Software Product Lines and Code Generation: A Perfect Match?

Danilo Beuche
pure-systems GmbH
Germany
## Tutorial Structure

<table>
<thead>
<tr>
<th>Part</th>
<th>Topics</th>
<th>Duration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>Scope, Terminology</td>
<td>25 min</td>
</tr>
<tr>
<td>Problem Space</td>
<td>Building (Feature) Models</td>
<td>10 min</td>
</tr>
<tr>
<td>Solution Space</td>
<td>Generative Programming, Aspect-Oriented Programming, Model-Driven Development</td>
<td>30 min</td>
</tr>
<tr>
<td>Summary</td>
<td></td>
<td>10 min</td>
</tr>
</tbody>
</table>
A Few Words of Warning

This talk

- will show how software product line development uses and benefits from code generation techniques

- will not tell you the full story, but focus on a few ideas we or our clients have been applying in practice

- will not be objective (being from a software product line expert)
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.“
Systematic Handling of Variants and Variability

- **Process Level**
  - variant management throughout product lifecycle
  - selection of suitable methods

- **Method Level**
  - description of variability
  - design of variable and flexible architectures
  - selection of suitable tools

- **Tool Level**
  - automation
  - quality assurance
Terminology: Basics (1)

- **Software Product Line** terminology is used throughout the talk:
  - Problem Space vs. Solution Space
  - Domain Engineering vs. Application Engineering
  - Variation Point
  - (Core) Assets
  - ...

...
Terminology: Basics (2)

Domain Engineering

Problem Space
- PL Properties and Relations

Solution Space
- Solution Assets

Product Properties

Product

Single Problem

Single Solution

Application Engineering
Terminology: Roles

Problem Space
- Domain Expert
- Application Analyst
- Single Problem

Solution Space
- PL Architect and Developers
- Application Developer
- Single Solution

Domain Engineering
- PL Expert

Application Engineering
Terminology: Product Derivation

Domain Engineering

Problem Space

Solution Space

Core Assets: Feature Models

Core Assets: Models, Code, ...

Production Plan

Product Asset: Feature Configuration

Product Assets: Code, Binary, ...

Single Problem

Application Engineering

Single Solution
**Terminology: 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`
Summary

• Software Product Line Development is about
  
  − Rationalization of product development for similar-but-not-identical products
  
  − Coordination of evolution of products and their implementations
  
  − Equal treatment of domain engineering and application engineering
Problem Space: Modeling

- problem space modeling puts together isolated variation points and finally defines the product line's problem space

Modeling approaches:

- variable use cases
- variable textual requirements
- decision tables (PuISe/IESE) [5]
- feature models (FODA, FORM, ...) [6]
- domain specific modeling languages
Problem Space: Feature Modelling

• feature models are “en vogue” 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

• but
  – there is no standard yet (especially for graphical notation)
  – specialized tool support is required for effective use of (larger) feature models
  – cannot be used to express all problem space concepts
Feature Models: What is a Feature?

- an abstract concept
- an entity property relevant for a stakeholder
- could be
  - a requirement
  - a technical function/function group
  - a non-functional property
- concrete instantiation of this concept has to be defined before defining features
  - what entity is to be described?
  - who are stakeholders?
  - what will the features being used for?
Feature Models: What is a Feature Model?

- an abstract concept
- a formalized description of variability in terms of features relevant for a stakeholder
- always used for communication
- could be used for
  - domain analysis and scoping
  - system development
  - entity configuration
- concrete instantiation of this concept has to be defined before defining features
  - what purpose shall the models serve?
  - who are stakeholders?
Weather Station Variants

- Thermometer: LCD, Temperature
- Home: LCD, Temperature, Pressure
- Outdoor: LCD, Temp., Pressure, Wind

- Deluxe variants: + PC Connection
- PC-only variants: + PC Connection - LCD

- Serial PC Connection
- USB PC Connection
- ...
Feature Models: Weather Station
Feature Models: Tools

- spread sheet programs
  - decision table, product-feature matrix
  - examples: MS Excel, OpenOffice Calc*

- mind map tools
  - feature models, any kind of tree structured info
  - examples: MindManager, Freemind*

- specialized product line tools
  - feature models
  - examples: pure::variants, FeaturePlugin*, XFeature*

* open source/freeware
Summary

- Feature models are a way to easily model variability in the problem space but not the only one.

- They do not describe a particular way of implementing those features.

- Feature configurations provide information about the particular set of features a product line instances has to provide.
Software Product Lines and Code Generation: A Perfect Match?

Solution Space
Overview

• introduction
  – application-driven vs. feature-driven product derivation

• feature-driven generative programming
  • frame processors
  • policy-based design
  • aspect-oriented programming
  • model-driven software development
Platform Tailoring

- a transformation process with several steps
  - code selection
  - generation
- at each level features can be realized or excluded

**Diagram:***
- **Application**
- **Running platform instance**
- **user, events**
- **state transitions**
- **Generation**, **Compilation**, **Linking**, **Loading**

**Application-specific code and models**

**PL platform**
Application-Driven Product Derivation

- traditional approach
- the application code directly drives the tailoring process, e.g. ...
  - in the running system
    - object-oriented and component-oriented composition techniques
    - global state variables
  - at link time
    - "function-level linking"
  - at compile time
    - using "inline functions"
    - using only parts of the offered class hierarchy
    - template instantiation
Application-Driven Product Derivation

- although simple and well-understood there are drawbacks ...
  - complex platform interfaces
  - low abstraction level
  - platform tailoring and application development cannot be separated
  - overhead of run time selection and composition mechanisms
  - missing ability to tailor internals
  - limitations due the programming language implementation
Feature-Driven Product Derivation

... introduces another level of indirection and abstraction

concrete problem

application-specific code / models

application developer

f2 ✓
f6 ✓
...

platform customizer

required features

solution space

PL platform

required API subset

product derivation techniques

concrete solution

PL platform instance

required features

problem space

domain expert

features and dependencies
Feature-Driven Product Derivation

- many advantages over purely application-driven product derivation
  - an additional transformation step of arbitrary complexity
    - internals, non-functional features, more generative power
  - separation of application programming and platform tailoring
    - customization for a class of applications
  - reduced API complexity
    - improved compilation times, no overhead (focus on static configuration)
  - separation of concerns
    - configuration logic is separated from application code, separate re-use
- yet requires a feature model and tool support
- co-exists with application-driven product derivation
Feature-Driven
Generative Programming (1)

- code generators are versatile and powerful tools
  - code patterns are either built-in or provided separately
  - input drives the generation process

**patterns**
(code or description)

- x-frames
- C++ template
- aspects
- templates

**input**
(code or description)

- specification x-frame
- instantiations
- pointcuts
- model

**output**
(generated code)

- xvcl
- C++ compiler
- aspect weaver
- generator
by combining GP with rule-based asset selection or configuration we gain "generative power"

patterns
(code or
description)

input
(code or
description)

output
(generated code)

generator

examples
x-frames
C++ template
aspects
templates

specification x-frame
instantiations
pointcuts
model

xvcl
C++ compiler
aspect weaver
generator

feature selection

$f_2$
$f_6$
...
What is a Frame?

„When one encounters a new situation (or makes a substantial change in one's view of a problem) one selects from memory a structure called a frame. This is a remembered framework to be adapted to fit reality by changing details as necessary.“

M. Minsky
A Framework for Representing Knowledge
1975
Example: the XVCL Frame Processor

- XML-based Variant Configuration Language [2]
  - Frame are XML-tagged text files

XVCL Processor

-x-framework

Composition and Adaptation

SPC

configured components

specification x-frame

component

component

component
Example: the XVCL Frame Processor

- XML-based Variant Configuration Language [2]
  - Frame are XML-tagged text files

XVCL Processor

- Composition and Adaptation
- SPC
- Specification x-frame
- Concrete problem
- Configured components
- Concrete solution
x-frames consist of
- XML tags
- arbitrary text

With adapt another frame can be processed
Output:

AAA before
BBB before
DDD
BBB
EEE
CCC before
EEE
CCC
FFF
CCC after
AAA
AAA after
XVCL – File Generation

- normally *adapt* works like `#include`
  - composition of adapted text fragments
- in combination with `outfile` the frame hierarchy can be used to generated various files

A.cc:
AAA before
BBB
AAA
AAA after

C.h:
CCC
by using \textit{break} and \textit{insert} (-before, -after) higher-level frames can insert text into lower-level frames at predefined positions
variables, loops, and conditions can be used for flexible text generation
XVCL Example: Adaptive Manual

- the user's manual is adapted according to the selected features

**User's Manual**

1. **Sensors**
   Sensors are needed to measure weather data.
   
   1.1 **Wind Speed**
   The wind speed measurement is important to avoid the loss of umbrellas ...

2. **Display**
XVCL Example: Sensor Section Frame

- configurable manual sections are implemented as x-frames
  - the variable parts are left open – just marked as a "break"

```xml
<?xml version="1.0"?>
<!DOCTYPE x-frame SYSTEM "DEFAULT">
<x-frame name="sensors">
  <![CDATA[<h2>Sensors</h2>]]>
  Sensors are needed to measure weather data.

  <break name="sensordesc"/>

</x-frame>
```
XVCL Example: SPC Adaptation

- with pure::variants the specification x-frame can easily be generated/adapted by "conditional XML tags"

```xml
<x-frame name="manual" outfile="Weather Station.html">
  <![CDATA[ <html> ]]>  
  <adapt x-frame="head"/>
  <adapt x-frame="intro"/>
  <adapt x-frame="sensors">
    <insert break="sensordesc">
      <adapt x-frame="temp" condition="pv:hasFeature('demo_wm_Temperature')"/>
      <adapt x-frame="wind" condition="pv:hasFeature('demo_wm_Wind')"/>
      <adapt x-frame="pressure" condition="pv:hasFeature('demo_wm_Pressure')"/>
    </insert>
  </adapt>
  <adapt x-frame="display" condition="pv:hasFeature('demo_wm_Display')"/>
  <adapt x-frame="pcconnection" condition="pv:hasFeature('PC_Connection')"/>
  <adapt x-frame="warranties"/>
  <![CDATA[ </html> ]]>  
</x-frame>
```
Rule-Based Asset Configuration

- generative power: low, depends on configuration tool
- suitability for legacy code migration: medium
- granularity: intra asset
- useful applications
  - fine-grained adaptation, e.g. within a function
  - adaptation of non-code artifacts, e.g. user's manual
  - save resources by applying static instead of dynamic configuration
- common pitfalls
  - configurable assets depend on a tool and an environment, thus re-use becomes more complicated
  - not usable for "crosscutting concerns"
Feature-Driven Policy Selection

- **Policy-Based Design** [3], Alexandrescu, 2002
- idea: avoid hard coded type relations when postponement of design decisions makes sense
- code patterns: generic code, C++ templates
- input: type instantiations
- terminology:
  - **host class** – a parameterized (generic) type
  - **policy class** – a class or class template that is used as parameter; implements a configurable concern in a specific way
  - **policy** – the interface, which the host class expects; all policy classes have to implement this interface
a class template 'NewCreator' that creates new objects of any type

- however, using 'new' might not be appropriate in any case

```cpp
template <class T>
class NewCreator {
public:
    static T *create() {
        return new T;
    }
};
...
Widget *w = NewCreator<Widget>::create();
```
'FastPoolCreator' alternatively allocates objects in a special memory pool

```cpp
template <class T>
class FastPoolCreator {
public:
    static T *create() {
        void *buf = FastPool::malloc(sizeof(T));
        if (!buf) return 0; // or throw an exception
        return new (buf) T; // placement new
    }
};
...

Widget *w = FastPoolCreator<Widget>::create();
```
Policy-Based Design – Example (3)

- 'NewCreator' and 'FastPoolCreator' are policy classes
  - they implement a 'CreationPolicy' by providing a 'create' function
- the policy can now be used by host classes

```c++
template <template <class> class CreationPolicy>
class List {
    void newElement() {
        ListElement *e = CreationPolicy<ListElement>::create();
    }
    ... 
};

// code generation is triggered by template instantiation
typedef List<FastPoolCreator> FastList;
```
Feature-Driven Policy Selection

- generative power?
  - consider a host class that uses 3 policies
  - each policy is implemented by 3 policy classes
    ➔ implemented: 10 classes (host + 3x3 policy classes)
    ➔ we can instantiate 27 (= $3^3$) different variants of the host!

- by feature-driven configuration of a template instantiation
  we can exploit the generative power for product derivation
Feature-Driven Policy Selection

- generative power: high
- suitability for legacy code migration: low
- granularity: policy configuration
- useful applications
  - static combination of orthogonal features
    - avoids the combinatorial explosion of the number of classes
  - feature-driven selection hides the complexity of template libraries from the application
- common pitfalls
  - code bloat
  - complexity of C++ templates

- Kiczales, 1997
- aspects support the modular implementation of "crosscutting concerns"
- supported by several languages and frameworks
  - AspectJ, AspectC++, Spring, JBoss, ...
- benefits for PL development
  - decoupling of components
  - better mapping of features to assets
  - simple but powerful configuration tool
Aspect-Oriented Programming (AOP)

- ...provides means for a modular implementation of cross-cutting concerns

- examples: tracing, synchronization, security, buffering, error handling, checking of contracts and invariants, ...

![Diagram showing well-modularized concern with and without AOP]

- with AOP
  - Aspect
- without AOP
  - badly modularized
  - well-modularized concern
AspectC++ Language Extension

- AspectC++ extends C++ by typical AOP features
  - GPL, download from [http://www.aspectc.org](http://www.aspectc.org)

```cpp
aspect ElementCounter {
  advice call("% Queue::enqueue(...)") : after() {
    std::cout << "Queue::enqueue" << std::endl;
  }
  ...
}

ElementCounter.ah
```
General AOP Benefits

- **reusability**
  - component code can be used with or without aspects

- **readability**
  - AOP avoids code duplication
  - components implement only the core functionality

- **extensibility**
  - aspects and component code can be evolved separately
  - well-implemented aspects automatically affect even future extensions

- **quality**
  - policies can often be directly expressed and enforced
```c
int main() {
    Weather data;
    Sink    sink;

    while(true) {
        // acquire data
        data.measure();

        // process data
        sink.process(data);

        wait();
    }
}
```

Weather Station: Functional Decomposition

- `Weather::measure()`
- `Sink::process()`
- `Pressure::measure()`
- `Wind::measure()`
- `Temperature::measure()`
- `process_data(Pressure)`
- `process_data(Wind)`
- `process_data(Temperature)`
Sensor Integration

Weather::measure() {
    _pressure.measure();
    _wind.measure();
    _temp.measure();
}

Sink::process(const Weather& w) {
    process_data(w._pressure);
    process_data(w._wind);
    process_data(w._temp);
}

```cpp
Weather::measure() {
    _pressure.measure();
    _wind.measure();
    _temp.measure();
}

Sink::process(const Weather& w) {
    process_data(w._pressure);
    process_data(w._wind);
    process_data(w._temp);
}
```
Sensor Integration

Weather::measure() {
    _pressure.measure();
    _wind.measure();
    _temp.measure();
}

Sink::process(const Weather& w) {
    process_data(w._pressure);
    process_data(w._wind);
    process_data(w._temp);
}

...crosscuts the modules
Sensor Integration with Aspects

```cpp
Weather::measure() {
}

Sink::process(const Weather& w) {
}
```

- **Weather**
  - `measure()`

- **Sink**
  - `process(Weather)`

- **Pressure Handling**
  - **Pressure**
    - `measure()`
  - **Wind**
    - `measure()`
  - **Temperature**
    - `measure()`
Sensor Integration with Aspects

```cpp
Weather::measure() {
}

Sink::process(const Weather& w) {
}
```

```
Pressure
measure()

Wind
measure()

Temperature
measure()
```

```
Weather
---
measure()

Sink
---
process(Weather)
```

```
Pressure Handling
Wind Handling
```

```cpp
Sink::process(const Weather& w) {
}
```
Sensor Integration with Aspects

Weather::measure() {
}

Sink::process(const Weather& w) {

Pressure::measure()
Wind::measure()
Temp::measure()

Sink
process(Weather)
Sensor Integration with Aspects

Weather::measure() {
  _pressure.measure();
  _wind.measure();
  _temp.measure();
}

Sink::process(const Weather& w) {
  process_data(w._pressure);
  process_data(w._wind);
  process_data(w._temp);
}

loose coupling of sensor slices
Design Conclusions

By using aspects, we achieved...

- complete decoupling of components
  - component slices are merged in by advice
  - actors and sensors “integrate themselves”
  - not a single `#ifdef`
- Plug & Play of components

...without sacrificing efficiency

- minimal stack usage due to advice code inlining
- everything is resolved at compile-time
  - no dynamic data structures to manage sensors/actors
  - no virtual functions
...made easy

... is simple module selection
Aspect-Oriented Programming

- generative power: very high, even crosscutting concerns
- suitability for legacy code migration: high, no modifications
- granularity: high, at each join point
- useful applications
  - generic implementation of (optional) crosscutting concerns
  - decoupling of components
  - development support (tracing, profiling, etc.)
- common pitfalls
  - code bloat
  - use of "fragile pointcuts"
  - AOP without proper tool support
MDSD is about using models as the central artifact for software development
  - formal models specify (aspects of) systems
  - model transformers and code generators synthesize artifacts that can be run on a given execution platform

using this approach,
  - development efficiency is increased (less repetitive work)
  - software quality is increased (generators don't make accidental errors)
  - software architecture can be enforced (developers have less freedom for decisions)
  - value of the development artifacts is increased (models isolates description of aspects of the system -> better reuse)
openArchitectureWare (oAW)

- a suite of tools and components assisting with model driven software development implemented in Java.
- supports arbitrary import (model) formats, meta models, and output (code) formats. It can also work with Eclipse EMF
- tooling (such as editors and model browsers) is based on the Eclipse platform.
- the core of oAW consists of a workflow engine that allows the user to define transformation workflows including prebuilt workflow components for
  - reading and instantiating models
  - checking them for constraint violations
  - transforming them into other models and
  - for generating code.
Definition of **Code Generation Templates** in the Eclipse-based Template editor. Templates reference the metamodel underlying the model.

Definition of **Constraints**. These are boolean expressions that state invariants about the model. Constraints are based on the oAW expression language.
oAW Example: Metamodel for Datagram

- Ecore meta model describes the possible elements of a message
- Ecore meta model is used to generate editors and java classes
- variability is modeled as selection via the `condition` attribute of the `Item` element
oAW Example: Model of Datagram

- generated Eclipse editor enables modeling
- the **condition** property is used to define the rules for element selection
- conditions are evaluated by the feature modeling tool
- elements with FALSE condition are removed by the transformation
oAW Example: Generated Code from Model

- template generates code from customized model
- only temperature and air pressure was selected for this example

```cpp
#include "PCConnection.h"
#include "String.h"

FOREACH item AS i
#include "i.sensor.h"
ENDFOREACH

aspect SNGConnection :
  public PCConnection {
    FOREACH item AS i
    i.type m_i.sensor;
    ENDFOREACH

#include "PCConnection.h"
#include "String.h"
#include "Temperature.h"
#include "Pressure.h"

aspect SNGConnection :
  public PCConnection {
    UInt16 m_Temperature;
    UInt8  m_Pressure;
```
Model-Driven Software Development

- generative power: unlimited
- suitability for legacy code migration: low
- granularity: very high
- useful applications
  - generation of consistent implementations of interacting components
  - different implementations derived from same model
- common pitfalls
  - generating too complicated code not suitable for human debugging
  - model design too close to the implementation structure
Software Product Lines and Code Generation: A Perfect Match?

Summary
Summary: When to use what?

- Rule Based Configuration
- Policy
- AOP
- MDSD
Summary

- using code generation in product lines generally makes a lot of sense
- it is not necessary to generate whole applications
- there is a huge range of mechanisms to do the coupling, but no silver bullet
- first steps are simple so give it a try
- the percentage of generated code increases incrementally with previous successes
Thank you for your attention!