Seminar

Software Product Lines and pure::variants

Dr. Danilo Beuche
danilo.beuche@pure-systems.com
About pure-systems

• **Business Areas**
  - Development Tools
  - Software Development
  - Consulting & Professional Services
  - Training

• **Customer**
  - Mainly embedded systems manufacturers

• **Founded 2001, Location Magdeburg, Germany**
Presenter

- Danilo Beuche
  - 2001-*: managing director pure-systems GmbH, Magdeburg, Germany
    - consulting in embedded software development and product line development
  - 1997-2003: Research assistant/PhD Student University Magdeburg, Germany
    - PhD on Software families for Embedded Systems
    - operating systems group, focus embedded operating system families
  - 1995-1997: Research associate GMD FIRST Berlin, Germany
    - parallel operating systems group, focus on families
Introduction to
Software Product Lines
Terminology

- Software Product Line terminology is used throughout the seminar:
  
  - Problem Space vs. Solution Space
  
  - Domain Engineering vs. Application Engineering
  
  - Variant vs. Version
  
  - Variation Point
  
  - (Core) Assets
Terminology (2)

Domain Engineering

Problem Space

PL Properties and Relations

Solution Space

Assets

Product Properties

Single Problem

Application Engineering

Single Solution

Production Plan
Terminology (3)

Domain Engineering

Problem Space
- Domain Expert

Solution Space
- PL Architect/Developer

PL Expert

Application Engineering

Single Problem
- Application Analyst

Single Solution
- Application Developer
# The Version Hell

<table>
<thead>
<tr>
<th>Component A</th>
<th>Product 1</th>
<th>Product 2</th>
<th>Product 3</th>
<th>Product 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.1</td>
<td>1.3</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>Component B</td>
<td>1.0</td>
<td>1.2</td>
<td>2.1</td>
<td>2.4</td>
</tr>
<tr>
<td>Component C</td>
<td>1.0</td>
<td>1.0</td>
<td>2.3</td>
<td>4.0</td>
</tr>
</tbody>
</table>
Orthogonality of Variants and Versions

Variants

<table>
<thead>
<tr>
<th>Variants</th>
<th>v1.0</th>
<th>v1.2</th>
<th>v1.3</th>
<th>v1.4</th>
<th>v2.0</th>
<th>v2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0</td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1.2</td>
<td></td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1.3</td>
<td></td>
<td></td>
<td>🔴</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>v1.4</td>
<td></td>
<td></td>
<td></td>
<td>🔴</td>
<td></td>
<td></td>
</tr>
<tr>
<td>v2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>🔴</td>
<td></td>
</tr>
<tr>
<td>v2.2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>🔴</td>
</tr>
</tbody>
</table>

Version / Time
Terminology (4): Version and Variants

• Version (also Revision)
  
  – Versions of an object represent the same object at different times. The object may or may not have changed in different versions.

• Variant
  
  – Variants represent objects with different user-distinguishable properties. They are derived from a common base object.
Terminology (5): Variant vs. Version

- Variants may be represented by versions

- Variants may be derived from a single version

- A single variant may over time be derived from different versions of an object

In single system development, variant and version are often used interchangeably, when variants are exclusively represented by changes over time and are thus always coupled to a specific version.
Terminology (6): Variation Point

• Variation Points
  – identify all places where members of a product line may differ from each other
  – exist in problem and solution space
  – have a binding time such as compile time, link time or run time.
  – Example:
    • Problem space:
      “The car has either two or four passenger doors”
    • Solution space:
      #ifdef / #else / #endif encapsulated code fragments
Quick Look at SPL-Development

Essential SPLD Activities (SEI SPLP Framework)

Taken from SPLPFW 4.2, copyright SEI CMU.
Quick Look at SPL-Development

Core Asset Development I/O (SEI SPLP Framework)

Taken from SPLPFW 4.2, copyright SEI CMU.
Parnas (1976): „We consider a set of programs to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties and then determining the special properties of the individual family members.“
Reuse
Reuse Technology – Challenges

- no first level concept of variation points available
- variations cross-cut all levels
- low degree of formalization
- shared responsibility between stakeholders with very different perspectives
Reuse Process - When and Why Reuse Fails (1)

- product development done by engineering companies
  - they sell projects typically based on the expected effort
- creating and maintaining reusable assets is more expensive than one-of-a-kind development style
  - technology mismatch for given domain/organisation
  - not enough education of stakeholders
  - highly parallel development of similar functionality in separate project teams
Reuse Process - When and Why Reuse Fails (2)

- development of unused/unusable assets
  - application team does not use components developed by a specific core asset team
  - products do not meet acceptance criteria of customers
- management does not support development organization well enough
  - projects with a high degree of reuse tend to cost more in the beginning but managers want a cheap start
Reuse - Key Issues

- Economics
- Education
- Communication and Coordination
- Technology
Variant and Variability Management

MIND THE GAP
Variant and Variability Management

- takes care of description and maintenance of
  - variation point information for a product line (variability)
  - instances of the product line (variants)
- closes the gap between reusable assets and instances of them
- is all about dealing with complexity in various areas
Variant and Variability Management

... is complex because

- variability grows more or less exponentially
  
  - each option doubles the number of potential variants
  
  - each group of n alternatives still creates n-times more variants

- it has to deal with anticipated and unanticipated variation requests

- involves more stakeholders compared to single system development
Variant and Variability Management with
**ALM and Variant Management**

<table>
<thead>
<tr>
<th>pure-systems</th>
<th>Variant Management with pure::variants</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Borland</strong></td>
<td>Portfolio Management</td>
</tr>
<tr>
<td></td>
<td>Tempo</td>
</tr>
<tr>
<td><strong>IBM</strong></td>
<td>Portf. Manager</td>
</tr>
<tr>
<td><strong>Telelogic</strong></td>
<td>Focal Point</td>
</tr>
<tr>
<td><strong>MKS</strong></td>
<td>Portfolios</td>
</tr>
</tbody>
</table>

**Application Lifecycle Management (ALM)**
Integration into Development Processes

- keep and use existing code base and tool environment
- version management support
- independent from technology

Efficient Variant Modelling

- accumulation of configuration knowledge
- validation of variant configurations

Automated and Resource-Efficient Variant Generation

- source code packaging (e.g. from version management repositories)
- generation of code, documents, bills of material
Development with Variant Management

Problem Space
Collection of Features and Relations
Feature Model

Solution Space
Collection of Family Elements
Family Model

Single Problem
Desired/Required Features
Variant Model

Single Solution
Variant
Variant Realization
Workflow with pure::variants

1. **Feature Model**
   - definitions of requirements, properties and relations.
   - (design of a product line)

2. **Family Model**
   - collection of artifacts (code, components, parts, documentations)
   - (realization of a product line)

3. **Configuration** = Selection of Constituent Parts

4. **Transformation** = Generation of Variants

Link to Feature Model
### Example Weather Station: Product Variants

<table>
<thead>
<tr>
<th>Variant</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thermometer:</td>
<td>Display, Temperature</td>
</tr>
<tr>
<td>Indoor:</td>
<td>Display, Temperature, Pressure</td>
</tr>
<tr>
<td>Outdoor:</td>
<td>Display, Temperature, Pressure, Wind</td>
</tr>
<tr>
<td>Deluxe:</td>
<td>+ PC Data Recording</td>
</tr>
<tr>
<td>Internet Edition:</td>
<td>+ TCP/IP</td>
</tr>
<tr>
<td>PC Interfaces:</td>
<td>+ Serial Interface RS232</td>
</tr>
<tr>
<td></td>
<td>+ USB Interface</td>
</tr>
</tbody>
</table>
Problem Space – Feature Model

WeatherMon

Output

PCConnection

Conflicts: 'Trace'

Display

RS232Line

USBLine

DebuggingSupport

Trace

partial view
Example Weather Station: Hardware

- **Sensors**
  - Wind
  - Pressure
  - Temp

- **µController (AVR)**
  - 4kB RAM, 8kB Flash

- **Display**

- **I²C**
  - USB
  - RS232

- **UDP/IP or Plain Text over USB/RS232**
Solution Space – Family Model

WeatherStationSolutions

- Trace
  - hasFeature('Trace') and conflictsFeature('PCConnection')

- Display
  - hasFeature('Display')

- PCLine
  - hasFeature('Trace')
  - hasFeature('PCConnection')

- PS: class: RS232Line
  - hasFeature('RS232') or hasFeature('Trace')

- PS: class: USBLine
  - hasFeature('USBLine')
LIVE - DEMO

at work
Product Variants

pure::variants Professional
- models are stored locally in file system as XML
- collaboration using standard software configuration management tools such as CVS, Subversion, Perforce, ...
- versioning and branching handled by software configuration management tool

pure::variants Enterprise
- models are stored in a centralized database
- collaboration in real-time, changes will be automatically visible to any connected user
- integrated reporting and history
- versioning and branching handled by pure::variants server
- all functionalities of p::v Professional

- full support for japanese user interface in pure::variants 3.0
<table>
<thead>
<tr>
<th>Development Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>....</td>
</tr>
</tbody>
</table>

- **Generic Data**
- **Variability Data**
- **Variant Data**

**pure::variants**
Extensions integrate pure::variants optimally into existing tool chains.

- Synchronizer for DOORS
- Synchronizer for CaliberRM
- Connector for MATLAB / Simulink
- Connector for Source Code Management
- Connector for Version Control Systems
- Connector for SAP
- Connector for ClearQuest
- Connector for ClearCase
- Connector for Bugzilla
- Connector for Reporting with BIRT
Integration Interfaces

**p::v Eclipse Client**
- **p::v Eclipse Plugins**
- **p::v Java Core**

**Eclipse API**
- Eclipse Extension Points

**Java API**
- Java Interfaces

**SOAP Interface**

**p::v Server**
- **p::v Runtime System**

**C++ API**
- DLL/COM/OLE
Example Danfoss

**Problem**

- For 4 market segments control software is provided by different, worldwide distributed teams (about 70 persons). Software is based on very similar hardware.

- Reuse mostly "ad-hoc".

- Cost and development effort were high.

**Task**

- Migration to controlled and systematic reuse based on common software platform.
Example Danfoss

Our Role

- Participation during process definition
- Coaching and evaluation of newly setup platform team
- Supporting the migration to pure::variants

Results

- 1\textsuperscript{st} stage of migration finished successfully after 6 month: all projects develop based on common platform
- 2\textsuperscript{nd} stage of migration (introduction of pure::variants) almost completed
- Double number of products with about the same development capacity
Example Automotive Supplier

**Problem**

- Configuration of control software with about 2000 features could not be handled efficient manually.

**Task**

- Modelling of complete configuration knowledge in pure::variants and generation of configuration files.
- Distributed real-time remote access to models and configurations.
Example Automotive Supplier

Our Role

- Product supplier
- Adaptation of pure::variants to customer demands: integration of customer specific rule language.

Results

- Configurations are created in minutes instead of days.
Transforming Legacy Systems into Software Product Lines
Transition and the SPLP-Framework

- Basically, the transition process is used to collect and create all information described in the framework from pre-existing artefacts.

- What is different to a start from the scratch?
  - Much required information is already present, only hidden.
  - Changes to processes and artefacts may be hard to achieve. It is more complicated to introduce adequate processes and/or artefacts.
Questions Before Starting a Transition

• What is missing for a SPL/SPLD in my organisation?
  – Is it...
    ... systematic core asset development?
    ... problem and solution space specifications?
    ... applications build on top of p-space & s-space?

• Why make the transition?
  – Is it...
    ... overall development cost reduction?
    ... shorter development time for similar products?
    ... increased product quality?
    ... ?
Transition Steps

1. Pre-SPL
2. Product Relation Pattern Matching
3. Transition Scenario Identified
4. Variability Analysis
5. Model Building
6. SPL
Product Relation Pattern: Product Forest

No shared software core assets
Similar user-visible functionality
Product Relation Pattern: Product Bush

Shared software core assets
Product variation through development path branching
Product Relation Pattern: Product Gang

Shared software core assets
Product variation through solution configuration
No systematic variant management

Time
Product Relation Pattern: Product Family

Shared software core assets
Systematic variant management
Transition Scenarios

Current Software Assets

Separate Products Merger
- Characteristics:
  - similar problem domains
  - mostly separate solution domains
- Related Patterns:
  - Product Forest
  - (Product Bush)

Reuse Improvement
- Characteristics:
  - similar problem domains
  - same or very similar solution domains
- Related Patterns:
  - Product Bush
  - Product Gang
Transition Steps

1. Pre-SPL
2. Product Relation Pattern Matching
3. Transition Scenario Identified
4. Variability Analysis
5. Model Building
6. SPL
Variability Analysis

• **What to do:**
  - Identification and extraction of (potential) variation points of the product line
  - Extraction of variation point constraints

• **Where to look:**
  - source code
    - structure, algorithms, technologies
  - documentation
    - user manuals, internal documents, code comments
  - management strategies
    - version control, configuration
Variability Analysis (2)

• Results:
  - initial set of problem space variation points
  - initial set of solution space variation points
  - initial set of variation point constraints

• Purpose:
  - providing input for
    • product line scoping
    • problem space definition
    • core asset identification
    • production plan creation
Variation Point Description

- A variation point description consists of
  - choices (features) available at that point
  - variation constraints (requires/conflicts...)
  - space it belongs to (problem/solution)
  - relevance/importance w.r.t. product line variability
  - binding time
    - product configuration time, compile time, link time, installation time, user configuration time, run time
  - binding mode
    - static vs. dynamic
  - used variability pattern
Analysis Direction

- Decide which direction to go:
  - **forward**: start with problem space variability
    - good if user manuals and/or product specification provide good insight into variabilities between existing products; works for product forests
    - permits to perform product line scoping w/o (closely) looking at the existing software assets
  - **backward**: start with solution space variability
    - good if software design and architecture reflect problem space closely; especially suited for product bushes and gangs
    - requires problem space variability extraction before product line scoping can be started
Variation Point Extraction

1. Decide where to look for variability.

2. Build list of representative variation point patterns for your software.

3. Search for variability patterns in your input material:
   - manually
   - supported by tools
Problem Space VP Extraction

• Typical inputs:
  - user documentation
    • installation and user manuals
    • white papers
  - development documentation
    • requirements specifications, use cases
    • architecture design documents
    • log files in version control system (esp. product bush)
    • configuration files (esp. product gang)
    • communication with customers (feature requests etc.)
The Commonality and Variability Extraction Approach

- developed by Fraunhofer IESE in CAFÉ project as part of their Product Line Software Engineering Framework

- focusses on user documentation, but may be used also with development documentation

- operates on textual structure and text comparison
CaVE - Approach

Preparation

User Documentation
Extraction Pattern
Documentation Entities
Selected Extraction Pattern

Analysis

Common and Variable Requirements Fragments

Selection

Partial Product Line Requirements

Process step
In/Output

Domain Expert
Product Line Engineer

Image © IESE 2004
# CaVE - Pattern Template

<table>
<thead>
<tr>
<th>Name</th>
<th>The name of the pattern.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>A one sentence description of the pattern.</td>
</tr>
<tr>
<td>Input</td>
<td>The input model element.</td>
</tr>
<tr>
<td>Output</td>
<td>The output model element.</td>
</tr>
<tr>
<td>Recall</td>
<td>The average recall (the percentage of the total relevant elements retrieved by the pattern).</td>
</tr>
<tr>
<td>Precision</td>
<td>The average precision (the percentage of relevant elements in relation to the number of total elements retrieved).</td>
</tr>
<tr>
<td>Value</td>
<td>The value or relevance this pattern has for the stakeholder “domain expert” (given as --, -, 0, +, ++).</td>
</tr>
<tr>
<td>Transition</td>
<td>The input and output level for the pattern in the conceptual model.</td>
</tr>
<tr>
<td><strong>Long Description</strong></td>
<td>A longer description of the pattern, including keywords.</td>
</tr>
<tr>
<td>Related Pattern</td>
<td>A list of other patterns this pattern is related to (e.g. composed of these pattern).</td>
</tr>
<tr>
<td>Example</td>
<td>An example for an input element where the pattern holds and the elicited result.</td>
</tr>
</tbody>
</table>
### CaVE - Pattern Example (1)

<table>
<thead>
<tr>
<th>Name</th>
<th>Headings</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>Headings usually represent features</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>Headings</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>Feature</td>
</tr>
<tr>
<td><strong>Recall</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>++</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>-</td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>Transition Documentation -&gt; Product Line Artifact</td>
</tr>
<tr>
<td><strong>Long Description</strong></td>
<td>Since features describe functionalities that are of importance for the user, they are found at prominent places in the user documentation.</td>
</tr>
<tr>
<td><strong>Related Pattern</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Example</strong></td>
<td>In a mobile phone user documentation “Sending an SMS” is a heading that describes a feature</td>
</tr>
</tbody>
</table>
CaVE - Pattern Example (2)

<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter-Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>If Sentences or Phrases are identical in different documents but include a different numerical value, this can be a parametrical variability or alternative values</td>
</tr>
<tr>
<td>Input</td>
<td>Sentence; phrase</td>
</tr>
<tr>
<td>Output</td>
<td>Alternative</td>
</tr>
<tr>
<td>Recall</td>
<td>-</td>
</tr>
<tr>
<td>Precision</td>
<td>++</td>
</tr>
<tr>
<td>Value</td>
<td>+</td>
</tr>
<tr>
<td>Transition</td>
<td>user documentation -&gt; requirements concept</td>
</tr>
<tr>
<td>Long Description</td>
<td>Parameters in the systems that have a different value could be realized differently in the software</td>
</tr>
<tr>
<td>Related Pattern</td>
<td></td>
</tr>
<tr>
<td>Example</td>
<td>“The phone can send SMS with at most 124 characters” &lt;-&gt; “The phone can send SMS with at most 136 characters”</td>
</tr>
</tbody>
</table>
CaVE - Summary

• Pro:
  – splits work between product line engineer, an (external) expert in product line development, and the internal problem space experts, thus reduces work load for domain experts
  – well documented with industrial case studies in [CaVE]

• Con:
  – no (documented) experiences with larger and/or very diverse documentation
  – almost no tool support available
Transition Steps

1. Pre-SPL
2. Product Relation Pattern Matching
3. Transition Scenario Identified
4. Variability Analysis
5. Problem Space Model Building
6. SPL
Problem Space Modelling – Feature Modelling

• feature models are used most often because:
  
  – the feature model concept is easy to understand

  – even complex feature models are easy to navigate

  – have enough expressive power for most real-world problems
Transition Steps

1. Pre-SPL
2. Product Relation Pattern Matching
3. Transition Scenario Identified
4. Solution Space Variability Analysis
5. Model Building
6. SPL
Solution Space VP Extraction

- Typical inputs:
  - architecture design documents
  - source code
  - configuration scripts
  - project build descriptions (e.g. makefiles)
  - version control system structure and usage
  - configuration files and file formats
# Solution Space VP – Pattern #1

<table>
<thead>
<tr>
<th>Name</th>
<th>#1 Preprocessor-based Text Fragment Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>If source code files contain preprocessor statements which conditionally (de)activate program statements, this usually identifies a variation point</td>
</tr>
<tr>
<td>Input</td>
<td>source file</td>
</tr>
<tr>
<td>Output</td>
<td>selection conditions and functional differences</td>
</tr>
<tr>
<td>Environment</td>
<td>C/C++ code, frame/macro processor input files, code generator input</td>
</tr>
<tr>
<td>Precision</td>
<td>+</td>
</tr>
<tr>
<td>Value</td>
<td>o</td>
</tr>
<tr>
<td>Transition</td>
<td>source code -&gt; variation point</td>
</tr>
<tr>
<td>Long Description</td>
<td>The textual exchange of program statements is an often used concept to encapsulate platform specific code or to optionally add functionality statically during compile time.</td>
</tr>
<tr>
<td>Related Pattern</td>
<td>#2, #6</td>
</tr>
<tr>
<td>Example</td>
<td></td>
</tr>
</tbody>
</table>
  ```c
  #ifdef FEA_TEMP_ALARM
      if (t > t_al) send_alarm(TEMP_ALARM, t);
  #endif
  ``` |
### Solution Space VP – Pattern #2

<table>
<thead>
<tr>
<th>Name</th>
<th># 2 Constant/variable based configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>If a source code file contains almost exclusively constant or variable definitions, it may be used for configuration purposes.</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>source file</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>selection conditions and functional differences</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>All programming languages</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>source code -&gt; variation point</td>
</tr>
<tr>
<td><strong>Long Description</strong></td>
<td>Constants defined in a file/files referenced by many other files usually provide access to shared information such as compile and/or runtime configuration</td>
</tr>
<tr>
<td><strong>Related Pattern</strong></td>
<td>#1</td>
</tr>
</tbody>
</table>

**Example**

C header files:
```
#define FEA_TEMP_ALARM 1
const int max_buffer_size = 512;
int max_buffer_count = 8;
```
<table>
<thead>
<tr>
<th>Name</th>
<th># 3 Use of Build Configurations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short Description</td>
<td>IDE or build tool uses configuration files to produce variants</td>
</tr>
<tr>
<td>Input</td>
<td>build configuration information</td>
</tr>
<tr>
<td>Output</td>
<td>alternative oo hierarchy and/or functional differences</td>
</tr>
<tr>
<td>Environment</td>
<td>IDE or build tool like make, ant</td>
</tr>
<tr>
<td>Precision</td>
<td>++</td>
</tr>
<tr>
<td>Value</td>
<td>++</td>
</tr>
<tr>
<td>Transition</td>
<td>configuration file -&gt; structural and/or functional variability</td>
</tr>
<tr>
<td>Long Description</td>
<td>IDE or project build tools such as make can be parametrized and thus produce different output for the same input (file structure).</td>
</tr>
<tr>
<td>Related Pattern</td>
<td>#6</td>
</tr>
<tr>
<td>Example</td>
<td>make could be used with different makefiles/make variable settings as input to generate different builds of a project. The differences in makefiles may be stored in different files or as different versions&amp;branches of same file in version control systems. Differences in parameters are often visible in build scripts.</td>
</tr>
</tbody>
</table>
## Solution Space VP – Pattern #4

<table>
<thead>
<tr>
<th>Name</th>
<th># 4 Conditional code execution</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>Conditional code execution is controlled by configuration values</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>source code</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>conditions for alternative state flow/event processing/…</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>All programming languages</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>0</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>0</td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>source code -&gt; functional variability</td>
</tr>
</tbody>
</table>

**Long Description**

Some conditional execution paths are controlled by configuration data instead of user data. Indicators are that the condition references configuration constants or uses methods to access configuration values such as getters for global configuration objects.

```java
if (config.getValue("TEMP_ALARM" && t > t_al) {
    alarm.send("TEMP_ALARM",t);
}
```
#5 Customer-specific product variants are managed using branches in version control systems

<table>
<thead>
<tr>
<th><strong>Short Description</strong></th>
<th>#5 Customer-specific product variants are managed using branches in version control systems</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong></td>
<td>version control system usage policy / version structure</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>list of customer specific product variants</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>version control system</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>o</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>version branches -&gt; product variants</td>
</tr>
</tbody>
</table>

Customer-specific product variants are often created by copying parts or whole software systems into separate branches in version control systems. Often naming of branches reveals such a policy. Also combined with labeling of version representing product variants.

### Example
- BMW_V1
- PSA_V1
- VW_V2
<table>
<thead>
<tr>
<th><strong>Name</strong></th>
<th># 6 File level variation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>Different files with similar names are used as alternative implementations</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>file name list</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>variation point</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>any language</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>+</td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>files -&gt; variant point alternatives</td>
</tr>
</tbody>
</table>

**Long Description**

For implementation of static code variability where the code changes significantly between variants, often each variant is placed in a separate file. The configuration mechanism selects one of the files at compile time latest.

**Related Pattern**

#1, #3

**Example**

List of file names:
- temp_normal.c
- temp_alarm_sound.c
- temp_alarm_visual.c
# Solution Space VP – Pattern #7

<table>
<thead>
<tr>
<th>Name</th>
<th># 7 Configurable Factory object</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Short Description</strong></td>
<td>The software uses the factory pattern and the factory object is configurable</td>
</tr>
<tr>
<td><strong>Input</strong></td>
<td>factory pattern members</td>
</tr>
<tr>
<td><strong>Output</strong></td>
<td>variation point and variation point constraints</td>
</tr>
<tr>
<td><strong>Environment</strong></td>
<td>any language</td>
</tr>
<tr>
<td><strong>Precision</strong></td>
<td>o</td>
</tr>
<tr>
<td><strong>Value</strong></td>
<td>o</td>
</tr>
<tr>
<td><strong>Transition</strong></td>
<td>architecture pattern -&gt; variant point alternatives</td>
</tr>
</tbody>
</table>

**Long Description**
The factory pattern is used to decouple the act of object creation from the place where the object is created. If the type or implementation of the factory object is changeable by configuration, it may represent a variability.
Solution Space VP Detection

- In many cases looking at a small amount of code reveals used patterns. (Programmers tend to repeat themselves)

- Since often embedded in syntactically restricted code structures, solution space patterns can be more easily detected using (simple) tools.

- The amount of detected variation point candidates is often huge.
Pattern #1

Standard Unix tools used only (also for Win32)

Find and count all \#if[n]?def/#elif (#define) statements:

```
find  -name "*.h[cl]*" -exec egrep "(#if[n]?def|elif)" {} \; | awk '{ print $2 }' |sort | uniq -c | sort -r >ifdef.lst
```

Analyse name intersection:

```
awk '{ print $2 }' ifdef.lst defines.lst | sort | uniq -d
```

Get all files where \#if[n]?def/#elif is used

```
ifnames `find -name "*.h[cl]*``
CVSNT (a cross platform re-implementation of CVS)

- 327 KLOC
- 1079 different constant used in #if* conditions
- 3775 different #define constants
- Constants defined in
  - makefiles
  - (generated) header-files
  - implementation files
  - Visual Studio Project files

Top 20 #if* constants

<table>
<thead>
<tr>
<th># Occur.</th>
<th>Name</th>
<th>Defined internally</th>
</tr>
</thead>
<tbody>
<tr>
<td>154</td>
<td>_WIN32</td>
<td></td>
</tr>
<tr>
<td>124</td>
<td>___cplusplus</td>
<td>yes</td>
</tr>
<tr>
<td>109</td>
<td>SERVER_SUPPORT</td>
<td>yes</td>
</tr>
<tr>
<td>74</td>
<td>SUPPORT_UTF8</td>
<td>yes</td>
</tr>
<tr>
<td>61</td>
<td>XML_DTD</td>
<td>yes</td>
</tr>
<tr>
<td>52</td>
<td>DEBUG</td>
<td>yes</td>
</tr>
<tr>
<td>43</td>
<td>WIN32</td>
<td></td>
</tr>
<tr>
<td>33</td>
<td>emacs</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>IPV6</td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>SUPPORT_UCP</td>
<td>yes</td>
</tr>
<tr>
<td>25</td>
<td>HAVE_CONFIG_H</td>
<td>yes</td>
</tr>
<tr>
<td>25</td>
<td>CVS95</td>
<td>yes</td>
</tr>
<tr>
<td>24</td>
<td>_DEBUG</td>
<td>yes</td>
</tr>
<tr>
<td>24</td>
<td>USE_SHARED_LIBS</td>
<td>yes</td>
</tr>
<tr>
<td>23</td>
<td>XML_NS</td>
<td>yes</td>
</tr>
<tr>
<td>21</td>
<td>UTIME_EXPECTS_WRITABLE</td>
<td>yes</td>
</tr>
<tr>
<td>19</td>
<td>CVSGUIPIPE</td>
<td>yes</td>
</tr>
<tr>
<td>14</td>
<td>_UNICODE</td>
<td>yes</td>
</tr>
</tbody>
</table>
Solution Space VP Analysis Tools

DMS (SemanticDesign): rule based language analyser and transformer, many languages

PUMA*: C/C++ parser and manipulator (part of AspectC++)

AspectJ*: Java, pointcuts and “declare”

Eclipse JDT*/CDT*: internal analysers, good analysis quality esp. for Java

* open source/freeware
Transition Steps

1. Pre-SPL
2. Product Relation Pattern Matching
3. Transition Scenario Identified
4. Variability Analysis
5. Solution Space Model Building
6. SPL
Solution Space – Architecture

- reference architecture
  - recovery
    - product gang, product bush
  - building
    - product forest

- core asset identification
  - weight identified architectural components by
    - expected use in future products
    - measured use in existing products
    - expected effort for adaptation
  - not all existing assets have to be used!
Production Plan: Problem – Solution Mapping

- problem space decisions do not always have one-to-one mapping into solution space variation points

- a problem space to solution space mapping is required to select/configure core assets from problem space configurations

- pure::variants' family models are used to model solution space and the mapping, often in combination with other tools
Production Plan Automation: Tool Support

- **Script-based configuration tools**
  
  `autoconf*`: Unix-specific configuration tool, especially for multi-platform development with C/C++ language

- **MDSD tools**
  
  `MetaEdit+`: powerful domain specific modelling tool

  `openArchitectureWare*`: powerful java-based model transformer and code generator, generates with appropriate input model variant-specific code

- **Product Line tools**
  
  `pure::variants`: explicit solution space modelling with integrated/external model transformation
Management Aspects of Software Product Lines
Transition to Variant Management

• Transition does not happen in a single moment; it's a process.

• Choosing the right approach depends on:
  - history and expected future
  - available funding
  - required time to market
  - skill level in the organisation
Approaches: Platform-Centric

- Project 1
- Project 2
- Project 3
- Project 4
- Project 5
- Project 6

PL Team

PL Core Assets

Time
Approaches: Incremental Platform-Centric

- Project 1
- Project 2
- Project 3
- Project 4
- Project 5
- Project 6

PL Core Assets

PL Team

Time
Approaches: Project-Centric

- Project 1
- Project 2
- Project 3
- Project 4
- Project 5
- Project 6

PL Management Team

PL Core Assets

Time
<table>
<thead>
<tr>
<th>Approach Comparison</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Platform-Centric</strong></td>
</tr>
<tr>
<td>+ Pro:</td>
</tr>
<tr>
<td>- lowest influence from projects</td>
</tr>
<tr>
<td>- no legacy</td>
</tr>
<tr>
<td><strong>Con:</strong></td>
</tr>
<tr>
<td>- organisational impact</td>
</tr>
<tr>
<td>- slow start</td>
</tr>
<tr>
<td>- highest risk</td>
</tr>
</tbody>
</table>

© pure-systems GmbH 2007
all you need for product lines  www.pure-systems.com
Summary

- Avoid variability, compare cost of variation to benefit

- If it is economically feasible/necessary to introduce variability make variation points and variants first level elements of your approach

- Explicit variant and variability management is the key to successful reuse

- Don't try to do all at once, incremental adoption lowers both effort and risk

- If you do it for the first time, ask someone who has done this before
Summary

• Transforming existing legacy software into product lines is possible
  
  – before starting a transition, careful risk assessment is necessary

• Results depend on:
  
  – amount and quality of extractable knowledge,
  
  – skills and management support of transition team.
References & Tool Links


AspectJ: www.eclipse.org/aspectj
autoconf: www.gnu.org/software/autoconf
DMS: www.semanticdesigns.com
Eclipse CDT: www.eclipse.org/cdt
openArchitectureWare: architecturware.sourceforge.net
PUMA: www.aspectc.org
pure::variants: www.pure-systems.com
Linking Variants with Defects and Tests
The Challenge

In variant rich systems not every test or defect is relevant for all variants.

The following questions have to be answered:

- Which tests, defects, change requests are relevant for which variants?

- What is the current development state regarding those items for the individual variants?

The approach should be applicable to existing databases and tools.
The pure::variants Approach

- pure::variants is used to manage the variability of the products. Each relevant product is described by a variant model.

- Each pure::variants element (feature, component, part, source) can be linked to external items using target list attributes.

- For each to be connected tool or database a pure::variants connector plugin resolves these links on demand in real-time.

- Depending on the tool the connector can also provide actions like creation of a new defect or test case from within pure::variants.
pure::variants – Sample Workspace

Full Text Bug Listing

Bug 19 - LCD does not display values for pressure when serial connection is active

Bug#: 19  Product: TestProduct  Version: other  Platform: PC
OS/Version: Linux  Status: REOPENED  Severity: normal  Priority: P2
Model with Test and Defect Information

Markers show current defect/test state of elements (here an open bug and a failing test case). Succeeded tests and resolved defects are not shown.

Elements are linked to test case or defects using special list attributes.

Restrictions can be used to make a defect or test link valid only under defined conditions (here the selection of two features).
Relations View – Quick Target Info Access

Relations view lists related relation targets for the selected element. For defects and tests the summary is shown.

Additional information is accessible via the tool tip and optionally in specialized views.
Matrix View – Element Selection States

- The Matrix view enables quick overview for multiple variants. Here the selection is shown (default visualization).
- Each column represents a variant, rows are selectable elements like features.
- Other visualizations can show variant specific defect and test states.
Matrix View – Element Test States

- The test state visualization shows for all selected features the success of the related tests (indicated by different icons).
- It is visible here that not all variants have the same set of tests.
Matrix View – Element Defect States

- The defect state visualization shows for all selected features the existence of the related open defects (indicated by different icons).
- It is visible here that not all variants have the same set of defects.
pure::variants – Extension Availability

• The Connector API has been published with release 2.4
  – The API is available for public use by third parties

• pure-systems provides the following Connectors as commercial products for release 2.4:
  – pure::variants Connector for ClearQuest
  – pure::variants Connector for Bugzilla
Connecting pure::variants with Model-based Tools
Connection Approaches

• Read and Generate
  – External model is fully read into pure::variants model
  – pure::variants allows for modification and addition of rules etc.
  – pure::variants generates variant-specific external model from p::v model
  – Examples: Connector for Simulink

• Link and Communicate
  – only variation point related information is extracted from external source
    • possibly most information is entered in external tool
  – pure::variants actuates variation points during configuration
  – relevant configuration for tool is communicate to tool (online/offline)
  – Examples: Connector for Doors, new Simulink Configurator
Simulink Model with Variation Points
Simulink Variation Point Setting

```
>> VAR_FreezeingProtection

VAR_FreezeingProtection =

    0

>>
```
pure::variants based Simulink Configurator
Simulink Variation Point Setting

Command Window

```matlab
>> VAR_FreezingProtection

VAR_FreezingProtection =

   0

>> VAR_FreezingProtection

VAR_FreezingProtection =

   1

>> |
```