:Alt, Concerns:Data, values={if V1=lu then LuMessage...},
 - V3: Type:Opt Concerns:Data, values={if V1=lu then pick randomly a quote from the list defined in LuxQuotes}.

It is worth noticing that we do not consider alternative scenarios as a mean to document variability within the use cases; alternative scenarios have always to be supported by the product if the uses cases are mandatory or explicitly chosen. It is however possible to define variation points within an alternative scenario.

7.2.3 REQET Validation

The intent of the REQET is to focus on the variations the PL offers. Hence it is important to ensure a certain degree of coherence amongst the variation description within the template. For instance, the same concept (`LuMessage`) can be used obligatorily in a use case (consider a new mandatory use case that

⁵ Part of the Luxembourgish National Anthem. See http://www.gouvernement.lu/tout_savoir/histoire_monarchie/hymne_national/index.html for translations.

⁶ It has to be noted that in some cases it is interesting to distinguish alternatives from exceptions in order to handle the latter properly (e.g. Fault Tolerant systems). Domain specific extensions to the template may be developed for such a need.

would display user's name following the luxembourgish welcome message) and as alternative in an other one (e.g. in `uc01` above). In the DOMET of this such extended PL, we would have to choose between *Alt* and *Mand*; we suggest that it is the strongest usage (*Mand*) that should be reported in the DOMET hence providing useful information to product line developers. Intra-use case validation rules may also apply; if a concept seems mandatory within an optional use case description, this concept should be considered as optional for the PL. We can apply the same kind of validation rules to the behavioral descriptions within a use case. More on REQET validation is given in [9].

7.3 The FIDJI Approach to Product Line Analysis

In this section we sketch FIDJI prescriptions for performing the analysis of a PL. A more detailed description is given in [13].

FIDJI methodology builds on the natural synergy one can notice between frameworks and PLs [8]. It uses model transformation to instantiate and to derive architectural framework elements to produce the final product ones at all stages of its development. Hence models of the AF and those of a product use exactly the same notation and all the variability information is encoded using a composition of model transformation operations.

7.3.1 Analysis Model

The FIDJI analysis model [13] proposes the following notational elements:

- **Domain Concepts:** concepts of the domain are represented in a OMG's UML 2.0⁷ class diagram. Concepts are mapped into classes with attributes but without method. There are also regrouped in a table describing their purposes and relationships like a "standard" data dictionary,
- **Use Cases:** Use cases are following Cockburn's template [10] where we add OMG's OCL 2.0 expressions for pre, postconditions and scenarios steps. For each use case, contextual information for all OCL expressions is given via a *use case component* diagram (represented using UML 2.0 component notation as shown in Figure 18). A use case component describes the domain concepts handled by a particular use case and operations associated to it,
- **Operations:** Operations represent units of behavior that are composed to form the AF functionality. Operations descriptions are also following a template detailing domain concepts they handle and defining their behaviors in terms of OCL pre/postconditions.

⁷ For all OMG's specifications see <http://www.omg.org>

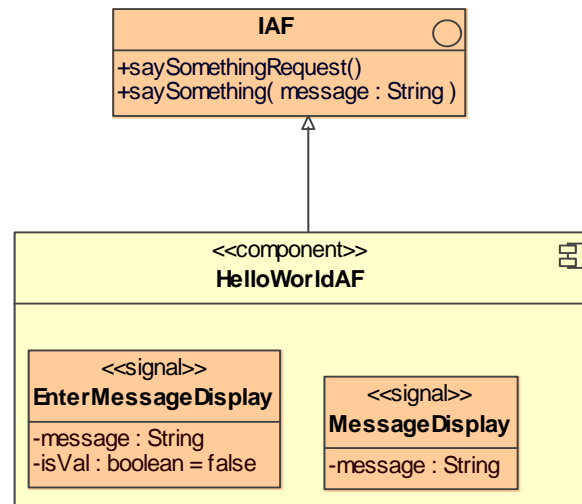


Figure 18: Use Case Component

As noted above we do not provide any dedicated notation related to variants within the AF analysis model, we rather use model transformation operations and product line boundaries constraints as explained below.

7.3.2 Implementing PL Variants with Model Transformations

In FIDJI, we use model transformations for two purposes: refinement and derivation. Refinements are vertical transformations that relate models at several levels of abstraction (analysis, design...) while derivations are horizontal transformations defining models of the product based on the one of the architectural framework at a given abstraction level. In the remaining of this Section we will focus on derivation as it represents the FIDJI way of implementing variability.

Although it is possible to describe derivations using standard model transformation languages constructs such as OMG's QVT, we found useful to define high level transformation operations that suits the structure of FIDJI models and eases the analyst's job. These operations cover addition, removal and update of model elements for every FIDJI model (analysis, design). These operations are declaratively defined in terms of OCL 2.0 pre/postconditions. For example, Figure 19 below gives the (incomplete) definition of an operation that creates a classifier in a model according to a reference to this model, the classifier name and its UML 2.0 classifier subtype.

```
addClassifier(m:Model  
  className:String,type:Type  
  post:m.ownedType ->  
    exists(c|oclIsTypeOf(type)and  
      c.name = className )
```

Figure 19: addClassifier Operation Specification

FIDJI model transformation operations are combined imperatively to form a transformation program, as shown in Figure 20 below. This program operates on the elements defined in Figure 18 by copying them as a whole in a new, initially blank, model and updates some elements by changing their names. The selection of variants for a particular product is then made by the analyst who combines transformation operations in the program.

Model transformation operations together with the imperative language which allows to compose them are currently been defined. We believe they represent a handy way to derive product models from architectural framework ones. Furthermore, since the variability is deduced from the program and not from the AF models a more flexible product line development is possible. Actually, products that do not belong explicitly to the original product line may be easily reached via a new transformation program. However, all transformation programs are not acceptable for two reasons. The first one is technical, when the analyst decides to remove or update one model element in his model, depending model elements may be impacted. If this impact is too big, he will have to change a lot of model elements. Thus, he may not benefit from reuse provided by the architectural framework and encounter difficulties to refine his models towards the final product implementation. The second reason is conceptual; although an architectural framework may technically allow some of its assets to be removed, doing so may cross the acceptable boundaries defined for the product line hence realizing a product that does not belong to the identified scope the PL addresses. For example although an AF may accommodate to use `FrMessage` and `LuxQuotes` in the same product (displaying a Luxembourgish citation after a French welcome message), this product would not have any sense with respect to the original PL and hence should be prohibited. The AF analysis model provides complete allowed and prohibited transformation programs as well as constraints concerning some AF analysis model elements. Programs and constraints are given in the Product Line Boundaries (PLB) package of the AF analysis model.

```
\* UCCHelloLang = {} * \
copyModel(UCCHelloWorldAF,
  UCCMyHelloLang );
updClassifier(UCCHelloLang,HelloWorldAF,C
  omponent ,
  EnterMessageDisplay,Signal,EnterLanguag
  eDisplay,Signal );
updClassifierProp(UCCHelloLang,HelloWorld
  AF,Component ,
  EnterMessageDisplay,Signal,message,Stri
  ng,language,String );
updateOpPar(UCCHelloLang,HelloWorldAF,Com
  ponent,saySomething,message,String,lang
  uage,String );
```

Figure 20: Transformation Program

7.4 Integration of Requirements Elicitation and Analysis via Consistency Rules

In this section, we present how the integration of our REQET with the FIDJI analysis model of an architectural framework can be achieved by means of consistency rules. We then show that the same approach can be employed to relate feature models with FIDJI analysis one and finally, we strive to synthesize the approach in order to draw some conclusions that may be applied with other variability mechanisms.

7.4.1 Requirement Elicitation Template & FIDJI Analysis

To REQET elements (concepts, use cases...) corresponds FIDJI analysis modeling elements or group of elements. Before we can reason on variability mechanisms integration, we need to state a few conditions.

Firstly, we assume a well-formed template description according to the hints given in Section 7.2. Secondly, we apply a kernel first approach [12] for the architectural framework analysis such that all the mandatory concepts and behaviors present in the REQET are mapped to the AF analysis model elements. Finally, it is also expected that model transformation operations for adding variable concepts and behaviors are also defined. Thus, in the PL exemplified in Section 7.2, the AF analysis models would include elements corresponding to the mandatory concept `LangKey` as well as the required model transformation operations for creating `FrMessage`, `LuMessage` and `LuxQuotes` analysis correspondences.

Our consistency rules are used to validate the transformation program yielding the final product line member. Since mandatory concepts and behaviors have already been processed by the kernel first approach, we focus on alternative and optional variants:

- **Rule 1: “For each set of related alternative elements (concepts or behaviors) in the REQET, the transformation program should result in exactly one group of elements that represents one REQET alternative”.**
- **Rule2: “To each optional REQET element, its corresponding model transformation operations can be found in the transformation program”.**

These two rules work well for individual elements but inter-elements dependencies have also to be taken into account. Some dependencies (such as generalization between two concepts) are handled when establishing the mapping between a PL REQET description and a FIDJI analysis model while for others such as mutually exclusive optional elements, additional consistency rules should be defined. As the REQET does not state on the nature of dependencies that have to be given (it is up to the REQET user to define them informally) it is not possible to define precise rules; such dependencies have to be handled on a per case basis.

7.4.2 Feature Models for Requirements Elicitation & FIDJI analysis

Two problems arise when trying to relate feature modeling approaches with “classical” software engineering approaches, namely “What is a Feature?” and “Which Feature Notation to choose?”.

Several definitions of the notion of feature have been given in the literature ranging from “a set of requirements” to “any characteristic of the system that is relevant for its stakeholders”. In the following, we will retain the definition given by Bosch: “a logical unit of behavior that is specified by a set of functional and quality requirements” [14]. Therefore features models represent a way of grouping requirements in a meaningful manner. These groupings may include diverse elements hardware or software, behavior and the concepts they are relying on. However we focus only on software related features.

The second problem relates to the various notations given for feature modeling from the original notation [7] to UML based ones and the semantics that is conveyed with them. In [6] Schobbens *et al* proposed a generic language, called Free Feature Diagram (FFD), able to support most of the approaches for feature modeling in a formal way. We will focus on the constructs given in this language in order for our rules to be easily translated in a specific notation. FFD proposes the following operators for representing variants:

- **And_n**: evaluates to true if all the n ($n \in \mathbb{N}$, $n > 0$ n represents the operator’s arity) subfeatures for a feature are present in the final product (mandatory features). **Or_n** and **xor_n** are defined the same way: for at least one (resp. exactly one) of their n subfeature(s) are (resp. is) selected in the final product.

E.g. xor_n has the same semantics as the alternative operator we have defined in our template,

- **Opt_n**: always evaluates to true, (opt semantics),
- **Vp(n..m)**: ($n, m \in \mathbb{N}, n, m > 0$ and $m > n$) evaluates to true if at least n and at most m of the sub-features are selected for the product.

Moreover, two binary operators are defined to express dependencies between features: *mutex* for mutually exclusive features (\mid) and *requires* (\Rightarrow) for compulsory dependencies between features.

To each feature, we associate an ordered set of transformation operations (i.e. a transformation sub-program) that defines the specification of the features in the analysis model based on the AF analysis model. As features do not separate behavior from data, each set of transformations may contain both transformation operations concerning data (FIDJI analysis concepts) and behaviors (FIDJI analysis operations). Associating features with model transformations is an approach that has also been promoted in the Fireworks project [15]. The semantics of the FFD operators enables the definition of the following consistency rules:

- **Rule 1:** For all the features that are operands of an and_n , their corresponding transformation subprograms must be part of the transformation program yielding the final product,
- **Rule 2:** For all the features that are operands of an or_n (resp. xor_n), at least one (resp. exactly one) of their corresponding transformation subprograms must be part of the transformation program yielding the final product,
- **Rule 3:** For all the features that are operands of an opt_n , their corresponding transformation subprograms can be part of the transformation program yielding the final product,
- **Rule 4:** For all the features that are operands of an $\text{vp}(n..m)$, at least n and at most m of their corresponding transformation subprograms must be part of the transformation program yielding the final product,
- **Rule 5:** For all the features that pertain to a *mutex* relationship, only one of their corresponding transformation subprograms can be present in the transformation program (xor_n semantics),
- **Rule 6:** For all the features that pertain to a *requires* relationship, all their corresponding transformation subprograms must be present in the program (and_n semantics),

We should also ensure that the PLB package of the AF analysis model is consistent with its related elicitation FFD:

- **Rule 7: Every allowed (resp. prohibited) transformation program in the PLB must (resp. cannot) correspond to a valid path in the feature diagram.**

7.4.3 Synthesis

In this paragraph we expose the factors that influence selection of variability mechanisms on the basis of the two integrations presented above.

Prior to reasoning about variability selection and integration it is necessary to relate elements present in the requirement elicitation documents or models with the analysis specification of core assets. In our examples, we notice that having the same separation between concepts and behavior, both at the requirements elicitation and analysis levels, eases integration. Feature models require additional description of the features to be able to relate elements.

One can also remark that the precision and richness of the variability mechanisms offered at the requirement elicitation level impact the soundness of the integration rules we can define between the two levels. In this case FFDs provide sound variation operators that make the relationships with transformation operations at the analysis level straightforward. More work is required in the case of the REQET since variability mechanisms differ between the domain table and the use case and have to be coherently managed first before establishing consistency rules.

Finally, if these two factors are successfully taken into account, it is possible to use the definition of consistency rules as a key criterion for selecting the variability mechanisms. For instance, FFDs seems to be suited to automation of core assets selection for a given product in a well delimited product line (as also illustrated by Czarnecki *et al.* [16]) whereas use case variants, emphasizing on PL members behavior, provide flexible variation mechanisms useful for identifying core assets in a coarse grained way in a PL under inception. In both cases, these rules suggest that a generative approach for PL analysis such as the one FIDJI proposes is compatible with these two popular variability mechanisms used at the requirements elicitation level.

7.5 Related Work

Savolainen *et al* [17] presents similar consistency rules helping management of constraints between elements at the requirements level (definition of the problem), the features level (abstract definition of the solution) and the assets level

(elements of the architecture). However the same variability mechanism (*à la* feature modeling) is adopted at these three levels.

Gomaa *et al.* [18] promotes the usage of OCL as a mean to define consistency checks between several variability mechanisms used in the multiple view approach that they propose. Nevertheless, they tend to have one-to-one mappings between PL elements while we consider one-to-many relationships in our rules.

Another promising approach to guarantee consistency and traceability of variability mechanisms amongst PL abstraction levels is proposed in [19]. It consists of a conceptual model that describes all the variability information related to a particular feature in a uniform way. However, to our knowledge, details of this model are not available yet.

7.6 Conclusion

In this paper we highlighted the importance of defining consistency rules to properly integrate variability mechanisms amongst product line requirement elicitation and analysis levels. In particular, we introduced a novel template for requirements elicitation based on use case variants. The integration of our generative analysis phase has been compared to a generic form of feature models (FFD). We finally exhibited traceability and precision as first-class factors to successful integration and variability selection.

The definition of consistency rules also served to identify some possible improvements: the REQET may need to have a global view on the variability mechanisms it offers. In order to be capable to define consistency rules we would need more information on the relationships between elements. To extend the FIDJI methodology at the requirements elicitation level with feature modeling, we would need to formalize consistency rules between FFD and FIDJI analysis level. Thus, an automatable approach to PL development based on such models could also be defined.

7.7 Acknowledgements

We thank our colleagues Barbara Gallina, Andreea Monnat, Cédric Pruski, Patrick Heymans and Jean Christophe Trigaux for their contributions or relevant comments related to this paper.

7.8 References

- [1] J. van Gurp; J. Bosch, and M. Svahnberg, On the Notion of Variability in Software Product Lines, *WICSA 2001*.
- [2] G. Halmans, and K. Pohl, Communicating the Variability of a Software-Product Family to Customers, *Software and Systems Modeling* **2**(1), 15-36, 2003.
- [3] M. Svahnberg; J. van Gurp, and J. Bosch, A taxonomy of variability realization techniques, *Software Practice and Experience*. **35**(8), 705—754, 2005.
- [4] J. Bosch; G. Florijn; D. Greefhorst; J. Kuusela; H. Obbink, and K. Pohl, Variability Issues in Software Product Lines, *PFE 4*, pp. 11—19, 2001.
- [5] S. Deelstra; M. Sinnema, and J. Bosch, Experiences in Software Product Families: Problems and Issues during Product Derivation, *SPLC3*, pp. 165—182, 2004.
- [6] P. Schobbens, P. Heymans, J. Trigaux, and Y. Bontemps, Feature Diagrams: A Survey and A Formal Semantics, *RE'06*, 2006
- [7] K. Kang; S. Cohen, and J. Hess, W. Novak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, 1990, Technical report n° CMU/SEI-90-TR-21, SEI, Carnegie Mellon University.
- [8] N. Guelfi, and G. Perrouin, Using Model Transformation and Architectural Frameworks to Support the Software Development Process: the FIDJI Approach, *MSEC 2004*, pp. 13-22.
- [9] B. Gallina, N. Guelfi, A. Monnat, G. Perrouin, A Template for Product Line Requirement Elicitation, Technical report n° TR-LASSY-06-08, *To be published*, University of Luxembourg.
- [10] A. Cockburn, *Writing Effective Use Cases* Addison-Wesley; 2000
- [11] A. Fantechi, S. Gnesi, G. Lami and E. Nesti, A Methodology for the Derivation and Verification of Use Cases for Product Lines, *SPLC3*, pp255-265, 2004.
- [12] H. Gomaa, *Designing Software Product Lines with UML*, Addison Wesley, 2004
- [13] G. Perrouin, Architecting Software Systems using Model Transformation and Architectural Frameworks, TR-LASSY-06-02, University of Luxembourg.

- [14] J. Bosch, *Design and Use of Software Architectures*, Addison Wesley, 2000.
- [15] M. Ryan, and P. Schobbens, Fireworks: A formal transformation-based model-driven approach to features in product lines, *Workshop on Software Variability Management for Product Derivation at SPLC 3*, 2004
- [16] K. Czarnecki, S. Helsen, and U. Eisenecker, Staged configuration through specialization and multilevel configuration of feature models, *Software Process: Improvement and Practice*, 2005, 10, 143-169
- [17] J. Savolainen; I. Oliver; M. Mannion, and H. Zuo, Transitioning from Product Line Requirements to Product Line Architecture: *COMPSAC'05*, 2005, 186-195
- [18] H. Gomaa, and, M.E. Shin, Multiple-View Meta-Modeling of Software Product Lines, *ICECCS '02*, 2002.
- [19] Kathrin Berg, Judith Bishop and Dirk Muthig, Tracing Software Product Line Variability - From Problem to Solution Space, *SAICSIT 2005*.

8 Product Line Architecture Variability Mechanisms

Steve Livengood

Software product lines have been adopted by many companies as a way to achieve significant improvements in quality, cost, and delivery time when a series of similar software products are to be developed. Effective development of software product lines requires the creation of a software architecture that covers the entire product line, providing variation points in the architecture to address the variation within the product line. This paper discusses one criterion for categorizing the types of architectural variation which can occur. This criterion may be used to assess how such variation mechanisms impact the development of further product-line assets and of the individual products.

8.1 Product Line Architectures

8.1.1 Background

The *Software Product Line* approach is an approach to developing and maintaining a set of common software artifacts (requirements, architecture, designs, components, plans, etc.) such that they can be used to instantiate many different software products (the *Product Line*) over time (refer to [SPL42] for further details). Each product uses the common artifacts—the *Core Assets*—in a prescribed way by configuring or adjusting them and assembling them according to the rules of a common, product-line-wide architecture. Reuse of the core assets between products is enabled and enforced through *preplanned* variation mechanisms.

There is a tendency in many software development organizations to treat the set of source code as the complete set of core assets. After all, don't measures of reuse usually focus on how well source code is reused product-to-product? But in reality, reuse of higher-level software artifacts is even more important—and is a key part of the software product line approach. In particular, it is necessary to have a *Product Line Architecture* (PLA) that defines the overall software structure of the entire product line. The PLA is the first point where the products' variation is represented in design and the first point where preplanned variation mechanisms must be introduced. The specific mechanisms by which the PLA addresses the variation is somewhat dependent on the architectural style and approach used, however it should be possible to categorize the types of mechanisms available and to describe their impacts on the development of further product-line assets and of the individual products. This paper provides some initial thoughts on one criterion that can be used to categorize these mechanisms.

8.1.2 Product Line Architecture vs. Product Architecture

Bass, et al [SAP03] define the architecture of a system as “the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.”. Unlike a traditional system’s architecture, a PLA must cover the entire product line, which includes identifying how the structure, the elements, the properties of the elements, and the relationships must be used, modified, and/or augmented based on the variation in the product line.

In contrast to the PLA that spans the entire product line, a *Product Architecture* (PA) describes the architecture of an individual product in the product line. A product’s architecture differs from its PLA by making variant-specific decisions based on variation points specified by the PLA. In other words, the PLA must address how the variability in the software requirements (functional and non-functional) is used to derive the various product architectures.

Figure 21 illustrates the relationship between the PLA and the individual PAs, and provides an example for the hypothetical product line of printers and multi-function peripherals.

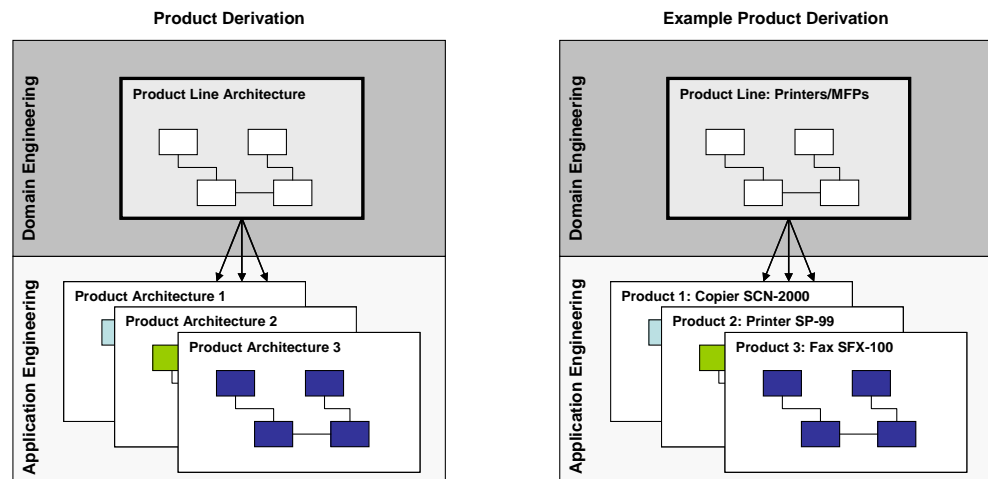


Figure 21: Product Architecture Derivation

8.1.3 Product Families

For complex product lines, PAs are not directly derived from the top-level PLA. Instead, the product line scope may include products that can be grouped into logical families based on common variation selections. To facilitate communication of the PLA and development of specific PAs, it is sometime convenient to describe the architectures of these families of products; these are called Product

Family Architectures (PFAs)⁸ to distinguish them from the full PLA. Each family resolves some of the variation present in the entire product line, but still has considerable variation. Figure 22 illustrates this situation by way of an example for printers and multi-function peripherals.

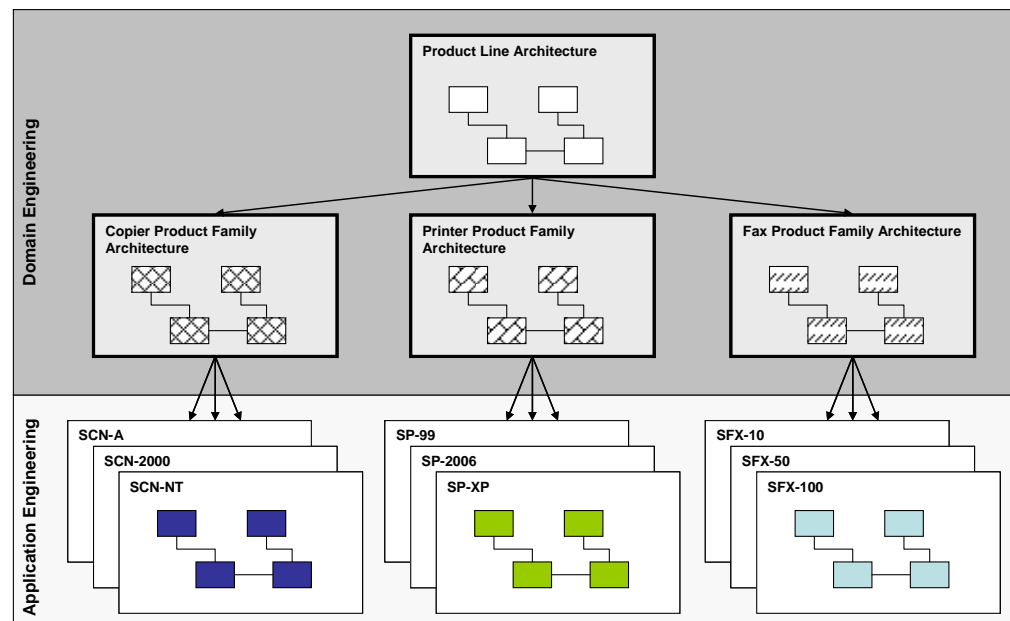


Figure 22: Product Family Example

In complex systems, PFAs may be further refined into their own derived PFAs. For example, a PFA describing printers might be refined into PFAs for network printers and personal printers, or a PFA describing copiers might be refined into PFAs for All-in-Ones and Multi-Function Peripherals.

The PLA, the PFAs, and the PAs represent a natural recursive approach to addressing the variation in the product line. This is a similar approach to the recursive breakdown of complex systems into subsystems, components, modules, etc. In these types of approaches, interpretation of certain terms depends on the point of view. Although this paper will specifically discuss addressing variation in the PLA, these same concepts should apply to addressing variation in the PFAs. Since either PFAs or PAs may be derived from the PLA, the term *derived architecture* will be used to refer to either type of architecture.

⁸ The term Product Family Architecture is sometimes used interchangeably with Product Line Architecture. In this paper, this term refers only to variants of the Product Line Architecture that address specific groups of products by specifying values for some of the variation in the product line.

8.2 The Unify or Divide Criterion

To be complete, the PLA must address all architecturally significant variation⁹. There are many mechanisms for addressing variation at an architectural level (see [MVSA01] for a discussion of architectural variability mechanisms). There has also been much work in describing and cataloging mechanisms and patterns (see [NVSPLO1] for a discussion of binding levels, binding times, and different patterns). Of the mechanisms and patterns that apply at an architectural level, we can ask a fundamental question: “When this mechanism is applied, does it cause the derived architectures to differ from each other?” The answer to this question determines whether the mechanism *unifies* or *divides* the derived architectures. More specifically, it:

Unifies the derived architectures if the derived architectures are identical to the PLA with regard to the particular variation.

Divides the derived architectures if the derived architectures are different to the PLA with regard to the particular variation (that is, the PLA describes a point of flexibility so that the derived architectures can differ).

Whether or not an architectural variation mechanism unifies or divides the derived architectures has an impact on the further development of core assets. If one derived architecture is different in some way from another derived architecture, there will be a point of difference in the assets associated with the two architectures, and hence a lack of commonality. Conversely, two derived architectures that are unified may be able to share some set of assets. It is believed that future work could develop a reasoning framework for choosing variability mechanisms by combining the unify or divide criterion with other criteria.

8.3 Examples

A set of examples may be useful to further understand the unify or divide criterion. The following sections discuss some of the major elements of architecture with regard to the unifying or dividing criterion. This is not intended to be a complete compendium of architectural variability mechanisms; it is simply illustrative of how the unify or divide criterion can be applied.

8.3.1 Component Relationships & Structure

An architecture breaks the system into components, allocates functionality to these components, and describes their expected behavior. A part of this de-

⁹ Depending on the architectural approach, certain variation may not be architecturally significant. For example, support for different hardware platforms may be required, but may be handled by mechanisms not of architectural interest (for example, within the operating system).

scription is the structural relationship between the components to satisfy the requirements of the product line. In many product lines it may be possible to create a component structure that does not vary across the product line. However, in most cases it will be necessary to address product line variability by providing for some variation in the component structure.¹⁰ This type of variation can be categorized by whether the PLA unifies the variation or allows the derived architectures to differ.

Unifying Variation

A particular variation is unified at the PLA level with respect to component structure if the derived architectures have exactly the same component structure regardless of which variant is present.

One mechanism that fits into this category is component abstraction (typically achieved via component replacement or plug-in technology). This allows the PLA to define a fixed component structure even when different components are used to achieve variation. In this mechanism, the PLA describes a fixed role in the architecture for a component. This role has functionality assigned to it, along with a description of how the actual components must vary based on the variation in the product line. Figure 23 illustrates this approach. In this example, assume there is product line variability that defines different types of image processing that needs to be performed. Regardless of this variation, the PLA indicates that the ImageProcess component will interact with the Scan and Print components.

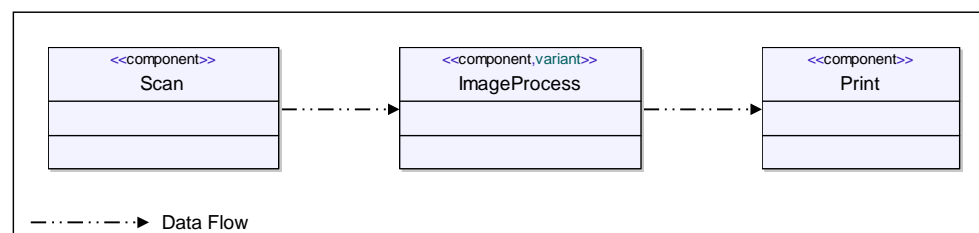


Figure 23: Unified Component Relationships

Dividing Variation

In some cases, the variation in the product line may require the component structure to differ between different products. In this case, the PLA introduces a variation point, dividing the derived architectures into groups based on which variation is used by the derived architectures.

¹⁰ Note that the concept of optional components is not specifically included in either of the two cases. Instead, it can be addressed in either way—modelling optional components by using a dummy component variant unifies the variation architecturally while modeling them by altering the structure when the component is not needed divides the variation.

One mechanism that fits into this category is the use of a configurable architecture. In this approach, components are connected in different ways to achieve different variations. The Unix-style pipe-and-filter architecture is a good example of this approach. A similar example is illustrated in Figure 24, where one variation requires a component structure that performs scaling before contrast reduction, while another requires contrast reduction before scaling. The result of these differences is a difference in the component relationships, but not in the actual components used. In this case, the PLA identifies two possible choices for the derived architectures¹¹.

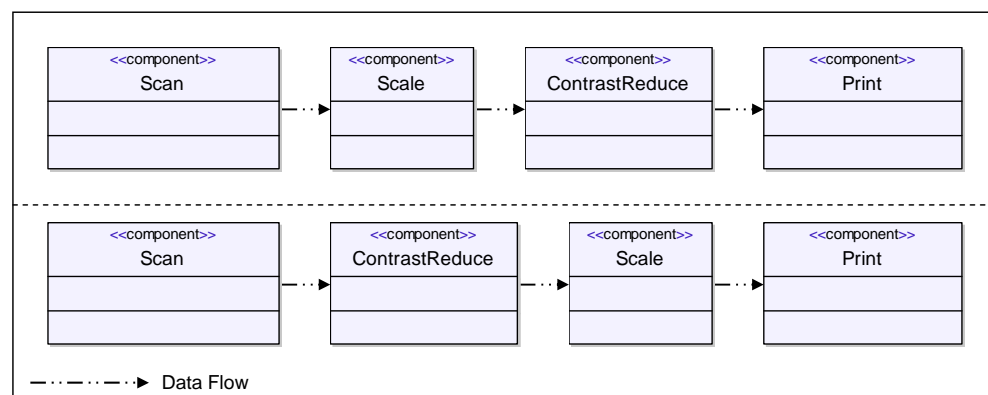


Figure 24:

Divided Component Relationships

An important point to note is that the rigor of the architectural definition does not impact the categorization of the way in which the variation is addressed. For example, the PLA may specifically describe all possible configurations based on the variability in the product line, or it may express rules about potential configurations based on characteristics of the products. The latter case is much less rigorous than the former, but is still an example of dividing variation.

8.3.2 Component Interfaces

With the component relationships identified, the next topic is the specification of the interfaces between the components. In most PLAs, separation of interface from implementation is important. For example, component replacement is enabled by being able to provide different implementations for the same interface, such that the actual component implementation may be varied without impact on clients. Also, configurable architectures are enabled by binding cli-

¹¹ To be effective, the PLA must describe the intended ways in which the PLA can be configured. Configurable architectures can achieve good flexibility, but when multiple areas of the architecture are configurable, the combinatorial aspects pose a difficult problem. Different aspects of variation may result in different configuration possibilities for different parts of the architecture, and describing all supported combinations is neither efficient nor desirable [and it doesn't readily provide the ability to derive new combinations as needed]. The introduction of PFAs at this point is a technique that can help express the intent of the PLA architect.

ents to (instances of) interfaces instead of actual components, which is possible only if the interface is separated from the implementation.

Returning again to the two possible options for addressing variability, remember that for each variation in the product line, the architect can unify the variation within the PLA, or he can instead divide the variation by introducing a variation point in the PLA.

Unifying Variation

One approach that can be taken by the PLA is to fully define an interface such that no interface difference is introduced in any derived architecture—the interface is designed to cover all possible variations. This unifies the variation with respect to interfaces in the PLA. But in order to unify the interface definition in the PLA, the PLA must fully define the semantics of the interface. Consider the interface defined in Figure 25.

```
public interface AddressBook {  
    AddressBookEntry getEntry(int index)  
        throws CommunicationError;  
    void addEntry(AddressBookEntry entry)  
        throws AddressBookReadOnlyError;  
}
```

Figure 25:

Unified Interface

This example illustrates the interface to an address book, which in some products is provided by a local address book, and in other products by interfacing with an LDAP server. In this case, the interface includes the functionality for both local address books and for LDAP address books. For example, the `AddressBookReadOnlyError` exception is defined for the case where LDAP address books are used and “add” functionality is not supported. Likewise, for getting entries, the `CommunicationError` exception is defined to handle the case where the LDAP server cannot be reached. In this example, the interface has been defined to cover all possible variation.

Dividing Variation

An alternate approach is for the PLA to defer some aspects of an interface to a derived architecture. Depending on the style of architecture, it may be possible to cover variations by using inheritance (to add or refine methods based on certain features, or to refine data types), or the definition may simply be different for different variations. Continuing the same example, two different versions of the `AddressBook` interface could be specified based on whether local address books or LDAP address books were being used. These are illustrated in Figure 26.

It bears repeating that the rigor of the architectural definition does not impact the categorization of the way in which the variation is addressed. For example, the PLA may specifically describe all interface variations based on the variability in the product line, or it may simply state that the definition of the interface is deferred to the derived architectures.

```
For Local address books:
public interface AddressBook {
    AddressBookEntry getEntry(int index)
    void addEntry(AddressBookEntry entry);
}

For LDAP address books:
public interface AddressBook {
    AddressBookEntry getEntry(int index)
        throws CommunicationError;
}
```

Figure 26:

Divided Interface

8.3.3 Component Identification

Continuing the AddressBook example, and independent of whether interfaces are divided or unified, we could imagine a single AddressBook component in the structural definition of the system, with the understanding that there would be at least two flavors: one for LDAP support and one for a local address book. In the PLA or in some derived architecture, the requirements on these two flavors will be completely defined: what interfaces they must implement, what form the interfaces must take, what their functional requirements are, and how they must interact with other components.

Unifying Variation

Figure 27 illustrates one way to represent this situation. In this case, there is a single component that must support multiple variations—the AddressBook component itself has variation (that is, it can be built or configured to cover all possible variants). The implication and probable result will be that one team will be given the charter to design a component that can be configured or built to meet the requirements for both variants.

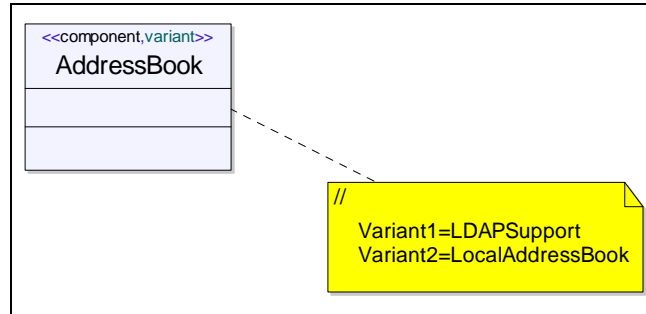


Figure 27: Unified Component Identification

Dividing Variation

Figure 28 illustrates another way to represent this situation. In this case, there are two components that must support the different requirements. The implication and probable result will be that two teams will be created, each working on a different component. Although this certainly doesn't preclude sharing of artifacts (for example, code) between these components, such sharing will likely not be as easy as the unified case, simply because of the division of work.

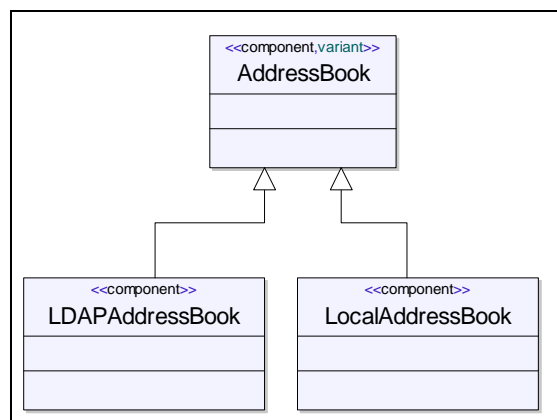


Figure 28: Divided Component Identification

8.4 Conclusions and Future Work

In most non-trivial systems, the PLA will use more than one type of variation mechanism, and in some cases, the same product-level variation may be addressed by multiple variation mechanisms. For example, to address the different type of address books that are present, the interfaces may be defined differently (dividing the interface), while component replacement is used to implement the local or LDAP functionality (unifying the structure, but dividing with respect to component identification). The structure of the architecture may

need to be altered to include additional components (dividing the structure), for example, to introduce a component to authenticate for LDAP. Having a framework to categorize the variation mechanisms in this case should provide to be useful.

This paper has presented one criterion for categorizing types of architectural variation. This criterion needs to be applied to various product line architectures to validate its usefulness. The hope is that this criterion can be used to describe the situations in which the various mechanisms are most appropriate, which will enable product line architects to make reasoned choices about which type of mechanism to use. In particular, it would be useful if the impact of these different types of mechanisms on the development of further assets can be generalized. For example, whether component identification is unified or divided, and if divided, how it is divided, can have a major impact on the success of the product line effort, and usually falls upon the architects to determine. Unifying many variations into one component can make the component difficult to develop and maintain; dividing can cause a duplication of effort when there is much commonality.

8.5 References & Further Reading

- [SAP03] Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice (2nd edition)*. Addison-Wesley 2003.
- [SPL42] Software Engineering Institute. A Framework for Software Product Line Practice Version 4.2, [see <http://www.sei.cmu.edu/productlines/framework.html>].
- [MVSA01] Bachmann, F. & Bass, L. "Managing Variability in Software Architectures. " *Symposium on Software Reusability*, Toronto, Canada, 18-20 May, 2001. [see <http://www.sei.cmu.edu/plp/variability.pdf>].
- [NVSPL01] Jilles Van Gorp , Jan Bosch , Mikael Svahnberg, "On the Notion of Variability in Software Product Lines", *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, p.45, August 28-31, 2001.

8.6 Author

Steve Livengood

s.livengood@samsung.com
s.livengood@thirdrailsoftware.com

Steve Livengood is the Chief Architect in the Advanced Printing Software Lab (APSL) of Samsung Electronics, Irvine, CA, USA, which is researching the institution of a software product line approach for the development of embedded printer/MFP software. His focus has been on the software architecture aspects of this project. He has over twenty years of experience in the development of embedded printer software and product line architectures. He received his BS in Computer Science from the University of Southern California, Los Angeles, CA and his MS in Computer Science from National Technological University, Fort Collins, CO.

9 Implementing a Variation Point: A Pattern Language

John M. Hunt and John D. McGregor
Department of Computer Science Clemson University Clemson,
South Carolina 29634-0974, USA

SPL research has provided guidance for variation points at the design level, but has provided only limited guidance on implementing a variation point. There are a number of decisions that are easily specified at the design level but require considerable effort to implement. These include cardinality, making a feature optional, and feature interaction. Our research attempts to provide specific guidance on these issues for a Java development environment enhanced with XVCL and AspectJ. Guidance is provided in the form of a pattern language on implementing variation points.

9.1 Introduction

Implementation has typically been discussed in terms of a list of general techniques, with a focus on the binding time related to the technique. Examples are Anastasopoulos and Gacek [6] and Svahnberg et al. [7]. This approach is more helpful to an architect trying to choose a development environment than a developer trying to live in one. The techniques are discussed in a language independent manner, which makes the advice more general, but at the cost of missing details important to the developer. The semantics of inheritance, for example, varies widely between languages. Finally, the issues involved with optional features and the effect of feature interaction are not addressed. Other available research looks at a particular technique, such as skeleton classes [2], aspects [8], inheritance [9], and other techniques. These papers typically ignore those types of product line variation which the studied technique has difficulty handling.

In contrast, our work is about implementing specific, though typical, variation points. These are implemented using a combination of techniques to best handle different parts of the problem. A limitation of this approach is the need to work in a particular development environment, in our case Java / J2ME, enhanced with some compatible tools such as XVCL and AspectJ.

As a result of this work we have noticed a number of variation point implementation issues that do not seem to be generally discussed in the literature. These include:

9.1.1 Single vs. Multi Value variation points

If only a single variation can be selected, techniques that naturally form exclusive choices have an implementation advantage. For example, inheritance has often been recommended as a mechanism to implement variant behavior. A sub-class is defined for each variation choice. To access the behavior at runtime an object is instantiated. As Java's new operator allows only a single class to be specified, this provides a natural implementation of a single value variation point. However, this exclusive behavior of the new operator becomes a drawback if the variation point allows multiple values. Using inheritance for the multi value case requires a collection of objects, one for each variant selected, which in turn requires the implementation to manage the collection [Lee2004]. Other techniques differ in the advantages / disadvantages that they provide in these two cases. For example, aspects are designed to act independently of each other. This is a disadvantage in the single value case, as they will not naturally exclude each other, but an advantage in the multi value case where more than one aspect can be active without additional management.

9.1.2 Optional Features

If an optional feature is omitted from a product, the code related to this feature should be cleanly and completely omitted from the product. For many techniques, recommended in the literature to implement variants, this is not possible. For example, if inheritance is used to implement variants the code can not be completely omitted. We will either create a subclass with empty methods or we will not create the object. If we do not create the object we will need runtime code to check for a null reference. This problem is generally shared by component approaches [10].

9.1.3 Feature Interactions

Feature interaction involves two features call them - F1, F2. Assume F1 is a feature in the product. If F2 is added to the product, then the behavior of the system may be different then it would have been with only F1. For example, in our research, which uses arcade style games for its domain, we have a feature called "practice mode". If practice mode is included in the product and the user selects to be in practice mode then the scoring is disabled and an unlimited number of tries at scoring is allowed.

Feature interactions have not generally been discussed as part of variation point implementation; however, they constitute one of the ways that products vary as features are selected for a product variant. If we accept that the variation point is the place in the product where we see the consequences of feature

choices then feature interaction should be part of variation point design and implementation.

Feature interaction has been understood in a variety of ways in different fields of software engineering, primarily in telecommunications [11]. Our approach to feature interactions is based on work by Lee and Kang [10], and takes advantage of a design pattern that they present. We extend their work several ways: Our work is at the variation point / implementation level rather than a feature / design level. As a result we are working with the problems of handling feature variants and optional features while handling feature interactions. Finally, Lee and Kang limit their discussion to component approaches while we are looking at how aspects and frames can supplement components.

We want to leave the code related to handling feature interaction out of the product if the particular set of features chosen does not interact with each other. It is preferable to keep the code handling the interaction separate from code implementing the feature. A given feature may be affected by more than one other feature, and thus take part in multiple feature interactions. Thus, if we have not kept the code to handle the interactions separate, a new version of each feature will be needed for each possible combination of feature interactions.

Given these variation point implementation issues we wish to provide advice to the core asset developer. It is desirable to make the advice applicable to many situations, yet detailed enough to provide useful guidance. The format we use to provide this advice is a pattern language. "A pattern language is a structured method of describing good design practices within a particular domain. Pattern languages are used to formalize decision-making values whose effectiveness becomes obvious with experience but that are difficult to document and pass on to novices. They are also effective tools in structuring knowledge and understanding of fundamentally complex systems without forcing oversimplification" [12]

9.2 A Pattern Language for Variation Points

Name: Implement a Variation Point

Pre-Condition: In the course of implementing a variant SPL feature we realize the need to provide for variation at a particular point in the code; that is, we recognize the need to implement a variation point.

Problem: SPL research has provided guidance for variation points at the design level, but has provided only limited guidance on implementing a variation point. Much of this guidance has been deliberately language independent to

make it more general. This results in guidance that does not deal with the language limitations with which a developer must cope.

A variation point must provide the opportunity to select a particular value from a set of variants. Depending on the problem addressed, and thus the design for solving the problem, the selection might be limited to a single value or open to include multiple values in the same product. This difference is simply expressed at the design level using a feature model. However, implementations for single and multi value selection will differ from each other. Advice on these differences has not typically been provided. At the design level features may be optional, meaning that they may be omitted from the product. However, the problems of omitting code related to an optional feature completely from an implementation are rarely addressed. Feature interaction is not normally discussed in relation to variation points. However, feature interaction requires that an implementation modifies its behavior depending upon the features selected for a given product. Since we would like to limit the effect of the feature selection on an implementation to the variation points, we must cope with feature interaction as part of the variation point implementation.

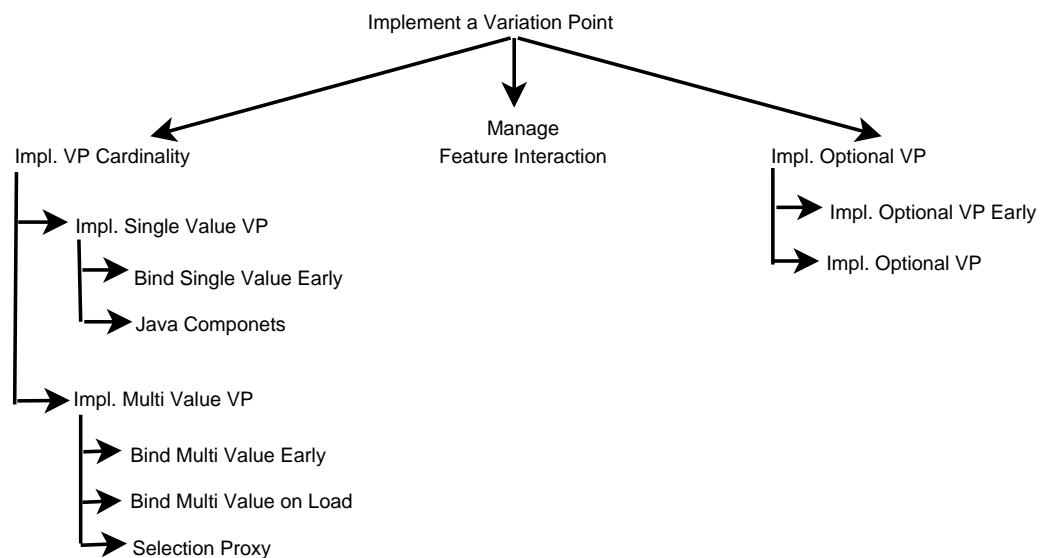


Figure 29: Overview of the Implement a Variation Point Pattern Language

Implementation advice for developers should address the consequences of these three types of design choices - cardinality, optionalness, and feature interaction.

Therefore:

Solution: Our pattern language provides advice for handling these three different types of design decisions that affect a variation point. They are:

- Implement Variation Point Cardinality
- Implement Optional Variation Point
- Manage Feature Interactions

Either Implement *Variation Point Cardinality* or *Implement Optional Variation Point* may be done first. *Manage Feature Interactions* should be done after the other two. All variation points must address cardinality; hence, will use *Implement Variation Point Cardinality*. Variation points related to mandatory feature will not need to *Implement Optional Variation Point*. Not all features interact with other features, so not all variation points will need to *Manage Feature Interactions*.

Constraints: This pattern language provides advice specific to implementing a variation point using Java. Two additional tools will be discussed that are compatible with Java - XVCL, a language independent general purpose tool that we use as a pre-processor, and AspectJ, which provides an aspect extension to the Java language. While we use XVCL, the techniques we discuss should be applicable to any pre-processor.

9.3 Name: Implement Variation Point Cardinality

Problem: For some features only one value at a time may be selected. For example, a car must have exactly one transmission; however this may be either a manual or an automatic transmission. In other cases selecting multiple values at the same time is possible. A car's sound system may (or may not) have a radio, a CD player and a tape deck. The implementation of the variation point must be able to support the cardinality of features specified in the design.

Little distinction has been made between implementing a single value variation point as opposed to a multi value variation point. An example of the differences can be seen in the widely discussed approach of using inheritance to implement the variants. To isolate the client code from the choice of a particular variant we can write the client code in terms of the specification provided by the parent class and implement the variations as sub-classes. For the single selection case we hold a reference of the parent type and instantiate one of the sub-classes. This naturally gives us a single choice. However, in the case of a multi selection variation we need to create an object for each selected variant. We will need to hold these in a collection which will then have to be managed. Yet, the additional problems of managing the collection are typically ignored when inheritance is recommended as a way of implementing variants.

A mechanism that can manage multiple selected values can also manage a single value. Thus, it could be argued that only the multi value case needs to be considered. However, using a multi value mechanism for a variation point that holds only a single value imposes unnecessary costs. These costs occur both in

terms of program execution, requiring additional memory and CPU cycles; and cognitive costs to the programmer, due to more complex code. Separate advice specifically for the single value variation point will be provided.

Therefore:

Solution: Based on the product design the developer first notes if a single or multi value variation point is called for. If the design specifies that only a single variant may be selected for a product then *Implement a Single Value Variation Point*. If the design specifies that multiple variants may be selected for a product then *Implement a Multi Value Variation Point*.

Name: Implement a Single Value Variation Point

Pre-Condition: The design allows for several possible choices at this site in the code and only one of these choices may be included in a particular product.

Problem: The design allows for several possible choices at this variation point; however, we wish to isolate the client code outside the variation point from changing when different variants are selected. Aspects are not recommended for this case because they do not naturally exclude each other.

Therefore:

Solution: If an early binding time is acceptable and the variant requires small amounts of code that appears in only a few places, consider *Bind Single Value Early*.

Both inheritance and interface implementation can be used to build features and to isolate which variant of a feature is selected from client code. The approach to prefer is largely dependent on the architectural context of the product. Both inheritance and a Java language interface provide a natural way to limit selection to a single variant. For inheritance, this is done by having the sub-class that provides the variant extends a parent class used by the client code. For Java language interface, this is done by having the class that provides the variant implement the interface. In this case, the client holds a reference to the interface and instantiates one of the classes that implement the interface. Chose an interface definition that will suffice for all the variants; which should simplify having each variant work with a parent class or interface. This should be possible in the single selection case as the variants should be substitutes for each other. Both inheritance and Java language interface select a variant by instantiating an object. This means that the binding time may be as late as run-time, if the selection is from a closed set of variants known at construction time. It is possible in Java, but not J2ME, to extend these to an open set of variants by using reflection to instantiate classes not known at construction time. Selection and instantiation of variants may be centralized using the *GOF Factory Pattern*, and made available throughout the code using parameterization.

9.4 Name: Implement a Multi Value Variation Point

Pre-Condition: The design allows for several possible choices at this site in the code and more than one selection may be included in the product.

Problem: The design allows for several possible choices at this variation point; however, we wish to isolate the client code outside the variation point from changing when different variants are added to the product. The implementation must support multiple selections being included and active in the product at the same time.

Therefore:

Solution:

If an early binding time is acceptable and the variant requires small amounts of code that appears in only a few places *Bind Multi Value Early*.

If a later binding time is desired *Bind Multi Value on Load* may work if there are appropriate places to hook on aspects.

If places to hook aspects can not be found, or if runtime binding is required, then a *Selection Proxy* may be used.

9.5 Name: Implement Optional Variation Point

Pre-Condition: *The variation point belongs to an optional feature.*

Problem: If the feature has been omitted from a particular product variant we should completely omit the related code at the variation point without leaving a trace of the feature. Note that the binding time of including or omitting a feature may be different than the binding time of selecting a variant. For example, we can decide that a particular product variant will include a scoreboard at construction time, but allow the user to select which type of scoreboard will be displayed at runtime.

Therefore:

Solution: If we know whether to include the feature at construction time *Implement Optional Variation Point Early*.

If there is a suitable program construct, such as a call that can be used as a hook for an aspect, and a binding time of program load is acceptable to decide if the feature is included *Implement Optional Variation Point*.

9.6 Name: Manage Feature Interactions

Pre-Condition: The variants for a variation point have been implemented. The developer for the variation point code has realized that this variation point will affect another existing variation point.

Problem: Feature interaction involves two features - F1, F2. If F2 is added to the product, then the behavior of F1 changes depending on the state of F2. An example from the PPL is to consider the scoreboard as a feature whose behavior is changed if the practice mode is added to the product.

When implementing feature interaction, we want to leave the code related to the feature interaction out of the product if the feature that introduces the interaction is left out of the product. Preferably we would like to implement the interaction in such a way that a feature is not aware of interacting features. A given feature may be affected by more than one feature, and thus take part in multiple feature interactions.

Therefore:

Solution: Use aspects to implement the feature interaction without affecting the feature implementation code. This is a great advantage in avoiding code tangling between features and one that only aspects were able to deliver. Multiple feature interactions can be implemented by defining aspects for each, which has the nice property that they can be unaware of each other.

Place the feature interaction code in its own Java package. Write an AspectJ point cut to add the needed calls to each of the variation points in the feature affected by the feature interaction, the point cut file is added to the package containing the feature interaction code. Produce a jar file containing the package. Including the jar on the command line running the program, along with the AspectJ runtime jar file, will cause the feature to install itself into the product at program load time. Note that the feature code does not need to be modified to add the feature interaction code.

This pattern can be extended to handle multiple feature interactions. We assume that the different interactions involving the feature are independent of each other, but each feature needs to have an opportunity to check the state it is concerned with before the default behavior for the variation point executes. This functionality is provided by adding a declare precedence keyword to the point cut file of the modifying feature.

Consequences:

To control the addition of the feature the code must be organized into its own package and the build process must produce a jar file.

Features may be added without modifying existing code.

Features may be omitted as late as program load time.

The code must have program features, such as methods, that can serve as hooks for the point cuts. Aspects are not designed to insert code at arbitrary points in the program.

9.7 Name: Implement Optional Variation Point Early

Problem: We wish to completely omit an optional feature from the code without leaving a trace. We know by construction time if the feature is being included in the product.

Therefore:

Solution: Use XVCL to cleanly omit the variation point's code while inserting the variation point at an arbitrary place in the code. Begin by creating two frames; one containing the code related to the feature at this variation point, the other frame has no content. An adapt command using a XVCL variable is placed at the variation point. Set the variable to choose between the frames with the feature related code and the empty frame. Select the empty frame to omit the feature code from this variation point. An empty frame has the effect of inserting a single space into the code which will not cause any Java code generation.

Consequences:

The choice to include the feature must be made during the build phase, prior to compilation.

Sample Code:

```
<set var="SCOREBOARDTYPE" value="Digital"/>
```

```
...
```

```
<adapt x-frame=""?@SCOREBOARDTYPE?BV.XVCL"/>
```

Related Patterns:

Providing multiple frames and related variables extends this pattern to *Bind Single Value Early*.

Include an empty frame as a choice to allow a multi selection feature to be omitted from the product, thus combining this pattern with *Implement Optional Variation Point Early*.

9.8 Name: Bind Single Value Early

Problem: We wish to select from among different variants to implement a variation point. Selections are mutually exclusive.

Therefore:

Solution: Create a frame to hold the code for each variant. An adapt command using a XVCL variable is placed at the variation point. Setting the variable chooses a particular variant by including its frame.

Consequences:

The choice of which variant to include must be made at during the build phase, prior to compilation.

Sample Code:

```
<set var="SCOREBOARDTYPE" value="Digital"/>
...
<adapt x-frame="'?@SCOREBOARDTYPE?BV.XVCL'"/>
```

Related Patterns:

Include an empty frame as a choice to handle optional features as described by *Implement Optional Variation Point Early*.

To allow multiple selections to be chosen at the same time *Bind Multi Value Early*.

9.9 Name: Bind Multi Value Early

Problem: We wish to select one or more variants from a set of possible variants.

Therefore:

Solution: Create a frame to hold the code for each variant. The name of the frame for each variant to be included is set into a XVCL multi var. XVCL multi var's allow a list of values. An XVCL while command feeds each of the selections from a multi var into an adapt command.

Consequences:

The choice of which variant to include must be made at during the build phase, prior to compilation.

Efficient code is automatically produced when only one variant is selected.

Sample Code:

```
<set-multi var= "ServiceChoices"  
  value="SVPauseUn,SVSaveLoad,..."/>  
.  
.  
.  
<while using-items-in="ServiceChoices">  
  
    <adapt x-frame=" ?@ServiceChoices?.XVCL"/>  
  
</while>
```

Related Patterns:

To insure an exclusive selection from a set use Bind Single Value Early.

Including an empty frame as a choice extends this pattern to handle optional features as described by Implement Optional Variation Point Early.

9.10 Name: Implement Optional Variation Point

Problem: We wish to completely omit an optional feature from the code without leaving a trace.

Therefore:

Solution: Place the optional feature code in its own Java package. Write an AspectJ point cut to add the needed calls to each of the variation points affected by adding the feature, the point cut file is added to the feature's package. Produce a jar file containing the package. Including the jar on the command line running the program, along with the AspectJ runtime jar file, will cause the feature to install itself into the product at program load time.

Consequences:

To control the addition of the feature the code must be organized into its own package and the build process must produce a jar file.

Features may be added with modifying existing code.

Features may be omitted as late as program load time.

The base code must have program features, such as methods, that can serve as hooks for the point cuts.

Related Patterns:

Breaking code for a feature into multiple packages and jar files results in Bind Multi Value on Load.

9.11 Name: Bind Multi Value on Load

Problem: We wish to select one or more variants from a set of possible variants.

Therefore:

Solution: Use aspects to implement the multi-selection case, each aspect can be added independently without concern for being an exclusive choice. Aspects install themselves, this is particularly useful if multiple points in the code are affected. Place code for each separately selectable feature in its own Java package. Write an AspectJ point cut to add the needed calls to each of the variation points affected by adding the feature, the point cut file is added to the feature's package. Produce a jar file for each selection containing the package. Include the jar on the command line when running the program to install the feature at program load time.

Consequences:

To control the addition of the feature the code must be organized into its own package and the build process must produce a jar file.

Selections may be added with modifying existing code.

Selections may be omitted as late as program load time.

The base code must have program features, such as methods, that can serve as hooks for the point cuts. Aspects are not designed to insert code at arbitrary points in the program.

The case where only one selection is made is handled in an efficiently without additional coding.

Related Patterns:

If there is a single selection to be either included or omitted this approach becomes *Implement Optional Variation Point*.

9.12 Name: Selection Proxy

Problem: We wish to select one or more variants from a set of possible variants. Binding time for our selection may be as late as runtime.

Therefore:

Solution: Separate the different selections into separate classes to make the inclusion of different choices modular. These classes should be accessed in a consistent way. Either sub-class off of a common parent class, which will allow the parent class to act as a common interface; or have each of the classes im-

plement the same Java language interface. In either case, instantiate one object for each choice resulting in a set of objects at runtime.

The set is hidden from the client code making it appear as a single object. The details of the accessing the collection are hidden from the client code using *Selection Proxy*. While the variants in the multi selection case are related to each other, they may not be substitutes for each other as they are in the single selection case. This may make it difficult to provide a single interface definition for all of the variants. If a single interface definition is not naturally available it may complicate both the inheritance and interface implementations.

Create a proxy object that holds and controls access to the set of variant objects. This could also be considered an example of a facade pattern; however, the interface provided in this application has an essentially one-to-one mapping with the contained variant objects, rather than providing the simplified convenience interface normally associated with a facade. The client makes a call on the proxy object which searches the collection for the appropriate variant, passes the call onto it, and returns the results back to the caller.

Consequences:

This approach has a number of runtime inefficiencies including: The creation of multiple objects (proxy, collection, and one for each variant). An extra call for each access, one on the proxy object, one on the variant object. The time required to search the collection for the right variant. The need to marshal and un-marshal parameters and return values from a general form for the proxy to a specific form for the variant.

This pattern causes additional classes of runtime errors, such as not finding a requested variant in the collection.

Additional code is required to avoid the runtime costs of the collection when only a single variant of the set is included in the product.

Variants can be added even during runtime from either a closed set known at build time or by using reflection may add new variants after build time.

9.13 References

- [1] M. Anastasopoulos, "Personalized cost-efficient product line implementation," Fraunhofer Institute, Tech. Rep. IESE-Report 056.04/E, 2004.
- [2] T. J. Brown, I. Spence, P. Kilpatrick, and D. Crookes, "Adaptable components for software product line engineering," in Software Product Line Conference, ser. LNCS, G. J. C. ed., Ed., vol. 2379. Springer, 2002, pp. 154–175.

- [3] D. Muthig and T. Patzke, "Generic implementation of product line components," in Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World, ser. LNCS, no. 2591. Springer-Verlag, 2002, pp. 313 – 329.
- [4] D. Muthig and C. Atkinson, "Model-driven product line architectures," in Software Product Line Conference, ser. LNCS, no. 2379. Springer, 2002, pp. 110–129.
- [5] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard, "Separating features in source code: An exploratory study," in 23rd International Conference on Software Engineering (ICSE'01), 2001.
- [6] M. Anastasopoulos and C. Gacek, "Implementing product line variabilities," in Symposium on Software Reusability, 2001, pp. 109 –117.
- [7] M. Svahnberg, J. van Gurp, and J. Bosch, "A taxonomy of variability realization techniques," *Software Practice & Experience*, vol. 35, no. 8, pp. 705 – 754, July 2005.
- [8] M. Griss, "Implementing product-line features with component reuse," in International Conference on Software Reuse. Springer-Verlag, June 2000, pp. 137–152.
- [9] B. Keepence and M. Mannion, "Using patterns to model variability in product families," *IEEE Software*, vol. 16, no. 4, pp. 102–108, July/August 1999.
- [10] K. Lee and K. C. Kang, "Feature dependency analysis for product line component design," in International Conference on Software Reuse, ser. LNCS, no. 3107, 2004, pp. 69–85.
- [11] A. Mehta and G. T. Heineman, "Evolving legacy system features into fine-grained components," in ICSE '02: Proceedings of the 24th International Conference on Software Engineering. New York, NY, USA: ACM Press, 2002, pp. 417–427.
- [12] Wikipedia, "Pattern language," Wikipedia, The Free Encyclopedia, May 2006.

10 On the Architectural Relevance of Variability Mechanisms in Product Family Engineering

Arnd Schnieders

In this paper we derive the need to represent variability mechanisms already in product family architectures based on an analysis of the main stakeholder use cases in product family development. We also sketch an approach for integrating variability mechanisms into product family architectures for process oriented systems.

10.1 Introduction

Variability mechanisms are techniques for adapting software development artifacts. Up to now, in product family engineering research, mainly implementing variability mechanisms like e.g. conditional compilation [SvB00] or parameterization [CIN01] for realizing the adaptation support in generic (i.e. adaptable) implementation artifacts, have been in the center of interest. However, we think that variability mechanisms should already be regarded during the development of the product family architecture, since they can have a considerable impact on the properties of the product family later. Moreover, variability mechanisms are also required for providing an efficient adaptation support for the product family architecture itself. In this paper based on an analysis of the main stakeholder use cases in product family development we derive the need to represent variability mechanisms already in the product family architecture. We also outline an approach for variability mechanism centric product family architecture modeling for process oriented systems.

Section 10.2 gives an overview of the main product family engineering stakeholders and their use cases, which are analyzed regarding the role of variability mechanisms in section 10.3. Based on this analysis we derive the need for product family architecture variability mechanisms in section 10.4. In section 10.5 we outline a corresponding approach for process oriented systems. In section 10.6 we give a short summary of the paper contents as well as an outlook to future research.

10.2 Stakeholder Groups Related to Product Family Engineering

This section gives an overview of the main product family engineering stakeholders and their use cases, which provide the basis for analyzing the architectural relevance of variability mechanisms in section 10.3. In the following we roughly distinguish between the roles customer, business manager, architect, and engineer as done by most of the product family methods [Mat04].

The *customer* selects a product configuration mainly based on the generic requirement artifacts.

The responsibilities of the *business manager* comprise making strategic and economic decisions related to the product family development. These particularly refer to product family scoping and pricing issues. The scope of the product family first of all depends on the features, which are desired for the product family members due to market strategy considerations. Second, it is guided by economic considerations. The product family members have to have sufficient commonalities in order for the product family approach to pay off ("economies of scope" [Boh96]). We assume that the price for an application orients itself towards the features requested by the customer. These may comprise features, which have not been regarded by the product family infrastructure so far.

The central task of the *architect* is to design a product family architecture that balances the potentially conflicting system requirements as good as possible according to their priorities. The system requirements comprise on one hand the customer specific requirements regarding the application(s) under development, which can refer for example to the performance, security, availability, or usability of the systems. Additionally, there are so called development requirements [Bos00], which comprise requirements regarding the maintainability, minimal complexity, reusability, flexibility, evolvability, testability, and time to market of the development artifacts. Development requirements also refer to the system architecture itself, i.e. the system architecture itself shall be maintainable, flexible, etc. For a family oriented software development especially the development requirements flexibility, evolvability, and maintainability have a high priority [PBL05]. Additionally, product family engineering artifacts shall support inter product variability. Support for inter product variability comprises easy application specific configuration of generic development artifacts and the optimal re-use of common system parts within the product family.

Since the architect knows the structure of the product family much better than the business manager, he has also a deeper insight into the amount of common and variable parts the product family will contain and the additional effort required for realizing the product family variability. Thus, he can support the business manager in setting the scope of the product family. The architect can also support the pricing activity by giving estimations concerning the costs for the generation of customer specific products based on the product family infrastructure.

The *engineer* creates generic implementation artifacts according to the product family architecture. During application development, the configured product family architecture serves for the engineer as the blueprint for configuring the generic implementation artifacts. Even though in the ideal case the configuration of the design and implementation artifacts can be performed automatically, normally these manual configurations and adaptations are still required.

The use case diagram in Figure 30 gives an overview of the use cases, which have been discussed in this section.

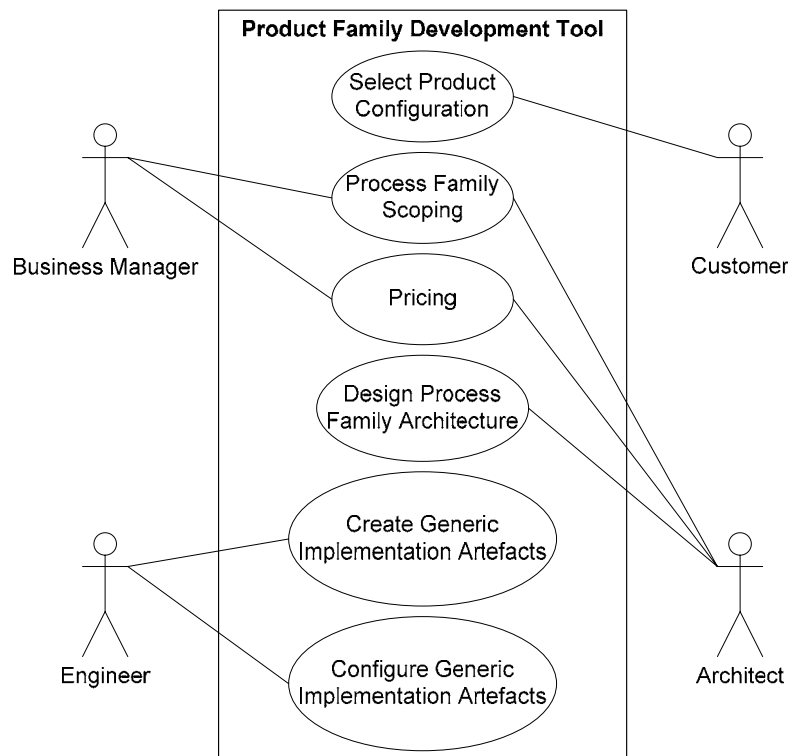


Figure 30: Use Cases of Product Family Architecture Stakeholders

10.3 Variability Mechanisms in Product Family Engineering

This section points out the architectural relevance of variability mechanisms based on a systematic analysis of the use cases shown in Figure 30.

Select Product Configuration. The customer selects a product configuration based on the generic requirements artifacts.

Product Family Scoping/Pricing. As already discussed in section 10.2 the answer to the question whether a product family pays off or not depends considerably on the amount of commonality between the members of the product family as well as on the ability to exploit them. The basis for the optimal exploitation of commonalities between product family members has to be set with the product family architecture. The degree up to which the commonalities between the product family members can be exploited also depends on the variability mechanisms selected for realizing the product family variability. Consider for example the variability mechanism *polymorphism*, which allows for the in-

vocation of varying subsystem implementations via an invariant subsystem interface [Gom05, GoW04, GBS01]. The reuse properties of this variability mechanism shall be illustrated by a UML Activity diagram [OMG05], where the CallBehaviorAction in Figure 31 represents the subsystem interface for two alternative subsystem implementations represented by the Activities in Figure 32.

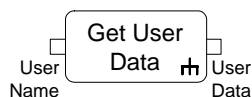


Figure 31: Common Interface for Variant Implementations

In the example shown in Figure 32 the encapsulation of varying implementations leads to redundancy of the Actions *Check Data Validity* and *Correct Data* in the implementations *Get User Data - No Account* and *Get User Data - Account* resulting in a suboptimal exploitation of commonality. The portion of reused elements could be increased by only replacing and omitting the variable elements. Such a fine-grained kind of reuse is for example possible by applying the variability mechanism *conditional compilation* [FLR02].

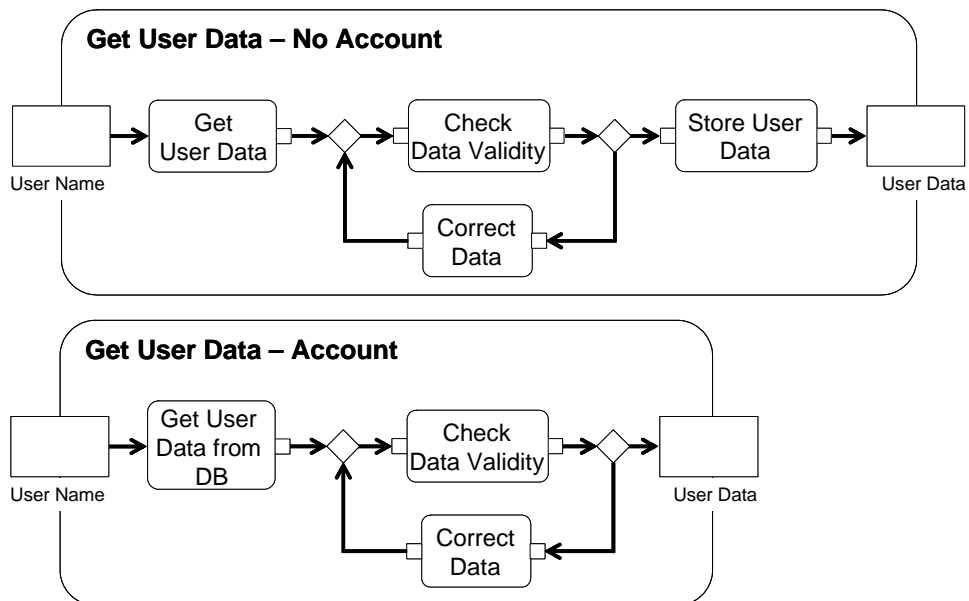


Figure 32: Variant Implementations of Common Interface

The potential economic benefit of a product family development is mainly capitalized on during application engineering. The effort for configuring the generic product family artifacts according to the customer requirements and thus the overall effort for application engineering also depends on the variability mechanisms used for realizing the product family variability. *Parameterization* for example is a very efficient way for the easy configuration of generic product fam-

ily artifacts. However, the prerequisite for *parameterization* is that all possible variants are provided in the subsystem's code [SvB00, BaB01, Gom05]. In very large and complex systems, the utilization of *parameterization* can support the efficiency of the configuration process. The customization of an SAP-ERP system for example would be even more costly if it would not be realized primarily by changing the values of configuration parameters. Variability mechanisms like *user exits* or *modifications* are therefore preferably avoided [BHM01].

Concerning the case that some customer requirements haven't been regarded by the product family so far, the variability mechanisms can also have a considerable impact on the effort required for extending the product family by the new customer requirements. *Parameterization* for example doesn't support well the extension of the product family by additional features. The problem is that extensions require adaptations in any place in the system, where the optional or alternative behavior is performed in dependency on the parameter value. In this respect *polymorphism* is a much better choice, since the variability is encapsulated in interfaces and new features can be added more easily by adding additional implementations as long as they stick to the invariant interface.

Design Product Family Architecture. The main task of the *architect* is to transfer the product family requirements into a corresponding product family architecture. The choice of a variability mechanism can thereby impact the customer specific, as well as development related properties of the implementation artifacts described by the product family architecture.

Parameterization for example on one hand allows for reconfigurations after the system has already been installed. On the other hand, this requires that also the code of the deactivated variants remains in the implementation artifacts after configuration. This leads to higher memory requirements as well as to a possible runtime performance decrease due to the need for selecting the right variant at runtime. Thus, for implementing product lines of performance and memory critical systems the variability mechanism *conditional compilation* would be a better choice in this respect. However, from the perspective of maintainability and evolvability *conditional compilation* isn't the best choice, since the variability is spreaded over the code. *Conditional compilation* also doesn't allow for such a flexible configuration as *parameterization*, since it requires a recompilation of the system in case of reconfiguration.

In this section only the impact of the variability mechanisms on the properties of the implementation artifacts has been discussed so far. However, the product family architecture itself represents an important generic development artifact that also requires variability mechanisms for realizing the product family variability according to the product family architecture development requirements. The corresponding discussion will be postponed to section 10.5 after a set of variability mechanisms for product family architectures (for process oriented systems) has been identified.

Create Generic Implementation Artifacts. Based on the product family architecture the engineer develops the generic implementation artifacts. Therefore, he needs to know the variable parts of the system as well as the technique for realizing the variability appropriately.

Configure Generic Implementation Artifacts. Ideally, the generic software development artifacts can be configured fully automatically according to the application requirements. However, this requires that full configuration automation is provided by respective tool chains and that at the time of the application development all application specific artifact parts have already been created. Otherwise, manual configuration effort is unavoidable. Therefore, the engineer requires information concerning the place at which the generic implementation artifacts have to be adapted (i.e. the variation points), how the application specific adaptations (i.e. the variants) look like, and which kinds of adaptations have to be done. Does he only have to set a parameter value? Or are there any modules, which have to be exchanged? Or maybe he just has to change a separate line of code? So, the engineer also has to know the variability mechanism to apply for configuration.

10.4 Architectural Relevance of Variability Mechanisms in Product Family Engineering

As shown in section 10.3 the variability mechanism selected for variability realization can have an impact on the scoping and pricing considerations of the business manager. This information should therefore be presented to the business manager in a preferably concentrated and descriptive form. A product family architecture model would serve well for this purpose. Moreover, the desired system properties are typically balanced thoroughly by the architect during the design phase. Due to their impact on the system properties, variability mechanisms have to be taken into consideration during the design of the product family. Finally, the (configured) product family architecture serves as the main blueprint for the requirement compliant implementation and configuration of the implementation artifacts. Therefore, the (configured) product family architecture also needs to provide information about which variability mechanisms to apply.

To sum up these considerations, we think that information concerning the realization of the product family variability should be provided by the product family architecture.

The representation of variability mechanisms in the product family architecture hasn't been analyzed systematically up to now. Most related approaches regard variability mechanisms only at model level and only sporadically [Cla01, Gom05, Rvd05, ZHJ03]. A conceptual description of variability mechanisms for process models can be found in [BeK04]. However, a concrete notation as well as con-

siderations regarding the implementation is missing. Both aspects are covered in [JGJ97], but the selection of variability mechanisms is only exemplary.

10.5 Variability Mechanism Centric Process Family Architectures

In this section we outline an approach for variability mechanism centric process family architectures and analyze the impact of the process family architecture variability mechanisms on the development related properties of the process family architecture.

For representing variability mechanisms in product family architectures for process oriented systems (in the following also denoted as process family architectures) we have identified a set of architectural variability mechanisms and have mapped them to UML Activity diagrams [Sch06], State Machines [Sch06a], and BPMN [ScP06]. The variability mechanisms can be categorized into *basic variability mechanisms* and *variability mechanisms*, which are derived from other variability mechanisms. Basic variability mechanisms are stand-alone mechanisms, which don't require any other variability mechanisms. Basic variability mechanisms comprise *encapsulation of varying sub-processes*, *parameterization*, *addition/omission/replacement of single elements*, and *data type variability*. Concerning the second category of derived variability mechanisms we can further divide this category into *variability mechanisms derived by restriction* and *by combination*. *Process inheritance* and *extension* are two examples for variability mechanisms derived by restriction and *design patterns* are an example for variability mechanisms derived by combination.

During product family implementation these variability mechanisms represented in the process family architecture are then mapped to respective variability mechanisms used for implementing the variability. Our approach therefore also supports a more model-driven process family development. How the variability mechanisms are implemented depends of course on the application domain as well as on the programming language, which is outlined for example in [Sch06] for the variability implementation in C or in [ScP06] for the variability implementation in Java.

As mentioned already in section 10.3 apart from the impact on the system properties resulting from the mapping of the process family architecture variability mechanisms to implementing variability mechanisms, they also have an impact on the development related properties of the process family architecture itself. The variability mechanism *encapsulation of varying sub-processes* for example supports the maintainability of the architecture, since the variability can be identified more easily due to its reduction to a clearly separated region (better separation of concerns). Encapsulation of variability also supports the reuse of encapsulated subprocess variants in other projects as well as the evolvability of the architecture, since new variants can be added easily as long as they stick

to the subprocess interface. However, encapsulation not always leads to the optimal reuse of common architecture parts as already discussed in section 10.3. *Addition/omission/replacement of single elements* and *data type variability* perform better in this respect. They support the flexibility of the architecture by allowing for more fine-grained adaptations. On the other hand by spreading the variability over the process family architecture they make maintenance harder. The flexibility of the architecture also depends on the availability of appropriate adaptation techniques. Using variability mechanisms, the architect can realize an optional activity for example by either adding or deleting the activity (variability mechanism *addition/omission/replacement of single elements*) or by defining an extension point (variability mechanism *extension*) into which the optional encapsulated behavior can be integrated. While the first technique would probably lead to a more efficient implementation, since no placeholder for the optional behavior would remain in the code, the latter one better supports the evolvability of the architecture. In addition to the variability mechanisms introduced here, new variability mechanisms with different properties can be defined by deriving them by combination or restriction. For example, as already discussed in section 10.2, the derivation of structurally correct process variants from the process family architecture is an important issue in process family engineering. So, a new variability mechanism can be derived that guarantees for the preservation of the structural correctness during configuration. This holds for example for the variability mechanism *Activity diagram inheritance* [ScP05] that restricts the *addition/omission/replacement of single elements* to a subset of correctness-preserving transformations. For families of service-oriented applications a new variant of the variability mechanism encapsulation of varying sub-processes could be derived that restricts the possible sub-process (i.e. service) implementations to those service implementations providing the required functionality. The necessarily required functionality can be described by a process model. The suitability of potential service implementations can now be evaluated by comparing them to this process model using a bisimulation checker. However, this requires that formal process descriptions exist for the potential service implementations.

Figure 33 shows an example for a variant-rich Activity diagram where variability is modeled following our approach by showing the variation points (stereotype «*VarPoint*»), the variants which can be bound to the variation points (stereotype «*Variant*»), and the variability mechanism assigned as a third stereotype connecting the variants with their variation point. Additionally, the binding time can be displayed by means of a tagged value (tagged value key *bt*) of the variability mechanism stereotype and, if necessary, the implementing variability mechanism may be uniquely identified by adding an identifier (tagged value key *id*). The system requirements a variation point implements can be represented by means of a tagged value (tagged value key *feature*) of the variant stereotype, which can hold a list of system requirements.

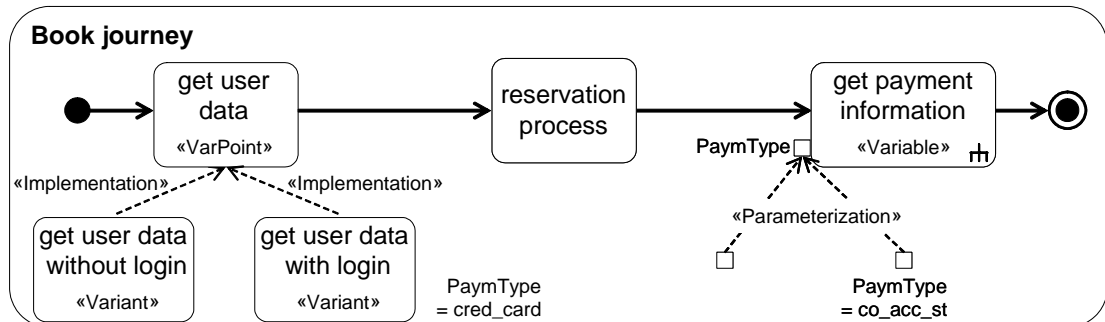


Figure 33: Example for Variant-Rich Activity Diagram

Figure 34 shows an Activity diagram variant that has been derived from the variant-rich Activity diagram in Figure 33. The example shows that the engineer can still deduce from the diagram which kind of configurations have to be done at which places in the system.

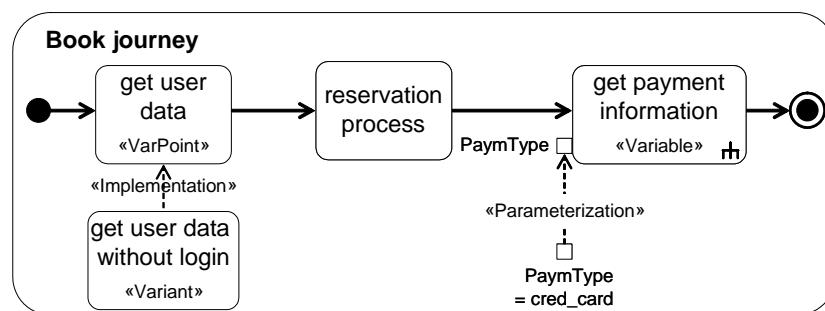


Figure 34: Activity Diagram Variant

10.6 Conclusions

In this paper, based on the requirements of their main stakeholders, we have motivated the need for representing variability mechanisms in product family architectures. We have also outlined an approach for modeling variability mechanism centric product family architectures for process-oriented software in UML.

Open issues for future research comprise a more systematic and comprehensive collection of properties derivable from the architectural representation of the variability mechanisms identified in section 10.5. We also want to analyze the correlation between the variability mechanisms for variant-rich processes and the properties of the derivable processes, such as their structural correctness.

10.7 References

- [BaB01] F. Bachmann and L. Bass. Managing Variability in Software Architectures. In Proceedings of SSR'01. ACM Press, 2001.
- [BeK04] J. Becker and R. Knackstedt, editors. Wissensmanagement mit Referenzmodellen: Konzepte für die Anwendungssystem- und Organisationsgestaltung (Knowledge Management with Reference Models: Concepts for Application System and Organization Design, in German). Physica, Verlag, Heidelberg, 2004.
- [BHM01] L. Brehm, A. Heinzl, and M. L. Markus. Tailoring ERP Systems: A Spectrum of Choices and their Implications. In Proceedings of HICSS'01, 2001.
- [Boh96] K. Bohr. Handwörterbuch der Produktionswirtschaft (Concise Dictionary on Production Management, in German), 2nd ed., ser. Enzyklopädie der Betriebswirtschaftslehre. Schäffer-Poeschel, 1996, vol. 7.
- [Bos00] J. Bosch. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley, 2000.
- [Cla01] M. Clauss. Modelling Variability with UML. In Proceedings of Young ResearchersWorkshop GCSE'01, the 3rd International Symposium on Generative and Component-Based Software Engineering, Erfurt, Germany, 2001.
- [CIN01] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. SEI Series in Software Engineering. Addison-Wesley, 2001.
- [FLR02] C. Fritsch, A. Lehn, R. Rashidi, and T. Strohm. Variability Implementation Mechanisms: A Catalog (Internal Paper). Robert Bosch GmbH, Tech. Rep., 2002.
- [GBS01] J. van Gurp, J. Bosch, and M. Svahnberg. On the Notion of Variability in Software Product Lines. In Proceedings of WICSA 2001, August 2001.
- [Gom05] H. Gomaa. Designing Software Product Lines with UML: From Use Cases to Pattern-based Software Architectures. Addison-Wesley Professional, 2005.
- [GoW04] H. Gomaa and D. Webber. Modeling Adaptive and Evolvable Software Product Lines Using the Variation Point Model. In Proceedings of HICSS'04. IEEE Computer Society Press, 2004.

- [JGJ97] Ivar Jacobson, Martin Griss, and Patrik Jonsson. Software Reuse: Architecture, Process and Organization for Business Success. Addison Wesley Longman, Harlow, England et al., 1997.
- [Mat04] M. Matinlassi. Comparison of Software Product Line Architecture Design Methods: COPA, FAST, FORM, KobrA and QADA. In Proceedings of ICSE'04. IEEE Computer Society, 2004.
- [OMG05] OMG. UML 2.0 Superstructure Specification. August 2005.
- [PBL05] K. Pohl, G. Böckle, and F. van der Linden. Software Product Line Engineering: Foundations, Principles, and Techniques. Springer, 2005.
- [Rvd05] M. Rosemann and W.M.P. van der Aalst. A Configurable Reference Modelling Language. BPM Center Report BPM-05-10, BPMcenter.org, Eindhoven University of Technology, 2005.
- [Sch06] A. Schnieders. Variability Mechanism Centric Process Family Architectures. In Proceedings of ECBS'06. IEEE Computer Society Press, 2006.
- [Sch06a] A. Schnieders. Modeling and Implementing Variability in State Machine Based Process Family Architectures for Automotive Systems. In Proceedings of SEAS'06. ACM Press, 2006.
- [ScP05] A. Schnieders and F. Puhlmann. Activity Diagram Inheritance. In Proceedings of BIS'05, 2005.
- [ScP06] A. Schnieders and F. Puhlmann. Variability Mechanisms in E-Business Process Families. In Proceedings of BIS'06, ser. Lecture Notes in Informatics (LNI), vol. P-85. Gesellschaft für Informatik, 2006.
- [SvB00] M. Svahnberg and J. Bosch. Issues Concerning Variability in Software Product Lines. In Proceedings of IW-SAPF-3. Springer-Verlag, 2000.
- [ZHJ03] Tewfik Ziadi, Loïc Hélouët, and Jean-Marc Jézéquel. Towards a UML Profile for Software Product Lines. In PFE, pages 129–139, 2003.

11 Good Practice Guidelines for Code Generation in Software Product Line Engineering

Neil Loughran, Iris Groher and Awais Rashid

The creation of generalized software artifacts is a crucial element in the development of a software product line. Code generation is a technique that allows such assets to be made more reusable in other contexts. However, guidelines for the creation of such 'core assets' using code generation are something not explicitly addressed in the literature. While there is an abundance of material which discusses variability per se, there is still a scarcity of information pertaining to how best to apply code generation in order to facilitate the development of generalized assets. In this paper we offer some general guidelines for creating highly reusable software components based upon our experiences of using code generation variability techniques in an aspect-oriented software product line.

11.1 Introduction

Product line engineering [1] is an approach which facilitates the development of a highly reusable and adaptable software architecture that targets a particular business domain. Potentially, the reuse of software assets allows high quality applications to be delivered quickly and to within economic constraints. The creation of reusable software artifacts is therefore one of the primary aims.

In this paper we illustrate our experiences of software product line development using aspect-oriented programming (AOP) [2], traditional variability mechanisms (i.e., inheritance, conditional compilation and so forth) and code generation [3][4][5] with the intention of eliciting guidelines and patterns. The guidelines we present are by no means exhaustive, complete or, in some cases, entirely earth shatteringly novel. Indeed, many of the guidelines could be described as basic common sense, or suggestions for providing some insights to the developer. Therefore they should be taken as means to motivate the research, discussion and promote the need for such guidelines to be made more explicit.

Our experiences lead us to believe that following explicit guidelines when combining AOP, traditional variability techniques and code generation, results in software that is flexible to adapt to new contexts while maintaining the principles of modularity, maintainability and evolvability.

11.2 Code Generation

In recent years there has been a re-emergence of interest in a *code-generation* driven approach to developing software. The term 'code generation' in days gone by was used to describe the process of turning source code into assembly code. Additionally, many programmers would use code generation routines (e.g., macros) to also automate the generation of frequently used assembly language subroutines. Code generation in the modern sense typically means the production of the programming code itself, or to put it simply, code which generates code. The input model (often called a *domain specific language* or DSL) will be of a much higher level of abstraction (e.g., XML script, graphical language, etc.), mapping to domain specific features and properties, encapsulating the complexity and finer details of program code. This allows developers and system configurators to concentrate on domain specific configuration details and variabilities.

It is often required that we need to go beyond language level variability mechanisms (e.g., inheritance, conditionals, generics etc.) in order to implement specific kinds of variabilities. This is especially true in a software product line or model driven development (MDD) [6] context. Code generation promotes an MDD approach to software generation, where the model (e.g. our specification of variabilities) drives the application generation.

For example, suppose a software developer wanted to develop a reservation/booking product line which could be adapted to the different needs of customers. A reservation can be made for practically anything, from a holiday or hotel booking to even a hairdressing appointment. These kinds of scenarios would require different kinds of data structures, GUI forms, data views and backend database tables at the very least. However, on the whole, a great deal of the application code in the domain will be common to all products e.g., main business logic, majority of GUI code, etc. Code generation does not limit itself to just the creation of program code; it is possible that many different artifacts (e.g., configuration scripts, tests, documentation, SQL, etc.) can be produced from a single suitable high level abstraction.

Moreover, code generation doesn't preclude language level variability mechanisms. Indeed, as detailed in Section 11.3, we have found that the appropriate usage of both code generation and language level variability techniques can greatly diminish the problems that they have when used in isolation.

The following subsections illustrate a variety of the different code generation approaches that are in use, and assesses their relative merits and demerits.

11.2.1 Brute Force Generation

Brute force code generation is a term coined by Kathleen Dollard in [4]. It refers to the embedding of program code within a program. Figure 35 demonstrates a simple 'Hello' generator which takes parameters from the command line and adds them to the embedded program which is then subsequently written to a file. While the example is no doubt very trivial, it demonstrates the simplicity of the approach.

```
import java.io.*;

class HelloGenerator {
    public static void writeFile(String s) {
        // code
    }

    public static void main(String args[]) {
        writeFile("class Hello {" +
            "    public static void main(String args[]) {" +
            "        System.out.println(\"Hello "+args[0] + "\");" +
            "    }" +
            "};");
    }
}
```

Figure 35: Brute Force Approach

The brute force approach is useful for small scale concerns which are not likely to change much, if at all, in their lifetime. However, as the size of a program increases, the code within code approach gets very difficult to visualize and maintain. Nonetheless, the approach is easy to comprehend and doesn't require special parsing tools.

11.2.2 Template-based Generation

The template based approach involves the creation of specially written code templates that are then processed via a generator which performs a transformation to generate output. A specification file containing the customizations is often used to drive the technique (as shown in Figure 36). Code templates contain program code combined with directives for providing functionality such as iteration, setting and getting of meta values, program segmentation and conditionals, to name but a few. The generator reads in both the specification and required template files and outputs the appropriate program code based upon the developer's requirements.

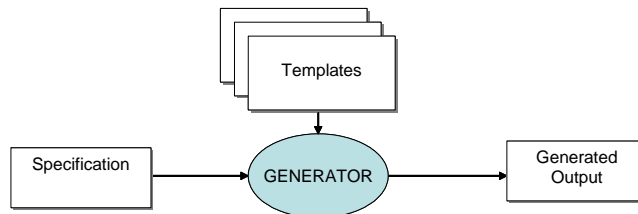


Figure 36: Template-based Generation

There are a wide variety of template languages available such as XSLT [7], XVCL [8], CodeSmith [9], Velocity [10] and XPand [11] to name but a few techniques. Many techniques use XML or XML-like notations in order to mark the code. The example in Figure 37 illustrates an XVCL-like template to create classes, fields and associated setter and getter methods. By providing parameters (in this case, name of TABLE, and TYPE and NAME lists) to the template from a separate configuration file, a class can then be generated.

```

<frame name = "TABLE">

class <value-of expr = "TABLE"/> {
  <while using-items-in = "TYPE,NAME">
    private <value-of expr = "TYPE"/> <value-of expr = "NAME"/> ;

    public void set<value-of expr = "NAME"/> (<value-of expr = "TYPE"/> s) {
      <value-of expr = "NAME"/> = s;
    }
    public <value-of expr = "TYPE"/> get<value-of expr = "NAME"/> {
      return <value-of expr = "NAME"/>
    }
  }
</while>
</frame>
  
```

Figure 37: Example of Class Generation Template

The template based approach has the advantage of having the code externalized from the generation mechanism. This allows the code to be effectively maintained and evolved. However, a generator must be utilized or specially written to work with the templates in order to provide the necessary transformations.

11.2.3 Annotation Pre-processing

Languages such as Java provide an annotation mechanism for adding meta-data to source code. JavaDoc [12] was an early example of using annotations in the automation and creation of class documentation. XDoclet [13] uses annotations (although they call them attributes) for the creation of boiler plate code and configuration files. While the technique was originally developed with J2EE

[14] beans in mind, it has now developed into a fully fledged general purpose code generation engine.

Since the release Java 5 [15], annotations have developed into a language level feature in their own right, allowing user defined annotations to be created. The annotations can then be parsed via the annotation processing tool (APT), which is somewhat equivalent to the template-based generation approach discussed earlier in Section 11.2.2. The key difference here being that annotations only allow external artifacts to be created and not transformations to the source code in which they are located. It is interesting to note that annotations have become a popular and convenient method of providing AOP [16] [30].

11.2.4 Reflection-based Generation

Reflection based code generation facilitates the generation of code based upon executing context. For example, an application could discover at run time, the structure of data has changed thus requiring changes in its persistent nature (e.g. generation of SQL). In particular, code generation provides a workaround for the restrictions imposed by Java reflection, which is limited to introspection. As stated in [17], it is possible to generate code based upon currently executing context, then automatically compile and execute/load that code thus simulating behavior changing capabilities. In a similar fashion, the persistence aspect in [18] contains a 'brute force' implementation for generating SQL code based upon introspection of the applications data structures. Reflection based code generation is useful when certain information can only be ascertained at run time.

11.2.5 Pre-processor and Template Meta-programming

Languages such as C and C++ make great usage of a pre-processing facility. A pre-processor reads in a source file which is annotated with directives and then performs inclusion (e.g. `#include`), conditional compilation (e.g., `#ifdef`, `#ifndef` etc.) and macro substitution (e.g., `#define`) as appropriate. The pre-processor is often seen as a specialized code generation mechanism for supporting among others, portability between different target platforms and variants in a product line. Many programmers try to get around the need for a pre-processor by using static final variables in conditional statements, thus relying on the compiler to perform the necessary code optimizations on unreachable statements. Similarly, it is often said that modern programming languages such as Java have largely obviated the need for a pre-processor. However, in the product line context, e.g., mobile phones, the need for a pre-processor is still very much in evidence in dealing with vendor and device specific APIs. The J2ME platform still needs to deal with how different vendors implement specific

types of graphics and sound. Even the APIs for making a mobile phone vibrate can be quite different.

Template meta-programming [19] allows the compiler to act as an interpreter thus allowing programs to be generated at compile-time. Figure 38, illustrates a factorial function using template meta-programming.

```
template <int N>
struct Factorial
{
    enum { value = N * Factorial<N - 1>::value };
};

template <>
struct Factorial<0>
{
    enum { value = 1 };
};
```

Figure 38: Template Meta-programming

Passing an integer (e.g., **Factorial<7>**) into the template instructs the compiler to generate and optimize (if needed) the appropriate code before it is turned into binary. Template meta-programming can be seen as a convenient way of creating programs from smaller programs, although the syntax can be rather esoteric and difficult to maintain.

11.2.6 Summary and Discussion

The brute force approach is particularly useful for encapsulating functionality that is unlikely to change too much but suffers from scalability issues. Therefore, it is generally only useful for well contained concerns rather than complete applications.

The template-based approach provides a much clearer alternative which is scalable and evolvable, but requires tools to be written for processing the templates. While general purpose languages are available which can help in this regard, the languages often don't provide the necessary abstractions for a particular domain. However, most general purpose languages can work on any kind of textual artifact (e.g., code, scripts, documentation, etc), thus providing a convenient way to implement variants on all kinds of different artifacts.

Annotation based approaches provide a convenient way of generating useful artifacts that support the application (e.g., descriptor files, scripts etc). However, at present, annotations do not allow the source code in which they exist to be transformed.

Reflective code generation is an interesting way to generate code dynamically, based upon application context. However, the technique (when used with Java)

still does not give us the complete vision of reflective capabilities. Nonetheless, the technique is a promising one and we believe offers up some interesting new research avenues.

The C/C++ pre-processor has been the standard method of configuring variants in the industry for some time now. Template meta-programming offers a powerful way of using code generation in the C++ language. However, the technique is not easy to use in practice and is language specific.

One thing that has not been mentioned so far has been the different perspectives of the creator (i.e., the person who creates the artifacts) and user (i.e., the person who utilizes the artifacts for their own needs). It is highly likely in practice that these will be different people. The user of the artifact may only be interested in configuration details and require domain specific abstractions. Code generation techniques can facilitate such approaches by only exposing variants to the user.

On the whole code generation has the potential to be very flexible allowing any kind of parameterization on a wide variety of artifacts. It shields the user of the artifact from its underlying complexity by exposing only domain concepts.

However, the technique is certainly not without its disadvantages:

- Static validation of the templates is not often possible.
- Potential decrease in comprehensibility and evolvability.
- Compile-time and run-time errors refer to generated code not templates.
- General purpose languages don't always provide the abstractions needed for given domains.
- Debugging heavily annotated templates is very difficult.

In the next section we describe guidelines that we have found useful which go some way in to rectifying the above demerits.

11.3 Guidelines and Patterns for Software Generation

In this section we provide guidelines that we believe can facilitate the use of code generation.

Before going any further let us present the following maxim:

Maxim. We want to create high quality, reusable assets which are easy to create, adapt and evolve with acceptable levels of performance.

We have attempted some categorization of the guidelines although this is merely to help with their organization. Therefore, many of the guidelines tend to crosscut categories. Additionally, it should also be noted that many of the guidelines we propose are based upon the template-based generator approach as detailed in Section 11.2.2.

11.3.1 Identification

The first thing to decide is whether a code generation approach is an appropriate solution to the problem at hand. If the problem involves a lot of repetition and variability, then it's highly likely that code generation can provide the necessary solution. A one-off application is probably not going to be an appropriate use of code generation.

Identifying where code generation can be utilized in the software product line context is an important activity. At its most basic we can simply take existing legacy code, identify the variants and then parameterize code appropriately. However, while this approach can work, it may not be the most appropriate for a product line which is likely to evolve. Certainly, there may need to be a significant amount of refactoring and redesign involved.

Repetitive processes are excellent candidates for code generation. Such examples of repetitive tasks exist in the relational database world where there are lots of schemas, relationships, SQL queries, domain objects, data access objects (DAO), descriptors and so forth.

Commonality and variability analysis [20] can facilitate the identification of domain concepts, vocabularies and concerns. In creating a product line from scratch we have found through our own experiences that it is more fruitful to identify concerns and aspect candidates as early as possible using aspect-oriented requirements engineering (AORE) principles [21]. The EA-Miner tool [22], a natural language processing tool, can help in this regard by discovering concerns and aspect candidates in requirements documents.

Aspects greatly help with the simplification of code generation by facilitating modularization and thus reducing much of template language meta-code. This is most noticeable when dealing with crosscutting and tangled conditional directives. Aspects can be part of both the common and variable parts of an application framework. Indeed, aspects can benefit from being variable in themselves.

Feature modeling [23] provides a way to model variant features diagrammatically, therefore providing a way to explicitly state configuration details. Common and variable elements can be delineated in order to aid configurability.

11.3.2 Creation

The first guideline for the creation of code generation artifacts is “use common sense”. Code generation permits a great deal of flexibility, but it should be used carefully and scope should be clearly defined and adhered to.

In order to aid composability it is important to separate variable and common content from one another. This can be done using language level variability techniques such as inheritance, virtual classes, partial classes and so forth. However, the use of a code generation approach does not preclude the use of language level variability mechanisms. In essence, code generation should be used to complement the variability mechanisms that are available natively in the language. Therefore, a developer should use language level variability mechanism where they are sufficient in order to keep as much code as possible under the control of the IDE and compiler. This can facilitate debugging and syntactical correctness of the program.

In certain scenarios code generation may be used for generating a subset of a program (e.g., persistence tier, domain objects, GUI, etc.) rather than an entire application. Therefore, respect must be given to handcrafted code by modularizing it away from generated code [4].

A basic, though not ‘hard’, guideline that we have found beneficial, is to keep each code generation artifact at the same modularity as the intended generated output. In other words, there should ideally be a 1:1 mapping of templates to the generated classes. The exceptions to the rule being classes which contain other classes (i.e., inner classes, multiple classes in a single file) and partial classes as in C#. If the intent is to generate a class, then the template for generating that class should not be split into arbitrary fragments of text (e.g., a method implementation) in separate files. This improves debugging support whereby compile time and run time errors can be tracked down to a particular template.

If there is a need to break up a module (e.g., module is becoming too large) then this should be accommodated using inheritance, aspects, patterns, partial classes and so forth.

It is possible to utilize code generation in order to provide variability features that aren’t supported natively in the language. For example, the Java language does not support parameterized mixin inheritance [24]. We can emulate this behavior using code generation by simply parameterizing the inheritance parameter accordingly, as shown in Figure 39. Thus the parameter (indicated using `<@baseClassA>`) will select a different base class.

```
class A extends <@baseClassA> {  
  
    // impl..  
}
```

Figure 39: Parameterized Mixins using Code Generation

Similarly it is possible to lessen code generation conditional compilation directives using a number of techniques. If the variants are well contained then we can use language level variability mechanisms and parameterization together. In Figure 40 we use a compiler optimization trick which removes the appropriate unreachable code.

```
private static final String vendor = <@vendor>;  
  
public void foo() {  
    if(vendor == "NOKIA") {...impl...}  
  
    else if (vendor == "SIEMENS") {...impl...}  
  
    else if (vendor == "ERICSSON") {...impl...}  
  
    etc..  
}
```

Figure 40: Using Compiler Optimization and Code Generation

If the variants exhibit a crosscutting nature (e.g. persistence) then aspects can be utilized to modularize the features. The aspects can then also be generalized via code generation if required. For example a persistence feature, which may be optional, can benefit from being generalized to allow for different schemas, relationships and drivers to be expressed.

It can be worthwhile constraining the range of possible parameters that are passed to the generation mechanism. Mapping such constraints from the feature model can be done in a variety of ways, using XML or an IDE. For example, pure::variants [25] allows the mapping of such constraints to be made explicit, giving feedback to the developer if an incorrect configuration or parameter is included.

Templates should ideally be created from something that is known to work. Therefore, the developer should create an example of the kind of artifact that they want to generate, test it then reify or refactor this into an appropriate template and test again.

In our experience, templates can get very complex if they try and do too much. Therefore, if a template is starting to get too complex the developer should consider refactoring it or creating a new template to simplify the code. Doing so can ease configuration details as well as the evolution of the asset itself.

11.3.3 Quality

The generated application should be of an equal or higher quality to a hand-written one. It should allow effective testing and be maintainable [4]. It is often stated that code generators produce bad code. While there is certainly evidence of generators which generate thousands of lines of ugly hard to understand code (e.g., the GUI generator in Visual C++), we are of the opinion that by following our guidelines, a developer can go some way to dispelling such statements. Indeed, when it comes down to it, there is nothing to stop people writing bad code in any language.

11.3.4 Comprehensibility

One of the main problems of using a code generation approach is the comprehensibility of the heavily annotated templates. As previously stated, using language variability mechanisms such as inheritance to separate variability 'hot-spots' from the common code can improve the situation dramatically.

Particularly with XML approaches, the developer should be able to create specifications that specify their intentions. Therefore, strive for a structured data model in the specification in order to simplify the specification itself and the templates.

Convention over configuration principles allow for default values in order to ease configuration and specification details [26].

The use of an adapter can improve intelligibility and provide more domain specific abstractions. For example, a simple parser can be written that converts meta-tags that the developer wants to use into that used by a general purpose code generator such as XSLT.

Code beautifiers (on both the templates and generated code) can be utilized to ensure the readability of code. Many IDE tools offer this option explicitly.

11.3.5 Usage

Editing of generated code should be avoided unless there is a systematic way in which to integrate the fix back into the templates. However, such 'round trip' engineering tools are very difficult to provide in reality.

In [4] Dollard maintains that it should be possible for anyone to regenerate the code as a 'one click' process. This also makes the distinction that the person who is responsible for the generation code may not be the person who was responsible for its creation.

The use of tool support in the creation and generation stages is vitally important. An IDE such as Codesmith [9] facilitates template creation by allowing previews of the generated code as the developer is creating the template itself. The Eclipse platform [28] has a lot of support for a variety of MDD and code generation tools due to the availability of plug-ins.

It is useful to generate other kinds of artifacts other than code. For example, templates which generate HTML and WordML [27] can describe the structure of the system as it evolves. Any changes to the model propagate to all artifacts giving immediate feedback to the developer.

11.3.6 Maintenance and Evolution

As change is the only certainty, the developer must have control of the generalized assets (i.e., the templates) so that they can be evolved to fit the needs of the future [4]. For this reason, in general it is preferable to use templates, rather than embedding programs within programs.

11.4 Code Generation in a Model-Driven Aspect-oriented Framework

This section describes our experiences of using the code generation approach in an aspect-oriented framework.

11.4.1 Object-oriented Frameworks

Object-oriented frameworks [29] modularize common core behavior in an application domain while exposing variation points via inheritance. This separation of common and variable functionality allows application developers to concentrate on just the customizable content rather than the whole system. The framework is therefore abstract and incomplete until a developer creates a subclass by hand which concretizes the variation point to their own requirements.

11.4.2 Aspect-oriented Frameworks

Aspect-oriented frameworks build upon object-oriented frameworks by allowing otherwise crosscutting features of the system to be modularized as aspects. These crosscutting features can be internal to the core framework or part of the variabilities. By using an AOP language, such as AspectJ [30], customization points for aspects (i.e., pointcuts) can be defined in abstract aspects. Aspects can then use these pointcuts in order to provide an implementation. Using abstract aspects for pointcut declarations purposely restrict the available points to

which aspects can bind to the framework core, thus reducing the likelihood for errors (e.g., incorrect binding sites).

11.4.3 Generative Aspect-oriented Framework

Using AOP and OO to handle variants provides a lot of flexibility and improves modularity. However, in order to fulfill their usefulness in a software product line context those aspects and classes often need to be generalized (i.e. via code generation). Code generation and AOP have often been seen as competing techniques for solving similar problems. However, the collaborative work between European partners within the AOSD-Europe project [31] highlighted a number of cases where code generation and AOP were combined together in order to simplify the creation of key concerns [32], notably persistence, advanced transaction management, schema type versioning and caching amongst others. Combining AOP with code generation improves evolution of concerns by allowing otherwise crosscutting features to be modularized in one place. Additionally, any changes to the model are propagated throughout the code hence it's suitability in the MDD context. Figure 41 illustrates how the configuration model selects specified features and generates the variable part accordingly.

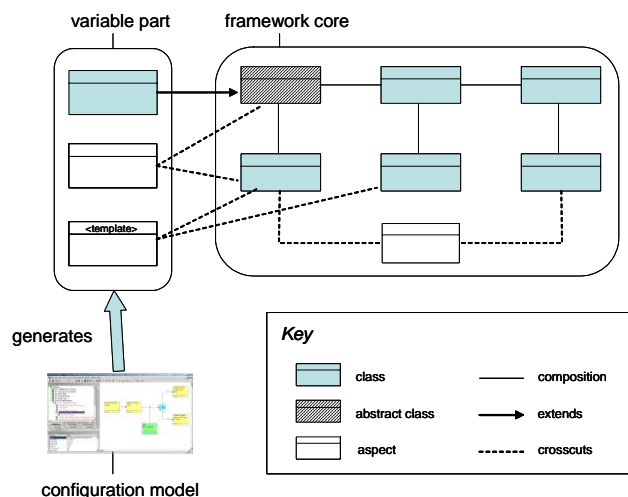


Figure 41: Code Generation-driven Aspect-oriented Framework

The core part of the code generation driven aspect-oriented framework consists of the target language code (e.g., Java) and is frozen (i.e., it does not change, except when it needs to evolve). Code generation is utilized to generate the variable part of the framework. The appropriate aspects and classes are generalized using a template approach as previously discussed in Section 11.2.2. We utilize the plug-in nature of AOP to handle crosscutting optional and alternative

features, and use the guidelines and patterns as discussed in Section 11.3 to improve comprehensibility and maintainability.

11.5 Conclusions

In this document we have illustrated how an AOP software product line can be facilitated using code generation. We have also demonstrated that explicit guidelines and patterns are required in order to lessen the negative connotations that the code generation approach has garnered. Using code generation with, language level variability mechanisms and frameworks have lead to our convictions that code generation can be a vitally important mechanism if used correctly. On the whole we believe that general purpose code generation techniques do not always provide the necessary abstractions, often leading to verbose meta-code. In order to address this we believe that code generation tools should be extensible in order to provide abstractions that are closer to the domain in which they are intended. While this places greater demands on tool support we believe that the effort is worth it.

11.6 Acknowledgments

This is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

11.7 References

- [1] P. Clements and L. Northrop, Software Product Lines - Practices and Patterns, Addison Wesley, 2002.
- [2] R. Filman, T. Elrad, S. Clarke and M. Aksit, Aspect-Oriented Software Development, Addison Wesley, 2004.
- [3] J. Herrington, Code Generation in Action, Manning, 2003.
- [4] K. Dollard, Code Generation in Microsoft.NET, Apress 2004.
- [5] Code Generation home page <http://www.codegeneration.net>
- [6] S. Beydeda, M. Book and V. Gruhn, Model Driven Software Development, Springer, 2005.
- [7] XSLT Home Page <http://www.xslt.com/>.

- [8] Wong, T., et al., XML implementation of frame processor. ACM SIGSOFT, 2001.
- [9] Codesmith Tools home page <http://www.codesmithtools.com>
- [10] Velocity home page <http://jakarta.apache.org/velocity>
- [11] Open Architectureware home page
<http://www.openarchitectureware.org/PureVariants>
- [12] JavaDoc home page <http://java.sun.com/j2se/javadoc>
- [13] XDoclet Home Page <http://xdoclet.sourceforge.net/xdoclet/index.html>.
- [14] J2EE home page <http://java.sun.com/javaee>
- [15] Java 5 home page <http://java.sun.com/j2se/1.5.0>
- [16] JBoss AOP home page <http://labs.jboss.com/portal/jbossaop/index.html>
- [17] I. Forman and N. Forman, Java Reflection in Action, Manning, 2004.
- [18] A. Rashid and R. Chitchyan, Persistence as an Aspect, AOSD 2003.
- [19] T. Veldhuizen, "Using C++ template metaprograms," C++ Report Vol. 7 No. 4 (May 1995), pp. 36-43.
- [20] J. Coplien, D. Hoffman, D. Weiss, "Commonality and Variability in Software Engineering," IEEE Software, vol. 15, no. 6, pp. 37-45, Nov/Dec, 1998.
- [21] A. Rashid, P. Sawyer, A. Moreira and J. Araujo, Early Aspects: A Model for Aspect-Oriented Requirements Engineering. IEEE Joint International Conference on Requirements Engineering. IEEE Computer Society Press. Pages 199-202
- [22] A. Sampaio, R. Chitchyan, A. Rashid, and P. Rayson, EA-Miner: a tool for automating aspect-oriented requirements identification. Conference on Automated Software Engineering 2005.
- [23] K. Kang, et al., Feature Oriented Domain Analysis Feasibility Study. 1990.
- [24] G. Bracha and W. Cook, Mixin-Based Inheritance, ECOOP/OOPSLA 1990, 303-311.

- [25] pure::variants home page http://www.pure-systems.com/Variant_Management.49.0.html
- [26] Ruby on Rails homepage <http://www.rubyonrails.org>
- [27] XML in Microsoft Office homepage
<http://msdn.microsoft.com/office/tool/xml/default.aspx>
- [28] Eclipse home page <http://www.eclipse.org>
- [29] M.E. Fayad and D.C. Schmidt. Special issue on object-oriented application frameworks. Comm. of the ACM, 40, October 1997.
- [30] <http://www.eclipse.org/aspectj/>
- [31] AOSD Europe homepage <http://www.aosd-europe.net>
- [32] N. Loughran, F. Sanen, A. Jackson, et al, A domain analysis of key concerns - known and new candidates, AOSD-Europe Deliverable D43, AOSD-Europe-KUL-6, 27 February 2006, pp 1-2

12 Beyond Code: Handling Variability in Art Artifacts in Mobile Game Product Lines

Vander Alves¹², Gustavo Santos¹³, Fernando Calheiros²,
Vilmar Nepomuceno², Davi Pires², Alberto Costa Neto¹, Paulo Borba¹

Variability management is at the core of software product lines. Such variability spans various artifacts, from requirements to code and tests. Based on our industrial experience, we address variability management of images and sound in the Mobile Game Product Lines domain. We present variability mechanisms for such artifacts, provide guidance for their choice according to a set of criteria, and assess the impact of such variability in terms of source code. Finally, we provide a more abstract view of these mechanisms so that they can be used with other artifacts.

12.1 Introduction

In Software Product Line (SPL) engineering [1], while focusing on exploiting the commonality within the products, adequate support must be available for customizing the SPL core in order to derive a particular SPL instance. The more diverse the domain, the harder it is to accomplish this task. This, in some cases, may outweigh the cost of developing the SPL core itself. Therefore, variability management is at the core of SPL. Such variability spans various artifacts, from requirements to code and tests. Depending on the domain, additional artifacts should also be considered.

In particular, in Mobile Game Product Lines [2,3,4], art-related artifacts such as image and sound need to be addressed. Such artifacts are part of the core assets and their design and maintenance demand significant resources from organizations in this domain. Additionally, during product derivation, in which a game is ported to many devices, the great diversity of such devices complicates managing variability for sound and image. Failing to address variability in these artifacts adequately affect product derivation and also impact on other artifacts, such as code, thereby increasing the difficulty in managing its variation.

Based on our industrial experience, we address variability management of images and sound in the Mobile Game Product Lines. First, we briefly review variability issues in this domain and how they arise (Section 2). Next, we present

¹² Informatics Center, Federal University of Pernambuco. Brazil. E-mail: {vra,acn,phmb}@cin.ufpe.br

¹³ Meantime Mobile Creations, Brazil. E-mail: {gustavo.santos, fernando.calheiros, vsn, davi.pires}@cesar.org.br

some variability mechanism for such artifacts and reason how their choice is influenced by some factors, such as performance, binding time, and reusability (Sections 3 and 4). Then, we evaluate how changing such mechanism affects variability management of code (Section 5). Finally, Section 6 *provides a more abstract view of these mechanisms so that they can be used with other artifacts*.

12.2 Variability in Mobile Game SPL

Variability in Mobile Game SPL arises mostly due to a strong portability requirement and to great device diversity. Indeed, portability also becomes a central business issue in the contract between game developers and service carriers, since is it not economically viable for the latter to deploy a game for a few devices, thus representing a very small fraction of customers. Additionally, since device variability is great, this is especially relevant for games, which explore most device-specific optimized features to achieve competitive quality. Although these devices are organized by similarity into families by device manufactures, service carriers and developers, there still are dozens of families, and game developers must develop a game for most of them. This gives rise to SPLs with significant variability.

Based on our experience in this domain, we identified the most relevant device features and described the incurred variability. We have categorized variability in this domain. These categories are shown in Table 1:

Category	Description
<i>Device specific variations</i>	<ul style="list-style-type: none"> • Differences regarding the device itself, like: • Screen sizes and key codes; • Sound playback approach • Presence of vibration API • Image transformation API
<i>Known issues</i>	General issues (bugs) encountered in more than one device that require a workaround
<i>General variations</i>	Support of multi-language and graphical font feature variations
<i>Service carriers policies</i>	<ul style="list-style-type: none"> • Network address of the server responsible for storing/retrieving information • Executable (JAR) file nomenclature
<i>Feature variations</i>	Presence or not of features like game ranking posting

Table 1: Variability in Mobile Game SPL

Addressing all these issues results in large SPLs. In fact, our SPLs currently have hundreds of instances.

12.3 Variability Mechanisms for Images

Image handling is a key activity in the game development process. Images are used for composing scenarios, characters, menus, and all the visual entities in a game. Considering the mobile device environment, the main factor causing image variations is the high number of device display sizes. Display size variation requires image resizing in such way that the figure elements fit into each display configuration. This way, besides code, images are product line assets affected by some factors that cause variations, thereby requiring corresponding variability mechanisms. In the following, we describe some of such mechanisms (automatic transformation, image decomposition, and location obliviousness) and reason on their choice by striking a balance among factors such as binding time, performance (space and time), and reusability.

The use of **automatic transformations** increases the reusability of images and demands a smaller effort from the graphics designer, who does not have to redraw all the images in a new scale for each device screen size. From the binding time perspective, there are two approaches to automatic image transformation: runtime and compile-time. In the former, the operations of image resizing, flipping, and color changing rely on an API and results in a decrease on the final executable size, which is the great advantage of this approach, since application size is one of the main development constraints for mobile devices [3]. This, however, has a moderate negative impact on performance, since the application now loads the image and transforms it, instead of just loading it; additionally, heap size usage also increases.

Compile-time automatic image transformation can be accomplished by the combination of image parameterization and image manipulation tools. In this approach, the game art is created for the largest screen size, and resizing parameters are set in a configuration file that is read by a tool creating resized images based on the reference image. It has the advantage of requiring less heap, and it does not have a negative impact on performance.

Both approaches of automatic image transformation may lead to visual quality loss, causing a bad game perception, making it impractical to use these operations. In such cases, the work of the designer is indispensable. The designer will have to create a new image for every transformation that cannot be accomplished using the aforementioned approaches, which leads to an increase in the size of the application's executable.

Image decomposition is a variability mechanism for decreasing the amount of images in the game and improving performance. It consists of dividing an image that is a part of an animation, or that can be reused by different elements

of the game, into several parts, considering that some of these parts are repeated in more than one of the animation frames.

Two examples of usage of this technique are in Meantime's games mobile My Big Brother [5] and Ronaldinho Total Control [6]. In BBB5 ,there was only one image used for the torso of every character, and in Ronaldinho Total Control the main character was divided into several parts (arms, head, torso and legs), where the ones that moved were the arms, head and legs, so the torso image used in every frame of the animation was the same one. This required positioning the images to form the animation at runtime, as illustrated in Figure 42:

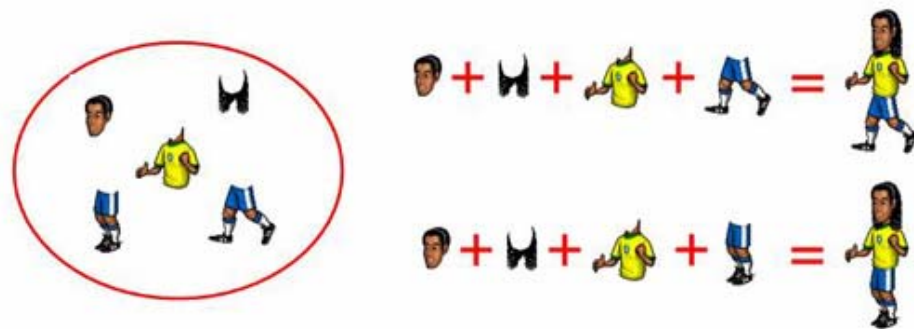


Figure 42:

Image decomposition

Location obliviousness. Variability of device display sizes affects not only image sizes, but also implies in the variability of the specification of image items positioning within these images. For every screen size, there is a need for different constants specifying such positioning. This results in the need for many constants in the code, resulting in many magic literals. The use of well-known refactorings such as *Replace Magic Number with Symbolic Constant* [7] does not suffice to address this issue, since there may be a few hundreds of such literals, most of which can be of fine granularity (not only class constants, but also as local variables). Alternatively, macro usage may impact on the legibility and IDE integration, since the code does not compile with the macro symbols. The same happens with preprocessing, a frequently used technique to address this variability. Instead, we propose addressing this variability at runtime by *Location Obliviousness*.

Most of the games' images are created by the designers in the SVG format [8], which is a XML file describing the images' elements and their positions. Packaging a SVG file and parsing it to get the values needed to paint the images at the appropriate positions is not viable since SVG, being a XML file, is very verbose and, thus, has a large file size.

In *Location obliviousness*, we convert the SVGs into a compact binary format, which has a small size and can be parsed efficiently at runtime. This format is

called BVG (Binary Vector Graphics) and supports a subset of the elements defined in the SVG standard. The use of BVG effectively removes most of the image positioning code, making it easier to read and maintain, and allows the developer to focus on the game logic. BVG supports the following elements: rectangle, image, image clip, line and arc. Each element description contains all the information needed to draw itself on the screen. For example, a rectangle element description in the BVG file contains its (x,y) position, width, height, color and whether it is a filled or a simple rectangle. Every element in the BVG file may be tagged, so that it can be identified from the game source code, so instead of the source code containing the drawing information, which required code duplication using preprocessing for each screen size, it now contains only the element's ID. Such ID is used to reference the element's information within the BVG file, which will be used for drawing the element.

12.4 Variability Mechanisms for Sound

Sound is being used in more intelligent ways in mobile applications development, especially in games. It creates a different environment, making the game more involving. The diversity of devices, their resources and the need to keep a high quality sound may demand a great effort from the sound designer. Very often the designer has to create several sound artifacts to take the maximum advantage of the devices' sound playback capabilities. As a consequence, each device family has different set of sounds.

Most devices work with MIDI audio files, but devices' constraints for sound playback lead to variations of sound artifacts that are managed by the creation of sound artifacts for more powerful devices and following a progressive reduction of audio channels, always trying to keep the quality and original sound identity. Indeed, this process cannot be completely automated. Removing some audio channel, voice or specific instrument from the audio object may cause a complete distortion of the original sound. This process is still quite dependent on the designer's artistic feeling.

Some porting tools [9] offer automatic transformations over audio resources, according to the target device, but the only guarantee is that the resulting transformed resource will be compatible with the corresponding device. However, it is frequently necessary to have a fine control over the resources file size, which requires a direct interference by the sound designer.

Sound variability can be managed by creating an audio core artifact, which is always reused as a key asset through the SPL, and the customizations are made via melody simplification and transformations between device specific formats.

Recently, another approach is being largely adopted by the mobile game industry: the use of a special MIDI format called Scalable Polyphony MIDI (SP-MIDI) [10]. In this standard, MIDI channels have a priority order and the sound de-

signer decides which sound component goes to each priority level. This way, MIDI channels can be seen as SPL assets and core-assets are determined by higher priorities. Different devices with discrepant sound capabilities can use the same artifact, but each one use only a compatible amount of channels from it. The Figure 43 summarizes the overall sound production process.

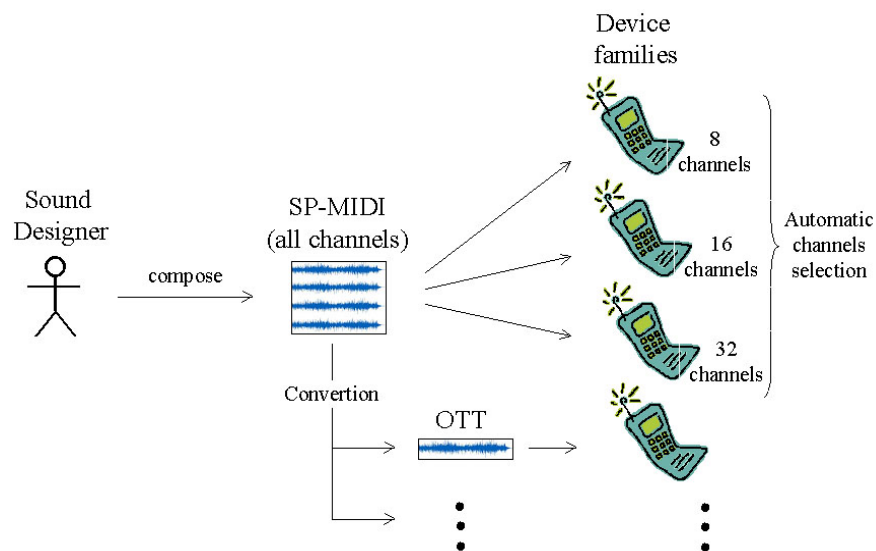


Figure 43: Sound as asset in Mobile Game Product Lines

Mobile devices are extremely restrictive regarding heap memory availability. As a direct consequence of this fact, the game programmer must be judicious about how much resources are being kept in the device heap memory at the same time. Two approaches can be applied to loading sounds: loading sounds at startup or on demand.

The first approach, **loading sounds at startup**, consists in loading sound resources to memory during game startup process. As a consequence, it slows down game initialization and requires a greater amount of heap memory. On the other hand, it reduces the existing delay to load sounds during game execution and simplifies the codification.

Loading sounds on demand consists in allocating in memory sound resources when they are necessary and deallocating them as soon as possible. This approach reduces game initialization delay and also demands less heap memory during game execution. The downside is the increase in the game execution processing and the code complexity.

12.5 Variability Across Artifacts

The choice of the variability mechanisms for sound and images directly affects code variability. The API choice, resources allocation, execution mode of the artifacts generated by these mechanisms and how these artifacts are going to be represented and used inside the code influence the flow of execution and memory allocation, both heap and non-volatile.

The variability mechanism chosen for image representation can influence both the game's executable file and used heap memory. If the information about images positioning is not present in the loaded image object, it will have to be expressed as constants inside the code, thus increasing the executable size. Placing this information on text-based properties files to be read at runtime may degrade performance. The solution presented in Section 3 (Location Obliviousness) solves these problems. It decreases the number of code constants and, since it uses a binary file, it occupies less space in the executable and is parsed more efficiently, demanding less processing power than a properties-based solution.

The choice of the variability mechanism for sound also affects the project's source code, as a consequence of different devices using different APIs and some of these APIs are more limited than others (such as the Nokia API for MIDP 1.0 devices). Additionally, in some cases the devices do not support sequential sound playback, making it necessary to create separate threads in the game flow so that playability is not affected. There are also restrictions on the type of the file supported by some devices. For these devices that contain that discrepancy it is necessary to use the file's content-type. The values that it may present are "audio/mid", "audio/x-mid", "audio/midi" and "audio/x-midi". Another variation is how the sound resources will be allocated: on demand, on devices that have low heap memory availability, or if they will be preloaded at the beginning of the game, which makes their execution response time faster. The creation of a uniform API for all devices that can be altered by code isolation using preprocessing directives is already the approach used in the industry, utilizing the preprocessor Antenna [11], a collection of Ant [12] tasks.

12.6 A General View of Variability Primitives

We discussed in the previous sections sound and image variability and how the source code is affected by variations in those artifacts. Taking a more abstract view about such variations, it is possible to generalize some key concepts and use the variability primitives in other contexts beyond sound and image handling. In a more general way, we can classify the resource/components variations according to the following taxonomy:

8. **Resource formats.** It is common in the software development environment the existence of several file formats to represent the same perceptive effect. An example is the diversity of sound formats (as discussed in Section 4). Such variability can occur for several other resources like images, texts, and

so on. All software that handles external resources needs some mechanism for treating this kind of variability. The solutions presented in the previous sections for managing different sound formats can be generalized for any other resource.

9. **Composition/Combination:** image and sound resource require some dynamic composition and filtering, like explained in Sections 3 and 4. Such combinations can occur in several other contexts, either for composing multimedia resources or composing software components. Preparing the software to deal with fragmented resources is an essential requirement to support compositional variability.
10. **Transformations:** we discussed in Section 4 the importance of dynamic transformations for images. However, transformations are frequently applied for many other resources. Indeed, dynamic transformation is a powerful mechanism for dealing with some variations that cannot be resolved statically. In general, transformations are accomplished by providing additional APIs for such purpose. On the other hand, transformations sometimes can be applied statically, either before or during building time. This can be achieved by the use of transformation tools for code and resources.

12.7 References

- [1] P. Clements and L. Northrop. Software Product Lines: Practices and Patterns. Addison-Wesley, 2002.
- [2] Vander Alves, Pedro Matos, Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In Proceedings of the 9th International Software Product Line Conference (SPLC'05), volume 3714 of Lecture Notes in Computer Science, pages 70-81, September 2005. Springer-Verlag
- [3] Vander Alves, Ivan Cardim, Heitor Vital, Pedro Sampaio, Alexandre Damasceno, Paulo Borba, and Geber Ramalho. Comparative Analysis of Porting Strategies in J2ME Games. In Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05), Budapest, Hungary, pages 123-132, September 2005. IEEE Computer Society.
- [4] Vander Alves. Identifying Variations in Mobile Devices. Journal of Object Technology, 4(3):47-52, April 2005.
- [5] My Big Brother web site, http://www.meantime.com.br/games_meubbb.html, 2006.
- [6] Ronaldinho Total Control web site, <http://www.ronaldinhomobile.com/>, 2006.

- [7] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. Refactoring: Improving the Design of Existing Code. Addison–Wesley, 1999.
- [8] World Wide Web Consortium, <http://www.w3.org/Graphics/SVG/>, 2006.
- [9] Unified Mobile Application framework web site, <http://www.unifiedmobiles.com/>, 2006.
- [10] SP-MIDI, <http://www.midi.org/about-midi/abtspmidi.shtml>, 2004.
- [11] Antenna web site, <http://antenna.sourceforge.net/>, 2006.
- [12] Ant web site, <http://ant.apache.org/>, 2006.

Document Information

Title: Variability Management –
Working with Variability
Mechanisms

Date: October 15, 2006
Report: IESE-152.06/E
Status: Final
Distribution: Public

Copyright 2006, Fraunhofer IESE.
All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means including, without limitation, photocopying, recording, or otherwise, without the prior written permission of the publisher. Written permission is not needed if this publication is distributed for non-commercial purposes.



Discussion Group on Test Strategy

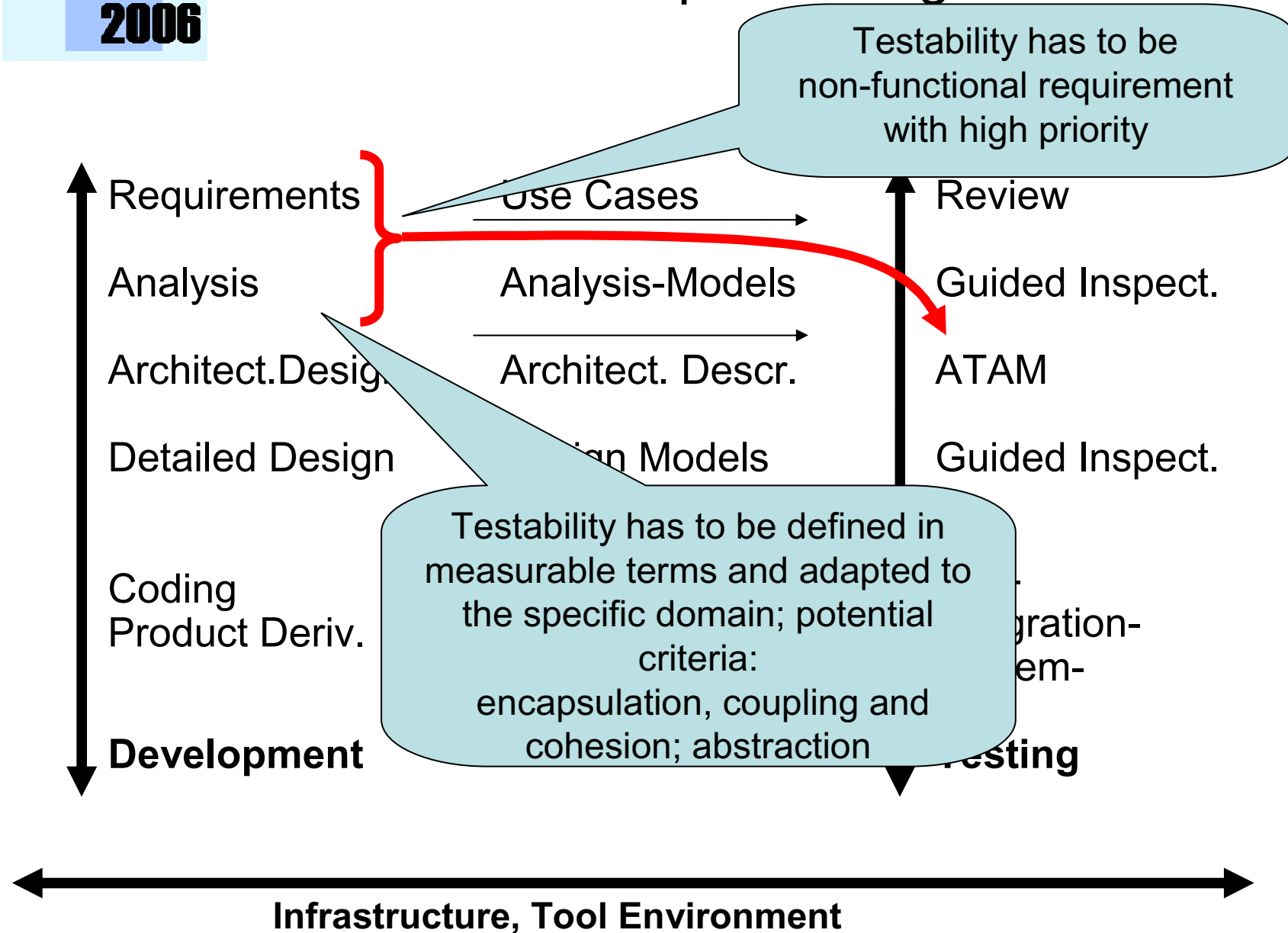
- Problem:
 - Define a competitive test strategy
when, what, how, to what extent, and by whom
a particular product line is tested at least cost to gain the most
- Challenges
 - Lack of understanding of business impact of testing
- Assessment of Effectiveness of Test and Risk
 - Composability in testing space
- Criteria for determining the Test Strategy
 - Balance between testing core assets over their variation space and testing applications
 - Balance the granularity of the artifact for observability and controllability
 - Optimize testing for cost and/or time



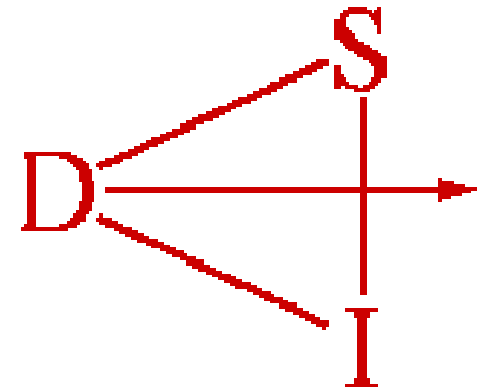
Discussion Group on Adapting Techniques and Methods for Testing Software Product Lines

- Problem: Variability in SPLs
 - Test configuration management problems
 - Feature interaction problem
- There are ...
 - Methods:
RUP, Agile, IEEE standards, Model-based testing, Formal methods (model checking), etc.
 - Variability mechanisms:
Compiler directives, Inheritance, Parameters, Templates, Separation of concerns, etc.
 - Many methods for testing single systems
- We need concrete guidelines for choosing
 - methods for testing
 - variability mechanisms for test artifacts

Discussion Group on Design for Testability

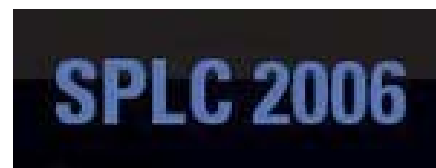


PHILIPS

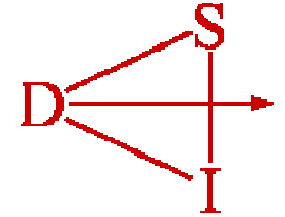


1st International Workshop on Open Source Software and Product Lines OSSPL06

Frank van der Linden
PMS CTO Office
August 22, 2006

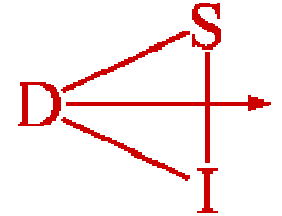


Participants



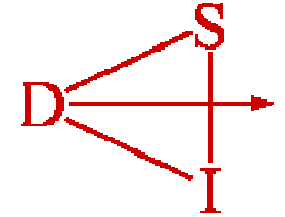
- | | |
|------------------------|--------------------------|
| • Frank van der Linden | Philips Medical Systems |
| • Jesús Bermejo | Telvent |
| • Andrew Gordon | Unisys |
| • Jilles van Gurp | Nokia Research |
| • Svein Hallsteinsen | Sintef |
| • Timo Käkölä | University Jyväskylä |
| • Jeajoon Lee | Fraunhofer IESE |
| • Henk Obbink | Philips Research |
| • Liam O'Brien | LERO-ISERC |
| • Rob van Ommering | Philips Research |
| • Arnd Schnieders | Hasso-Plattner Institute |
| • John Scott | Radiant Blue Technology |

Background



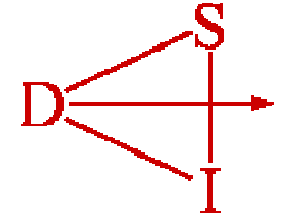
- Product-line engineering organisations
 - Use open source software
 - Effective way for good software
- Diverse use of open source software
 - Product-line development is an option for open source communities
- Presently completely different worlds
- Workshop aims to improve understanding
 - insight how they can profit from each other

Presentations



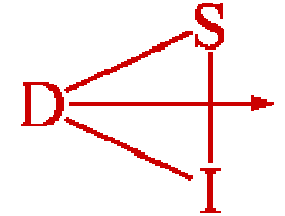
- Open source strengths for defining software product line practices
 - [Jesús Bermejo](#), and Naci Dai
- Feature-Based Determination of Product Line Asset Types: In-house, COTS, or Open Source?
 - [Jaejoon Lee](#), and Dirk Muthig
- Open Source in the Software Product Line: An Inevitable Trajectory?
 - Pär J Ågerfalk, Brian Fitzgerald, Brian Lings, Björn Lundell, [Liam O'Brien](#), and Steffen Thiel
- OSS Product Family Engineering
 - [Jiles van Gurp](#)

Discussion



- Human issues
 - The way that people are recruited and fired
 - Those that stay in OSS are those with quality
 - Culture
- Tools
 - OSS has simple useful tools – they integrate!
 - Good OS asset management tools
 - implicit variation, embedded in packages
- Processes & maturity
 - Are all SPL practices necessary?
 - Some OSS practices are working already in SPL development
- Organisational issues
 - Fear of the unknown by commercial organisations
 - Ownership
- Architecture
 - SPL concentrates on models, OSS on code/configurations
 - OSS quality can be better assessed than close source
- Business
 - How/when to use OSS in SPL – what about participation?
 - Questions about licenses

Agreements



- 2 initial papers:
 - Use of OSS practices in SPL
 - Frank, Jaejoon, Liam
 - Adoption of SPL practices in OSS communities
 - Timo, Jilles, Svein, Jesús
 - Lead to journal article?
- 2nd round: Cross conclusions
 - Jesús, Frank, ...
- Prepare Dagstuhl workshop – 3 days on this subject!
 - invite OSS people as well
- Submit papers to SPLC 2007/OSS 2007
 - Basis for the Dagstuhl workshop



Open source strengths for defining software product line practices

Jesús Bermejo (1), Naci Dai (2)

(1) Telvent, Seville, Spain
jesus.bermejo@telvent.abengoa.com

(2) Eteration, Istanbul, Turkey
naci.dai@eteration.com

Abstract

Open source is emerging as a new global paradigm challenging the conventional approach in software development. The fact that product line is a natural evolution in the maturity process of software development is leading to the adoption of related practices by open source communities. The paper presents some examples for supporting the expectation of increasing levels of adoption.

Introduction

A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [1]. The competitive advantage of software product lines is currently worldwide recognised due to increased productivity, flexibility and customisation. In fact, product lines practices are a natural evolution of software maturity development derived from transversal engineering across a set of products or a specific market.

Open Source is emerging as new global paradigm challenging the conventional approach in software development. It does not just mean access to the source code. The distribution terms of the software must not restrict any party from selling or giving it away and must allow modifications and derived works [2].

The success of an open source development is strongly related with the level of adoption outside. The need to satisfy multiple users and in many cases from different domains is frequently a direct consequence. The fact that the source code is available through Internet often leads to the use of parts in different developments and domains as opposite to inner developed software where the use of parts is limited to the developing organisation (and frequently with important organisational constraints). On the other hand, the collaboration among open source projects is common due to the flexibility in achieving it. The very favourable scenario to adopt product line core practices is in fact one of the strengths of the open software that leads to a very high quality in the long term for successful projects.

The Debian GNU/Linux case

The need to quickly address a very large variety of requirements with flexibility has pushed operating system vendors and open source communities to develop automatic configuration and deployment infrastructures taking advantage of Internet. Debian [3] was one of the first GNU/Linux distributions with tool support to cover deployment tasks such as: dependencies resolution, installation, configuration and update of packages. The Debian packaging system is one of the best existing methods for installing, upgrading, and removing software available. The approach is similar to that used in Red Hat Package Manager (RPM).

A package is a collection of files with instructions on what to do with them. It usually contains programs (although sometimes it has documentation, window themes, or other files). The package contains installation information, required libraries or other dependencies, setup instructions, and scripts for basic configuration. Although in some cases additional configuration is needed the system is some sort of on-line product line derivation support from a large catalogue of package assets. There are also several open

source projects addressing the building of Linux tailored distributions from the existing code base to minimise the effort of developing distributions for specific needs.

The Eclipse case

Multiple development and configuration tools are available for a large diversity of software intensive systems but they are typically not integrated with each other and do not necessarily build on standards. Enterprise software vendors have tooling that support their proprietary technologies; Microsoft has its Visual Studio .Net. BEA systems supports its commercial WebLogic Server environment through WebLogic Workshop. Likewise IBM has tools that support its own runtime environment WebSphere, and the tool named WebSphere Studio. These tools supporting proprietary solutions are limited in adaptability and extensibility. In the open source world, Eclipse [4] provides a modular development platform that includes all kinds of developer tools for most programming languages. Eclipse has currently the potential to become a truly cross-platform IDE and tools platform. It runs on a wide range of operating systems; it provides GUI and non-GUI tooling support; it is language-neutral; it permits unrestricted content types such as HTML, Java, C, JSP, EJB, XML, GIF; and most importantly it facilitates seamless tool integration to allow new tools to be added to existing installed products. More than five hundred plug-ins exist currently. Probably key success factors for Eclipse have been its open source nature, the target domain (a tool to facilitate the development is frequently an extra cost for a platform/ product provider and this facilitates the open source industrial cooperation) and its core plug-in architecture that is an important enabler for product line engineering practices.

Open source, service oriented architectures and product line experiences

Two relevant reference examples documenting relationships between open source developments and product line practices have been described in previous paragraphs. Many other examples can be found although there is not an explicit awareness of product line engineering practices at the moment in open source communities. Probably the relationship between open software and product line practices is more for those initiatives addressing platforms/middleware than for open source applications. This is due to the internal interest of promoting the reuse across projects and to improve interoperability.

ITEA Osmose R&D project executed from 2002 to mid 2005 provided interesting and relevant experiences in the links between open source and product line engineering practices. Discussions during the planning phase in 2002 led to the interest of an open source middleware allowing dynamic deployment of systems. At that moment OSGi was the closer specification targeting this. A community grew rapidly around the platform helped by the modularity in the architecture required for alignment with the specification. The fact that the developers and users were guided to think in terms of “bundles” (small deployable application units vs. large and monolithic components) contributed importantly to the opening of the architecture and as derived consequence to its potential of reuse across domains. The contributors to different building blocks discovered collaboration opportunities and many joined from very diverse application areas. It was also interesting to see during the project that similar approach was also adopted for Eclipse. Once Osmose finalised it raised the interest of Apache community where its relevance for the Apache Directory and other developments such as Harmony and Cocoon [5] is being discussed. ITEA OSIRIS (Open Source Infrastructure for Run-time Integration of Services) [6] project (Osmose follow-up stated in mid 2005) together with COSI (Co-development using inner & Open source in Software Intensive products) [7] is providing the opportunity to explore further the relationship between service oriented architectures, open source engineering practises and product lines embedding the complete development life cycle.

Conclusions

The interest of providing technologies for building systems faster, with lower cost and higher quality has led to advances in technologies such as component-based development, asynchronous middleware, service-oriented architectures, product line and open source.

Some product line practises can be identified in many relevant open source initiatives and communities and frequently they have been key factors for the success of the projects. Probably the relative relevance of the diverse product line practices in the context of an open source project is not the same that for inner development and this area requires further research. Nevertheless, the increasing strength of open source in creating global and “de facto” standards together with the fact that product line practices are a natural evolution in the maturity process of software development is creating strong synergies between these two fields although it has not yet been recognised explicitly.

References

Following are links of references mentioned in the text

[1] www.sei.cmu.edu/productlines/

[2] www.opensource.org

[3] www.debian.org

[4] www.eclipse.org

[5] www.apache.org

[6] www.itea-osiris.org

[7] www.itea-cosi.org

Open source strengths for defining software product line practices

10th International Software Product Line Conference

**OSSPL - First International Workshop on
Open Source Software and Product Lines**

**22 August 2006
Baltimore, Maryland, USA**

Jesús Bermejo
jesus.bermejo@telvent.abengoa.com
Naci Dai
naci.dai@eteration.com

Contents

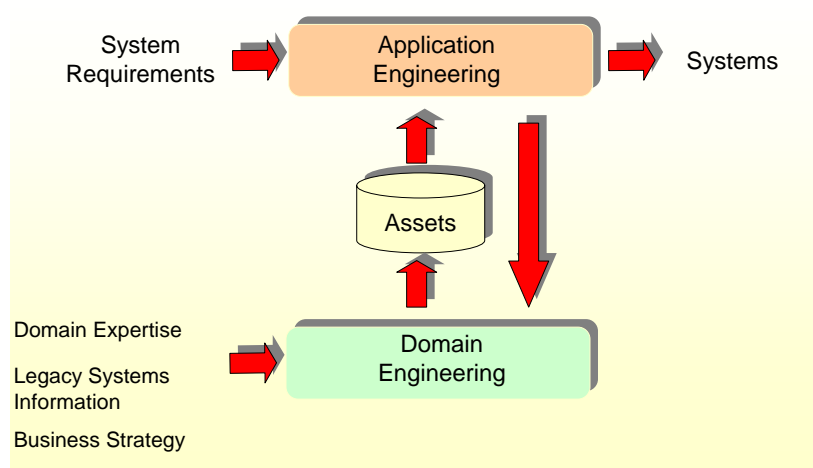
- Introduction
- Some examples
 - GNU/Linux
 - Eclipse
 - PHP
 - Osmose, Osiris, Cosi, Cosiris
- Conclusions

Introduction

- A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the **specific needs of a particular market segment or mission** and that are **developed from a common set of core assets in a prescribed way**.
i.e. Optimised reuse for a market segment/mission
- Open Source is emerging as a new global paradigm challenging the conventional approach in software development. It does not just mean **access to the source code**. The **distribution terms** of the software must not restrict any party from selling or giving it away and must allow **modifications and derived works**.

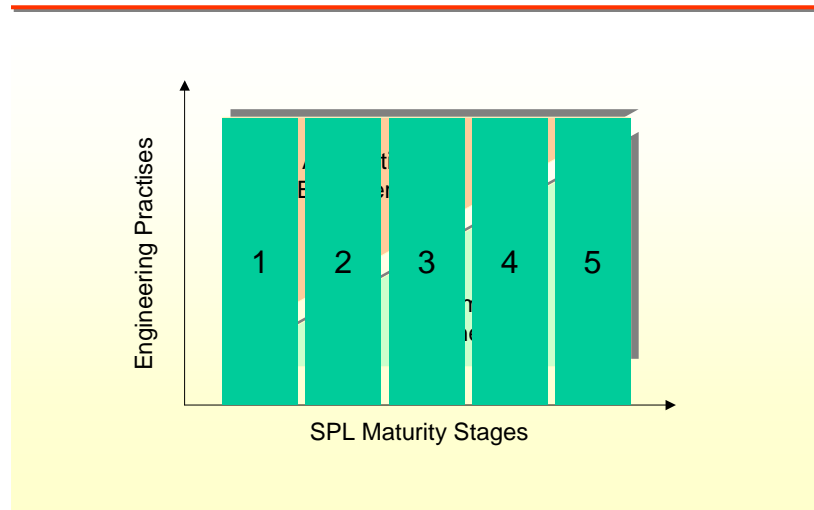
3

SPL Concept Chart



4

Maturity Stages



5

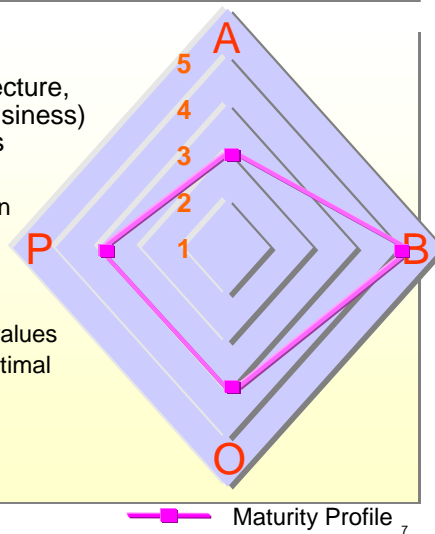
Maturity Levels

- Level 1: **Independent Product Development**
 - No Domain Engineering (only Application Engineering). Products are developed independently although ad-hoc reuse could exist
- Level 2: **Standardised Domain Independent Infrastructure**
 - Common software infrastructure (such as middleware or COTS) is defined nevertheless there is not formal reuse of domain specific assets
- Level 3: **Software Platform**
 - Domain commonality is captured and implemented in a software platform. This Platform is used for the different products. The platform could be configured nevertheless there is not variability support for product derivation
- Level 4: **Derivable Variant Products**
 - Domain commonality and variability is captured and a System Family architecture is specified. Domain assets include support for deriving products
- Level 5: **Automated Product Derivation**
 - Only Domain Engineering (no Application Engineering). Products can be derived automatically from the domain without product specific development

6

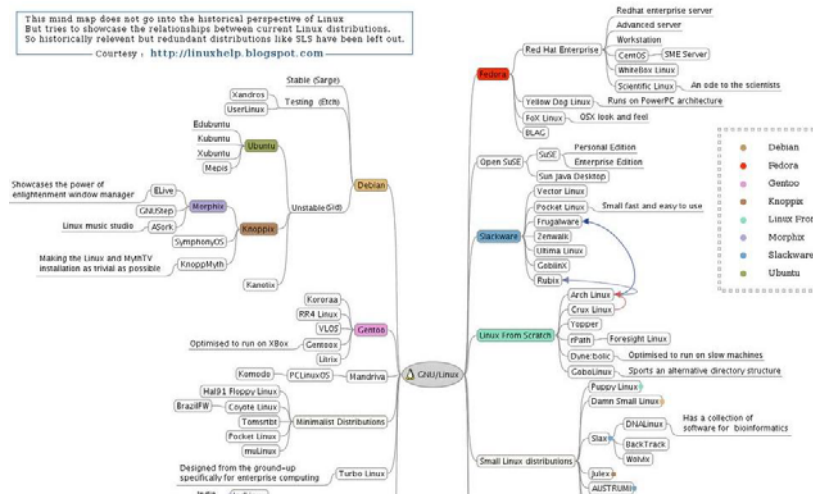
Maturity Levels/Dimensions

- Four FEF-dimensions (Architecture, Process, Organisation and Business) evaluated via aspects & levels
- Aspects
 - Main factors for the evaluation
- 5 Levels
 - Extent of aspect coverage
- Profile
 - Evaluation result of the four values
 - The maximum may not be optimal
 - A high level may involve
 - Costs
 - Overhead
 - Time



Maturity Profile 7

The GNU/Linux Case



The GNU/Linux Case

Last 12 months		Last 6 months		Last 3 months		Last 1 month	
1	Ubuntu	2735	1	Ubuntu	2784	1	Ubuntu
2	openSUSE	1858	2	openSUSE	1830	2	openSUSE
3	Mandriva	1302	3	Fedora	1417	3	Fedora
4	Fedora	1259	4	MEPIS	1010	4	MEPIS
5	MEPIS	1008	5	Mandriva	944	5	PCLinuxOS
6	Damn Small	854	6	Damn Small	851	6	Mandriva
7	Debian	785	7	Debian	776	7	Damn Small
8	KNOPPIX	717	8	PCLinuxOS	747	8	Debian
9	Gentoo	624	9	KNOPPIX	676	9	Stackware
10	PCLinuxOS	621	10	Gentoo	640	10	FreeSpire
11	Stackware	605	11	Stackware	581	11	Gentoo
12	FreeBSD	520	12	FreeBSD	521	12	Zenwalk
13	Kubuntu	496	13	Kubuntu	468	13	KNOPPIX
14	Vector	415	14	CentOS	441	14	Puppy
15	CentOS	388	15	SLAX	387	15	FreeBSD
16	KANOTIX	374	16	Zenwalk	387	16	SLAX
17	SLAX	359	17	Vector	383	17	CentOS
18	Xandros	348	18	Puppy	363	18	Xubuntu
19	Puppy	340	19	Xandros	344	19	Xandros
20	Zenwalk	310	20	KANOTIX	342	20	Vector
21	PC-BSD	307	21	Xubuntu	307	21	PC-BSD
22	Arch	275	22	Arch	280	22	KaliOS

<http://distrowatch.com/stats.php?section=popularity> (from 356 distributions)

9

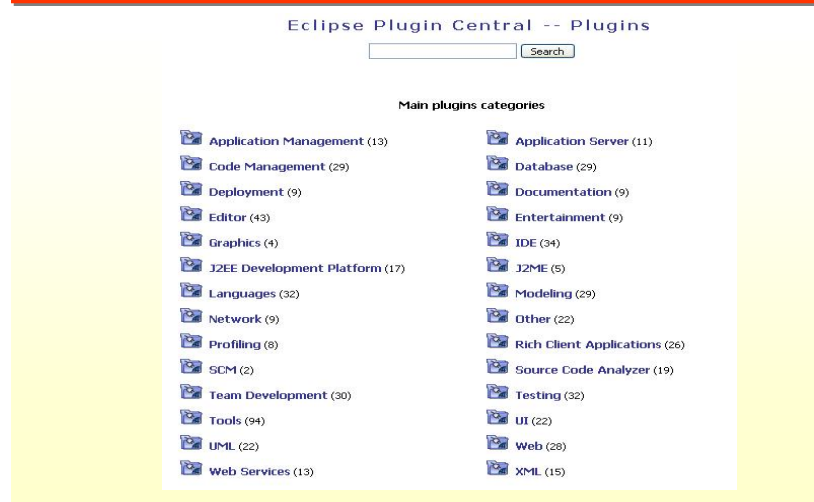
The Debian GNU/Linux – Variability Management Support

Debian GNU/Linux Package Management System: process of installing, updating and removing software (the system or specific packages) automating the retrieval, the configuration, the compiling (sometimes) and the installation

- Central repository of over 17,000 software packages
- Any number of additional repositories can be added
- Support for several sources, (ie. Internet, local network, or CD)
- CDs available for download for non-networked machines
- Control of preferences when conflicting sources
- Supports several packages .deb, rpm
- Runs on other operating systems such as Mac OS X
- Automatically fetches, configures and installs the dependencies
- Several front-ends package managers (Synaptic, aptitude, KPackage, Adept ..)

10

The Eclipse Case



11

The Eclipse Case

Most Active Plugins	Top Rated Plugins	New and Updated Plugins
WindowBuilder Pro - SWT/Swing Designer 68	MyEclipse Enterprise 9.3 (2656)	Spket IDE [Update]
Tomcat For Eclipse 39	Workbench 9.2 (4040)	Aptana [New]
UML Tool for Eclipse 36	Hibernate Synchronizer 9.1 (375)	Checkstyle Plug-in [Update]
Hibernate Synchronizer 36	CFEclipse 9.1 (1368)	Genady's RMI Plugin for E... [Update]
Subversive - SVN client 34	Azzurri Clay - Database Modeling in Eclipse 9.0 (517)	Wicked Shell [Update]
Jigloo SWT/Swing GUI Builder 33	WindowBuilder Pro - SWT/Swing Designer 8.9 (3215)	Crystal Reports for Eclipse [New]
Azzurri Clay - Database Modeling in Eclipse 27	Jigloo SWT/Swing GUI Builder 8.9 (2751)	TUM - a YUM like Eclipse ... [New]
CodePro AnalytiX 27	FreeMarker IDE 8.9 (206)	EclipseBugz [New]
EclipseUML Free Edition 22	Acceleo 8.9 (23)	DBEdit for PointBase [New]
MyEclipse Enterprise Workbench 16	Mylar 8.9 (46)	EchoStudio2 [New]
More Plugins...	More Plugins...	More Plugins...

12

The PHP Case

- **PEAR - PHP Extension and Application Repository** is a framework and distribution system for reusable PHP components.
- The components are provided in the form of so called "Packages".
 - The complete list can be browsed on-line
 - Includes on-line search facilities for packages through keywords
 - Provides a command-line interface that can be used to automatically install packages



- Detailed information is provided through on-line manual, FAQ and news. In case of needed support (general or a package in special), there is compiled a list of the available support resources
- Registering for developers for a PEAR website account is available

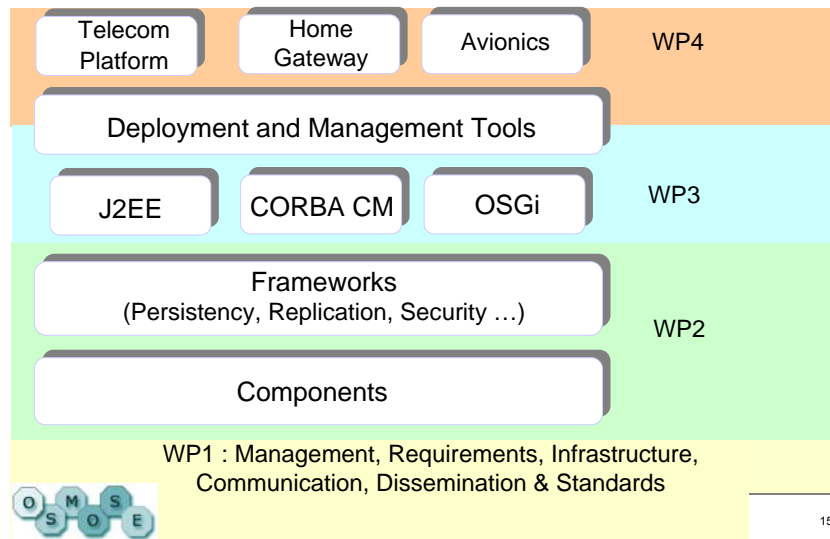
13

The PHP Case

Authentication (8) Auth , Auth_HTTP , Auth_PrefManager , Auth_PrefManager2 »	Benchmarking (1) Benchmark
Caching (2) Cache , Cache_Lite	Configuration (1) Config
Console (6) Console_Color , Console_Getopts , Console_Getopt , Console_Progressbar »	Database (30) DB , DBA , DBA_Relational , DB_adg »
Date and Time (3) Calendar , Date , Date_Holidays	Encryption (8) Crypt_Blowfish , Crypt_CBC , Crypt_CHAP , Crypt_HMAC »
Event (1) Event_Dispatcher	File Formats (23) Archive_Tar , Archive_Zip , Contact_AddressBook , Contact_Vcard_Build »
File System (4) File , File_Find , File_SearchReplace , VFS »	Gtk Components (4) Gtk_FileDrop , Gtk_ScrollingLabel , Gtk_Styled , Gtk_VarDump »
Gtk2 Components (6) Gtk2_EntryDialog , Gtk2_FileDrop , Gtk2_IndexedComboBox , Gtk2_PHPConfig »	HTML (34) HTML_AJAX , HTML_BBCodeParser , HTML_Common , HTML_Common2 »
HTTP (12) HTTP , HTTP_Client , HTTP_Download , HTTP_Header »	Images (17) Image_3D , Image_Barcode , Image_Canvas ,

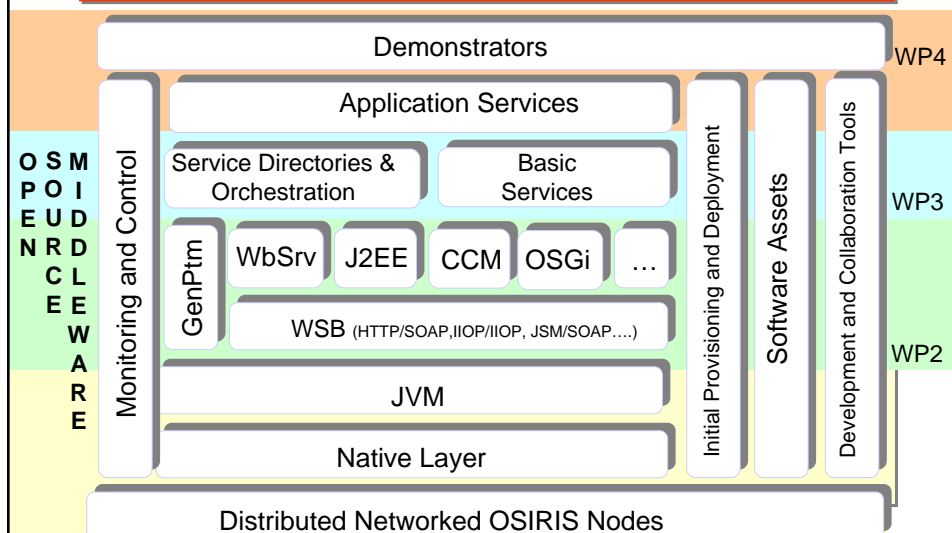
14

The OSMOSE R&D Project Case



15

The OSIRIS R&D Project Case



The COSIRIS Case



17

Conclusions

Product line practises can be identified in many relevant open source initiatives and communities. Frequently they have been key factors for the success of the projects.

The success of an Open Source project/initiative depends of the level of the use outside, the evolution of OSS towards an increasing adoption of product line practices is a direct consequence

OSS SPL strengths to achieve optimised reuse is reinforced through:

- OSS allows the "use of parts" which is a key enabler for a "product lining" process
- Broader potential inputs of needs (users involved in the development process)
- Flexibility for "branching" to address specific needs segments
- Developers not only from profit organisations (shared effort)
- Flexible (and real-time) cooperation across related initiatives
- Bottom-up approach, "evolving embedded in the social impact" of the network

18

References/additional information

- www.sei.cmu.edu/productlines/
- www.esi.es/en/Projects/Families/
- www.opensource.org
- www.debian.org
- www.linuxhelp.blogspot.com
- www.distrowatch.com
- www.eclipse.org
- www.apache.org
- www.pear.php.net/
- www.itea-osiris.org
- www.itea-cosi.org

19

Thank you for your attention!

Feature-Based Determination of Product Line Asset Types: In-house, COTS, or Open Source?¹

Jaejoon Lee and Dirk Muthig

Fraunhofer Institute for Experimental Software Engineering (IESE),
Fraunhofer Platz 1, 67663 Kaiserslautern, Germany
{[jaejoon.lee](mailto:jaejoon.lee@iese.fraunhofer.de), [dirk.muthig](mailto:dirk.muthig@iese.fraunhofer.de)}@iese.fraunhofer.de

1 Introduction

One important activity in product line engineering is product line production planning [1,2], during which stakeholders of a product line determine what and how product line assets are developed and used for product development. Moreover, decisions on which assets should be developed as in-house assets or purchased as COTS are made.

Recently, there have been increasing interests in using open source software for product development [3,4]. As pointed out in [3], it seems reasonable to make some common components of a product line as open source software or acquire them from exiting open source communities. However, it is still not clear what a “common” component means and how it can be identified. Suppose, for example, that a switching component for voice communications in a telephony product line is a common component and is required for every product of the product line. If the overall quality of a switching system mainly depends on the quality of the switching component, then it may be difficult to develop such components as open source software, as they may be developed based on lots of know-how of a company. Therefore, we need a systematic approach or guidelines that can be used to determine which product line assets to be developed as open source software.

In this position paper, we propose a feature-based approach to identifying product line assets and determining their development strategies during product line production planning. The approach is an extension of [5], and a feature model [6], which captures commonality and vari-

¹ This research is partially carried out in the Cluster of Excellence 'Dependable adaptive Systems and Mathematical Modeling' project, which is funded by the Program 'Wissen schafft Zukunft' of the Ministry of Science, Education, Research and Culture of Rhineland-Palatinate, Germany, AZ.: 15212-52 309-2/40 (30).

ability information of a product line, is used as primary input to the strategy selection.

2 Product Line Asset Type Determination

After features of a product line are identified, we group features into feature binding units, each of which includes features of the same binding time [5]. Then, we determine which features or feature binding units will be developed as core assets, product-specific assets, or open-source asset, or purchased as COTS. Therefore, for each feature or feature binding unit, its asset type (i.e., core asset, product-specific asset, open-source asset, or COTS) should be determined with consideration of the budget and time-to-market constraints and other business/technical considerations such as expected frequency of feature usage, estimated cost for development, availability of in-house expertise, and availability of open source software. (**Table 1** shows some of the identified product line assets of a Home Integration Systems (HIS) product line [2,5].)

Table 1 Identified Product Line Assets and Their Types

A representing name for a set of features with the same binding time	Constituent features	Frequency of feature usage	COTS availability / COTS price (which is compared to the estimated in-house development cost)	Open source software availability	Asset type
FIRE	Fire, Smoke, Smoke sensor, Sprinkler, ...	High	No / -	No	Core asset
FLOOD	Flood, Moisture, Moisture sensor, Alarm, ...	Medium	No / -	No	Core asset
MESSAGE	Message, Voice	Medium	Yes / Higher	No	Core asset
	Communication	Medium	Yes / Higher	Yes	Open-source asset
SECURITY	Security, Access-control, ...	Low	No / -	Yes	Product-specific asset
	Biometric	Low	Yes / Lower	No	COTS

For example, in the HIS product line, the *FIRE* feature binding unit has high frequency of usage (i.e., all products in the product line include it) and the estimated cost for development is low; the features of *FIRE* are identified as core assets. The *Communication* feature, however, has medium frequency of usage and is available as open source software; this feature is identified as an open-source asset, i.e., it will be acquired from an open source community when it is needed. For

another example, the *Biometric* feature, which is used to authenticate users, must be developed in a short period but in-house expertise and open-source software for the biometric technique is not available; COTS components will be purchased to realize this feature.

The considering factors (e.g., frequency of usage, etc.) and decision criteria for each type may vary from one organization to other. For example, if a company considers a feature as a ‘killing’ feature, then the company would not make the feature as open source software, even though similar features are available from an open source community. Also, some features may be open to an in-company-open-source community so that these features can be developed, improved, and shared by engineers belong to different departments/teams of the company.

3 Discussions

In this position paper, a feature-based approach to identifying product line assets and determining their development strategies during product line production planning is proposed. We claim that our approach can provide asset developers with an explicit way to identify core assets, and determine asset types with technical and business/management considerations. We believe that our approach makes it visible where to adapt open source software for product line asset development. We hope that this research will lead us to develop more detailed guidelines for open-source based development in the context of product line engineering.

4 References

1. Chastek, G., McGregor, J.D.: Guidelines for Developing a Product Line Production Plan, *Technical Report CMU/SEI-2002-TR-006*, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University (2002)
2. Chastek, G., Donohoe, P., McGregor, J.D.: Product Line Production Planning for the Home Integration System Example, *Technical Note CMU/SEI-2002-TN-029*, Pittsburgh, PA, Software Engineering Institute, Carnegie Mellon University (2002)
3. Co-development using inner & Open source in Software Intensive products (COSI) project, <http://itea-cosi.org/modules/wikimod/index.php?page=WikiHome>
4. Free/Open Source Research Community, <http://opensource.mit.edu>
5. Lee, J, Kang, K., Kim, S. A Feature-Based Approach to Product Line Production Planning, SPLC2004, LNCS 3154, pp. 183-196, 2004

6. Lee, K., Kang, K., Lee, J. Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. ICSR7, LNCS 2319. pp. 62-77, 2002

Feature-Oriented Determination of Product Line Asset Types: In-House, COTS, or Open Source? (Position Paper)

OSSPL - First International Workshop on Open
Source Software and Product Lines

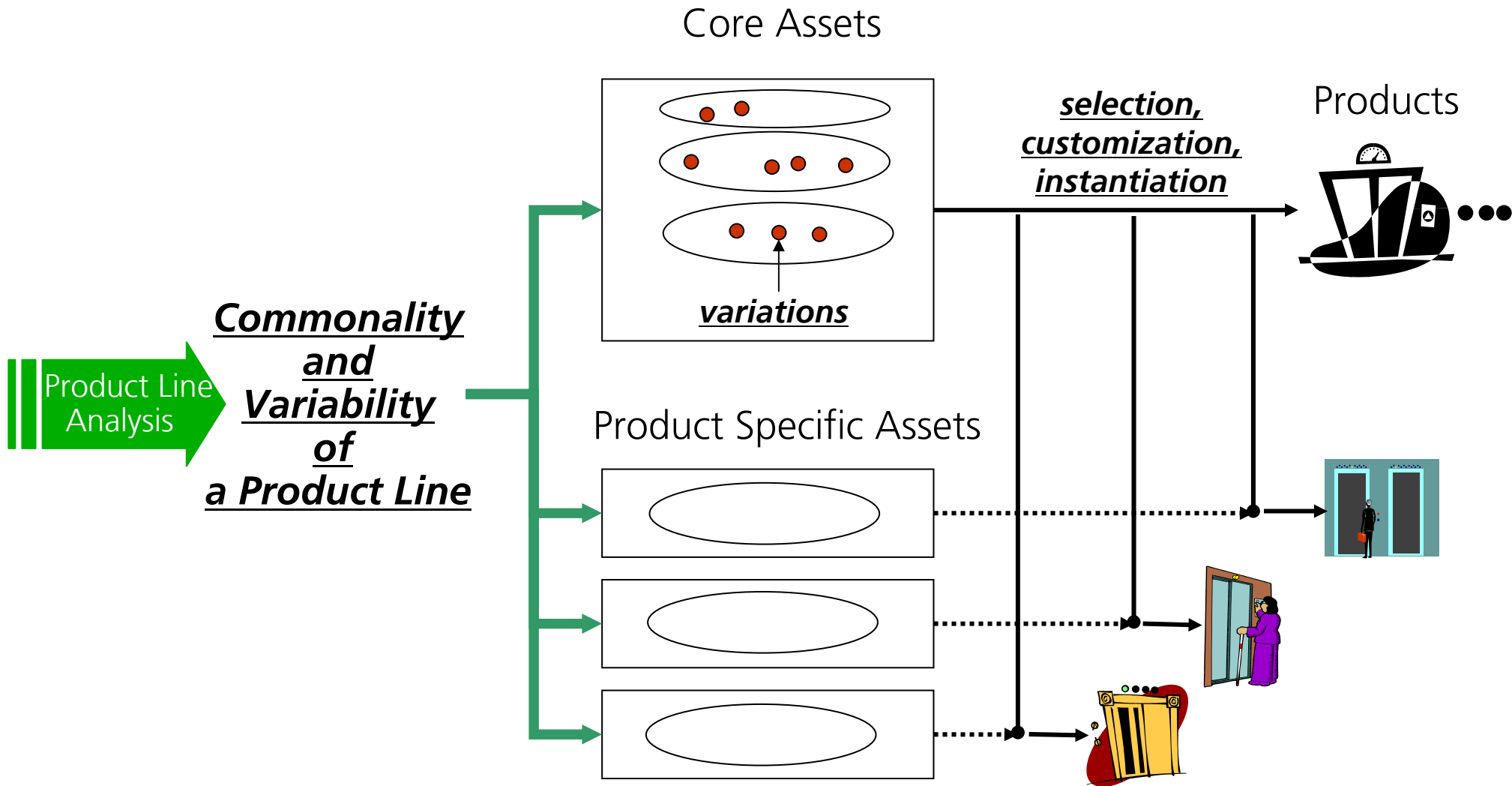
Jaejoon Lee and Dirk Muthig

jaejoon.lee@iese.fraunhofer.de

Tel.: 49-0631-6800 2289

Product Line Engineering

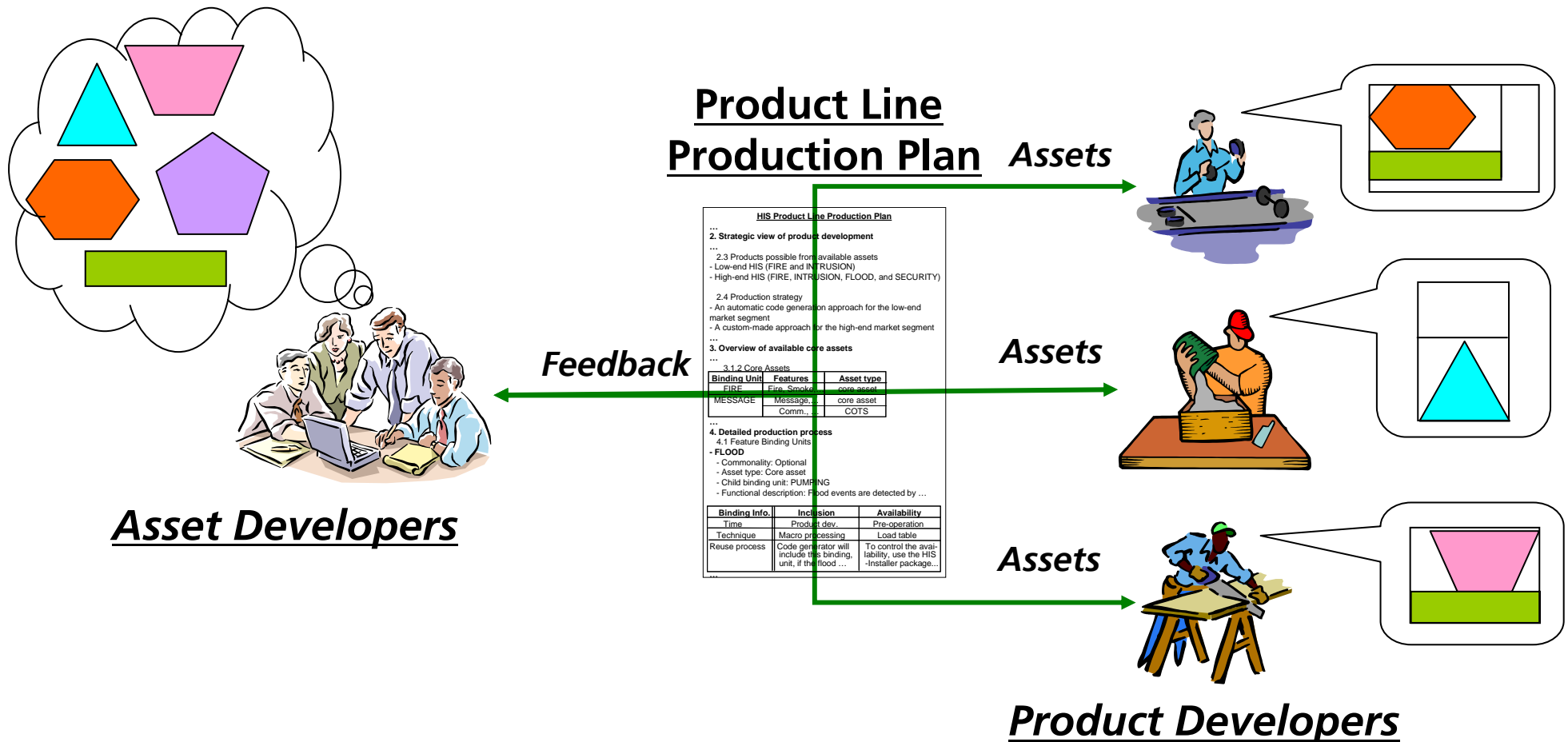
Introduction



Product Line Production Plan

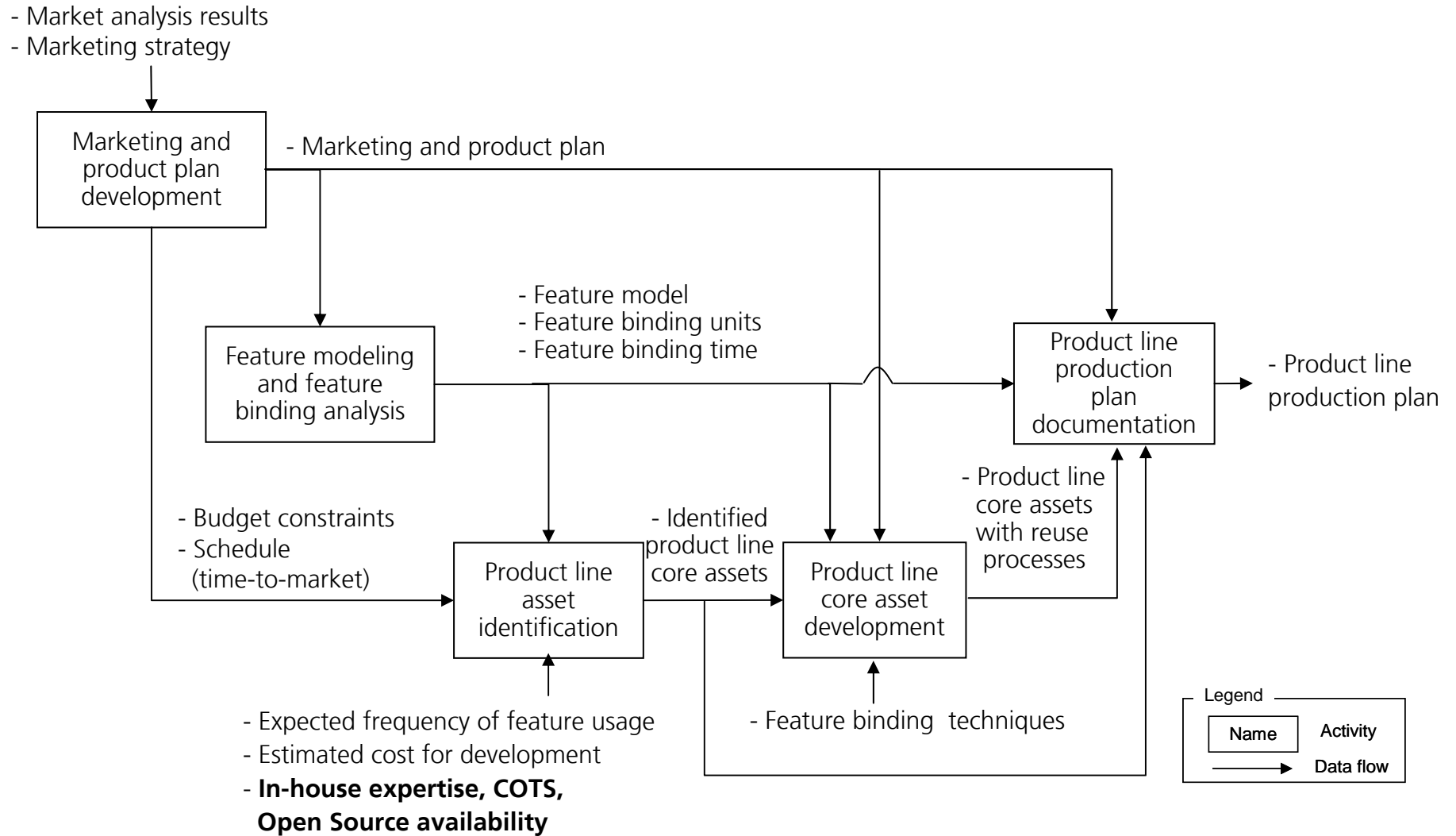
Introduction

A product line production plan, which describes **how the core assets are used to develop products**, has an important role in product line engineering as **a communication medium** between asset developers and product developers.



Product Line Production Planning Activities

Introduction

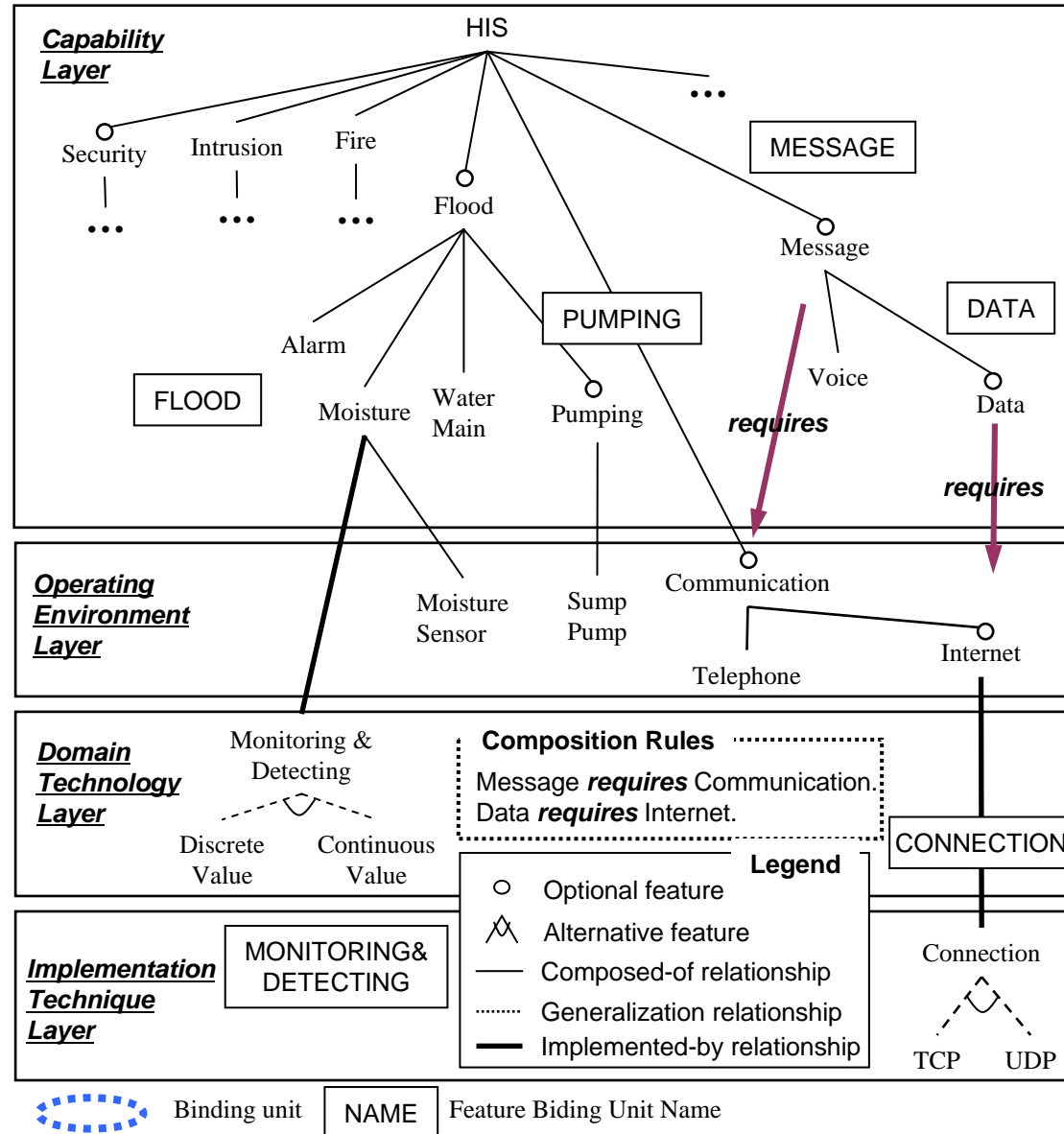


Marketing and Product Plan Development

Marketing and Product Plan for HIS product line			
Market segments		Office building (high-end product)	Household (low-end product)
Marketing plan	Need assessment	The customer's choices of features for high-end products are in the wide range of variability. Moreover, the <i>Security</i> feature has customer-specific requirements.	The customers are budget-conscious and they only require features that are essential for HIS products.
	Time-to-market	Less than six months	Less than three months
	Price range	To be a competitive product, the price should be less than 20,000 dollars.	Less than 1,000 dollars
	Marketing strategy (product delivery methods)	Develop and deliver a product for each customer	Prepackaged
Product plan	Product features	Fire, Intrusion, Flood, Security, and other customer specific features	Fire, Intrusion
	Quality attributes	Safety, Reliability, Scalability	Safety, Reliability, Usability

Feature Modeling

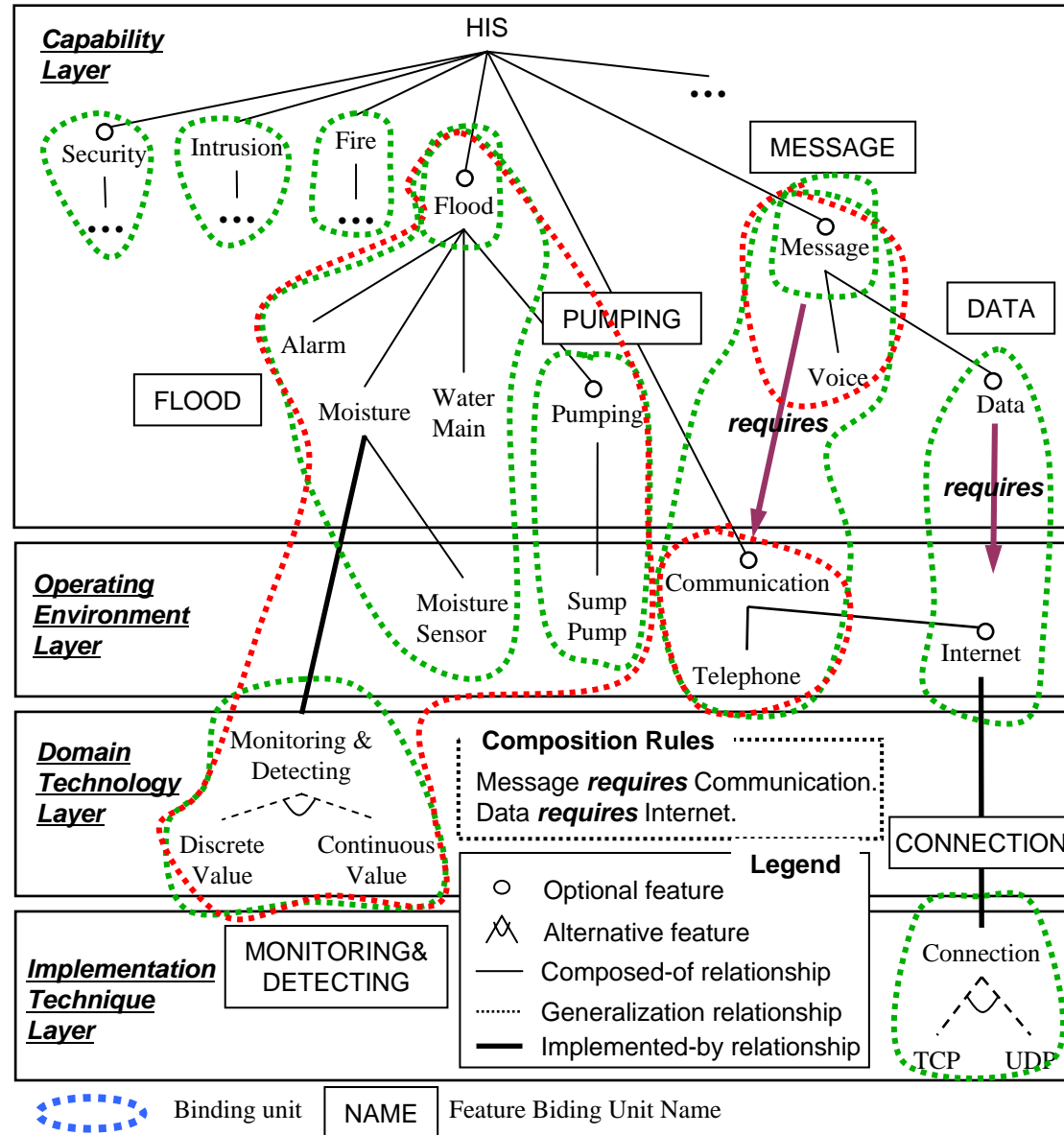
Feature Modeling and Binding Analysis



- What is a feature binding unit?
 - We define a feature binding unit as a set of features that are related to each other via compose-of, generalization/specialization, and implemented-by relationships and composition rules (i.e., require).
- Feature binding unit identification starts with identification of independently configurable service features.
 - A service feature represents a major functionality of a system and may be added to and removed from as a unit.
 - A service feature uses other features (e.g., operational, environmental, and implementation features) to function properly.
 - The constituents of a binding unit can be found by traversing the feature model along the feature relationships and composition rules.

Feature Binding Units

Feature Modeling and Binding Analysis



Asset Type Determination

Product Line Asset Identification

Feature binding unit	Constituent features	Frequency of feature usage	COTS or OS availability / COTS price (which is compared to the estimated in-house development cost)	Asset type
FIRE	Fire, Smoke, Smoke sensor, Sprinkler, ...	High	No / -	Core asset
FLOOD	Flood, Moisture, Moisture sensor, Alarm, ...	Medium	No / -	Core asset
MESSAGE	Message, Voice	Medium	Yes / Higher	Core asset
	Communication, Telephone	Medium	Yes / Lower	Open Source
SECURITY	Security, Access -control, ...	Low	No / -	Product specific asset
	Biometric	Low	Yes / Lower	COTS

- We introduced a feature-based approach to product line production planning and illustrated how a feature model and feature binding information are used to identify assets and develop a product line production plan.
- We believe that our approach can provide asset developers with an explicit way to identify and organize core assets, and determine asset types with technical and business/management considerations.
- Also, a production plan should be easily customized to a product-specific production plan so that units of product configurations as well as their integrating techniques can be managed consistently across a product line.

Open Source in the Software Product Line: An Inevitable Trajectory?

Pär J Ågerfalk¹, Brian Fitzgerald¹, Brian Lings², Björn Lundell^{2,1},
Liam O'Brien¹, and Steffen Thiel¹

¹ Lero – The Irish Software Engineering Research Centre, University of Limerick, Ireland.

² School of Humanities and Informatics, University of Skövde, Sweden.

1. Introduction

The open source software (OSS) landscape has changed dramatically in recent years. While OSS and its Free Software antecedent were largely driven by ideology and individual commitment, the main driving force of OSS today is commercialization and opportunities for inter-organizational collaboration (Fitzgerald, 2006). OSS is no longer primarily about enthusiasts contributing to SourceForge projects but increasingly about commercial organizations developing software in “co-opetitive ecosystems” (Ågerfalk et al., 2006), and many companies are now actively involved in Open Source (Lundell et al., 2006).

Commercial involvement in OSS projects is often based on a dual licensing model. In such cases, a free version of a product is typically released under an OSS licence while a possibly more advanced version with support and other bundled value adding services is released under a proprietary licence (IONA’s Celtix/Artix and MySQL are good examples). Another approach is to limit OSS engagement to development and maintenance of “commodity” software components, while developing business critical components under a proprietary licence to build on top of the OSS foundation. The latter approach is particularly interesting as it allows for organizations to collaborate on a common core and focus attention on value adding activities. This way, companies can share the cost of developing and maintaining common commodity components while still compete in the marketplace with respect to the complete end-product. In many embedded software domains, such as automotive and medical devices, this is a viable approach since software, although critical, is only a component in a larger mechanical or mechatronic system (Cosi, 2006).

Operating in a product oriented context, many organizations in these embedded software domains have successfully employed software product line (SPL) technology. With this backdrop, the aim of this paper is to pinpoint some of the issues in using OSS in software product lines by raising a set of questions as input to the sketching of an initial research agenda in this area. As an illustration, we will use a brief example from Certus Technology, a UK based company specializing in database solutions for, amongst others, the biological and medical domains.

2. Software Product Lines and OSS

The basic philosophy of SPL engineering is intra-organizational reuse through the explicitly planned exploitation of similarities between related products. This philosophy has been adopted by a wide variety of organizations and has proved to yield remarkable improvements in productivity, time-to-market, cost reduction, and product quality (Clements and Northrop, 2001).

A SPL is set of software-intensive systems sharing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way (Clements and Northrop, 2001). SPLs are typically developed in a staged process: a domain engineering and concurrent application engineering processes. During domain engineering a reusable platform is established. The platform consists of artefacts such as the

requirements specification, architecture documentation, design specification, implementation, and test cases. Domain engineering defines the commonality and variability between products in the product line and includes the implementation of an adequate product line architecture from which the commonalities can be exploited economically while retaining the ability to vary the products. In application engineering the product line products are then derived from the platform. Application engineering makes use of the pre-planned product line variability and ensures the correct binding of variation points to the specific needs of the customer products (Pohl, Böckle and van der Linden, 2005).

Nonetheless, introducing OSS components as core input assets in an SPL is far from straightforward. For example, even with “commercially friendly” OSS licences (such as LGPL, the “Library/Lesser GNU Public Licence”) available, how OSS components are demarcated and integrated in a product depends no longer only on traditional software engineering criteria but also on legal issues. The inclusion of an OSS component may, for example, require that the complete product be released under the same OSS licence. Also, SPL is often associated with model-based development while OSS components are often not well documented and reuse typically happens on a source-code level.

3. Brief Case Example

Certus Technology Associates is a small company specialising in the development of biomedical information systems, and technical and business IT (Pumphrey, 2006). It uses an SPL business model and an agile MDD development model. One product line supports QA in laboratories offering genetic testing for diseases. There are many such schemes around Europe, all with significant overlap in core functional requirements but all with customer-specific requirements also. One advantage of SPL is that a customer can perceive the advantage of sharing core functionality, as continual enhancement occurs because of the widened customer base.

Consistent with the agile philosophy, the company’s development is split over two sites in the North and South of England, for close-to-customer operation. Its tool base is largely OS, with all development done in a Linux environment. Core tools and technologies include CVS, AndroMDA, JBoss, PostgreSQL, Maven, and Forrest.

Critical to the company’s development model is the utilization of existing components wherever possible, and this increasingly means that OSS infrastructure solutions are looked at competitively with in-house solutions. All components are wrapped in a service layer, which is the target of the MDD transforms developed within the company. This degree of isolation future-proofs development investment, not only offering platform independence, but also reducing the potential for component lock-in. This is considered important whether the wrapped component is OSS or internal.

Choosing an OSS tool or component requires careful research, of product licensing and the quality of both the product and of the community developing it. The profusion of licensing agreements potentially makes this more of a problem than perhaps is necessary. However, Certus has not encountered any serious licensing issues to date. In fact, most of its investment is in tailoring its MDD infrastructure (so-called agile tools) and in customer services other than code delivery. Hence, opening up the code of a developed system would not necessarily be an issue; in fact, in some of its international collaborations with research groups it has offered to do just that.

For most commodity components there are still many competitors in the OS market place even after considering licensing, but it has been found from experience that these can quickly be reduced to a handful with fairly crude quality criteria. The support of a significant organization is one such criterion, and no dependence on other components which do not meet the criteria. Further, the use of an OSS component is clearly facilitated by its adherence to open, or at least transparent standards, so other criteria are based around standards. Adherence to relevant open data standards are of particular importance.

One open issue is whether, as OS components are incorporated, different architectural models can be easily accommodated. Clearly, with MDD it is a relatively orthogonal task to change architectures – particularly as a service model is used already to incorporate components. However, more formal architectural descriptions would protect some of the investment in developing transforms; and clearer OSS architectures are major advantages, as witnessed through the Eclipse project.

4. Open Research Questions

4.1 Licence forms and engineering vs legal decisions.

Several licence/legal questions arise in the context of using OSS components under a LGPL:

- What are the legal issues in using LGPL software components in commercial product lines?
- What would the situation be for the case of an LGPL component that is a core asset? Would all products developed using this core asset have to be under an LGPL licence?
- What would the situation be if the LGPL OSS component was a product-specific component? Would this limit the need for an LGPL licence to just that product?

4.2 Model-based development vs “code is king”

Several questions arise related to use of OSS components in product line development:

- What are the implications of using components within product line development where only the source code is available?
- What happens in cases where there is little or no documentation or high-level models of the components in existence?
- If product line engineering is based on models is there a need to develop models of OSS components and if so who develops and verifies the models?
- What control does an organization have over OSS components that they use in their product line development? Who is responsible for controlling changes and updates to OSS components?

4.3 Benefits of OSS components use in SPL:

There are some benefits to applying OSS in the context of product line development. OSS promotes open standards and spread of reference architectures, which could be important to facilitate software product lines. There are several questions that have to be addressed to fully understand the implications of using OSS:

- What areas within the software product line community would benefit from the use of OSS approaches?
- Would OSS-based software product line frameworks which allow people to build products specific to their needs be useful?

4.4 Issues in using OSS in SPL:

There are many issues related to quality requirements and trust that have to be addressed:

- What are the implications of using OSS components in the automotive domain and similar domains that have high safety, security and reliability requirements?
- Who would guarantee these requirements for OSS components? What level of analysis and testing is required to ensure that the components meet the requirements?
- Are organizations going to trust OSS components that are developed outside of their organizations? Currently trust is built up between organizations over many years of working together. If components can be developed and added to by anyone then with whom does an organization build trust? Who has ownership of the OSS components?
- Are there any organizations currently using OSS components in the development of business critical applications that have these requirements? What has their experience been in doing this and what lessons can be learned?

5. Conclusions

While there is potential for the use of OSS in software product lines there are many questions and issues that have to be addressed. Currently little research is being undertaken in this area, but if the OSS and SPL trajectories are to meet then many of these issues will have to be addressed.

References

- Ågerfalk, P. J., Fitzgerald, B., Holmström, H., Ó Conchúir, O. (2006) Open-Sourcing as Offshore Outsourcing Strategy, Proceedings of the 29th Information Systems Research Seminar in Scandinavia (IRIS 2006), Helsingør, Denmark, 12–15 August 2006.
- Clements, P., Northrop, L. (2001) Software Product Lines: Practices and Patterns, Addison-Wesley, 2001.
- Cosi (2006), ITEA COSI Project, www.itea-cosi.org, accessed 19 May 2006.
- Fitzgerald, B. (2006) The Transformation of Open Source Software, *MIS Quarterly*, Vol. 30, No. 3.
- Lundell, B., Lings, B. and Lindqvist, E. (2006) Perceptions and Uptake of Open Source in Swedish Organisations, In *The Second International Conference on Open Source Systems*, 8-10 June, Springer (to appear).
- Meyer, M. H., Lehnerd, A. P. (1997) The Power of Product Platforms. New York, NY: Free Press, 1997.
- Pohl, K., Böckle, G., van der Linden, F. (2005) Software Product Line Engineering: Foundations, Principles, and Techniques, 1st ed. New York, NY: Springer, 2005.
- Pumphrey, R. (2006) Distributed model-driven development in a small company using Open Source tools, In *An International Research and Practice Workshop: Distributed Development, Open Source & Industry – Opportunities, Suspicions & Strategies*, International CALIBRE Workshop, 3rd March 2006, University of Skövde, Skövde, Sweden.



lero

*THE IRISH SOFTWARE
ENGINEERING RESEARCH CENTRE*

Open Source in Software Product Line: An Inevitable Trajectory

Pär Ågerfalk, Brian Fitzgerald, Brian Lings, Björn Lundell,
Liam O'Brien, Steffen Thiel

Presentation by Liam O'Brien – Lero

Date in format 22 August 2006



Overview

- Open Source Software (OSS)
- OSS and Software Product Lines
- Open Research Questions

Open Source Software (OSS)

Driving force for OSS has changed from individual commitment to

- commercialization
- opportunities for inter-organizational collaboration

Approaches to OSS involvement

- Commercial involvement driven by dual licensing model:
 - Free version under OSS license
 - More advanced version under proprietary license
- Limit OSS engagement to “commodity” software components and develop business critical components under proprietary license on top of OSS foundation



OSS and Software Product Lines

Introducing OSS components as core/domain assets in an SPL is far from straightforward.

There are several open research questions that we are interested in addressing in areas such as:

- License forms and Engineering vs Legal Decisions
- Model-based development vs “code is king”
- Benefits in using OSS Components in SPLs
- OSS Components and Quality Requirements

Open Research Questions – 1

License forms and Engineering vs Legal Decisions

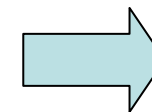
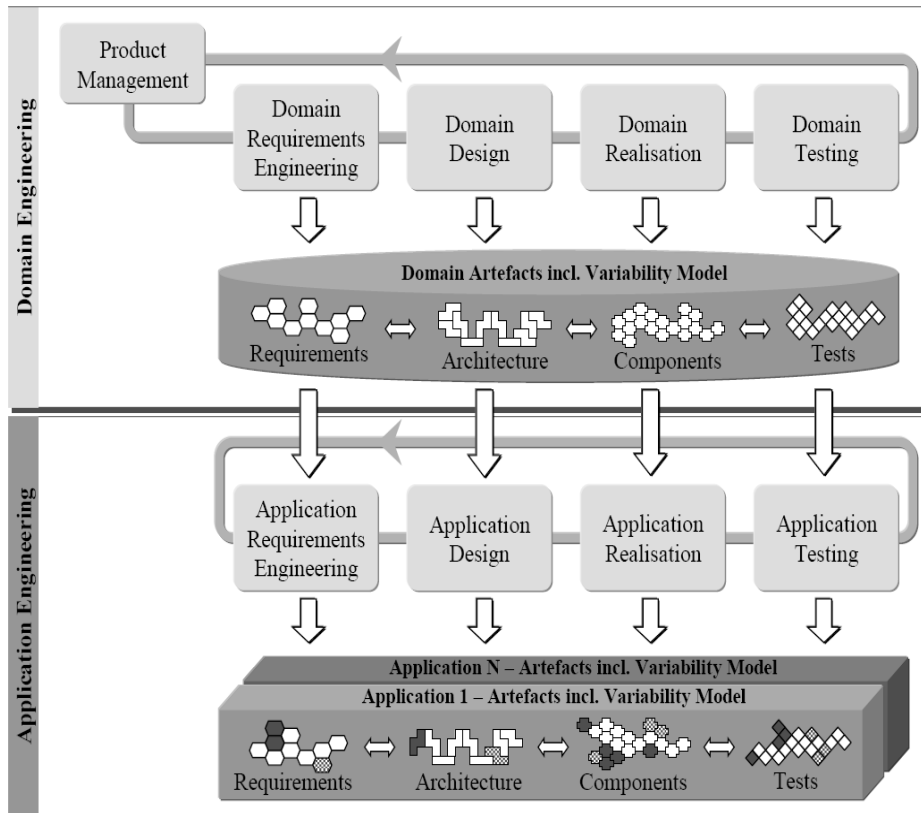
- What are the legal issues in using Library/Lesser GNU Public License (LGPL) software components in commercial PLs?
- What if an LGPL component is a core asset? Would all products developed using this core asset have to be under an LGPL licence?
- What if the LGPL OSS component is a product-specific component? Would this limit the need for an LGPL licence to just that product?

Open Research Questions – 2

Model-based development vs “code is king”

**Core Asset Development/
Domain Engineering**

**Application Engineering/
Product Development**



Products

Open Research Questions – 3

Model-based development vs “code is king” (cont’d)

- What are the implications of using components in PL development where only the source code is available?
- What happens if there is little or no documentation or high-level models of the components?
- Is there a need to develop models of OSS components and if so who develops and verifies the models?
- What control does an organization have over OSS components? Who controls changes and updates to OSS components?



Open Research Questions – 4

Benefits in using OSS Components in SPLs

OSS promotes open standards and the spread of reference architectures which facilitates SPL approaches

- What areas within the SPL community would benefit from the use of OSS approaches?
- Would OSS-based software product line frameworks, which allow people to build products specific to their needs, be useful?

Open Research Questions – 5

OSS Components and Quality Requirements

- What are the implications of using OSS components within SPLs in the automotive domain and similar domains that have high safety, security and reliability requirements?
- Who would guarantee such requirements for OSS components?
- What level of analysis and testing is required to ensure that the components meet the requirements?

Open Research Questions – 6

OSS Components and Quality Requirements (cont'd)

- Are organizations going to trust OSS components?
- Currently trust is built up between organizations over many years of working together. If components can be developed and added to by anyone then with whom does an organization build trust?
- Who has ownership of the OSS components?
- What are the legal/regulatory issues beyond just the licensing of OSS components?
- Are organizations currently using OSS components in the development of business critical applications that have these requirements? What has been their experience?



lero

THE IRISH SOFTWARE
ENGINEERING RESEARCH CENTRE

thank you

OSS Product Family Engineering

Jilles van Gurp

Nokia Research Center, Software and Application Technology Lab

jilles.vangurp AT nokia.com

Abstract

Open source projects have a characteristic set of development practices that is, in many cases, very different from the way many Software Product Families are developed. Yet the problems these practices are tailored for are very similar. This paper examines what these practices are and how they might be integrated into Software Product Family development.

1. Introduction

The notion of software reuse has been studied and practiced for decades. Over time, the attention in the technological dimension has shifted from subroutines to modules, frameworks and finally Software Product Families. In the organizational domain, focus has grown from code reuse by the author of the code to code reuse by others than the author of the code working on the same software, working in the same organization and finally between organizations. Software Product Family engineering is very much about intra-organizational reuse.

The open source movement was born out of a pragmatic need to share code among individuals. This need arose in the late sixties and early seventies when researchers started to share code for common assets such as compilers, system libraries and later operating systems such as UNIX. During the eighties, the practice of code sharing was given a legal framework in the form of license agreements such as the BSD license and the GNU public license. Finally, during the late nineties, when Linux emerged as a mainstream operating system, the term open source started to be used to refer to this practice of collaborative development, licensing and distribution of software.

Currently a wide variety of programs, components and frameworks is available under an open source license. Many software companies now depend on open source components for their core business. For example, the Gnu Compiler is widely used across the industry and crucial for many embedded system companies. Similarly, the Linux operating system kernel is used by many embedded systems companies. Even Microsoft is

known to use BSD licensed components in e.g. their network stack.

Open source components form a rapidly growing, shared repository from which, depending on the specific license, anybody can just take what they need and use it. Open source is very much about inter-organizational reuse.

It turns out that, as the scale of development is growing, inter-organizational reuse is increasingly important. Few organizations can afford to develop everything in house. For some years, COTS have been pushed as the solution for this problem. However, lack of source code, support, perfectly matching feature sets, and other factors have prevented the widespread adoption of COTS.

However, many organizations are now replacing their non-diversifying, in house developed components with open source, or even making their entire software available as open source (e.g. Sun). Open source is succeeding where COTS has failed.

Open source software is enabling interested parties to share code under a legal umbrella that sufficiently protects the rights of the using and producing parties. The use and production of OSS in the context of Software Product Families is both an obvious and inevitable solution to the problem that in house developed software is an increasingly smaller (relative, not absolute) portion of the total amount of software required. Eliminating non value adding development in Software Product Family development is key to reducing cost.

Arguably, open source development and Software Product Family development can claim to represent the two most successful strategies for reusing software. This position paper explores several of the practices common in open source communities with examples from three major open source projects (Eclipse, Mozilla, and Linux). Additionally some discussion is presented on how these practices may be applied to Software Product Family development.

1.1 Remainder of this paper

The rest of this paper consists of three parts. First (section 2) we characterize more precisely what we

understand the OSS development practice to be. Then we illustrate this with three open source projects: Eclipse, Mozilla, and Linux. Finally, we reflect in section 4 on how the identified practices could be integrated into Software Product Family development and we conclude our paper in section 5.

2. OSS development practice

Open source in the narrow definition refers only to the license used to make the software available. As such, the use of open source is completely orthogonal to the use of Software Product Family development practices (i.e. one could develop a Software Product Family using the conventional methods for doing so and then make the resulting software available as open source). However, in its wider definition it may also be understood to include a set of development practices and a certain style of development that is very different from the way Software Product Families are developed by many organizations. In this section, several of these practices are discussed.

2.1 Communication

Many open source projects are developed by people that are geographically distributed, may be in different time-zones and work for different organizations. Consequently, many forms of communication that are common in enterprises such as phone calls, face to face meetings are impractical. Additionally, the practice of one individual (a.k.a. the boss) telling other individuals what to do is not that common. Decisions are based primarily on consensus rather than authority.

In the open source community, email and IRC are the primary means of communication rather than face to face meetings. Technical discussions are preferably conducted on mailing lists which are generally archived for future reference. IRC or similar instant messaging tools are used for a more direct style of communication. These conversations tend to be less formal and they are generally not archived. Cases are known of OSS developers sending each other emails while sitting at the same table for the purpose of archiving the discussion or simply conducting it in public.

2.2 Tool centric development

A key characteristic of open source development is that open source projects are organized around a set of enabling tools. Generally, these tools (in addition to the usual development tools such as compilers and IDE's) include:

- A version management system. CVS is historically popular in open source projects but is now rapidly being replaced by the much more modern Subversion (e.g. the Apache Foundation and Sourceforge use Subversion nowadays).

- A bug tracking system. Bug tracking systems are commonly used both for tracking bugs, requirements and even project planning. Many open source projects require any change committed to the version management system to be related to a bug or issue in the bug tracking system.
- WIKI's are increasingly popular for document management. Particularly end user documentation, development documentation and project documentation (e.g. roadmaps) tend to be maintained in WIKI's.
- Build and integration tools (e.g. maven, ant, Make). Many open source projects depend on automated builds, integration and testing tools for receiving feedback about project progress and status.

Open source development is necessarily tool centric because its developers are generally distributed geographically. The tools are effectively their only interface to the project. Consequently, development practices that are incompatible with this interface are rarely found in open source projects. It therefore is quite common for OSS projects to not have explicit design documentation; use case diagrams or even an architecture design phase. However, that does not mean that such projects do not have architecture, design and requirements.

Instead, these assets, insofar deemed relevant by the developers, exist in the tools. Use cases are rare but detailed requirements and requirements change requests are managed through the bug tracking system. Architecture documentation is generally lacking but then the audience for such documentation is not necessarily the developers either in organizations that do write architecture documentation.

2.3 Strong code ownership

Though the source code of an OSS project may (legally) be modified and redistributed by anyone the actual occurrence of someone taking open source software, modifying it and distributing it independently from the original (a practice known as forking) is quite rare. Generally, open source projects have strong ownership with a small group of developers coordinating and guarding the development.

Source code ownership is governed through version repository access rights. Typically, a limited set of individuals has the right to make changes to particular directories in the version management system. It is also quite common that approval of key individuals is needed to make any kind of change. The strong ownership enforces code reviews take place and that changes are tested properly.

2.4 Technical roadmap

Unlike commercial software development where managers, customers and other stakeholders determine what is developed, the evolution of open source software projects is primarily determined by:

- Developer interest. Developers generally prioritize features that they are personally interested in.
- Corporate funding. Most large open source projects are developed by developers who are paid to work on the project. Of course, the reason they are paid is that their companies have a strategic interest in the project and presumably want to influence the direction of the project.
- Project organization. Many open source projects are led by a small group of, more or less, independently operating individuals whose personal vision strongly influences technical decisions made in the project.

In order to prioritize features or make major technical changes to the software, interested parties need to work in this structure. They need to convince whatever individual is in charge that the suggested change is a good one; generate interest among developers to actually get the change implemented and maybe arrange some funding to allow developers to work on the change.

2.5 Quality management

A consequence of developers being in charge of the technical roadmap is that generally developers prioritize quality attributes that interest them. For example, the open BSD project has a strong security focus. The open BSD lead developers all have strong engineering backgrounds in security related matters. Its products are generally considered to be of exceptional quality in this regard (e.g. open SSH or the open BSD kernel). Additionally, any issues related security are handled swiftly once the developers are notified of them. Other quality issues outside the scope of the developer's interest receive much less attention (for example, usability is often sacrificed in favor of configurability).

Similar to the technical roadmap, the quality management can be influenced through funding, argumentation, etc.

2.6 Release Management

Release management is the process of converting source code in the version management into a stable, well tested software package that can be distributed to end users. Many open source projects have well defined processes for producing a release. Generally, there are a few differences with comparable processes in commercial projects:

- The software is released when it is 'done'. This moment is generally agreed on either by leading individuals in the process or by consensus. Despite this, many open source projects try to follow date driven roadmaps where milestones and releases are planned to occur. In commercial projects, such deadlines tend to be much harder and inflexible, however.
- The software release is preceded by a series of public alpha, beta and release candidate milestones. During this period, interested third parties not taking part in the development test the software and provide feedback. Though technically it is possible for them to use so-called nightly builds straight from the version management repository, few people outside the developer community are actually willing to take the risk.
- Because the eventual release is scrutinized in public, quality tends to be high (in so far of interest to the involved users and developers).

Especially for large open source projects, the release process tends to be well defined.

3. Examples

To illustrate the claims made in the previous section, we present three case studies which highlight all of the practices mentioned in three large open source projects with solid reputations in the software industry.

3.1 Eclipse

The Eclipse foundation is responsible for the development of the Eclipse IDE and a rapidly growing number of associated software packages (plugins). Originally, the Eclipse source code was contributed by IBM who still provides a significant amount of funding. However, the Eclipse foundation is now an independent organization that oversees the development. In addition, other companies, including competitors of IBM, now contribute funding and development resources to the foundation.

Communication. Communication happens primarily through email, IRC, the Bugzilla bug tracking system, the WIKI website, mailing lists and the Eclipse.org website. Eclipse developers are distributed across the globe and mostly employed by (competing) corporations (e.g. BEA and IBM).

Tooling. Eclipse source code is maintained in a CVS repository, Bugzilla is used as the bug tracking system and project documentation is divided between the Eclipse.org website and the Eclipse WIKI. Additionally there are several mailing lists both for end users and developers.

Code ownership. The Eclipse foundation restricts write access to their code repository. Generally, the

process for contributors involves contacting a so-called committer for making a particular change. Typically, components have an owner and multiple committers. The role of the owner is to coordinate the work on that component. When receiving an external contribution, the committer either commits the change or (limited) commit rights are given to the new contributor [2]. A key element in the process is assuring that the contribution conforms to the legal framework which involves topics as copyrights, the license, patents and export rules concerning cryptography technology [1]. All contributions must be traceable and accountable. Procedures like this are common to many open source projects.

Technical roadmap. The Eclipse project strongly depends on development resources contributed by various software companies. Those companies have a strong influence on what is developed. A good example is the web tools project, a massive undertaking by IBM, BEA and several other companies to create a set of J2EE development plugins for the Eclipse IDE. Over the course of 1.5 year, this project went through a set of planned milestones with specified sets of features and managed to release a feature complete 0.7 release for the Eclipse IDE 3.1 release, a more mature 1.0 release half a year later and recently a 1.5 release. The input for the project was a set of contributed development tools from various vendors and a number of (public) J2EE specifications that these companies wanted to have support for.

Quality management. The core Eclipse project has seen many changes related to improving performance and memory usage in its recent versions. To accomplish this, the automated test suites that are run on nightly builds of the Eclipse software have been extended with tests to measure specific scenarios. Furthermore, target performance numbers have been defined and cases where performance targets are not met are treated as bugs. The test reports for the nightly builds and release candidates of the Eclipse 3.2 release list performance numbers relative to the 3.1 release. Each case where performance decreases is treated as a regression. Aside from performance, the nightly builds also include a large number of unit tests (thousands). Specific quality issues either identified automatically or through testing, are reported in the bug tracking tool.

Release management. The Eclipse project has well defined release cycles which are beyond the scope of this article to discuss in full. The key philosophy of the Eclipse release process is to be automation centric. The release practice is outlined in a FAQ [3] that provides answers on mostly technical topics such as how to set up the test suite; how to integrate components into the

build process, etc. Effectively, the build infrastructure implements and enforces a sophisticated system of checks and balances that ensures that produced releases meet predefined criteria.

In addition to the technical constraints, the release process is complemented by communication and coordination from project leads through the mailing list on such topics as roadmaps, schedules, code freezes, test plans, etc.

3.2 Mozilla

The Mozilla foundation which oversees the development of Firefox browser, the Thunderbird mail client and a number of related software projects has a similar history to the Eclipse foundation. Originally, the Mozilla browser was contributed by Netscape. The company Netscape has since been absorbed into AOL and was eventually liquidated. During this process, the Mozilla foundation was created which still employs some former Netscape employees but also a growing number of new employees. Similar to Eclipse, the Mozilla foundation receives corporate funding from a number of companies that have an interest in the continued existence of the Mozilla technology.

Communication. Similar to the Eclipse developers, the Mozilla developers are also distributed globally. In addition, they use similar communication tools.

Tooling. Similar to Eclipse, Mozilla development is very tool centric. In addition, Mozilla is famous for inventing its own tools. For example, Bugzilla is one of the software projects that is maintained by the Mozilla foundation. Other tools created by Mozilla include Bonsai for examining the CVS history, LXR for browsing the cross referenced source code through a web site, Tinderbox for monitoring the build process and Litmus for managing and running automated tests on Firefox. Many of these tools, most notably Bonsai and Bugzilla, have been adopted by other projects and have even been integrated into commercial tools.

Code ownership. The Mozilla project features strong code ownership. In practice, this means that every patch must be reviewed and approved by a component owner before being committed [4]. Component owners are generally either Mozilla foundation employees or individuals with a long history in the project employed by one of the high profile donating corporations (e.g. The Firefox project leader Ben Goodger is a Google employee).

Technical roadmap. Firefox development takes place in the context of a roadmap which is updated at regular intervals (once or twice per year). The roadmap features milestones and releases with a list of features and corresponding Bugzilla ids. While the foundation strives to release according to the roadmap, the Mozilla

release policy in practice appears to be much less rigid than e.g. the Eclipse project. Often releases are delayed for weeks or even months (as long as is needed). Also new milestones may be inserted into the roadmap. Finally, the roadmap acts mostly as a guide rather than a complete functional specification. It contains what the project leaders believe are relevant features to work on. Input for this comes from the mailing lists, the WIKI and IRC discussions.

Quality management. The Mozilla project has a number of quality attributes that are explicitly managed:

- **Code quality.** As part of the commit process, each patch is attached to a bugreport in Bugzilla that describes the problem and solution(s). Before being committed, the patch is reviewed and super reviewed.
- **Correctness.** The Firefox browser implements a large number of open standards. In addition to that it supports poorly defined incorrect interpretations of these standards (a.k.a. the quirks mode) of other browsers. Testing for compliance therefore is an extremely complicated affair that is supported by manual testing, half automated tests (a.k.a. smoke tests) and fully automated tests (e.g. using the Litmus tool).
- **Performance.** Similar to correctness, performance is explicitly managed through testing (automated and manually).
- **Security.** Browser security is of extreme importance to end users. In addition, it is a sensitive topic. Therefore, the Mozilla project has well defined procedures for reporting, solving and publicizing security issues. Additionally, recent versions of the Firefox browser include an auto update feature to stimulate rapid deployment of security related bug fixes.

Release management. The Mozilla foundation manages and oversees the release process. Generally the process involves a number of alpha release milestones followed by more or less feature complete beta releases (typically two) and finally followed by a series of release candidates (as many as is needed). During this process, the rules for committing changes become stronger. During the release process, no changes are committed before being extensively discussed by project leads. Additionally each of the milestone and beta releases has a mini release process which involves a few days of testing candidate builds and restricting commit access to the CVS repository.

3.3 Linux

The Linux kernel development is overseen by its inventor Linus Torvalds. The style in which he

manages the project is very different from Mozilla and Eclipse though still tool centric. Unlike the former two projects, Linux development is traditionally much more fragmented among thousands of developers and hundreds of contributing companies. In a recent interview, Torvalds estimates that there are around 50 developers he communicates with directly and he estimates that through them he is in contact with approximately 5000 kernel developers [5].

Communication. Linux kernel developers rely very much on mailing lists and private mail exchanges (or IRC conversations). Linus Torvalds style of leadership has often been referred to as that of a benevolent dictator: ultimately, he is the one who takes important decisions though in practice this responsibility is delegated to trusted individuals.

Tooling. The central leadership is also reflected in how the tooling works. The Linux project recently switched from using Bitkeeper to its own developed tool Git. Both are so-called distributed version management tools. Rather than pushing changes to a central repository, the lead kernel developers pull changes into their private repositories either by accepting patches from a mailinglist or by updating from somebody else's repository. The repositories available at kernel.org are read only for most developers. They are merely the places where lead developers publish their approved change sets from their private repositories. Other tools used in Linux development include Bugzilla and various news groups. However, email remains the most important tool.

The use of a distributed version management system on a large scale is a recent innovation that no doubt will be followed up by adoption in other projects as well. It has proven to be an effective way to orchestrate the development on a large software system with many active developers.

Code ownership. As the central leadership suggests, code ownership is very strong in the Linux project. To get a change committed in the Linux kernel the associated patch needs to be communicated by email to the relevant people that have the right to approve the change. Eventually the change will find its way to Linus Torvalds, who, after assuring that everything has been properly reviewed, approved and tested may or may not include the change at his discretion.

Technical roadmap. Linux development tends to be more anarchistic than Mozilla or Eclipse development. Essentially, there is no centrally maintained roadmap. Development consists of many subgroups working on e.g. drivers, new memory management routines, etc. Major versions of the kernel usually include some re-architecting as well. E.g., the current 2.6 version

included features to allow the kernel to scale better on distributed systems.

Quality management. Stability, performance, security, modularity are all important themes in the development of the Linux kernel. Linux is used on many mission critical servers, mainframes and desktops. Additionally it is embedded in devices. Therefore, all these quality attributes are critical. Despite this, there are few quality management tools or processes in the Linux development. Code review and testing by users seems to be the main way of controlling quality. The reason for this is that the Linux development and user community is extremely diverse. There are thousands of developers working on or depending on the latest kernel sources. Testing happens in a distributed fashion on a wide variety of devices by a wide variety of parties with a wide variety of interests (device drivers, processor architectures, file system development, real time behavior, ...). The testers include: individual desktop users, hardware vendors, Linux distribution vendors, and the developers themselves.

Release management. In principle, Linus Torvalds is the one who declares a release. His principle over the years has always been that "it's done when it's done and not sooner". Despite this, the process seems to involve a number of stages spanning several months during which progressively less changes are accepted and testing efforts are increased.

4. Improving SPF development practice

Open source development as outlined above represent the state of the art in the way software developers believe software should be developed. If left to their own devices, this is how they self organize.

In many respects that is very similar to how development takes place (or should take place) in traditional closed source environments. However, there are some differences. In this section, we examine how the practices discussed above may be integrated into Software Product Family development practice.

4.1 Communication

Software Product Family developers are faced with similar communication challenges as open source developers. Often development teams are large, may be geographically distributed and composed of different organizational entities. Additionally, a growing need for accountability (e.g. for legal reasons) makes it obvious that the solution to this communication challenge also needs to be similar (see e.g. [1] for the process for accepting contributions in the Eclipse project).

Additionally, many multinational companies are so large that the challenge of getting their developers to work together on projects requires a more or less

similar communication infrastructure to the OSS style of communicating. Email remains an important tool across such organizations. Consequently, many of the open source communication tools are already finding their way into the corporate world (e.g. WIKI's, bug tracking tools and instant messaging tools).

A problem remains that, in general, only the developing part of such companies uses such tools. Senior managers, sales departments and other parts of the organization are not using the same tools for communicating. This creates a conceptual gap between the development reality on the work floor and the management reality. The alternate management reality is an appealing ground to make important decisions that have major effect on the development reality: especially for people who should not be making those decisions.

The term slideware refers to software entities that only exist in PowerPoint slides and not in the relevant development tools [6]. The problem with slideware is that it doesn't have any corresponding representation in the development communication infrastructure. Once it does, it ceases to be slideware. Until it does, it does not exist. Problems arise when slideware fails to materialize in a timely fashion.

In open source projects, slideware does not exist. New requirements for features become WIKI documents. WIKI documents become bug reports. Bug reports are commented on and eventually are closed with either a reference to a patch or CVS commit or a message as to why the particular feature is no longer relevant.

4.2 Tooling

In a corporate setting tooling tends to be better (e.g. the use of commercial version management or document management tools is common; additionally expensive modeling tools, IDE's and other tools may be used). However, over the years, the open source community has produced its own set of tools that meets its requirements. Such tools include everything from the, now, industry standard GCC compiler, Bugzilla, the Mozilla tool chain outlined above to sophisticated distributed version management systems (e.g. Subversion and GIT). Many commercial development tools are simply based on open source components and either add value through support or by adding specific features.

A key feature of tools in the open source development community is that they are developer centric. Their primary objective is to make the developer's work (i.e. developing software) easier. Many tools used in Software Product Family development on the other hand are not developer centric (or even developer friendly). For example, many variability management

tools are aimed at requirements engineers or even sales departments; many architecture modeling tools are used by senior architects to communicate to their managers; UML modeling tools are used to document already developed software; model driven architecture tools are aimed at the consumers of the software (i.e. the people that design products) rather than the developers of the composed software. Often bureaucracy in the form of heavy processes is needed to enforce the proper use of such tools.

A key lesson that may be drawn from the open source style of tooling is that in order to be effective, tools need to integrate into other tools. The set of tools create their own reality in which the developer is active. Anything outside this reality integrates poorly into the communication structure used and quickly becomes irrelevant (for the developers). OSS developers seem to have little or no need for such tools and yet manage to scale development to impressive levels of scale, speed and quality.

A good example of an integrated tool from the OSS community is Bugzilla. In both the Mozilla and Eclipse projects (and in many other places) this tool is not only used for bug tracking but also for requirements engineering, release management and even process improvement. The imposed reality in these projects is that any change to anything is communicated through and documented in Bugzilla. Bugzilla in turn is integrated with email (notifications) and version management systems.

Many Software Product Family tools are plagued by a lack of integration. Design documentation tends to be incomplete (or non existent) because the document management system is not part of the development environment, variability management tools depend on extensive manual updates to stay in sync with source code level changes; requirement specifications need to be continuously validated and verified. Successful examples do exist however. For example, KOALA, the architecture description language used by Philips integrates with the build system and design [7]. The COVAMOF variability management tool proposed by Sinnema et al. integrates into visual studio [8].

A second problem with such tools is that they are not general purpose. This poses problems when product families become product populations and different sets of incompatible tools become obstacles that need to be bridged. A key driver for growth in the OSS communities is that everybody uses the same or similar tools. This lowers the barrier of entry for new contributors. The fact that the tools are comparatively primitive is compensated by the fact that everybody knows how to work with them. Similar consolidation in

SPF development tools is required as SPF are increasingly complemented with third party provided software components (open source and closed source).

4.3 Code ownership

While corporate interest in many OSS projects is huge (also financially), OSS projects tend to be self organizing in the sense that all important decisions are made by developers rather than managers. The relevance of opinions of individual developers is strongly related to their level of (technical) contribution to the project (within the Eclipse project this is called a meritocracy).

A key issue in Software Product Family developing companies, which are generally not organized as meritocracies, is that decisions are made based on authority, rank and status in the company. Especially when difficult technical decisions are taken, this may not be the most optimal strategy since it is not common that the person with the most authority also has the most technical competence. At best, he or she has the wit to trust the judgment of the competent subordinates who should be making the decision. In other words, important technical decisions are routinely taken by the wrong people; influenced by the wrong motives (e.g. short term market interests vs. quality) and misguided by a lack of relevant knowledge of domain, technology and software design.

To counter this problem, many organizations organize their Software Product Family development as a separate organizational entity to shield it from the short term interests that are present in depending organizational units that develop the products [9]. Despite this, influence of the other organizational units remains high through e.g. funding, upper management etc.

The conflict between the long term technical roadmap (development), the short term market interests (sales) and the long term market perspective (marketing) poses a risk to the long term technical health of the software. Open source projects solve this by being autonomous. That does not mean they are not affected by the market. Through funding, donations and man power companies exert influence over the technical roadmap, short term interests etc. For example, IBM maintains a strong influence in the Eclipse project (and in fact many other open source projects that are of strategic interest to them). While they cannot dictate their changes, they have a very strong influence on the technical direction of their project simply by funding development of features and components that are of interest to them.

4.4 Technical Roadmap

Software Product Families are a key investment for the companies that own them. Naturally, these companies

wish to have a strong influence on the roadmap of their product lines. As outlined above under code ownership, this can easily lead to a situation where decisions are made by the wrong people. A real problem is that these roadmaps tend to focus on functional requirements only (because that is what is marketable to customers).

For example refactoring is unlikely to feature in a SPF roadmap. Yet, when looking at OSS projects, refactoring is often a driving force for major new releases. For example, the Eclipse project was refactored extensively between version 2 and 3. In addition, the subsequent 3.1 and upcoming 3.2 have seen additional refactoring work done. This has led to major improvements in performance, usability and flexibility (which was the main reason for the refactoring). Additionally, it has enabled the development of new features. The Linux kernel has seen large portions of its code being rewritten several times in its 1.0, 1.2, 2.0, 2.2, 2.4 and 2.6 incarnations. Firefox started out as an attempt by a small group of individual Mozilla developers to refactor/rewrite the Mozilla user interface, against the explicit wishes of their AOL peers at the time. Firefox has since replaced Mozilla as the flagship product of the Mozilla foundation.

Refactoring is a good example of an activity that developers will put on a roadmap and companies will likely not until the need becomes obvious. Refactoring almost always conflicts with commercial product roadmaps and short term interests of companies.

A problem with OSS roadmaps is that they reflect what the developers would like to see done, which is not necessarily as important for end users or relevant for the companies financing the development. Clearly, this model is not applicable to commercial software development on Software Product Families. On the other hand, there is a much better understanding of the technical feasibility of requirements at the developer level than there is elsewhere in an organization. An SPF roadmap should be realistic in the sense that its requirements are technically feasible, desirable and in the sense that important development activities needed for maintaining or improving quality are covered.

4.5 Quality Management

Open source development relies on three powerful quality management tools: large scale testing by end users, code reviews and automated tests. Testing on a large scale may be impractical for some software product families. But both other approaches are not unique to the open source community and can and should be implemented in software product family development methodology (in so far that is not the case already).

What make code reviews particularly effective in open source communities is that they can block the commit of a change until the component owner decides that the quality of the commit is good enough. This aspect of code reviews is hard to duplicate in companies where the code reviewer generally has limited authority to block changes (especially if they address urgent issues through a quick hack). Automated tests and test driven development are also increasingly popular. For example, in earlier research we reported on the successful use of automated tests in improving quality in Baan ERP. Test driven development is a cornerstone of extreme programming [10].

4.6 Release Management

Depending on the number of customers for a particular piece of software, the release process can become quite sophisticated. For example releasing a new version of the Mozilla Firefox browser is a process that spans multiple months and involves exposing alpha, beta and release candidate versions to large groups of users and processing any feedback that comes back from these users. In Software Product Family development, the number of users is typically small. Despite this, it may be productive to have some form of release process in place. It also depends on the organizational model. If, as outlined above, the product family development is developed by a more or less independent organizational entity, it makes sense that the rest of the organization does not access the version repository directly and instead relies on properly packaged and tested releases provided by the product family developers. However, having no feedback from real users (i.e. the product developers) until after the release is likely to cause issues with respect to implemented requirements and faults that are discovered after the release.

The author's experience as the (ex) release manager of a Dutch content management Software Product Family suggests that a good strategy may be to expose increasingly large groups of internal developers to increasingly mature versions of the product. Combined with a transition period with e.g. bi weekly releases this ensures that feedback and development stability (for the product developers) are balanced. This is similar to the beta stage of many open source projects where typically third parties (at their own risk) get involved into testing the beta and release candidate releases.

5. Conclusion

This position paper looks at open source development practice and makes some observations as to how this practice is different from Software Product Family development practice and how improvements could be made to the latter.

This article does not, and cannot possibly tell Software Product Family owners how to develop their software. Instead, it merely suggests to them that there is this set of practices that may be found in many open source projects that is known to work well at least in that context. In so far these practices are not already integrated into the Software Product Family development practice, it is further outlined how that might be accomplished and what the tradeoffs are.

The key vision underlying this paper is that from the point of view of the experts, i.e. the developers, the open source style of working is the best practice in the context of large software projects that are worked on by many geographically distributed developers.

A key difference between open source projects and most Software Product Families is that in open source projects the developers are in charge. This works out surprisingly well for all the aspects discussed above. All of the three cited projects are performing excellent in terms of quality, features and development speed. Therefore, the key recommendation of this paper to Software Product Family owners is to carefully (re)consider the balance between product family developers and management. Empowering developers allows them to work in a way that they consider best (and who are we to disagree). At the same time, of course the point of Software Product Families is directly aligned with the owning company's core business.

6. References

- [1] Eclipse Foundation Process for accepting contributions,
<http://www.Eclipse.org/legal/EclipseLegalProcessPoster-v1.2.4.pdf>
- [2] The Eclipse development process.
<http://www.Eclipse.org/Eclipse/Eclipse-charter.html>.
- [3] Eclipse Release Engineering FAQ.
<http://WIKI.Eclipse.org/index.php/Platform-releng-faq>.
- [4] Hacking Mozilla,
<http://www.Mozilla.org/hacking/life-cycle.html>
- [5] Interview with Linus Torvalds
<http://edition.cnn.com/2006/BUSINESS/05/18/global.office.linustorvalds/>
- [6] Slideware definition,
<http://en.WIKIpedia.org/WIKI/Slideware>, 2005-05-31
- [7] R. van Ommering, Building product populations with software components, proceedings of the 24rd International Conference on Software Engineering, pp. 255-265, 2002.
- [8] Marco Sinnema, Sybren Deelstra, Jos Nijhuis, Jan Bosch: Modeling Dependencies in Product Families with COVAMOF. ECBS 2006: 299-307.

[9] Jan Bosch, Maturity and Evolution in Software Product Families: Approaches, Artefacts and Organization. SPLC 2002: 257-271.

[10] Kent Beck, "Test Driven Development", Addison Wesley, 2002.

Applying OSS development practices in software product line development

Jilles van Gorp

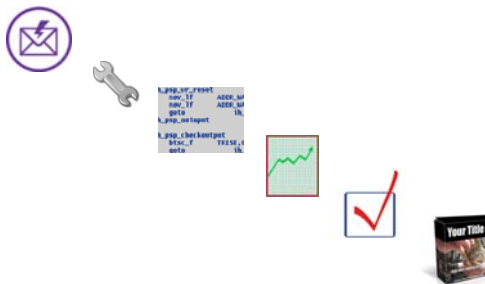
Nokia Research Center
Software & Applications LAB
Software Architecture Solutions Group



NOKIA
Connecting People

Overview

- OSS = license + set of *practices*
- Six development themes
 - Communication
 - Tooling
 - Code ownership
 - Technical roadmap
 - Quality management
 - Release management
- What is the OSS practice for these themes?
 - Paper provides examples from eclipse; mozilla & linux projects for each theme
 - How does this apply to SPL development practice?
 - What lessons can be learned?



NOKIA
Connecting People

Executive summary

OSS = *inter* organizational reuse

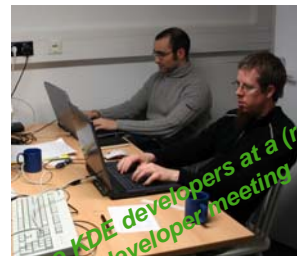
SPL = *intra* organizational reuse

- OSS development practices represent what developers feel is the *best way of developing software* if not obstructed by management + deadlines + corporate stupidity + etc.
- Judging by the *development speed* and *quality* of many OSS projects, these developers might be on to something.
- SPL engineering and OSS are the two most *successful strategies for large scale reuse*.
 - Why not do both?

Communication in OSS world



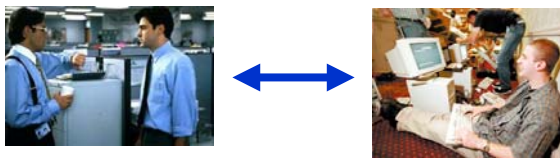
- In OSS projects development teams are
 - Large
 - Geographically distributed
 - Crossing organizational boundaries
 - Composed of developers only
- Communication infrastructure consists primarily of
 - **Email (private and mailinglists)**
 - Newsgroups
 - IRC
 - Project website
 - Development tools (next theme)
- Face 2 face meetings not so common



Communication issues in SPL development



- Large development teams
- Development may cross (intra) organizational boundaries
- Organizations are large, sometimes multinational
- Secrecy & customer relationships form obstacles for direct, open communication
- Significant involvement from people who are not developers
- Face 2 face meetings are important



So how can OSS practice help here?



- Use the same tools: email, mailinglists, instant messaging
 - Documented discussions
 - Information sharing
 - Consensus based decision making
- De-emphasize face to face meetings because
 - Generally excludes important people not on location
 - Eats up valuable development time
 - Leaves no trace in relevant tools
- Ensure non-developers stay in the loop
 - Make sure they are on the mailinglists, wikis, etc.



OSS Tools

- Minimum prerequisites for launching an OSS project:
 - Mailinglist
 - primary communication channel for the people working on the project
 - public discussion
 - Version control
 - Preferably Subversion these days but CVS remains popular too.
 - Bug tracking
 - Bugzilla or similarly capable system
- Optional
 - Build, integration and testing facilities
 - Website
 - A place where you list the access points to the tools above + maybe documentation and marketing information
 - WIKI
 - A place for developers/users to collaboratively work on non development artifacts

SourceForge.net

Log In - Create Account

Software Search Advanced

SF.net Projects My Page Help

Software Map Create Project New Releases Top Projects New Projects Help Wanted

SF.net » Projects » jEdit » Summary

jEdit Stats - Activity: 99.99% RSS

Summary | Admin | Home Page | Tracker | Plugin Bugs | JEdit Launcher Bugs | Plugin Central Submission | Bugs | Patches | Feature Requests | Mail | Screenshots | CVS | Subversion | Files

jEdit is a programmer's text editor written in Java. It uses the Swing toolkit for the GUI and can be configured as a rather powerful IDE through the use of its plugin architecture.

Download jEdit

Project Admins: daleanson, ezust, jchoyt, kpouer, mdillon, olearyni, orutherfurd, spetov, vampire0

Operating System: OS Independent (Written in an interpreted language)

License: GNU General Public License (GPL)

Category: Software Development, Text Editors

Need Support? See the support instructions provided by this project

Everything required to set up a project, for free

8 © 2006 Nokia Jilles van Gurp

NOKIA
Connecting People

The reality of tools

- OSS projects are **tool centric**
 - Tangible results of the project **live inside the tools**
 - Source code in the SCM
 - Bug reports in the bug tracking system
 - Documentation in the WIKI
 - People communicate through tools + mail
 - All other forms of communication optional
- Interesting consequences
 - Anything outside the tools is irrelevant.
 - [software_architecture.ppt](#) on somebody's laptop is not accessible to anyone and disconnected from the information in the SCM and bugtracking db. It **might as well not exist** and it is probably out of date/inacurate/obsolete/misleading/.... !
 - Tools are the **only interface** to the project
 - Tools are **not compatible with many of the traditional waterfall** model phases:
 - You won't find a [requirements specification for the linux](#) kernel
 - Nor is there a detailed [design document for the Firefox](#) browser
 - While there are some [open source UML tools](#), they are **rarely used** in open source projects!

community makes its own tools



SPLs and the other reality

- **Good news!:** tooling tends to be way better in corporations!
 - Expensive SCM systems, IDEs and UML tools are common.
 - + Variability modeling, build configuration, component technology,'
 - And we can buy even more if we need to!
- Bad news:
 - Tools are still incompatible with many of the traditional waterfall model phases
 - not necessarily development centric or even developer friendly
 - poorly integrated with each other
 - Information exists outside the tool chain: **alternative reality**
 - A lot of problems arise from the **problematic relation between the tool reality and the reality outside the tools**
 - Management makes decisions based on **slideware** - they don't understand the tool reality. Also there seems to be a **reality distortion field** clouding their judgement sometimes!
 - Sales sells software based on **requirement specifications**
 - Customers receive software based on **artifacts in the SCM**



Tool recommendations



- More focus on **integration**
 - Lesson from OSS community:
anything not in the integrated toolchain is irrelevant and distracts from the tool reality
- Reassess the added value of **non-developer oriented/friendly features** in commercial tools
 - Lesson from OSS community:
simple, standard tools lower the barrier of entry
- Reassess the added **value of requirements, architecture and design documentation**.
 - Are they really that important to the development team?
 - Lesson from the OSS community:
It is possible to develop large, complicated software systems on a tight schedule without those assets.

OSS Code ownership & governance

```
h_psp_reset  
mov_1f ADDR_1f  
mov_1f ADDR_1f  
goto 1h  
h_psp_noinput  
h_psp_checkoutput  
btsr_f TRISE,  
goto 1h
```

- project owner(s)
 - individuals
 - sometimes organized into foundation
 - coordinate development, communication, planning, sponsoring, legal issues, etc.
 - sometimes not developers
 - e.g. mozilla's president: Mitchel Baker
- Module/component owners
 - Senior developers with established **reputations**
 - **Safeguard** code quality & drive development
 - Code **reviews**
 - Commit **approval**
 - **Initiate** new development
- Sponsors
 - Influence by voting with your wallet



Ownership in SPL

```

C:\pdp>reset
nov_1f      ADDR_3d
nov_1f      ADDR_3d
goto        1h
h_psp_notput
h_psp_checkoutput
btest_f     TRISE
goto
  
```

- Problems
 - technical decisions complicated by non technical concerns
 - decision power with people without technical competence
 - no clear ownership or owner does not have full authority
 - conflicts of interest, e.g. product deadlines vs. product line quality
- Solutions
 - Introduce code ownership as is common in OSS projects
 - Must have **authority** to take actual decisions
 - Separate product and product line development as much as possible
 - higher degree of **autonomy**
 - **Don't micromanage** development teams
 - e.g. deciding on spending time on refactoring instead of feature development should be up to code owner, not his manager

Roadmaps in OSS



- Roadmap: **what are we building**
- Tend to be
 - sketchy
 - short term
 - realistic - what can be achieved in time frame X
 - not set in stone

Milestone	Release Date	Summary
1.0	Phoenix 2004-11-9	Inaugural Release
1.5	Deer Park 777 2005	New Gecko, ongoing HIG compliance, sw/update and extension manager improvements.
2a	Q1 2006	pre-feature complete alpha
2b	early Q2 2006	feature complete beta
2rc(s)	late Q2 2006	
2	late Q2/early Q3 2006	Final Release
3.0	Q1 2007?	

Firefox 2.0 roadmap

- Purpose
 - Ensuring relevant people know what is being worked on
 - Realizing long term project vision through having short term plan for specific features
 - Planning & coordinating development
 - Generate interest from users and contributors
- Decisions
 - Based on consensus, usually result of community discussion.
 - Painful decisions taken by respected/influential developers

Roadmaps in SPL organizations



- Tend to be
 - very detailed
 - Medium to long term
 - Basis for committing resources
 - Not very flexible after resources have been committed
 - e.g. product development dependencies
- Purpose
 - Planning
 - Marketing
 - Ensuring future competitiveness
- Decisions
 - Based on market demand + requirements + corporate strategy
 - Painful decisions are not taken by developers

SPL Roadmap: learning from OSS



- Purposes are different!
- Need to be more flexible, short term oriented
 - Agile!
 - Make more effective use of opportunities spotted by individual developers
- More room for refactoring and other non feature related activities

Quality management in OSS & SPF development



- OSS Developers value quality
 - not necessarily all quality attributes (e.g. usability)
- How
 - **Testing** of alpha versions & nightly builds by end users
 - Explicit code **reviews** (see code ownership)
 - Bug fixes are attached to bug report and only committed when approved
 - **Automated tests** & continuous integration
 - Functional - correctness, regression testing
 - Non functional - performance, security, memory usage, ...
 - Endless delay of the **perfect** 1.0 release.
- In principle SPF developers can do the same
 - Except maybe for exposing nightly builds/alpha versions to end users
 - I'm sure they do :-)

Release management in OSS



- Release process tends to be comparatively strong
 - due to amount of end users involved early on: they care
 - desire to deliver good quality
 - user feedback
- nightly builds >> alphas >> betas >> release candidates >> 1.0 >> 1.0.1 1.0.n
 - increasingly **strong commit review process** towards release
 - larger groups of testers
 - process does not stop after release
- Mentality: **it is done when it is done and not sooner!**
 - Release process can take several months after last alpha milestone
 - Some projects are quite good at predicting when they are done

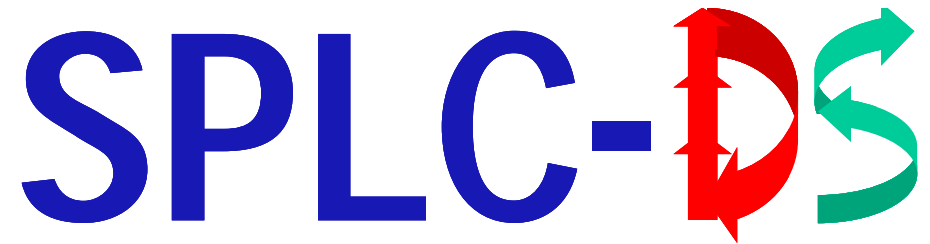
Release management in SPF context



- Problems
 - Much less 'users' (product developers)
 - Much pressure to release on time
 - users are waiting for release
 - product developers are less eager to alpha test
 - supporting old releases is time consuming and impractical
- Solutions (from my experience)
 - Don't expose every build to product teams
 - creates too much feedback, product teams need to have stable code
 - During release phase, test with one or two product teams
 - Don't give them new builds every day
 - Select low risk products
 - Have them plan to upgrade to the release version!
 - Expand to multiple product teams during beta phase
 - Don't release too early & don't commit to specific release date too early

Conclusions

- It's not black and white
 - in practice many OSS practices are already adopted in enterprise
 - OSS development practice has evolved from and is influenced by corporate development
- Some things to remember
 - **Meritocracy**: technical decisions are best taken by technically competent persons
 - OSS developers have the advantage of being in charge
 - If OSS developers don't waste time on certain things, why should we?
 - **How valuable is the non technical overhead really?**
 - Beware of the conflict between **tool reality** and **slideware reality**
 - If a non functional requirement is important: **test** for it and set targets!
- How to apply this in SPF context?
 - That is the big question :-), good luck.



Software Product Lines Doctoral Symposium

Organizers:

Isabel John

(Fraunhofer IESE, Germany)

Len Bass

(SEI, USA)

Giuseppe Lami

(ISTI-CNR ,Italy)

Reviewers/Panelists

- 1 Birgit Geppert - Avaya Labs, USA
- 1 Andre van der Hoek - University of California, USA
- 1 Kyo Kang - POSTECH, Korea
- 1 David Weiss - Avaya Labs, USA

panelists reviewed the papers and gave comments on the talks

6 Papers

Richard Biser
The Involvement of Non-Technicians in Product Configuration

Timo Asikainen
Methods for Modeling Variability in Software Product Families

Nan Frederik Mungard
Feature Model Based Product Derivation for Product Lines

Marcilio Mendonca, Toacy Oliveira, Donald Cowan
Collaborative and Coordinated Product Configuration

Karen Cortes Verdin, Cuauhtemoc Lemus Olalde
Aspect Oriented Product Line Architecture (AOPLA)

Uirá Kulesza, Carlos Joazeiro Lucena
An Aspect-Oriented Approach to Framework Development

Proceedings are online at the Symposium website:
<http://www1.isti.cnr.it/SPL-DS-2006>

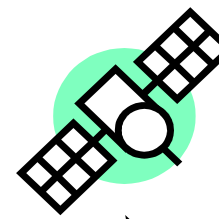
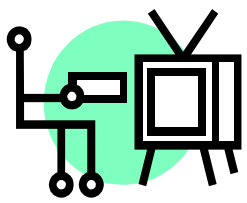
SPLC 06 Product Derivation Panel

Building Interactive TV Applications
with pure::variants

Danilo Beuche, pure-systems

About pure::variants

- **feature model** based variant and variability management tool
- solution asset modelling (**family model**)
- provides extensible **model transformation** framework
- does not require any specific implementation technique
- provides easy-to-use integration interfaces with other tools, e.g. for
 - requirements engineering
 - test management
 - code generation



pure::variants

Device Applications
(Family Model)

Programme Definition
(Feature Model)

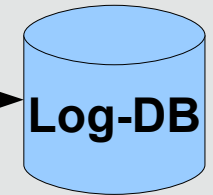
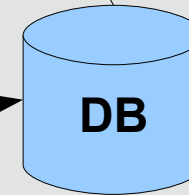


Producer

Editing Application
(Family Model)

Transmission Application

Device Application Instances
(Created with P::V)



Web-based Page Editing Application Instance

Page Templates



Director

Page Instances

Pages with Content

Content



(Created with pure::variants)

pure::variants Project Structure

Feature
Models

Programme Definition: Page Types, Colors, Channel, Time

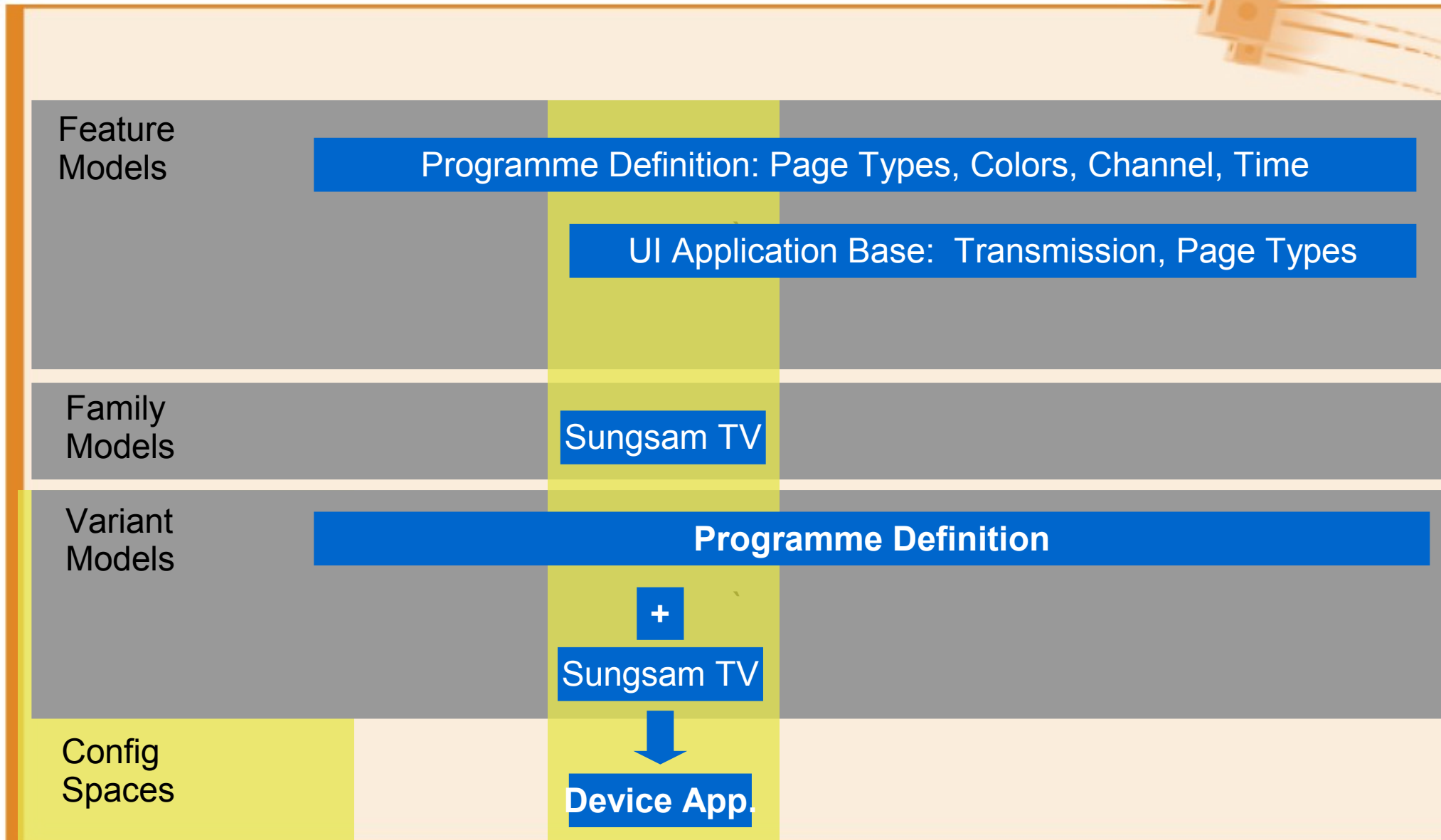
Family
Models

Variant
Models

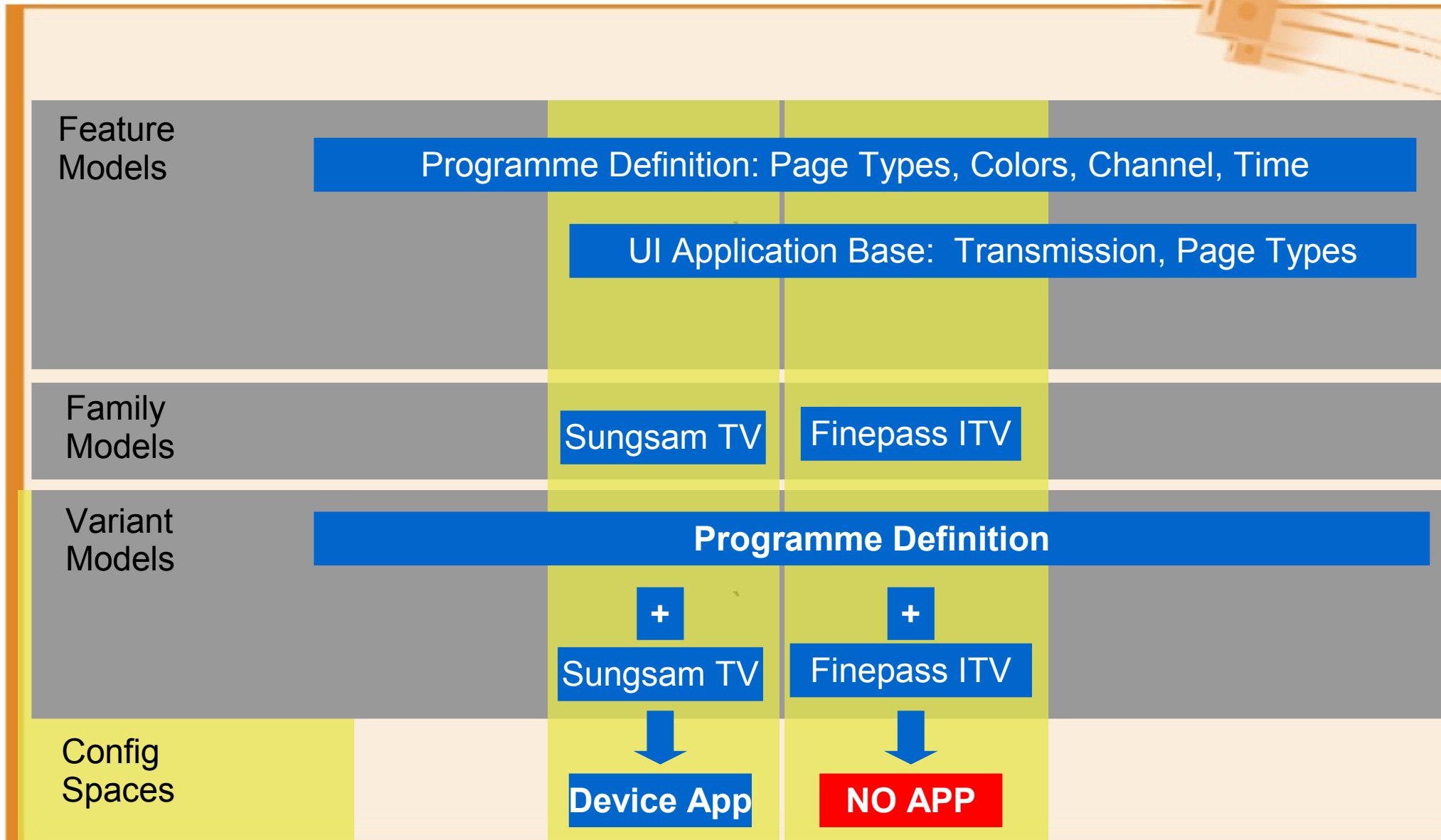
Programme Definition

Config
Spaces

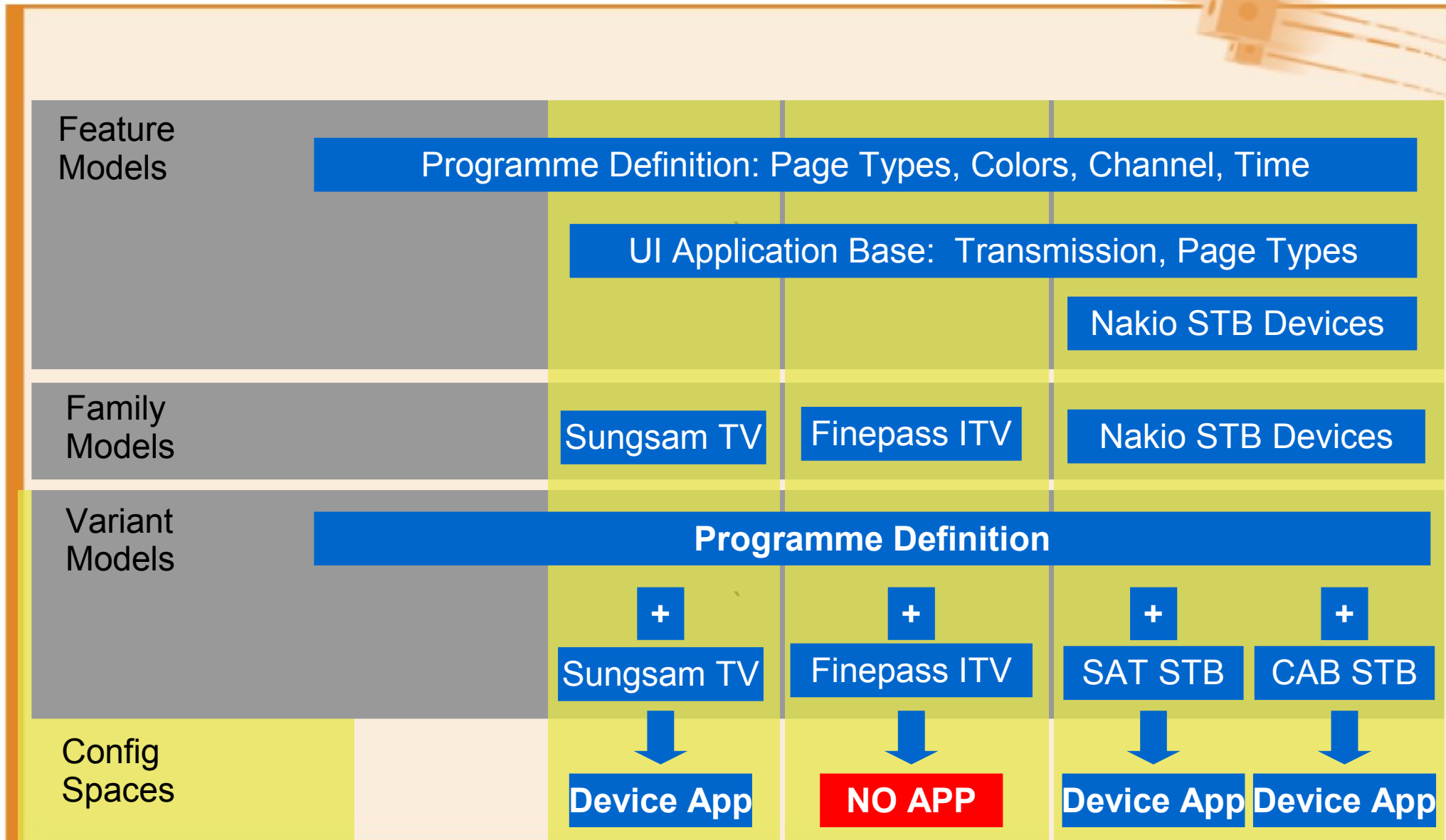
pure::variants Project Structure



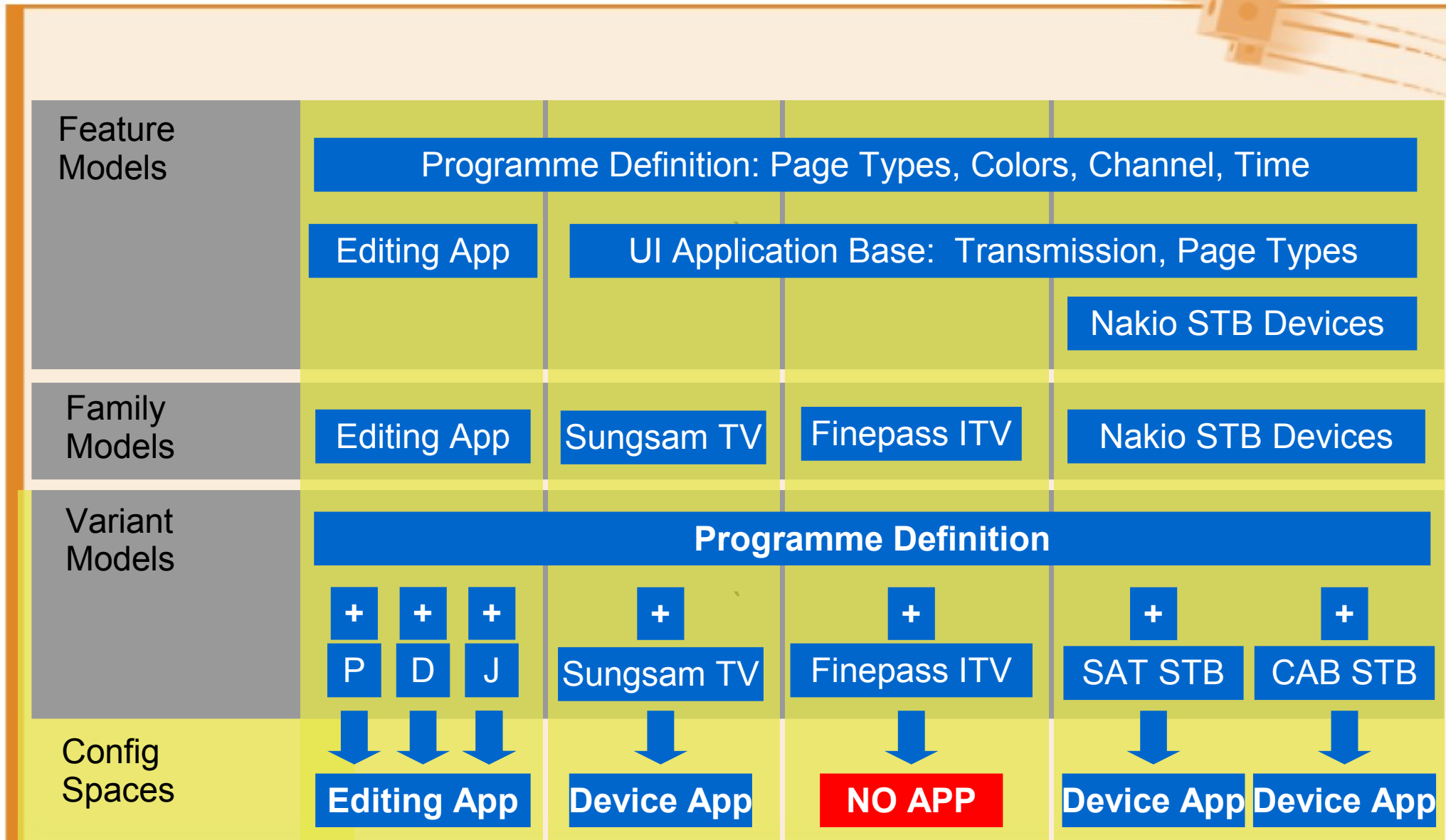
pure::variants Project Structure



pure::variants Project Structure



pure::variants Project Structure



Common Tasks



- Create Programme
 - create new programme's variant description (may inherit from templates)
 - generate device and editing application
- Change Programme (add/change used page types, change colors, ...)
 - change programme's variant description
 - regenerate device and editing application
- Add Support for New Device
 - add device feature/family model, source components (from software development)
 - create config space for device

More about pure::variants

See our demo: Tomorrow, 12.30

or visit

www.pure-systems.com/pv



BigLever Software Gears Solution

Product Derivation Approaches SPLC 2006 Panel

Charles W. Krueger
August 23, 2006

Consolidate.

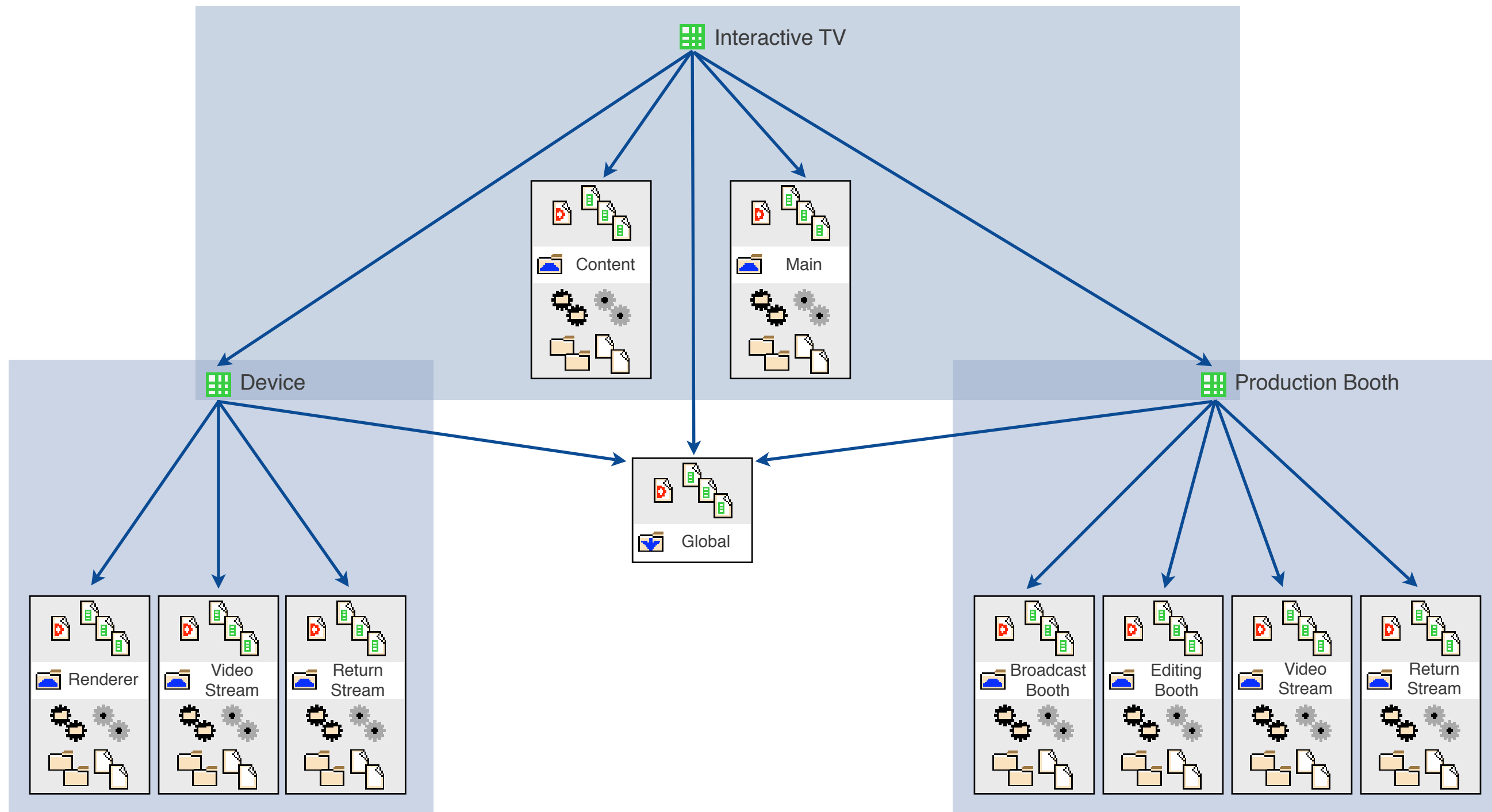
Simplify.

Leverage.

Introduction

- Content Management problem statement
 - Generate an XML data stream from an abstract content model
 - Non-technical TV Producers define the “products”
- Gears is a *Software Product Line Engineering* tool for Software Engineering and Product Marketing roles
- Our solution
 - Content Management
 - XML content stream generated from textual format
 - Two Gears pattern/substitution variation points
 - Two Firmware Product Lines
 - Rendering devices
 - Broadcast and editing booths

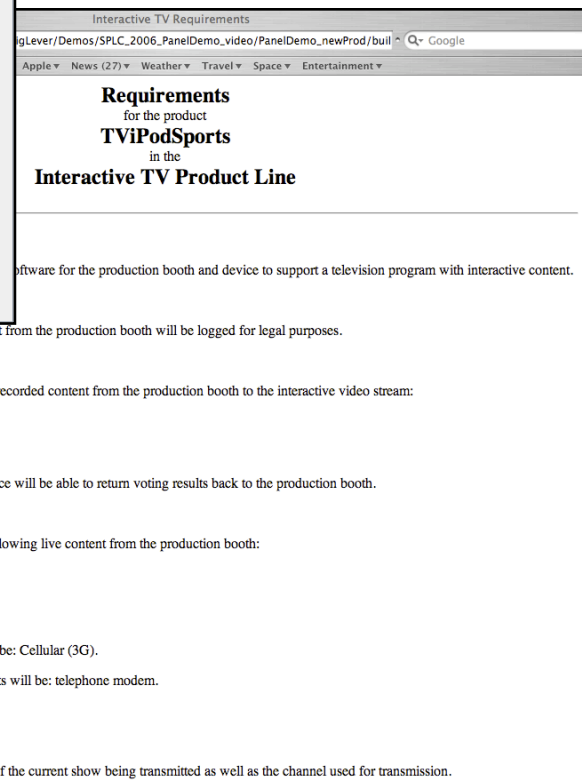
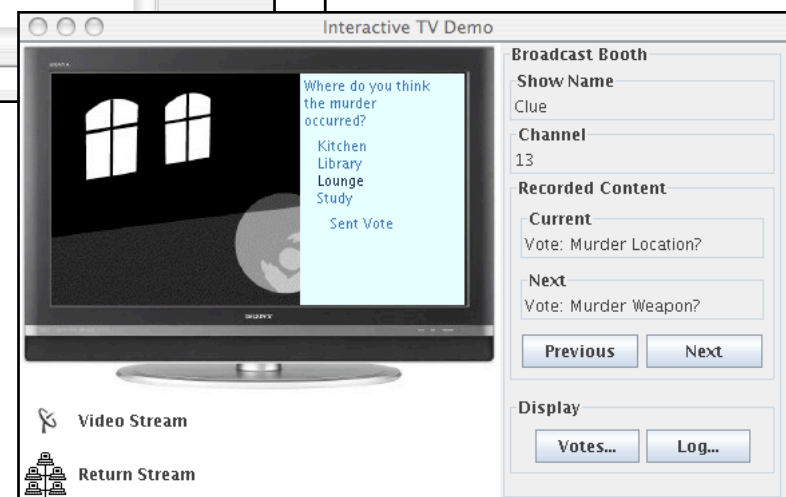
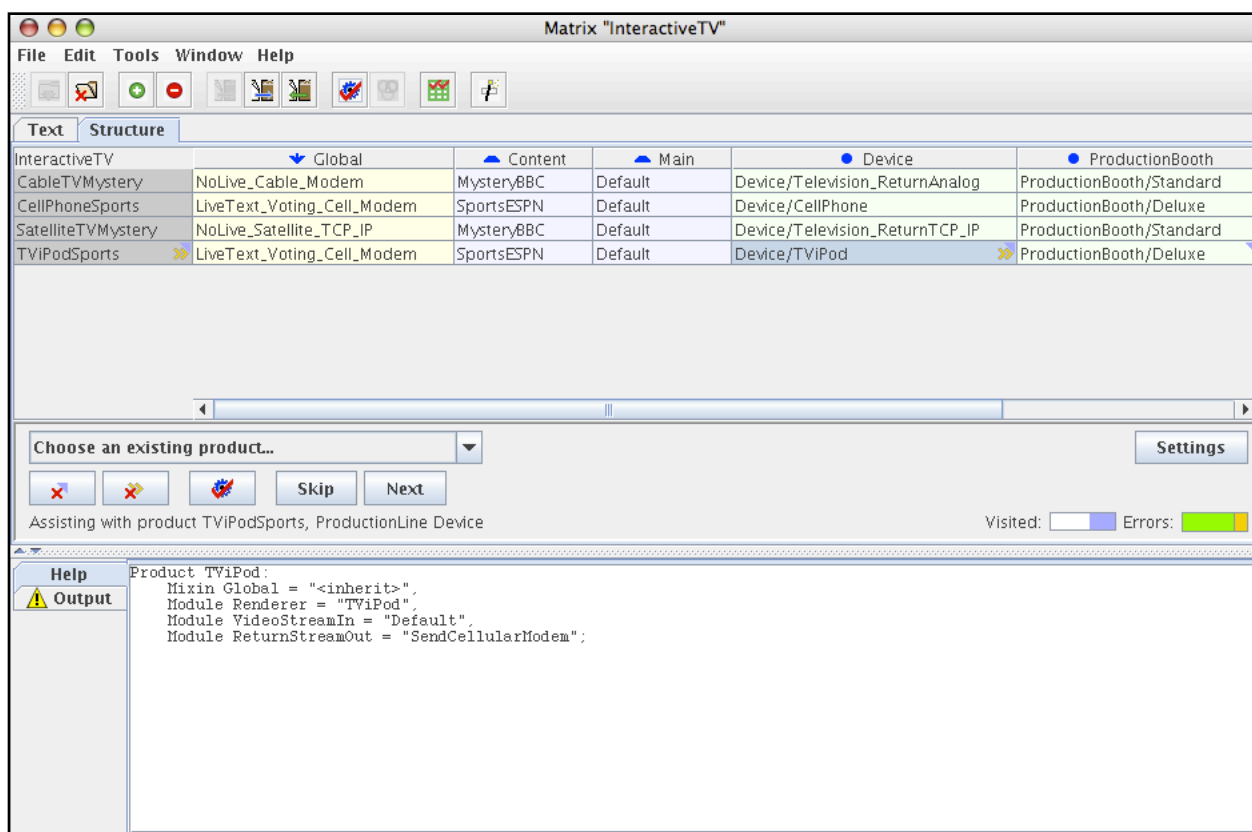
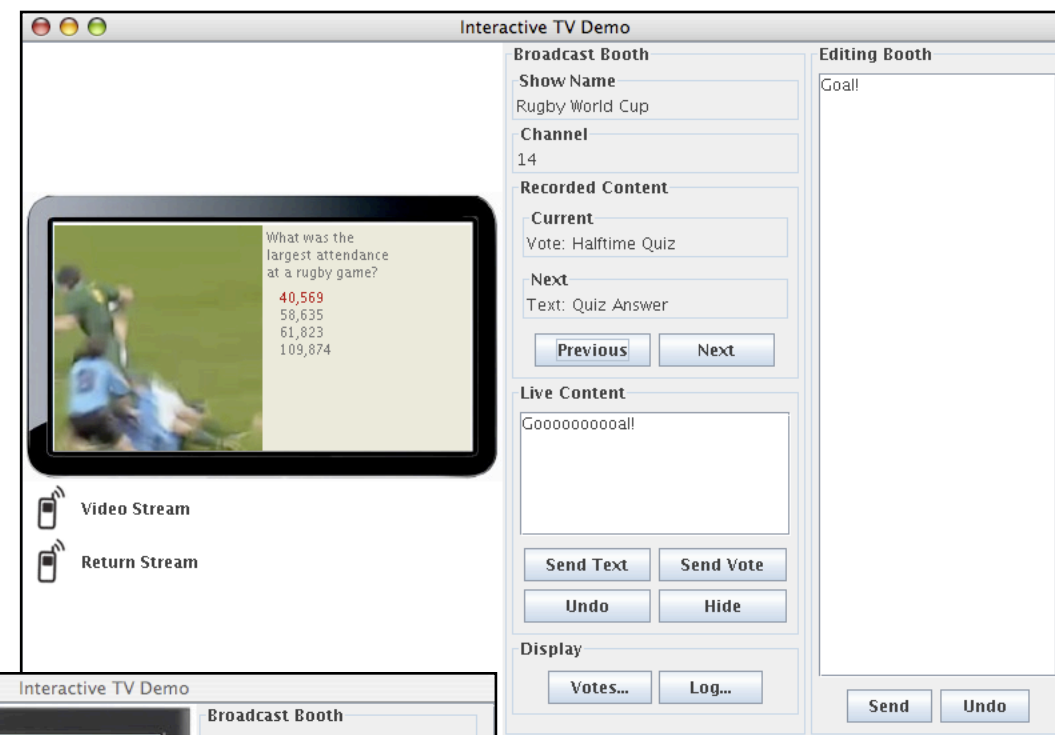
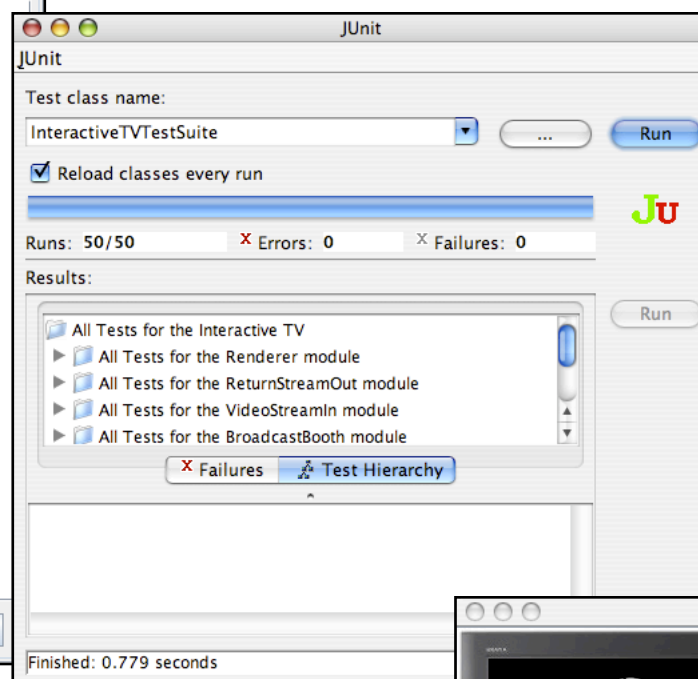
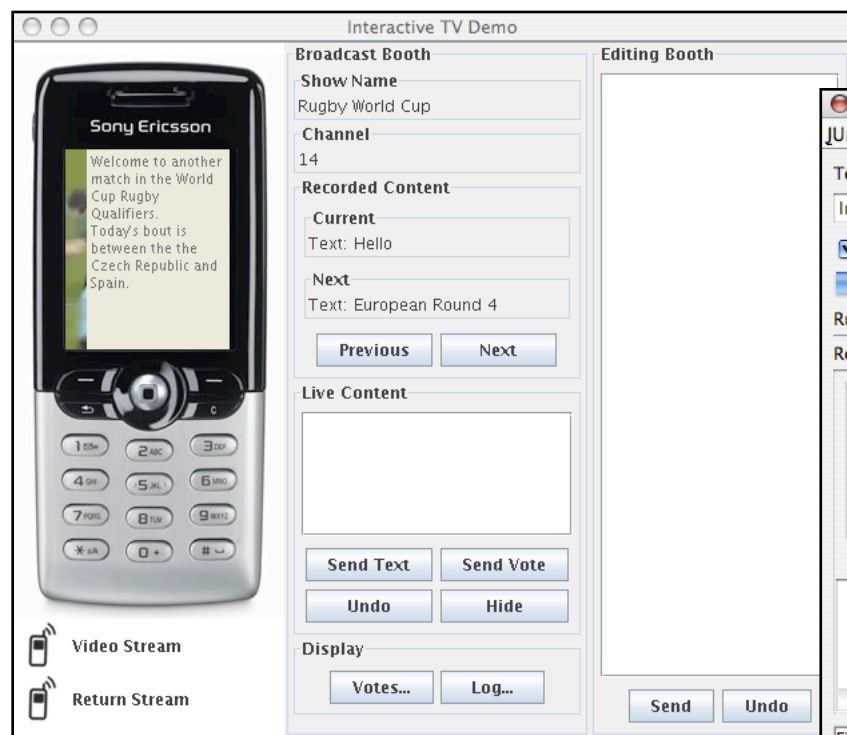
Interactive TV Architecture (Asset View)



Initial Scenario

- We start with clone-and-own bespoke application, possibly on CM branches or directory copies
 - Content for 2 programs
 - Firmware, requirements and test cases for 2 devices
 - Firmware, requirements and test cases for 2 production booths
- Using Gears, we consolidated these into SPL assets
 - Estimated level of effort: 1 developer-week for every million lines of cloned assets

Screenshots



The Three Questions

- How large a portion of a product is automatically derived?
 - In scope: 100%
 - Out of scope: Delta engineering
- How are new features and functionality developed?
 - Delta engineering in the *core assets, feature model* and *product profiles*
 - Example: TV iPod
- What is the cost and time to create a new feature or change the application platform?
 - Delta engineering and regression testing only
 - Less than 1% overhead to maintain feature, product and VP logic
 - Zero cost to automatically regenerate all existing products
 - Engineer the entire product line portfolio as a single system



Product Derivation Panel - Domain-Specific Modeling

SPLC 2006

23rd August

Juha-Pekka Tolvanen

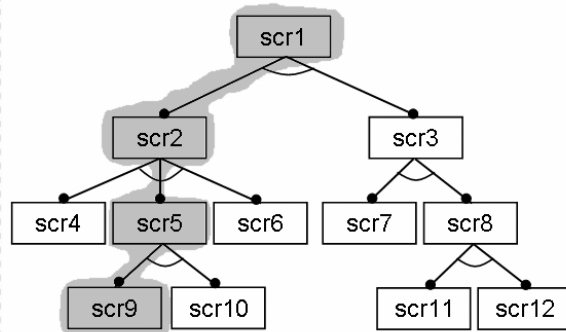
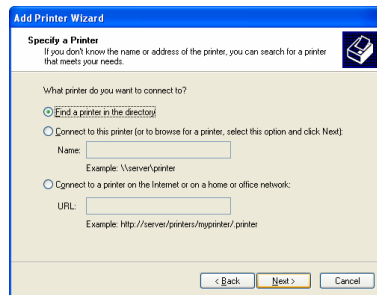
 MetaCase

Spectrum of variability mechanism*

Routine configuration

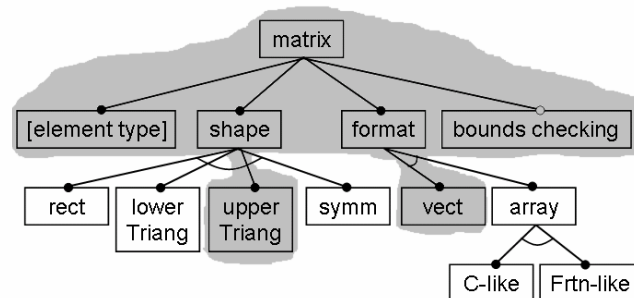
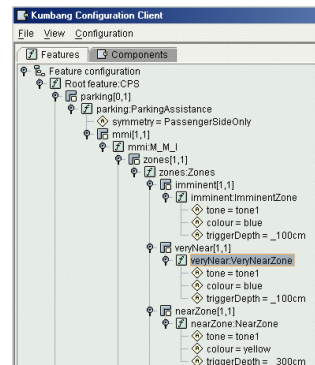
Creative construction

Wizards



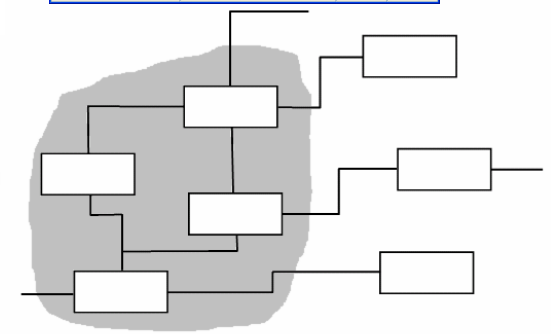
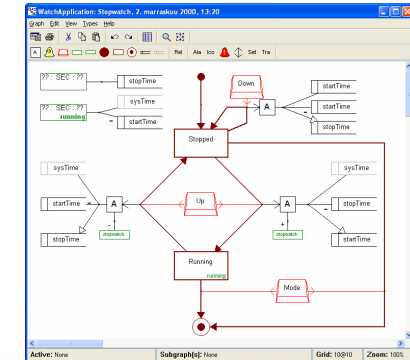
- Path through a decision tree
- All choices known
- Implementation available

Feature-based configuration



Subtree of a feature tree
All features known
Feature implementations available

Domain-Specific Language



- Subgraph of an infinite graph
- Variant space known, variants not
- New features can be implemented

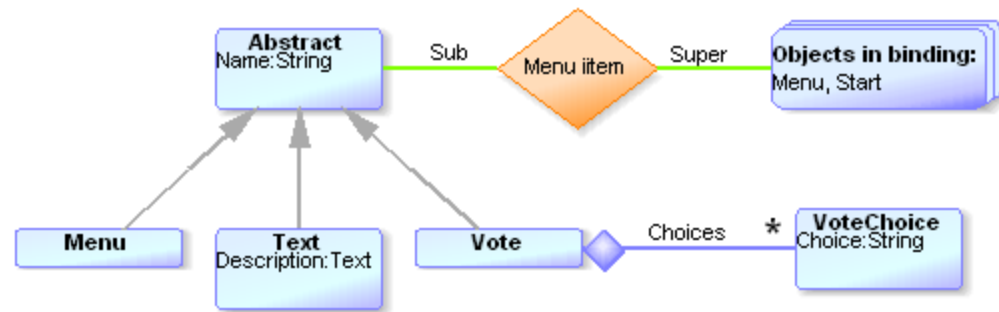
*Czarnecki&Eisenecker 2000



Domain Engineering

- Language (metamodel) defines the variation
 - everything we can model belongs to the product family!

Metamodel of
TV App structure:



- Generator produces variant code

- variation from model
and the common parts

```
do ~Super~Sub.() {  
  '<'; type; ' name="'; id; '">';  
  subreport; '_'; type; run;  
}
```

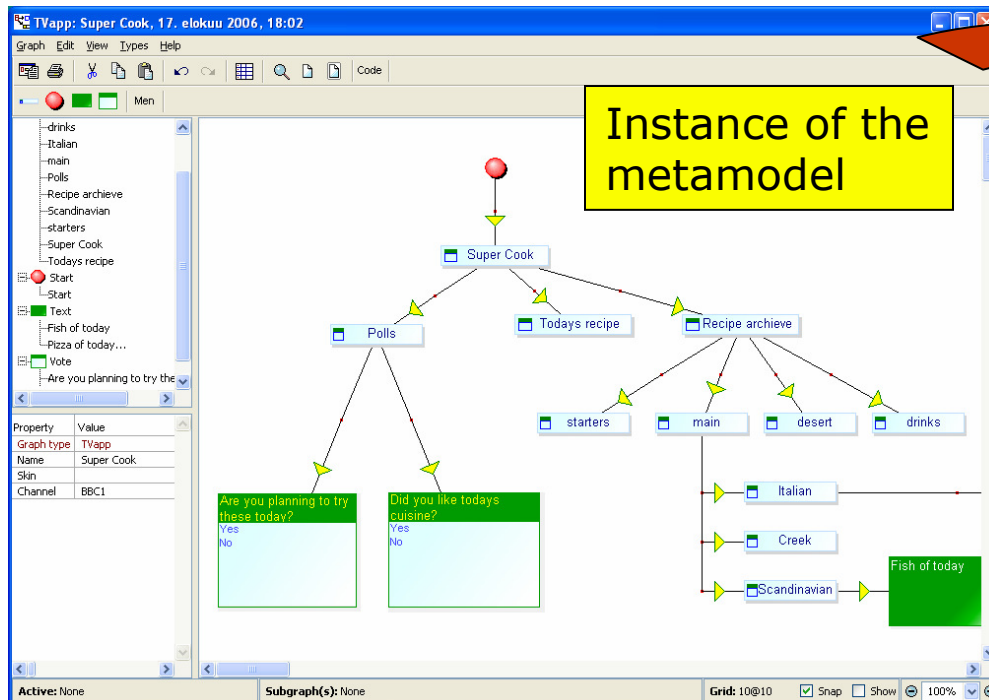
- Platform provides commonalities

- framework code to support generation and reusable units
(library, components, framework, middleware)

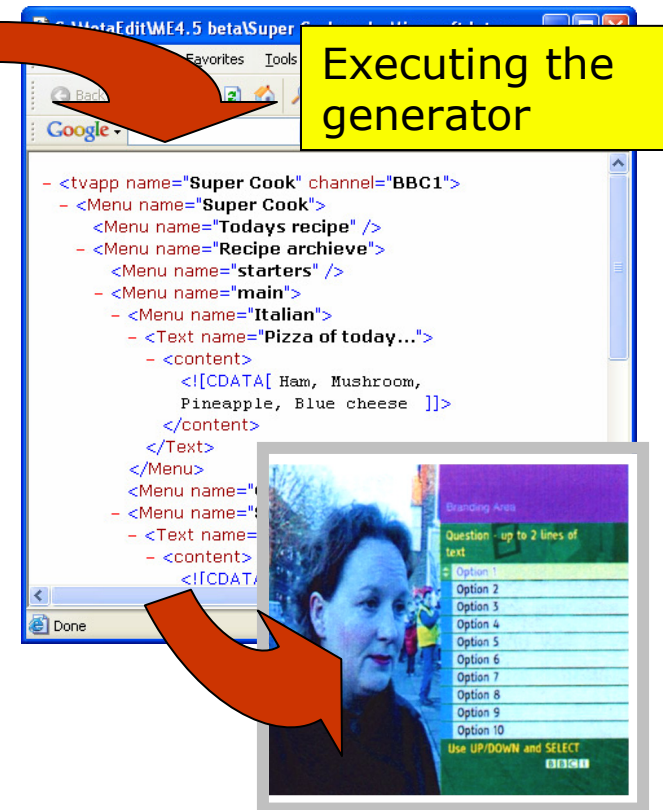


Application Engineering

- TV producer designs the applications....
- ... and its implementation code is generated, tested and passed to media (channel)



Instance of the metamodel

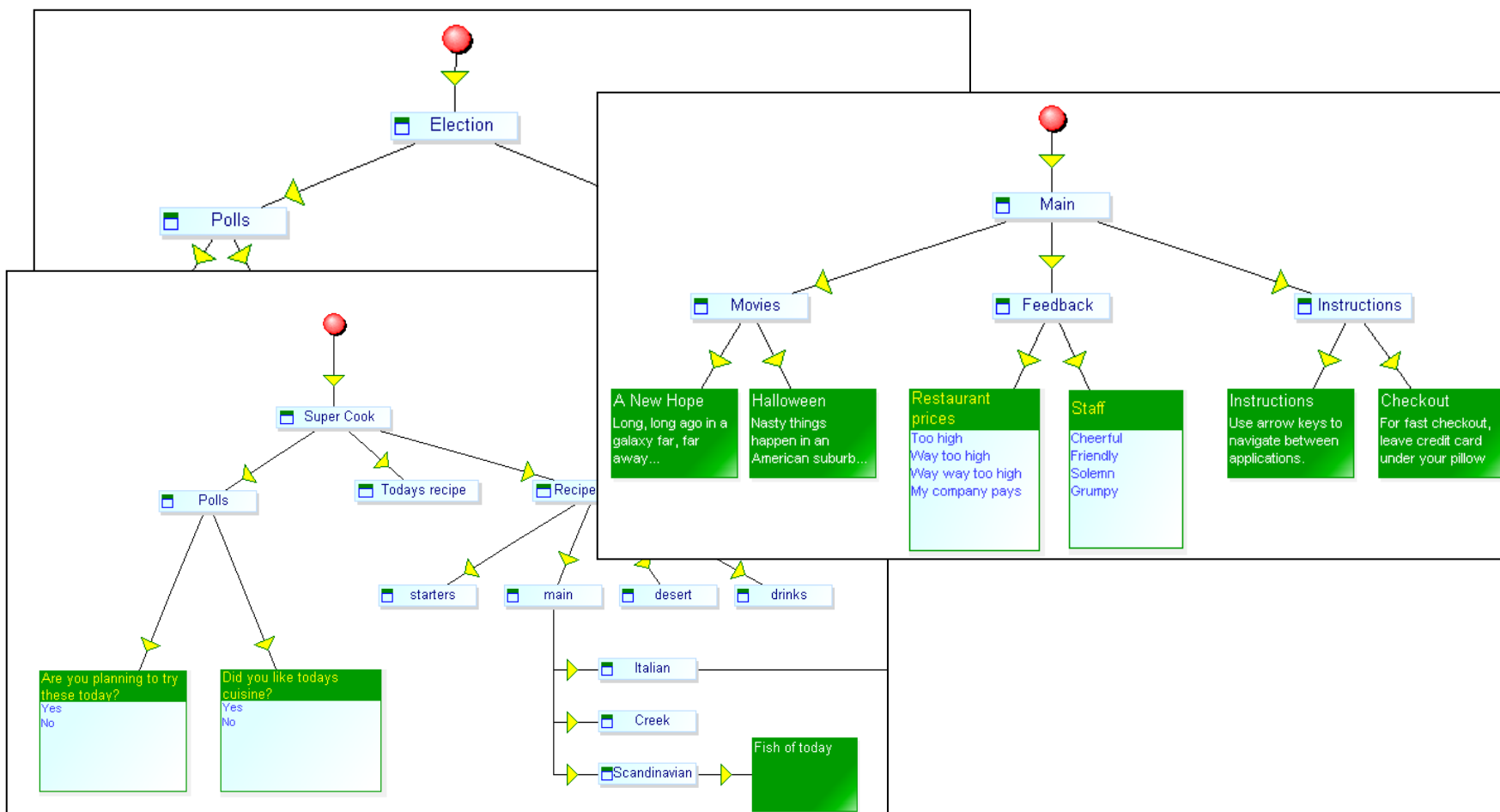


Executing the generator



Application Engineering: making other variants...

- HotelTV, BigBrother, Idols, election, Eurovision etc. etc.





Q&A

Q: How large a portion of a product is automatically derived?

A: Everything the metamodel is made for (in the TV example 100%)
– usually non-generated code can be integrated too (calls, partial classes, inheriting etc)

Q: How are new features and functionality developed?

A: By making a model using familiar product's concepts

Q: What is the cost and time to create a new feature

A: Very fast due to high abstraction of the language and automation;
e.g. new HotelTV application (with working code) took couple of minutes

Q: What is the cost and time to change the application platform

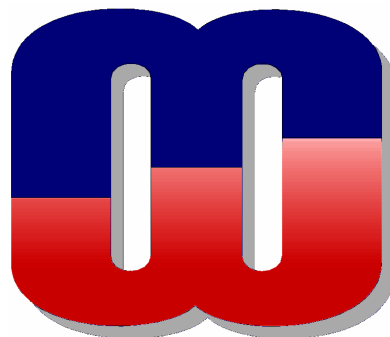
A1: If change in a target language then modify the generator (once!)

A2: If change in a product line then modify the metamodel (once!)



Thank you!

Questions, please?



www.metacase.com

USA:

MetaCase

5605 North MacArthur Blvd.
11th Floor, Irving, Texas 75038
Phone (972) 819-2039
Fax (480) 247-5501

Europe:

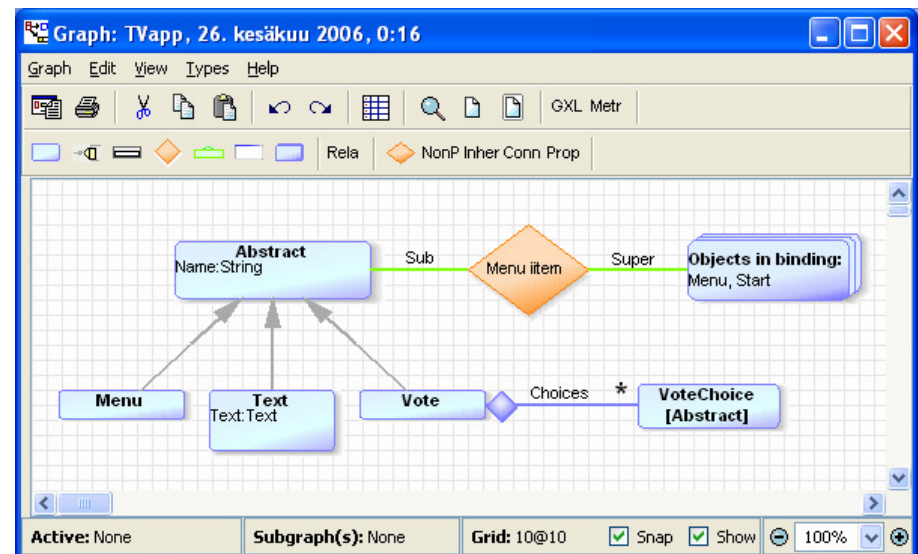
MetaCase

Ylistönmäentie 31
FI-40500 Jyväskylä, Finland
Phone +358 14 4451 400
Fax +358 14 4451 405



Domain Engineering in a tool

- Metamodel
... enables:
 - complexity hiding
 - early error prevention
 - use of familiar concepts and vocabulary
 - natural abstraction
- Generator
...enables:
 - automated derivation
 - early prototyping
 - analysis of designs
 - tests, metrics, simulation...



Report Browser for: Graph

Report Edit Breakpoint View Help

Hierarchical Metrics recurse

General Control External I/O

external ... execute
sep
prompt ... ask

```
Report 'recurse'
do ~Super~Sub.() {
  '<'; type; ' name="'; id; '>'; newline;
  subreport; '_'; type; run;
  subreport; 'recurse'; run;
  '</'; type; '>'; newline;
}
endreport
```

PHILIPS

Product Derivation Approaches

The Digital TV case and Koala



Rob van Ommering

Philips Research

SPLC 2006, Baltimore, August 23rd, 2006

Oops!

I'm in the wrong panel J !

Why am I in the wrong panel J ?

Task:

You are to design a system that will allow **non-technical** producers to build applications to accompany their programmes.

Why am I in the wrong panel J ?

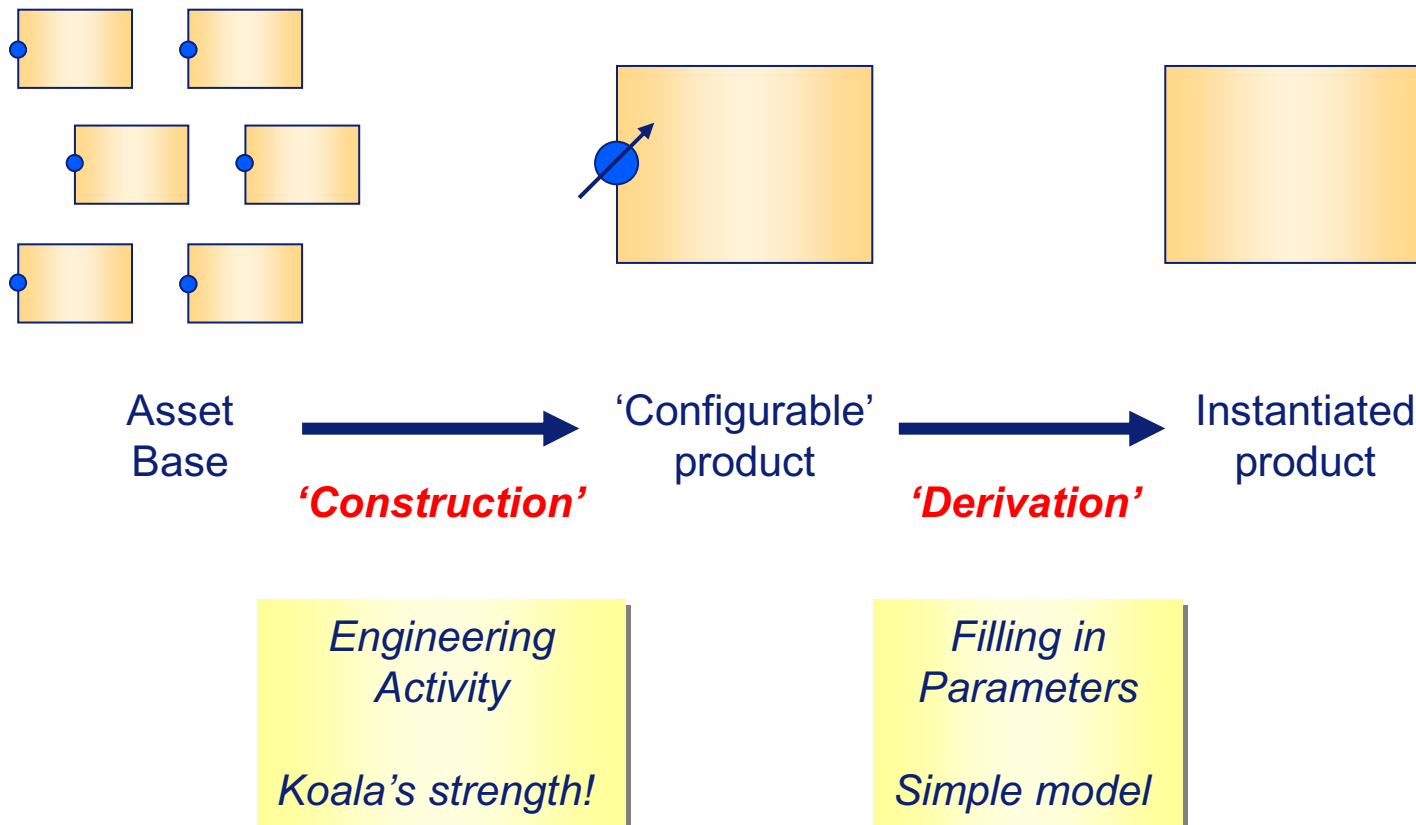
Task:

You are to design a system that will allow **non-technical** producers to build applications to accompany their programmes.

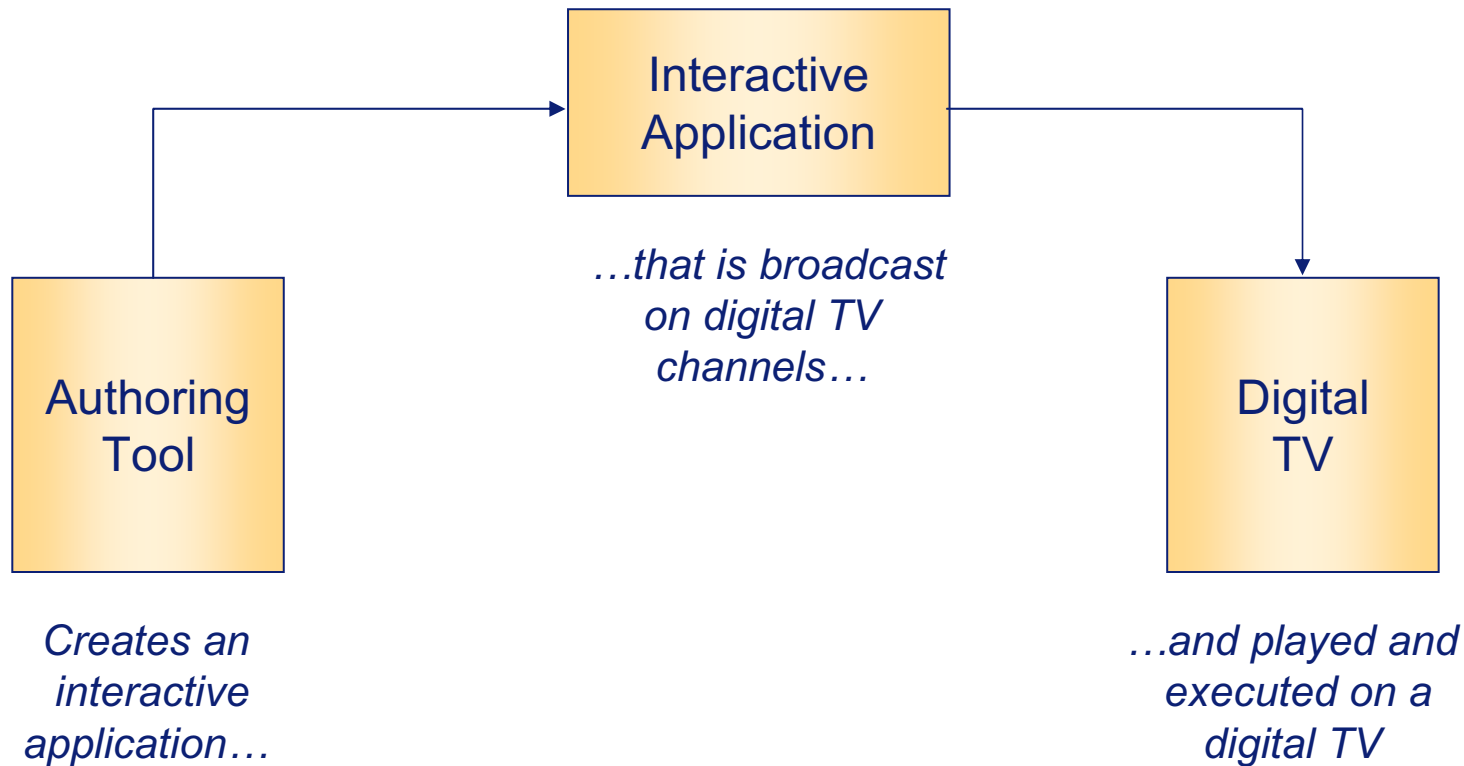
Koala's strength:

Allow **technical** engineers to create products from components

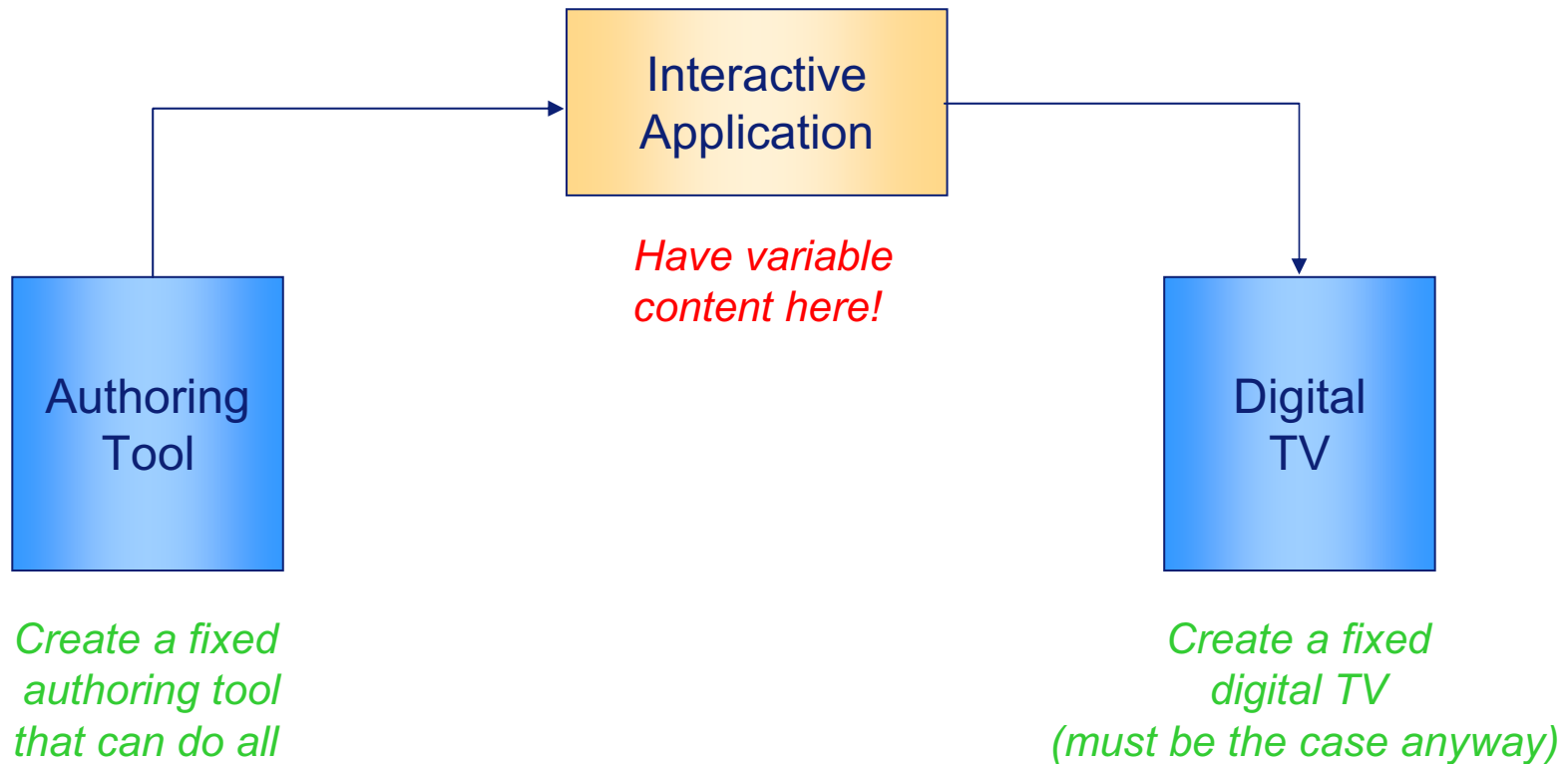
If I map this as best as I can to Koala...



OK, let's start with the case...

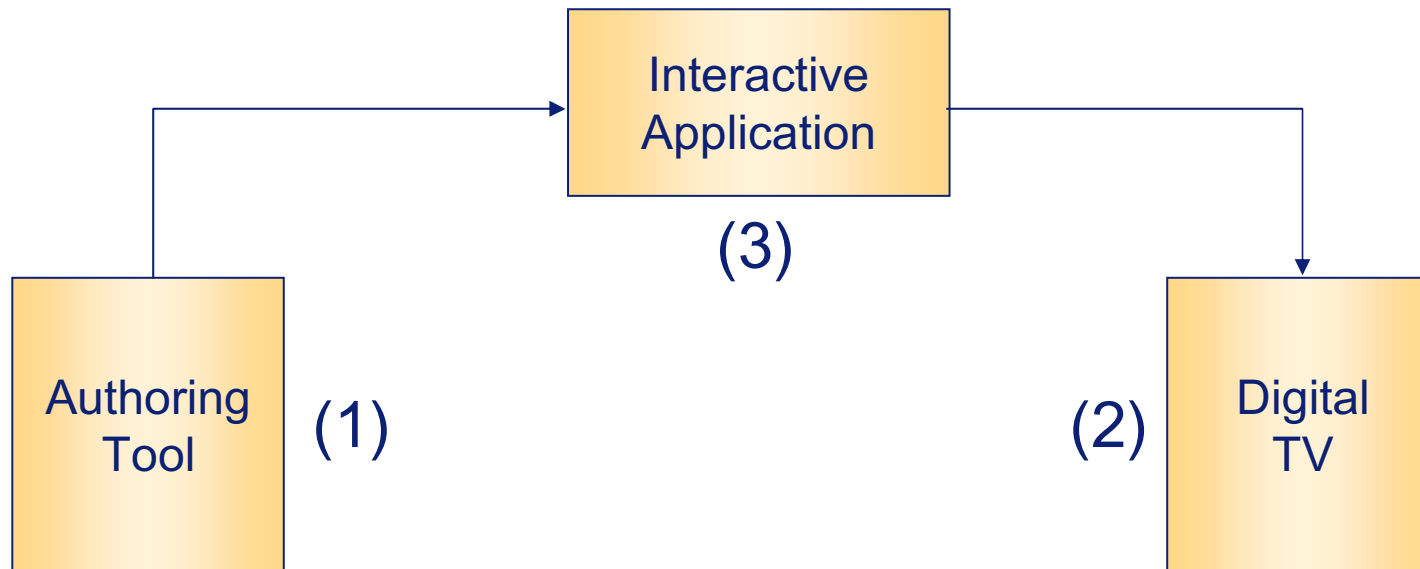


My first response would be...



This is in fact a content management problem!

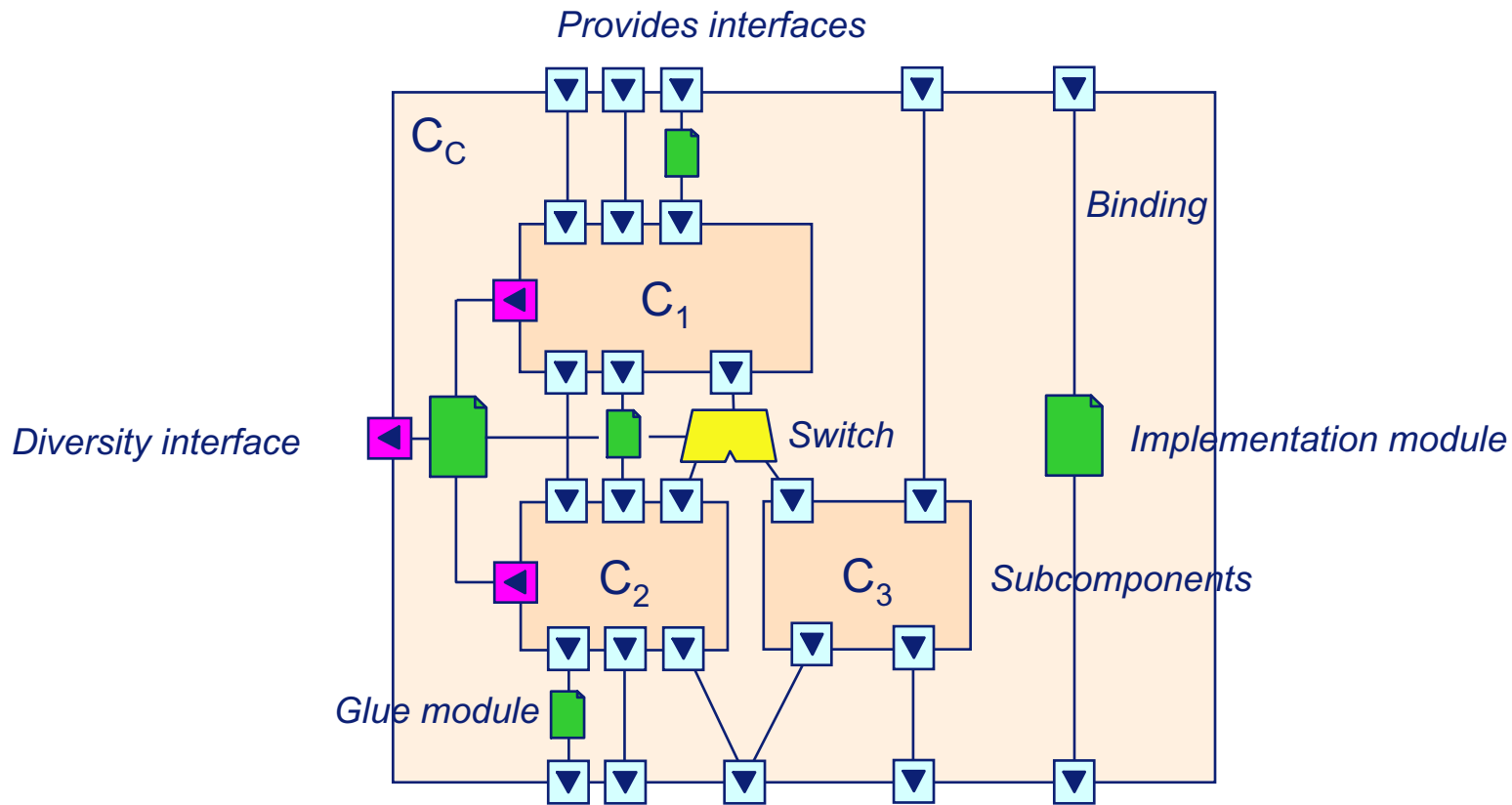
A more honest answer would be...



For *each* of the boxes I can create a product line...

Let's do this 丿 !

Koala...

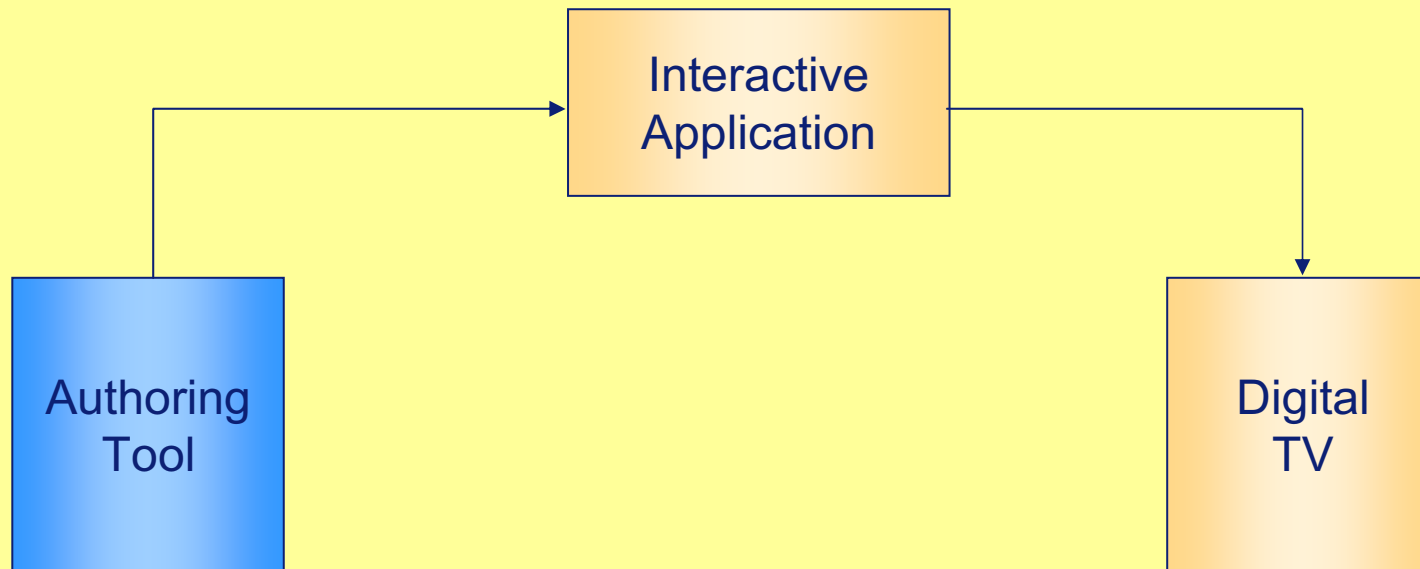


Koala is used in all of Philips' mid-range and high-end TVs

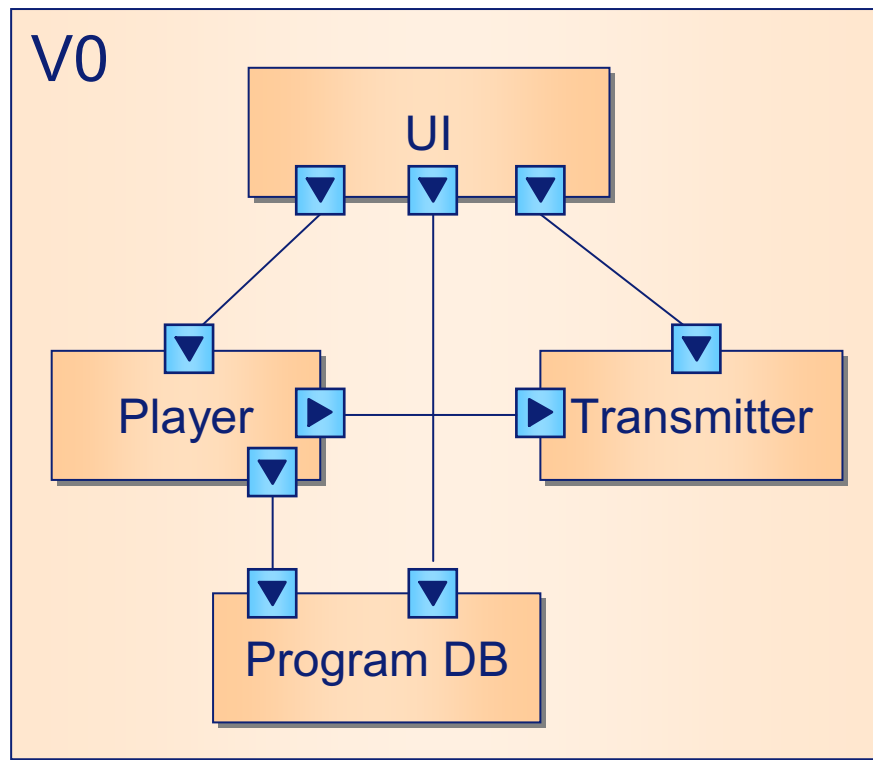
Requirements summarized

- Add page of text
- Customize colors and graphics
- Broadcast application
- Edit page and combine texts
- Add voting
- Maintain record of broadcast
- Move application including content

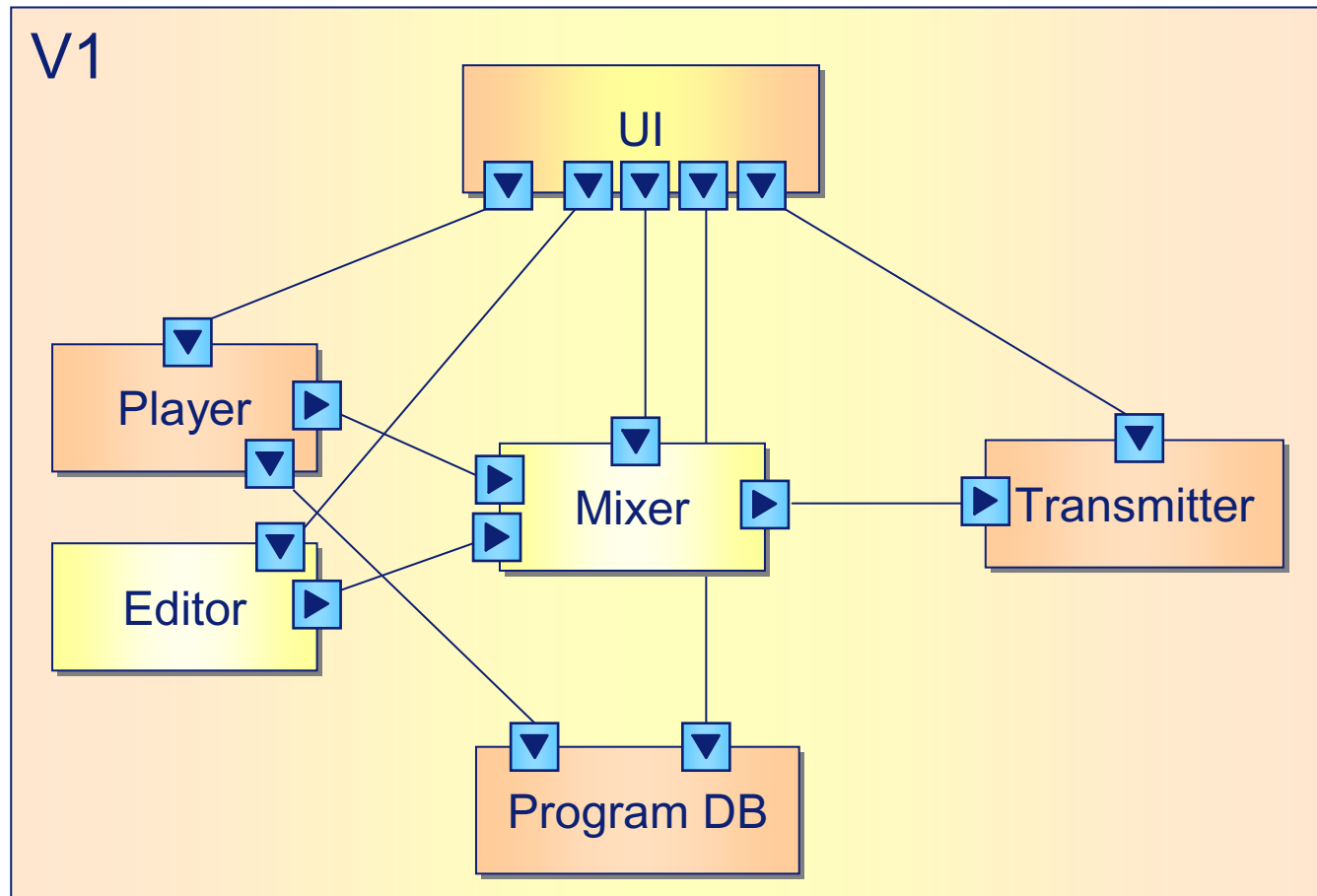
(1) The Authoring Tool



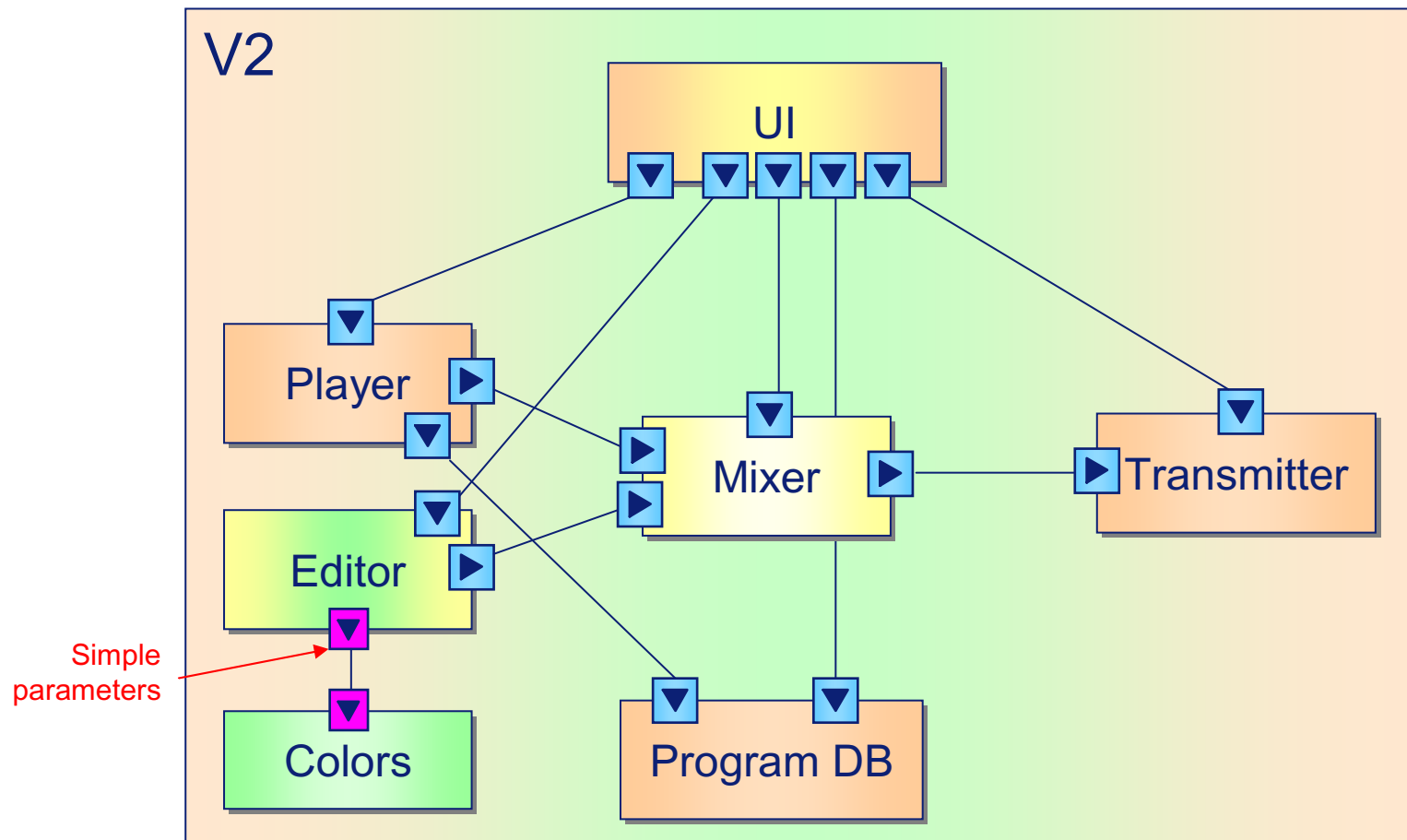
The Authoring Tool, version 0



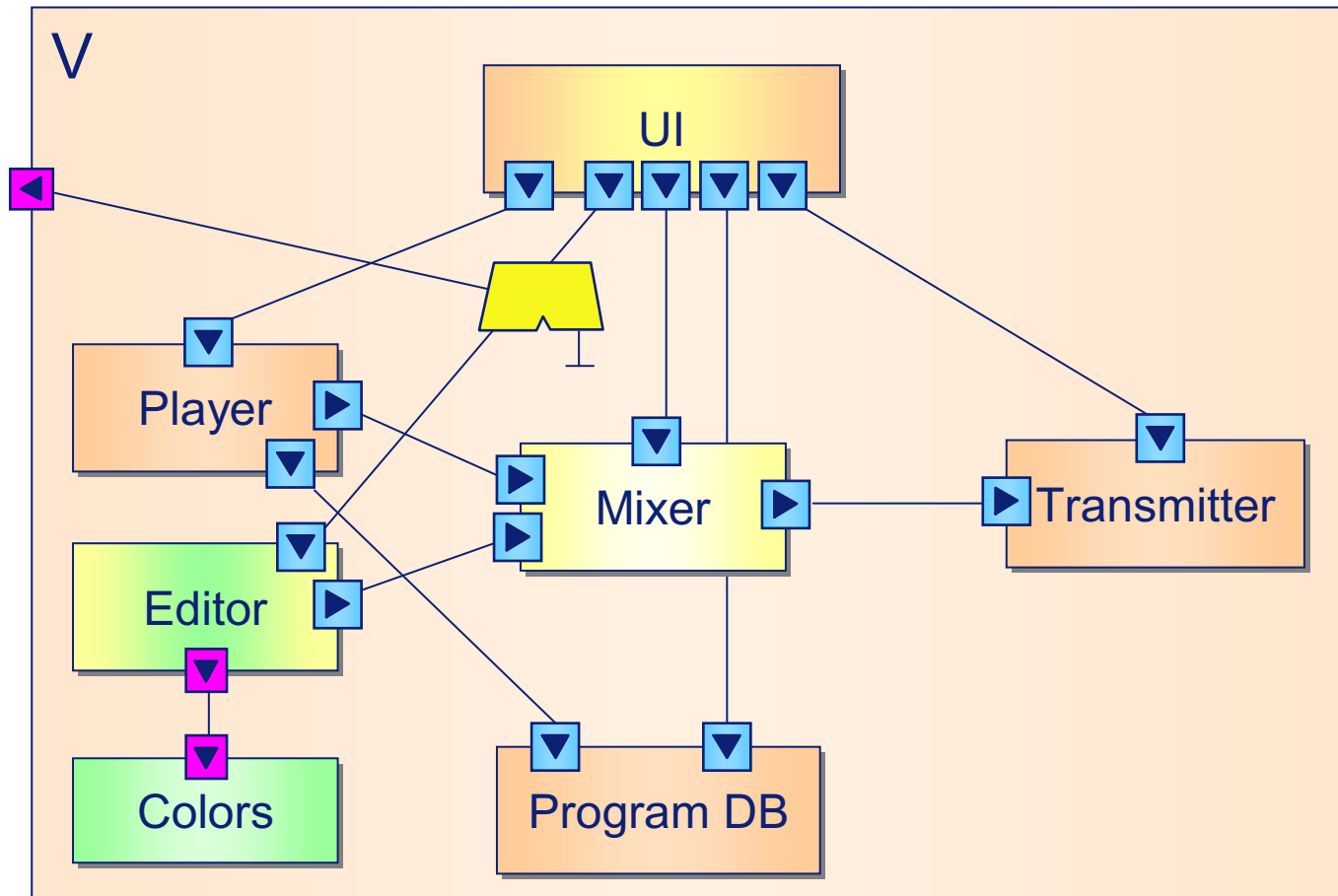
Adding a text page



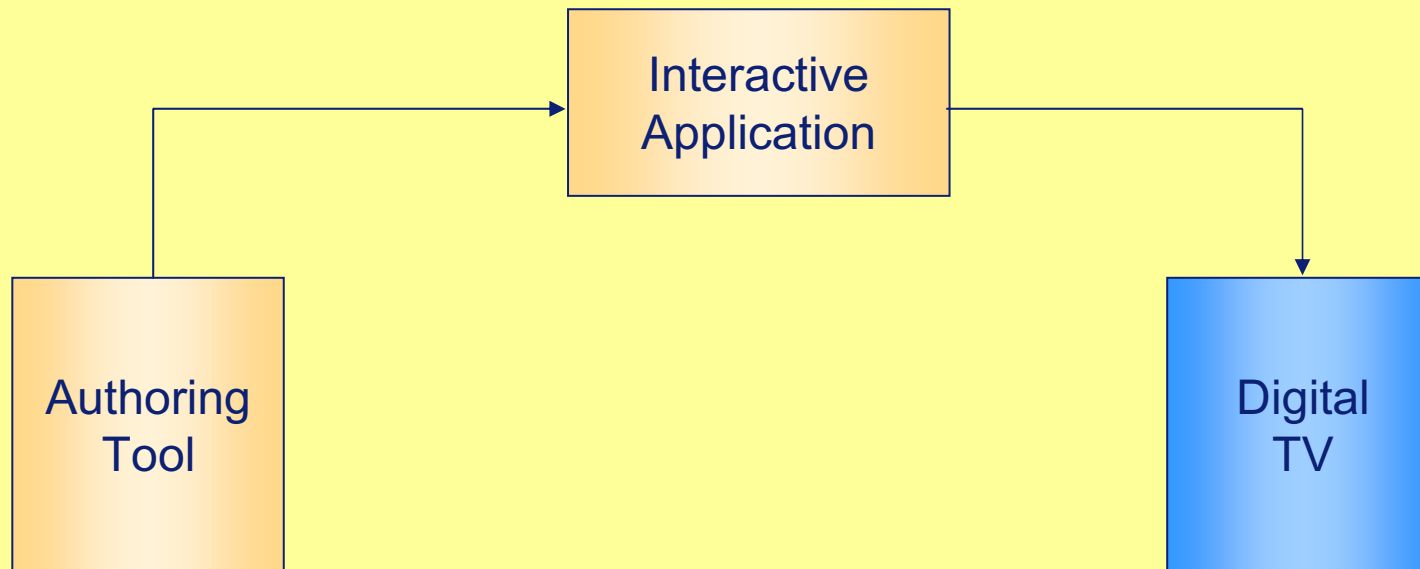
Customizing Colors



Creating a Configurable Product



(2) The Digital TV



Same Principle J

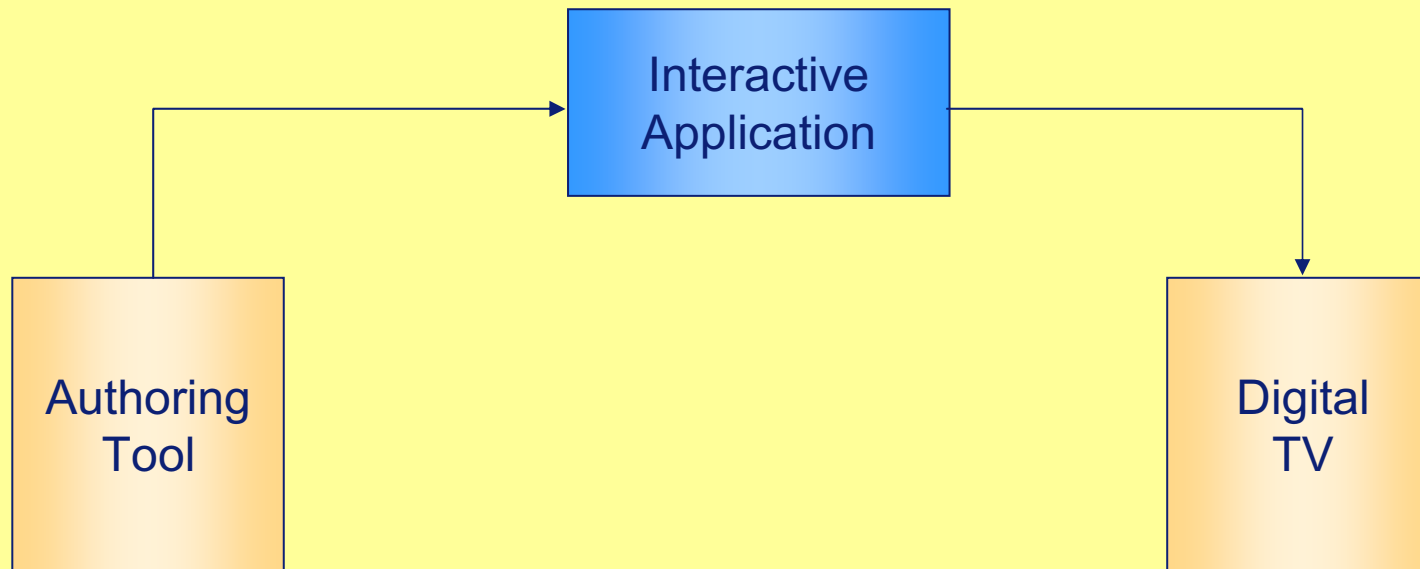
This was what Koala was designed for...



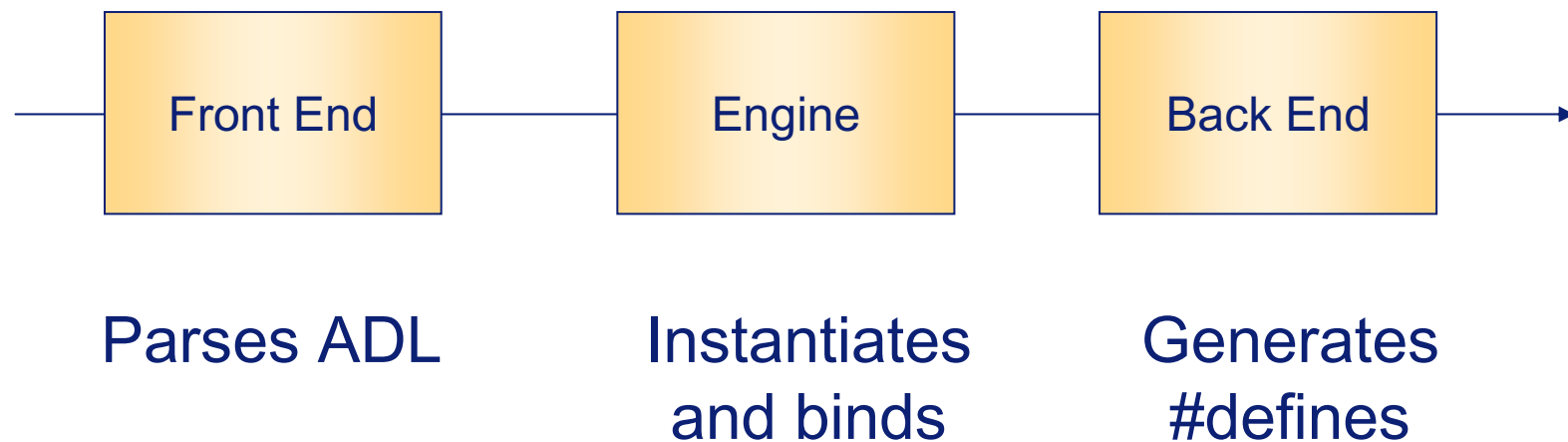
Koala inside?

But actually it must work with *any* kind of TV

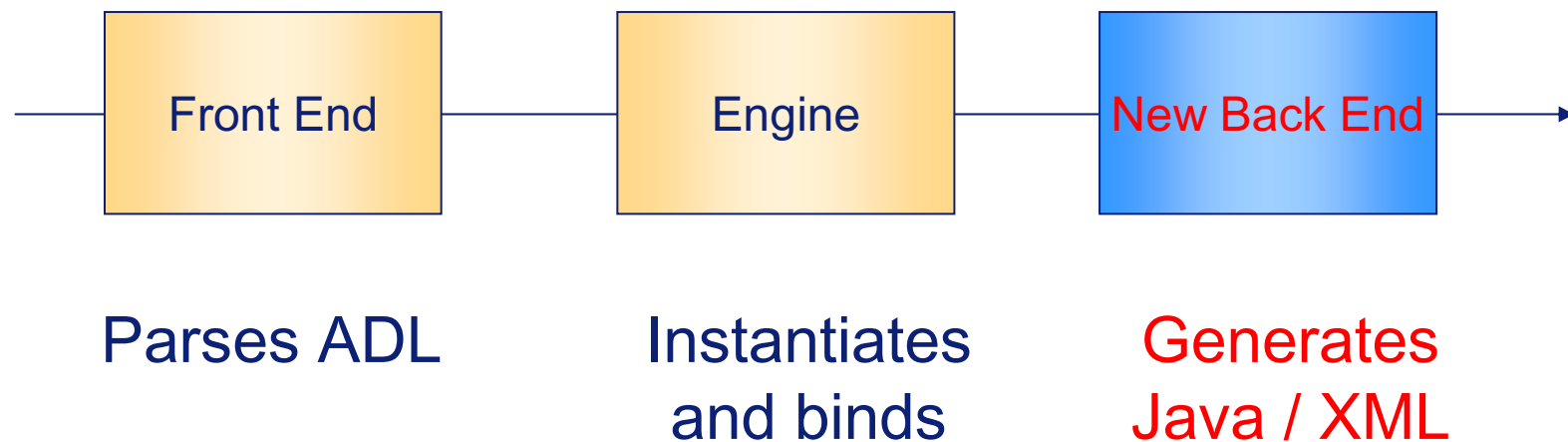
(3) The Interactive Application



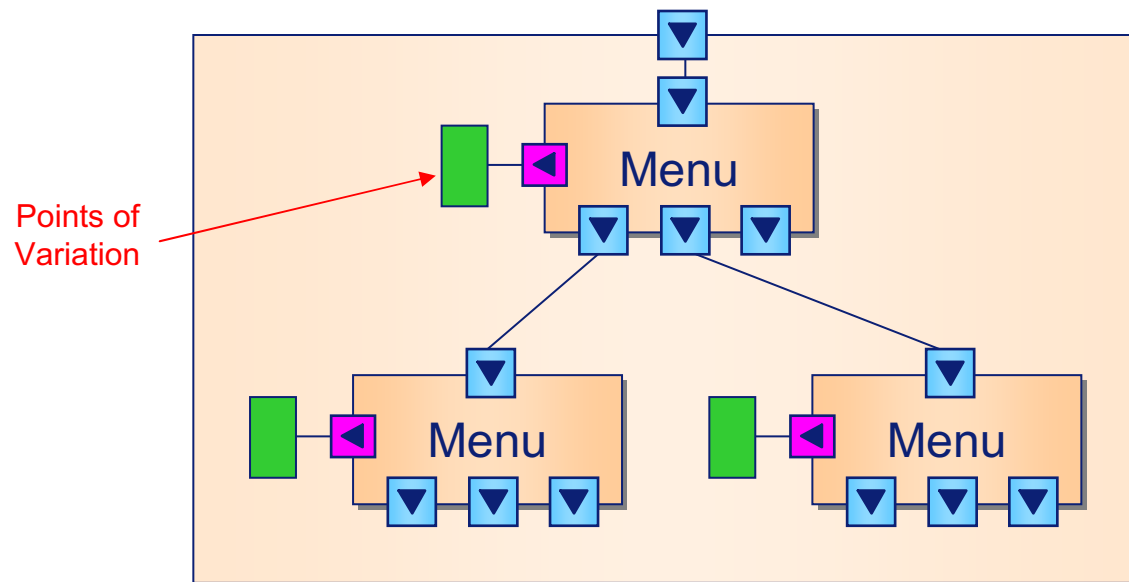
The Koala Compiler



Add a new back end



Build Menu Structure in Koala

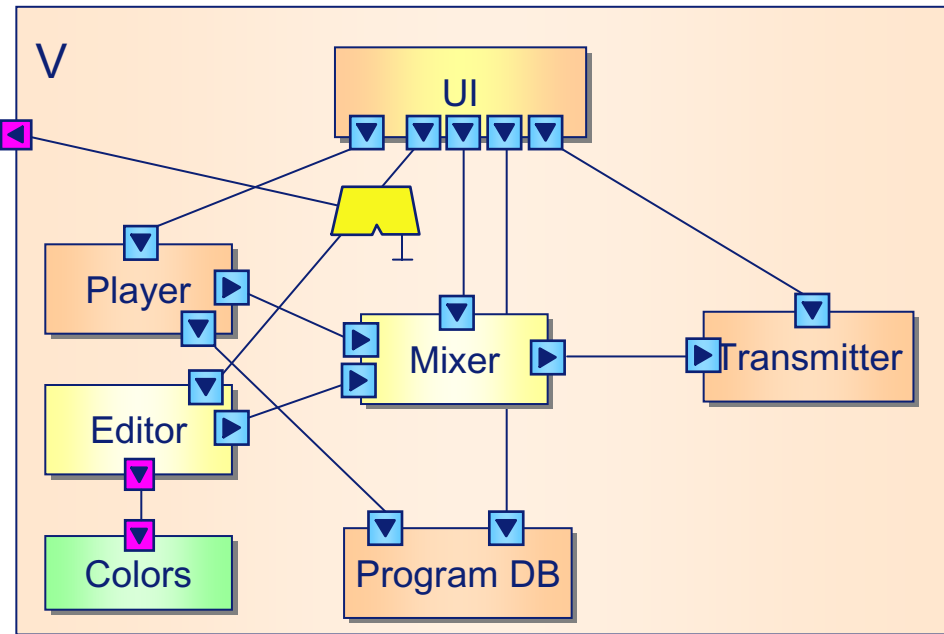
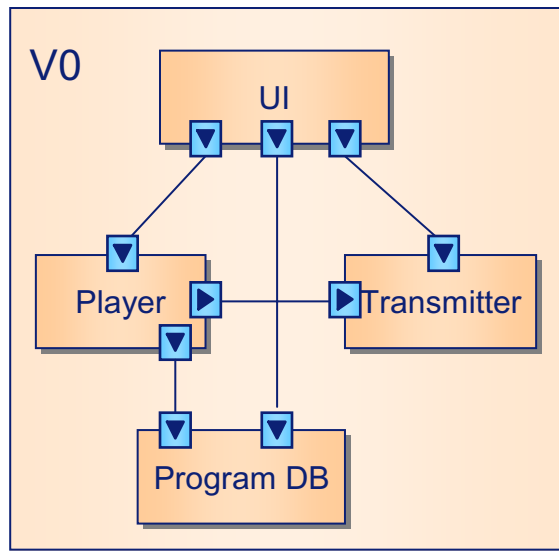


This was the domain of the predecessor of Koala...

Questions

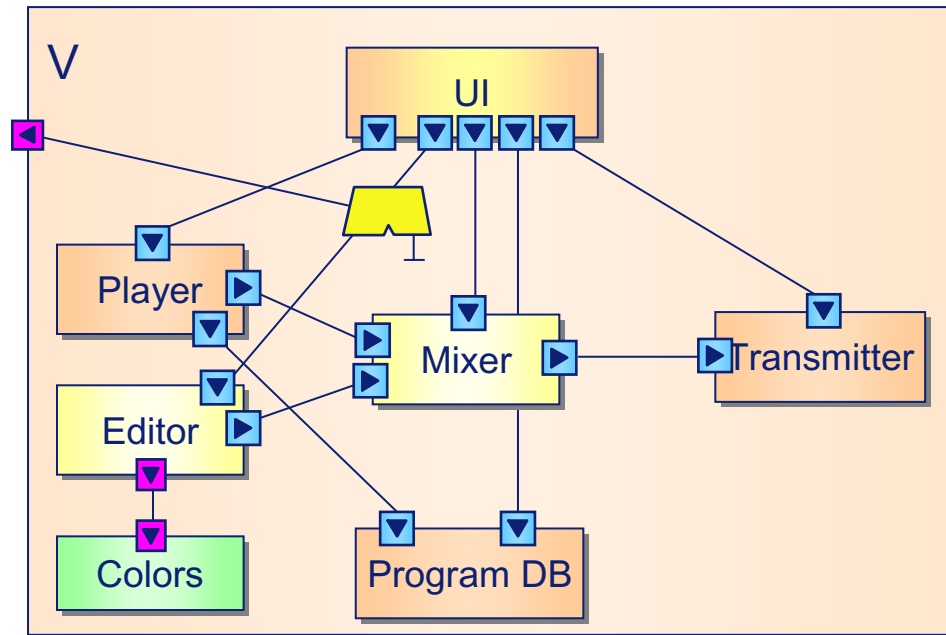
1. How large a portion of a product is automatically derived? Please answer in terms of some reasonably precise measure, such as percent of modules, classes, or KNCSL, or coverage in a feature model.
2. How are new features and functionality developed? Give an example, if possible.
3. What is the cost and time to create a new feature or change the application platform, e.g., in hours of effort as a fraction of effort needed to create the application engineering environment? Alternatively, how would you estimate the cost and time?

What portion is automatically derived?



Depends on where you start from \mathcal{J}
 Depends on what you call derivation \mathcal{J}

How are new features developed?



- Changing the component structure...
- Adding an existing component...
- Creating and adding a new component...

Cost of new feature...

Depends \mathcal{J}

- If a new component is needed, it has to be developed anyway
- Koala descriptions are currently 10% of the code base
 - they replace C header files

Product Creation Effort

Philips Consumer

10-20 product types/year
Millions of products/year

Uses Koala and C

Can spend a few person
years on each product

Product derivation is
engineering task

Philips Medical

100 products/year
Each product is unique

Uses a similar model in C#

Specific product per hospital
/ department / doctor

Product derivation should be
highly automated

Main Lesson Learned

There is no **single** Product Line Problem!!!

Thank You

Panel

Testing in a Software Product Line

Klaus Pohl



Lero
The Irish Software Engineering Research Centre
Univ. of Limerick, Ireland
www.lero.ie

Software Systems Engineering
University of Duisburg-Essen, Germany
www.sse.uni-due.de



Overview

- Introduction
 - Very brief introduction to product line testing
 - Our small example
 - The four panellist
- Four presentations (10 Minutes)
 - Please ask clarification questions at the end of each presentation
 - ... and please interrupt if someone is talks over 10 Minutes
- Discussion of the panel statements



Testing in Software Product Line Engineering

... differs from testing in single development !?

(1) Variability

- increases the complexity
- How to deal with variability in domain artefacts?

(2) Two development processes (domain and application engineering)

- When to test what ?
- e.g. all assets in application engineering ?

(3) Commonalities are part of each product of the SPL

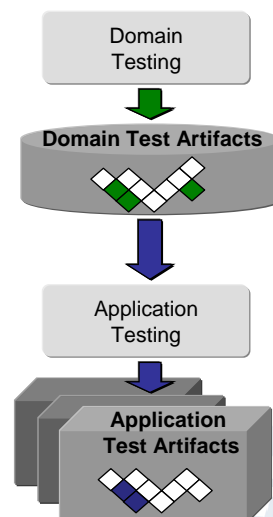
- defect in a commonality causes failure in all products
- How to avoid defects in commonalities?

(4) ...

Testing in Software Product Line Engineering (2)

- Should there be **two test processes**: domain tests and in application test?
- Should **systems test** be performed in domain engineering (or only in application engineering) ?
- **Can test artifacts be reused** in application testing? If so, which ones?
- Is there a **benefit of reusing domain test artifacts** and even domain test results in application engineering?
- Is there a benefit of **model-based test case derivation** of application test cases from domain test cases?

▪ ...



The Example

A simple E-Shop Product Line



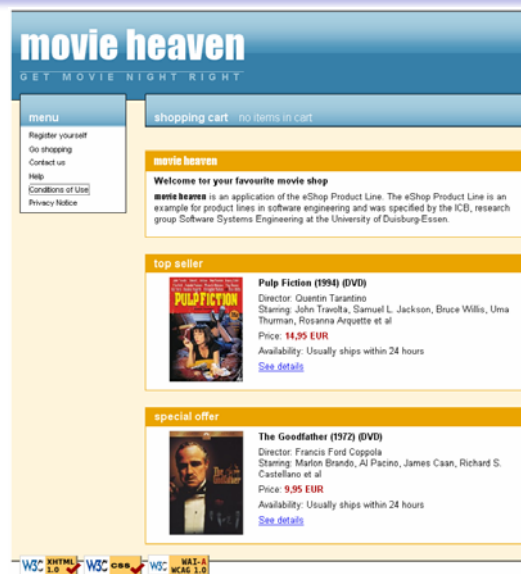
The E-Shop Product Line

■ Commonalities

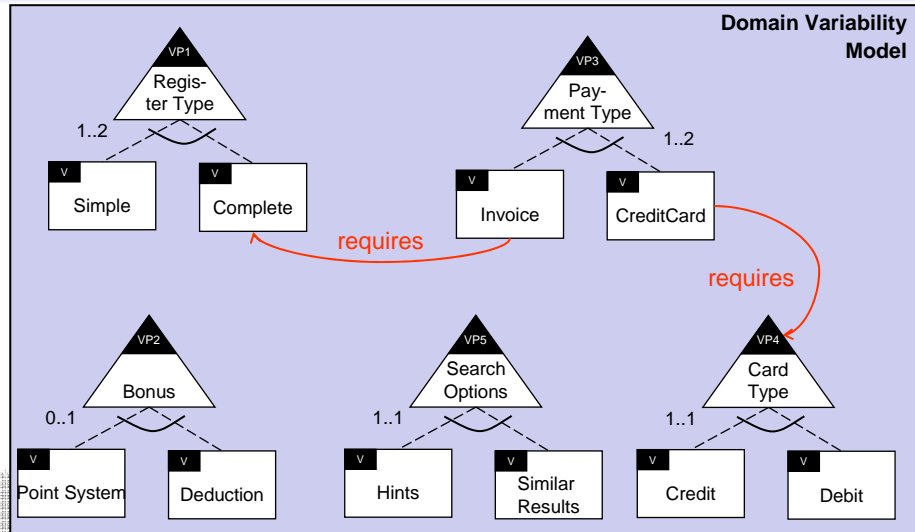
- register customer
- buy product
- search product

■ Variability

- different registrations
- different bonus programs
- different payment method
- different search options



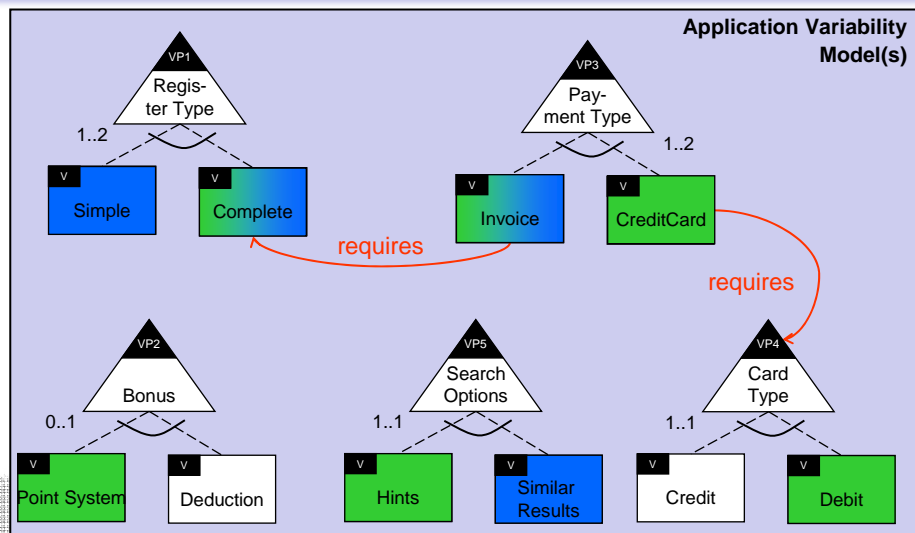
The E-Shop Product Line



SPLC Testing Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 7

The E-Shop Product Line



● chosen for Application 1 ● chosen for both ● chosen for Application 2

SPLC Testing Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 8

The Panelists

- **Georg Grütter**

- Robert Bosch GmbH, Germany



- **John D. McGregor**

- Clemson University, USA



- **Andreas Metzger**

- University of Duisburg-Essen, Germany



- **Tim Trew**

- Philips Research, The Netherlands



SPLC Testing Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 9

Short Summary of the Presentations:

- **Georg**

- perform risk-based SPL testing - only test the assets with the highest risk
 - don't create reusable test assets without knowing whether you reuse it or not

- **John**

- test core assets as far as possible in domain testing
 - detect remaining defects in application testing which are mainly caused by unexpected interactions

- **Andreas**

- test the commonalities and the most frequently used variants during domain engineering
 - test the selected remaining variants in application engineering

- **Tim**

- no domain testing ... but create core test assets, e.g. test models
 - reuse previous test results when testing new applications

SPLC Testing Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 10

Position Statement for testing the eShop product line

Georg Grütter
Corporate Research
Robert Bosch GmbH

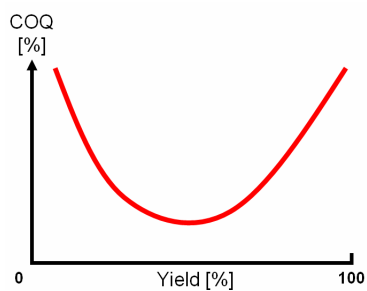
1

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.



BOSCH

Guiding Principles



- Test profitably
- YAGNI
- Fly like a rocket

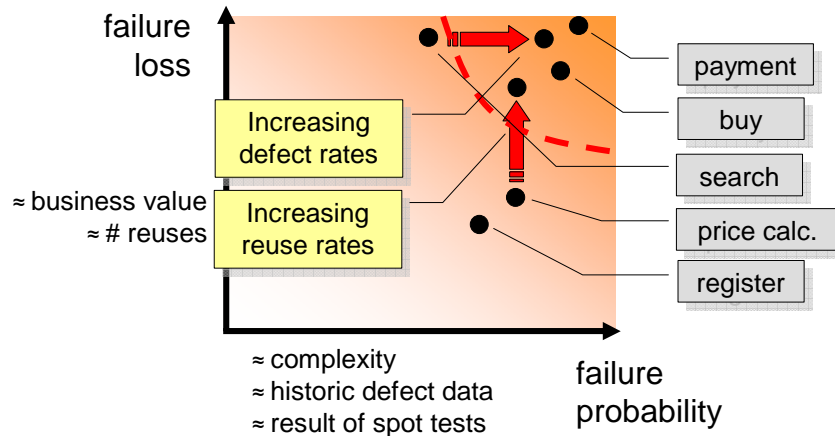
2

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.



BOSCH

Choosing test subjects in Domain Engineering



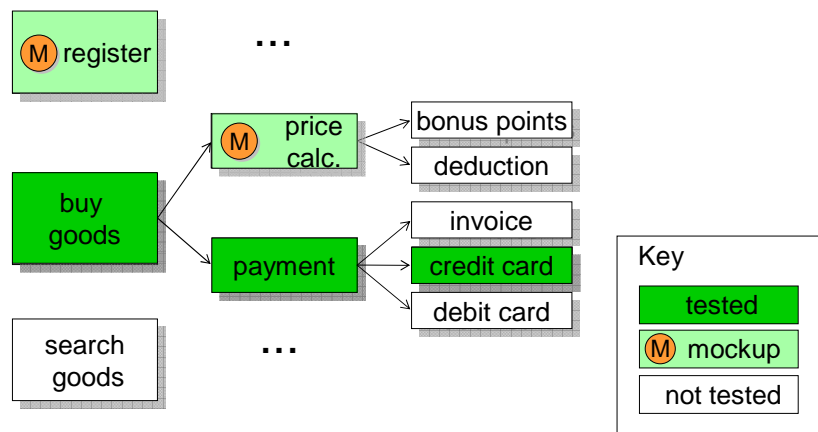
3

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.



BOSCH

Variable test depth in Domain Engineering



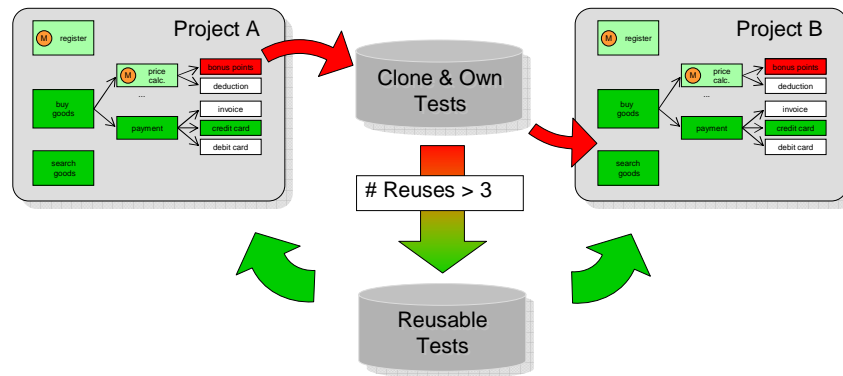
4

CR/AEA | 2006-08-02 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.



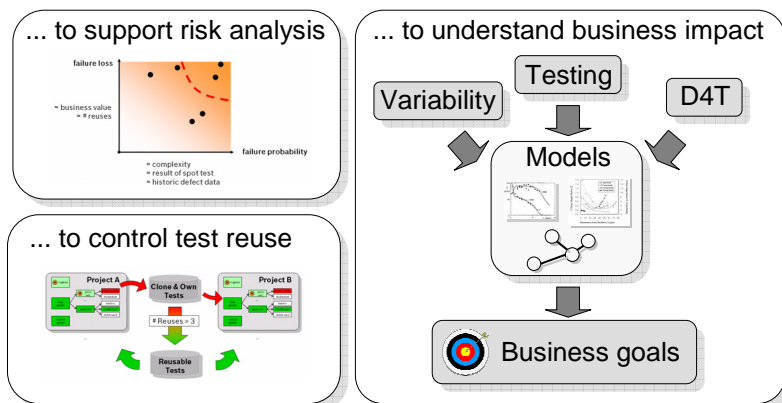
BOSCH

Testing in Application Engineering



Testing in General

→ Measure, understand and model the testing process ...



Testing the eShop Product Line

John D. McGregor
SEI
Clemson University
Luminary Software

The assignment

- Design the system test of the eShop
- But, first let me point out that if strategic reuse is the key to product line success
- Then testing early is essential to ensure a core asset base that possesses all the qualities that are sought

Desired Profile

This is the preferred profile of defect detection

- Unit testing – 90% of defects
- Integration testing – 9%
- System testing – 1%

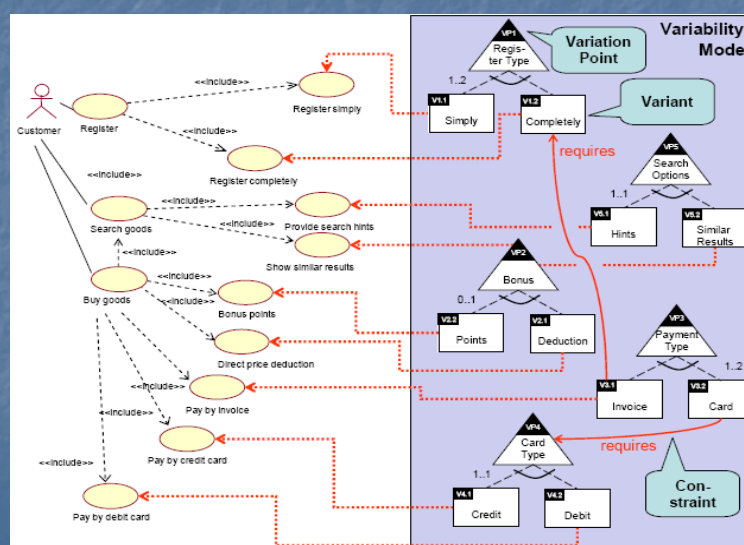
Core Asset Testing

- ALL core assets are validated as soon as they are created
 - ATAM for the software architecture
 - Guided Inspection for the design assets
 - Combinatorial test techniques for software
- Test coverage should be comprehensive
 - Priorities are determined by business goals
 - Coverage levels determined by domain

System Tests in Product Line Development

- Test sample applications
 - Combinations of choices at variation points
 - Full scale integration tests
 - Limited to what can be built at the moment
 - Involve product builders early
- Test specific application
 - Tests a specific product prior to deployment
 - Rerun some of the selected products' test cases
 - Feedback results to core asset builders

Now to the problem



OATS-based Test Strategy for Identifying Selected Applications

- Orthogonal Array Testing System (OATS)
- One factor for each variation point
- One level for each variant within a factor
- "All combinations" is usually impossible but pair-wise usually is manageable.
- Constraints identify test cases that are invalid

Factor	Level	Constraint
VP1	VP1.1	
	VP1.2	
	Both	
VP2	None	
	VP2.1	
	VP2.2	
VP3	VP3.1	Requires VP1.2
	VP3.2	Requires VP4
	Both	
VP4	VP4.1	
	VP4.2	
VP5	VP5.1	
	VP5.2	

L2x7 – 7 factors each with 3 levels

- Use standard pre-defined arrays
- This one is larger than needed but that will work
- Each of the factors has values 0,1,2
- Defined to include all pair-wise combinations

1	2	3	4	5	6	7	factor
0	0	0	0	0	0	0	
1	1	1	1	1	1	0	
2	2	2	2	2	2	0	
0	0	1	2	1	2	0	
1	1	2	0	2	0	0	
2	2	0	1	0	1	0	
0	1	0	2	2	1	1	
1	2	1	0	0	2	1	
2	0	2	1	1	0	1	
0	2	2	0	1	1	1	
1	0	0	1	2	2	1	
2	1	1	2	0	0	1	
0	1	2	1	0	2	2	
1	2	0	2	1	0	2	
2	0	1	0	2	1	2	
0	2	1	1	2	0	2	
1	0	2	2	0	1	2	
2	1	0	0	1	2	2	

Variants

1	2	3	4	5	6	7
0	0	0	0	0	0	0
1	1	1	1	1	1	0
2	2	2	2	2	2	0
0	0	1	2	1	2	0
1	1	2	0	2	0	0
2	2	0	1	0	1	0
0	1	0	2	2	1	1
1	2	1	0	0	2	1
2	0	2	1	1	0	1
0	2	2	0	1	1	1
1	0	0	1	2	2	1
2	1	1	2	0	0	1
0	1	2	1	0	2	2
1	2	0	2	1	0	2
2	0	1	0	2	1	2
0	2	1	1	2	0	2
1	0	2	2	0	1	2
2	1	0	0	1	2	2

map →

Factor	Level	Constraint
VP1	VP1.1	
	VP1.2	
	Both	
VP2	None	
	VP2.1	
	VP2.2	
VP3	VP3.1	Requires VP1.2
	VP3.2	Requires VP4
	Both	
VP4	VP4.1	
	VP4.2	
VP5	VP5.1	
	VP5.2	

Mapped Array

VP1	VP2	VP3	VP4	VP5	
VP1.1	None	VP3.1	VP4.1	VP5.1	c
VP1.2	VP2.1	VP3.2	VP4.2	VP5.2	
Both	VP2.2	Both	2	2	x
VP1.1	None	VP3.2	2	VP5.2	x
VP1.2	VP2.1	Both	VP4.1	2	x
Both	VP2.2	VP3.1	VP4.2	VP5.1	
VP1.1	VP2.1	VP3.1	2	2	x,c
VP1.2	VP2.2	VP3.2	VP4.1	VP5.1	
Both	None	Both	VP4.2	VP5.2	
VP1.1	VP2.2	Both	VP4.1	VP5.2	
VP1.2	None	VP3.1	VP4.2	2	x
Both	VP2.1	VP3.2	2	VP5.1	x
VP1.1	VP2.1	Both	VP4.2	VP5.1	
VP1.2	VP2.2	VP3.1	2	VP5.2	x
Both	None	VP3.2	VP4.1	2	x
VP1.1	VP2.2	VP3.2	VP4.2	2	x
VP1.2	None	Both	2	VP5.1	x
Both	VP2.1	VP3.1	VP4.1	VP5.2	

- Every row is a system under test.
- 17 test products vs 72 possible combinations
- Columns 4 and 5 have more levels than are needed. Columns 6 and 7 are not needed at all.
- Where a "2" is in the column, this indicates the tester could repeat a value (one of the variants).

Legend: c = constraint; x = any choice for constant will work

Test suite

- Test cases for these sample applications must also be built, but
- They can be selected using functional and structural techniques, and
- Then constructed using generation technology

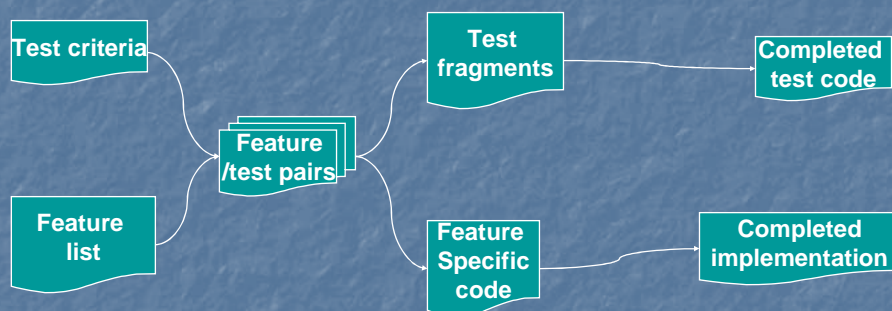
Factor test cases

- Test cases break at each variation point to allow the test case to be varied just as we have varied the product.
- Use similar mechanisms for variation in test cases as those used in the code
- Assemble the test cases in parallel to assembling the product code.

XVCL-based generation

- Frame-based technology
- Simultaneously generate system code and system test cases
- Tests are NOT embedded in the product but they are embedded side by side in the generation technology

Structure



- Xml-based Variant Configuration Language uses individual frames to compose an asset.

Top-level frame

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="main.xvcl">
  <!--add variation points that are selected-->
  <set-multi var="variationPoints" value="vp1.1,vp1.2"/>
  <!--set actual file names for variables-->
  <set var="codefile" value="c:\out.txt"/>
  <set var="testfile" value="c:\testout.txt"/>
  <adapt x-frame="baseSystem.xvcl"></adapt>
  <adapt x-frame="registerUseCase.xvcl"></adapt>
  <!--add other use cases-->
</x-frame>
```

Product variants

Generation targets

Use Case Frame-1

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE x-frame SYSTEM "default">
<x-frame name="registerUseCase.xvcl" outfile="?.@codefile?">
  code to start the registerUseCase
  <adapt x-frame="testBeginRegisterUseCase.xvcl"></adapt>
  <while using-items-in="variationPoints">
    <select option="variationPoints">
      <option value="vp1.1">
        <adapt x-frame="registerSimply.xvcl"></adapt>
      </option>
    </select>
  </while>
```

Use Case Frame-2

code between the variation points

```
<adapt x-  
  frame="testMiddleRegisterUseCase.xvcl"></adapt>  
<while using-items-in="variationPoints">  
  <select option="variationPoints">  
    <option value="vp1.2">  
      <adapt x-frame="registerCompletely.xvcl"></adapt>  
    </option>  
  </select>  
</while>
```

code to complete the registerUseCase

```
<adapt x-frame="testEndRegisterUseCase.xvcl"></adapt>  
</x-frame>
```

Variant frame

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<!DOCTYPE x-frame SYSTEM "default">
```

```
<x-frame name="registerSimply.xvcl"  
  outfile="?.@codefile?">
```

code for registerSimply

```
<adapt x-  
  frame="testRegisterSimply.xvcl"></adapt>  
</x-frame>
```

Conclusion

- Test early, test often
- Select test cases to achieve specific goals, e.g., maximize defect detection
- Structure test cases for reuse with variability mechanisms similar to those in the code or other assets
- Logically attach test cases to use cases or features

Panel on Testing in a Software Product Line

Position Statement

Andreas Metzger

Software Systems Engineering
Institute for Computer Science and
Business Information Systems (ICB)
University of Duisburg-Essen, Germany
www.sse.uni-due.de



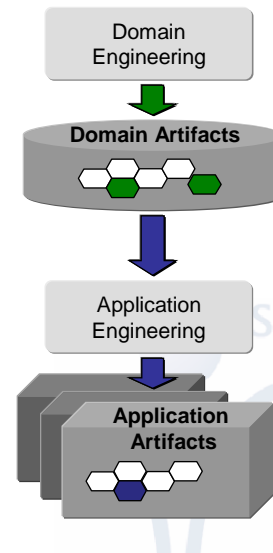
Statement

Balance the testing activities
between domain and application engineering.



This means...

- Domain Testing:
 - Test **commonalities**
 - Test **most frequently used variants**
- Application Testing:
 - Test **remaining variants** if contained in an application



Domain Testing in the E-Shop Product Line

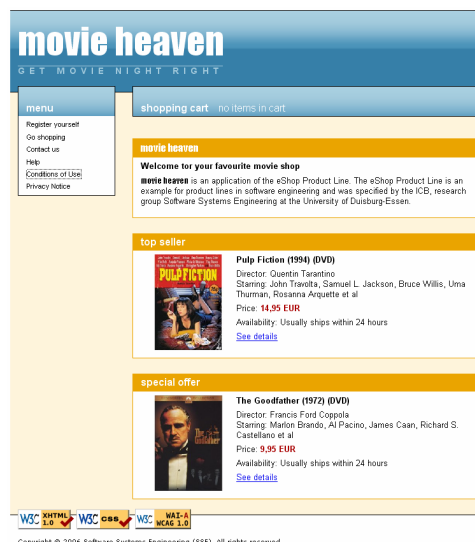
■ Commonalities

- register customer
- buy product
- search product

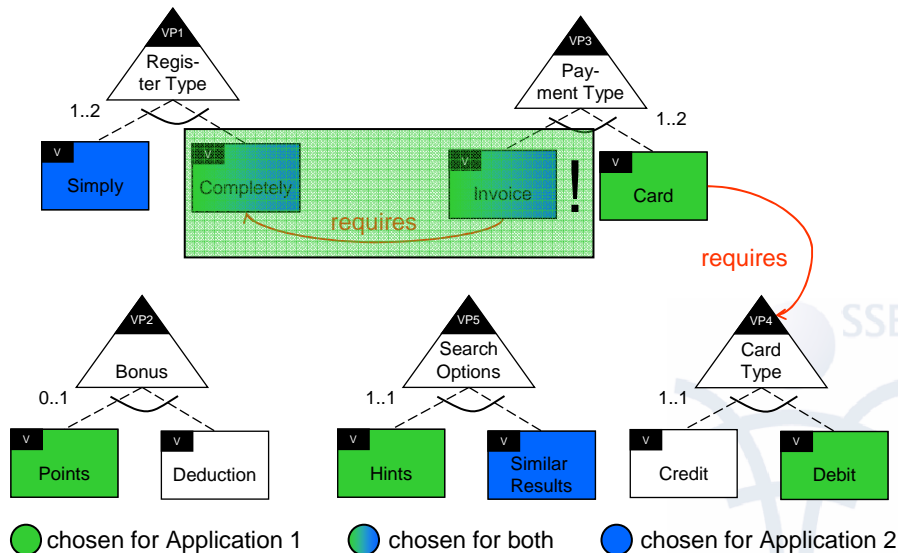


■ Variability

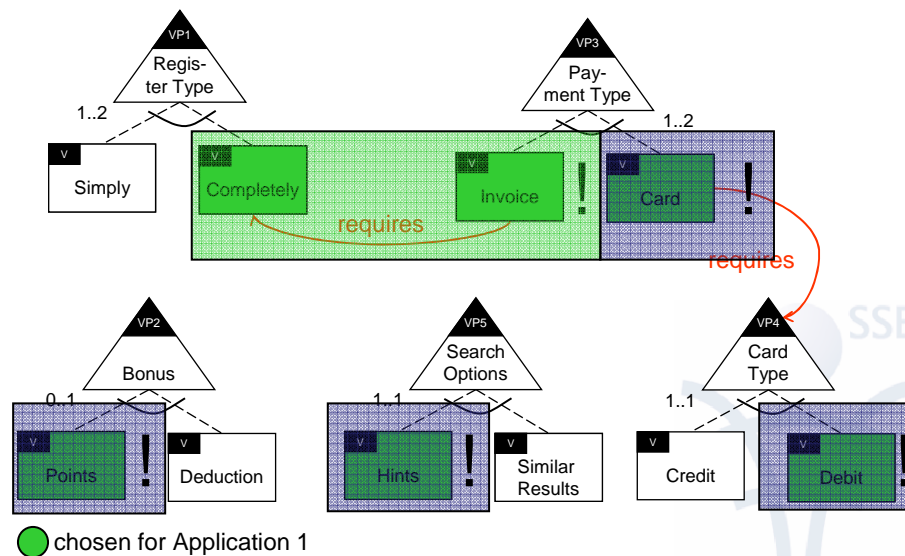
- type of registration
- kind of bonus program
- type of payment
- search options



Domain Testing in the E-Shop Product Line



Application Testing in the E-Shop Product Line Application 1



Benefits

- **Reduction of probability** that a **defect is introduced** in all or many of the SPL applications



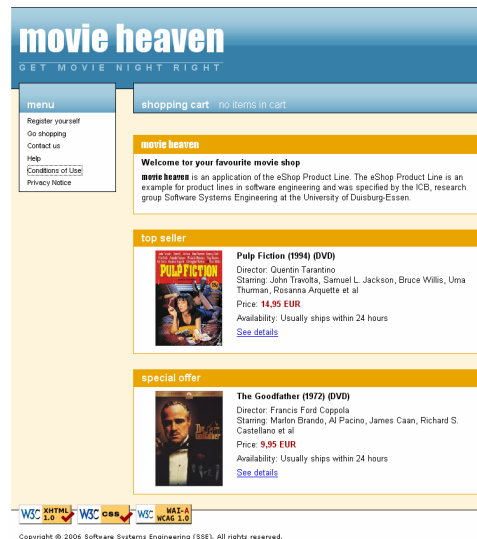
- **Reduction of costs** when compared to a comprehensive test of all core assets
 - **Costs** for testing „rarely“ used variants is „**delayed**“ until application development



PHILIPS

Testing the eShop Product Line

Tim Trew
Philips Research
SPLC, Baltimore, August 2006



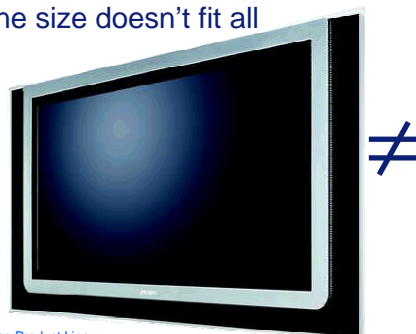
PHILIPS

My position

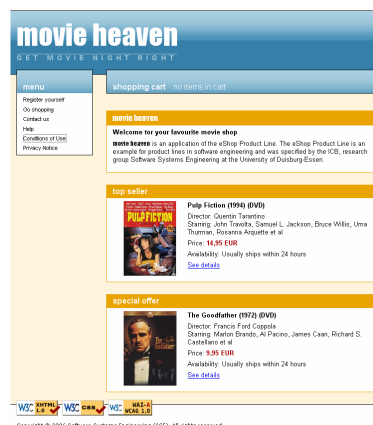
- Only test applications (not core software assets)
- Use reusable domain test assets
- Leverage the results of testing previous applications when testing new variants

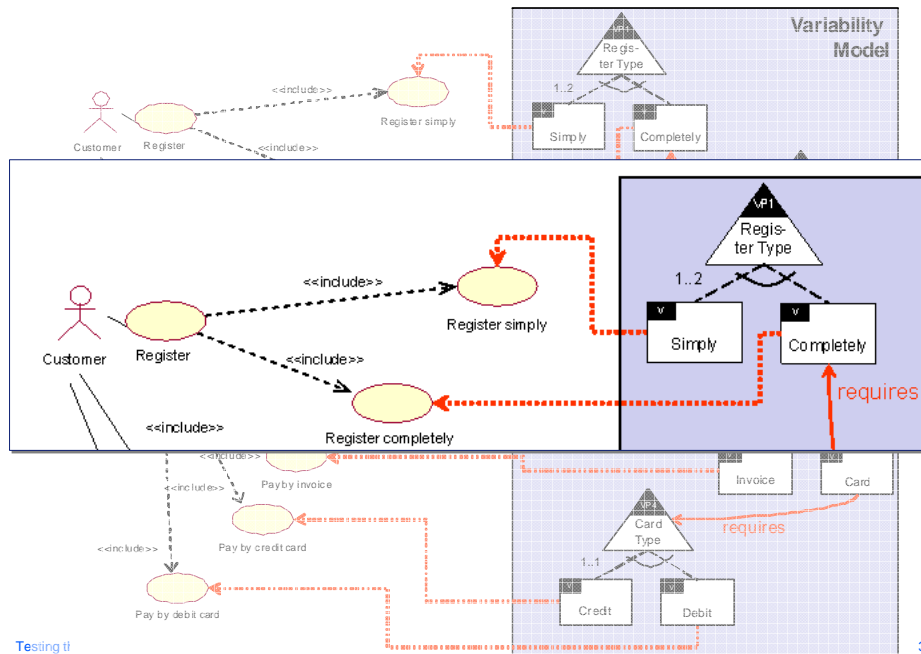
Caveat

- One size doesn't fit all



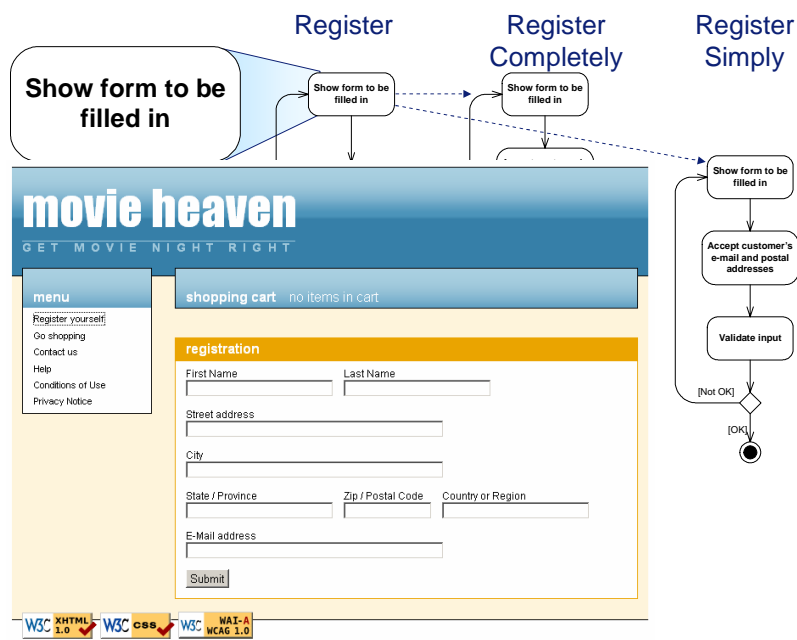
Testing the eShop Product Line





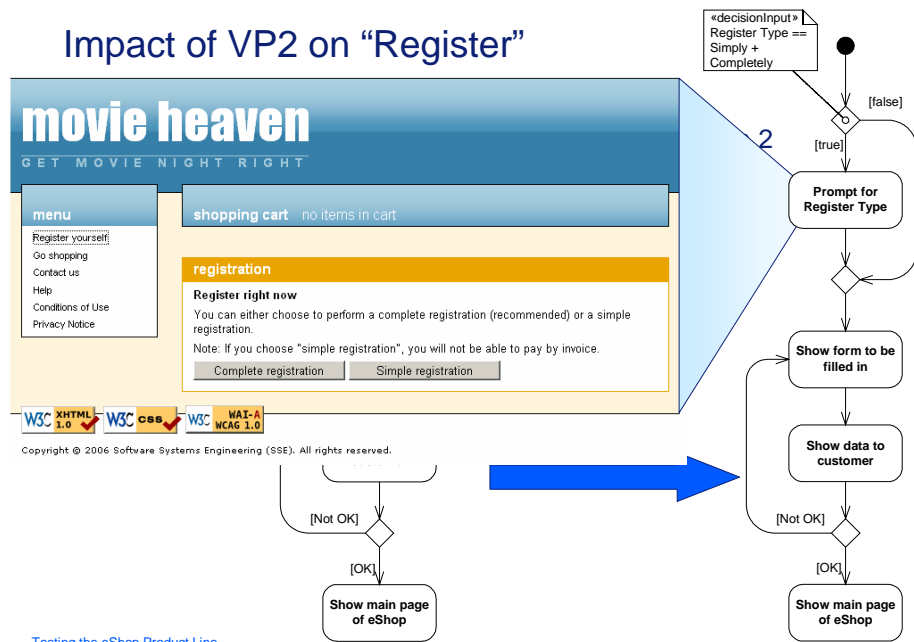
Testing ti

3



4

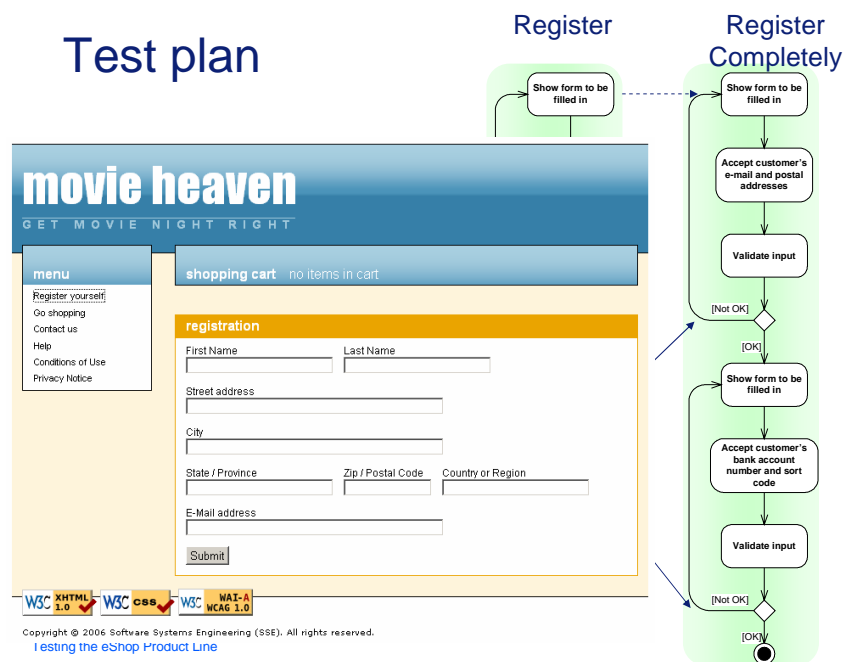
Impact of VP2 on "Register"



Testing the eShop Product Line

5

Test plan



Testing the eShop Product Line

6

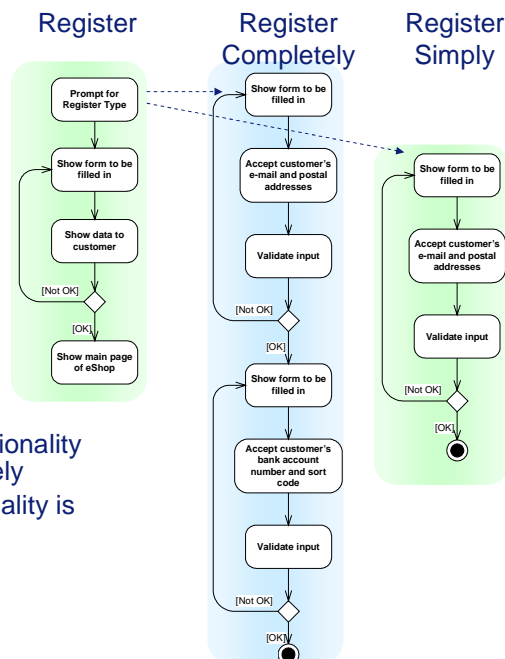
Test plan

Application 2

- Test “Register” and “Register Simply” to MCDC/LCSAJ coverage
- Test that “Register Completely” is executed

Test strategy

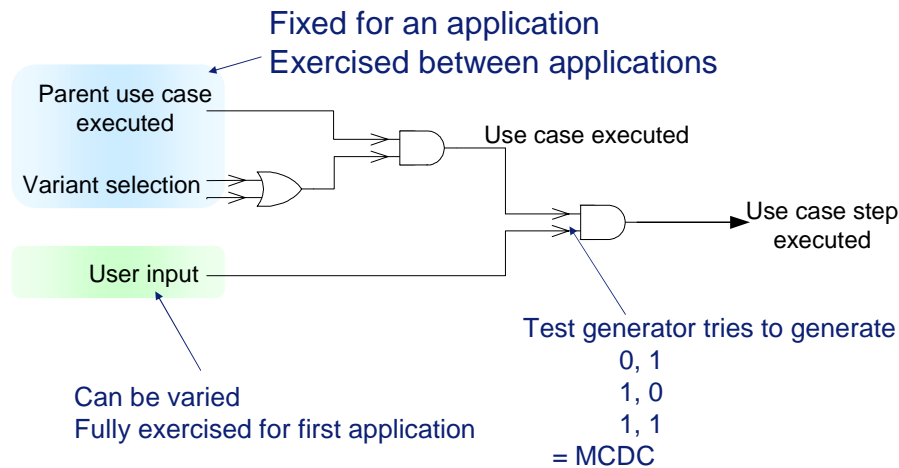
- For each application, test new functionality and functionality impacted by VPs extensively
- Check that reused functionality is invoked at the correct time



Generating test cases

- We don't want to depend on knowledge of the order in which applications will be developed when developing test cases
 - Too inflexible to changes in product line requirements
 - Too uninteresting to suggest at SPLC ☺
- Develop domain test artefacts that can be configured for application testing
 - State model of all possible variants
 - Activity diagram LSCAJ \equiv state model transition coverage
 - Cause-effect graph
 - Good traceability to use cases with variation points

Exploitation of previous test results



```
TEST#1 -- eShop REGISTRATION
  Variant: Register Completely available
  Action: User activates registration
  Usecase start: Register
    Usecase start: Register Completely
      Check: The system presents part 1 of the
            registration form
      Action: The customer fills in the registration
            form with infeasible e-mail address
      Check: The system rejects the registration
      Check: The system presents part 1 of the
            registration form
      Action: The customer fills in the registration
            form correctly
      Check: The system accepts the registration
      Check: The system presents part 2 of the
            registration form
      Action: The customer fills in the account
            information correctly
      Check: The system accepts the registration
    Usecase end: Register Completely
  Usecase end: Register
```


Rationale

- eShop has loosely-coupled use cases
 - Main use case determines whether a sub-ordinate use case executes, not its detailed execution
 - Allows test results to be reused between applications
- Variability determines availability of use cases
 - Not fine-grain parameterisation
- Dependencies between use cases make it more difficult to test components in isolation
 - E.g. “Buy goods” requires that there is a registered user and that some goods have been selected
- We will have to do some application testing to ensure that the variation points have been set correctly
- The execution of the application is easy to automate
 - E.g. for regression testing



lero

*THE IRISH SOFTWARE
ENGINEERING RESEARCH CENTRE*

SPLC 2006 Research Panel:

Product Line Research: Lessons Learned
from the last 10 years and Directions for
the next 10

Panel Moderator – Liam O'Brien

24 August 2006

Overview

The panel members will explore:

- Lessons learned and outcomes from the past 10 years
- Directions and potential outcomes for the next 10 years

Examine the lessons, outcomes and directions from the practitioners' perspective.



Panel Members

- Paul Clements – Software Engineering Institute, USA
- Kyo Kang, Pohang University of Science and Technology, Korea
- Dirk Muthig, Fraunhofer IESE, Germany
- Klaus Pohl, Lero – The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany



Industry Judges

- Bruce Trask, MDE Systems
- Gunther Lenz, Siemens

Format

The format for the panel will be:

- Panellist's presentation: 10 minutes
- Judge's comments: 3 minutes
- Panellist's response: 2 minutes

Following the presentations there will be a general/interactive discussion.

Product Line Research

Panel Statement

Dirk Muthig (Ph. D.)
Fraunhofer IESE
Kaiserslautern, Germany

dirk.muthig@iese.fraunhofer.de

Achievements – Community

Product lines as
holistic reuse approach
(cost, ttm, quality, ...)

- Definition of Product Lines and Product Line Engineering
 - Terminology
 - Life cycle model
 - Product line criteria (PLHoF)
- Knowledge Base and Network
 - Books and Papers
 - Websites
 - Conferences

Achievements – Industrial Success Stories

Practical issues as
driver of
PL research activities

Product lines as
vision and driver of
improvement activities

- Collecting success stories
 - Situations after product line introduction
 - Examples
 - CelsiusTech
 - Cummins
- Creating success stories
 - Companies that have been observed during transition from traditional to product line development
 - Examples
 - market maker
 - Salion

Problems – Adopting Product Line Technology

organization-specific

domain-specific

market-specific

“it depends ...”

- Descriptions of product line methods in literature are, in general, too abstract
- Coverage of existing case studies is too small to illustrate product line approach as a whole
- Size and complexity of existing case studies are too small to clearly motivate need for variability management
- Different product-line methods and techniques cannot be compared objectively
- It is difficult for an organization to select, introduce, and apply product line technology practically (and thus to get all its benefits)

Outlook – PL Research

PL research
must be possible
without years of
industrial experience

- PL-specific aspects of all kinds of SE practices, e.g.,
 - Capturing family requirements
 - Testing of generic components
 - ...
- PL-specific topics, e.g.,
 - Variability management, modeling, ...
 - Scoping
 - Economic models
 - ...
- Consolidation and „standardization“
- Systematic validation (beyond action research)

Slide 4

Software Product Line Research Topics

Kyo Kang

SPLC2006

August 24, 2006



**Pohang University of Science and Technology
(POSTECH)**



- Technical advances
- Technology
- Process
- Management



- Paradigm change
 - From single systems to product line/family
- Commonality and variability analysis
 - Feature analysis
- Components and architectures (from objects and collaborations)
 - Variants and variation points
 - “high option potentials”
- Domain specific languages and generators



- Domain analysis
 - Different domains may require different approaches
 - Service analysis may be good for business applications domains
 - Goal analysis may be good for some embedded controller applications domains
 - “Goal -> Service” as a unified method?
 - Modeling mechanisms
 - Feature model is popular but many extensions
 - Should this be standardized?
 - Formalization
 - Deciding the right level of abstraction; how to structure
 - Feature explosion problem
 - How to model, analyze, and manage
 - Feature interaction problem



- Goal-oriented assembly and adaptation of components
 - Knowledge-based adaptation
 - Quality attributes or user-goals (e.g., balanced use of equipments)
- Going from domain analysis to architecture and component design
 - Designing architectures and components based on the analysis results (commonality and variability information)
 - SOA vs. agent-based vs. other architecture models
 - Building variability into architectures and components
 - Selecting appropriate mechanisms for the problem



- Specification of models
 - Reuse contexts and assumptions
- Verification of quality attributes of integrated systems
 - Safety, reliability, etc.
 - Detecting feature interaction problems
- Configuration management
 - Version control of components and architectures with multi-product nature
 - Evolution of the product line itself



- PL for systems in the newly emerging computing environments
 - Service Oriented Architecture
 - Ubiquitous computing environment
 - Dynamic binding of features
 - From compile-time engineering to run-time engineering
 - Embedment of SE knowledge in running systems
- Tools!

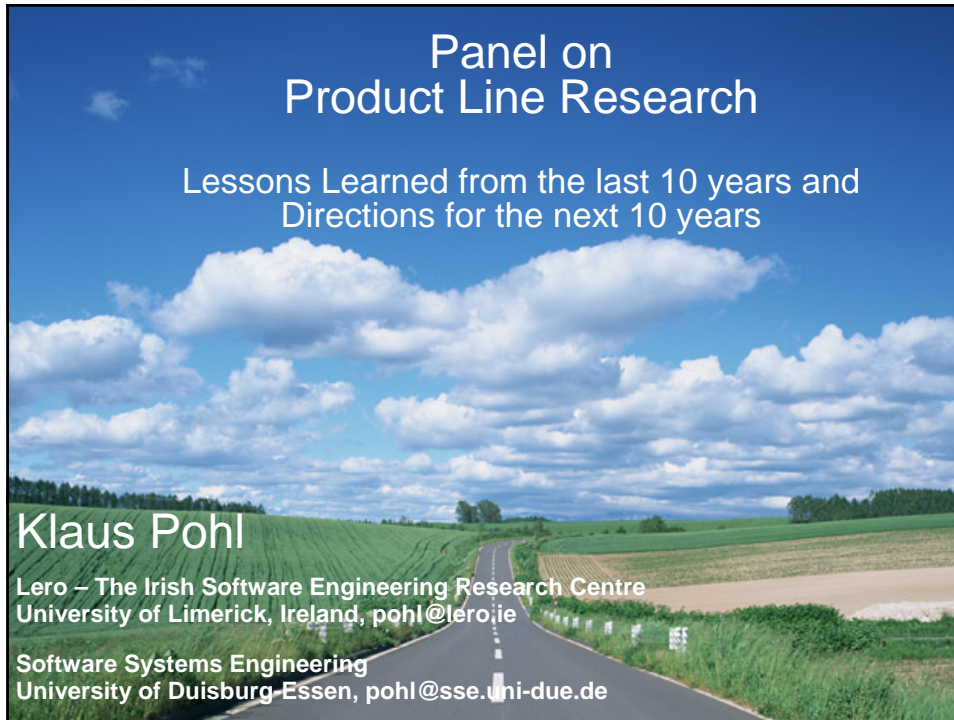


- How to change to PL-based organization
 - How to evolve: staged process model for reuse adoption
 - Key process areas
 - Best practices
 - Metrics
 - Key indicators: cost of production, time to market, project completion time, etc.
 - Relationship between reuse, quality, and productivity
 - Relationship between reuse and ROI for sustainability of a reuse program
- Process models
 - Proactive vs. reactive vs. extractive models
 - Best practices
 - PL process vs. agile methods



- ROI analysis
 - Estimating ROI from a reuse program
 - Estimating benefits from strategic market position
- Asset management (How to make PL-based development happen in an organization)
 - Who should develop assets (with variation points)
 - Who should maintain assets (variation management)
 - Who will be responsible for quality assurance
 - Who should enforce the use of assets
 - Models (best practices)
 - Centralized vs. distributed





Panel on Product Line Research

Lessons Learned from the last 10 years and
Directions for the next 10 years

Klaus Pohl
Lero – The Irish Software Engineering Research Centre
University of Limerick, Ireland, pohl@lero.ie
Software Systems Engineering
University of Duisburg-Essen, pohl@sse.uni-due.de

The last 10 years

Shift in focus

- from **purely technical aspects**, e.g. architecture, variability binding mechanism
to **non technical aspects**, e.g. variability management, economical aspects, processes, e.g. CMMI for product lines
- from **domain engineering**, e.g. reference architecture
to **application engineering**, e.g. product derivation

Maturity of the field

- the SPL community
 - **ONE community** resulting from the “merge” of the European and US initiatives
 - **SPLC as flagship conference**
 - growing in size and diversity
- increased **recognition in industry**



SPLC Research Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 2

The last 10 years



Software Product Line Engineering
is an established field
It works over 20 reported examples

BUT: Shift from single development
to product lines is difficult
... more difficult than from
C to real C++

SPLC Research Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 3

The next 10 years ...

Challenge No. 1 Software Intensive Systems ... not just software



Product Lines for Software Intensive Systems

Multi-functional technical systems:

- Software **empowers flexible interplay of physical component**, e.g.
 - A physical sensor can contribute to many functions
 - An actuator can be influenced by several functions
- Hard- and Software **feature-interactions**

Increased context-awareness:

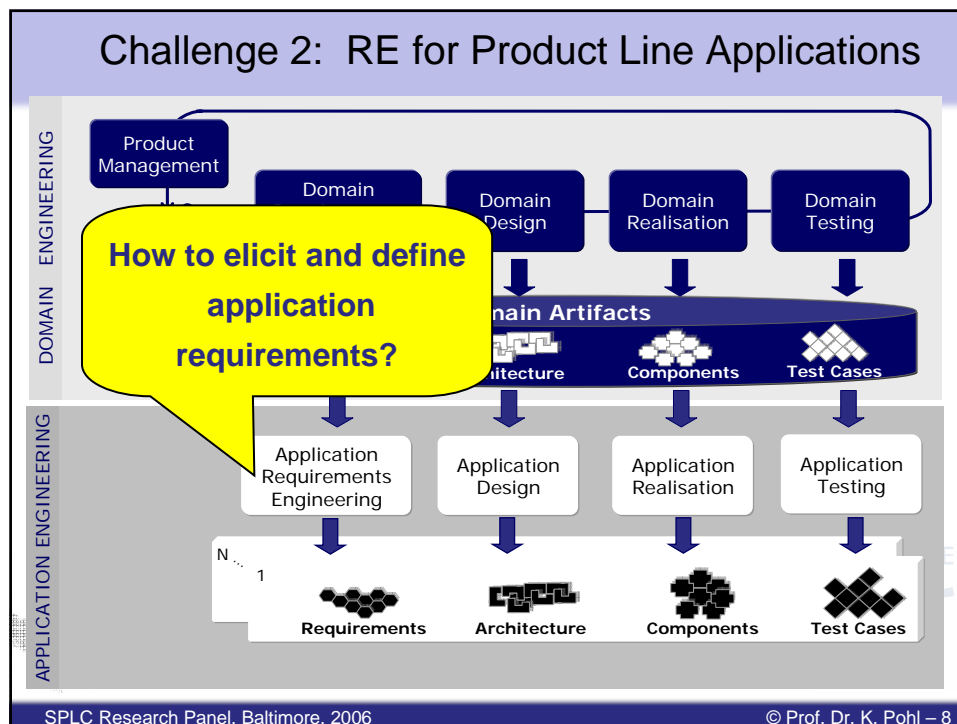
- Software-intensive Systems are tighter integrated with their context
- How to consider contextual aspects in design and deployment?
- How to adapt to context changes?

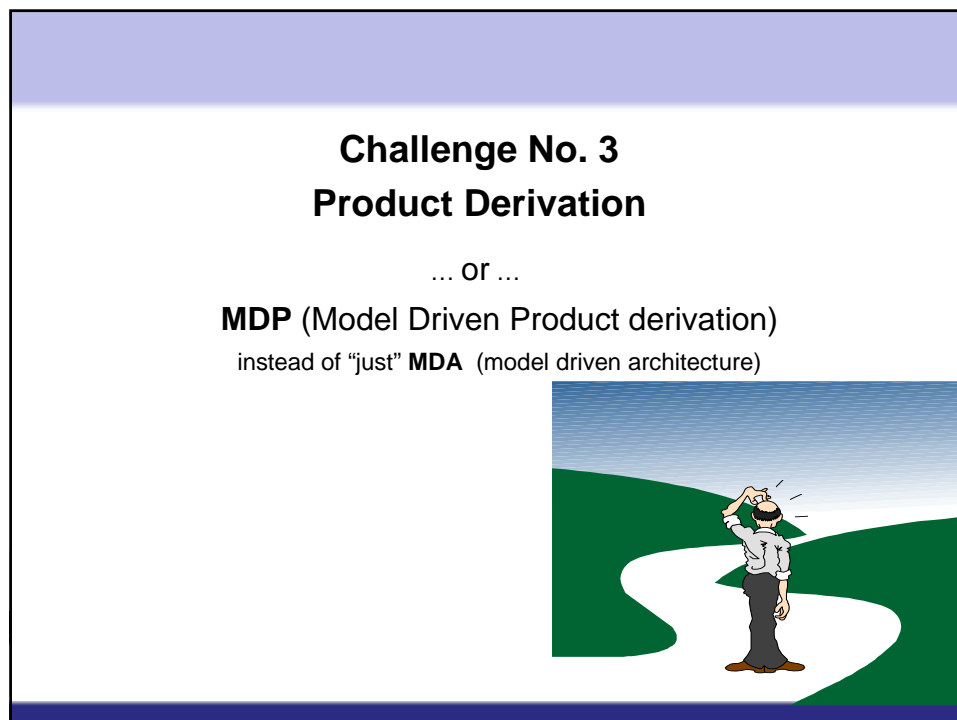
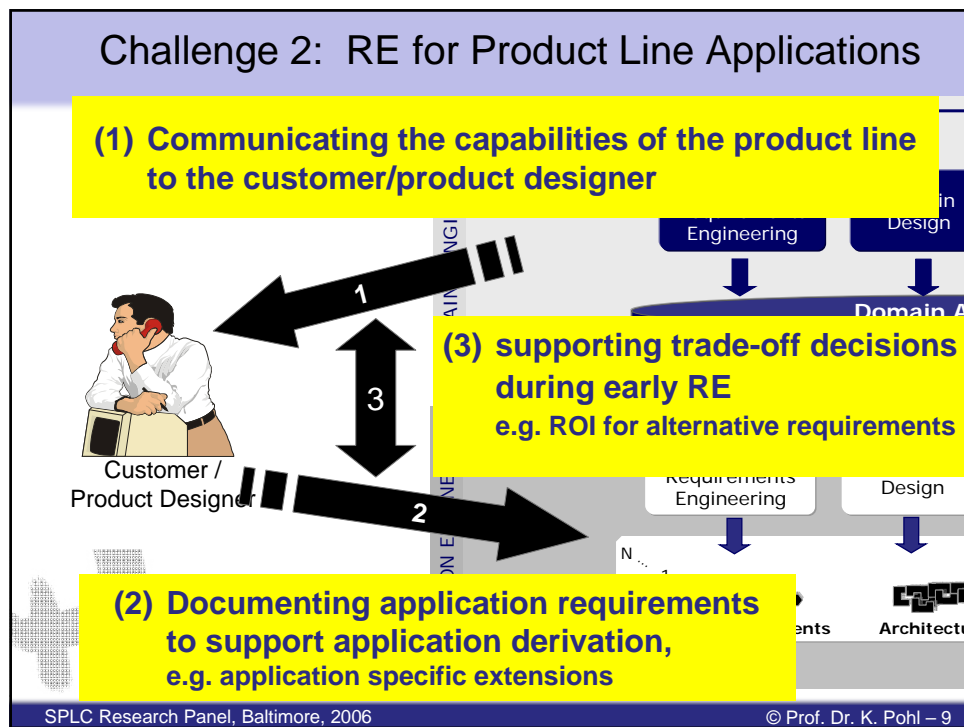
Service orientation:

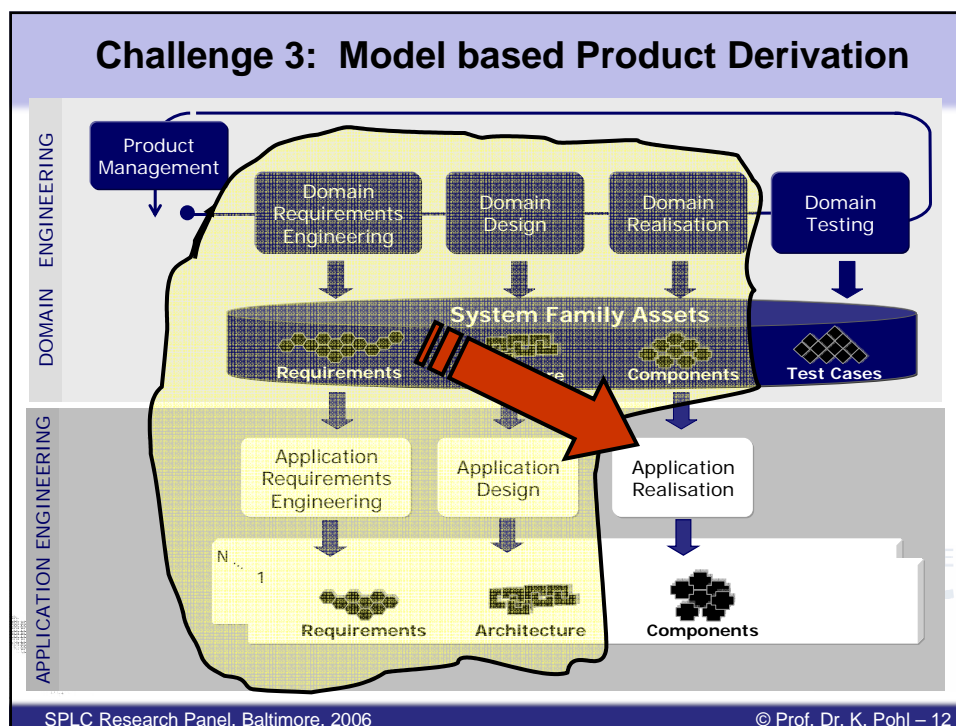
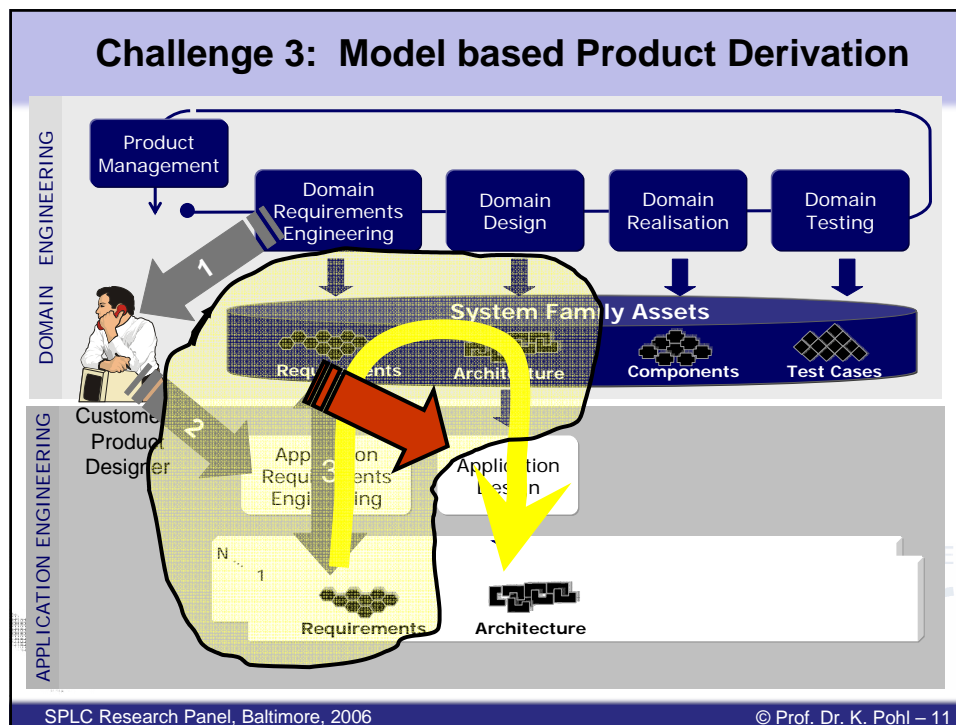
- Systems will be used differently than designed – how to address this?
- How to check service quality at run time?
- Certifications – how can systems recognize certified components?

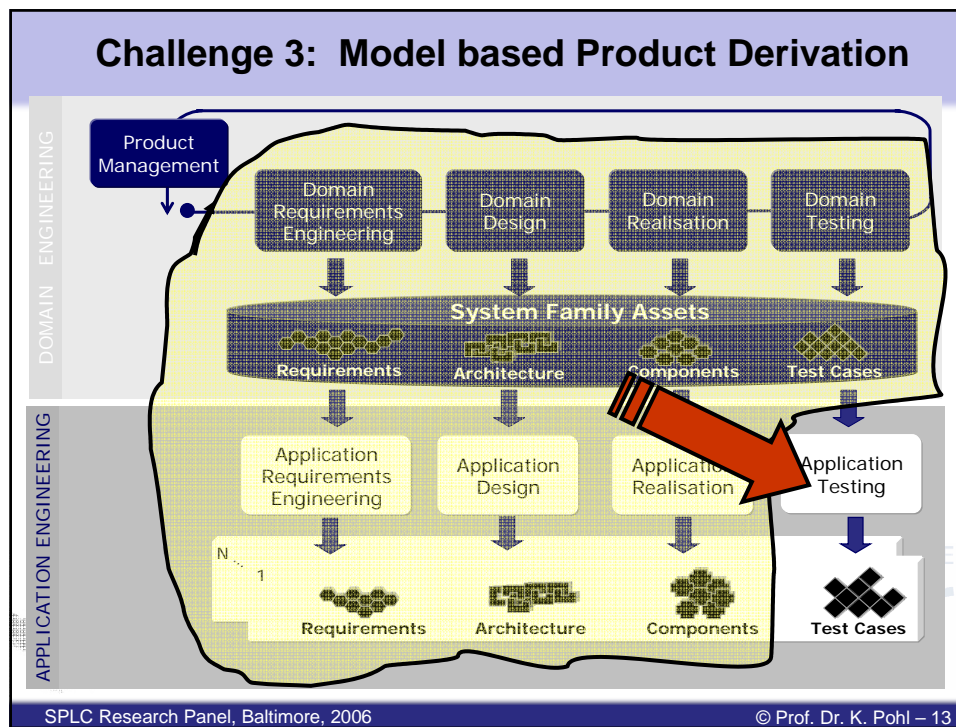
Challenge No. 2

Requirements Engineering for Software Product Line Applications







Challenge No. 4 Empirical Evidence



Challenge 4: Establish Empirical Evidence

- Establish evidence for SPLE
 - We need more **case studies**
 - We need more **experiments**
 - We need more **experience reports**
 - We need **detail reports of SPL practice**
 - ...
- Not just about **technical aspects**, but also about
 - **organisational aspects**
 - **financial aspects**

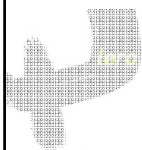


SPLC Research Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 15



Questions ?



SPLC Research Panel, Baltimore, 2006

© Prof. Dr. K. Pohl – 16



Software Product Line Research: Lessons Learned from the Last Ten Years and Directions for the Next Ten

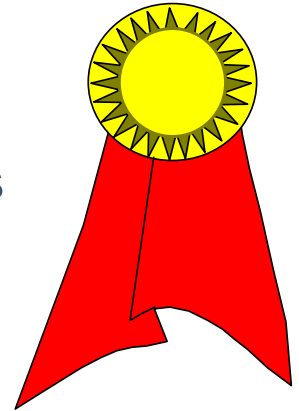
Paul Clements
Software Engineering Institute
Carnegie Mellon University
SPLC 2006

Sponsored by the U.S. Department of Defense
© 2006 by Carnegie Mellon University



Lessons Learned from the Past Ten Years

1. Software product lines work.
 - They have demonstrated the ability to bring *order-of-magnitude* improvements in schedule, budget, and quality.
 - We have not seen productivity improvement numbers like this since the advent of high-level languages.
 - They belie the truism that among “faster, better, cheaper” you can have any two.



(Remember all the “existence proof” papers we used to see, trying only to convince us that product lines are a Good Idea?)

Lessons Learned from the Past Ten Years

2. Software product line development is still software development – but with a twist.
 - Example: Configuration management is (more) important (and a bit more complex)
 - Architecture: more critical than ever; provides variability
 - Scoping emerges as an important activity
 - Project management: emphasis on coordination
 - Feature modeling: important role
 - Strategic business goals: more explicit role

Lessons Learned from the Past Ten Years

3. Organizational and business issues are more important than we might have anticipated.
 - Product line adoption strategies have emerged as important areas of work
 - Organizational structure, funding models, institutionalization, ...
4. Technical/technology issues are less important than we might have anticipated
 - Feature modeling, architecture, domain analysis, generative programming, domain-specific languages are technical areas making important contributions.

Lessons Learned from the Past Ten Years

5. We've grown a set of meaningful terminology and concepts
 - “Core assets,” “production capability,” “scope”
 - Krueger’s “proactive” and “reactive”
 - van Ommering’s “product populations”
 - Weiss’s commonality and variability analysis
 - Positive patterns: SEI’s “Adoption Factory,” etc.
 - Anti-patterns: “clone-and-own”
6. We've largely avoided fragmentation and unproductive methodological competition
 - Many fields undergo a painful “conceptual unification” stage. I'm not sure we need to.



SEI contributions and research areas - 1

Codifying the practical steps necessary for an organization to succeed with software product lines

- Framework for Software Product Line Practice
- Product line practice patterns
- Product line practices for acquisition organizations
- Product Line Technical ProbeSM
- Adoption roadmap

Evangelism and education

- Books, papers, tech reports, web site
- Product line curriculum and certificate programs
- SPLC



SEI contributions and research areas - 2

Technologies and technical approaches

- Foundational domain analysis work
- Product Line Analysis
- Structure Intuitive Model for Product Line Economics (SIMPLE)
- Approaches for creating
 - Business case
 - PL operational concept
 - Production plans
 - PL measurement programs
- Methods for architecture...
 - creation
 - evaluation
 - documentation
 - recovery





Where the field might go -1

1. Tooling: Folding in product line capabilities with “conventional” tools and IDEs
 - Making variations “first class citizens”
 - Shift the paradigm to supporting the development of a *family* of systems with specific commonalities and variations.
 - Developing a single system needs to become a (quaint) special case.
 - Signs of hope: Microsoft’s “Software Factories” approach and the *many* technology providers demonstrating at and supporting this conference
2. Product line sustainment and evolution practices. Many organizations understand how to launch a product line, but not how to keep the products unified over time – or when to let them split apart.

Where the field might go -2



3. Product line version of “round trip engineering,” in which we have strong traceability among
 - Business goals for a product line
 - Variations and commonality in a product family
 - Scope definitions
 - Variations supported/provided in
 - Requirements
 - Architecture
 - Code
 - Documentation
 - Test artifacts
- 3a. Language constructs to help express variability conditions, and compiler-like tools to generate code to automatically install the “right” variability mechanisms.

Where the field might go -3



4. “Product-line-aware” testing models that help to minimize testing across a family of products.

Questions—Now or Later

Linda Northrop

Director

Product Line Systems Program

Telephone: 412-268-7638

Email: lnn@sei.cmu.edu

Paul Clements

Email: clements@sei.cmu.edu

U.S. Mail:

Software Engineering Institute

Carnegie Mellon University

4500 Fifth Avenue

Pittsburgh, PA 15213-3890

World Wide Web:

**[http://www.sei.cmu.edu/
architecture](http://www.sei.cmu.edu/architecture)**

SEI Fax: 412-268-5758



Product Line Adoption: A Vice President's View & Lessons learned

Aug 24, 2006, *Rev A*

Salah Jarrad

sjarrad@yahoo.com

Salah.Jarrad@freescale.com

Former VP of Engineering at Panasonic Mobile
Currently Head of Multimedia Platforms at Freescale Semiconductor Inc.

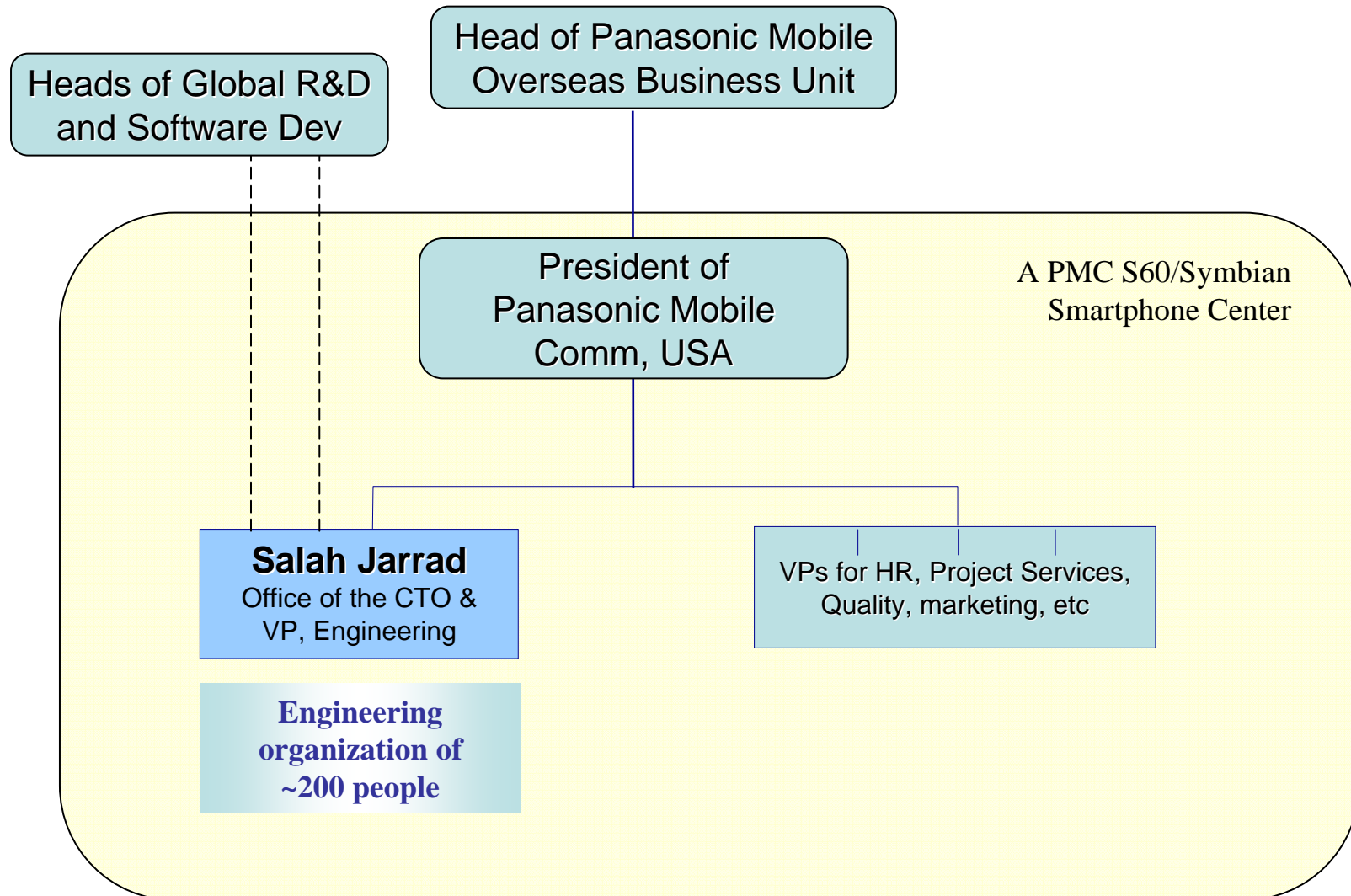
Use or reproduction is prohibited without the prior express written permission of Salah Jarrad.
Copyright (c) 2006 Salah Jarrad

Content

- ❑ Introduction
 - ❑ Why Product Line
 - ❑ Timeline of Mobile Phones Product Line experience
 - ❑ Product highlights & achievements
- ❑ Our Approach to PL (3 phases)
- ❑ Product Line Experience & Recommendations
 - ❑ Our findings in comparison to SEI
 - ❑ Lessons learned
- ❑ Summary



My Role At Panasonic



Why Product Line

- ❑ Our “new design” product development cycle was 18 to 24 months, followed by 6 to 12 months for follow-on products, all with limited variability (limited customization & configuration)
- ❑ Each customer “mobile phones service provider” wanted their own flavor of product, customized according to their market vision (their own apps and services)

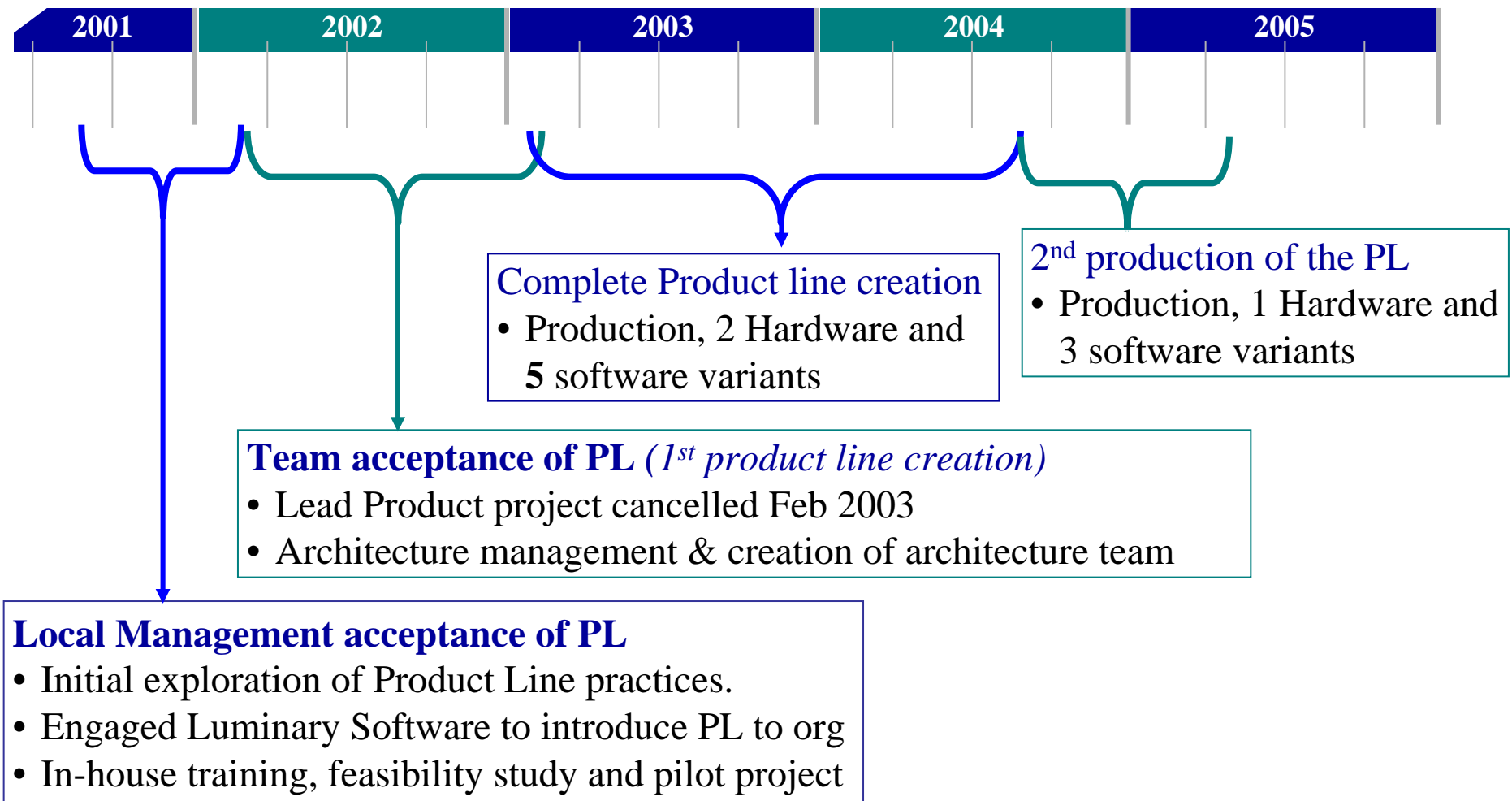
Creating product variants for each customer took too long and was limited to fewer customers



Product Line Concept was
the obvious answer

Timeline Of Product Line Experience

- ❑ Introduced Product Line to organization ~Sep 2001
- ❑ Product Line life cycle from mid 2002 to 2005



Panasonic X700, X701 & X800 HW & SW Variants

Hardware Variants

Software Variants

X700



X701



X800



Product Line Highlights & Achievements

- ❑ Produced 8 mobile phone products in one program (8 major software tracks / builds, each resulting in a unique product)
- ❑ Created process and tools to enable customization at production time, both at the design center and manufacturing facility (*A 1st within Nokia series 60 Platform community*)
- ❑ Highest level of product customization, product line sold in more than 30 countries
- ❑ Tailored code base management - daily and weekly build, incremental integration, stringent guidelines on what makes a successful build (partial vs. complete), etc.

Our Approach



Phase I Education & Initiation

- ❑ Champion of change
- ❑ Support of local management
- ❑ Educate self and group



Phase II Acceptance & PL Development

- ❑ Initiate formal product line projects
- ❑ Implement needed changes
- ❑ Complete Design cycle



Phase III Production

- ❑ Produce products from the line
- ❑ Involve all other business functions involved in the PL

Our Recipe For Rolling Out PL

Phase I: First 7 to 9 months

- 1) VP of Software & systems and staff self educate on Software product line
- 2) Volunteered a few engineers from the technical ranks to enroll in the education (trail blazers)
- 3) Engaged Luminary Software to run a small pilot project and train the organization
- 4) Started to educate local site management team (President and staff)

Our Recipe For Rolling Out PL

Phase II: First PL design

- 5) Initiated first formal product line project (lead mobile phone product for anchor customer, and 3 derivatives)
- 6) Reorganized all engineering into two organizations, Platform (asset creation) org and Products org
- 7) Started to educate corporate (HQ) executives on PL

Phase III: Production

- 8. Testing the variations and customer acceptance on each customized product
- 9. Introduced factory and sales force to product line
- 10. Mass production of each product in the line (simultaneous launch across the world)

Content

- ❑ Introduction
 - ❑ Why Product Line
 - ❑ Timeline of Mobile Phones
Product Line experience
 - ❑ Product highlights & achievements
- ❑ Our Approach to PL (3 phases)
- ❑ **Product Line Experience & Recommendations**
 - ❑ Our findings in comparison to SEI
 - ❑ Lessons learned
- ❑ Summary

Organizational Benefits (SEI Claim)

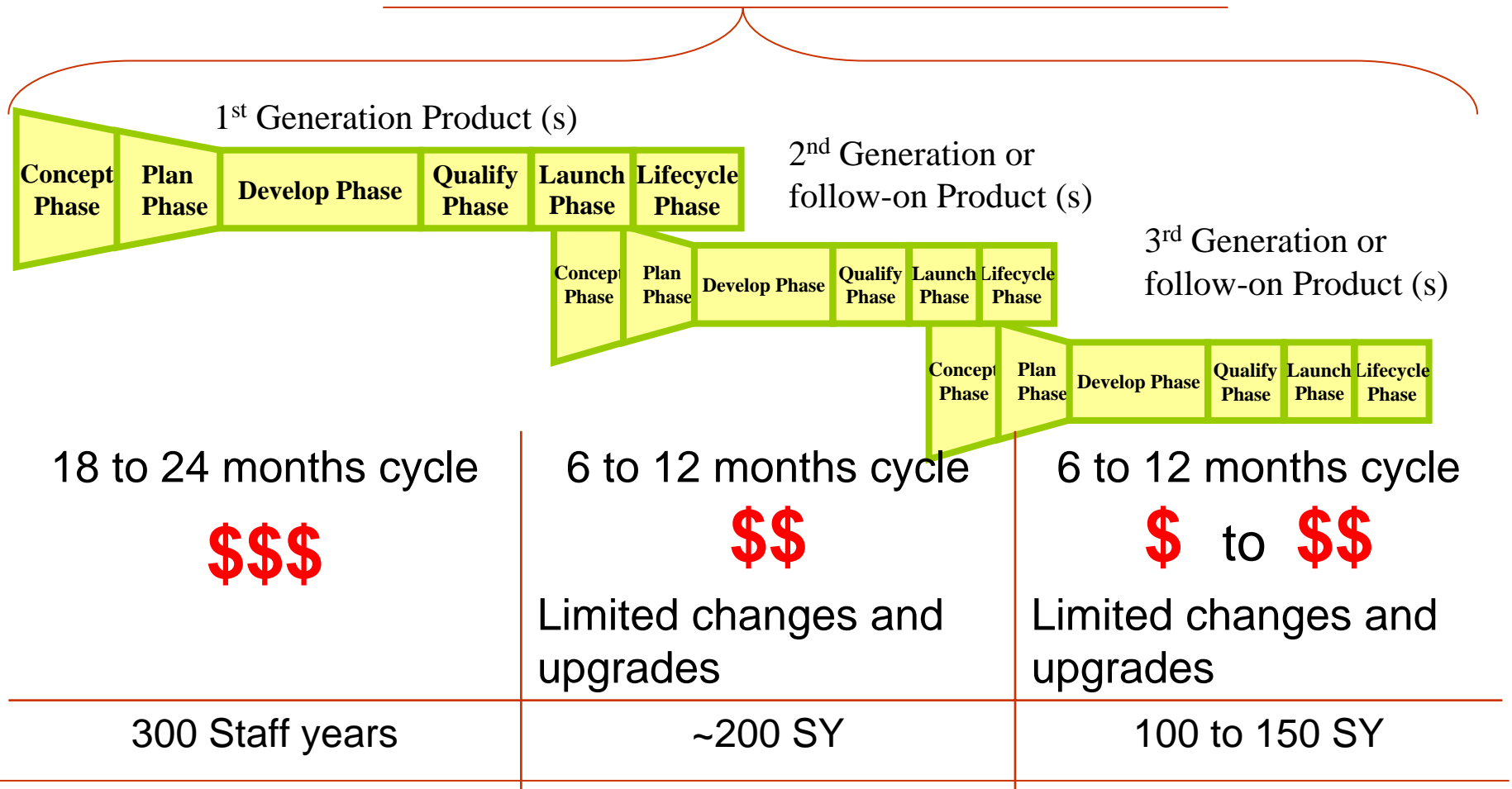
From SEI web site: Summary of PL Organizational Benefits

- 1) Improved productivity
 - By as much as 10x
- 2) Decreased time to market (to field, to launch)
 - By as much as 10x
- 3) Decreased cost
 - By as much as 60%
- 4) Decreased labor needs
 - By as much as 10x fewer software developers
- 5) Increased quality
 - By as much as 10x fewer defects

Pre-PL Product Cycle

- ❑ Produce lead product in new design first, followed by 6 to 12 months for follow-on products.
- ❑ Development cycle for Lead product varies depending on introduction of new technology (*available new technology vs. technology development in conjunction with product development*)

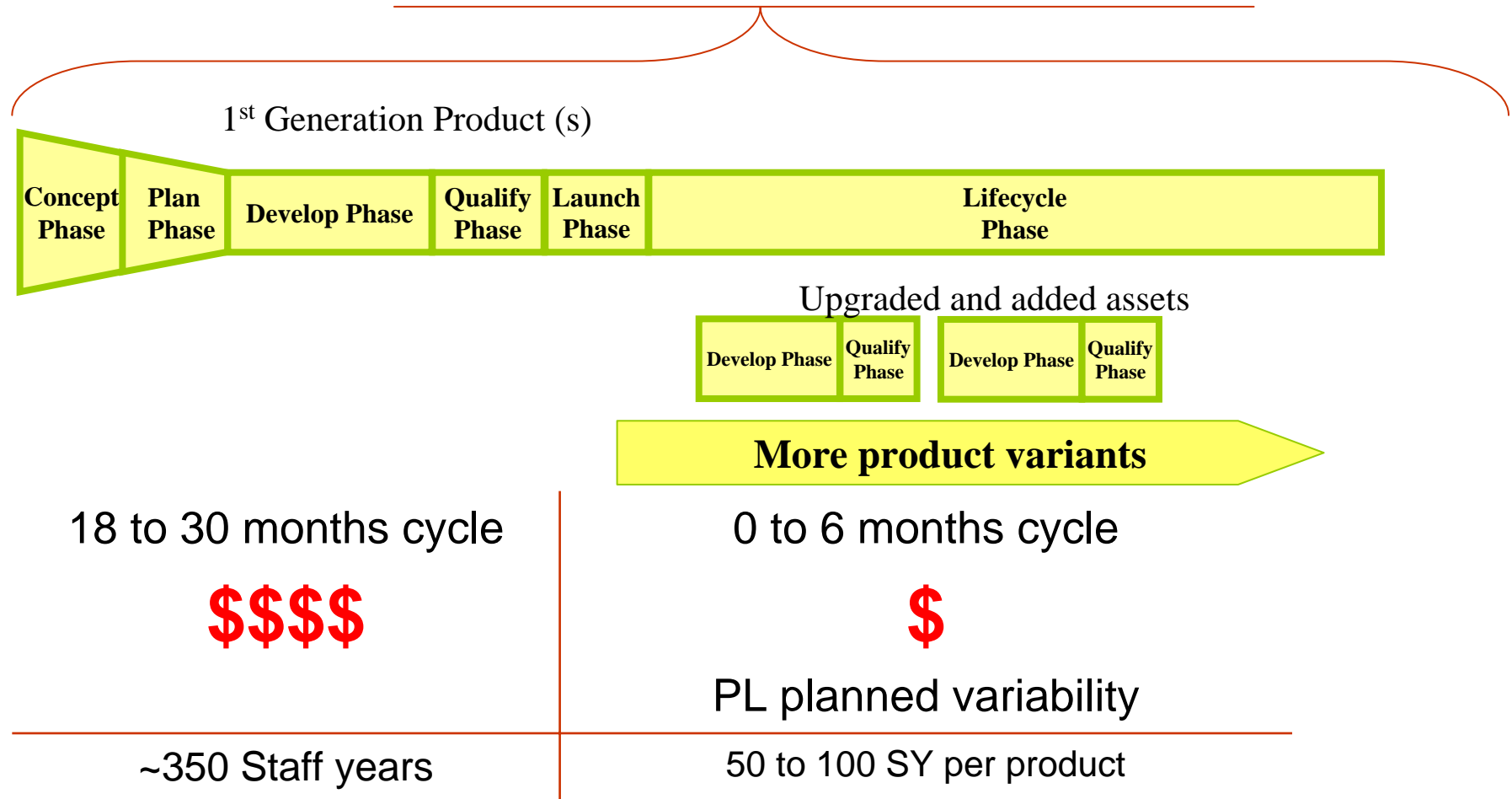
Design life span for Handset Product family before need for total re-design or total technology change (3 to 5 years)



Our PL Product Cycle

- Slightly longer cycle to develop assets and reach 1st production of products, but many more products can be produced

Design life span for Handset Product family before need for total re-design or total technology change (3 to 5 years)



Benefits

- ❑ Difficult for companies to recognize the benefits of PL (before & after comparisons) without measuring customer coverage and satisfaction

Market coverage (Addressing more customers needs)

Before

**1 main product, 2 derivatives
with limited customization / variation**

After

**Multiple products,
1 or more for each customer
Greater level of customization**

1st product: 300 Staff years
2nd Product: ~200 staff years
Other follow-on: ~100 SY

600 SY or more for 3 Products with limited
customization / Variation

350 SY for 5 products
100 SY for follow-on 3 products

650 SY for 8 product in the PL, with
greater level of customization

- ❑ Response of Business manager “About time! We expected this and more from engineering all along, finally they are beginning to deliver”

Summary: Organizational Benefits

- 1) Improved productivity
 - By as much as 10x
- 2) Decreased time to market
 - By as much as 10x
- 3) Decreased cost
 - By as much as 60%
- 4) Decreased labor needs
 - By as much as 10x fewer software developers
- 5) Increased quality
 - By as much as 10x fewer defects

Was not as much the case for us, however:

- ⇒ Asset creation takes much more, but gains realized when producing multiple products
- ⇒ For follow-on products, also allows for more variants (customization for each customer)
- ⇒ For total PL when used for full product (s) life (3 or more years)
- ⇒ Needed to maintain the same level of staffing, but output is increased
- ⇒ Didn't experience quality difference, but increased ability to offer more products

Lesson Learned



- ① Management as well as rank and file resistance



- ② Impact of multiple changes at the same time (org, process, design and assembly methods)
- ③ Architecture lessons



- ④ Addressing late introduction of PL to factory and sales

1

Resistance

- ❑ Encountered resistance in all levels across the company

Our Recipe for educating/
rollout. Sequential ramp-up

- 1) VP of Software & Staff (*later VP of Eng*)
- 2) Technical people
- 3) Local site management
- 4) Corporate HQ management
- 5) Other business functions

Recommendation from our experience

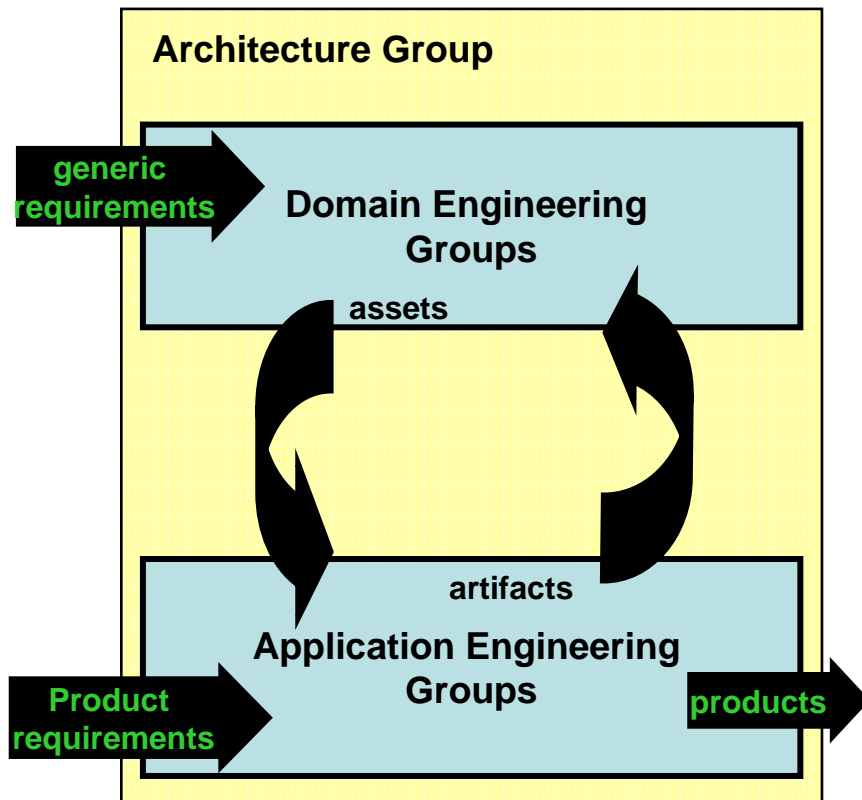
- ⇒ Identify management and technical staff champions
- ⇒ Roll-out in parallel to management and technical staff
- ⇒ Be aggressive in obtaining corporate HQ management support
- ⇒ Start with all business functions as soon as HQ managers support the effort
- ⇒ **Limit changes but deploy broidery**

2

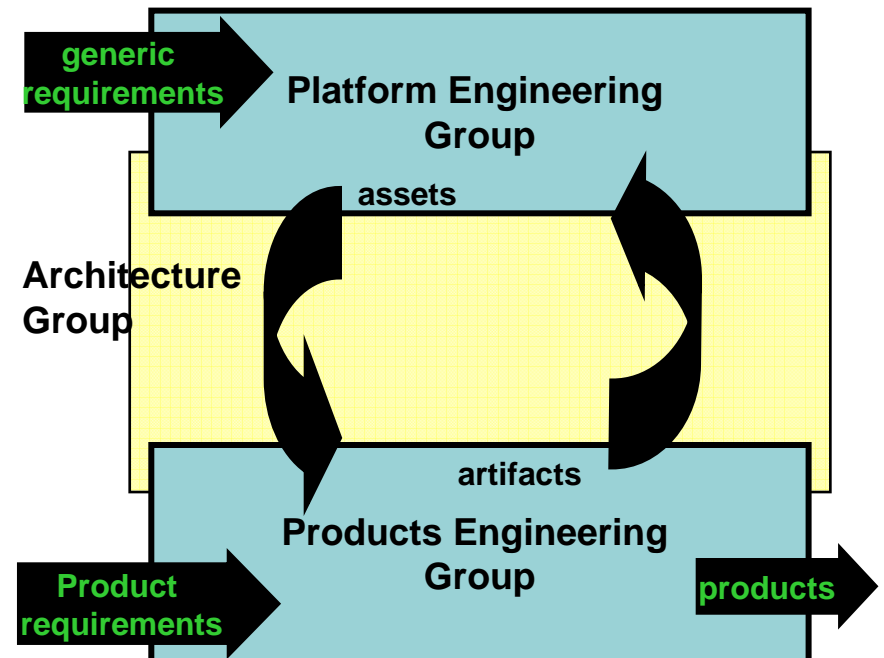
Group Structure & Interactions

- ❑ Introduced PL concept, operational changes and total organization changes all at once → *confused organization for well over a year*

SEI publication



Our Initial Implementation



2

Recommendations

- ☞ Decide how you will create assets, within product development or before. Test what organization and business culture will support, seek input thru surveys and questionnaires
- ☞ Roll out changes slowly, assess impact of each change
- ☞ Focus on defining roles & position, prepare and train people on new roles rather than forcing organization changes
- ☞ Use as much of org established terminology as possible

3

Architecture lessons

- ❑ Most consumer electronics product designs don't start from scratch, most depend on many acquired sub-systems and components

☞ Focus on creating architecture framework as a communication and decision making tool

- ❑ Architecture and PL concepts can be introduced independent of one another. PL helps you manage your design and business, architecture supports PL needs

☞ Separate introducing architecture design (*in not already established*) and changes from introducing PL concepts to avoid organization resisting to both as one and the same

4

Enrolling Other business functions

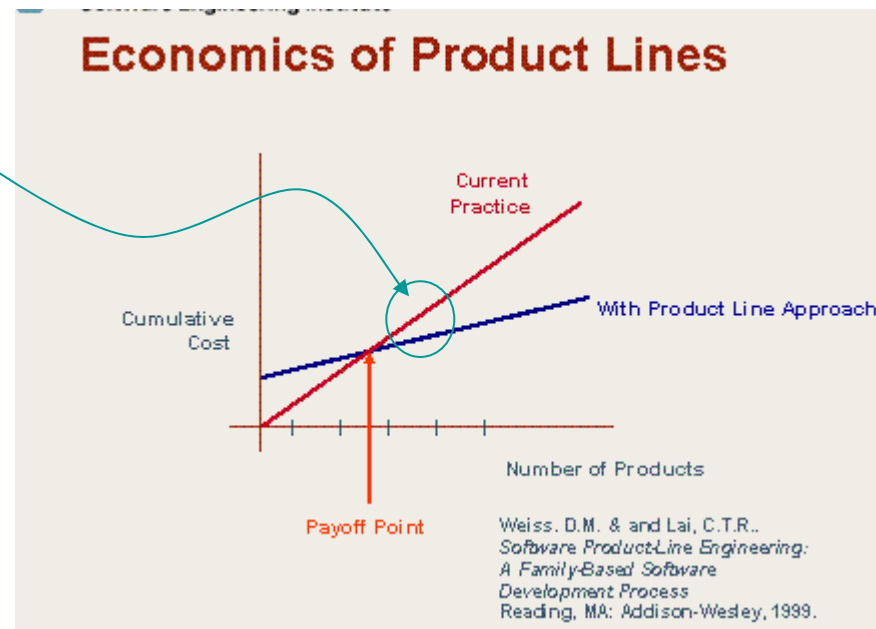
- ❑ We introduced the PL concept (multiple products from 1 design, higher level of customization) to factory and sales very late in the design cycle, we encountered tremendous resistance, confusion and abdication of responsibility
- ❑ We assumed more responsibilities and had to develop product assembly and customization tools for factory use

👉 Don't wait, involve all other business functions in PL very early on, get their buy in and prepare them for the positive change

Other Lessons

- ❑ PL pays off when sustained beyond one product design cycle, and when more products are produced from the PL

We never reached far beyond the payoff point

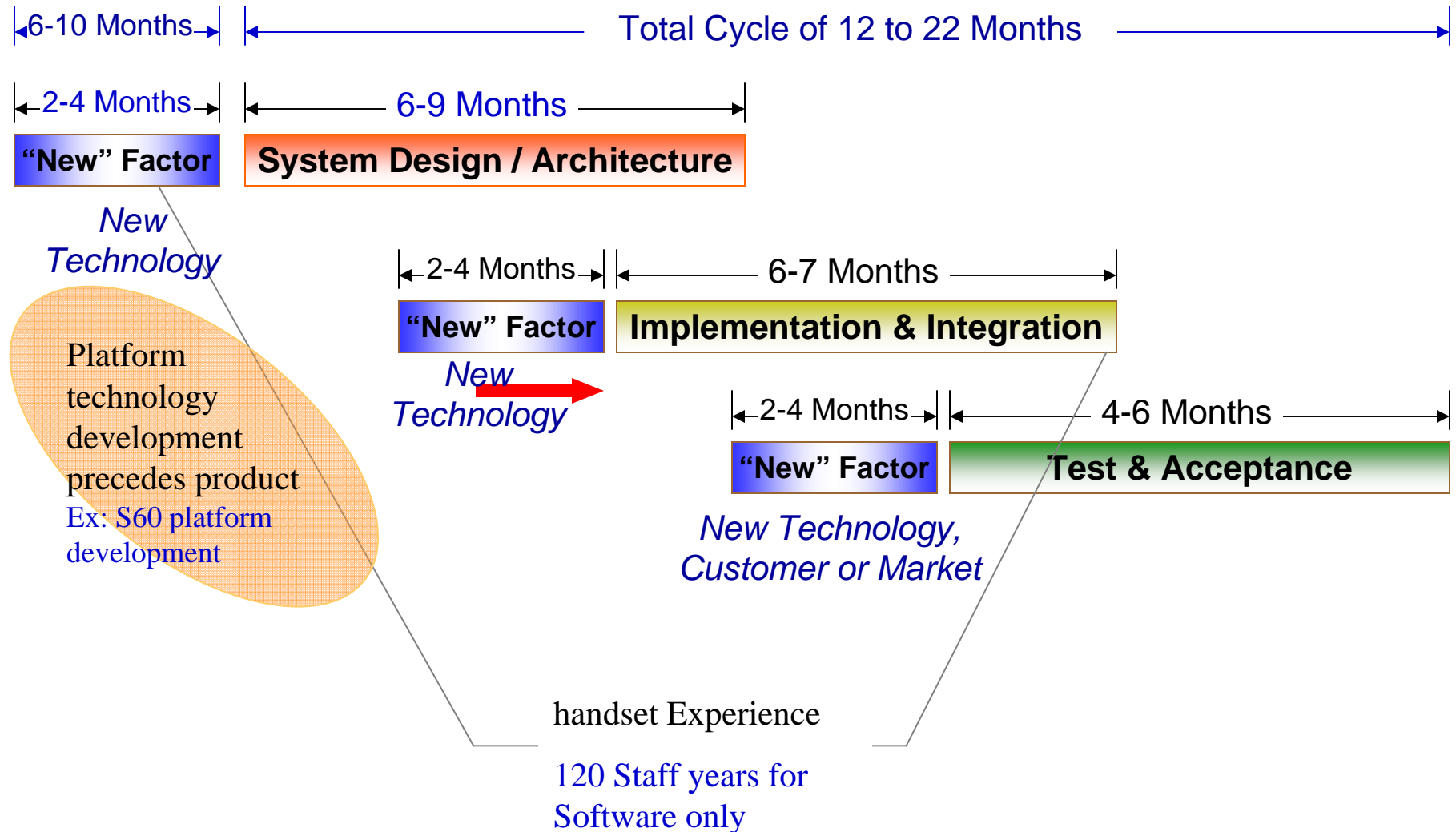


Summary

- ❑ Very positive results from doing Product Line
 - Gained efficiency (2 to 3x)
 - Significant flexibility and customization for each customer
- ❑ Change is always difficult, spend more time planning for change, do less but deploy broadly
 - i.e involve all business functions early on
- ❑ Give Product Line Practice more time to see maximum benefits

- ❑ Backup Slides or slides that will be eliminated

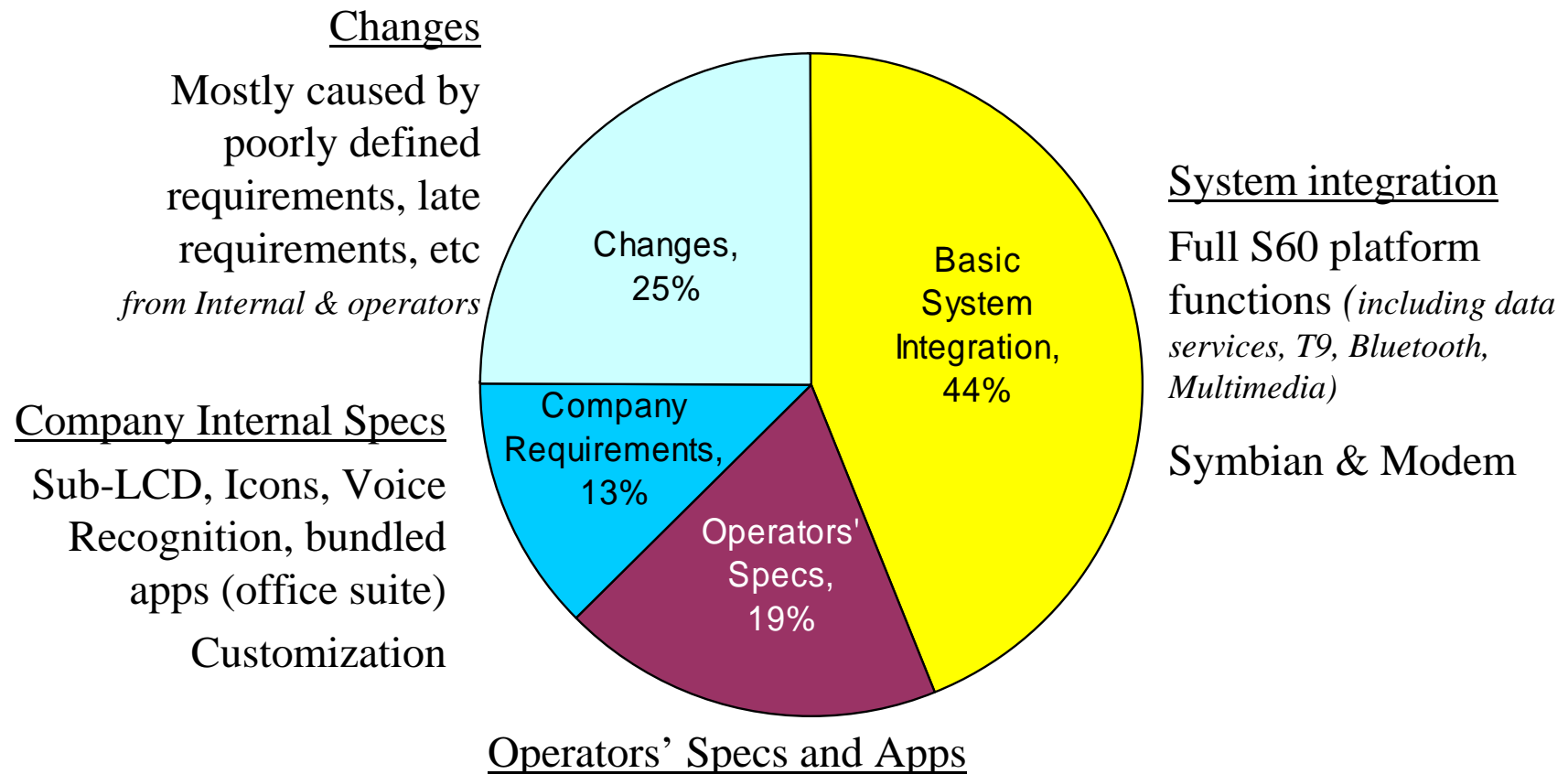
Simple View of Product Development Cycle



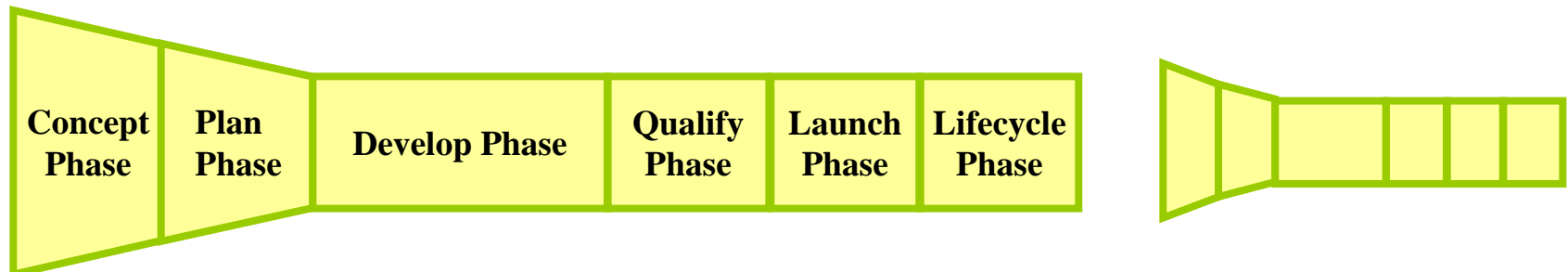
Handset Software Development Effort

- ❑ Development effort & cost to integrate platform (s) still relatively high due to increase product complexity

Breakdown of Development Effort



First, Some Clarification



Traditional	Shorter cycle, Focus on requirements and plans for single of first product	Shorter for single product	Short for single or first product	Maintain product	Follow on product could take much longer for unplanned requirements
Product Line	Capture requirements for the line, plan assets and products	Long cycle to develop assets first then products	Longer for 1 st product, much shorter for all other products from the line	Ability to spin more variants, maintain assets	Much easier to update and product variants within the life time of design



Consolidate.

Simplify.

Leverage.

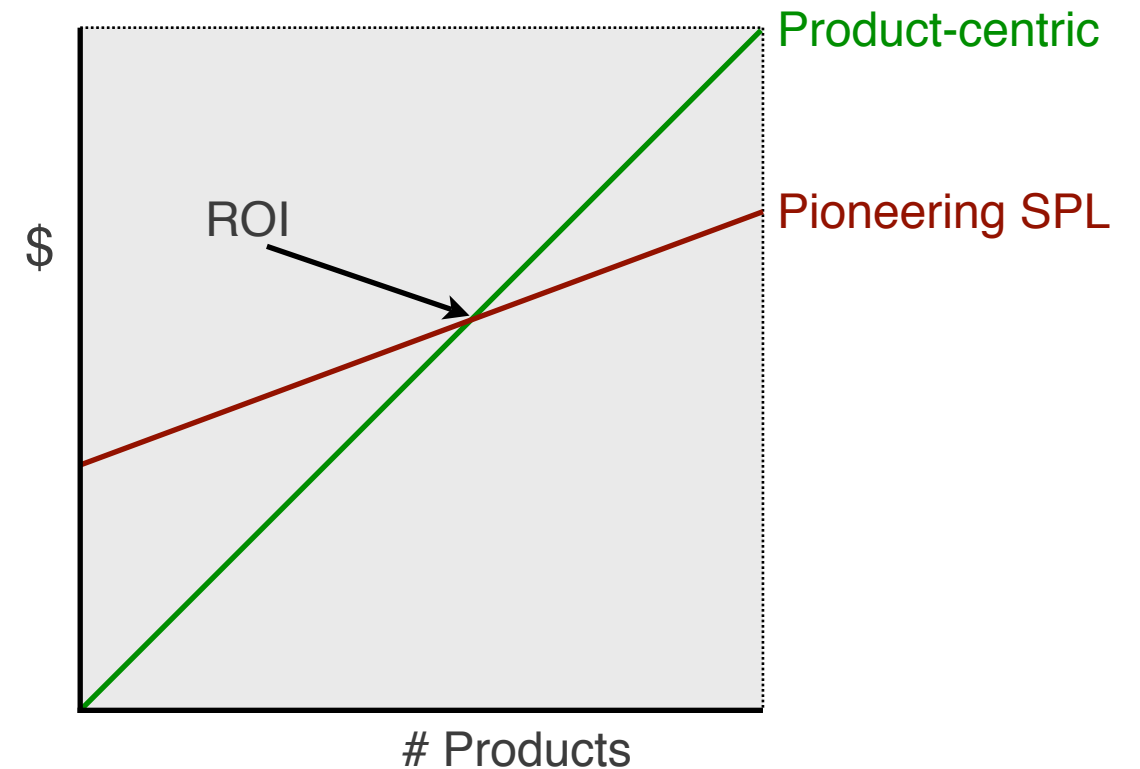
New Methods in Software Product Line Development

**Charles W. Krueger
BigLever Software**

**SPLC 2006
August 23, 2006
Baltimore, Maryland, USA**

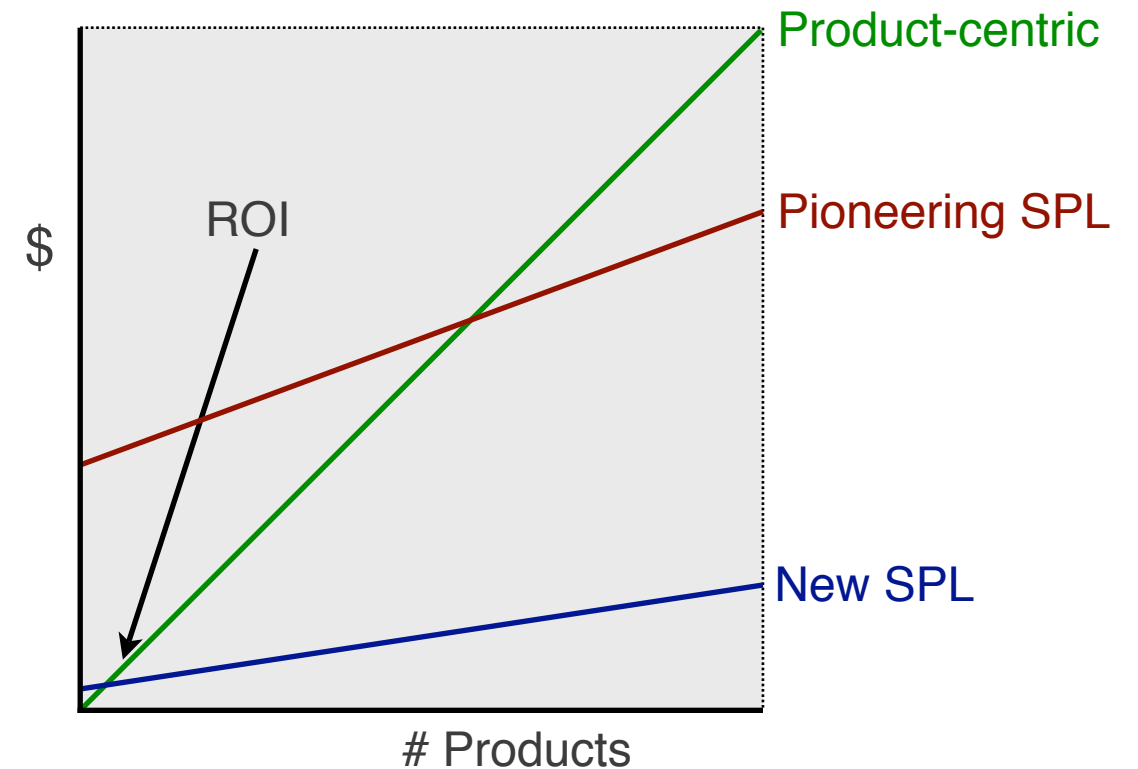
Pioneering versus New Generation SPL

- Pioneering case studies are 10-20 years old
- What's new?
- *New Generation* reflects best practices learned
 - methods
 - tools
 - techniques
- Order-of-magnitude improvements enable mainstream adoption



Pioneering versus New Generation SPL

- Pioneering case studies are 10-20 years old
- What's new?
- *New Generation* reflects best practices learned
 - methods
 - tools
 - techniques
- Order-of-magnitude improvements enable mainstream adoption



Three New Generation Methods

- Software Mass Customization
 - *Application Engineering Considered Harmful*
- Minimally Invasive Transitions
 - *Work Like a Surgeon, Not Like a Coroner*
- Bounded Combinatorics
 - *As a practical limit, the number of possible products in your product line should be less than the number of atoms in the universe*

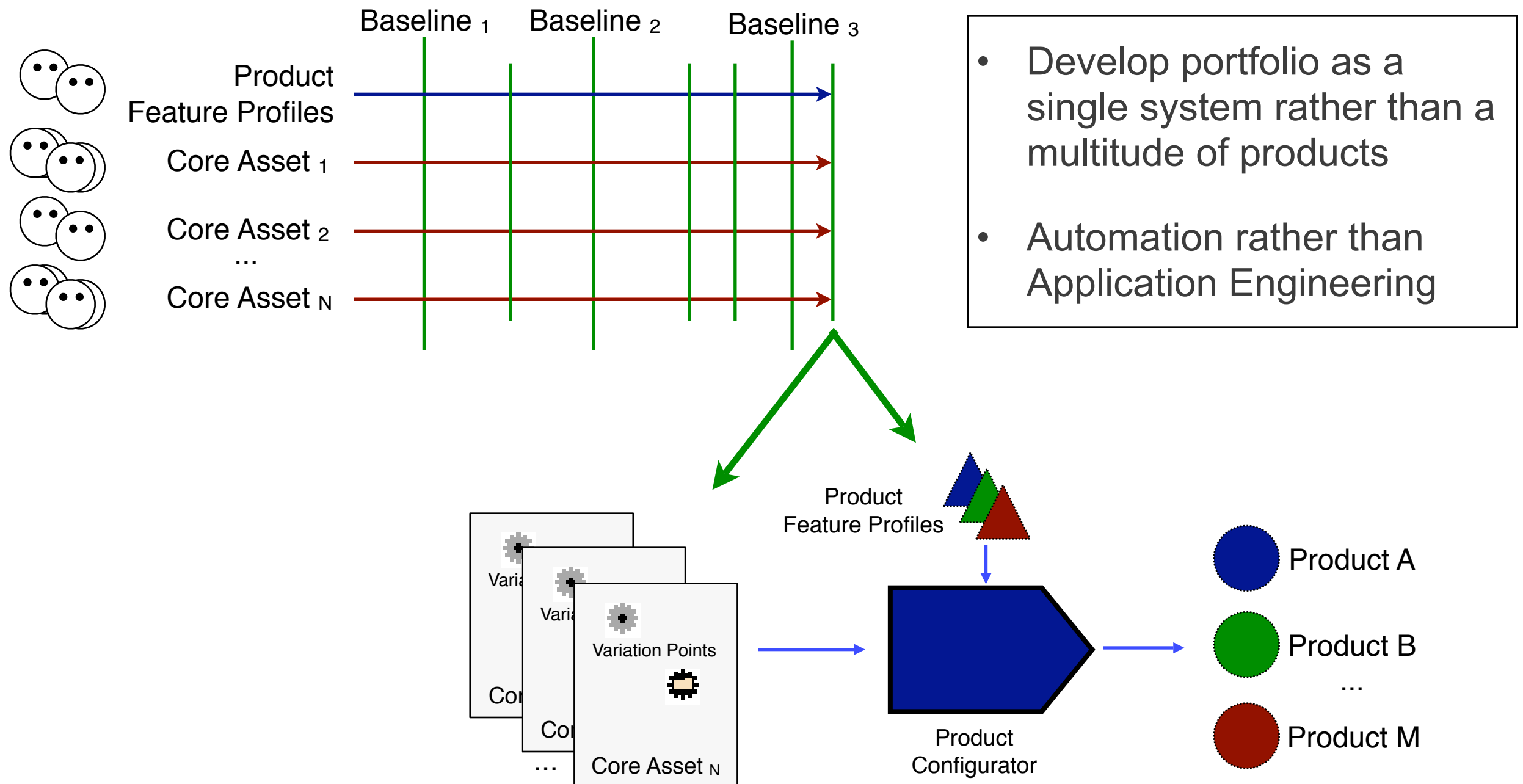
Software Mass Customization

Application Engineering Considered Harmful

Why Software Mass Customization?

- Manual application engineering is harmful to software product line economics
 - Leads to labor intensive duplication, divergence, merging and coordination, particularly in evolution and maintenance
 - Has many of the product-centric characteristics of clone-and-own
 - Glue code is one-off development, which inhibits domain-level reuse, refactoring and evolution
 - Dichotomy of domain engineers and application engineers creates an “us versus them” cultural divide in the organization

Software Mass Customization



Minimally Invasive Transitions

Work Like a Surgeon, Not Like a Coroner

Why Minimally Invasive Transitions?

- Most organizations cannot tolerate significant disruption to ongoing production schedules
 - SPL Return-on-Investment arguments suggest an easy business case
 - Not true when the Investment means diverting enough expertise to disrupt ongoing production
 - Primary impediment to moving to SPL in mainstream practice

Minimize Start State to Target State

- Assumption: there is already a product line
 - Work like a surgeon. Not like a coroner.
 - Avoid the lure of the green field
- Reuse legacy assets for SPL assets
- Don't introduce the domain engineering and application engineering dichotomy in the organization
- Start with minimalist and reactive scoping

Incremental Return on Incremental Investment

- Decompose initial transition into incremental repetitions
 - One product at a time
 - One lifecycle phase at a time
 - One component or subsystem at a time
 - One team at a time
- Use improvements from one increment to pay for next
- Non-disruptive transitions with accelerating production

Bounded Combinatorics

As a practical limit, the number of possible products in your product line should be less than the number of atoms in the universe

Why Worry about Combinatorics?

- 216 boolean features == 10^{65} feature combinations
 - That's the number of atoms in the universe
- 33 boolean features == 8×10^9 feature combinations
 - One for every human on the planet
- Domain engineers gleam over models with 1000 features
- Test engineers spontaneously combust over models with 1000 features

Harnessing Combinatoric Complexity

- Time-tested computer science techniques apply well
 - Abstraction
 - Modularity
 - Controlling scope
 - Controlling entropy
- Uniquely applied for software mass customization

Software Mass Customization Abstraction Layers

Composition Profile

A composition of profiles of subsystems. A composition is itself a profile. Abstraction for hierarchical product line assemblies.

Feature Profile

An instantiation, based on a set of decisions in the feature models. Abstraction for a point solution, with specific benefits.

Feature

The feature model. Abstraction for variability in the application domain. Localized or aspect-oriented.

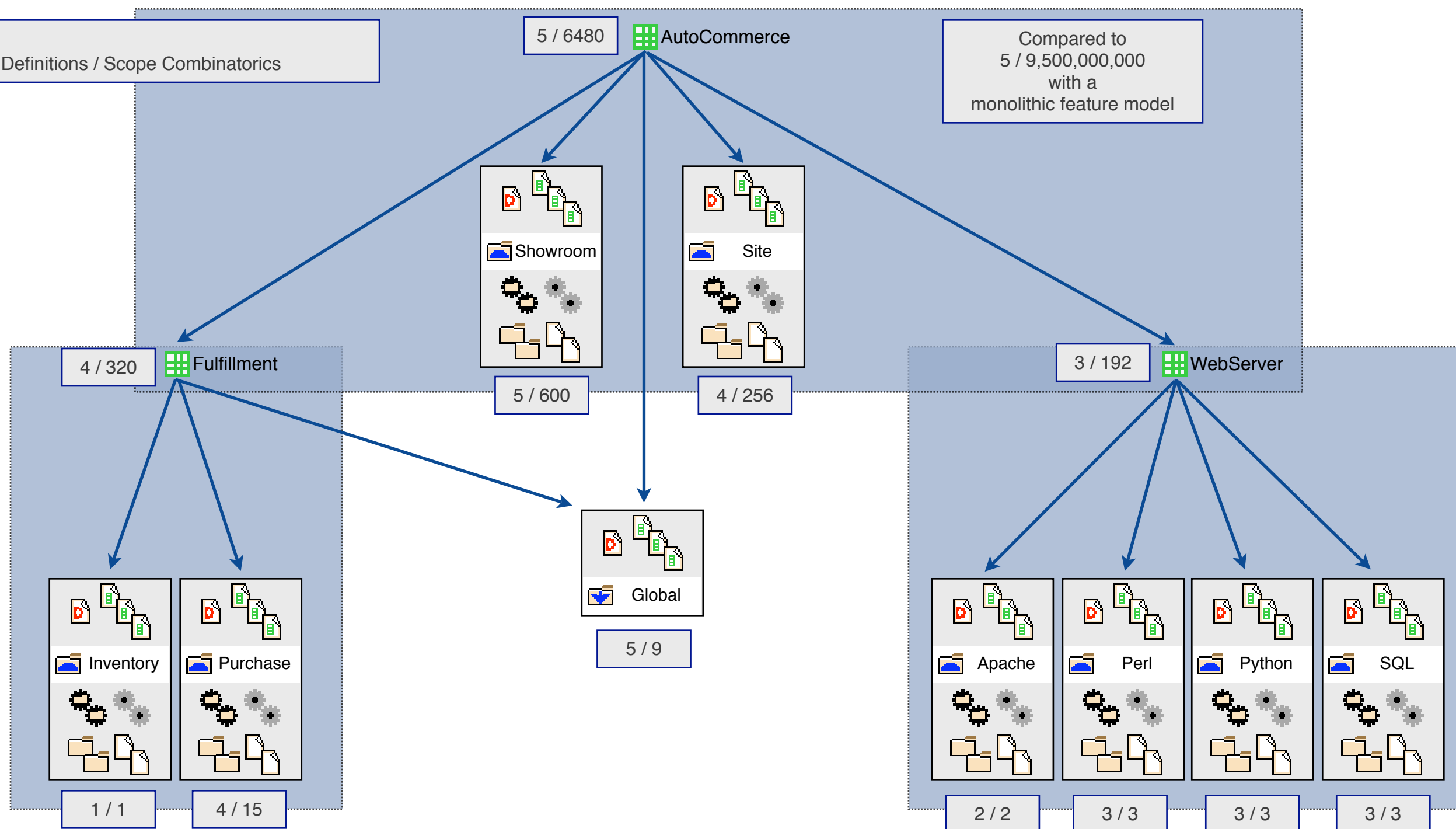
Variation Point

Source-level variation in the product line assets. Alternatives expressed in terms of feature values.

Composition and Hierarchical Product Lines

Key:
Product Definitions / Scope Combinatorics

Compared to
5 / 9,500,000,000
with a
monolithic feature model

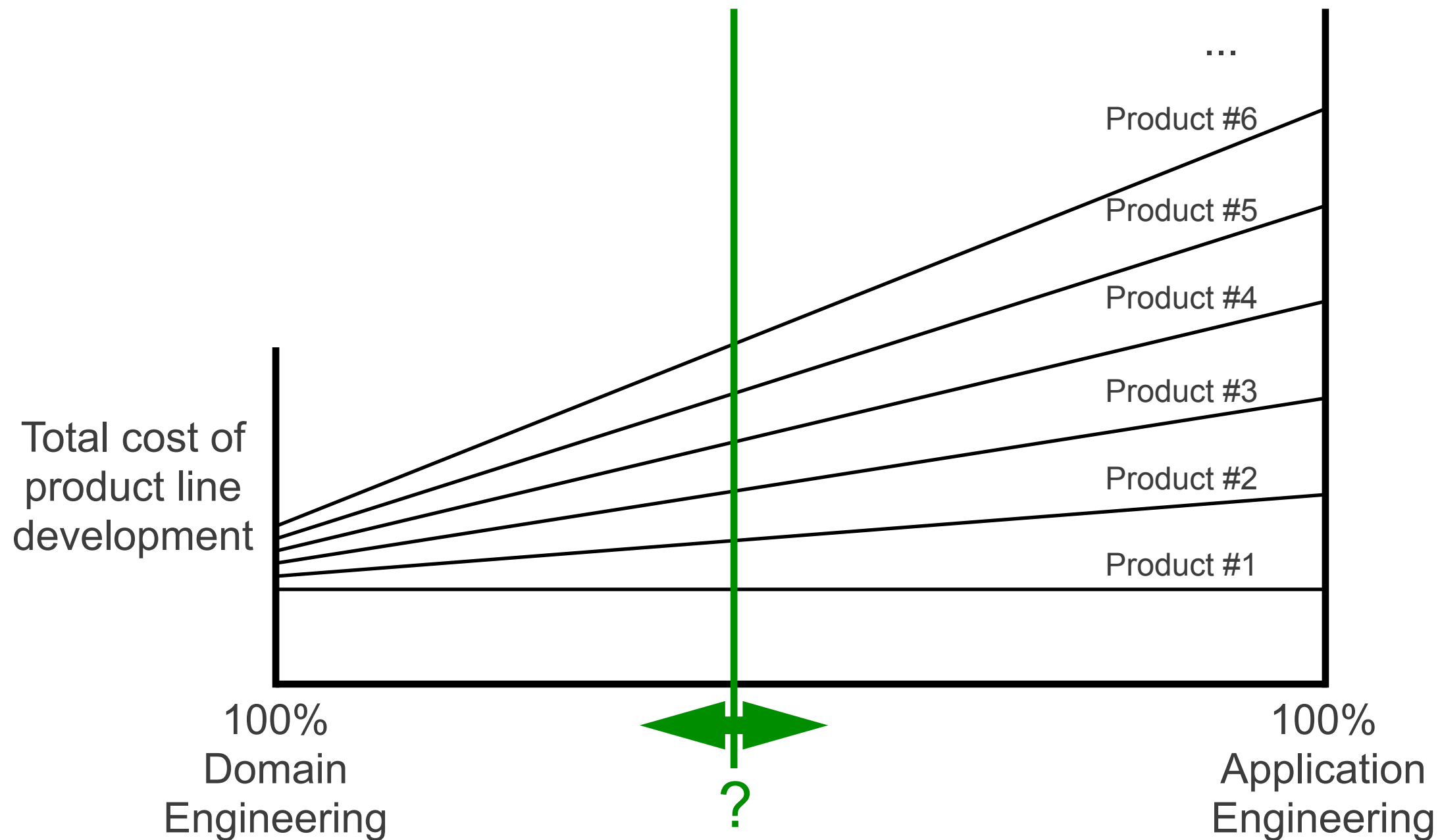


Conclusions

- SPL field has 10-20 years of experience to draw on
- A new generation of case studies is emerging based on new methods, tools and techniques
 - Software mass customization
 - Minimally invasive transitions
 - Bounded combinatorics
- Making it orders of magnitude easier to gain even greater benefits of SPL approach



Ideal Balance of Domain Engineering versus Application Engineering?





**10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA**

2006 Software Product Line Hall of Fame Inductee

RAID controller firmware product line, LSI Logic - Engenio Storage Group



The Engenio Storage Group of LSI Logic produces high performance, high availability RAID storage systems. Engenio has established a reputation of providing high-value scalable systems, consistently releasing leading edge performance and being first to market with key technology transitions. Engenio sells products in an OEM business model through strategic partnerships with other companies who deliver complete end user solutions with unique combinations of hardware, software and services for applications including transaction processing, e-mail, data warehousing and scientific research.

Engenio transitioned to a software product line approach for its embedded RAID controller firmware in order to satisfy growing customer demand for product differentiation as well as to support an expanding set of controller hardware platforms. The product line was created with about 4 developer months of effort using an extractive approach by merging multiple existing code sets into a single set of code with engineered variation points. The product line was strategically deployed to the development staff such that no product delivery schedules were impacted. New products were added to the product line using a reactive approach, restructuring and re-architecting when necessary to meet development requirements. Two years after the initial deployment, the product line was capable of producing nearly 90 different controller firmware products, supporting multiple controller hardware platforms and multiple customer customizations.

- BigLever Software Case Study: Engenio on <http://www.biglever.com/>
- Hetrick, W., Moore, J. and Krueger, C. Incremental Return on Incremental Investment: Engenio's Transition to Software Product Line Practice. OOPSLA Proceedings 2006. Portland, Oregon. October 2006.

What is the Software Product Line Hall of Fame?

A hall of fame serves as a way to recognize distinguished members of a community in a field of endeavor. Those elected to membership in a hall of fame represent the highest achievement in their field, serving as models of what can be achieved and how. Each Software Product Line Conference culminates with a session in which members of the audience nominate systems for induction into the Software Product Line Hall of Fame. These nominations feed discussions about what constitutes excellence and success in product lines. The goal is to improve software product line practice by identifying the best examples in the field. Nominations are acted on by a panel of expert judges, who decide which nominees will be inducted into the Hall of Fame.

You can read about the current members of the Software Product Line Hall of Fame at http://www.sei.cmu.edu/productlines/plp_hof.html.

Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).

Bosch Gasoline Systems: Engine Control Software Product Line

Dipl.-Ing. Christian Tischer
Dipl.-Ing. Andreas Müller
Robert Bosch GmbH

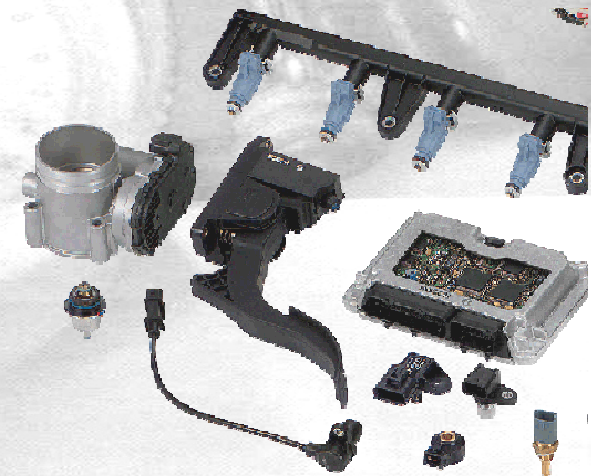
Hall of Fame Presentation Aug 24th 2006



BOSCH

Bosch Gasoline Systems (GS)

- ⌚ **GS is a system provider for gasoline engine systems**
 - 1 Software for Engine Control Units (ECU)
 - 1 Hardware for ECUs
 - 1 Sensors and Actuators
 - 1 Software Calibration
- ⌚ 4 Million ECUs per year
- ⌚ About 1000 software and calibration engineers
- ⌚ About 1500 program versions per year with
 - 1 400 Functions
 - 1 1600 Files
- ⌚ Variance of the software products is mainly driven by
 - 1 Customers: vehicles, engines, gear units, ...
 - 1 Countries: emission laws, diagnosis laws, fuel types, theft protection, ...



Gasoline Systems

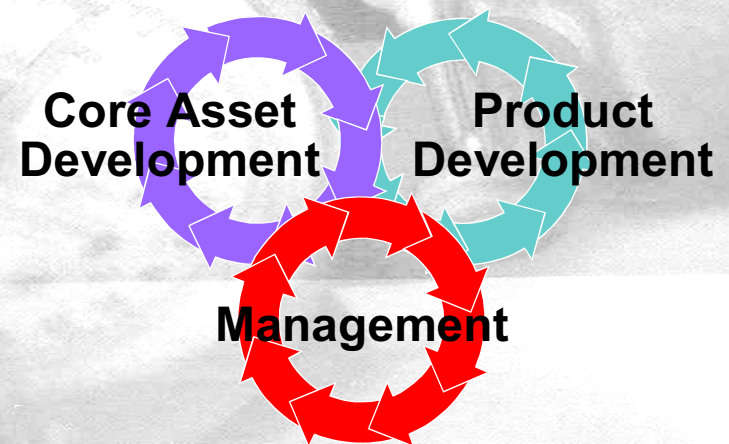
GS-EC/ESA | 8/24/2006 | Nr. 20918 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.



BOSCH

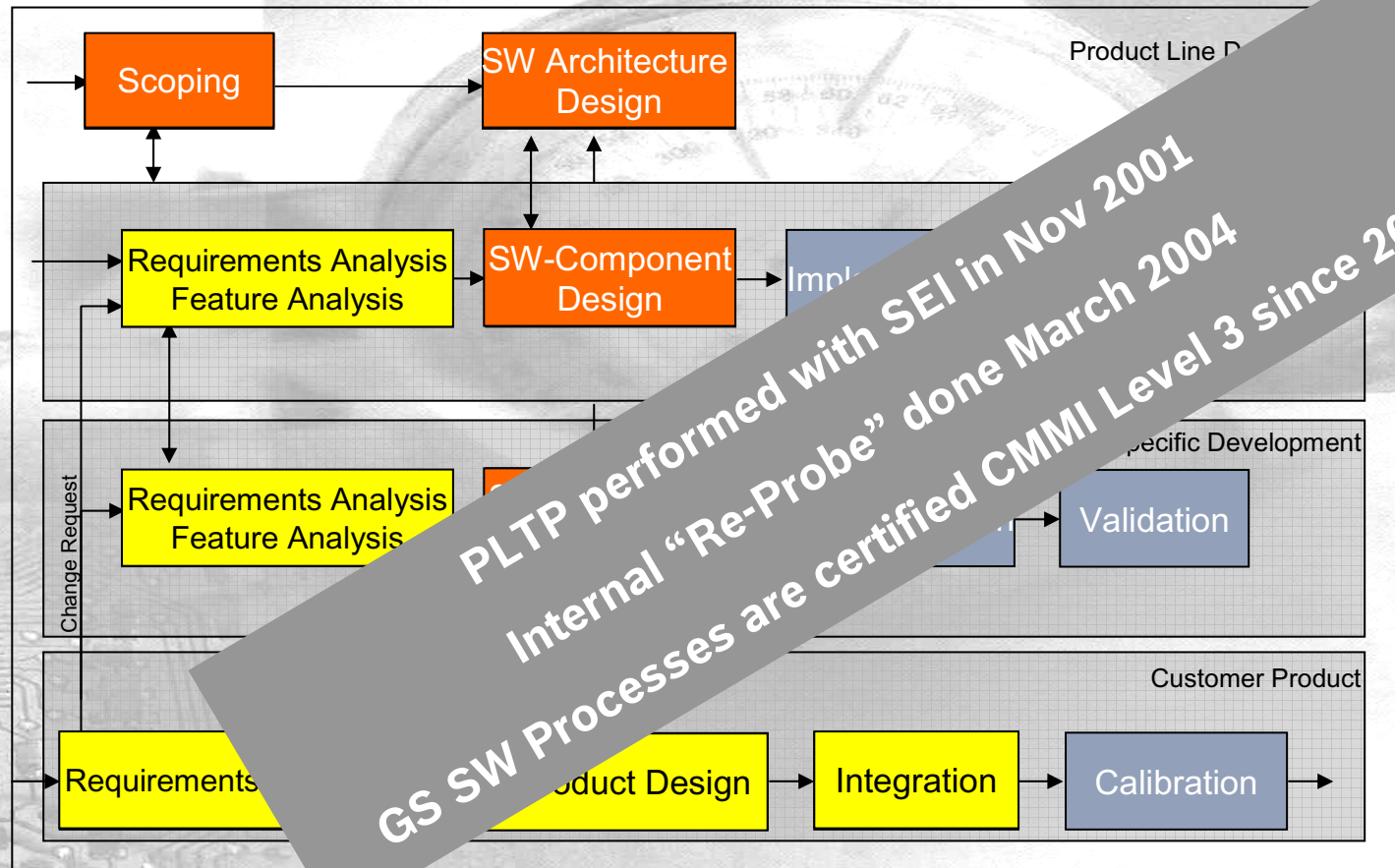
Achievements from PLA Oct. 2000 – Aug. 2006

- Organisation and processes were streamlined for product line development
- Two product lines established successfully:
 - New Value Motronic product line
 - Standard Motronic product line
- Layered architecture enabled new business model:
ECU Hardware plus HWE and Infrastructure-SW



Bosch GS-EC Software Product Line

Processes and Organization



Gasoline Systems

GS-EC/ESA | 8/24/2006 | Nr. 20918 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.

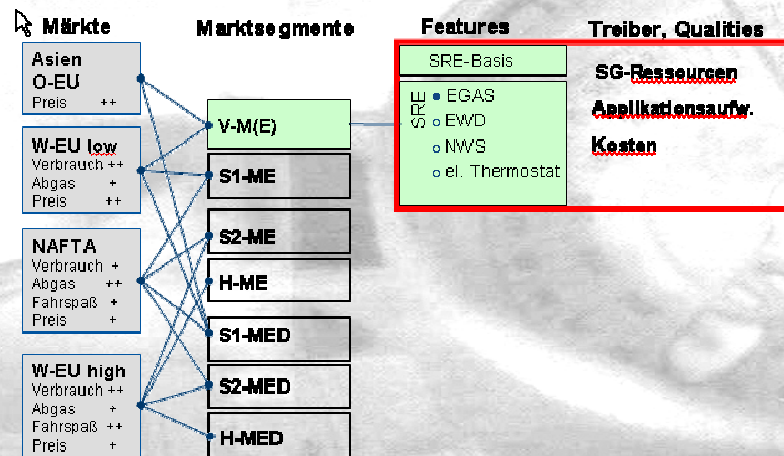


BOSCH

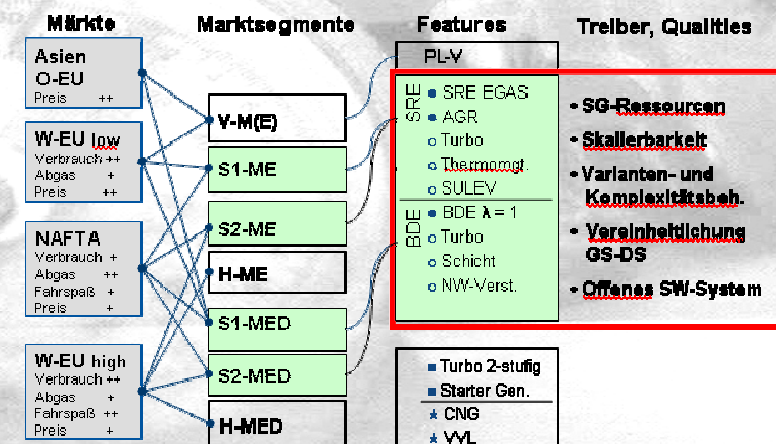
Bosch GS-EC Software Product Line

Scoping for GS Engine Control Units

Produktlinie PL-V (Value-Motronic)



Produktlinie PL-S (Standard-Motronic)



Standard product line successfully redesigned from legacy software (-25% memory consumption, improved calibratability and reusability)

New Value Motronic product line is well accepted by the addressed market segments

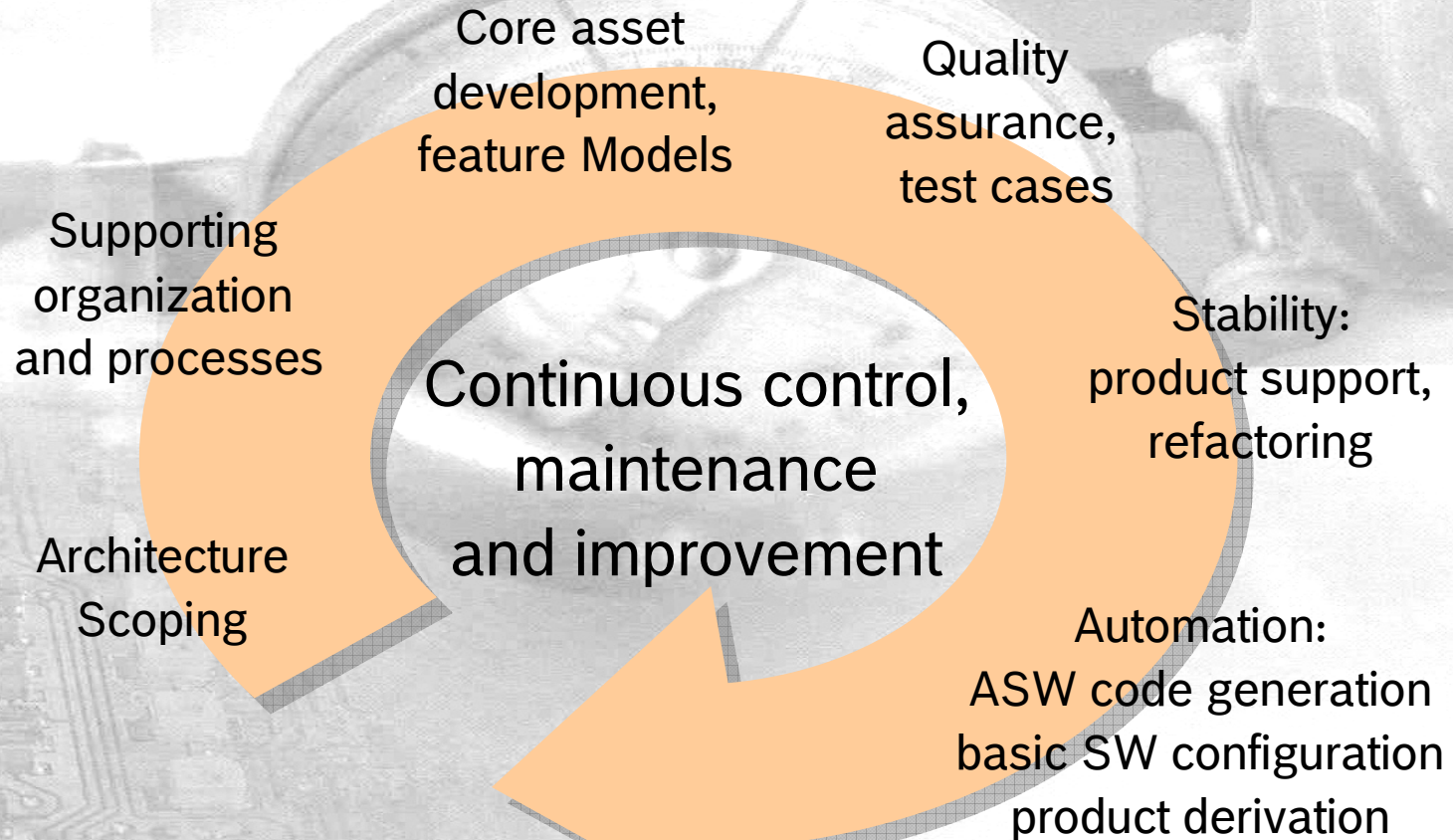
Gasoline Systems

GS-EC/ESA | 8/24/2006 | Nr. 20918 | © Robert Bosch GmbH reserves all rights even in the event of industrial property rights. We reserve all rights of disposal such as copying and passing on to third parties.



BOSCH

Elements of GS' Product Line Initiative





**10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA**

Keynote Speakers

[Carliss Baldwin](#), Harvard Business School - [Unmanageable Design Architectures: What They Are and Their Financial Consequences](#)

[Gregor Kiczales](#), University of British Columbia - [Radical Research In Modularity: Aspect-Oriented Programming and Other Ideas](#)

Unmanageable Design Architectures: What They Are and Their Financial Consequences
by Carliss Y. Baldwin

Behind every innovation lies a new design. Large or complex designs, involving many people, require architectures that create a sensible subdivision of the design tasks.

Design architectures (and the systems built from them) may be "manageable" or "unmanageable." By manageable, I mean that the artifacts created within the architecture will stay within the boundaries of a single enterprise (or a supply chain controlled by a dominant firm). Windows and Office are manageable architectures by this definition, whereas Apache and Linux are coordinated but not manageable. "Manageable" architectures give rise to product lines and product families, while "unmanageable" architectures give rise to modular clusters and open source communities.

There are important technical properties of a design architecture that affect its manageability. In this speech, I will talk about how designs draw resources from the economy, and what technical properties make an architecture "manageable" or "unmanageable." These properties, I will argue, are not good or bad in themselves, but they affect economic incentives and patterns of competition over new products and designs. Thus design architecture is an important consideration in formulating a sound product line strategy.

Radical Research In Modularity: Aspect-Oriented Programming and Other Ideas
by Gregor Kiczales

Modularity is a motherhood principle in our field. Just as politicians love to kiss babies for the camera, computer scientists love to preach the virtues of good modularity.

But what does modularity mean? In our field the idea has typically been equated with a notion of cellular or even block structure, where each block or module defines its interface with the surrounding modules. A close examination suggests this notion is too restrictive: it fails to support construction of complex systems, it fails to account for practice, and it fails even to be intuitively satisfying.

Work in biology and other fields suggests that there are many other possible kinds of modularity, and recent research in aspect-oriented programming shows what some new forms of modularity for software might be. Building on this we outline a line of attack for discovering other kinds of modularity and making productive use of new kinds of modularity in building real systems.

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



Technical Program
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

Conference Program

21-22 August 2006

Conference workshops (W), tutorials (T), and the Software Product Lines Doctoral Symposium (DS).

23-24 August 2006

Research papers (R), experience reports (E), panels (P), demonstrations, product line hall of fame, and birds-of-a-feather sessions.

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Monday, 21 August 2006			
7:00 - 8:30 Conference Registration			
8:30 - 12:00		1:00 - 5:00	
T1	An Introduction to Product Line Requirements Engineering Brian Berenbach	T4	Creating Reusable Test Assets in a Software Product Line John McGregor
T2	New Methods Behind the New Generation of Software Product Lines Success Stories Charles Krueger	T5	Leveraging Model Driven Engineering in Software Product Lines Bruce Trask, Angel Roman
T3	Introduction to Software Product Lines Patrick Donohoe	T6	Introduction to Software Product Line Adoption Linda Northrop, Larry Jones
T8	Software Product Line Variability Management Klaus Pohl, Frank van der Linden, Andreas Metzger		
W2	APLE - 1st International Workshop on Agile Product Line Engineering Organizers: Kendra Cooper, Xavier Franch		
W3	Managing Variability for Software Product Lines: Working With Variability Mechanisms Organizers: Paul Clements, Dirk Muthig		

Note: Breaks are scheduled from 10:00 - 10:30 and 3:00 - 3:30. Lunch will be served from 12:00 - 1:00.

Tuesday, 22 August 2006			
7:00 - 8:30 Conference Registration			
8:30 - 12:00		1:00 - 5:00	
T9	Domain-Specific Modeling and Code Generation for Product Lines Juha-Pekka Tolvanen	T12	Lightweight Dependency Models for Product Lines Neeraj Sangal
T10	The Scoping Game Mark Dalgarno	T13	Transforming Legacy Systems into Product Lines Danilo Beuche

T11	<i>Using Feature Models for Product Derivation</i> Olaf Spinczyk, Holger Papajewski	T14	<i>Feature Modularity in Software Product Lines</i> Don Batory
T15	<i>Generative Software Development</i> Krzysztof Czarnecki		
DS	<i>Software Product Lines Doctoral Symposium</i> Organizers: Isabel John, Len Bass, Giuseppe Lami		
W4	<i>SPLiT'06: 3rd Workshop on Software Product Line Testing</i> Organizers: Peter Knauber, Charles Krueger, Tim Trew		
W5	<i>OSSPL - First International Workshop on Open Source Software and Product Lines</i> Organizers: Frank van der Linden, Piergiorgio Di Giacomo		
5:00 - 7:00 Conference Reception			

Note: Breaks are scheduled from 10:00 - 10:30 and 3:00 - 3:30. Lunch will be served from 12:00 - 1:00.

Wednesday, 23 August 2006		
7:00 - 8:30	Conference Registration	
7:30 - 8:30	Breakfast	
8:30 - 9:00	Opening Remarks	
9:00 - 10:00	Keynote Address <i>Unmanageable Design Architectures: What They Are and Their Financial Consequences</i> Carliss Baldwin , Harvard Business School	
10:00 - 10:30	Break	
10:30 - 12:00	Session R1: Product Management Session moderator: Joe Bauman, Hewlett-Packard <i>A Practical Guide to Product Line Scoping</i> Isabel John, Fraunhofer IESE Jens Knodel, Fraunhofer IESE Theresa Lehner, Fraunhofer IESE Dirk Muthig, Fraunhofer IESE <i>Predicting Return-on-Investment for Product Line Generations</i> Dharmalingam Ganesan, Fraunhofer IESE Dirk Muthig, Fraunhofer IESE Kentaro Yoshimura, Hitachi <i>From Marketed To Engineered Software Product Lines</i> Andreas Helferich, University of Stuttgart Klaus Schmid, University of Hildesheim Georg Herzworm, University of Stuttgart	Panel P1: Product Derivation Approaches Panel moderator: David Weiss, Avaya Labs Panelists: Danilo Beuche, pure-systems Charles Krueger, BigLever Software Rob van Ommering, Philips Research Juha-Pekka Tolvanen, MetaCase Model problem: Interactive Television Applications
12:00 - 1:30	Lunch Demonstrations : IDI & BigLever	

1:30 - 3:00	<p>Session R2: Feature Modeling</p> <p>Session moderator: Gary Chastek, Software Engineering Institute</p> <p><i>A Unified Conceptual Foundation for Feature Modelling</i> Timo Asikainen, Helsinki University of Technology Tomi Männistö, Helsinki University of Technology Timo Soininen, Helsinki University of Technology</p> <p><i>Feature Models Are Views on Ontologies</i> Krzysztof Czarnecki, University of Waterloo Chang Hwan Peter Kim, University of Waterloo Karl Trygve Kalleberg, University of Bergen</p> <p><i>Weaving Behavior into Feature Models for Embedded System Families</i> Thomas Brown, Queen's University of Belfast Rachel Gawley, Queen's University of Belfast Rabih Bashroush, Queen's University of Belfast Ivor Spence, Queen's University of Belfast Peter Kilpatrick, Queen's University of Belfast Charles Gillan, Queen's University of Belfast</p>	<p>Session E1: Experience Reports</p> <p>Session chair: Peter Knauber, Mannheim University of Applied Sciences</p> <p><i>Transitioning to a Software Product Family Approach Challenges and Best Practices</i> Michael Kircher, Siemens AG Christa Schwanninger, Siemens AG Iris Groher, Siemens AG</p> <p><i>Experiences with Product Line Development of Embedded Systems at Testo AG</i> Ronny Kolb, Fraunhofer Institute for Experimental Software Engineering (IESE) Isabel John, Fraunhofer Institute for Experimental Software Engineering (IESE) Jens Knodel, Fraunhofer Institute for Experimental Software Engineering (IESE) Dirk Muthig, Fraunhofer Institute for Experimental Software Engineering (IESE) Uwe Haury, Testo AG Gerald Meier, Testo AG</p> <p><i>The JTRS Program: Software-Defined Radios as a Software Product Line</i> Eric Koski, Harris Corporation Charles Linn, Harris Corporation</p>
3:00 - 3:30	Break	
3:30 - 5:30	<p>Session R3: Realization/Derivation</p> <p>Session moderator: Svein Hallsteinsen, SINTEF ICT</p> <p><i>Organizing the Asset Base for Product Derivation</i> John Hunt, Clemson University</p> <p><i>Optimizing the Selection of Representative Configurations in Verification of Evolving Product Lines of Distributed Embedded Systems</i> Kathrin Scheidemann, BMW Car IT GmbH</p> <p><i>Service Grid Variability Realization</i> Jilles Van Gurp, Nokia Research Center, Helsinki Juha E. Savolainen, Nokia Research Center, Helsinki</p>	<p>Panel P2: Testing in a Software Product Line</p> <p>Panel moderator: Klaus Pohl, University of Duisburg-Essen</p> <p>Panelists: Georg Grütter, Robert Bosch GmbH, Germany John D. McGregor, Clemson University, USA Andreas Metzger, University of Duisburg-Essen, Germany Tim Trew, Philips Research, The Netherlands</p> <p>Model problem: The eShop Product Line</p>
5:00 - 5:30	<p><i>New Methods in Software Product Line Development</i> Charles Krueger, BigLever Software</p>	<p>DoD Experience Report</p> <p>The Advanced Multiplex Test SYstem (AMTS): A Product Line Approach for Army Aviation Maintenance Ken Capolongo</p>
6:00 - 7:30	SEI Reception	

7:30	Birds-of-a-Feather Sessions <ul style="list-style-type: none"> • DoD Birds-of-a-Feather Session • BigLever Birds-of-a-Feather Session • Other Birds-of-a-Feather Session(s)
-------------	--

Thursday, 24 August 2006		
7:00 - 8:30	Conference Registration	
7:30 - 8:30	Breakfast	
8:30 - 9:00	Workshop Reports	
9:00 - 10:00	Keynote Address Radical Research In Modularity: Aspect-Oriented Programming and Other Ideas Gregor Kiczales , University of British Columbia	
10:00 - 10:30	Break	
10:30 - 12:00	Session R4: Variability Management Session moderator: Krzysztof Czarnecki, University of Waterloo <i>Combining Feature-Oriented Analysis and Aspect-Oriented Programming for Product Line Asset Development</i> Kwanwoo Lee, Hansung University Kyo Kang, Pohang University of Science and Technology Minseong Kim, Sogang University Sooyong Park, Sogang University <i>Requirements Management for Product Lines: Extending Professional Tools</i> Klaus Schmid, University of Hildesheim Karsten Krennrich, HOOD GmbH Michael Eisenbarth, Fraunhofer IESE <i>Extending UML2 Metamodel for Complementary Usages of the «extend» Relationship within Use Case Variability Specification</i> Alexandre Braganca, Polytechnic Institute of Porto Ricardo Machado, Minho University	Session E2: Experience Reports Session chair: Paul Clements, Software Engineering Institute <i>Product Line Adoption: A Vice President's View</i> Salah Jarrad, JarrNet, LLC <i>She said, he said.</i> Ann Martin, Engenio Storage Group William (Bill) Hetrick, Engenio Storage Group <i>Using Model-Driven Engineering to Complement Software Product Line Engineering in Developing Software Defined Radio Components and Applications</i> Vikram Bhanot, PrismTech Corporation Dominick Paniscotti, PrismTech Corporation Angel Roman, PrismTech Corporation Bruce Trask, PrismTech Corporation
12:00 - 1:30	Lunch Demonstrations : Pure-Systems, Meta-case, & ESI	

1:30 - 3:00	Session R5: Run Time Dynamics Session moderator: Frank van der Linden, Philips Medical Systems <i>A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering</i> Jaejoon Lee, Fraunhofer Institute for Experimental Software Engineering (IESE) Kyo C. Kang, Pohang University of Science and Technology <i>Using Product Line Techniques to Build Adaptive Systems</i> Svein Hallsteinsen, SINTEF ICT Arnor Solberg, SINTEF ICT Erlend Stav, SINTEF ICT Jacqueline Floch, SINTEF ICT <i>PLA-based Runtime Dynamism in Support of Privacy-Enhanced Web Personalization</i> Yang Wang, University of California, Irvine Alfred Kobsa, University of California, Irvine André Van Der Hoek, University of California, Irvine Jeffery White, University of California, Irvine	Panel P3: Product Line Research Panel moderator: Liam O'Brien, Lero, The Irish Software Engineering Research Centre Panelists: Paul Clements, Software Engineering Institute, USA Kyo Kang, POSTECH, Korea Dirk Muthig, Fraunhofer IESE, Germany Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany
3:00 - 3:30	Break	
3:30 - 4:30	Product Line Hall of Fame	
4:30	Conference Ends	



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Conference Workshops

Workshop Chair: [Birgit Geppert](#), AVAYA Labs

Note: Some of the workshops extended their submission deadline. Please check the workshop descriptions for more detail. * denotes workshops with an extended deadline.

21 August 2006

W2* [APLE - 1st International Workshop on Agile Product Line Engineering](#)

Organizers: Kendra Cooper, Xavier Franch

W3 [Managing Variability for Software Product Lines: Working With Variability Mechanisms](#)

Organizers: Paul Clements, Dirk Muthig

22 August 2006

W4* [SPLiT'06: 3rd Workshop on Software Product Line Testing](#)

Organizers: Peter Knauber, Charles Krueger, Tim Trew

W5* [OSSPL - First International Workshop on Open Source Software and Product Lines](#)

Organizers: Frank van der Linden, Piergiorgio Di Giacomo

Note: In addition to the above workshops, the doctoral symposium will be held on the 22nd.

DS [Software Product Lines Doctoral Symposium](#)

Organizers: Isabel John, Len Bass, Giuseppe Lami

[Show Complete Details](#)

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



**10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA**

Location/Hotel Reservation

[Baltimore Marriott Waterfront](#)
700 Aliceanna Street
[Baltimore](#), Maryland, USA 21202

Phone: 1-410-385-3000
Fax: 1-410-895-1900
Toll-Free: 1-800-228-9290

A block of rooms has been reserved at the Baltimore Marriott Waterfront in Baltimore, MD. The room rate is \$179 and the government rate is \$141 for single/double occupancy. The cut-off date for reservations for the SPLC Conference is Thursday, August 3. After that date, the hotel cannot guarantee a room at the \$179 rate.

Conference Information

You can make your reservations two ways.

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines
Doctoral Symposium](#)
- [Software Product Line
Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program
Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

By phone:

Call Marriott Reservations at 800-228-9290 and ask for the "SPLC 2006 Conference" Rate

On line:

Go the Baltimore Marriott web site by following the link below (<http://marriott.com/property/propertypage/BWIWF>) and entering "splsp1a" in the Group Code area.

Note: The per diem rate for government attendees will be available only to active duty or civilian government employees. ID will be required upon check-in. Retired military IDs do not qualify.



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Conference Tutorials

Tutorial Chair: [Daniel J. Paulish](#), Siemens Corporate Research

[Calendar View](#)

21 August 2006

- T1** [An Introduction to Product Line Requirements Engineering](#)
Brian Berenbach
(Half Day - AM)

- T2** [New Methods Behind the New Generation of Software Product Lines Success Stories](#)
Charles Krueger
(Half Day - AM)

- T3** [Introduction to Software Product Lines](#)
Patrick Donohoe
(Half Day - AM)

- T4** [Creating Reusable Test Assets in a Software Product Line](#)
John McGregor
(Half Day - PM)

- T5** [Leveraging Model Driven Engineering in Software Product Lines](#)
Bruce Trask, Angel Roman
(Half Day - PM)

- T6** [Introduction to Software Product Line Adoption](#)
Linda Northrop, Larry Jones
(Half Day - PM)

- T8** [Software Product Line Variability Management](#)
Klaus Pohl, Frank van der Linden, Andreas Metzger
(All Day)

22 August 2006

- T9** [Domain-Specific Modeling and Code Generation for Product Lines](#)
Juha-Pekka Tolvanen
(Half Day - AM)

- T10** [The Scoping Game](#)
Mark Dalgarno
(Half Day - AM)

- T11** [Using Feature Models for Product Derivation](#)
Olaf Spinczyk, Holger Papajewski
(Half Day - AM)

- T12** [Lightweight Dependency Models for Product Lines](#)
Neeraj Sangal
(Half Day - PM)

- T13** [Transforming Legacy Systems into Product Lines](#)
Danilo Beuche
(Half Day - PM)

- T14** [Feature Modularity in Software Product Lines](#)
Don Batory
(Half Day - PM)

- T15** [Generative Software Development](#)
Krzysztof Czarnecki
(All Day)

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Tutorial 1 (T1)

An Introduction to Product Line Requirements Engineering

Brian Berenbach

21 August 2006, (Half Day - AM)

Requirements elicitation and management has become ever more important as product lines become more complex and time to market is shortened. Outsourcing has added a new dimension to requirements management, exacerbating problems associated with transitioning from analysis to design. This half day tutorial will provide an introduction to product line requirements engineering from the perspective of project and product management: how it impacts project managers, quality assurance personnel, requirements analysts, developers and testers. Topics covered will include product line requirements, feature modeling, CMMI compliant requirements management and requirements analysis processes (both UML and text based).

Business analysts who are interested in using UML for modeling will also find the course interesting. No formal knowledge of programming is required.

Tutorial 2 (T2)

New Methods Behind the New Generation of Software Product Lines Success Stories

Charles Krueger

21 August 2006, (Half Day - AM)

A new generation of software product line success stories is being driven by a new generation of methods, tools and techniques. While early software product line case studies at the genesis of the field revealed some of the best software engineering improvement metrics seen in four decades, the latest generation of software product line success stories exhibit even greater improvements, extending benefits beyond product creation into maintenance and evolution, lowering the overall complexity of product line development, increasing the scalability of product line portfolios, and enabling organizations to make the transition to software product line practice with orders of magnitude less time, cost and effort. We explore some of the important new methods such as software mass customization sans application engineering, minimally invasive transitions, bounded product line combinatorics, and product line lifecycle management.

Tutorial 3 (T3)

Introduction to Software Product Lines

Patrick Donohoe

21 August 2006, (Half Day - AM)

Software product lines have emerged as a new software development paradigm of great importance. A software product line is a set of software intensive systems sharing a common, managed set of features, and that are developed in a disciplined fashion using a common set of core assets. Organizations developing a portfolio of products as a software product line are experiencing order-of-magnitude improvements in cost, time to market, staff productivity, and quality of the deployed products.

This tutorial will introduce the essential activities and underlying practice areas of software product line development. It will review the basic concepts of software product lines, discuss the costs and benefits of product line adoption, introduce the SEI's *Framework for Software Product Line Practice*, and describe approaches to applying the practices of the framework.

Tutorial 4 (T4)

Creating Reusable Test Assets in a Software Product Line

John McGregor

21 August 2006, (Half Day - PM)

This tutorial focuses on the test assets and test processes created by a software product line organization. The tutorial will allow participants to consider how to modify existing testing practices to take advantage of strategic reuse. The software product line approach blends organizational management, technical management and software engineering principles to efficiently and effectively produce a set of related products. The major test assets: test plans, test cases, test data, and test reports are created at multiple levels of abstraction to facilitate their reuse. A product line organization also defines a test process that differs from the test process in a traditional development organization. This tutorial will allow participants to consider how to modify existing testing practices to take advantage of strategic reuse. At the end of this tutorial you will be able to:

- Understand the basic concepts of testing in software product line organizations.
 - Understand the benefits, costs and risks of creating reusable test assets.
 - Define a test process for your product line organization.
 - Identify the steps necessary to initiate these activities for your organization.
-

Tutorial 5 (T5)

Leveraging Model Driven Engineering in Software Product Lines

Bruce Trask, Angel Roman

21 August 2006, (Half Day - PM)

Model Driven Engineering (MDE) is a new innovation in the software industry that has proven to work synergistically with Software Product Line Architectures. It can provide the tools necessary to fully harness the power of Software Product Lines. The major players in the software industry including commercial companies such as IBM, Microsoft, standards bodies including the Object Management Group, and leading universities such as the ISIS group at Vanderbilt University are fully embracing this MDE/PLA combination. IBM is spearheading the Eclipse Foundation including its MDE tools. Microsoft has launched their Software Factories foray into the MDE space. Software groups such as the ISIS group at Vanderbilt are using these MDE techniques in combination with PLAs for very complex systems. The Object Management Group is working on standardizing the various facets of MDE. The goal of this tutorial is to educate attendees on what MDE technologies are, how exactly they relate synergistically to Product Line Architectures, and how to actually apply them using an existing Eclipse implementation.

Tutorial 6 (T6)

Introduction to Software Product Line Adoption

Linda Northrop, Larry Jones

21 August 2006, (Half Day - PM)

The tremendous benefits of taking a software product line approach are well documented. Organizations have achieved significant reductions in cost and time to market and, at the same time, increased the quality of families of their software systems. However, to date, there are considerable barriers to organizational adoption of product line practices. Phased adoption is attractive as a risk reduction and fiscally viable proposition. This tutorial describes a phased, pattern-based approach to software product line adoption. A phased adoption strategy is attractive as a risk reduction and fiscally viable proposition. The tutorial begins with a discussion of software product line adoption issues and then presents the Adoption Factory pattern. The Adoption Factory pattern provides a roadmap for phased, product line adoption. The tutorial covers the Adoption Factory in detail, including focus areas, phases, subpatterns, related practice areas, outputs, and roles. Examples of product line adoption plans following the pattern are used to illustrate its utility. The tutorial also describes strategies for creating synergy within an organization between product line adoption and ongoing CMMI or other improvement initiatives.

Tutorial 8 (T8)

Software Product Line Variability Management

Klaus Pohl, Frank van der Linden, Andreas Metzger

21 August 2006, (All Day)

Tutorial participants will become familiar with the key concepts of software product line engineering and will learn how to apply variability management in practice. The participants will be able to differentiate between the two processes domain engineering and application engineering, and will have an understanding of the differences between single-system development and the development activities in product line engineering. The focus will be on requirements engineering and architectural design activities, and the relationships between them. The participants will further have learned about the concept of variability, have practiced the concepts through exercises, and will be able to model variability in requirements and design artifacts by using the orthogonal variability modeling approach (OVM).

Tutorial 9 (T9)

Domain-Specific Modeling and Code Generation for Product Lines

Juha-Pekka Tolvanen

22 August 2006, (Half Day - AM)

Current modeling languages provide surprisingly little support for automating product line development. They are either based in the code world using the semantically well-defined concepts of programming languages (e.g. UML) or based on an architectural view using a simple component-connector concept. In both cases, the languages themselves say nothing about a product family or its variants. This situation could be compared to that of a programmer being asked to write object-oriented programs where the language does not support any object-oriented concepts.

Most domain engineering approaches emphasize a language as an important mechanism to leverage and guide product development in product lines. Domain engineering results in creating a language (with related tools) for the variant specification and production that goes beyond configuring pre-built components. Previously, the effort for implementing textual or graphical languages and related tools was considerably high. This limited the use of domain engineering to a few cases only and hindered the use of true product family development methods. However, recent advances in metamodeling and related technology (e.g. metamodeling tools, Software Factory concept) as well as tools provide better support for language and generator creation. This tutorial describes how to create domain-specific languages and generators to automate product derivation. We inspect 20+ industry cases on language creation and demonstrate their use with hands-on examples. Industrial experiences of this approach show remarkable improvements in productivity (5-10 times faster variant creation) as well as capability to handle complex and large product lines (more than 100 product variants).

Tutorial 10 (T10)

The Scoping Game

Mark Dalgarno

22 August 2006, (Half Day - AM)

Product Line Scoping is the activity of determining what products constitute the product line. i.e. the *Product Line Scope*. This tutorial will introduce and explore Product Line Scoping.

By the end of the tutorial participants should:

- Understand Scoping and why it is an essential Product Line activity.
- Understand Scoping as an economic decision driven by business objectives and involving Scope trade-offs.
- Understand the sources of information which underpin Scoping.
- Be able to identify stakeholders in the Scoping activity and relate this to their own organization.
- Be aware of alternative Scoping approaches.
- Understand Scoping as an iterative, on-going activity.
- Understand Scoping's position with respect to other Product Line activities.
- Know where to look for more information.

Tutorial 11 (T11)

Using Feature Models for Product Derivation

Olaf Spinczyk, Holger Papajewski

22 August 2006, (Half Day - AM)

The implementation of a software product line leads to a high degree of variability within the software architecture. For an effective development and deployment it is necessary to resolve variation points within the architecture and source code automatically during product/variant derivation. Given the complexity of most software systems tool support is necessary for these tasks. This tutorial shows how feature models combined with appropriate tools can provide this support. The importance of the separation of problem space modeling and solution space modeling is discussed. Concepts how to connect both spaces using constraints and/or generative approaches are shown. Furthermore, some typical patterns of variability in the solution space are shown and their automatic resolution in common languages like C/C++ and Java is demonstrated. Integration of code generators, aspect-oriented programming and software configuration management systems into the derivation process is also discussed. The tutorial is accompanied by demonstrations of the presented concepts with freely available tools.

Tutorial 12 (T12)

Lightweight Dependency Models for Product Lines

Neeraj Sangal

22 August 2006, (Half Day - PM)

This tutorial will present a practical technique for managing the architecture of software product lines using Lightweight Dependency Models. We will demonstrate that the matrix representation used by these models provides a unique view of the architecture and is highly scalable compared to the directed graph approaches that are common today. We will also show a variety of matrix algorithms and transformations that can be applied to analyze and organize the system into a form that reflects the architecture and demonstrates the importance of managing dependencies in product lines.

During the tutorial, we will illustrate our approach by applying it to real applications each consisting of hundreds or thousands of files. We will show how dependency models can be created for product lines and how formal design rules can be specified to manage the evolution of these architectures. Finally, we will use the actual dependency models to demonstrate how architecture evolves and how it often begins to degrade.

Tutorial 13 (T13)

Transforming Legacy Systems into Product Lines

Danilo Beuche

22 August 2006, (Half Day - PM)

Not every software product line starts from the scratch, often organizations face the problem that after a while their software system is deployed in several variants and the need arises to migrate to systematic variability and variant management using a software product line approach. The tutorial will discuss issues coming up during this migration process mainly on the technical level, leaving out most of the organizational questions. The goal of the tutorial is to give attendees an initial idea how a transition into a software product line development process could be done with respect to the technical transition. The tutorial starts with a brief introduction into software product line concepts, discussing terms such as problem and solution space, feature models, versions vs. variants. Tutorial topics are how to choose adequate problem space modeling, the mining of problem space variability from existing artifacts such as requirements documents and software architecture. Also part of the discussion will be the need for separation of problem space from solution space and ways to realize it. A substantial part will be dedicated to variability detection and refactoring in the solution space of legacy systems.

Tutorial 14 (T14)

Feature Modularity in Software Product Lines

Don Batory

22 August 2006, (Half Day - PM)

Feature Oriented Programming (FOP) is a design methodology and tools for program synthesis in software product lines. Programs are specified declaratively in terms of features. FOP has been used to develop product-lines in widely varying domains, including compilers for extensible Java dialects, fire support simulators for the U.S. Army, network protocols, and program verification tools. The fundamental units of modularization in FOP are program extensions (aspects, mixins, or traits) that encapsulate the implementation of an individual feature. An FOP model of a product-line is an algebra: base programs are constants and program extensions are functions (that add a specified feature to an input program). Program designs are expressions - compositions of functions and constants - that are amenable to optimization and analysis. This tutorial reviews core results on FOP: models and tools for synthesizing code and non-code artifacts by feature module composition, automatic algorithms for validating compositions, and the relationship between product-lines, metaprogramming, and model driven engineering (MDE).

Tutorial 15 (T15) **Generative Software Development**

Krzysztof Czarnecki

22 August 2006, (All Day)

Product-line engineering seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way. In product-line engineering, new system variants can be rapidly created based on a set of reusable assets (such as a common architecture, components, models, etc.). Generative software development aims at modeling and implementing product lines in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages (DSLs).

In this tutorial, participants will learn how to perform domain analysis (i.e., capturing the commonalities and variabilities within a system family in a software schema using feature modeling), domain design (i.e., developing a common architecture for a system family), and implementing software generators using multiple technologies, such as template-based code generation and model transformations. Available tools for feature modeling and implementing DSLs as well as related approaches such as Software Factories and Model-Driven Architecture will be surveyed and compared. The presented concepts and methods will be demonstrated using a sample case study of an e-commerce platform.



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Conference Panels

23 August 2006

P1 [Product Derivation Approaches](#)

Panel moderator: David Weiss, Avaya Labs

Model problem: [Interactive Television Applications](#)

P2 [Testing in a Software Product Line](#)

Panel moderator: Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany

Model problem: [The eShop Product Line](#)

24 August 2006

P3 [Product Line Research](#)

Panel moderator: Liam O'Brien, Lero, The Irish Software Engineering Research Centre

[Show Complete Details](#)

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line](#)
- [Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program](#)
- [Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



**10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA**

Conference and Program Committees

Conference Committee

[John D. McGregor](#), Clemson University - Conference Chair
Frank van der Linden, Philips Medical Systems - Program Chair
[Robert L. Nord](#), Software Engineering Institute - Program Chair

[Daniel J. Paulish](#), Siemens Corporate Research - Tutorials Chair
[Birgit Geppert](#), Avaya Labs - Workshop Chair
Isabel John, Fraunhofer Institute for Experimental Software Engineering - Symposium Chair
Dave Weiss, Avaya Labs - Hall of Fame Chair
[Linda M. Northrop](#), Software Engineering Institute - Steering Committee Chair
[Patrick Donohoe](#), Software Engineering Institute - Public Relations Chair
[Liam O'Brien](#), Lero - Irish Software Engineering Research Centre - Proceedings Editor
Melissa L. Russ, Space Telescope Science Institute - Local Publicity and Arrangements

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines
Doctoral Symposium](#)
- [Software Product Line
Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program
Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

For general information, contact [John D. McGregor](#).
For web site information, contact [Bob Krut](#).

Program Committees

Research Papers

Frank van der Linden, Philips, The Netherlands (Chair)
Robert L. Nord, Software Engineering Institute, USA (Chair)

Miguel Ángel Oltra, Telvent, Spain
Joe Bauman, Hewlett-Packard, USA
Gary Chastek, Software Engineering Institute, USA
Krzysztof Czarnecki, University of Waterloo, Canada
Hans Petter Dahle, ICT, Norway
Piergiorgio Di Giacomo, University of Florence, Italy
Stefania Gnesi, ISTI-CNR, Italy
Svein Hallsteinsen, SINTEF ICT, Norway
Oystein Haugen, University of Oslo, Norway
André van der Hoek, University of California, USA
Jean-Marc Jézéquel, IRISA, France
Kyo chul Kang, Pohang University of Science and Technology, Korea
Tomoji Kishi, JAIST, Japan
Philippe Kruchten, University of British Columbia, Canada
Charles W. Krueger, BigLever Software, USA
Tomi Männistö, Helsinki University of Technology, Finland
Gail Murphy, University of British Columbia, Canada
Rob van Ommering, Philips, The Netherlands
Dave Sharp, Boeing, USA
Louis J. M. Taborda, Macquarie University, Australia
Martin Verlage, Vereinigte Wirtschaftsdienste, Germany
David M. Weiss, Avaya, USA
Tanya Widen, Nokia, Finland
Marion Wittmann, Siemens, Germany

Experience Papers

John D. McGregor, Clemson University

Günter Böckle, Siemens

Sholom Cohen, Software Engineering Institute

Claudia Fritsch, Bosch

Timo Käkölä, University of Jyväskylä

Dirk Muthig, Fraunhofer Institute for Experimental Software Engineering

Judith Stafford, Tufts University



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Demonstrations

23 August 2006

1:00-1:25

[IDI](#)

[Keith L. Musser](#)

[BigLever](#)

[Charlie Kreuger](#)

24 August 2006

12:30-12:55

[Pure-Systems](#)

[Danilo Beuche](#)

[Meta-case](#)

[Juha-Pekka Tolvanen](#)

1:00-1:25

[ESI](#)

[Jason Xabier Mansell](#)

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

IDI

Product Line Studio (PLS) is a commercial tool for developing and maintaining highly scalable software product lines for large and geographically distributed teams. In this demonstration, we will use PLS to configure and build a software application for a Java-enabled mobile phone or PDA. We will select and configure optional and mandatory features, and PLS will generate both executable software and documentation tailored to the chosen configuration. You'll be able to install the customized application on your PC, phone, or PDA. You'll also see how PLS is configured to deliver this capability, how it handles variability modeling, and how to use its "collaboration" features such as change notification, approvals, asset linking, and asset validation. (<http://www.idi-software.com>)

BigLever

Gears is a software product line development tool that allows you to engineer your product line portfolio as though it is a single system. The BigLever demonstration will provide insight into how Gears allows you to shift your development focus from a multitude of products to a single software production line capable of automatically producing all of the products in your product line portfolio. The demo will spotlight key elements of Gears including feature models, product feature profiles, configurable software assets and variations points, as well as the Gears product configurator, power tools and development environment. Gears has played an instrumental role in some of the industry's most notable real-world success stories including Salion, 2004 Software Product line Hall of Fame Inductee, and Engenio/LSI Logic, 2006 Software Product Line Hall of Fame elected nominee.

Pure-Systems

pure:variants is a specialist Software Product Line toolset for the whole life cycle. This demonstration will briefly illustrate how Software Product Line methodology is seamlessly integrated into activities such as Requirements Management, Testing and Defect Tracking with pure:variants. Using a real-world example we will show how pure:variants handles requirements variability where requirements are managed in external tools (Doors, CaliberRM etc.). Code Generation for product variants using model-driven code generators will then be covered. Finally, since Product Line Engineering does not just involve applying technologies, you'll also see how pure:variants improves communication and team productivity when handling variable test cases and the inevitable bugs in core assets, and how it supports integration with existing tools used for these tasks. Come along and find out why leading companies such as Robert Bosch, Daimler Chrysler and Audi are using pure:variants in their Product Line activities.

Metacase

MetaEdit+: generate product variants from high-level product specifications.

MetaEdit+ is aimed at the expert developer looking to gain productivity by generating full code directly from models. As the modeling abstraction can be raised higher than that of programming or code visualization with UML, product development can be carried out significantly faster and with better quality. This demo shows how you can define modeling languages to describe product variants along with code generators. (<http://www.metacase.com>)

ESI

The GNSIS tool developed by the European Software Institute (ESI) is used to:

- Design, code, test and maintain the Flexible Components needed to produce the target products in a specific domain.
- Assemble the Flexible Components and Business code in a Work Order to produce the final assets.
- Analyze the final assets in terms of usage statistics, reuse metrics and traceability issues.
- Provide a detailed analysis of how to produce new programs faster, cheaper, with lower maintenance costs and to the standard required.

Depending on the complexity of the applications to be built and the programming language chosen (the tool is completely language independent) GNSIS uses between 30 and 40 Flexible Components in each specific domain.

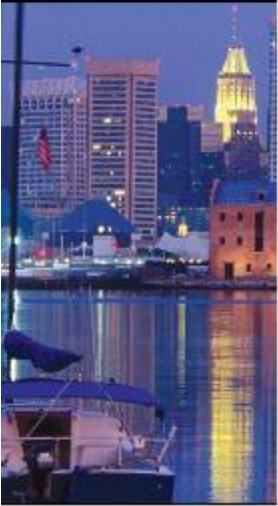


Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).

SPLC 2006



Corporate Supporters
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

Sponsored by



Carnegie Mellon
Software Engineering Institute

Software Engineering Institute
www.sei.cmu.edu

We would like to take this opportunity to thank our corporate supporters. Without their help and support, we would not be able to host such a conference.

SPLC 2006 Corporate Supporters

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Gold level



MDE Systems
www.mdesystems.com

PHILIPS

Philips Medical Systems
www.philips.com

Microsoft
Your potential. Our passion.™
Microsoft Corporation
www.microsoft.com

Silver level

AVAYA

Avaya
www.avaya.com



BigLever Software Inc.
www.biglever.com



Integrated Dynamics, Inc.
www.idi-software.com



MetaCase
www.metacase.com

NOKIA

Nokia
research.nokia.com



Pure Systems GmbH
www.pure-systems.com

If you are interested in joining this group, we invite you to sponsor SPLC by [become a corporate supporter](#).



Carnegie Mellon
Software Engineering Institute

Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Product Line Hall of Fame

Hall of Fame Chair: David M. Weiss, Avaya Labs Research

A hall of fame serves as a way to recognize distinguished members of a community in a field of endeavor. Those elected to membership in a hall of fame represent the highest achievement in their field, serving as models of what can be achieved and how. Each Software Product Line Conference culminates with a session in which members of the audience nominate systems for induction into the Software Product Line Hall of Fame. These nominations feed discussions about what constitutes excellence and success in product lines. The goal is to improve software product line practice by identifying the best examples in the field. Nominations are acted on by a panel of expert judges, who decide which nominees will be inducted into the Hall of Fame.

You can read about the current members of the Software Product Line Hall of Fame at http://www.sei.cmu.edu/productlines/plp_hof.html.

Inductees from 2005 will be announced at the SPLC 2006 Hall of Fame session.

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Criteria for Election to the Software Product Line Hall of Fame

Members of the software product line hall of fame should serve as models of what a software product line should be, exhibiting most or all of the following characteristics:

- The family that constitutes the product line is clearly identified, i.e., there is a way to tell whether or not a software system is a member of the product line, either by applying a known rule or a known enumeration.
- The family that constitutes the product line is explicitly defined and designed as a product line, i.e., the commonalities and variabilities that characterize the members of the product line are known and there is an underlying design for the product line that takes advantage of them.
- The product line has had a strong influence on others who desire to build and evolve product lines, and has gained recognition as a model of what a product line should be and how it should be built. Others have borrowed, copied, and stolen from it in creating their product lines or in expounding ideas and practices for creating product lines.
- The product line has been commercially successful.
- There is sufficient documentation about the product line that one can understand its definition, design, and implementation without resorting solely to hearsay.

Hall of Fame Judges

Paul C. Clements, Software Engineering Institute
Kyo Kang, Pohang University of Science and Technology
Charles Krueger, BigLever Software



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).

10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA



Help Publicize SPLC 2006

Feel free to copy and paste the following html snippets to use in your website.

SPLC 2006 Logo

If you want to use this logo on your website, copy and paste the following HTML into your web-page:

```
<a href="http://www.sei.cmu.edu/splc2006/?ref_url=www.cs.clemson.edu/~johnmc"
target="_blank">
  
```

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line](#)
- [Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)

- [Corporate Supporters](#)
 - [Conference & Program Committees](#)
 - [Location/Hotel](#)
 - [Past Conferences](#)
 - [Contact Information](#)
-



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Birds-of-a-Feather Sessions

The conference birds-of-a-feather (BoF) sessions will be held Wednesday, 23 August 2006, at 7:30 p.m. Additional BoF session(s) may be scheduled by contacting [John D. McGregor](#).

DoD Birds-of-a-Feather Session

Wednesday, 23 August, 7:30 p.m.

Since 1998, the SEI has held a DoD Software Product Line Workshop. Recently those workshops have been in Washington, DC in September. Because SPLC is close both in location and date, the SEI will instead sponsor a BoF session at SPLC for the DoD acquisition and contractor community.

If you have an interest in product line practice within the government, please join us to share and learn.

Reports from the previous DoD workshops may be found at: <http://www.sei.cmu.edu/productlines/workshops.html>

For more information contact, [Larry Jones](#) or [John Bergey](#).

BigLever Birds-of-a-Feather Session

Wednesday, 23 August 2006, 7:30 p.m.

Join BigLever and friends for this informal "meet and greet" session. Come discuss our latest software product line innovations and notable success stories -- and what BigLever has planned for the future.

For more information contact, [Charles Krueger](#).

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



Workshop 5 (W5)
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

OSSPL - First International Workshop on Open Source Software and Product Lines

<http://www.dsi.unifi.it/osspl06/>

22 August 2006

Organizers:

[Frank van der Linden](#), Philips Medical Systems, The Netherlands

[Piergiorgio Di Giacomo](#), University of Florence, Firenze, Italy

Contact: osspl06@dsi.unifi.it

Description

Open source software is getting much attention lately. Using open source software appears to be a profitable way to obtain good software. This is also applicable for organizations doing product line engineering. On the other hand, because of the diverse use of open source software, product line development is an attractive way of working in open source communities. However, at present open source and product line development are not related. This workshop aims to get a better understanding between the two communities to get an insight how they can profit from each other.

The workshop deals with the following issues:

- Ownership, control and management of product line assets in an open source community
- Visibility of the code: when it is valuable to share proprietary code and how to take the right decision.
- Creation of different levels of architecture visibility: proprietary, among closed consortium, public. Is this possible?
- Product line requirements, roadmaps and planning in open source development
- Using the open source community to evolve components and being explicit about variability
- Variability representation and management in an open source community
- Open source for the platform and in applications
- Cohabitation of product line management and agile processes
- Open source asset management tools in product line development
- The meaning of domain and application engineering in an open source context
- Recognition and recovery of a product line in an open source asset base
- Aspects dealing with evolutionary, variability or distribution of development relating to legal risks involving: liability, warranties, patent infringements etc.

Submission (extended!): The extended deadline for submissions is June 15, 2006. For more information please visit the workshop homepage at <http://www.dsi.unifi.it/osspl06/>.

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



Panel 1 (P1)
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

Panel on Product Derivation Approaches

23 August 2006

Panel moderator: David Weiss, Avaya Labs

Panelists:

Danilo Beuche, pure-systems
Charles Krueger, BigLever Software
Rob van Ommering, Philips Research
Juha-Pekka Tolvanen, MetaCase

Abstract

This panel looks at product derivation approaches and their differences, strengths and weaknesses in different PLE situations. Each panelist will examine a common problem (the [Interactive Television Applications](#)) and provide an overview of their product derivation approach and how it was used to solve the problem.

Overview

At some point, no matter how wonderful your product line process is, you have to ship the products. The panelists will each present a different approach to PLE, concentrating on how actual products are derived from specifications. The approaches presented include feature modeling, architecture description languages, UML and domain-specific modeling languages.

A common product specification and derivation task will be given to all panelists, and they will show how their approach works on it. The audience can - and is warmly encouraged to - participate, ask additional questions, heckle, and hopefully laugh. A major goal is to identify the classes of PLE situations that best suit each approach.

Following are some of the questions and issues to be addressed by the panel.

1. How large a portion of a product is automatically derived? Please answer in terms of some reasonably precise measure, such as percent of modules, classes, or KNCSL, or coverage in a feature model.
2. How are new features and functionality developed? Give an example, if possible.
3. What is the cost and time to create a new feature or change the application platform, e.g., in hours of effort as a fraction of effort needed to create the application engineering environment? Alternatively, how would you estimate the cost and time?

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).

Interactive Television Applications

1 Abstract

Digital television allows interactive content to accompany standard broadcasts. The development of bespoke interactive content is expensive. You are to design a system that will allow the non-technical producers of television programmes to build interactive content from a set of high-level building blocks.

2 Background

Digital television is becoming increasingly popular in the UK. In addition to providing higher quality video and increased channel capacity, it allows interactive content to accompany standard broadcasts. Interactive applications have been used to enhance traditional broadcasts in many ways:

- Viewers can play along with quizzes (Figure 1).
- Viewers can choose different camera angles during sporting events.
- Viewers can take part in discussions and comment on events through message boards.
- Viewers can remind themselves of the important developments in a drama's plot (Figure 2).

Viewers can receive digital television via satellite, cable or a normal aerial (terrestrial), given the necessary decoding hardware. Each reception mechanism has its own standard defining how interactive applications are created.



Figure 1. Playing along with *Test the Nation*

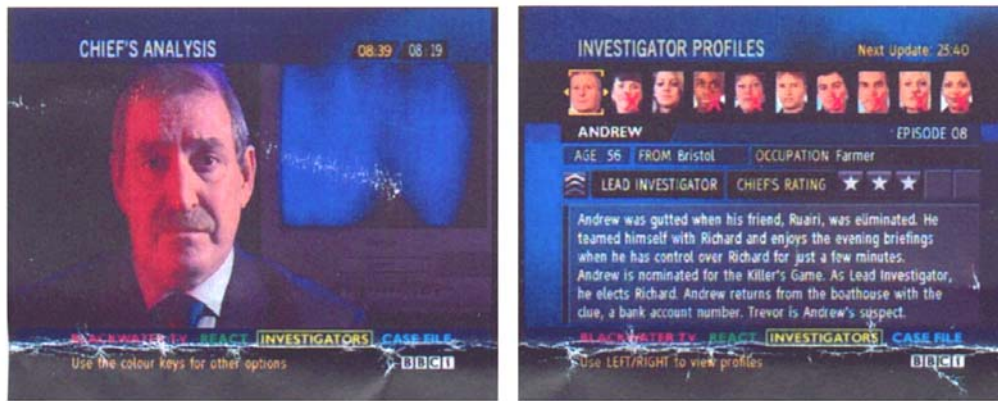


Figure 2. Viewing developments in *The Murder Game*

3 Problem

Currently, each interactive application is bespoke. This greatly limits the number of programmes that can be accompanied by interactive content, as the applications are expensive to develop. This is compounded by the fact that three different applications must be developed, one for each of the satellite, cable, and terrestrial platforms. You are to design a system that will allow non-technical producers to build applications to accompany their programmes.

The application will sit on the right hand side of the screen (Figure 3), and display one or more of the following pieces of content:

- A page of text, to be used for news stories, background information etc.
- A multiple choice selection for voting, for example "Man of the match" in a football match.
- A menu that allows the user to view items of content, including sub-menus.

The basic on-screen layout and navigational structure of the application has been defined by the user interface department, and producers aren't able to change it.



Figure 3. Different types of content

4 Scenarios

The following set of scenarios emerged during discussions with the producers. They are ordered by their value to the producers, most valuable first.

4.1 Case 1

A producer would like to use a page of text to provide analysis of the recent events in a rugby match. A journalist with a laptop will need to change the text on the page throughout the match, from the stadium. The journalist shouldn't be able to change any other content.

4.2 Case 2

A producer would like to customise the colours and graphics used in the application, so that it better matches the branding of their programme.

4.3 Case 3

A producer has built their application within the system, and would like it to appear on a particular TV channel.

4.4 Case 4

A director can edit the page after the journalist finishes with it. Editing should include the ability to undo recent changes to the page. The director can concurrently put pages of text from different journalists on the part of the screen that the application controls.

4.5 Case 5

A panelist in a discussion programme has just made a controversial statement. The producer would like to add a vote to the accompanying application, asking the viewers if they agree.

4.6 Case 6

A viewer has complained that the application accompanying a programme last week contained libellous statements. The legal department have asked for a record of all interactive content that was broadcast during the programme.

4.7 Case 7

During the closing stages of the biggest sporting event of the year, the machine hosting your system catches fire. The system administration people need to move your system onto another machine, without losing content that has been entered, and before anything

important happens in the game.

5 Interactive application architecture

Designing interactive applications requires relatively detailed knowledge of the standards for each platform. For this reason, you should concentrate on the system used by producers to define the application content, and its interfaces to black box components that build the actual application. To define these interfaces, you will need some knowledge of the basic architecture of interactive applications. The following crash-course should suffice.

Digital televisions contain a basic operating system, and a set of libraries providing functions to display text and graphics, change the currently displayed video stream, etc. Applications are designed and specified with domain-specific modelling language (that you develop), including possibly a menu-driven approach, a table-driven approach, a palette approach, or other approach easy for producers to use, and delivered to digital televisions by inserting it as XML into the same broadcast stream that contains the video and audio content. Once running, messages can be sent to an application by inserting them into the broadcast stream. Applications can send reply messages to your system, usually via a standard modem built into the television. Sending these reply messages is very slow, and involves the viewer paying call charges. For these reasons, reply messages can only be used for viewer initiated actions, such as responding to a vote.



Panel 2 (P2)
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

Panel on Testing in a Software Product Line

23 August 2006

Panel moderator: Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany

Panelists:

Georg Grütter, Robert Bosch GmbH, Germany

John D. McGregor, Clemson University, USA

Andreas Metzger, University of Duisburg-Essen, Germany

Tim Trew, Philips Research, The Netherlands

Abstract

This panel is about system testing of software product line artifacts. The panelist will present different approaches for software product line testing. Together, we will discuss their pros and cons. As a kind of benchmark, a common example of an online store ([The eShop Product Line](#)) will be used to ease the comparison of the different testing approaches.

Overview

Each panelist will present an approach to test the domain and application artifacts in software product line engineering. The decision whether to test the domain artifacts in domain engineering or if testing is delayed to application engineering is left to the panelists.

To facilitate a better comparison of the different test approaches, each panelist will illustrate his approach using a running example of an online store product line.

The discussions will, among others, cover the following questions:

- Should there be system testing in domain engineering, or should system tests be performed during application engineering only?
- Which test artifacts can be reused during product line testing?
- Is there an advantage of creating domain test artifacts which are reused during application engineering?
- Can application test cases be generated? And if so, should they be generated from domain test cases or just from application engineering artifacts?
- Does the model-based test case derivation offer benefits when compared with deriving test cases directly from natural language requirements?

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).

The eShop Product Line

Klaus Pohl^{*, +} and Andreas Metzger^{*}

^{*} Software Systems Engineering,
University of Duisburg-Essen
Schützenbahn 70, 45117 Essen, Germany

⁺ Lero (The Irish Software
Engineering Research Center)
University of Limerick, Ireland

1 Situation

A manufacturer of online store software has provided a specification of his eShop software product line. He asks you – as an expert in the field of software product line testing – to advise him on how to perform system tests for his eShop product line. Especially, he wants two concrete eShop applications to be tested (see Sect. 3).

The commonalities and the variability of the product line have been defined by product management, considering market trends and technical constraints. The variability has been explicitly documented in a variability model (see right hand side of Figure 1).

Further, common and variable requirements of the product line have been elicited and documented by use cases (see use case diagram on the left hand side of Figure 1 and the use case descriptions in Sect. 4). In addition, the manufacturer of the online store provides you with detailed scenario descriptions, including pre- and post-conditions, scenario steps, alternative scenarios, etc.

2 Domain Description

Running an eShop shall allow an online merchant to sell his goods to customers via the internet. An online merchant typically approaches the manufacturer of the eShop product line with specific needs on the functionality of the eShop. Based on the available variability of the product line, the manufacturer will then be able to create an individual application specifically for the merchant.

The variability of the eShop product line is shown on the right hand side of Figure 1. The variability model is documented using the OVM approach (see Pohl, K.; Böckle, G.; van der Linden, F. Software Product Line Engineering – Foundations, Principles, and Techniques, Springer, Berlin, Heidelberg, New York, 2005). Triangles document variation points (“what does vary?”), rectangles document variants (“how does it vary?”), and relations between these elements describe constraints on the possible choice of variants; e.g., for the variation point “bonus” at most one variant may be selected.

The left hand side of Figure 1 shows the use case diagram of the eShop product line. The variable use cases are identified by trace links from the variability model to the use case elements.

2.1 Commonalities of the eShop Product Line

All applications of the eShop product line share the common functionality: **register customer**, **buy product**, and **search product**. For searching a product, the customers can enter the name of the product or an order number.

The **buy product** use case includes the use case **search product** (which can be repeated as often as needed). Each product that a customer wants to buy can be placed in a shopping cart. Once the customer wants to complete his/her purchase, he/she can check out and pay the ordered goods, thereby triggering the dispatch of the goods.

Before customers can buy any goods in the eShop, they must register (use case **register customer**) such that their identity (i.e., name, billing and dispatch address) is known.

2.2 Variability in the eShop Product Line

The eShop product line contains (for simplification reasons only) five variation points (see the variability model in Figure 1). These variation points – together with their variants – allow a total of 72 applications to be derived from the product line.

VP1: Register Type

Customers have to register for the eShop before they are allowed to purchase goods. Therefore, in each application of the eShop product line, at least one of the two different types of registration has to be contained (documented in the variability model by the constraint 1..2).

The two different kinds of registration are described by the use cases **register normally** and **register completely**. During the normal registration process, customers provide their e-mail address and their postal address. During the register completely process, in addition to the address information of the register normally use-case, the bank account (e.g., account number) must be provided and the customers must agree that the online merchant can contact the bank for more information.

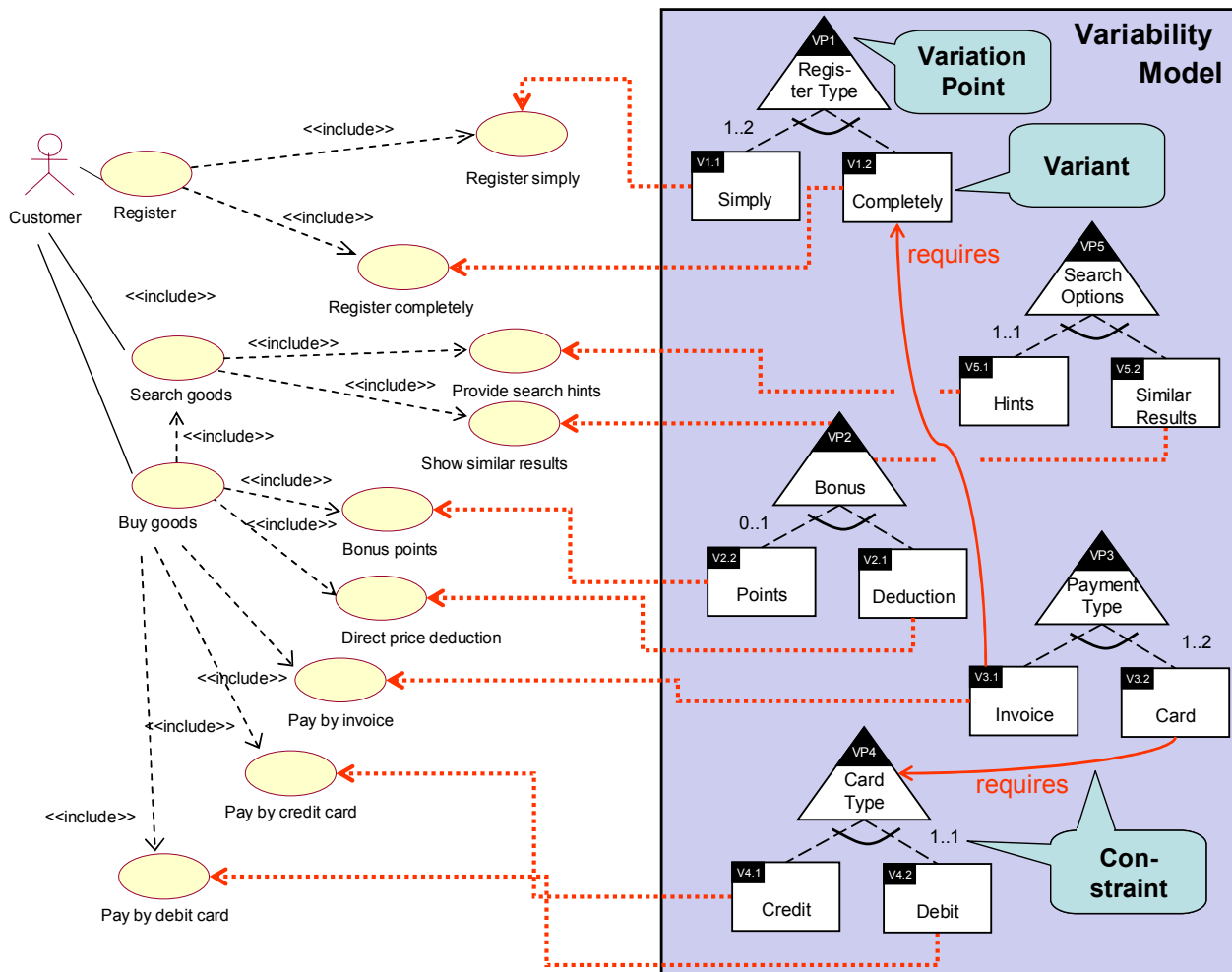


Figure 1: Variability Model and Use Case Diagram of eShop Product Line

VP2: Bonus

The merchant can optionally choose to offer his customers a bonus program. If he decides to do so, he has to decide on whether **bonus points** can be collected or whether a **direct price deduction** will be offered once the value of the order exceeds a certain amount.

The bonus points are calculated based on the goods that have been previously ordered and paid. When offering a direct price deduction, this deduction will be directly reduced from the invoiced amount.

The product line offers only one of the two types of bonus programs to be included in an eShop application (documented by the constraint 0..1 in the variability model).

VP3: Payment Type

The eShop product line offers two major kinds of payments. First, an eShop can allow the customers to **pay by invoice**. Second, the eShop application can offer a **payment by card**. An eShop application can offer both types of payments at once.

If an eShop application shall offer **pay by invoice**, the variant **register completely** is required such that the bank information of a customer is available as a security (or guarantee).

VP4: Card Type

Once the merchant has decided to offer **payment by card**, he further has to refine his selection. Product management of the eShop manufacturer has decided to allow a choice between the (mutually exclusive) alternatives **pay by credit card** and **pay by debit card**. This is expressed by the 1..1 constraint.

VP5: Search Options

The customer can search for a product by entering the product name or the order number of the product using the **search product** use case. If the customers provide an incorrect product name or order number, the **search product** use case might deliver no hits. By using the variant **provide search hints**, the eShop customer is offered a help text on how to modify his search request for better results.

In contrast, the **show similar results** variant will automatically modify the search request of the customer and display all similar results.

Only one of the two variants (show similar results and provide search hints) can be chosen, expressed by 1..1 constraint.

3 Description of the Applications

Two applications have been derived from the product line so far and shall be tested by you. The applications are defined by the variants that have been bound.

Application 1:

- VP1: Register Type = Completely (V1.2)
- VP2: Bonus = Points (V2.2)
- VP3: Payment Type = Invoice (V3.1) + Card (V3.2)
- VP4: Card Type = Debit (V4.1)
- VP5: Search Options = Hints (V5.1)

Application 2:

- VP1: Register Type = Simply (V1.1) + Completely (V1.2)
- VP2: Bonus = -none- (not desired by merchant)
- VP3: Payment Type = Invoice (V3.1)
- VP4: Card Type = -none- (VP has not to be bound)
- VP5: Search Options = Similar results (V5.2)

4 Use Cases

The following tables present the use case descriptions of the eShop product line.

The use case descriptions are based on a suggestion for documenting variable use cases by Bertolino et al. (see Bertolino, A.; Fantechi, A.; Gnesi, S.; Lami, G.; Maccari, A.; „Use Case Description of Requirements for Product Lines”; Proceedings of the International Workshop

on Requirements Engineering for Product Lines 2002 (REPL'02), Technical Report: ALR-2002-033, AVAYA Labs, 2002).

The dependencies between the use cases are expressed by pre- and post-conditions or are shown in Figure 1 by means of the <<includes>> relationships.

Use Case Name	Register	
Brief Description	A customer must register before purchasing goods	
Actors	Customer	
Goal	Register to be able to purchase goods	
Trigger	A customer wants to buy goods online	
Pre-condition	-	
Result	The customer is registered	
Post-condition	The customer is registered	
	Step	Action Description
Main Scenario	1	Customer activates registration
	2	System shows the form to be filled in {VP1}
	3	System shows the data of the customer
	4	Customer confirms his data
	5	System shows the main page of the eShop.
Scenario Extensions	4a	The customer has found an error in his data and wants to correct it; Scenario continues at Step 2.
Variation Points		Variants
VP1		V1.1: UC Register simply V1.2: UC Register completely

Use Case Name	Buy goods	
Brief Description	A customer searches, orders and pays goods that he has selected	
Actors	Customer	
Goal	Buy goods	
Trigger	Customers begins his purchase	
Pre-condition	Customer has been registered	
Result	Goods are ordered, payment information is known	
Post-condition	-	
	Step	Action Description
Main Scenario	1	Customer searches goods (refined by UC Search goods)
	2	Customer selects goods
	3	System adds goods to the shopping cart
	4	Customer checks out
	5	System calculates and shows amount to be paid
	6	System requests payment information {VP3}
	7	Customer confirms order
	8	System executes order
	9	System triggers dispatch of the goods
	10	Customer leaves the eShop
Scenario Extensions	5a & 10a	Customer likes to shop for additional goods; Scenario continues at step 1 {VP2}
Variation Points		Variations

VP2	V2.1: UC Direct price deduction V2.2: UC Bonus points
VP3	V3.1: UC Pay by invoice V3.2: requires {VP4}
VP4	V4.1: UC Pay by credit card V4.2: UC Pay by debit card

Use Case Name		
Brief Description		
Customer searches goods in the eShop		
Actors		
Customer		
Goal		
Find goods		
Trigger		
Customer clicks on “search goods” button		
Pre-condition		
-		
Result		
Search results are presented to customer		
Post-condition		
-		
	Step	Action Description
Main Scenario	1	Customer enters a search term
	2	Customer initiates search
	3	System presents search results
	4	Customer chooses desired results
	5	Systems shows details on the selected goods
Scenario Extensions	1a1	Customer chooses detailed search
	1a2	System presents “detailed search” form
	1a3	Customer enters details for searching the goods
	3a1	System shows that no results have been found; Scenario continues at step 1
	4a	Customer starts a new search, because he did not find what he searched for; Scenario continues at step 1
Variation Points		Variations
VP5	3b1	V5.1 (UC Provide search hints): System gives search hints; Scenario continues at step 1
	3b2	V5.2 (UC Show similar results): System shows similar results; Scenario continues at step 3

Use Case Name		
Register simply		
Brief Description		
The customer registers with the eShop in a simple way		
Actors		
Customer		
Goal		
see UC Register		
Trigger		
Alternative 1: The customer has chosen to register simply Alternative 2: The eShop only offers UC Register simply		
Pre-condition		
-		
Result		
Customer has filled in the form for simple registration		
Post-condition		
see UC Register		
	Step	Action Description
Main Scenario	1	The system presents a registration form
	2	The customer fills the fields e-mail address and postal address
	3	System checks the plausibility of the input (e.g., correctness of e-mail address)
Scenario Extensions	3a	System detects an error in the input data; Scenario continues at step 1

Use Case Name	Register completely	
Brief Description	The customer registers completely with the eShop	
Actors	Customer	
Goal	see UC Register	
Trigger	Alternative 1: The customer has chosen to register completely Alternative 2: The eShop only offers UC Register completely	
Pre-condition	-	
Result	Customer has filled in the form for complete registration	
Post-condition	see UC Register	
	Step	Action Description
Main Scenario	1	The system presents a registration form (part 1)
	2	The customer fills the fields e-mail address and postal address
	3	System checks the plausibility of the input (e.g., correctness of e-mail address)
	4	The system presents a registration form (part 2)
	5	The customer (additionally) fills in his bank account information (e.g., IBAN, ...)
	6	System checks the plausibility of the input (e.g., validity of the IBAN)
Scenario Extensions	3a	System detects an error in the input data; Scenario continues at step 1
	5a	System detects invalid bank account information; Scenario continues at step 4

Use Case Name	Bonus points	
Brief Description	Customer orders goods and receives bonus points	
Actors	Customer	
Goal	Buy goods	
Trigger	Customer proceeds to checkout in UC Buy goods	
Pre-condition	-	
Result	Order of goods, update of bonus points	
Post-condition	-	
	Step	Action Description
Main Scenario	1	System calculates bonus points and adds them to the bonus points of the customer

Use Case Name	Direct price deduction	
Brief Description	Customer orders goods and receives a price deduction	
Actors	Customer	
Goal	Buy goods	
Trigger	Customer proceeds to checkout in UC Buy goods	
Pre-condition	-	
Result	Order of goods, update of bonus points	
Post-condition	-	
	Step	Action Description
Main Scenario	1	System calculates price deduction and reduces the invoiced amount

Use Case Name	Pay by invoice	
Brief Description	Customer pays his goods by invoice	
Actors	Customer	
Goal	Buy goods and pay	

Trigger	Customer proceeds to checkout	
Pre-condition	Alternative 1: Customer has chosen to pay by invoice Alternative 2: System only offers UC Pay by invoice	
Result	Goods have been ordered; payment has been authorized; invoice information has been stored, and bill has been printed	
Post-condition	-	
	Step	Action Description
Main Scenario	1	System requests billing address
	2	Customer enters billing address
	3	Customer authorizes transaction
	4	System checks validity of billing address
	5	System requests confirmation of billing address and form of payment
	6	Customer confirms
Scenario Extensions	5a1	Billing address is faulty; System shows errors in billing address
	5a2	System requests modification/correction of address
	5a3	Customer corrects address; Scenario continues at step 3

Use Case Name	Pay by debit card	
Brief Description	Customer pays his goods by debit card	
Actors	Customer	
Goal	Buy goods and pay	
Trigger	Customer proceeds to checkout	
Pre-condition	Alternative 1: Customer has chosen to pay by debit card Alternative 2: System only offers UC Pay by debit card	
Result	Goods have been ordered; payment information is available; payment has been processed by the customer’s bank	
Post-condition	-	
	Step	Action Description
Main Scenario	1	System requests debit card details
	2	Customer enters debit card details
	3	Customer authorizes transaction
	4	System checks validity of debit card
	5	System requests confirmation of debit card details and form of payment
	6	Customer confirms
Scenario Extensions	4a1	Debit card details are faulty; System shows errors in debit card details
	4a2	System requests modification/correction of debit card details
	4a3	Customer corrects debit card details; Scenario continues at step 3
	1a1	System identifies the customer as having registered completely and therefore presents the account information (i.e., the debit card details)
	1a2	System requests a confirmation of the data
	1a3	Customer confirms

Use Case Name	Pay by credit card
Brief Description	Customer pays his goods by credit card
Actors	Customer
Goal	Buy goods and pay
Trigger	Customer proceeds to checkout
Pre-condition	Alternative 1: Customer has chosen to pay by credit card Alternative 2: System only offers UC Pay by credit card
Result	Goods have been ordered; payment information is available; payment has been processed by the credit card company
Post-condition	-

	Step	Action Description
Main Scenario	1	System requests credit card details
	2	Customer enters credit card details
	3	Customer authorizes transaction
	4	System checks validity of credit card
	5	System requests confirmation of credit card details and form of payment
	6	Customer confirms
Scenario Extensions	4a1	Credit card is not valid; System shows error and requests correction of card information (or to provide an alternative credit card)
	4a2	Customer corrects credit card details; Scenario continues at step 3

Use Case Name	Provide search hints	
Brief Description	The system provided no matches for the query; Therefore, hints for searching are provided	
Actors	Customer	
Goal	The customer shall find results for his query	
Trigger	No search results	
Pre-condition	-	
Result	Search hints are given	
Post-condition	-	
	Step	Action Description
Main Scenario	1	The system notifies the customer that no results have been found
	2	The system provides search hints

Use Case Name	Show similar results	
Brief Description	The system provided no matches for the query; Therefore, similar results are automatically retrieved	
Actors	Customer	
Goal	The customer shall find similar results to his query	
Trigger	No search results	
Pre-condition	-	
Result	Similar results are shown	
Post-condition	-	
	Step	Action Description
Main Scenario	1	The system computes and shows similar results



DoD Experience Report
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

The Advanced Multiplex Test SYstem (AMTS): A Product Line Approach for Army Aviation Maintenance

Ken Capolongo

Army Aviation vehicles are complex systems of systems and require many resources to operate and sustain, especially in a combat environment where aircraft availability and readiness are essential to the successful completion of battlefield missions. The Communications-Electronics Life Cycle Management Center (C-E LCMC) Software Engineering Center (SEC) is responsible for providing diagnostic products to support these aircraft in the field and is facing the challenge to produce more products with similar or fewer resources, brought on by the current business environment and operational tempo (OPTEMPO). This paper discusses how the C-E LCMC SEC is meeting the challenge through the adoption of software product line engineering practices and the development of a reliable software product framework. The framework and practices are used to facilitate production of avionics maintenance software products that improve avionics field maintenance practices, reduce sustainment costs, and increase aircraft readiness.

Conference Information Outline:

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

- The AMTS Concept
- Organization History and Mission
- Business Goals: Platform, System, and Subsystem Maintenance
- Software Architecture
- Obstacles: Organizational, Funding, and Utility
- Adoption: Implemented Strategies and lessons learned
- Benefits: Analysis of Collected Metrics
- Growth and Future Opportunities



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



Panel 3 (P3)
10th International Software Product Line Conference
(SPLC 2006)
21-24 August 2006
Baltimore, Maryland, USA

Panel on Product Line Research:

Lessons Learned from the last 10 years and Directions for the next 10

24 August 2006

Panel moderator: Liam O'Brien, Lero, The Irish Software Engineering Research Centre

Panelists:

Paul Clements, Software Engineering Institute, USA

Kyo Kang, POSTECH, Korea

Dirk Muthig, Fraunhofer IESE, Germany

Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Abstract

This panel is about past and future research in software product lines. The panelists will look back at the past 10 years to examine outcomes and lessons learned and will look forward to the next 10 years and will give potential outcomes and directions for the future of software product line research. The outcomes will be examined for relevance to the practitioner community.

Overview

Each panelist will present their lessons learned from the past and the directions for the next 10 years. Several industry judges will be asked to make a determination as to how useful the outcomes have been or will be for practitioners. The panelists will have an opportunity to respond to the judges comments and this will lead to a general discussion.

The discussions will, among others, cover the following questions:

- What have been the main lessons and outcomes for research in software product lines over the past 10 years?
- What will likely be the major directions and outcomes for research in software product lines over the next 10 years?
- How relevant have the past outcomes been for practitioners and how relevant will future outcomes be for practitioners?
- What will be the next big breakthrough for software product line research?



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Conference Workshops

Workshop Chair: [Birgit Geppert](#), AVAYA Labs

Note: Some of the workshops extended their submission deadline. Please check the workshop descriptions for more detail. * denotes workshops with an extended deadline.

21 August 2006

W2* [APLE - 1st International Workshop on Agile Product Line Engineering](#)

Organizers: Kendra Cooper, Xavier Franch

W3 [Managing Variability for Software Product Lines: Working With Variability Mechanisms](#)

Organizers: Paul Clements, Dirk Muthig

22 August 2006

W4* [SPLiT'06: 3rd Workshop on Software Product Line Testing](#)

Organizers: Peter Knauber, Charles Krueger, Tim Trew

W5* [OSSPL - First International Workshop on Open Source Software and Product Lines](#)

Organizers: Frank van der Linden, Piergiorgio Di Giacomo

Note: In addition to the above workshops, the doctoral symposium will be held on the 22nd.

DS [Software Product Lines Doctoral Symposium](#)

Organizers: Isabel John, Len Bass, Giuseppe Lami

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Workshop 2 (W2)

APLE - 1st International Workshop on Agile Product Line Engineering

<http://www.lsi.upc.edu/events/aple/>

21 August 2006

Organizers/Contacts:

[Kendra Cooper](#), Dept. of Computer Science, University of Texas at Dallas - Texas, USA

[Xavier Franch](#), Software Department, Universitat Politècnica de Catalunya - Catalunya, Spain

Description

The need to rapidly develop high quality, complex software continues to drive research in a number of (separate) areas in the software engineering community. For example, software product line development techniques have been of keen interest as means to re-use and tailor technical assets including models (requirements specifications, design), implementation, and test cases. A main focus in this area is to effectively create sets of related products by re-using and tailoring managed assets. Agile development techniques have also been proposed to rapidly develop software by focusing on developing working code; they seek to minimize the amount of documentation, process definition, and model development. It is interesting to note that although the goals of the two techniques have similarities (rapidly develop high quality, complex software), the solutions to realize the goals in the techniques seem to conflict. The theme of this

workshop is to probe the following question: Given the similar goals but different foci of agile and product-line development techniques, to what degree can (or should) they be integrated?

Publication of Selected Papers

It's our pleasure to announce a special issue on "Agile Product Line Engineering" of the Journal of Systems and Software (JSS, Elsevier). Extended versions of selected papers presented at this workshop will be considered for publication in this issue.

Submission (extended!): The extended deadline for submissions is June 9, 2006. For more information please visit the workshop homepage at <http://www.lsi.upc.edu/events/aple/>.

Workshop 3 (W3)

Managing Variability for Software Product Lines: Working With Variability Mechanisms

http://www.sei.cmu.edu/splc2006/variability_workshop.html

21 August 2006

Organizers:

[Paul Clements](#), Software Engineering Institute

[Dirk Muthig](#), Fraunhofer Institute for Experimental Software Engineering

Contact: clements@sei.cmu.edu

Description

Managing variability is the essence of software product line practice. Variability enters the product line picture through the need for different features, deployment on different platforms, the desire for different quality attributes, and the accommodation of different deployment scenarios. Eventually, every need for variability manifests itself in one way or another in the actual artifacts that populate a product line's core asset base. "Variability mechanisms" is the name we give to the constructs that achieve variation at the artifact level. Selecting the correct variability mechanism(s) can have a dramatic effect on the cost to deploy new products, react to evolutionary pressures, and in general maintain and grow the product line. But selection remains an ad hoc process in nearly all product line organizations. This workshop is intended to fill the void between variability requirements visible to those who deal with features and other product-level concerns, and the variability mechanisms visible to creators and consumers of a product line's core assets. The goal of the workshop is to begin to codify a body of knowledge for the informed and purposeful selection of variability mechanisms to use in a software product line's core assets. The workshop will be highly interactive and focused on making tangible progress towards answering specific questions relating to best practices in variability management.

Submission: The deadline for submissions is July 7, 2006. For more information please visit the workshop homepage at http://www.sei.cmu.edu/splc2006/variability_workshop.html.

Workshop 4 (W4)

SPLiT 2006 - 3rd Workshop on Software Product Line Testing

<http://www.biglever.com/split2006/>

Organizers:

[Peter Knauber](#), Mannheim University of Applied Sciences, Germany

[Charles Krueger](#), BigLever Software, Austin, TX, USA

[Tim Trew](#), Philips Research, Eindhoven, The Netherlands

Contact: split@biglever.com

Description

Product line engineering (PLE) has become a major topic in industrial software development, and many

organizations have started to consider PLE as state of the practice. One topic that needs greater emphasis is testing of product lines. Product line testing is crucial to the successful establishment of PLE technology in an organization.

The workshop addresses some of the open fundamental challenges of testing in a PLE setting. Given the improvements in productivity that PLE delivers to development, how does a test organization keep pace? To what extent can we test reusable assets and how much can this reduce the testing obligations for each product? What kinds of changes or extensions have to be made to the PL infrastructure to support testing appropriately? Can we leverage our established testing tools and procedures? What properties of a PL architecture improve the testability of reusable assets and products and how can these be enforced during architectural design? Are there PLE techniques that can provide similar efficiency gains for testing as are possible for development? Without adequate answers, testing becomes the bottleneck in PLE.

In this workshop we aim to bring together both researchers and practitioners on all aspects of PL testing, from designing for testability, through test coverage, to testing tools. We are especially interested in exchanging industrial experience in PL testing and comparing different approaches to enable an integration of different ideas. Our goal is to provide a context for such an information exchange and to provide an opportunity to discuss innovative ideas, setting a research agenda, and starting collaborations on this topic. We intend to invite experts not only from product-line engineering, but also testing experts.

Submission (extended!): The extended deadline for submissions is June 19, 2006. For more information please visit the workshop homepage at <http://www.biglever.com/split2006/>.

Workshop 5 (W5)

OSSPL - First International Workshop on Open Source Software and Product Lines

<http://www.dsi.unifi.it/osspl06/>

22 August 2006

Organizers:

[Frank van der Linden](#), Philips Medical Systems, The Netherlands

[Piergiorgio Di Giacomo](#), University of Florence, Firenze, Italy

Contact: osspl06@dsi.unifi.it

Description

Open source software is getting much attention lately. Using open source software appears to be a profitable way to obtain good software. This is also applicable for organizations doing product line engineering. On the other hand, because of the diverse use of open source software, product line development is an attractive way of working in open source communities. However, at present open source and product line development are not related. This workshop aims to get a better understanding between the two communities to get an insight how they can profit from each other.

The workshop deals with the following issues:

- Ownership, control and management of product line assets in an open source community
- Visibility of the code: when it is valuable to share proprietary code and how to take the right decision.
- Creation of different levels of architecture visibility: proprietary, among closed consortium, public. Is this possible?
- Product line requirements, roadmaps and planning in open source development
- Using the open source community to evolve components and being explicit about variability
- Variability representation and management in an open source community
- Open source for the platform and in applications
- Cohabitation of product line management and agile processes
- Open source asset management tools in product line development
- The meaning of domain and application engineering in an open source context
- Recognition and recovery of a product line in an open source asset base
- Aspects dealing with evolutionary, variability or distribution of development relating to legal risks involving: liability, warranties, patent infringements etc.

Submission (extended!): The extended deadline for submissions is June 15, 2006. For more information please visit the workshop homepage at <http://www.dsi.unifi.it/osspl06/>.

Software Product Lines Doctoral Symposium (DS)

<http://www1.isti.cnr.it/SPL-DS-2006/>

22 August 2006

Doctoral Symposium Organizers:

Isabel John - Fraunhofer IESE, Germany

Len Bass - Software Engineering Institute, USA

Giuseppe Lami - I.S.T.I./C.N.R., Italy

Contact: SPL-DS@isti.cnr.it

Call for Papers: <http://www1.isti.cnr.it/SPL-DS-2006/DS-CFP.html>

Description

The doctoral symposium provides a platform for young researchers to present their work to an international audience and discuss it with each other and with experts in the field. Experienced researchers will comment on the presented work and give feedback for further development. This event is a unique opportunity for the presenting young researchers and doctoral students to receive invaluable expert feedback, make contact with other researchers in the field, professionally present their work, and become familiar with other approaches and future research topics. The doctoral symposium addresses research activities in the field of software product lines (SPLs). Topics of interest include all aspects of the development phases, management, evaluation, reuse, and maintenance of SPLs. The peculiarity of this doctoral symposium is that it is addressed specifically to young researchers with original ideas and initiatives in the SPL field. Although it addresses mainly PhD work in progress, we also encourage the submission of other work in progress such as master's degree or diploma theses.

Important Dates:

Submission deadline	12 May 2006
Notification of acceptance	19 June 2006
Camera ready version	14 July 2006
Symposium date	22 August 2006

Panelists/Reviewers:

Birgit Geppert - Avaya Labs, USA

Andre van der Hoek - University of California, USA

Kyo Kang - POSTECH, Korea

David Weiss - Avaya Labs, USA



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Conference Tutorials

Tutorial Chair: [Daniel J. Paulish](#), Siemens Corporate Research

Calendar View

21 August 2006			
AM		PM	
T1	An Introduction to Product Line Requirements Engineering Brian Berenbach	T4	Creating Reusable Test Assets in a Software Product Line John McGregor
T2	New Methods Behind the New Generation of Software Product Lines Success Stories Charles Krueger	T5	Leveraging Model Driven Engineering in Software Product Lines Bruce Trask, Angel Roman
T3	Introduction to Software Product Lines Patrick Donohoe	T6	Introduction to Software Product Line Adoption Linda Northrop, Larry Jones
T8	Software Product Line Variability Management Klaus Pohl, Frank van der Linden, Andreas Metzger		

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

22 August 2006			
AM		PM	
T9	Domain-Specific Modeling and Code Generation for Product Lines Juha-Pekka Tolvanen	T12	Lightweight Dependency Models for Product Lines Neeraj Sangal
T10	The Scoping Game Mark Dalgarno	T13	Transforming Legacy Systems into Product Lines Danilo Beuche
T11	Using Feature Models for Product Derivation Olaf Spinczyk, Holger Papajewski	T14	Feature Modularity in Software Product Lines Don Batory
T15	Generative Software Development Krzysztof Czarnecki		

Tutorial 1 (T1)

An Introduction to Product Line Requirements Engineering

Brian Berenbach

21 August 2006, (Half Day - AM)

Requirements elicitation and management has become ever more important as product lines become more complex and time to market is shortened. Outsourcing has added a new dimension to requirements management, exacerbating problems associated with transitioning from analysis to design. This half day tutorial will provide an introduction to product line requirements engineering from the perspective of project and product management: how it impacts project managers, quality assurance personnel, requirements analysts, developers and testers. Topics covered will include product line requirements, feature modeling, CMMI compliant requirements management and requirements analysis processes (both UML and text based). Business analysts who are interested in using UML for modeling will also find the course interesting. No formal knowledge of programming is required.

Tutorial 2 (T2)

New Methods Behind the New Generation of Software Product Lines Success Stories

Charles Krueger

21 August 2006, (Half Day - AM)

A new generation of software product line success stories is being driven by a new generation of methods, tools and techniques. While early software product line case studies at the genesis of the field revealed some of the best software engineering improvement metrics seen in four decades, the latest generation of software product line success stories exhibit even greater improvements, extending benefits beyond product creation into maintenance and evolution, lowering the overall complexity of product line development, increasing the scalability of product line portfolios, and enabling organizations to make the transition to software product line practice with orders of magnitude less time, cost and effort. We explore some of the important new methods such as software mass customization sans application engineering, minimally invasive transitions, bounded product line combinatorics, and product line lifecycle management.

Tutorial 3 (T3)

Introduction to Software Product Lines

Patrick Donohoe

21 August 2006, (Half Day - AM)

Software product lines have emerged as a new software development paradigm of great importance. A software product line is a set of software intensive systems sharing a common, managed set of features, and that are developed in a disciplined fashion using a common set of core assets. Organizations developing a portfolio of products as a software product line are experiencing order-of-magnitude improvements in cost, time to market, staff productivity, and quality of the deployed products.

This tutorial will introduce the essential activities and underlying practice areas of software product line development. It will review the basic concepts of software product lines, discuss the costs and benefits of product line adoption, introduce the SEI's *Framework for Software Product Line Practice*, and describe approaches to applying the practices of the framework.

Tutorial 4 (T4)

Creating Reusable Test Assets in a Software Product Line

John McGregor

21 August 2006, (Half Day - PM)

This tutorial focuses on the test assets and test processes created by a software product line organization. The tutorial will allow participants to consider how to modify existing testing practices to take advantage of strategic reuse. The software product line approach blends organizational management, technical management and software engineering principles to efficiently and effectively produce a set of related products. The major test assets: test plans, test cases, test data, and test reports are created at multiple levels of abstraction to facilitate their reuse. A product line organization also defines a test process that differs from the test process in a traditional development organization. This tutorial will allow participants to consider how to modify existing testing practices to take advantage of strategic reuse. At the end of this tutorial you will

be able to:

- Understand the basic concepts of testing in software product line organizations.
- Understand the benefits, costs and risks of creating reusable test assets.
- Define a test process for your product line organization.
- Identify the steps necessary to initiate these activities for your organization.

Tutorial 5 (T5)

Leveraging Model Driven Engineering in Software Product Lines

Bruce Trask, Angel Roman

21 August 2006, (Half Day - PM)

Model Driven Engineering (MDE) is a new innovation in the software industry that has proven to work synergistically with Software Product Line Architectures. It can provide the tools necessary to fully harness the power of Software Product Lines. The major players in the software industry including commercial companies such as IBM, Microsoft, standards bodies including the Object Management Group, and leading universities such as the ISIS group at Vanderbilt University are fully embracing this MDE/PLA combination. IBM is spearheading the Eclipse Foundation including its MDE tools. Microsoft has launched their Software Factories foray into the MDE space. Software groups such as the ISIS group at Vanderbilt are using these MDE techniques in combination with PLAs for very complex systems. The Object Management Group is working on standardizing the various facets of MDE. The goal of this tutorial is to educate attendees on what MDE technologies are, how exactly they relate synergistically to Product Line Architectures, and how to actually apply them using an existing Eclipse implementation.

Tutorial 6 (T6)

Introduction to Software Product Line Adoption

Linda Northrop, Larry Jones

21 August 2006, (Half Day - PM)

The tremendous benefits of taking a software product line approach are well documented. Organizations have achieved significant reductions in cost and time to market and, at the same time, increased the quality of families of their software systems. However, to date, there are considerable barriers to organizational adoption of product line practices. Phased adoption is attractive as a risk reduction and fiscally viable proposition. This tutorial describes a phased, pattern-based approach to software product line adoption. A phased adoption strategy is attractive as a risk reduction and fiscally viable proposition. The tutorial begins with a discussion of software product line adoption issues and then presents the Adoption Factory pattern. The Adoption Factory pattern provides a roadmap for phased, product line adoption. The tutorial covers the Adoption Factory in detail, including focus areas, phases, subpatterns, related practice areas, outputs, and roles. Examples of product line adoption plans following the pattern are used to illustrate its utility. The tutorial also describes strategies for creating synergy within an organization between product line adoption and ongoing CMMI or other improvement initiatives.

Tutorial 8 (T8)

Software Product Line Variability Management

Klaus Pohl, Frank van der Linden, Andreas Metzger

21 August 2006, (All Day)

Tutorial participants will become familiar with the key concepts of software product line engineering and will learn how to apply variability management in practice. The participants will be able to differentiate between the two processes domain engineering and application engineering, and will have an understanding of the differences between single-system development and the development activities in product line engineering. The focus will be on requirements engineering and architectural design activities, and the relationships between them. The participants will further have learned about the concept of variability, have practiced the concepts through exercises, and will be able to model variability in requirements and design artifacts by using the orthogonal variability modeling approach (OVM).

Tutorial 9 (T9)

Domain-Specific Modeling and Code Generation for Product Lines

Juha-Pekka Tolvanen

22 August 2006, (Half Day - AM)

Current modeling languages provide surprisingly little support for automating product line development. They are either based in the code world using the semantically well-defined concepts of programming languages (e.g. UML) or based on an architectural view using a simple component-connector concept. In both cases, the languages themselves say nothing about a product family or its variants. This situation could be compared to that of a programmer being asked to write object-oriented programs where the language does not support any object-oriented concepts.

Most domain engineering approaches emphasize a language as an important mechanism to leverage and guide product development in product lines. Domain engineering results in creating a language (with related tools) for the variant specification and production that goes beyond configuring pre-built components. Previously, the effort for implementing textual or graphical languages and related tools was considerably high. This limited the use of domain engineering to a few cases only and hindered the use of true product family development methods. However, recent advances in metamodeling and related technology (e.g. metamodeling tools, Software Factory concept) as well as tools provide better support for language and generator creation. This tutorial describes how to create domain-specific languages and generators to automate product derivation. We inspect 20+ industry cases on language creation and demonstrate their use with hands-on examples. Industrial experiences of this approach show remarkable improvements in productivity (5-10 times faster variant creation) as well as capability to handle complex and large product lines (more than 100 product variants).

Tutorial 10 (T10)

The Scoping Game

Mark Dalgarno

22 August 2006, (Half Day - AM)

Product Line Scoping is the activity of determining what products constitute the product line. i.e. the *Product Line Scope*. This tutorial will introduce and explore Product Line Scoping.

By the end of the tutorial participants should:

- Understand Scoping and why it is an essential Product Line activity.
- Understand Scoping as an economic decision driven by business objectives and involving Scope trade-offs.
- Understand the sources of information which underpin Scoping.
- Be able to identify stakeholders in the Scoping activity and relate this to their own organization.
- Be aware of alternative Scoping approaches.
- Understand Scoping as an iterative, on-going activity.
- Understand Scoping's position with respect to other Product Line activities.
- Know where to look for more information.

Tutorial 11 (T11)

Using Feature Models for Product Derivation

Olaf Spinczyk, Holger Papajewski

22 August 2006, (Half Day - AM)

The implementation of a software product line leads to a high degree of variability within the software architecture. For an effective development and deployment it is necessary to resolve variation points within the architecture and source code automatically during product/variant derivation. Given the complexity of most

software systems tool support is necessary for these tasks. This tutorial shows how feature models combined with appropriate tools can provide this support. The importance of the separation of problem space modeling and solution space modeling is discussed. Concepts how to connect both spaces using constraints and/or generative approaches are shown. Furthermore, some typical patterns of variability in the solution space are shown and their automatic resolution in common languages like C/C++ and Java is demonstrated. Integration of code generators, aspect-oriented programming and software configuration management systems into the derivation process is also discussed. The tutorial is accompanied by demonstrations of the presented concepts with freely available tools.

Tutorial 12 (T12)

Lightweight Dependency Models for Product Lines

Neeraj Sangal

22 August 2006, (Half Day - PM)

This tutorial will present a practical technique for managing the architecture of software product lines using Lightweight Dependency Models. We will demonstrate that the matrix representation used by these models provides a unique view of the architecture and is highly scalable compared to the directed graph approaches that are common today. We will also show a variety of matrix algorithms and transformations that can be applied to analyze and organize the system into a form that reflects the architecture and demonstrates the importance of managing dependencies in product lines.

During the tutorial, we will illustrate our approach by applying it to real applications each consisting of hundreds or thousands of files. We will show how dependency models can be created for product lines and how formal design rules can be specified to manage the evolution of these architectures. Finally, we will use the actual dependency models to demonstrate how architecture evolves and how it often begins to degrade.

Tutorial 13 (T13)

Transforming Legacy Systems into Product Lines

Danilo Beuche

22 August 2006, (Half Day - PM)

Not every software product lines starts from the scratch, often organizations face the problem that after a while their software system is deployed in several variants and the need arises to migrate to systematic variability and variant management using a software product line approach. The tutorial will discuss issues coming up during this migration process mainly on the technical level, leaving out most of the organizational questions. The goal of the tutorial is to give attendees an initial idea how a transition into a software product line development process could be done with respect to the technical transition. The tutorial starts with a brief introduction into software product line concepts, discussing terms such as problem and solution space, feature models, versions vs. variants. Tutorial topics are how to choose adequate problem space modeling, the mining of problem space variability from existing artifacts such as requirements documents and software architecture. Also part of the discussion will be the need for separation of problem space from solution space and ways to realize it. A substantial part will be dedicated to variability detection and refactoring in the solution space of legacy systems.

Tutorial 14 (T14)

Feature Modularity in Software Product Lines

Don Batory

22 August 2006, (Half Day - PM)

Feature Oriented Programming (FOP) is a design methodology and tools for program synthesis in software product lines. Programs are specified declaratively in terms of features. FOP has been used to develop product-lines in widely varying domains, including compilers for extensible Java dialects, fire support simulators for the U.S. Army, network protocols, and program verification tools. The fundamental units of modularization in FOP are program extensions (aspects, mixins, or traits) that encapsulate the implementation of an individual feature. An FOP model of a product-line is an algebra: base programs are constants and

program extensions are functions (that add a specified feature to an input program). Program designs are expressions - compositions of functions and constants - that are amenable to optimization and analysis. This tutorial reviews core results on FOP: models and tools for synthesizing code and non-code artifacts by feature module composition, automatic algorithms for validating compositions, and the relationship between product-lines, metaprogramming, and model driven engineering (MDE).

Tutorial 15 (T15) **Generative Software Development**

Krzysztof Czarnecki

22 August 2006, (All Day)

Product-line engineering seeks to exploit the commonalities among systems from a given problem domain while managing the variabilities among them in a systematic way. In product-line engineering, new system variants can be rapidly created based on a set of reusable assets (such as a common architecture, components, models, etc.). Generative software development aims at modeling and implementing product lines in such a way that a given system can be automatically generated from a specification written in one or more textual or graphical domain-specific languages (DSLs).

In this tutorial, participants will learn how to perform domain analysis (i.e., capturing the commonalities and variabilities within a system family in a software schema using feature modeling), domain design (i.e., developing a common architecture for a system family), and implementing software generators using multiple technologies, such as template-based code generation and model transformations. Available tools for feature modeling and implementing DSLs as well as related approaches such as Software Factories and Model-Driven Architecture will be surveyed and compared. The presented concepts and methods will be demonstrated using a sample case study of an e-commerce platform.



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Conference Panels

23 August 2006

P1 [Product Derivation Approaches](#)

Panel moderator: David Weiss, Avaya Labs

Model problem: [Interactive Television Applications](#)

P2 [Testing in a Software Product Line](#)

Panel moderator: Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany

Model problem: [The eShop Product Line](#)

24 August 2006

P3 [Product Line Research](#)

Panel moderator: Liam O'Brien, Lero, The Irish Software Engineering Research Centre

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

Panel 1 (P1)

Panel on Product Derivation Approaches

23 August 2006

Panel moderator: David Weiss, Avaya Labs

Panelists:

Danilo Beuche, pure-systems

Charles Krueger, BigLever Software

Rob van Ommering, Philips Research

Juha-Pekka Tolvanen, MetaCase

Abstract

This panel looks at product derivation approaches and their differences, strengths and weaknesses in different PLE situations. Each panelist will examine a common problem (the [Interactive Television Applications](#)) and provide an overview of their product derivation approach and how it was used to solve the problem.

Overview

At some point, no matter how wonderful your product line process is, you have to ship the products. The panelists will each present a different approach to PLE, concentrating on how actual products are derived from specifications. The approaches presented include feature modeling, architecture description languages, UML and domain-specific modeling languages.

A common product specification and derivation task will be given to all panelists, and they will show how their approach works on it. The audience can - and is warmly encouraged to - participate, ask additional questions, heckle, and hopefully laugh. A major goal is to identify the classes of PLE situations that best suit each approach.

Following are some of the questions and issues to be addressed by the panel.

1. How large a portion of a product is automatically derived? Please answer in terms of some reasonably precise measure, such as percent of modules, classes, or KNCSL, or coverage in a feature model.
2. How are new features and functionality developed? Give an example, if possible.
3. What is the cost and time to create a new feature or change the application platform, e.g., in hours of effort as a fraction of effort needed to create the application engineering environment? Alternatively, how would you estimate the cost and time?

Panel 2 (P2)

Testing in a Software Product Line

23 August 2006

Panel moderator: Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany

Panelists:

Georg Grütter, Robert Bosch GmbH, Germany

John D. McGregor, Clemson University, USA

Andreas Metzger, University of Duisburg-Essen, Germany

Tim Trew, Philips Research, The Netherlands

Abstract

This panel is about system testing of software product line artifacts. The panelist will present different approaches for software product line testing. Together, we will discuss their pros and cons. As a kind of benchmark, a common example of an online store ([The eShop Product Line](#)) will be used to ease the comparison of the different testing approaches.

Overview

Each panelist will present an approach to test the domain and application artifacts in software product line engineering. The decision whether to test the domain artifacts in domain engineering or if testing is delayed to application engineering is left to the panelists.

To facilitate a better comparison of the different test approaches, each panelist will illustrate his approach using a running example of an online store product line.

The discussions will, among others, cover the following questions:

- Should there be system testing in domain engineering, or should system tests be performed during application engineering only?
- Which test artifacts can be reused during product line testing?
- Is there an advantage of creating domain test artifacts which are reused during application engineering?
- Can application test cases be generated? And if so, should they be generated from domain test cases or just from application engineering artifacts?
- Does the model-based test case derivation offer benefits when compared with deriving test cases directly from natural language requirements?

Panel 3 (P3)

Product Line Research: Lessons Learned from the last 10 years and Directions for the next 10

24 August 2006

Moderator: Liam O'Brien, Lero, The Irish Software Engineering Research Centre

Panelists:

Paul Clements, Software Engineering Institute, USA

Kyo Kang, POSTECH, Korea

Dirk Muthig, Fraunhofer IESE, Germany

Klaus Pohl, Lero, The Irish Software Engineering Research Centre & University of Duisburg-Essen, Germany

Abstract

This panel is about past and future research in software product lines. The panelists will look back at the past 10 years to examine outcomes and lessons learned and will look forward to the next 10 years and will give potential outcomes and directions for the future of software product line research. The outcomes will be examined for relevance to the practitioner community.

Overview

Each panelist will present their lessons learned from the past and the directions for the next 10 years. Several industry judges will be asked to make a determination as to how useful the outcomes have been or will be for practitioners. The panelists will have an opportunity to respond to the judges comments and this will lead to a general discussion.

The discussions will, among others, cover the following questions:

- What have been the main lessons and outcomes for research in software product lines over the past 10 years?
- What will likely be the major directions and outcomes for research in software product lines over the next 10 years?
- How relevant have the past outcomes been for practitioners and how relevant will future outcomes be for practitioners?
- What will be the next big breakthrough for software product line research?



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).



10th International Software Product Line Conference (SPLC 2006) 21-24 August 2006 Baltimore, Maryland, USA

Invitation to Corporate Sponsorship

The Tenth International Software Product Line Conference (SPLC) 2006 will be held in Baltimore, Maryland, USA on August 21–24, 2006. Software product lines represent an important and growing software development paradigm, and SPLC is a leading forum for researchers and practitioners working in the field. SPLC 2006, the tenth official gathering of the software product line community, is a result of the merging of two former conferences: the Software Product Line Conference (SPLC), which began in 2000 in the USA, and the Product Family Engineering (PFE) Conference, which began in 1996 in Europe. SPLC 2006 will provide a venue for practitioners, researchers, and educators to reflect on the achievements made during the past decade, assess the current state of the field, and identify key challenges still facing researchers and practitioners.

The conference will feature research and experience papers, topical panels, tutorials, workshops, demonstrations, birds-of-a-feather discussions, and other opportunities for members of the product line community to interact. This year's program will span a wider range of product line interests than ever before. The keynote speakers will be Carliss Baldwin of the Harvard Business School and Gregor Kiczales, originator of aspect-oriented programming. Special sessions focusing on the research agendas of leading research organizations, testing and quality assurance in product lines and several other topics will be conducted in addition to the usual conference sessions. SPLC 2006 will also support a Doctoral Symposium in which the next generation of researchers will receive guidance and support.

We invite you to become a corporate sponsor of SPLC. There are three levels of sponsorship, described below. Your sponsorship, in addition to helping with the general expenses of the conference, will assist aspiring research students to attend the conference.

Your support will help SPLC be an effective venue for sharing and learning. I will be happy to answer any questions you may have.

Thanks for your consideration,

John D. McGregor
Conference Chair
johnmc@cs.clemson.edu

Conference Information

- [SPLC 2006 Home](#)
- [Keynote Speakers](#)
- [Technical Program](#)
- [Tutorials](#)
- [Workshops](#)
- [Panels](#)
- [Software Product Lines](#)
- [Doctoral Symposium](#)
- [Software Product Line Hall of Fame](#)
- [Birds-of-a-Feather](#)
- [Important Dates](#)
- [Corporate Supporters](#)
- [Conference & Program Committees](#)
- [Location/Hotel](#)
- [Past Conferences](#)
- [Contact Information](#)

	Gold Level (\$5,000)	Silver Level (\$2,500)	Bronze Level (\$1,000)
Call for participation:	Logo	No	No
WWW page:	Mentioned + logo	Mentioned + logo	Logo
Conference program:	One page advert + logo	Logo	Mentioned
Conference proceedings:	Logo	Mentioned	Not mentioned
Other adverts:	Yes	Yes	Yes
Email distribution mentioned:	Yes	Yes	Yes
Adverts in conference bag:	2 full pages of presentation of the company/product	1 full page of presentation of the company/product	No
Opening and closing session:	Explicitly mentioned	Explicitly mentioned	Not explicitly mentioned
Banners:	If provided, one Banner at Conference Side	No	No

Posters:	One-two A0 Poster in plenary conference room one A1 Poster in other conference rooms	One A1 Poster in main conference room	One Poster with all Supporters in main conference room
Free attendance:	Three free conference registrations (does not include tutorials or workshops)	Two free conference registrations (does not include tutorials or workshops)	No free registration



Contact Information:

For general information, contact [John D. McGregor](#).

For web site information, contact [Bob Krut](#).