Lectures - Overview (BAPO Model)

- Business
- Organisation
- Architecture
- Process
- Economics
- Planning
- Strategy
- People
- Structures
- Roles
- Responsibilities
- Relationships
Lectures - Overview (BAPO Model)
Domain and Application Engineering

## Variability Management

- SPL = Commonality + Explicit Variability
- Variability is Explicitly Managed, i.e.
  - Defined, Represented, Discussed, Exploited, Implemented, Evolved etc.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Prod. 1</th>
<th>Prod. 2</th>
<th>Prod. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game engine</td>
<td>3D, C++</td>
<td>3D, C++</td>
<td>3D, C++</td>
</tr>
<tr>
<td>Score upload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lead character</td>
<td>Mario</td>
<td>Ferrari</td>
<td>None, puzzle</td>
</tr>
</tbody>
</table>
Variability Management

- SPL = Commonality + Explicit Variability
- Variability is Explicitly Managed, i.e.
  - Defined, Represented, Discussed, Exploited, Implemented, Evolved etc.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Prod. 1</th>
<th>Prod. 2</th>
<th>Prod. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game engine</td>
<td>3D, C++</td>
<td>3D, C++</td>
<td>3D, C++</td>
</tr>
<tr>
<td>Score upload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lead character</td>
<td>Mario</td>
<td>Ferrari</td>
<td>None, puzzle</td>
</tr>
</tbody>
</table>
Variability Management

- SPL = Commonality + Explicit Variability
- Variability is Explicitly Managed, i.e.
  - Defined, Represented, Discussed, Exploited, Implemented, Evolved etc.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Prod. 1</th>
<th>Prod. 2</th>
<th>Prod. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game engine</td>
<td>3D, C++</td>
<td>3D, C++</td>
<td>3D, C++</td>
</tr>
<tr>
<td>Score upload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lead character</td>
<td>Mario</td>
<td>Ferrari</td>
<td>None, puzzle</td>
</tr>
</tbody>
</table>
Variability Management

- SPL = Commonality + Explicit Variability
- Variability is Explicitly Managed, i.e.
  - Defined, Represented, Discussed, Exploited, Implemented, Evolved etc.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Prod. 1</th>
<th>Prod. 2</th>
<th>Prod. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game engine</td>
<td>3D, C++</td>
<td>3D, C++</td>
<td>3D, C++</td>
</tr>
<tr>
<td>Score upload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lead character</td>
<td>Mario</td>
<td>Ferrari</td>
<td>None, puzzle</td>
</tr>
</tbody>
</table>

Commonality, part of SPL

Variation, supported in SPL

Variability is first-class concept!
Variability Management

- SPL = Commonality + Explicit Variability
- Variability is Explicitly Managed, i.e.
  - Defined, Represented, Discussed, Exploited, Implemented, Evolved etc.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Prod. 1</th>
<th>Prod. 2</th>
<th>Prod. 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Game engine</td>
<td>3D, C++</td>
<td>3D, C++</td>
<td>3D, C++</td>
</tr>
<tr>
<td>Score upload</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Lead character</td>
<td>Mario</td>
<td>Ferrari</td>
<td>None, puzzle</td>
</tr>
</tbody>
</table>

Variability is first-class concept!

Commonality, part of SPL

Variation, supported in SPL

Product-specific, not supported (now)
Types of Variability
Variability Documentation

- What varies?
  - Variation points

- Why does it vary?
  - Context, Reasons

- How does it vary?
  - Variants, Dependencies, Constraints

- For whom is it documented?
  - Internal & External Stakeholders

- Improves: Decision Making, Communication & Traceability
Graphical Variability Modeling

Class Diagram

Variability Diagram

Home Security by

Camera Surveillance

Motion Detection

Surveillance Device

Alarm Device

Motion Detector

Camera

Infrared Camera

B/W Camera

Color Camera

Control Area

Detection Range

Cullet Detector

Surveillance Object

1..n

0..n

0..1
Graphical Variability Modeling

Separate Model!
Same variability notation throughout
Packages of variants
Variability in packages/sub-systems
Reference Architecture

- Single, shared architecture, common to all products
  - Normal architecture for commonalities
  - Variation points, variants etc for rest
- Not always there in practice, too plan-driven
  - Extract the reference architecture gradually
Industry example: Meantime Game Company

- Brazilian company developing mobile games
  - 60 games, 400 devices, 6 languages, 40 developers
- Critical requirement: Portability (Many mobiles)
  - User Interface Differences
  - CPU, Memory and Size constraints
  - Support API differences (J2ME, BREW & Proprietary)
  - Carrier-specific requirements
  - Internationalization
Industry example: Meantime Game Company

- Developed MG2P = Meantime Game Porting Platform
  - Mobile Domain Database (MDD)
  - Meantime Base Architecture (MBA)
  - Meantime Build System (MBS)

- MDD captures basic Commonality + Variability
  - Variations: Device-specifics, Game types/APIs, Known issues, Language, Game features
  - Families of similar MobPs and Games (in porting context)
  - Typical device for each family chosen (least powerful, most issues)
### Configuration knowledge mapping device variability to preprocessing tokens.

<table>
<thead>
<tr>
<th>Category</th>
<th>Sub-Category</th>
<th>Variation</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device specific</td>
<td>Screen Size</td>
<td>128x117</td>
<td>device_screen_128x117</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128x128</td>
<td>device_screen_128x128</td>
</tr>
<tr>
<td></td>
<td></td>
<td>130x130</td>
<td>device_screen_130x130</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128x142</td>
<td>device_screen_128x142</td>
</tr>
<tr>
<td></td>
<td></td>
<td>128x149</td>
<td>device_screen_128x149</td>
</tr>
<tr>
<td>Game Features</td>
<td>Usage of Tiled Layer API</td>
<td>Meantime API</td>
<td>game_tiledlayer_api_meantime</td>
</tr>
<tr>
<td></td>
<td></td>
<td>MIDP 2.0 API</td>
<td>game_tiledlayer_api_midp2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Siemens Game API</td>
<td>game_tiledlayer_api_siemens</td>
</tr>
</tbody>
</table>
Industry example: Meantime Game Company

- **Meantime Base Architecture**
  - Same code base and file structure for all games
  - J2ME does not allow libraries => MBA copied for each new game
  - Pre-processing tokens from MDD handles variability

- **Meantime Build System**
  - Built on Antenna pre-processor and Ant, more flexible
Architectural Concerns

- Architecturally Significant Requirements
  - Key requirements affecting the whole architecture

- Conceptual Architecture
  - Key concepts of architecture

- Architectural Structure
  - Decomposition into components and relations

- Architectural Texture
  - Rules for using, instantiating and evolving architecture
Architecturally Significant Requirements

- Central to the purpose of the products, or
- Technically Challenging / Technical Constraints

Examples:

- The system must encrypt all network traffic
- The game must deploy on all mobile phones by the top 5 manufacturers that are released after 2007
- The system must always give responses to user queries within 3 seconds
- The system must provide a visual overview of the current flow of resources in the factory being managed

- Quality/Non-func requirements often decisive
Conceptual Architecture

- Most important concepts + their relations
- Mental model of domain to understand and simplify the problem
- (Related to “System Metaphor” in Extreme Programming)
Architectural Structure

- Division into components
  - Sub-systems/units with clear interfaces
- Connections between components
Architectural Texture

- "Manual" for the Reference Architecture
  - Guidelines, rules, "Philosophy" for
    - Using and
    - Evolving the RefArch
- Examples:
  - Coding standard
  - Design patterns
  - Architectural styles
Creating a Reference Architecture

- “Normal” architecting methods can be used
  - Attribute-Driven Design, ..., OO, ..., Design Patterns, ...

Differences:
- More products, often more Stakeholders => Communicate
- Also more Requirements conflicts => Resolve

Three basic ways to support Variability:
- Adaptation
- Replacement
- Extension
Variability Mechanisms

adaptation

replacement

extension
Variability Mechanisms

adaptation

replacement

extension

Only 1 component implementations
Adaptable behavior
Variability Mechanisms

- **adaptation**: Only 1 component implementation. Adaptable behavior.
- **replacement**: Multiple component implementations. Choose one, or develop product-specific.
- **extension**:
Variability Mechanisms

adaption

replacement

extension

Only 1 component implementations
Adaptable behavior

Multiple component implementations
Choose one, or develop product-specific

Generic interface for adding components
Adaptation mechanisms

- Inheritance
  - subclass changes/overrides behavior

- Patching
  - partial behavior change with little maintenance
  - DE: component, AE: patch

- Compile-time config
  - Pre-processors or macros, Makefiles

- Configuration
  - Interface to choose between multiple implementations
  - Parameters or configuration file to make choice
Replacement mechanisms

- Code generation
  - Generates code from high-level description (model, script)
  - Glue code or whole components/sub-systems

- Component replacement
  - Default component is replaced with another one
  - Often 3rd party components
  - Wrappers may be needed
Extension mechanisms

- Plug-ins
  - Architecture has interface to “plug in” components
  - Example: Corba, COM, etc
  - Example: Strategy Design Pattern
Variability & Commonality SPL Motivations

- Increase in the number of products that can be released
- Manage multiple, diverse products in one portfolio
- Improve product commonality
  - Not only for complexity management,
  - also for marketing (same look-and-feel)
Industry Case: Philips Consumer Electronics

- 16000 employees, €10Billion turnover (1/3 is TVs)
- 250 developers
- Single SPL for mid- and high-range TVs
- SPL developed 1996-2000, in use since then
- Trends, more complex SW:
  - More features (MPEG4, Sound processing, HW->SW)
  - Globalized market
  - Shorter product cycles and TTM
  - Product convergence
Industry Case: Philips Consumer Electronics

- Hundreds of Variability parameters -> Hierarchy
- Evolution rules: What can be changed without affecting other parts? (HW dependencies)
- Compositional approach technically
  - Describe which components to combine into new product
  - Simplified convergence (DVD+TV, TV+VCR, …)
Industry Case: Philips Consumer Electronics

- Koala Component Model
  - Component = Specification + Implementation
  - Hierarchical - group of components can be one component at higher level
  - Implemented in C, interfaces in separate files
  - Component descriptions to generate build/make files
    - Interface Description Language + Tools to work with it
  - No extra run-time costs (resource-constrained HW)
Industry Case: Philips Consumer Electronics

- Variability
  - Compound components can have “Diversity parameters”
  - Switches to choose sub-components
  - Packages group components and interfaces to larger units
  - Also the packages are hierarchical
  - Product is a selection of packages
Industry Case: Philips Consumer Electronics

- Reference architecture?
- What are the Variability mechanisms? (Adaptation, Replacement, Extension)
- Documentation of variability?
Industry Case: Philips Consumer Electronics

- Reference architecture?
  - No, since it would not help for creating combi-products
  - Maybe for small line of TVs, not for whole range over multiple years

- What are the Variability mechanisms? (Adaptation, Replacement, Extension)

- Documentation of variability?
  - Only: Component & Interface data sheets + sub-system design notes
Industry Case: Philips Consumer Electronics

Results / Lessons learned

- Diversity of products produced on time, Variability not a problem
- Late-joining architects don’t understand Koala’s motivation
- Architecture has lasted longer than any previous
- Took 3 years to be successful
- Config Management system fails at sub-file level variability
  - Better to solve variability in arch & use traditional CM
Evolving a Reference Architecture

- Evolution is a must:
  - Market changes
  - Features or products become redundant
  - Company mergers
  - 3rd party component updates
  - New technology

- Unintentional evolution:
  - Software/documentation rot, Maintenance, Erosion
  - Refactoring can counter
The game should support

- ... either 32-bit color output...
- ... or 16-bit color output...
- ... from the graphics engine.
The game should support...

... either 32-bit color output...

... or 16-bit color output...

... from the graphics engine.
The game should support

- either 32-bit color output
- or 16-bit color output
- from the graphics engine.
Requirements Variability - Textual

- The game should support
  - ... either 32-bit color output... Variation 1
  - ... or 16-bit color output... Variation 2
  - ... from the graphics engine.
Requirements Variability - Use Cases

Tabular Scenario

<table>
<thead>
<tr>
<th>Inhabitant</th>
<th>Home Security System</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Inhabitant approaches the front door</td>
<td></td>
</tr>
<tr>
<td>2. System requests authentication</td>
<td></td>
</tr>
<tr>
<td>3. Inhabitant enters the PIN</td>
<td></td>
</tr>
<tr>
<td>3. Inhabitant touches the fingerprint sensor</td>
<td></td>
</tr>
<tr>
<td>4. System permits entry to the home</td>
<td></td>
</tr>
</tbody>
</table>

Variability Diagram

- Door Lock
  - VP
  - Keypad
  - Fingerprint Scanner
Scoping

- Defining the scope of the product line
  - Which products are within the boundaries of the SPL?
  - Which products are not supported by the SPL?
- Product Portfolio Scoping
- Technical, Marketing and Strategic Decision

Other levels (built on PPS):
- Domain scoping = Identify major domains relevant for SPL
- Asset scoping = Define functionality for reusable components

Active research area
Example scoping: Philips Consumer Elec.

- Main SPL Scope = “Mid- and High-range TVs”
  - Support convergent/combi products
  - Not low-end TVs
    - Less features => less variability
    - Less product-to-product changes => less variability
    - HW+SW mainly bought from 3rd party

- Flexible and Ongoing Domain Scoping
  - Convergence & short cycles requires new domains

- Asset scoping built into component framework
Product Portfolio Scoping

1. Define Product Line Market

2. Determine relevant Product Types
   - Product Map = List of example products/types with their main features = Defines the Portfolio

3. Analyse Market Position & Define Products
   - KANO Model

4. Analyse interrelations between products
   - Competition - PL Cannibalisation
   - Support - Entry-level sells premium-level
Product Portfolio Scoping

1. Define Product Line Market
   - Product Map = List of example products/types with their main features = Defines the Portfolio

2. Determine relevant Product Types
   - KANO Model

3. Analyse Market Position & Define Products
   - Competition - PL Cannibalisation
   - Support - Entry-level sells premium-level

Identifying Commonality and Variability is natural in scoping => SPL good fit
Domain Requirements Engineering & Analysis

- Normal RE and Analysis but Precise VariabilityDefs
  - Commonality Analysis
  - Variability Analysis
  - Variability Modeling
- Methods
  - App-Req Matrix
  - Priority-based Analysis (KANO)
  - Checklists
Acronyms used

- DE = Domain Engineering
- AE = Application Engineering
- RefArch = Reference Architecture
- TTM = Time To Market
- SW = Software
- SPL = Software Product Line
- SPL = SPL Engineering (and course book!)
- Dev = Development
V. Alves, T. Camara, C. Alves, “Experiences with Mobile Games Product Line Development at Meantime”, SPLC’08, Limerick, Ireland, 8-12 Sept, 2008.