Software Architecture, Components and Reuse



Jan Bosch

Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

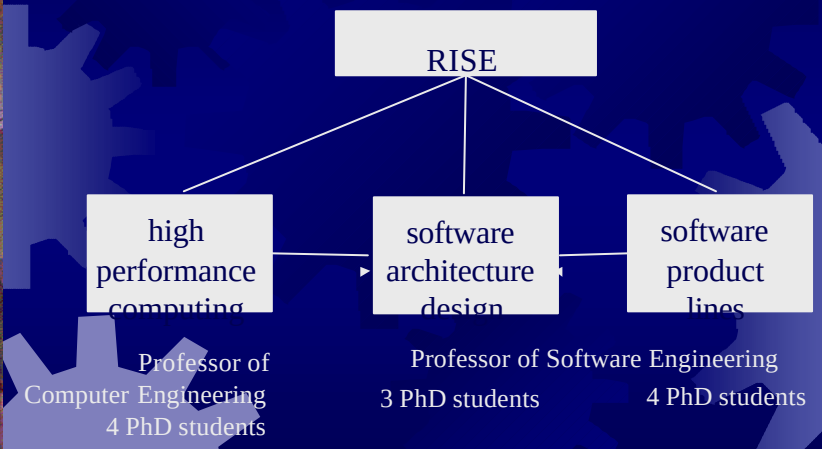
Background

- Head of Software Engineering group at University of Groningen - earlier head of RISE research group @ Blekinge Institute of Technology ©
- NOSAR - Nordic Network on Software Architecture Research NOSA'98 & NOSA'99 - Nordic Workshop on Software Architecture, NOSA 2001
- ECOOP workshops:
  - WCOP'97, '98, '99, '00, '01 - Component-Oriented Programming
  - OOSA'98 - Object-Oriented Software Architectures
  - OPLA'99 - Product-Line Architectures
- ICT-Architecture'99, Dutch National Architecture Congress 2000
- WICSA 2001, Amsterdam, August
- 3rd Int. WS on Dev. and Evo of SAs for Product Families
- EUREKA/ITEA ESAPS project, 300 man-years, 99-01

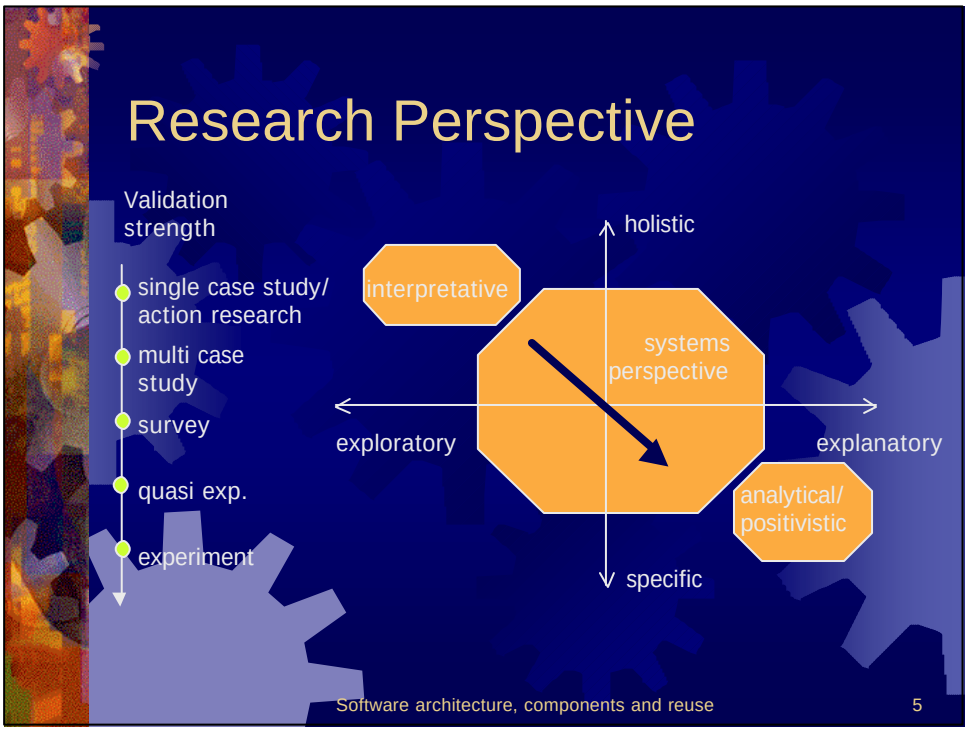
Industrial Partners

- Ericsson Software Technology
- Althin Medical
- Axis Communications
- Symbian (earlier Ericsson Mobile)
- EC-Gruppen
- Securitas Larm
- Ericsson Software Architecture Research Lab
- CombiTech Systems
- etc.

Research In Software Engineering

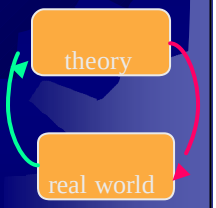


Research Perspective

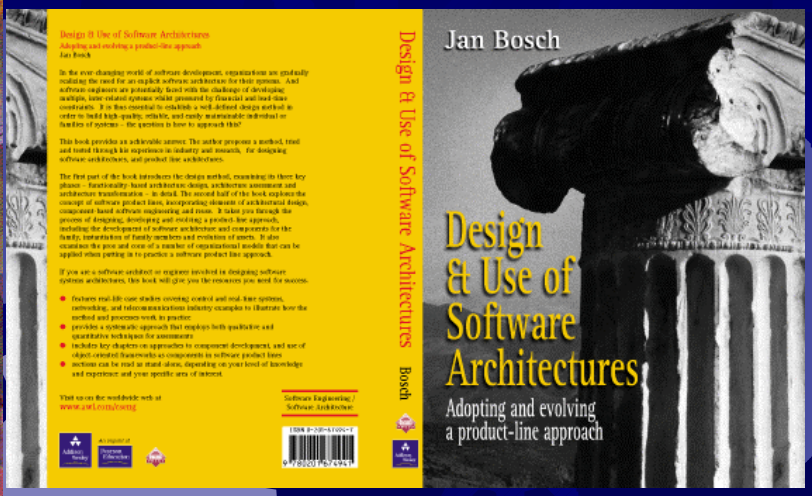


Research Methods

- analytical methods
  - formal models, theories, proofs, etc.
- empirical methods
  - experiment (quantitative)
  - survey (quantitative)
  - case study (qualitative or quantitative)
  - action research (qualitative)



Book!



Overview

- role of software
- software reuse
- software components
- software architecture
- software product lines
- conclusion

Role of software

- software constantly increasing in size and complexity
- software makes up *competitive edge*
- examples
  - Securitas Larm AB - fire alarm systems
  - business process reengineering and corporate information systems

Traditional approach to SD

- one application/product at a time
- focus on delivery and deadlines
- maintainability & evolution not in focus
- typical problems
  - time & budget
  - system quality
  - maintenance cost up to 80% of total product
  - decreasing competitiveness

Software Reuse

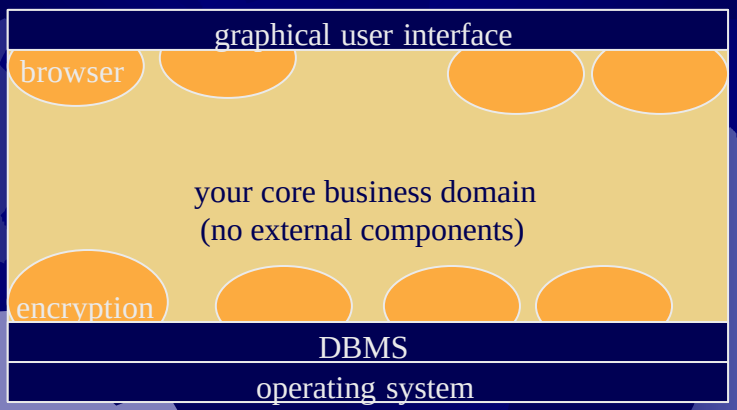
- drastically decrease cost of software development and maintenance
- increase quality of software
- reuse of existing software is one of the most promising approaches
- construct applications by composing reusable software pieces

(cost ⇙ TTM ⇙ staff)

Examples of reuse

- code “scavenging” (copy & paste)
- operating systems
- compilers
- class libraries
- design patterns
- distribution systems, e.g., CORBA, COM, etc.
- graphical user interface (GUI) systems
- design reuse (expertise)
- object-oriented frameworks
- database management systems (DBMS)
- application generators

Internal vs. External



External Component Lifecycle

- part of application
- identified as domain by visionaries
- prototype development (universities, research labs)
- productification (often by small (new) companies)
- adoption by the “pragmatists”
- incorporation in “infrastructure”, e.g. operating system

Two approaches to reuse

- **opportunistic**: the software engineer reuses pieces of software that fit the current problem and adds them to the software.
- **planned**: the organization puts explicit effort in developing reusable artifacts that provide the 'right' abstractions, 'right' level of variability and that fit into an higher level structure.

opportunistic reuse does not work in practice

Two approaches to reuse

- **bottom-up**: reusable components, once developed or mined, are added to a, possibly large, collection of assets. Software engineers search their way through these assets looking for suitable pieces.
- **top-down**: assets are developed and presented as parts fitting into a higher-level structure. Assets adhere to predefined provided and required interfaces.

bottom-up reuse does not work in practice

Software Components

Research

- Reusable assets are black-box components
- Assets have narrow interface through a single point of access.

Industry

- Assets are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks
- The asset interface is provided through entities, e.g., classes in the asset. These interface entities have no explicit differences to non-interface entities.

Software Components

Research (cont.)

- Assets have few and explicitly defined variation points that are configured during instantiation
- Assets implement standardized interfaces and can be traded on component markets
- Focus is on asset functionality and on the formal verification of functionality

Industry (cont.)

- Variation is implemented through configuration and specialization or replacement of entities in the asset.
- Assets are primarily developed internally. Externally developed assets go through considerable (source code) adaptation to match the product-line architecture requirements
- Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance

Software Architecture

- software architecture

  The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them. [Bass et al. 98]

- quality requirements

  Quality requirements can be categorised into development QRs, e.g. maintainability, and operational QRs, e.g. performance and reliability

Software Architecture and QAs

- the architecture of a software system constrains the quality attributes:
  - performance
  - maintainability
  - reliability
  - ...
- therefore, the driving quality attributes influence the software architecture (more than the functional requirements)

Why software architecture?

first class representation of SA during

- design
  - stakeholder-based assessment
  - assessment of quality attributes
- implementation
  - configuration management
  - software product lines
- run-time
  - dynamic software architectures, post-fielding upgrades, dynamic reorganization, etc.

Views on Software Architecture

Research

- architecture explicitly defined
- architecture consists of components and connectors
- ADLs used to describe architectures and generate applications

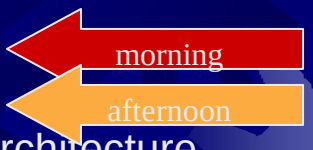
Industry

- mostly conceptual understanding of architecture, little explicit description
- no explicit first-class connectors (ad-hoc solutions for run-time binding and glue code)
- programming (C++) and script languages (Make) used to describe configuration of applications

Software Architecture Use

Three types of SA use:

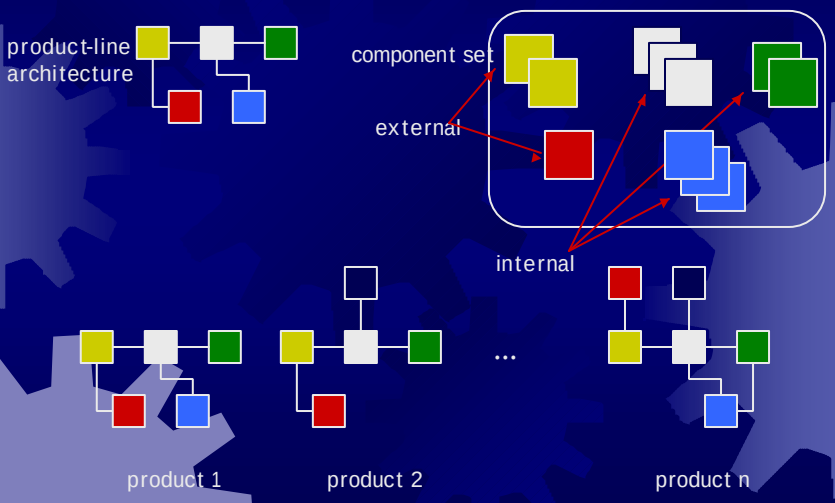
- single system
- software product line
- standardized domain architecture component framework [szyperski 98]



Software Product Lines

- A software product line consists of:
  - product line architecture
  - set of reusable components
  - set of products, where each product has
    - product architecture derived from PLA
    - instantiated and configured components
    - product-specific code

Software Product Lines



Software Product Lines

- variability versus commonality
  - domain analysis
  - product requirements

  only implement required and expected variability
- converting to reuse-based software development is a fundamental change that has technical, process, organizational and business issues

Software Project Lines

- context: consultancy company or IT department performing projects with partially overlapping requirements
- define common architecture and common components/subsystems
- develop (slightly) more general components in normal projects
- increase the variability and generality of the component in subsequent projects
- **balance investment and risk!**

Conclusion

- introduction to
  - software architecture
  - software components
  - software product lines (reuse)
- component-based software engineering requires a top-down (architectural) approach
- software product lines provide a framework for components

Software Architecture Design Processes

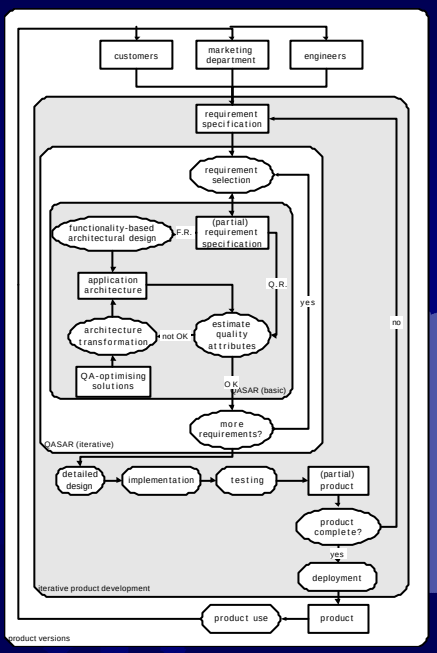


Jan Bosch

Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

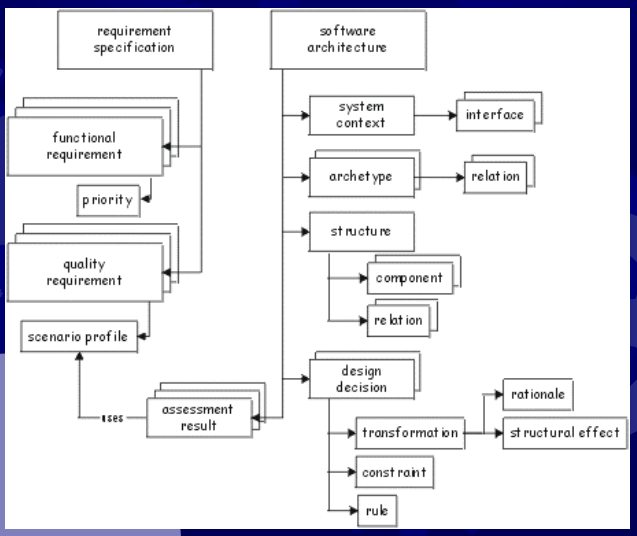
SAD Processes



SAD Processes

- four iterative processes
  - product versions
  - iterative product development
  - iterative architecture design
  - iterative architecture assessment and transformation

Artefacts



Artefacts

- requirement specification
  - functional requirements
    - required, preferred, optional
  - quality requirements
    - quality level
    - scenario profile

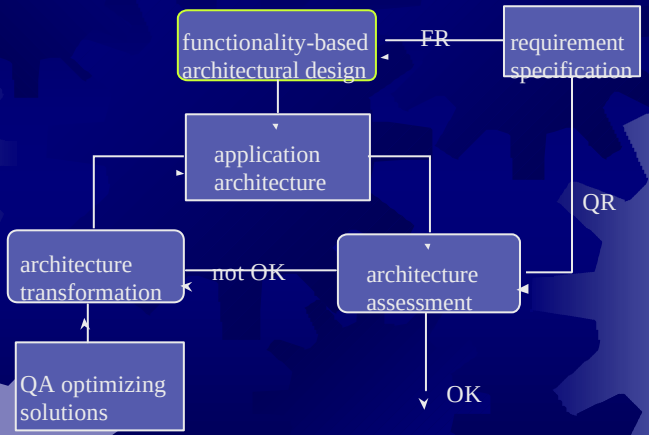
Artefacts

- software architecture
  - system context
    - interfaces
  - archetypes
    - relations
  - structure
    - components and connectors
  - design decisions
    - transformation, result, rule

Architecture Design

- based on industrial experiences:
  - fire-alarm systems
  - measurement systems
  - dialysis systems
- primary focus on QAs
- slightly idealized from concrete cases

Overview ARCS/ADM



Method

three main phases:

- functionality-based design
- assessment of QAs
- architecture transformation

Functionality-based design

four main steps:

- determine context of the system (boundaries and interfaces)
- identification of archetypes
- decompose architecture into components
- describe system instances

Architecture Assessment

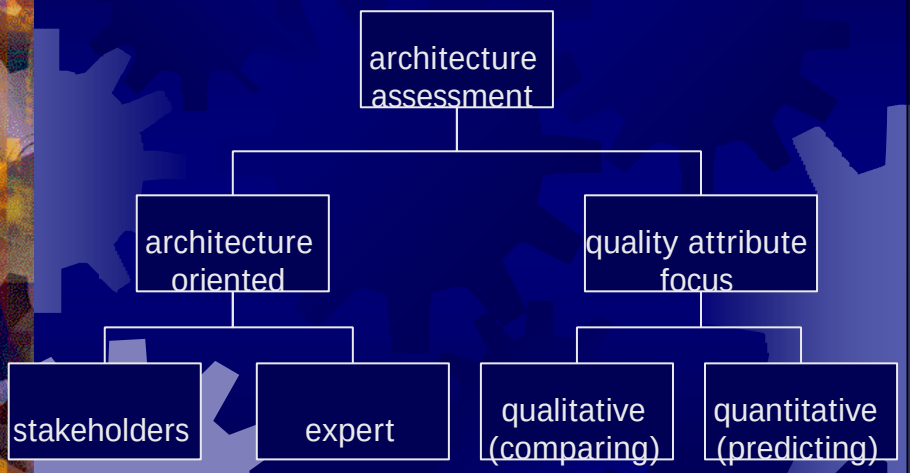
two approaches:

- after each design iteration
- as a ‘toll-gate’ before starting next phase

goals for assessment:

- quality attribute satisfaction
- stakeholder satisfaction
- support for software product line
- software system acquisition

Architecture Assessment

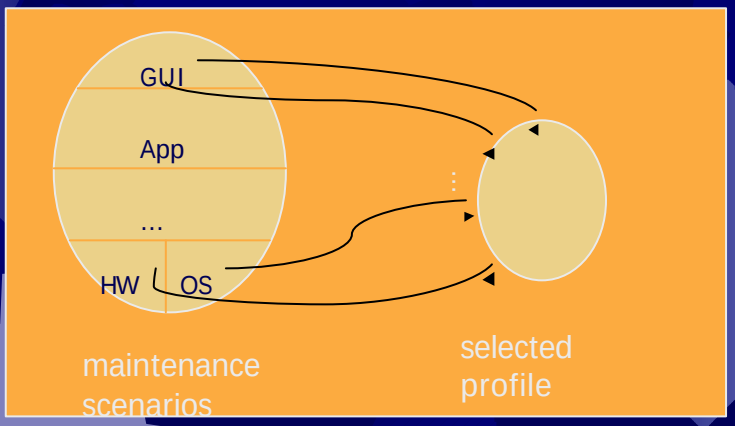


Assessing Quality Attributes

- assessment goals:
  - relative assessment
  - absolute assessment
  - assessment of theoretical maximum
- scenario profiles
- 3 + 1 assessment techniques

Scenario Profiles

- absolute versus selected profiles



Scenario Profiles

- top-down or bottom-up
- top-down profile development
  - (pre-)define scenario categories
  - selection and definition of scenarios for each category
  - each scenario is assigned a weight (either based on historical data or estimated)

Scenario Profile Development

- bottom-up profile development
  - interview stakeholders
  - categorize scenarios
  - assign weights to scenarios
  - iterate until sufficient coverage
- stopping criterion
  - coverage

Scenario Profiles - QAs

- performance: usage profile
- maintainability: maintenance profile
- reliability: usage profile
- safety: hazard profile
- security: authorization profile

Assessing Quality Attributes

- estimation techniques
  - scenario-based evaluation
  - simulation
  - mathematical modeling/metrics
  - experience-based reasoning

Architecture Transformation

- transformations (primarily) change the structure of the architecture
- affect quality attributes, but not functionality
- architecture transformation process:
  - identify unfulfilled quality attributes
  - for each quality attribute
    - identify locations where QA is inhibited
    - select the most appropriate transformation
    - perform the transformation on the architecture

Architecture Transformation

- transformation techniques
  - impose architectural style
  - impose architectural pattern
  - apply design pattern
  - convert QR to functionality

Related Work

- architecture design methods: Krutchen, Shlaer & Mellor
- architecture evaluation: Kazman
- QA-oriented communities
- Boehm - system development with NFRs
- program transformation

Conclusion

- architecture design method
- explicit treatment of QRs
- iterative process: evaluation and transformation
- experiences from 3 framework/architecture designs

Case Studies

Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

three main cases

- fire-alarm systems
- measurement systems
- dialysis systems

Fire alarm systems

domain description

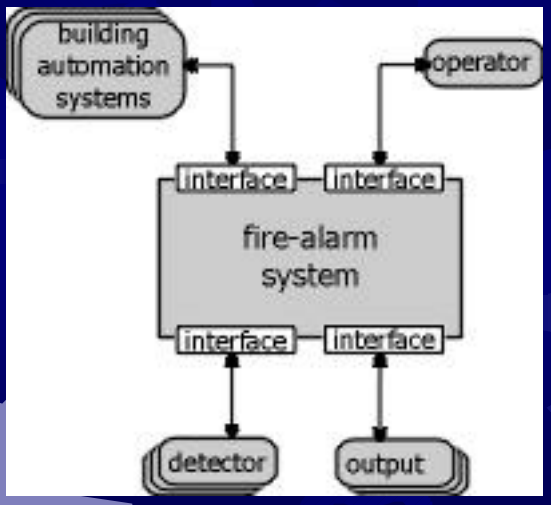
- detectors: many instances and types
- output devices: large variability
- system variability
- distributed

Fire alarm systems

quality attributes

- configurability
- demonstrability
- performance
- maintainability
  - detector technology
  - extinguishers
  - compatibility
  - etc.

Fire alarm systems

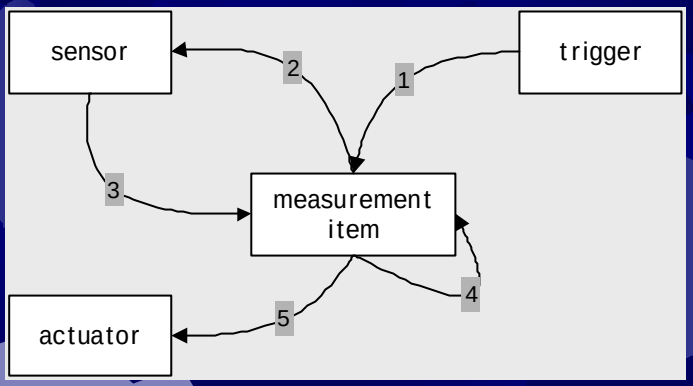


Measurement systems

domain description

- complex sensors and actuators
- strict timing requirements
- process
  - trigger
  - data collection
  - analysis
  - actuation

Measurement systems



Measurement systems

quality requirement

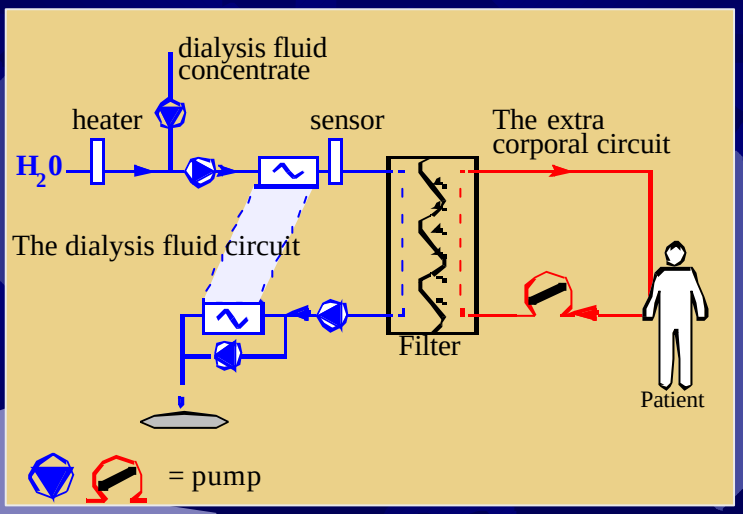
- configurability
- run-time flexibility
- real-time constraints

Haemodialysis systems

domain description

- extra corporal circuit (blood part)
- dialysis fluid circuit
- membrane
- unexpected variability

Haemodialysis systems



Haemodialysis systems

quality requirements

- maintainability
- configurability
- safety
- real-time
- demonstrability

Conclusion

- cases in embedded systems domain
- several untraditional requirements
  - configurability
  - maintainability
  - etc.

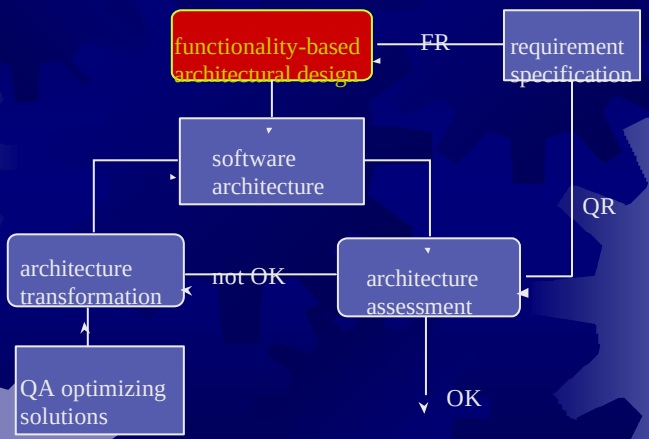
Functionality-based Architectural Design



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview ARCS/ADM



Functionality-based Design

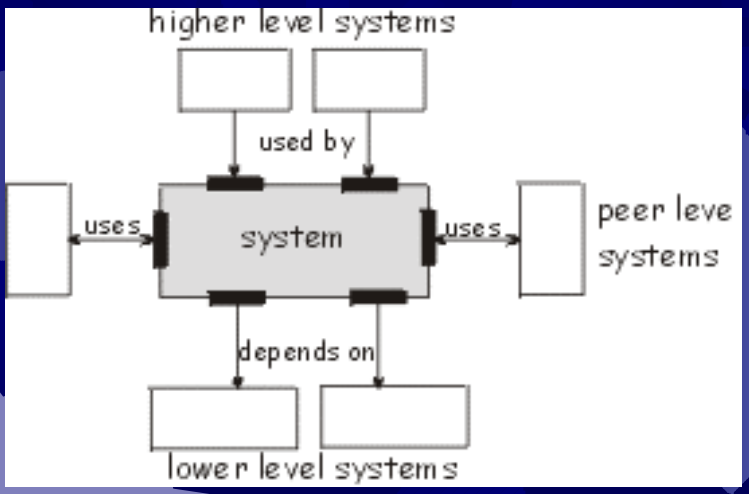
four main steps:

- determine context of the system (boundaries and interfaces)
- identify archetypes
- decompose architecture into components
- describe system instances

System Context

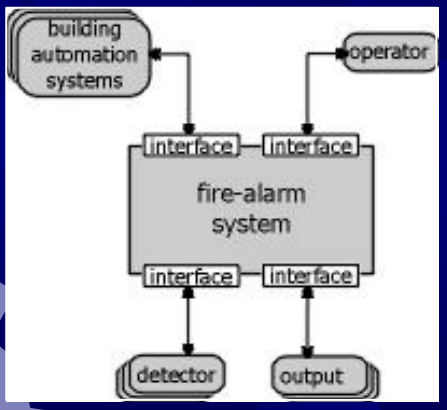
- define system interfaces to other entities
- associate requirements with each interface
  - functional requirements
  - quality requirements
- in case of PLA, determine variability at the interfaces

System Context

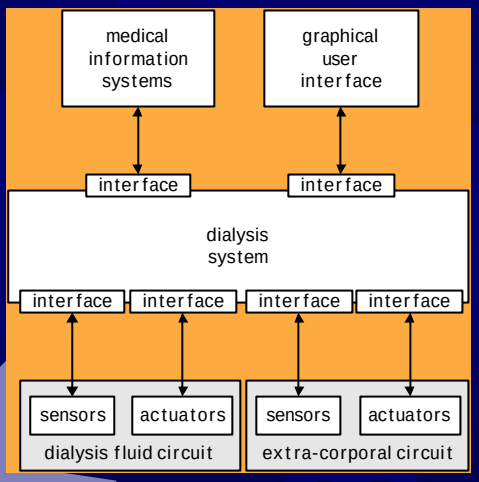


System Context

- example: fire-alarm system



System Context



Identify Archetypes

- identification of archetype candidates
- selection of small and stable set
- identification and selection of relations between archetypes

experience: ‘right’ abstractions often not ‘obvious’ domain entities

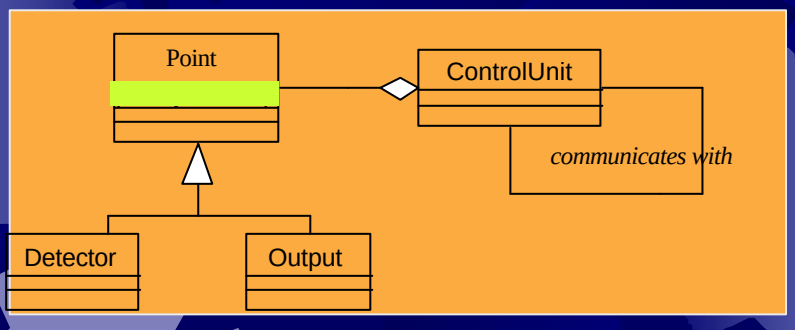
- example:
  - fire-alarm systems: points and deviations
  - dialysis systems: devices and controllers

Identify Archetypes

- recursive up and down abstraction hierarchy
- search for patterns that appear throughout the system
- different from domain analysis
- related archetypes may form system-specific patterns
- stopping criterion

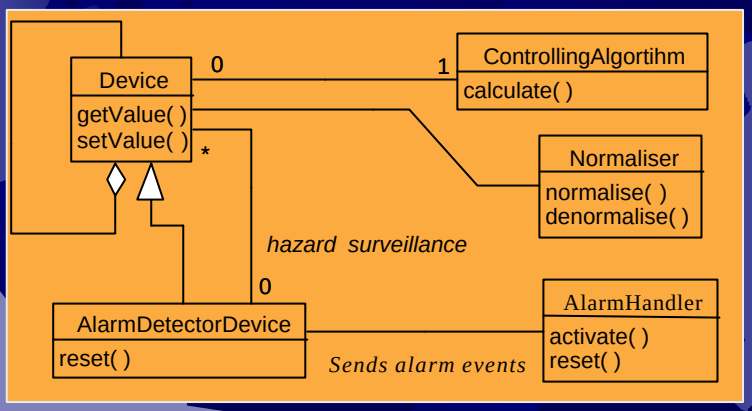
Identify Archetypes

- example: fire-alarm system



Identify Archetypes

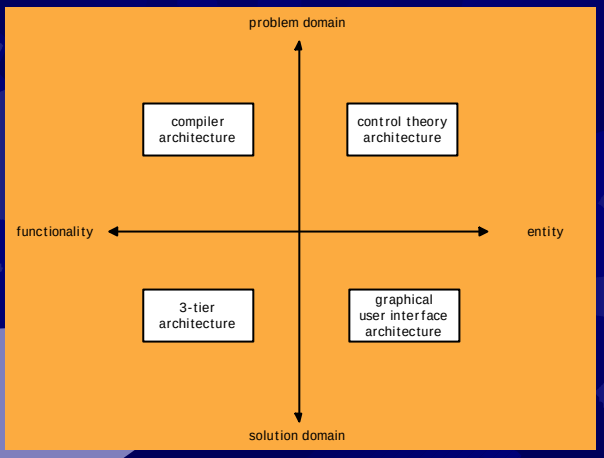
- example: dialysis system



Architecture to Components

- identify architectural components
  - interfaces
  - domains
  - abstraction layers
  - domain entities
  - archetype instantiation

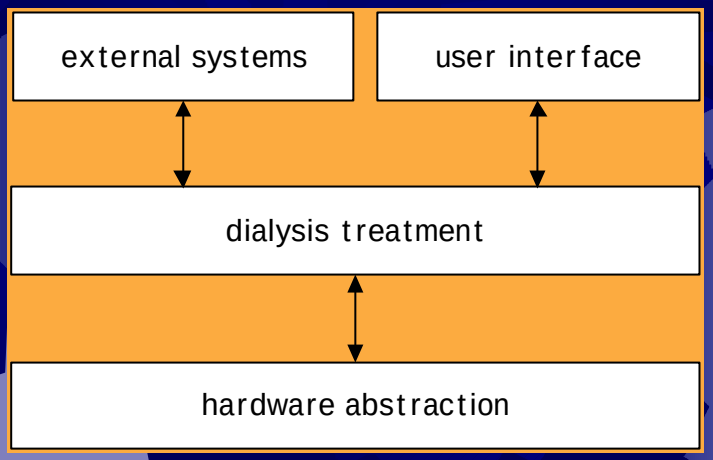
Architecture to Components



Architecture to Components

- decomposition: functional vs. entity
- domain: problem vs. solution
- component identification is recursive
- identification of component relations
  - domain model
  - script usage scenarios
  - keep relations application domain-specific

Architecture to Components

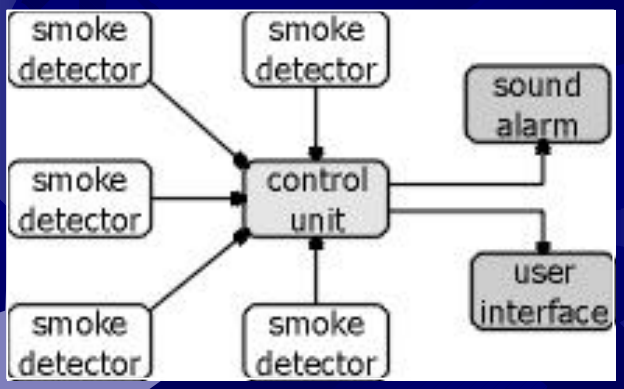


Describe System Instantiations

- recursively decompose system into subsystems
- populate subsystems (one or more archetype instances)
- instantiate relations between archetype instances and verify logical correctness
- in case of PLAs, verify variability by describing multiple system instantiations

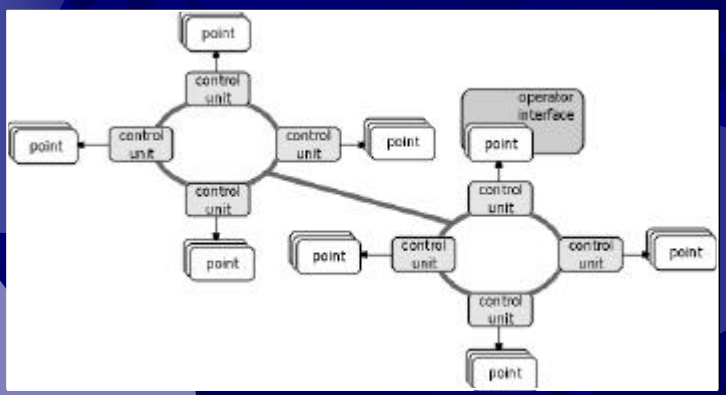
Describe System Instantiations

- example: fire-alarm system



Describe System Instantiations

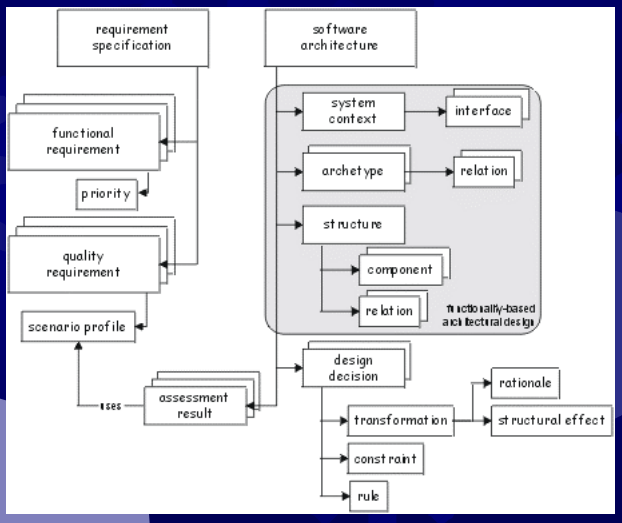
- example: fire-alarm system (2)



Conclusion

- system context
- identify archetypes
- decompose architecture into components
- describe system instances

Conclusion



Software Architecture Assessment



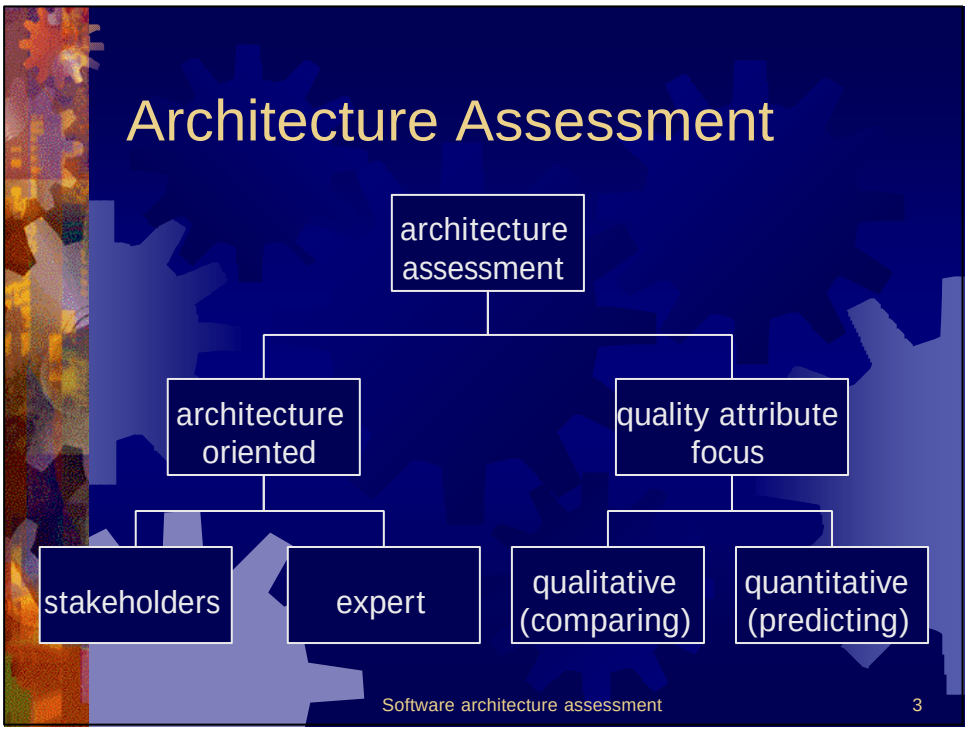
Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Architecture Assessment

two approaches:
- after each design iteration
- as a 'toll-gate' before starting next phase

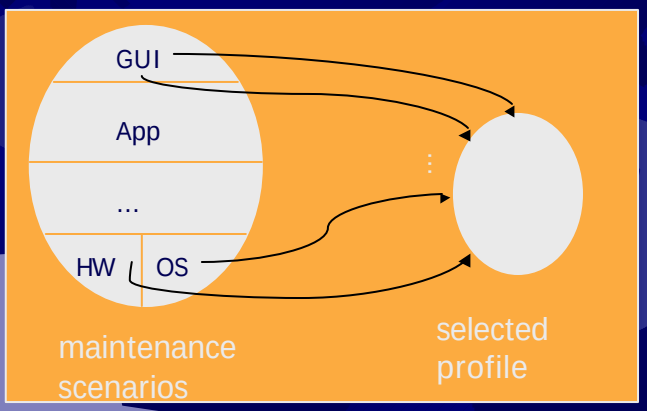
goals for assessment:
- quality attribute satisfaction
- stakeholder satisfaction
- support for software product line
- software system acquisition


Architecture Assessment
architecture assessment
architecture oriented
quality attribute focus
stakeholders
expert
qualitative (comparing)
quantitative (predicting)
Software architecture assessment
3


Assessing Quality Attributes
assessment goals:
relative assessment
absolute assessment
assessment of theoretical maximum
scenario profiles
3 + 1 assessment techniques
Software architecture assessment
4

Scenario Profiles

- absolute versus selected profiles



Software architecture assessment 5

Scenario Profiles

- top-down or bottom-up
- top-down profile development
  - (pre-)define scenario categories
  - selection and definition of scenarios for each category
  - each scenario is assigned a weight (either based on historical data or estimated)

Software architecture assessment 6

Scenario Profile Development

- bottom-up profile development
  - interview stakeholders
  - categorize scenarios
  - assign weights to scenarios
  - iterate until sufficient coverage
- stopping criterion
  - coverage

Software architecture assessment 7

Scenario Profiles - QAs

- performance: usage profile
- maintainability: maintenance profile
- reliability: usage profile
- safety: hazard profile
- security: authorization profile

Software architecture assessment 8

Dialysis System Maintenance Profile

Category	Scenario Description (Weight)
Market Driven	Change measurement units from Celsius to Fahrenheit for temperature in a treatment. (0.043)
Hardware	Add second concentrate pump and conductivity sensor. (0.043)
Hardware	Replace blood pumps using revolutions per minute with pumps using actual flow rate (ml/s). (0.087)
Hardware	Replace duty-cycle controlled heater with digitally interfaced heater using percent of full effect. (0.174)
Safety	Add alarm for reversed flow through membrane. (0.087)
Medical Adv.	Modify treatment from linear weight loss curve over time to inverse logarithmic. (0.217)
Medical Adv.	Change alarm from fixed flow limits to follow treatment. (0.087)
Medical Adv.	Add sensor and alarm for patient blood pressure (0.087)
Com.and I/O	Add function for uploading treatment data to patient's digital journal. (0.043)
Algorithms	Change controlling algorithm for concentration of dialysis fluid from PI to PID. (0.132)

Software architecture assessment 9

Assessing Quality Attributes

- estimation techniques
  - scenario-based evaluation
  - simulation
  - mathematical modeling/metrics
  - experience-based reasoning

Software architecture assessment 10

Scenarios

- 2 sets of scenarios: design & evaluation
- profile per QR (exceptions)
- primarily for ‘development QRs’
- example: dialysis system
  - change scenario profile
  - hazard scenario profile

Scenarios - Process

- develop a profile
- ‘script’ the scenarios for the architecture
- impact analysis: collect and interpret the results
- quality attribute prediction: state a conclusion
- state a list of architecture problems (possibilities for improvement)

Example: Maintainability

RS={r_1, …, r_p} SA={C, R}

C={c_1, …, c_q} where c_i=(I, cs, rt)

R={r_1, …, r_r} where r_i=(c_{source}, c_{dest}, type)

MP={cs_1, …, cs_s} where cs_i is set of new/changed requirements

IA={ia_1, …, ia_u} where ia_i=(CC_i,NP_i,NC_i, R_i)

(changed components, new plug-ins, new components)

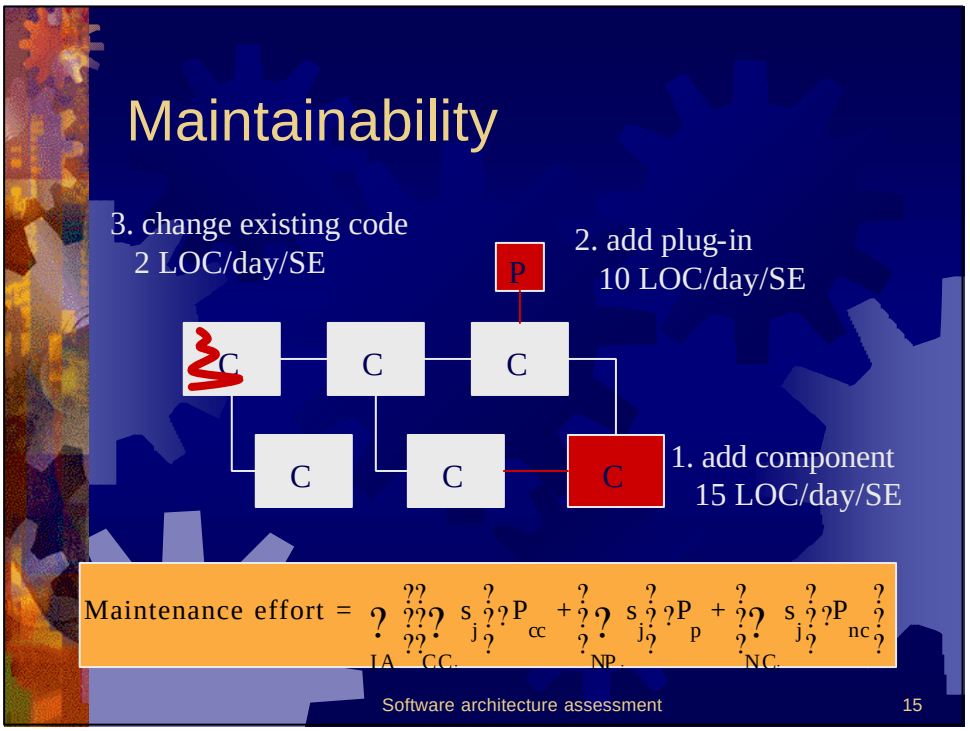
Maint. eff.= (avg. LOC/CR * #CR/yr) / LOC/day/SE

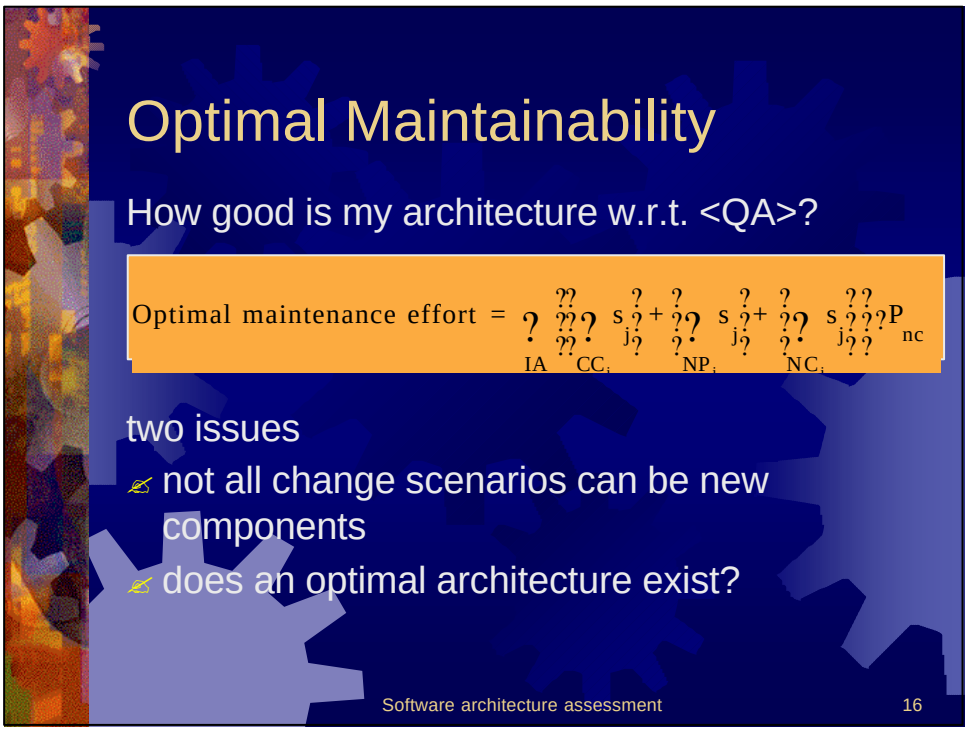
Software architecture assessment 13

Scenarios - Example

Scenario	Affected Components	Volume
C1	HDFTreatment (20% change) + new Normaliser type component	0.2*200+20 = 60
C2	ConcentrationDevice (20% change) + ConcCtrl (50% change) + reuse with 10% modification of AcetatPump and ConductivitySensor	0.2*100+0.5*175+0.1*100+0.1 *100 = 127,5
C3	HaemoDialysisMachine (10% change) + new AlarmHandler + new AlarmDevice	0.1*500+200+100 =350
C4	Fluidheater (10% change), remove DutyCycleControl and replace with reused SetCtrl	0.1*100= 10
C5	HDFTreatment (50% change)	0.5*200= 100
C6	AlarmDetectorDevice (50% change) + HDFTreatment (20% change) + HaemoDialysisMachine (20% change)	0.5*100+0.2*200+0.2*500= 190
C7	see C3	= 350
C8	new ControllingAlgorithm + new Normaliser	100+20= 120
C9	HDFTreatment (20% changes) + HaemoDialysisMachines (50% changes)	0.2*200+0.5*500= 290
C10	Replacement with new ControllingAlgorithm	= 100

Software architecture assessment 14


Maintainability
3. change existing code
2 LOC/day/SE
2. add plug-in
10 LOC/day/SE
P
C
C
C
C
C
C
1. add component
15 LOC/day/SE
Maintenance effort = ? ?? ?? ? s j ? ? P cc + ? ? ? s j ? ? P p + ? ? ? s j ? ? P nc ? ?
IA CC i NP i NC i
Software architecture assessment
15


Optimal Maintainability
How good is my architecture w.r.t. <QA>?
Optimal maintenance effort = ? ?? ?? ? s j ? ? + ? ? ? s j ? ? + ? ? ? s j ? ? ? P nc
IA CC i NP i NC i
two issues
not all change scenarios can be new components
does an optimal architecture exist?
Software architecture assessment
16

Optimal Maintainability

approach:

- small change maps to plug-in
- notion of interacting scenarios

$$\text{Opt. maint. effort} = \sum_{IA} \sum_{CC_i} \left(\sum_{NP_i} size_{ij} + \sum_{NC_i} size_{ij} \right) \begin{cases} P_{nc} & \text{if independent \& large scenario} \\ P_{p} & \text{if independent \& small scenario} \\ P_{cc} \cdot ratio + P_{nc} \cdot (1 - ratio) & \text{if interacting \& large scenario} \\ P_{cc} \cdot ratio + P_{p} \cdot (1 - ratio) & \text{if interacting \& small scenario} \end{cases}$$

Software architecture assessment 17

Worst-Case Maintainability

- map all change scenarios to changing existing code

$$\text{Worst-case Maintainability Effort} = \sum_{I} \sum_{CC_i} \left(\sum_{NP_i} size_{ij} + \sum_{NC_i} size_{ij} \right) \cdot P_{cc}$$

Software architecture assessment 18

Maintainability

- boundaries provide input to architecture design process



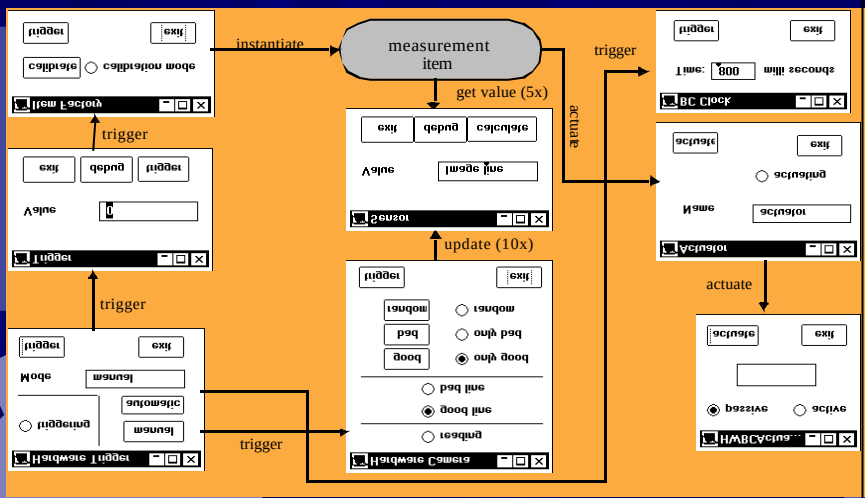
Simulation

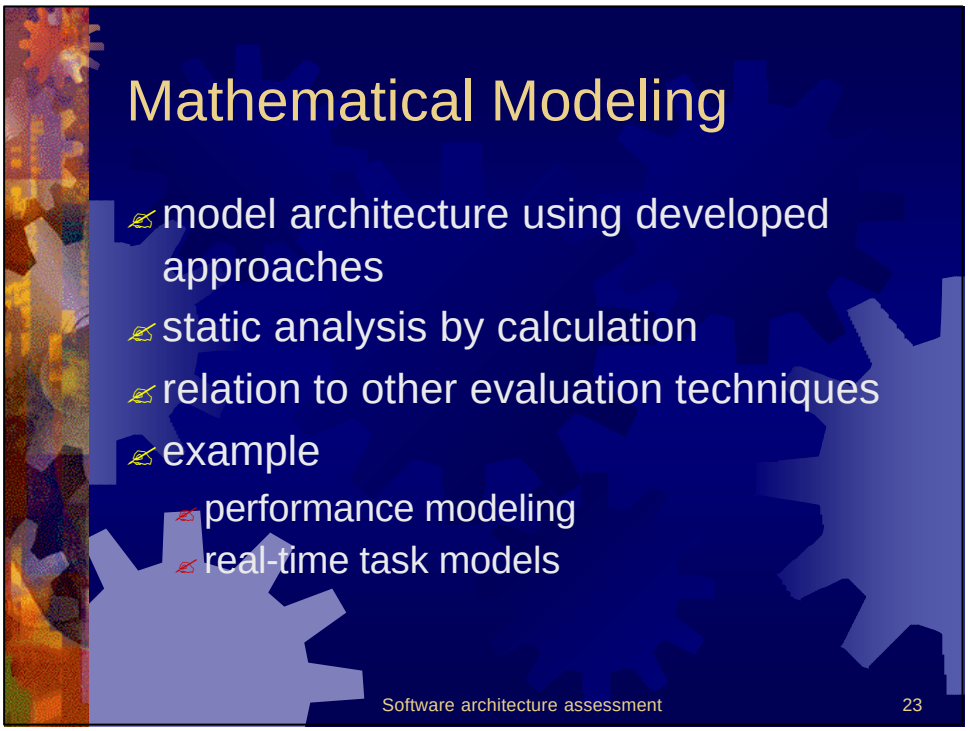
- prototype architecture implementation
- abstract simulated context
- evaluation through scenario execution
- example
  - correctness
  - performance
  - reliability & robustness

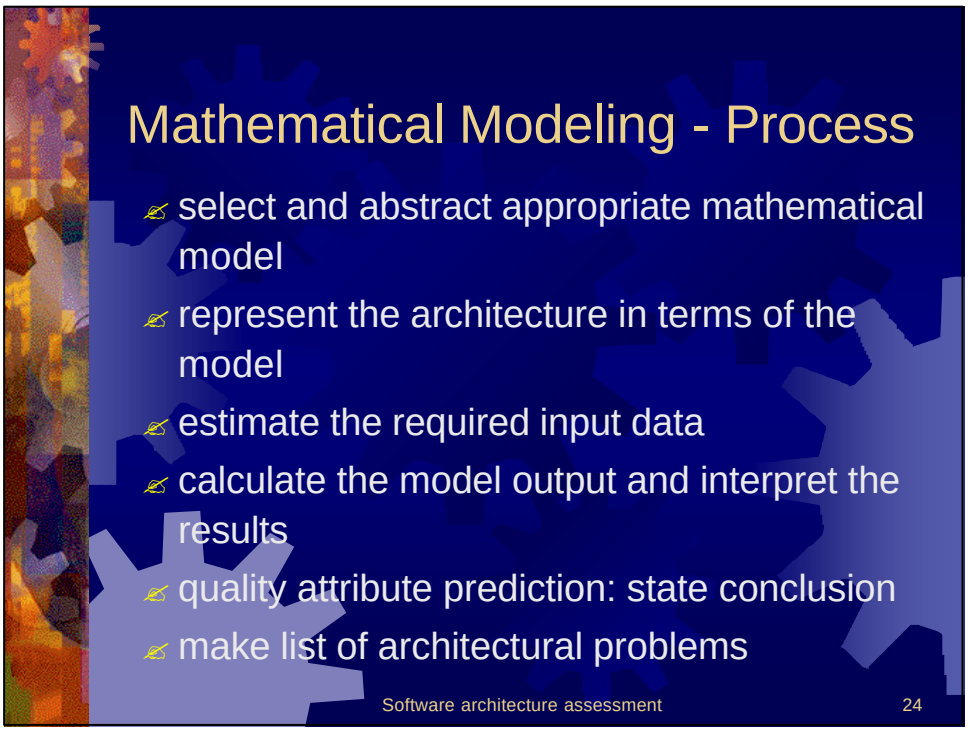
Simulation - Process

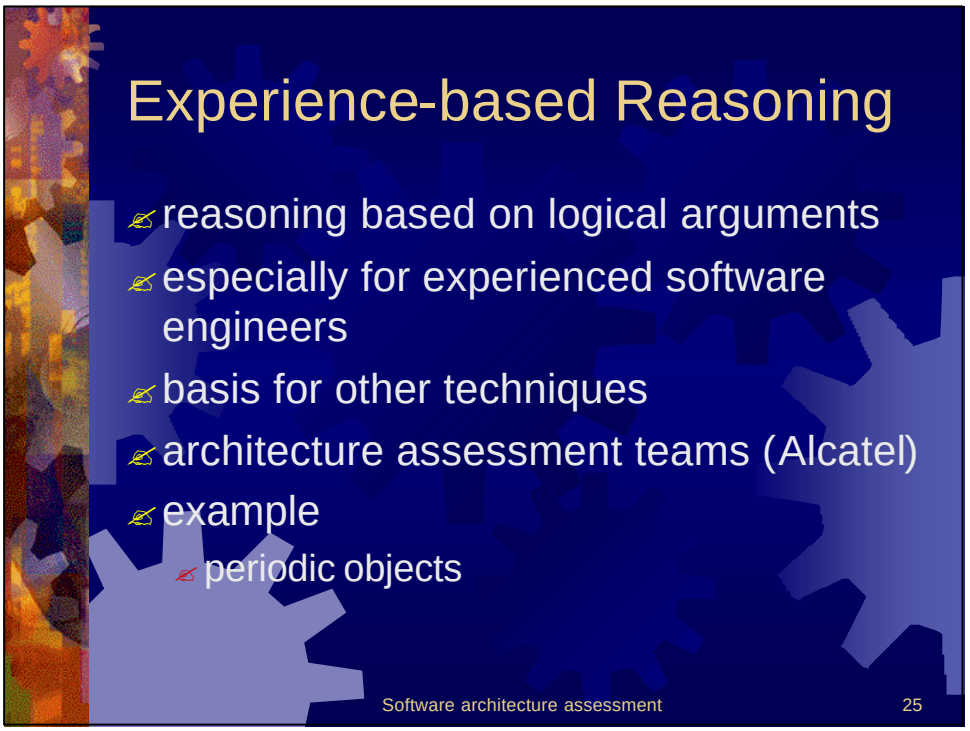
- define and implement abstract system context
- implement architecture and abstract components
- implement the profile(s)
- simulate system and initiate scenarios
- collect results and predict quality attributes
- identify functionality mismatches

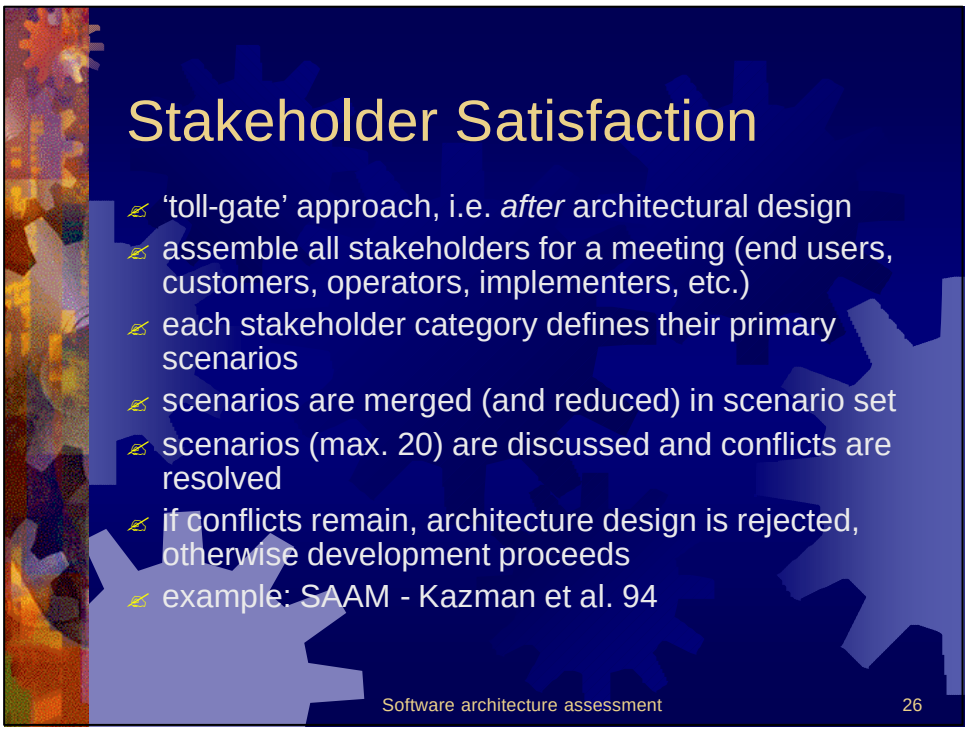
Simulation - Example




Mathematical Modeling
model architecture using developed approaches
static analysis by calculation
relation to other evaluation techniques
example
performance modeling
real-time task models
Software architecture assessment
23


Mathematical Modeling - Process
select and abstract appropriate mathematical model
represent the architecture in terms of the model
estimate the required input data
calculate the model output and interpret the results
quality attribute prediction: state conclusion
make list of architectural problems
Software architecture assessment
24


Experience-based Reasoning
reasoning based on logical arguments
especially for experienced software engineers
basis for other techniques
architecture assessment teams (Alcatel)
example
periodic objects
Software architecture assessment
25


Stakeholder Satisfaction
'toll-gate' approach, i.e. *after* architectural design
assemble all stakeholders for a meeting (end users, customers, operators, implementers, etc.)
each stakeholder category defines their primary scenarios
scenarios are merged (and reduced) in scenario set
scenarios (max. 20) are discussed and conflicts are resolved
if conflicts remain, architecture design is rejected, otherwise development proceeds
example: SAAM - Kazman et al. 94
Software architecture assessment
26


Software Product Lines
goal: determine ability of architecture to support all products in family
assessment approaches:
assess for reference context
assess for each family member
assess most important systems
assess low- and high-end systems
assess for future family members as well
Software architecture assessment
27


Software System Acquisition
context: organisation selecting a software system among alternatives
software architecture indicates several properties about the system that can be evaluated
supports selection process against relatively low cost
Software architecture assessment
28

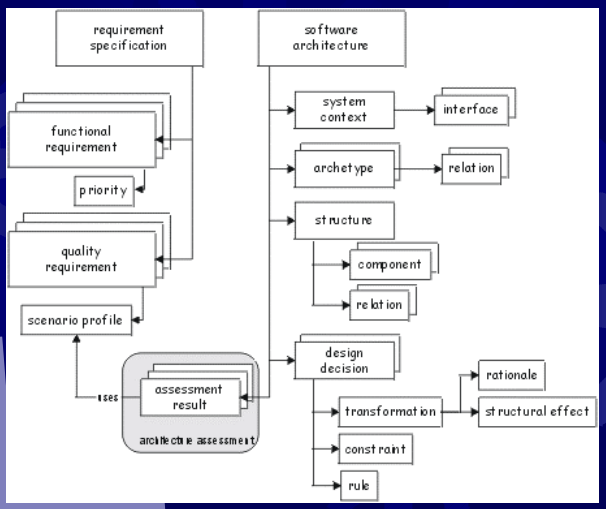
Related Work

- architecture design methods: Krutchen, Shlaer & Mellor
- architecture evaluation: Kazman
- QA-oriented communities
- Boehm - system development with NFRs
- program transformation

Conclusion

- Software architecture assessment
  - quality attributes
  - stakeholders
  - software product line
- 3+1 assessment techniques
  - scenarios
  - simulations
  - metrics/mathematical modeling
  - experience-based assessment (reviews)

Conclusion



Software architecture assessment 31

Some References

- J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach, Pearson Education (Addison-Wesley & ACM Press), ISBN 0-201-67494-7, May 2000.
- PerOlof Bengtsson and Jan Bosch, An Experiment on Creating Scenario Profiles for Software Change, *Annals of Software Engineering*,Vol. 9, pp. 59-78, May 2000.
- PO Bengtsson, J. Bosch, “Architecture Level Prediction of Software Maintenance”, *Proceedings of Third European Conference on Software Maintenance and Reengineering*, pp. 139-147, March 1999.
- Jan Bosch, PO Bengtsson, Assessing Optimal Software Architecture Maintainability, *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering* (CSMR 2001), April 2001.

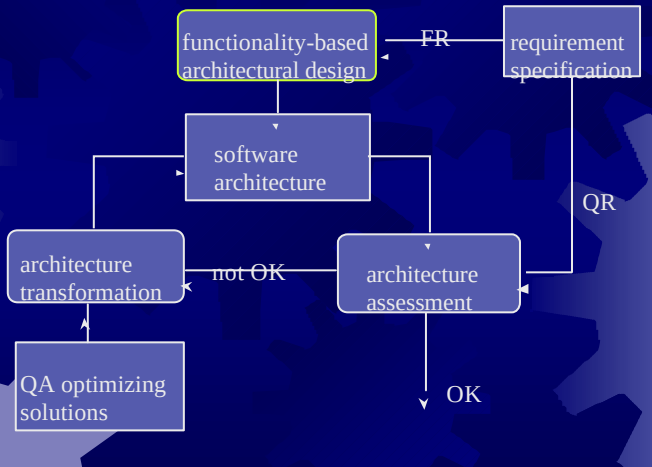
Software architecture assessment 32

Software Architecture Transformation

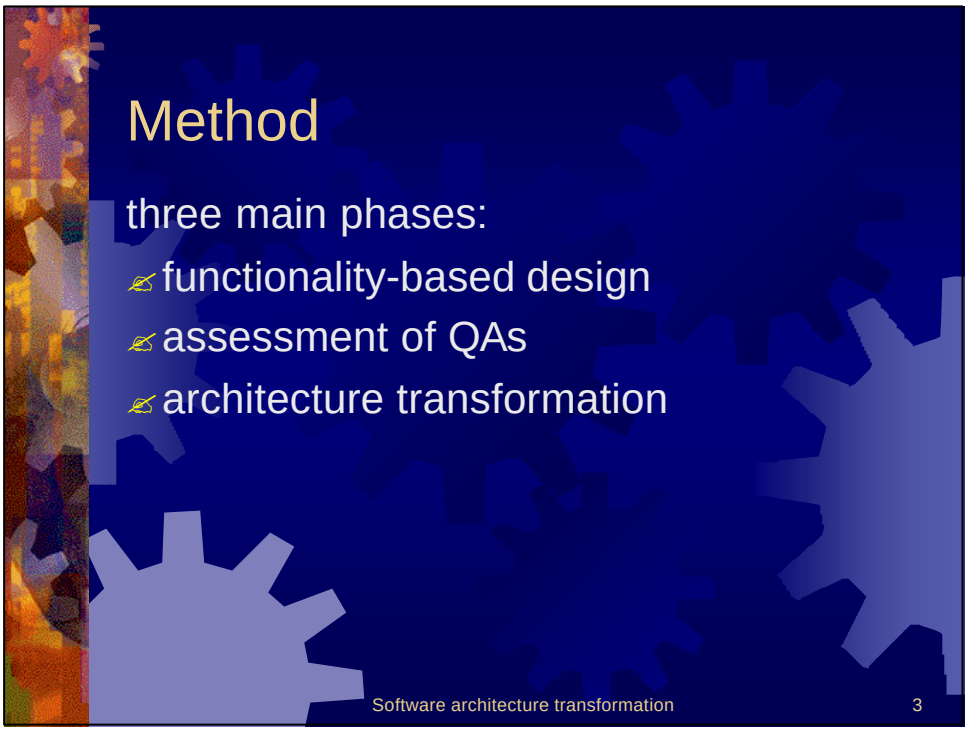
Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

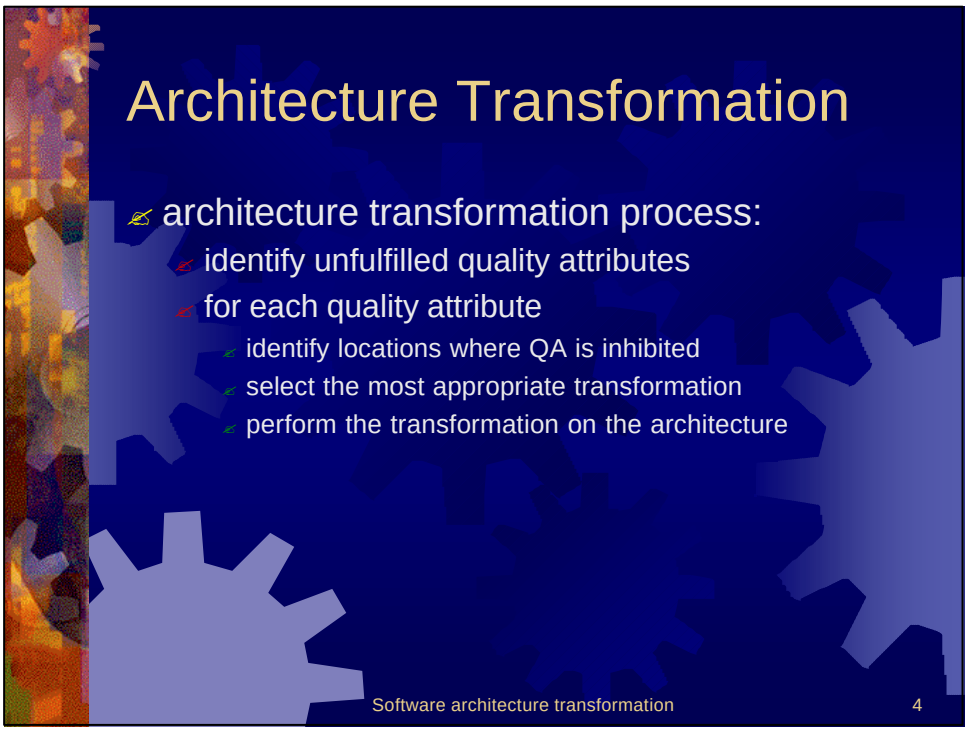
Copyright © 2001 Jan Bosch

Overview ARCS/ADM



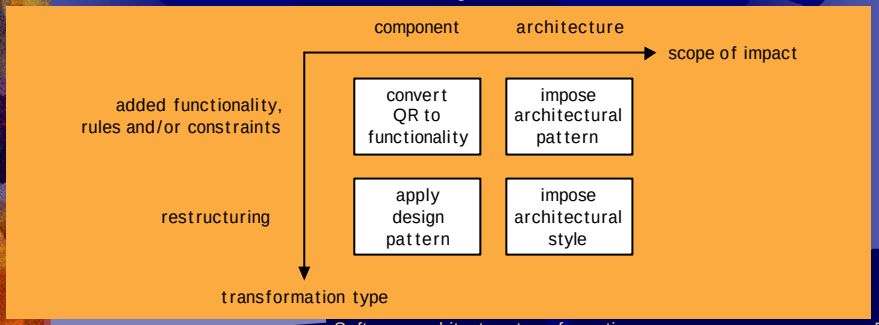
Software architecture transformation 2


Method
three main phases:
functionality-based design
assessment of QAs
architecture transformation
Software architecture transformation
3


Architecture Transformation
architecture transformation process:
identify unfulfilled quality attributes
for each quality attribute
identify locations where QA is inhibited
select the most appropriate transformation
perform the transformation on the architecture
Software architecture transformation
4

Architecture Transformation

- transformations (primarily) change the structure of the architecture
- affect quality attributes, but not application domain functionality

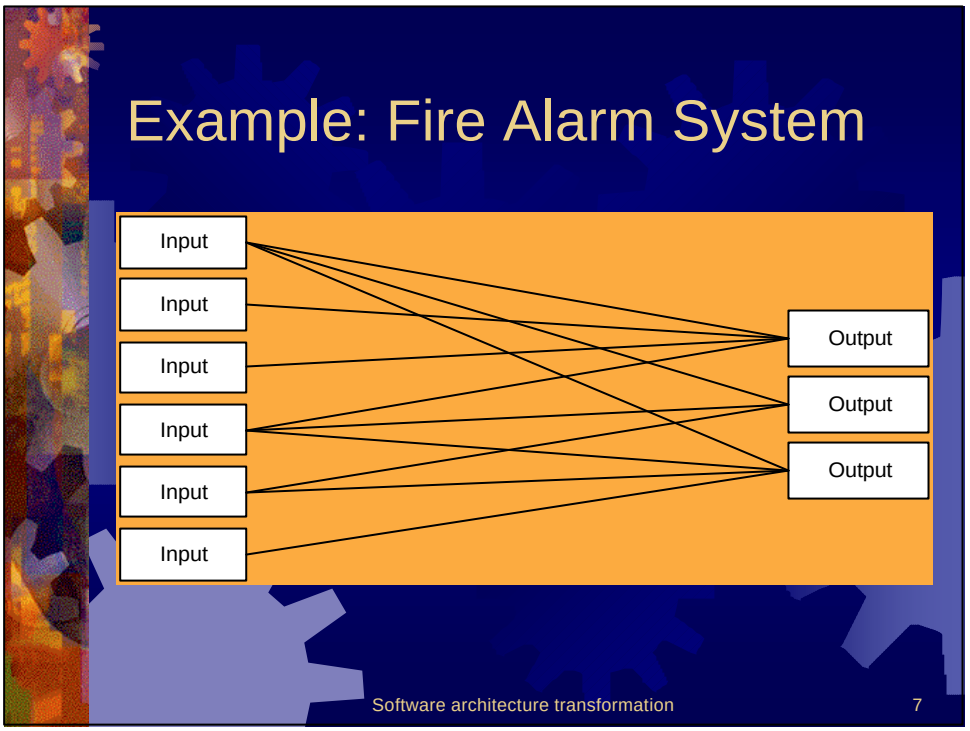


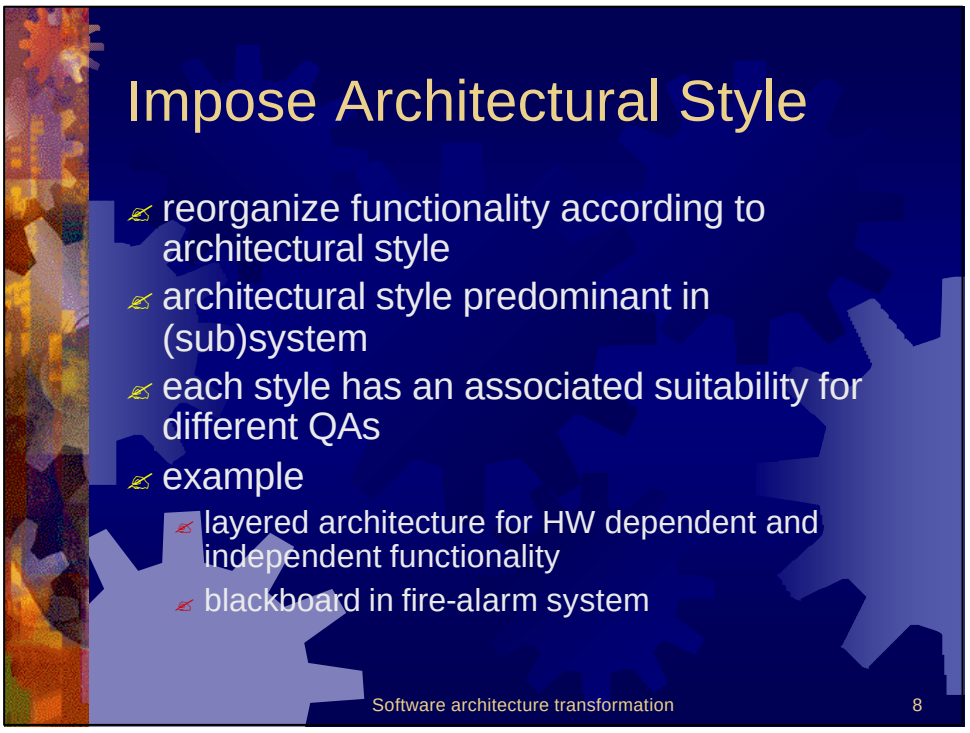
Software architecture transformation 5

Architecture Transformation

- transformation techniques
  - impose architectural style
  - impose architectural pattern
  - apply design pattern
  - convert QR to functionality

Software architecture transformation 6


Example: Fire Alarm System
Input
Input
Input
Input
Input
Input
Output
Output
Output
Software architecture transformation
7

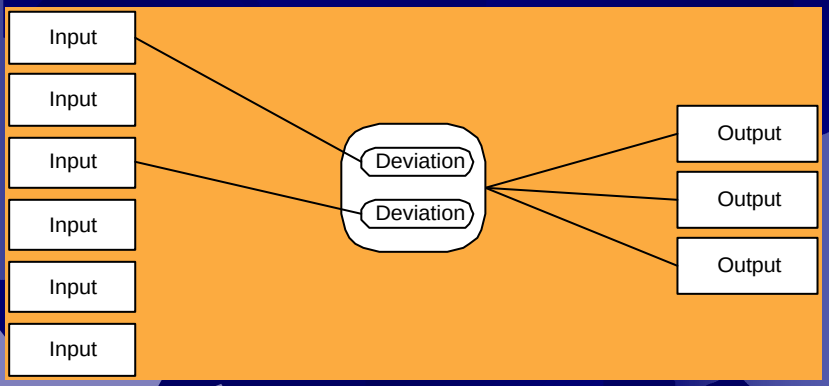

Impose Architectural Style
reorganize functionality according to architectural style
architectural style predominant in (sub)system
each style has an associated suitability for different QAs
example
layered architecture for HW dependent and independent functionality
blackboard in fire-alarm system
Software architecture transformation
8

Impose Architectural Style

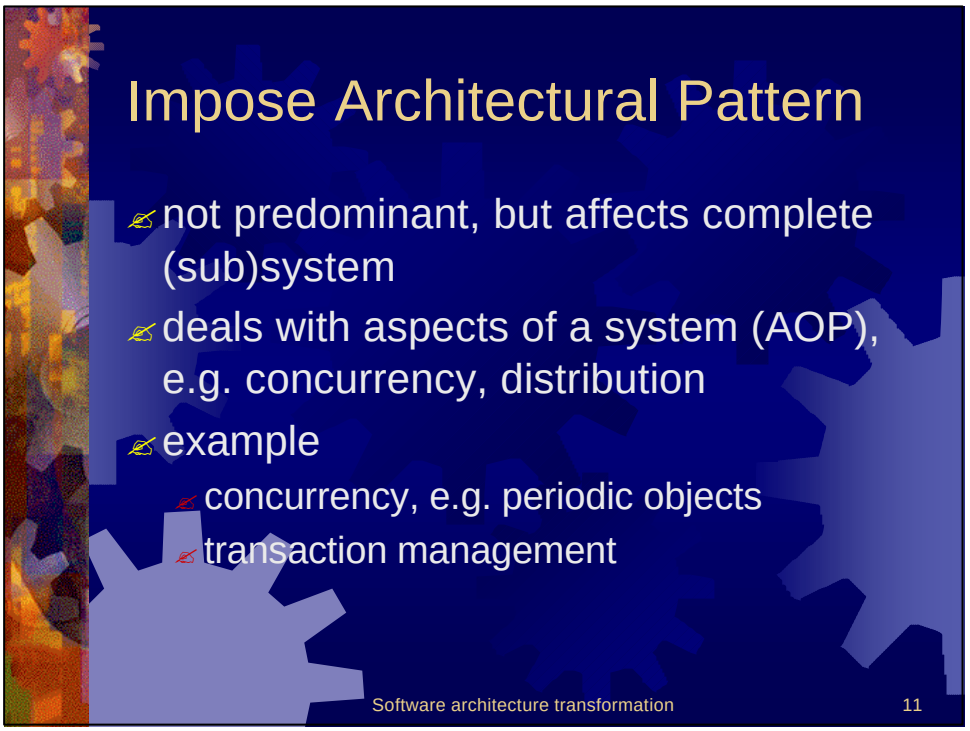
- pipes and filters
- layers
- blackboard
- object-orientation
- implicit invocation
- performance
- maintainability
- reliability
- safety
- security

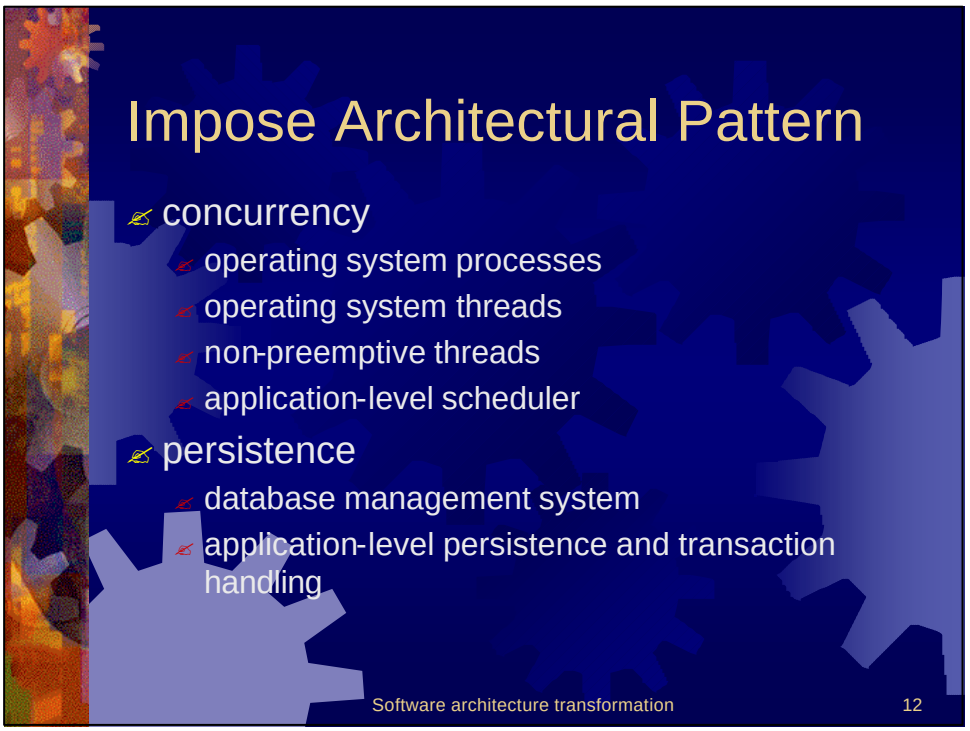
Software architecture transformation 9

Example



Software architecture transformation 10


Impose Architectural Pattern
not predominant, but affects complete (sub)system
deals with aspects of a system (AOP), e.g. concurrency, distribution
example
concurrency, e.g. periodic objects
transaction management
Software architecture transformation
11

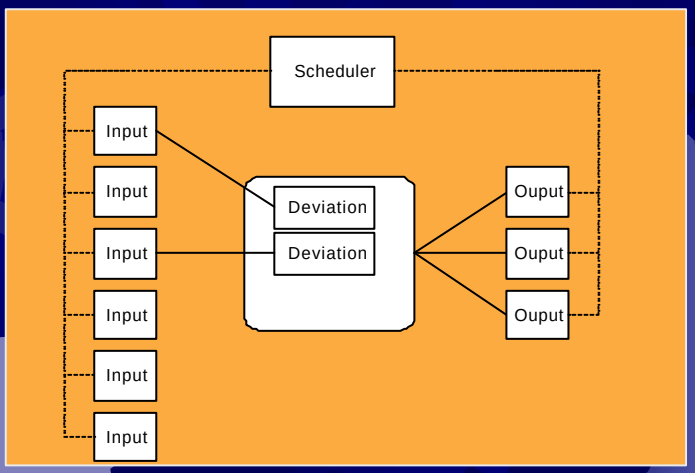

Impose Architectural Pattern
concurrency
operating system processes
operating system threads
non-preemptive threads
application-level scheduler
persistence
database management system
application-level persistence and transaction handling
Software architecture transformation
12

Impose Architectural Pattern

- distribution
  - brokers
  - remote method invocation
- graphical user interface
  - model-view-controller
  - presentation-abstraction-control

Software architecture transformation 13

Example



Software architecture transformation 14

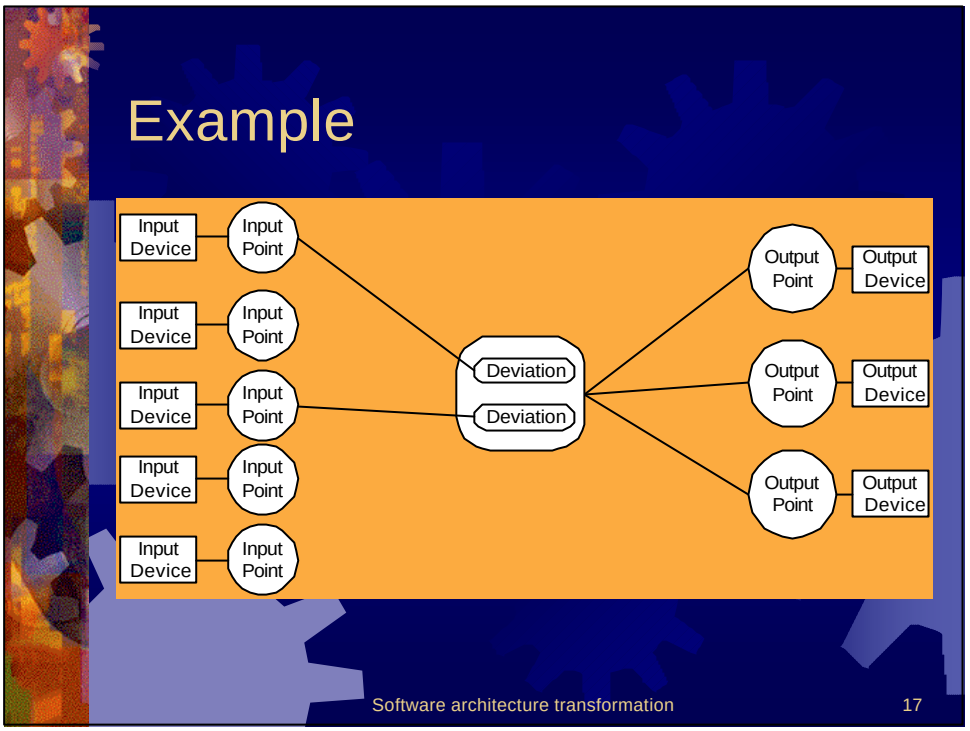
Apply Design Pattern

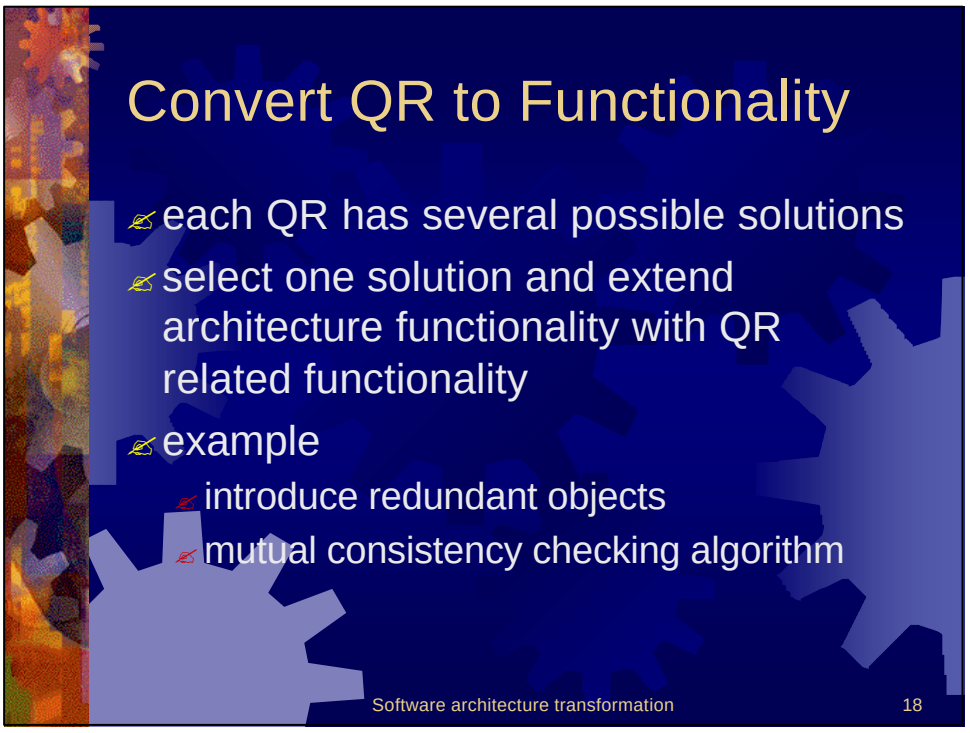
- affects a limited number of classes
- traditionally focussed on development QAs, e.g. flexibility and reusability (now extending)
- example
  - strategy pattern (flexibility ++, performance --)
  - abstract factory (extensibility++, performance-)

Apply Design Pattern

- Façade
- Abstract factory
- Strategy
- Observer

- performance
- maintainability
- reliability
- safety
- security


Example
Input Device
Input Point
Input Device
Input Point
Input Device
Input Point
Input Device
Input Point
Input Device
Input Point
Deviation
Deviation
Output Point
Output Device
Output Point
Output Device
Output Point
Output Device
Software architecture transformation
17

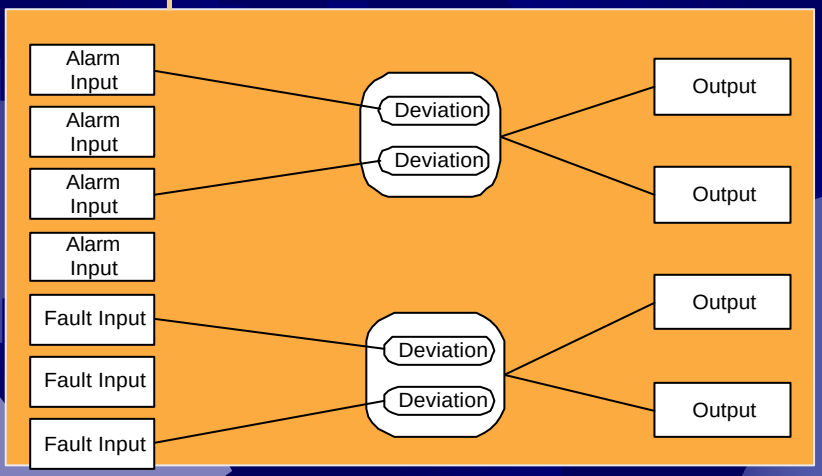

Convert QR to Functionality
each QR has several possible solutions
select one solution and extend architecture functionality with QR related functionality
example
introduce redundant objects
mutual consistency checking algorithm
Software architecture transformation
18

Convert QR to Functionality

- self-monitoring
  - top-level monitoring layer
  - entity-based monitoring
- redundancy
  - N-version programming
  - recovery blocks
  - acceptance tests

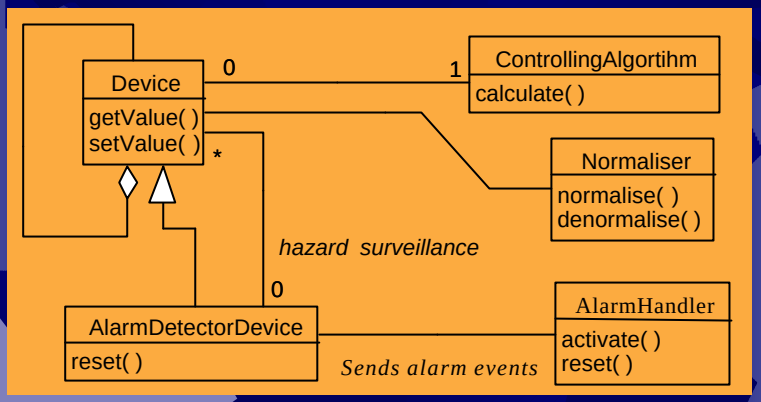
Software architecture transformation 19

Example



Software architecture transformation 20

Example

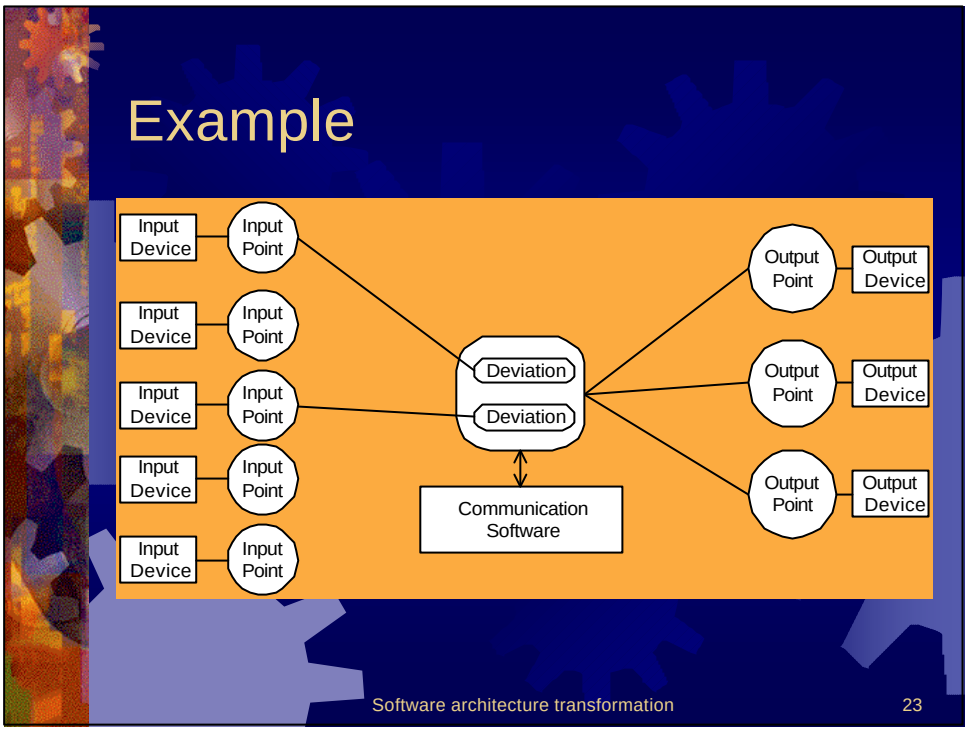


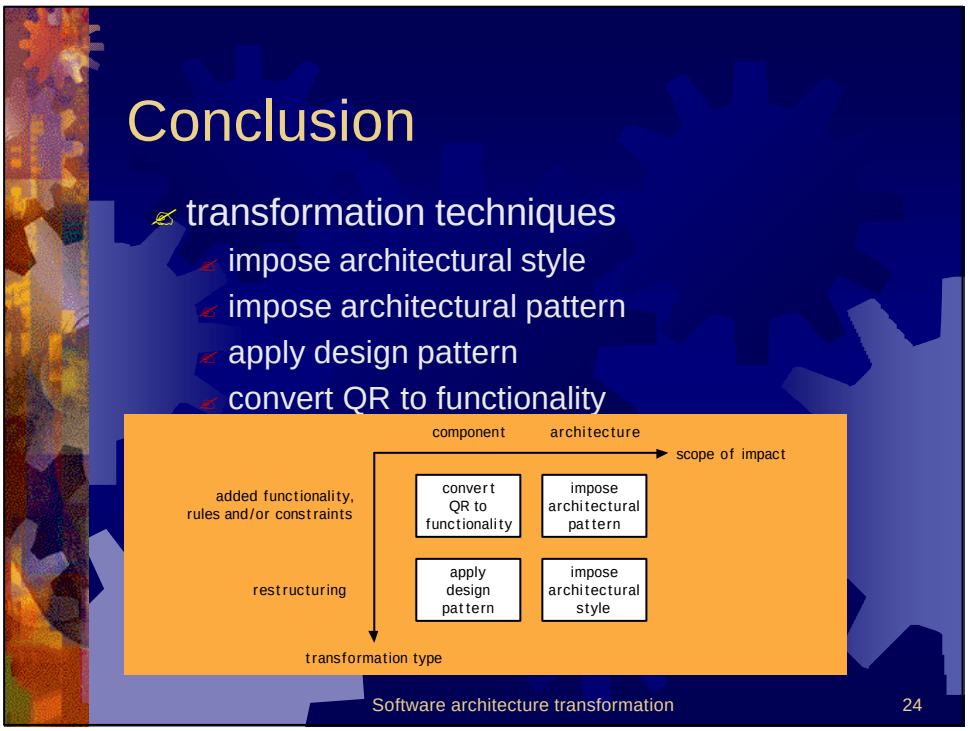
Software architecture transformation 21

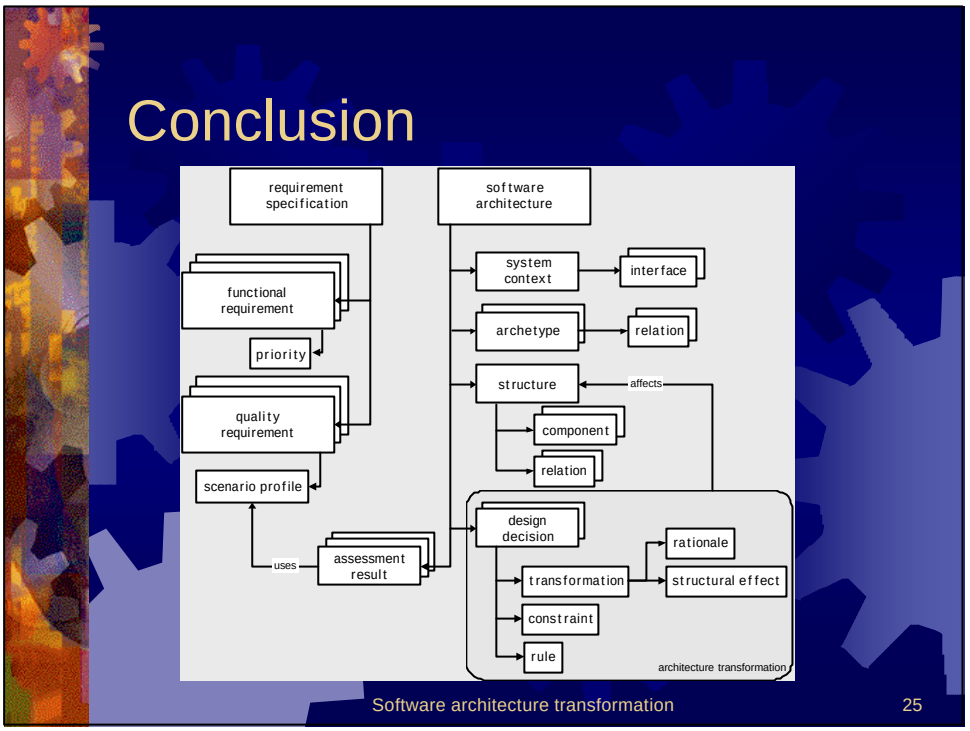
Distribute Requirements

- decompose system requirement to requirement for multiple components
- example
  - fault-tolerance divided into FT-communication and FT-computation
  - response-time requirement divided into shorter component response times

Software architecture transformation 22


Example
Input Device
Input Point
Input Device
Input Point
Input Device
Input Point
Input Device
Input Point
Input Device
Input Point
Deviation
Deviation
Communication Software
Output Point
Output Device
Output Point
Output Device
Output Point
Output Device
Software architecture transformation
23


Conclusion
transformation techniques
impose architectural style
impose architectural pattern
apply design pattern
convert QR to functionality
component
architecture
scope of impact
added functionality, rules and/or constraints
convert QR to functionality
impose architectural pattern
restructuring
apply design pattern
impose architectural style
transformation type
Software architecture transformation
24


Conclusion
requirement specification
software architecture
functional requirement
priority
quality requirement
scenario profile
uses
assessment result
system context
interface
archetype
relation
structure
affects
component
relation
design decision
rationale
transformation
structural effect
constraint
rule
architecture transformation
Software architecture transformation
25

Introducing Software Product Lines



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

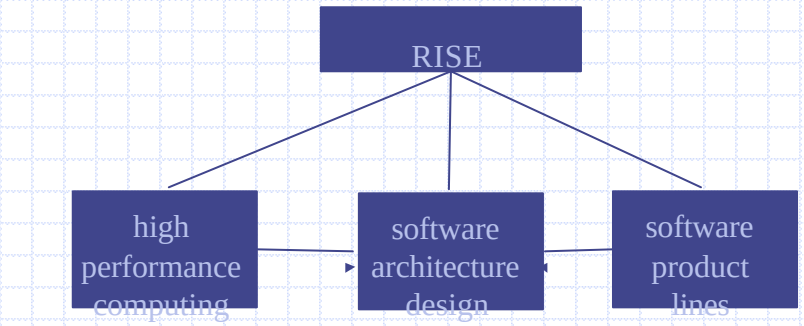
Background

- Head of Software Engineering group at University of Groningen - earlier head of RISE research group @ Blekinge Institute of Technology
- NOSAR - Nordic Network on Software Architecture Research NOSA'98 & NOSA'99 - Nordic Workshop on Software Architecture, NOSA 2001
- ECOOP workshops:
  - WCOP'97, '98, '99, '00,'01 - Component-Oriented Programming
  - OOSA'98 - Object-Oriented Software Architectures
  - OPLA'99 - Product-Line Architectures
- ICT-Architecture'99, November, Amsterdam
- SPLC1 - First product-line SA conference (organized by SEI)
- 3rd Int. WS on Dev. and Evo of SAs for Product Families
- EUREKA/ITEA ESAPS project, 300 man-years, 99-01

Industrial Partners

- Ericsson Software Technology
- Althin Medical
- Axis Communications
- Symbian (earlier Ericsson Mobile)
- EC-Gruppen
- Securitas Larm
- Ericsson Software Architecture Research Lab
- CombiTech Systems
- etc.

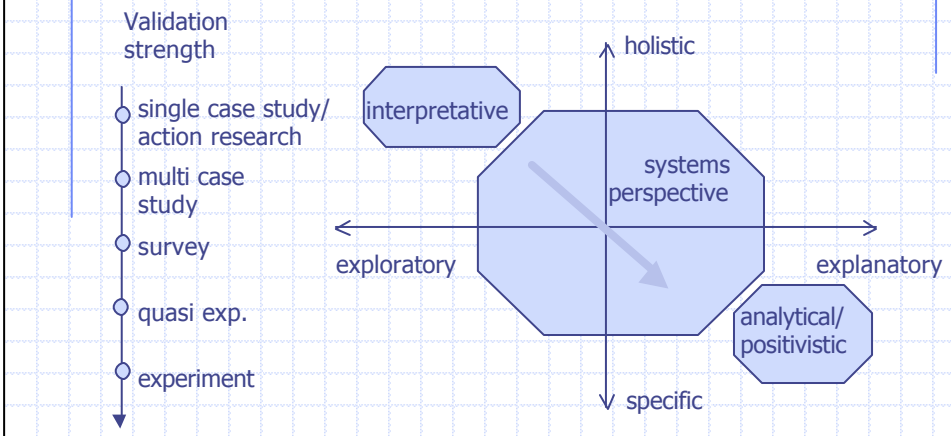
Research In Software Engineering



Professor of Computer Engineering
4 PhD students

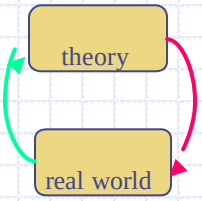
Professor of Software Engineering
3 PhD students
4 PhD students

Research Perspective

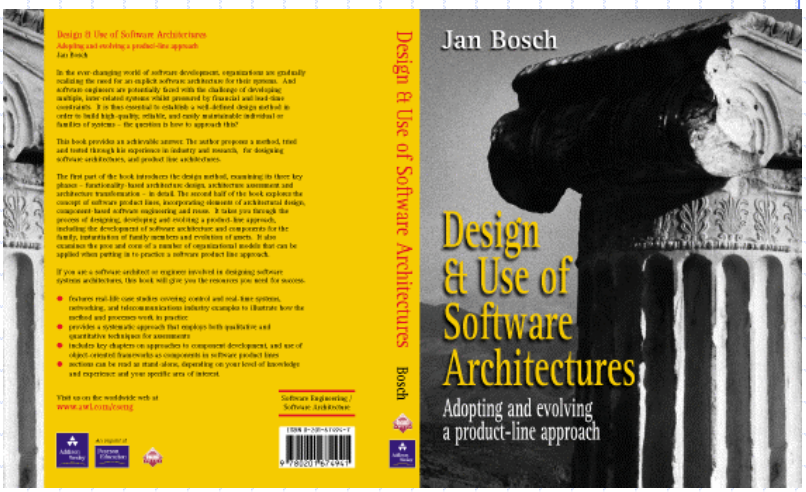


Research Methods

- analytical methods
  - formal models, theories, proofs, etc.
- empirical methods
  - experiment (quantitative)
  - survey (quantitative)
  - case study (qualitative or quantitative)
  - action research (qualitative)



Book!



Overview

- role of software
- software reuse
- software components
- software architecture
- software product lines
- conclusion

Role of software

- software constantly increasing in size and complexity
- software makes up competitive edge
- examples
  - Securitas Larm AB - fire alarm systems
  - business process reengineering and corporate information systems

Traditional approach to SD

- one application/product at a time
- focus on delivery and deadlines
- maintainability & evolution not in focus
- typical problems
  - time & budget
  - system quality
  - maintenance cost up to 80% of total product
  - decreasing competitiveness

Software Reuse

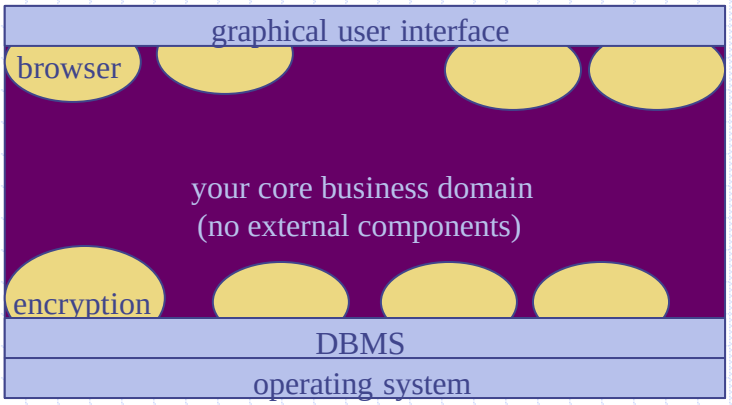
- drastically decrease cost of software development and maintenance
- increase quality of software
- reuse of existing software is one of the most promising approaches
- construct applications by composing reusable software pieces

(cost ✍ TTM ✍ staff)

Examples of reuse

- code "scavenging" (copy & paste)
- operating systems
- compilers
- class libraries
- design patterns
- distribution systems, e.g., CORBA, COM, etc.
- graphical user interface (GUI) systems
- design reuse (expertise)
- object-oriented frameworks
- database management systems (DBMS)
- application generators

Internal vs. External Components



External Component Lifecycle

- part of application
- identified as domain by visionaries
- prototype development (universities, research labs)
- productification (often by small (new) companies)
- adoption by the "pragmatists"
- incorporation in "infrastructure", e.g. operating system

Two approaches to reuse

- **opportunistic**: the software engineer reuses pieces of software that fit the current problem and adds them to the software.
- **planned**: the organization puts explicit effort in developing reusable artifacts that provide the 'right' abstractions, 'right' level of variability and that fit into an higher level structure.

opportunistic reuse does not work in practice

Two approaches to reuse

- **bottom-up**: reusable components, once developed or mined, are added to a, possibly large, collection of assets. Software engineers search their way through these assets looking for suitable pieces.
- **top-down**: assets are developed and presented as parts fitting into a higher-level structure. Assets adhere to predefined provided and required interfaces.

bottom-up reuse does not work in practice

Software Components

- three levels of component reuse
  - reuse of software components over subsequent versions of a product
    - we know this trick
  - reuse of components over product versions and various products
    - we're learning this trick
  - reuse of components over product versions, various products and different organizations
    - we're nowhere near learning this trick (except in very restricted domains, e.g., Visual Basic)

Software Components

Research

- Reusable assets are black-box components
- Assets have narrow interface through a single point of access.

Industry

- Assets are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks
- The asset interface is provided through entities, e.g., classes in the asset. These interface entities have no explicit differences to non-interface entities.

Software Components

Research (cont.)

- Assets have few and explicitly defined variation points that are configured during instantiation
- Assets implement standardized interfaces and can be traded on component markets
- Focus is on asset functionality and on the formal verification of functionality

Industry (cont.)

- Variation is implemented through configuration and specialization or replacement of entities in the asset.
- Assets are primarily developed internally. Externally developed assets go through considerable (source code) adaptation to match the product-line architecture requirements
- Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance

Software Architecture

- software architecture

  The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components and the relationships among them. [Bass et al. 98]

- quality requirements

  Quality requirements can be categorised into development QRs, e.g. maintainability, and operational QRs, e.g. performance and reliability

Why software architecture?

first class representation of SA during

- design
  - assessment of quality attributes
  - software product lines
- implementation
  - configuration management (SPLs)
- run-time
  - dynamic software architectures, post-fielding upgrades, dynamic reorganization, etc.

Views on Software Architecture

Research

- architecture explicitly defined
- architecture consists of components and connectors
- ADLs used to describe architectures and generate applications

Industry

- mostly conceptual understanding of architecture, little explicit description
- no explicit first-class connectors (ad-hoc solutions for run-time binding and glue code)
- programming (C++) and script languages (Make) used to describe configuration of applications

Software Architecture Use

Three types of SA use:

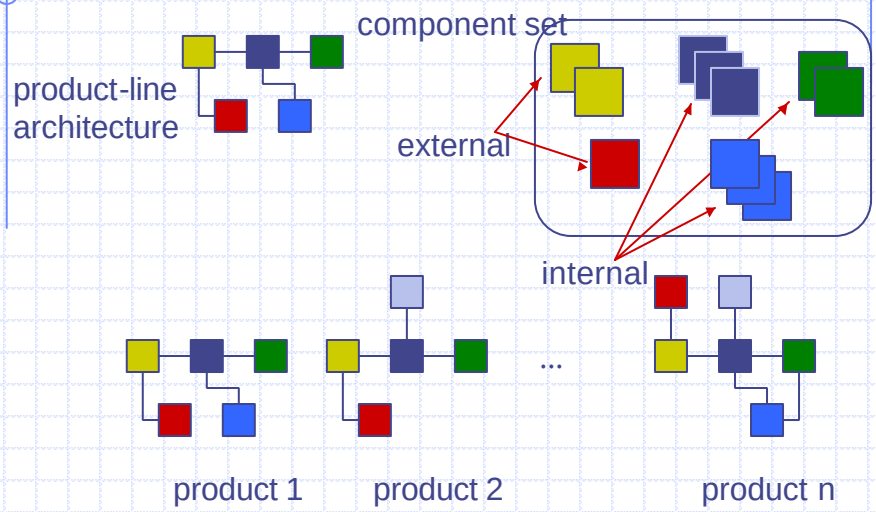
- single system
- software product line
- standardized domain architecture
  - component framework [szyperski 98]



Software Product Lines

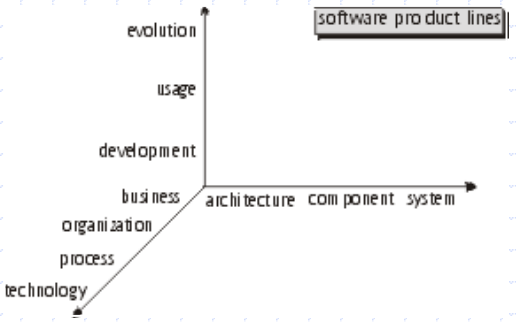
- A software product line consists of:
  - product line architecture
  - set of reusable components
  - set of products, where each product has
    - product architecture derived from PLA
    - instantiated and configured components
    - product-specific code

Software Product Lines



3D's to Decomposing SPLs

- Assets [Jacobsen]:
  - architecture
  - components
  - systems
- Views [SEI]:
  - business
  - organization
  - process
  - technology
- Lifecycle [Bosch]:
  - development
  - usage
  - evolution



Initiating a Product Line

	Evolutionary	Revolutionary
Existing product line	Develop vision for PLA Develop one comp. at a time by evolving existing components	Develop PLA, components and products based on requirement super set
New product line	PLA and components evolve based on requirements posed by new PL members	PLA and components developed to match requirements of all PL members

Evolving Existing Products

- advantages
  - reduced risk due to
    - small up front investment
    - early return on first shared components
  - relatively small effect on production schedule
- disadvantages
  - larger total investment

Replace Existing Products

- advantages
  - shorter conversion time
  - smaller total investment
- disadvantages
  - higher risk
  - negative effect on production schedule

role of hardware and mechanical parts

Applicability of SPL concepts

- context: consultancy company or IT department performing projects with partially overlapping requirements
- define common architecture and common components/subsystems
- develop (slightly) more general components in normal projects
- increase the variability and generality of the component in subsequent projects
- **balance investment and risk!**

SPL Overview

- development
  - design product line architecture
  - develop software components
- deployment
  - develop members of product line
- evolution
  - evolve all assets in the product line

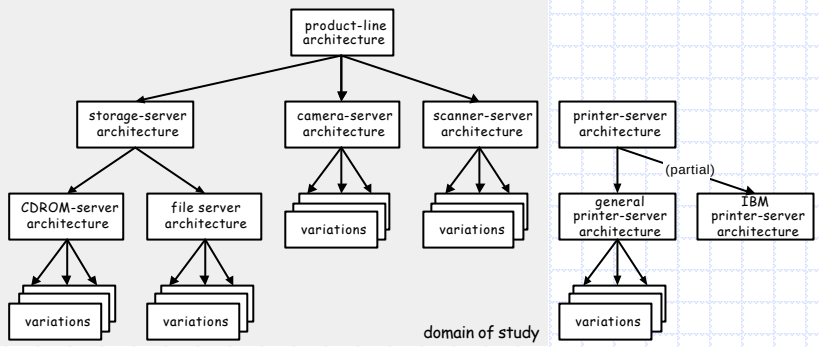
Cases

Much discussion will be based on three cases:

- Axis Communications AB
- Securitas Larm AB
- Symbian Ltd.

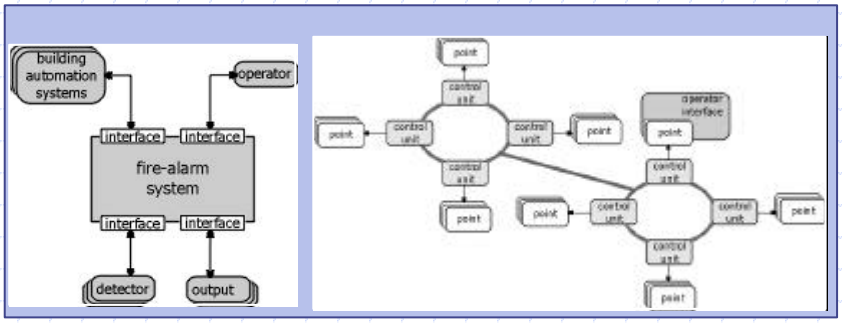
Axis Communications

- 1984, proprietary RISC CPU
- Assets: protocol FW, file system FW, etc.



Securitas Larm

- Former TeleLarm
- Asset: Fire Alarm FW



Symbian



- Former Psion Software (Ericsson Mobile)
- Main asset EPOC
  - an operating system, application framework and application suite
- for three DFRDs
  - landscape display communicator (640 x 200 pixels and up)
  - portrait display communicator (approx. 320 x 240 pixels)
  - smart phone platform

Conclusion

main conclusions

- component-based software engineering requires a top-down approach
- product-line architectures provide a framework for components
- three cases: Axis Communications, Securitas Larm and Symbian

Designing a Product Line Architecture



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

- business case analysis
- scoping
- product and feature planning
- product line architecture design
- component requirement specification
- verification

Business Case Analysis

- why adopt SPL?
  - decrease cost
    - software development
    - software maintenance
  - time to market
    - of new products
    - of new features in existing products
  - staff
    - unable to recruit new staff
    - need to maintain existing staff

Business Case Analysis

- analysis of current situation
  - collect key figures
- prediction of future cost and benefit current approach
  - use key figures to make a prediction
- investment analysis of adopting SPL
  - incorporate technical development cost, but also the cost associated with changes to processes, organization and business

Business Case Analysis

- prediction of future cost and benefit using SPL approach
  - divide cost over domain engineering and application engineering
- conclusion
  - theory: biggest $ figure wins
  - practice: balance short term costs of SPL approach to long term costs of traditional approach

Scoping

- Feature
  - captures logical unit of behaviour (from user perspective)
  - used to group requirements
  - consists of functional and quality requirements
- Feature relations
  - depends-on
  - mutually exclusive
  - conflicting

Scoping

- process:
  - candidate product selection
  - candidate feature selection
  - feature graph specification
  - product line scoping
  - specification of product requirements

Scoping - candidate products

- existing product set
  - all existing products
  - SPL initiation takes time - take delay into account
- new product line
  - strategic plan with greenfield approach
  - competitor products

read [Dikel et al 97]

Scoping - candidate features

- stakeholders
  - features from stakeholder perspective
- existing product set
  - harmonise features
- new product line
  - consider added value of feature to product line
- define product / feature matrix
  - initial pruning

Scoping - features graph

- present identified features in graph
  - relation types: depends-on, mutually exclusive, conflicting
- identify missing features
  - e.g. indirect usage
- draw product 'areas'

Scoping - conclusion

- decide scope of product line
  - relate to SPL goal, e.g. cost, TTM, staff
- minimalist approach
  - only features covered by all or most products
  - develop products as extensions to SPL
- maximalist approach
  - all or most features used in two or more systems
  - develop products by pruning SPL

Scoping - product requirements

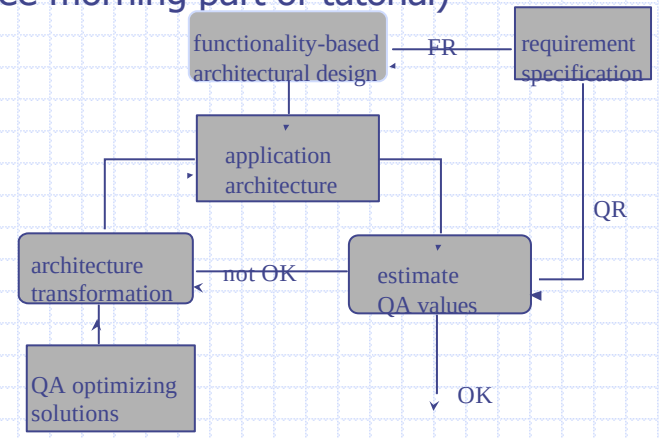
- boundary between SPL and product defined
- specify product specific requirements
  - additional features
  - pruned features
  - conflicting features
  - managing SPL and product quality requirements

Product and Feature Planning

- scoping only considers status quo
- incorporating new features
  - expected versus unexpected
- define technical roadmap for SPL
- extend feature graph with planned features

Product line architecture design

- (see morning part of tutorial)



Functionality based design

- defining system context
  - multiple contexts should be supported (products & variants)
- identifying archetypes
  - SPL and product archetypes (relations?)
- describing product instantiations

Architecture assessment

- not qualities of PLA, but of product architectures
  - how to make sure?
  - consider cost (assessment expensive!)
- product architecture assessment
  - reference product
  - extremes (e.g. low end versus high end products)
  - most important product (e.g. largest volume)
- incorporate assessment of future features

Architecture transformation

three main aspects

- variants
  - examples: abstract factory, strategy, mediator
- optionality
  - examples: blackboard, proxy, strategy
- conflicts
  - examples: adapter, proxy, mediator

Component requirements

specify component requirements

- **interfaces**: provided, required and configuration
- **functionality**: expected behaviour at interfaces
- **quality attributes**: needed to guarantee product behaviour
- **variability**: aspects of functional and quality requirements that differ between products

Verification

- SPL represents large and long-term investment
- design decisions incorporated in PLA hard to change
- stakeholder based evaluation suitable
- alternatively, employ architecture assessment team

Conclusion

six main steps:

- business case analysis
- scoping
- product and feature planning
- product line architecture design
- component requirement specification
- verification

Developing Product Line Components



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

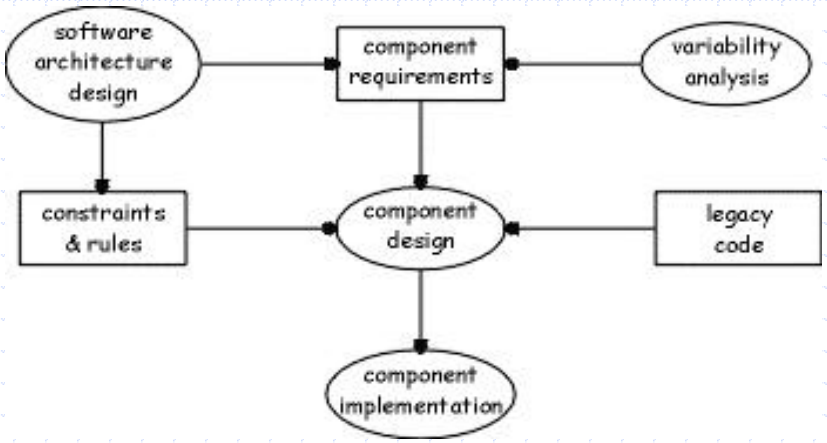
A software component is a unit of composition with explicitly specified provided, required and configuration interfaces only

- traditional component development
- object-oriented frameworks

Software Components

- three levels of component reuse
  - reuse of software components over subsequent versions of a product
    - we know this trick
  - reuse of components over product versions and various products
    - we're learning this trick
  - reuse of components over product versions, various products and different organizations
    - we're nowhere near learning this trick (except in very restricted domains, e.g., Visual Basic)

Component Development



Domains

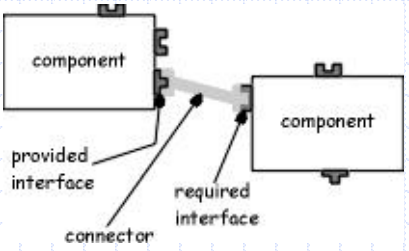
- a domain is an area of functionality identified by as an entity by the user of the domain
- application domains
- software domains
- domain taxonomies, e.g. OMG's special interest groups
- component versus domain: issue of granularity

Component interfaces

- an interface defines a contract between a component requiring certain functionality and a component providing that functionality
- issues with interfaces:
  - all operations available at all times
  - only support for call-reply
  - actual component interface more complex
  - syntax does not guarantee semantics

Component interfaces

- provided interface
  - an identifier
  - a list of operations
  - a list of interface identifiers
- required interface
  - not necessarily one-to-one!



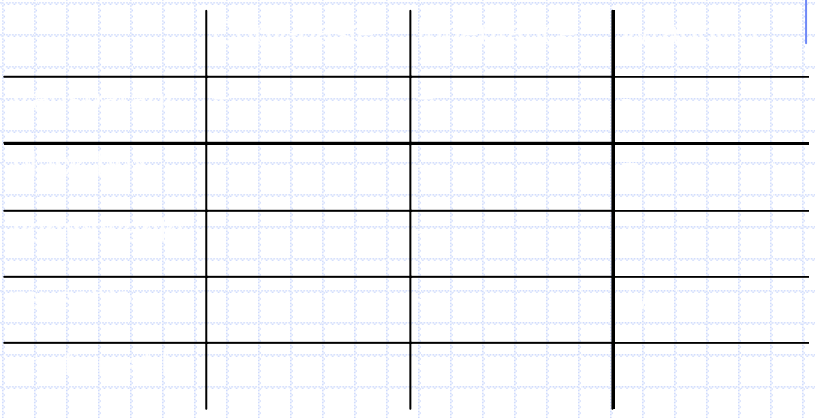
Component interfaces

- variability versus adaptation
- configuration interface (technical + documentation part)
- mechanisms:
  - inheritance
  - extensions
  - configuration
  - template instantiation
  - generation

Component adaptation

- requirements
  - black-box
  - transparent
  - composable
  - reusable
  - configurable
- mechanisms
  - copy-paste
  - inheritance
  - wrapping

Component adaptation



Aspects, Constraints and Rules

- **aspects**: concurrency, synchronization, persistence, transactions and distribution
- **constraints**, e.g. amount of resources (CPU, memory, communications, etc.)
- **rules**: standardised ways of performing certain tasks

Object-Oriented Frameworks

- A framework is a set of classes that embodies an abstract design for solutions to a family of related problems
- categorisations:
  - white-box versus black-box
  - calling versus called
  - evolution pattern: abstract white-box, white-box extensible, black-box, visual builder, DSL

Object-Oriented Frameworks

- concepts
  - core framework design
  - framework internal increment
  - application specific increment
  - object-oriented framework
  - application

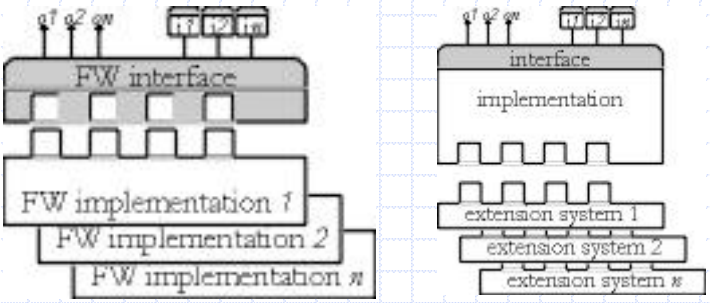
Componentizing frameworks

composition problems

- framework control
- legacy components
- framework gap
- composition overlap of framework entities
- entity functionality

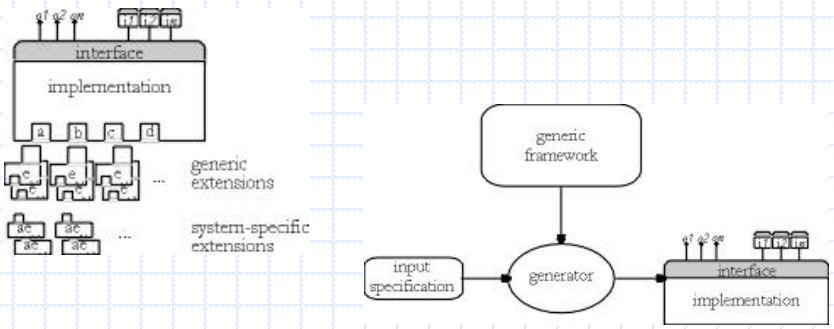
Framework component models

- system-specific extension model
- standard-specific extension model



Framework component models

- fine-grained extension model
- generator-based model



Designing frameworks

- domain and variability analysis
- architectural design
- framework design
- framework implementation
- framework testing
- test instantiation generation
- documentation

Conclusion

- two views on components
  - traditional
  - object-oriented frameworks
- domains
- interfaces
- adaptation
- framework component models

Product instantiation in Product Lines



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

- requirement specification
- product architecture derivation
- component selection and instantiation
- product specific component development
- integration
- validation
- packaging and release

Requirement Specification

- product requirements differences
  - organisational reasons
  - calendar time
- product feature graph
  - overlapping features
  - abundant features
  - conflicting features

Product architecture derivation

- architecture pruning
  - remove irrelevant PLA elements
- architecture extension
  - extend PA with product specific elements
- conflict resolution
  - conflicting, but embedded, functionality
  - missing functionality
  - conflicting quality attributes
  - missing quality attribute
- architecture assessment

SPL component selection

for the product, there may be:

- no implementations
  - implement product specific component
- one implementation
  - simple choice!
- multiple implementations
  - select best match

SPL component instantiation

- traditional component
  - optional behaviour
  - variant selection
  - quality attribute selection
- object-oriented framework
  - system-specific extension model
  - standard-specific extension model
  - fine-grained extension model
- code generator

Product-specific components

- similar to component development in general
- consider future generalisations



Product integration

- collecting components
- wiring connections
  - pipes&filters style
  - even-based style
  - cross-style mismatches
- system integration code
  - run-time component instantiation
  - component initialisation
  - binding system interfaces

Validation

- component testing
  - how to test SPL components?
- system testing
  - develop test cases as part of SPL development
  - specialise test cases
  - automate testing for each product
- field testing
  - must be performed anyhow?

Packaging and Release

- type of system, i.e. pure software or mixed
- first time use or upgrading
- etc.

Conclusion

- requirement specification
- product architecture derivation
- component selection and instantiation
- product specific component development
- integration
- validation
- packaging and release

Evolution in Software Product Lines



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

- new software product line
- new product
- new features
- extend standards support
- new version of infrastructure
- improvement of quality attribute

New software product line

Causes

- too large variability
- business reasons
- incorporating existing systems
- geographical distance
- cultural conflicts

New software product line

- cloning
  - create fully independent copy
  - when: local control and agility
- specialization
  - hierarchical software product line
  - why: increase independence, while maintaining SPL advantages

New product

causes

- market opportunity
- incorporate existing product
- extended family scope

New product

- identify commonalities
- match product architecture and PLA
- develop variation points for SPL components
- develop/reengineer product-specific code
- instantiate product

New feature

causes
- market investigations
- new technological opportunities
- competitors

may cause SPL reorganization

consider conflicts with existing product and feature subsets

Extend standards support

examples:
- network communication protocols
- component communication standards
- file systems

New version infrastructure

- functionality typically moves from border to center to infrastructure
- product -> product line -> context
- infrastructure categories:
  - hardware
  - operating system
  - third party component

Improve quality attribute

- aim: limit architectural impact
- techniques:
  - cache
  - memory management
  - indirect calls
  - wrapper

Post-fielding and run-time evo

- traditional post-fielding evolution
- run-time evolution (AXE switch)
  - component replacement
  - architecture evolution
- remote evolution
- customer-managed evolution

Conclusion

- new software product line
- new product
- new features
- extend standards support
- new version of infrastructure
- improvement of quality attribute
- post-fielding and run-time evolution

Organizing for Software Product Lines



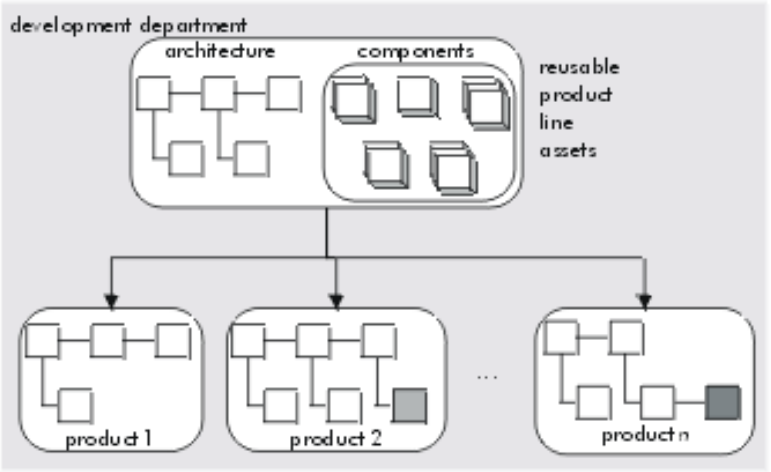
Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

- development department
- business units
- domain engineering units
- hierarchical domain engineering units

Development Department



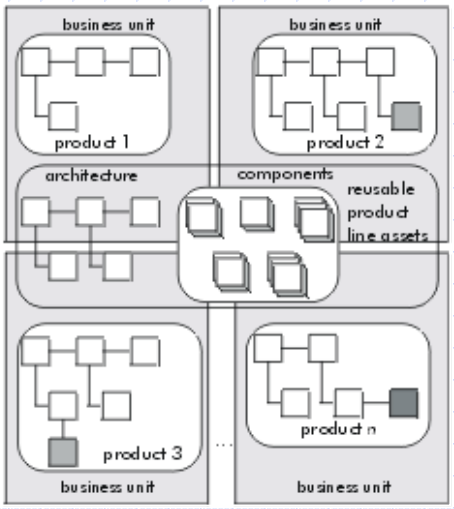
Development Department

applicability
- relatively small organizations (< 30)
- project rather product focus

advantages
- simplicity
- ease of communication

disadvantages
- not scaleable
- focus on domain or application engineering

Business Units



Business Units

three models
- unconstrained model
- asset responsibles
- mixed responsibility

conflicts
- origin of product line

Business Units

applicability

- size: 30 < org. size < 100

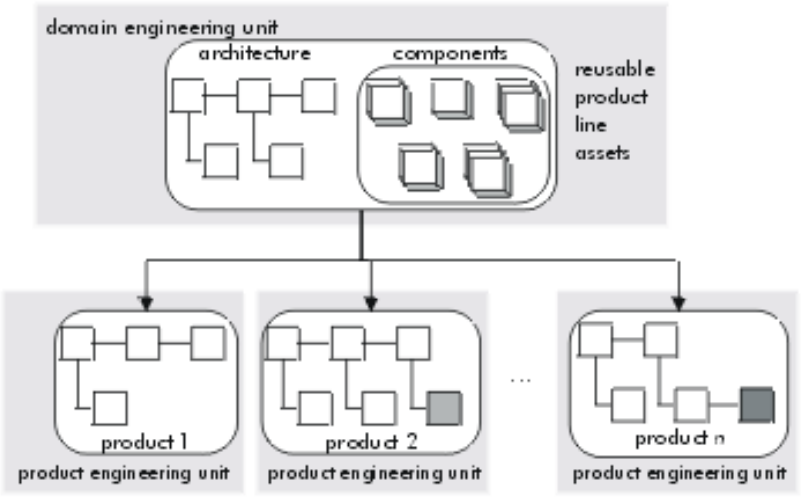
advantages

- effective sharing of assets (evolution!)
- better scaleability

disadvantage

- lack of attention to domain assets

Domain Engineering Unit



Domain Engineering Unit

two models
- single domain engineering unit
- one software architecture unit + component units

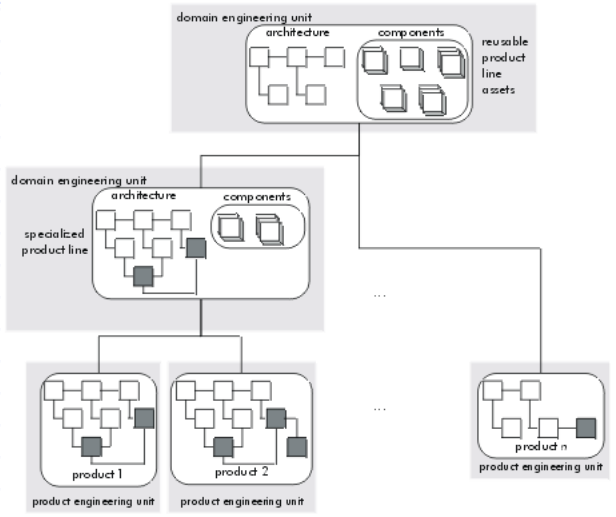
applicability
- size > 100 SEs but less than ??

Domain Engineering Unit

advantages
- reduces n-to-n communication to one-to-n
- focus on both domain engineering and product engineering
- even better scalability

disadvantages
- complex management of evolution & information exchange DU and PU
- time to market of new features

Hierarchical DE Units



Hierarchical DE Units

applicability

- large or very large size (hundreds)

advantages

- manages large, complex systems and large staff

disadvantages

- organisational overhead

Influencing Factors

- geographic distribution
- project management maturity
- organisational culture
- type of systems

Conclusion

- development department
- business units
  - unconstrained model
  - asset responsibles
  - mixed responsibility
- domain engineering units
  - single domain engineering unit
  - one software architecture unit + component units
- hierarchical domain engineering units

Industrial Experiences



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Case study

- Collect technical, process, organizational and business issues in the use of software product lines
- NUTEK-sponsored project, 3 years, 3 PhD students, 3 participating organisations (Axis, Symbian & Securitas Larm)
- Result: 9 problems and 5 issues

Research method

- Data collection
  - group interviews
  - in-depth interviews
  - documents
- Data processing resulted in two papers
- Result validation: feedback from company

Problems

- background knowledge
- information distribution
- multiple versions of assets
- dependencies between assets
- assets in new contexts
- documentation
- tool support
- management support
- effort estimation

Background knowledge

- Problem: SEs require too much knowledge of design rationale and concepts of PLA and assets
- Causes: inherent complexity, abstraction, documentation, standards
- Solutions: first-class representation of PLA & asset architecture, minimize interfaces, documentation
- Research: high-level abstractions, standards, alternative documentation techniques

Information distribution

- Problem: effective distribution of information concerning development efforts & results
- Causes: informal communication channels clogged, no defined and enforced process, business units, no visible differences
- Solutions: domain engineering units, defined and enforced processes
- Research: asset evolution processes, visualisation of asset versions

Multiple versions of assets

- Problem: due to conflicting QAs, two or more optimized asset implementations
- Causes: incompatible QRs, intertwining different types of behaviour, business units
- Solutions: relax QAs, domain engineering units
- Research: late composition of behaviour (e.g. aspect-oriented programming, evaluation techniques

Multiple versions of assets

- conflicting quality requirements
  - reusable assets in the printer server product
- variability implemented through versions
  - printer server product
- high-end versus low-end products
  - file system framework in scanner and camera
- business unit needs
  - file system framework versions. Unification at later point

Dependencies between assets

- Problem: assets tend to develop dependencies during evolution
- Causes: TTM pressure, lack of economic models, no structural maintenance of assets, low visibility
- Solutions: "code reviews", asset responsibles, management commitment, documenting dependencies
- Research: methods and tools for reengineering, high-level abstractions, economic models

Dependencies between assets

- component decomposition
  - authorization functionality in file system framework
- extensions cover multiple assets
  - authorization functionality required extension of multiple assets
- asset extension adds dependency
  - protocol framework: requirement changes required nested packets to refer to encapsulating packets

Assets in new contexts

- Problem: assets are hard to reuse in unintended contexts
- Causes: design for required variability, early mixing of domain, product-category, context and QA behaviour
- Solutions: stress prediction future requirements, explicitly separate different behaviour types
- Research: design for variation point extensibility, late composition of functionality, e.g. AOP & LayOM

Assets in new contexts

- mixed behaviour - components mix domain, product category, operating context and quality attribute related functionality
  - fire-alarm framework applied in intruder alarm systems
- design for required variability - recommended best practice
  - fire-alarm framework applied in intruder alarm systems requires unavailable points of variation

Documentation

- Problem: assets often lack (up to date) documentation
- Causes: documentation low priority, availability of experienced SEs, documentation backlog, lack of appropriate documentation techniques
- Solutions: explicit part of asset evolution process, prioritization by management, use mix of available approaches, e.g. example instantiations, recipes, cookbooks, pattern languages, interface & interaction contracts, etc.
- Research: see earlier slide

Tool support

- Problem: tool development costs development effort, lack of commercial tools
- Causes: commercial tools are 'closed', market too small or not functioning, management prioritization
- Solutions: treat tool development as SPLs: investment, automation of tedious tasks leads to more satisfied staff
- Research: opening up tools

Management support

- Problem: management one-at-a-time mentality, PLA delays TTM first products
- Causes: senior management lack of technical understanding, TTM focus, lack of economic models
- Solutions: expose management to project technology & details, PLA-related decisions are strategic and must be made by upper management
- Research: (see earlier slides)

Effort estimation

- Problem: asset development hard to plan (especially due to iterations)
- Causes: requirements less clear, high degree of variability, superset of QRs
- Solutions: clearer requirements (especially variability & QRs) through more thorough analysis, staff requirements higher for asset development
- Research: method for reusable asset development, extend effort estimation techniques with variation point analysis

Issues

- Domain engineering units
- When to split of products form product-line (and when to merge)
- Business units versus development department
- TTM versus asset quality
- Common feature core versus feature superset

Conclusion

- component-based software engineering requires a top-down approach
- software product lines provide a framework for components
- case study on SPLs involving Axis Communications & Securitas Larm
- 9 problems and 5 issues identified and described
- conclusion: components can be shared over a set of related products, but it is much harder over organizational boundaries

Variability in Software Product Lines



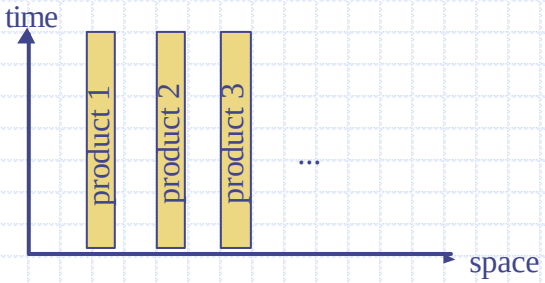
Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

Overview

- Notion of variability
- Feature
- Variation point
  - States
- Variability patterns
- Comparison

Variability



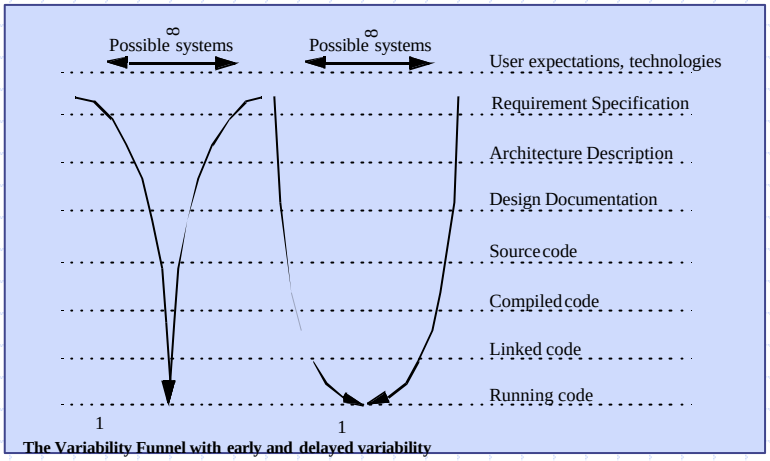
software product line should

- support variability in space (products)
- support 'variability' in time (evolution)
- not prohibit product specific variations

Variability

- key success factor: ability to manage variability!
- variation points delay design decisions
- software product line captures variability through variation points
- trend - where to handle variability:
  - traditional: at/before requirement specification
  - current: architecture/implementation level
  - future: post-fielding / at run-time

Variability

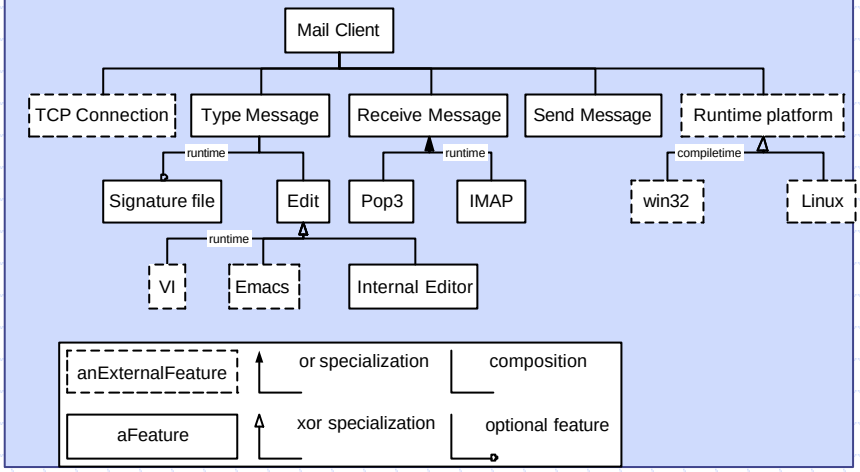


The Variability Funnel with early and delayed variability

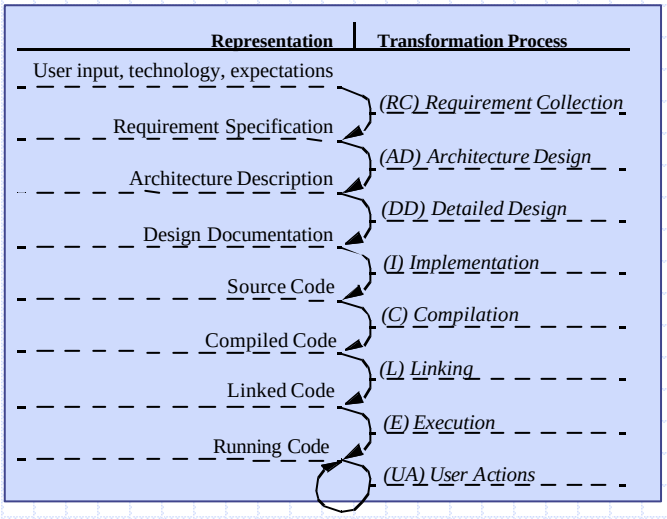
Features

- External features: contextual features that the system depends upon
- Mandatory features: features that identify a product
- Optional features: add, when enabled, some value to the core features of a product.
- Variant features: abstraction for a set of related features (optional or mandatory).

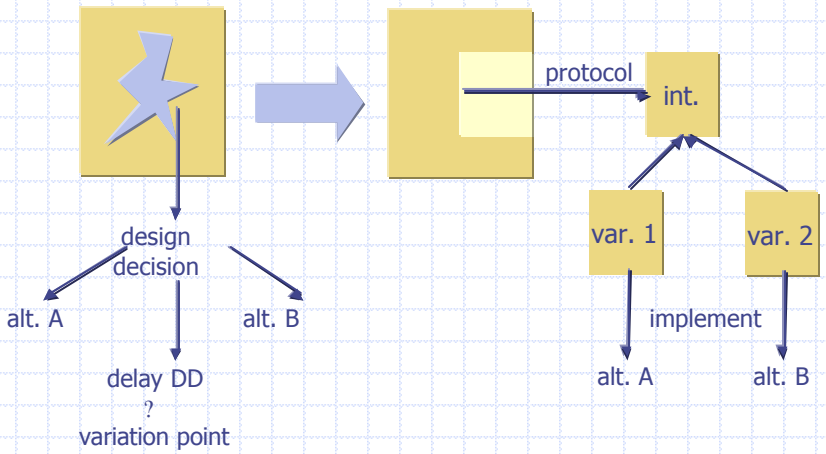
Feature graph



Representation and Transformation



Variation Points: Basic Mechanism



Sources of Variation Points

four possible sources

- at some level during development, the architect or designer does not want to take a decision yet
  - mail client: message editor can be internal or external and plain text or HTML
- differences in products in the software product family
  - mail client: variants for Win32 and Linux/Unix platform
- predicted future changes that are already taken in
  - Mail client: encryption may be support in the future
- during evolution, new requirements require a new variation point
  - mail client: voice/video mail product variant

Variation Points

- variation point
  - implicit – implicitly present in system
  - designed – delayed design decision lead to VP
  - bound – variation has been connected to VP

  times
  - open – possible to add new variants to set
  - closed – set of variants is closed
  - permanent – variant has been bound permanently
- can be introduced and bound at any level

Examples of Mechanisms

- architecture level ? presence/absence architectural component
- component level ? selection of component implementation for architectural component
  ? selection of plug-ins for component implementations
- component implementation level ? set appropriate parameters

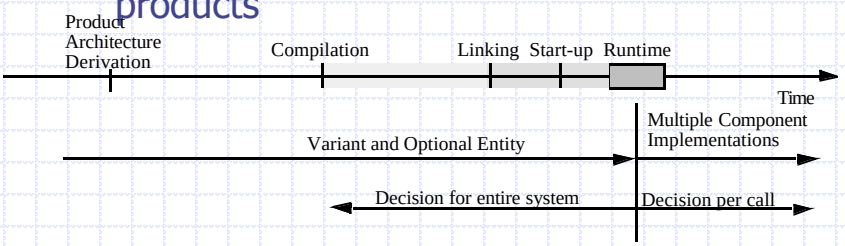
(see our paper for a taxonomy of mechanisms)

Recurring Patterns

- variant entity: only one variant is active in the system at any given moment
- optional entity: only one variant available, and the decision is instead whether to include it or not into the system.
- multiple coexisting entities: The last category is when the running system contains several alternate entities, and the decision of which to use is decided at runtime, before each call, or at least for each job, to the entity.

Binding Times

- binding can be performed
  - before delivery – software product lines
  - post delivery – configurable/dynamic products



Comparison between patterns

Characteristic	Variant Entity	Optional Entity	Multiple Coexisting Entity
Management	Separate from Call	Separate from Call	Performed in Call
Scope of Binding	Valid for Entire System	Valid for Entire System	Valid for one Call
Collection	Implicit or Explicit	Not Applicable	Explicit
Binding	External or Internal	External or Internal	Internal
Open and Closed	Depends on runtime environment	Immediately closed	Depends on tuntime environment

Variability Management

- identification – feature graph notation
- constraining variability:
  - choose binding time
  - decide when and how to add variants
  - select variability pattern
  - select representation for realization
- implementing variability
- managing the variants

Domain Design

1. Identify 'area' of behaviour that should be variable
2. Define protocol and interface between common and variant behaviour
3. Reorganize common behaviour design
4. Implement variants
5. Design variant management

depends on binding time, i.e. pre- or post delivery

Application Design/Implementation

1. Identify variation points that need to be bound during this phase
2. Select most appropriate variant for variation point
3. If no suitable variant available, develop application specific variant
4. For all remaining variation points (bound at installation or run-time):
   1. Select and implement variant management mechanism
   2. For open variation points, select and implement variation addition mechanism

Discussion

- Variation points introduce flexibility in an otherwise rigid structure
- Flexibility through variation points is costly because complexity in increased and, consequently, evolution is more expensive
- We can only predict a subset of all actual changes
- How can we build systems with generic flexibility?

Conclusion

- software product lines
- variability
- feature graph
- representation and transformation
- variability points
- variability management

Concluding Remarks



Jan Bosch
Professor of Software Engineering
University of Groningen, Netherlands
Jan.Bosch@cs.rug.nl
http://www.cs.rug.nl/~bosch

Copyright © 2001 Jan Bosch

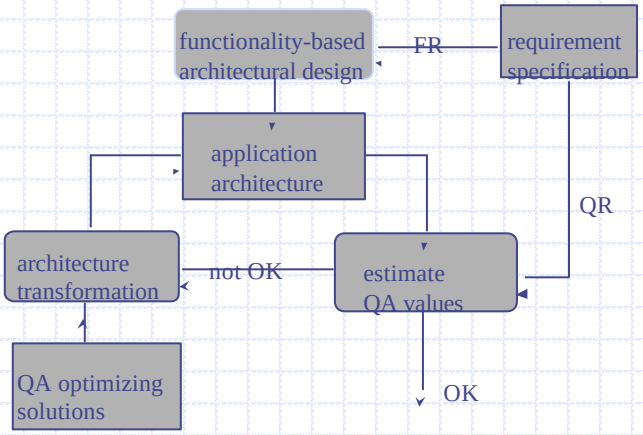
Seminar Overview

- development
  - design product line architecture
  - develop software components
- deployment
  - develop members of product line
- evolution
  - evolve all assets in the product line
- organizational issues
- industrial experiences

PLA design

- business case analysis
- scoping
- product and feature planning
- product line architecture design
- component requirement specification
- verification

Product line architecture design



Component development

- two views on components
  - traditional
  - object-oriented frameworks
- interfaces
- configuration vs. adaptation
- 4 framework component models

Product instantiation

- requirement specification
- product architecture derivation
- component selection and instantiation
- product specific component development
- integration
- validation
- packaging and release

Evolution

- new software product line
- new product
- new features
- extend standards support
- new version of infrastructure
- improvement of quality attribute

Organizational alternatives

- development department
- business units
  - unconstrained model
  - asset responsibles
  - mixed responsibility
- domain engineering units
  - single domain engineering unit
  - one software architecture unit + component units
- hierarchical domain engineering units

Industrial experiences

- background knowledge
- information distribution
- multiple versions of assets
- dependencies between assets
- assets in new contexts
- documentation
- tool support
- management support
- effort estimation

Conclusions

- development
  - design product line architecture
  - develop software components
- deployment
  - develop members of product line
- evolution
  - evolve all assets in the product line
- organizational issues
- industrial experiences

Thank you!

For more information:

Jan.Bosch@cs.rug.nl

http://www.cs.rug.nl/~bosch