Konfigurierbare Systemsoftware (KSS)

VL 2 – Software Product Lines

Daniel Lohmann
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme
Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 12 – 2012-05-09

http://www4.informatik.uni-erlangen.de/Lehre/SS12/V_KSS

About this Lecture

Agenda

2.1 Motivation: The Quest for Variety
2.2 Introduction: Software Product Lines
2.3 Case Study: i4Weathermon
2.4 Problem Space
2.5 Solution Space
2.6 References
Model Car Industry: Variety of an BMW X3

- Roof interior: 90,000 variants available
- Car door: 3,000 variants available
- Rear axle: 324 variants available

("Varianten sind ein wesentlicher Hebel für das Unternehmensergebnis" – Franz Decker (BMW Group))

Model Car Industry: Variety Increase

- In the 1980s: little variety
  - Option to choose series and maybe a few extras (tape deck, roof rack)
  - A single variant (Audi 80, 1.3l, 55 PS) accounted for 40 percent of Audi’s total revenue

- Twenty years later: built-to-order
  - Audi: $10^{20}$ possible variants
  - BMW: $10^{32}$ possible variants
  - At average there are 1.1 equal instances of an Audi A8 on the street

Product lines with fully automated assembly

320 features

For each human being

Optional, independent features

More variants than atoms in the universe!
Typical Configurable Operating Systems...

1250 features

11000 features

Challenges

1. How to identify the actually desired variability?
2. How to express the intended variability?
3. How to implement this variability in the code?
4. How to map variability options to the code?

Agenda

2.1 Motivation: The Quest for Variety
2.2 Introduction: Software Product Lines
   Terms and Definitions
   SPL Development Process
   Our Understanding of SPLs
2.3 Case Study: i4Weathermon
2.4 Problem Space
2.5 Solution Space
2.6 References

Definition: (Software) Product Line, Feature

Product Line (Withey) (Definition 1)

"A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market.


Software Product Line (SEI) (Definition 2)

"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way.


Remarkable:
SPLs are not motivated by technical similarity of the products, but by feature similarity wrt a certain market.
**Definition: (Software) Product Line, Feature**

**Product Line (Withey) (Definition 1)**

"A product line is a group of products sharing a common, managed set of features that satisfy the specific needs of a selected market."


**Software Product Line (SEI) (Definition 2)**

"A software product line (SPL) is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way."


**Feature (Czarnecki / Eisenecker) (Definition 3)**

"A distinguishable characteristic of a concept [...] that is relevant to some stakeholder of the concept."

Czarnecki and Eisenecker 2000: Generative Programming. Methods, Tools and Applications [3, p. 38]

**Program Family (Definition 4)**

"Program families are defined [...] as sets of programs whose common properties are so extensive that it is advantageous to study the common properties of the programs before analyzing individual members."


- Most research on operating-system families from the '70s would today qualify as work on software product lines [2, 4, 5, 9–11]
- Program Family ⇒ Software Product Line
- However, according to the definitions, the viewpoint is different
  - Program family: defined by similarity between programs
  - SPL: defined by similarity between requirements
  - A program family implements a software product line
- In current literature, however, both terms are used synonymously
  - Program Family ⇔ Software Product Line

**SPL Development Reference Process** [1]

- **Domain Analysis**
- **Domain Design**
- **Domain Implementation**
- **Application Analysis**
- **Application Design**
- **Application Implementation**

- Configuration is the property that denotes the degree of pre-defined variability and granularity offered by a piece of system software via an explicit configuration interface.

- Common configuration interfaces
  - Text-based: `configure` script or `configure.h` file (GNU tools)
    - configuration by commenting/uncommenting of (preprocessor) flags
    - no validation, no explicit notion of feature dependencies
  - Tool-based: KConfig (Linux, busybox, CiAQ, ...), ecosConfig (eCos)
    - configuration by an interactive configuration editor
    - formal model of configuration space, hierarchical features
    - implicit/explicit validation of constraints

**Our understanding: Configurable System Software**

- **Configurability**
  - the property denotes the degree of pre-defined variability and granularity offered by a piece of system software via an explicit configuration interface.
The i4WeatherMon Weather Station

- A typical embedded system
  - Several, optional sensors
    - Wind
    - Air Pressure
    - Temperature
  - Several, optional actuators (here: output devices)
    - LCD
    - PC via RS232
    - PC via USB
- To be implemented as a product line
  - Barometer: Pressure + Display
  - Thermometer: Temperature + Display
  - Deluxe: Temperature + Pressure + Display + PC-Connection
  - Outdoor: <as above> + Wind
- ...
Agenda

2.1 Motivation: The Quest for Variety
2.2 Introduction: Software Product Lines
2.3 Case Study: i4Weathermon
2.4 Problem Space
   Domain Analysis
   Feature Modelling
2.5 Solution Space
2.6 References

Challenges

1. How to identify the actually desired variability?
2. How to express the intended variability?
3. How to implement this variability in the code?
4. How to map variability options to the code?

Domain Analysis

- Domain analysis
  - Existing code
  - Expert knowledge
- Domain model

Domain Scoping
- Selection and processing of domain knowledge
- Restriction of diversity and variety

Domain Modelling
- Systematic evaluation of the gained knowledge
- Development of a taxonomy

~ Domain Model (Definition 6)

“ A domain model is an explicit representation of the common and the variable properties of the system in a domain, the semantics of the properties and domain concepts, and the dependencies between the variable properties. ”

Czarnecki and Eisenecker 2000: Generative Programming, Methods, Tools and Applications [3]

Elements of the Domain Model

- Domain definition specifies the scope of the domain
- Examples and counter examples
- Rules for inclusion/exclusion of systems or features
- Domain glossary defines the vocabulary of the domain
- Naming of features and concepts
- Concept models describe relevant concepts of the domain
  - Formal description (e.g., by UML diagrams)
  - Textual description
  - Syntax and semantics
- Feature models describe the common and variable properties of domain members
  - Textual description
  - Feature diagrams
Domain Definition: i4WeatherMon

- The domain contains software for the depicted modular hardware platform. Future version should also support new sensor and actuator types (humidity, alarm, ...).
- The externally described application scenarios thermometer, barometer, outdoor, ... shall be supported.
- The i4WeatherMon controller software is shipped in the flash memory of the µC and shall not be changed after delivery.

Domain Glossary: i4WeatherMon

- **PC Connection**: Optional communication channel to an external PC for the sake of continuous transmission of weather data. Internally also used for debug purposes.
- **Sensor**: Part (1 or more) of the i4WeatherMon hardware that measures a particular weather parameter (such as: temperature or air pressure).
- **Actuator**: Part (1 or more) of the i4WeatherMon hardware that processes weather data (such as: LCD).
- **XML Protocol**: XML-based data scheme for the transmission of weather data over a PC Connection.

Concept Models: i4WeatherMon

- **XML Protocol**: The following DTD specifies the format used for data transmission over a PC Connection:

```
<!ELEMENT weather (ambient, outdoors, barometer)> ...
```

- **PC Connection** ...

Challenges

1. How to identify the actually desired variability?
2. How to express the intended variability?
3. How to implement this variability in the code?
4. How to map variability options to the code?
Feature Models

- Describe system variants by their commonalities and differences
- Specify configurability in terms of optional and mandatory features
- Intentional construct, independent from actual implementation

Primary element is the Feature Diagram:
- Concept (Root)
- Features
- Constraints

Complemented by textual descriptions
- Definition and rationale of each feature
- Additional constraints, binding times, ...

Feature Diagrams – Language

Syntactical Elements
A shallow dot  indicates a mandatory feature:
\[ V = \{(C, f_1, f_2)\} \]

(b) Optional features \( f_1, f_2 \) can be included if their parent feature \( C \) is selected.

(c) Mandatory feature \( f_1 \) has to be included if their parent feature \( C \) is selected.

Syntactical Elements
Of course, both can be combined:
\[ V = \{(C, f_1), (C, f_2)\} \]

(b) Optional features \( f_1, f_2 \) can be included if their parent feature \( C \) is selected.

(c) Mandatory feature \( f_1 \) has to be included if their parent feature \( C \) is selected.

Syntactical Elements
The filled dot  indicates a mandatory feature:
\[ V = \{(C, f_1), (C, f_2)\} \]

(b) Optional features \( f_1, f_2 \) can be included if their parent feature \( C \) is selected.

(c) Mandatory feature \( f_1 \) has to be included if their parent feature \( C \) is selected.

(d) Exactly one alternative feature \( f_1 \) or \( f_2 \) has to be included if the group’s parent feature \( C \) is selected.

(g) At least one cumulative feature \( f_1, f_2 \) has to be included if the group’s parent feature \( C \) is selected.

(h) Not used. Equivalent to (a).

(i) Not used. Equivalent to (b).
Syntactical Elements

The shallow arc \( \triangleright \) depicts a group of alternative features:
\[ V = \{ (C, f_1), (C, f_2) \} \]

- (a) Mandatory features \( f_1 \) and \( f_2 \) have to be included if their parent feature \( C \) is selected.

- (b) Optional features \( f_1 \) or \( f_2 \) can be included if their parent feature \( C \) is selected.

- (c) Mandatory feature \( f_1 \) has to be included if their parent feature \( C \) is selected, optional feature \( f_2 \) can be included if the group’s parent feature \( C \) is selected.

- (d) Exactly one alternative feature \( f_1 \) or \( f_2 \) has to be included if the group’s parent feature \( C \) is selected.

- (e) At most one optional alternative feature \( f_1 \) or \( f_2 \) can be included if their parent feature \( C \) is selected.

- (f) Not used. Equivalent to (a).

- (g) At least one cumulative feature \( f_1 \) or \( f_2 \) has to be included if the group’s parent feature \( C \) is selected.

- (h) Not used. Equivalent to (b).

- (i) Not used. Equivalent to (b).

I4WeatherMon: Feature Model

WeatherMon

Actuators

... Alarm Display

PC Connection

RS232Line USBLine Protocol

SNGProto ...

Air Pressure Wind Speed ...

dl KSS (VL 2 | SS 12) 2 Software Product Lines | 2.4 Problem Space
Challenges

1. How to **identify** the actually desired variability?
2. How to **express** the intended variability?
3. How to **implement** this variability in the code?
4. How to **map** variability options to the code?

I4WeatherMon: Reference Architecture

Functional decomposition (structure and process):

```c
int main() {
    Weather data;
    Sink sink;
    while(true) {
        // acquire data
data.measure();
        // process data
sink.process( data );
        wait();
    }
}
```

Implementation Techniques: Classification

- **Decompositional Approaches**
  - Text-based filtering (untyped)
  - Preprocessors

- **Compositional Approaches**
  - Language-based composition mechanisms (typed)
  - OOP, AOP, Templates

- **Generative Approaches**
  - Metamodel-based generation of components (typed)
  - MDD, C++ TMP, generators

Agenda

2.1 Motivation: The Quest for Variety
2.2 Introduction: Software Product Lines
2.3 Case Study: i4Weathermon
2.4 Problem Space
2.5 Solution Space
  - Reference Architecture
  - Implementation Techniques Overview
  - Variability Implementation with the C Preprocessor
  - Variability Implementation with OOP (C++)
  - Evaluation and Outlook
2.6 References
Components should be fine-grained. Each artifact should either be mandatory or dedicated to a single feature only. The use of memory/run-time expensive language features should be avoided as far as possible. Decide and bind as much as possible at generation time. Changing the set of optional features should not require modifications in any other part of the implementation. Feature implements should be able to "integrate themselves". The same should hold for new optional features, which may be available in a future version of the product line.

Conditional compilation with the C Preprocessor (CPP) is the standard approach to implement static configurability. Simplicity: the CPP "is just there". Economy: CPP-usage does not involve any run-time overhead. Prominent especially in the domain of system software (Linux 3.2: 85000 #ifdef Blocks → "#ifdef hell")

I4WeatherMon (CPP): Implementation (Excerpt)

```c
struct Weather {
    #ifdef cfWM_HUMID
    UInt16 _w;
    #endif
    #ifdef cfWM_PRESSURE
    UInt16 _p;
    #endif
    #ifdef cfWM_TEMPERATURE
    Int16 _t1;
    #endif
    #ifdef cfWM_STACK
    unsigned int_manstack;
    #endif
};
```
Sensor (and actuator) integration both crosscut the structure of the main program, an interaction with a mandatory feature.
### I4WeatherMon (OOP): Design (Excerpt)

#### Sensor
- `measure()`
- `name()`
- `unit()`
- `str_val()`
- `init()`

#### Actuator
- `before_process()`
- `after_process()`
- `process()`
- `init()`

#### ChainBase
- `_next`

#### Wind
- `id`
- `measure()`
- `name()`
- `unit()`
- `str_val()`
- `init()`

#### Pressure
- `id`
- `measure()`
- `name()`
- `unit()`
- `str_val()`
- `init()`

#### Display
- `#print()`
- `before_process()`
- `process()`
- `init()`

#### SNGConnection
- `before_process()`
- `after_process()`
- `process()`
- `init()`

### I4WeatherMon (OOP): Evaluation

#### General
- Separation of concerns (SoC) ✔
- Resource thriftiness ?

#### Operational
- Granularity
  - Every component is either a base class or implements functionality of a single feature only.
- Economy
  - Run-time binding and run-time type information is used only where necessary to achieve SoC.
- Pluggability
  - Sensors and actuators integrate themselves by design patterns and global instance construction.
- Extensibility
  - "Plug & Play" of sensor and actuator implementations.

### I4WeatherMon: CPP vs. OOP – Footprint

#### Footprint

- **Flash (Byte)**
  - C: 96, 128, 160, 192, 224, 256, 288, 320, 352, 384
  - AO: 1024, 1280, 1536, 2048, 2560, 3072, 3584
  - OO: 512, 1024, 1536, 2048, 2560, 3072, 3584

- **RAM (Byte)**
  - C: 64, 128, 192, 256, 320, 384, 448, 512
  - AO: 1024, 2048, 3072, 4096, 5120, 6144, 7168
  - OO: 512, 1024, 1536, 2048, 2560, 3072, 3584

**OOP is way more expensive!** Requires a larger µC for each variant.
Implementation Techniques: Summary

- CPP: minimal hardware costs – but no separation of concerns
- OOP: separation of concerns – but high hardware costs

OOP cost drivers

- Late binding of functions (virtual functions)
  - Calls cannot be inlined (∼ memory overhead for small methods)
  - Virtual function tables
  - Compiler always generates constructors (for vtable initialization)
  - Dead code elimination less effective
- Dynamic data structures
- Static instance construction
  - Generation of additional initialization functions
  - Generation of a global constructor table
  - Additional startup-code required

Root of the problem:

With OOP we have to use dynamic language concepts to achieve loose coupling of static decisions. ∼ AOP as an alternative.

Referenzen