Konfigurierbare Systemsoftware (KSS)

VL 2 – Software Product Lines

Daniel Lohmann

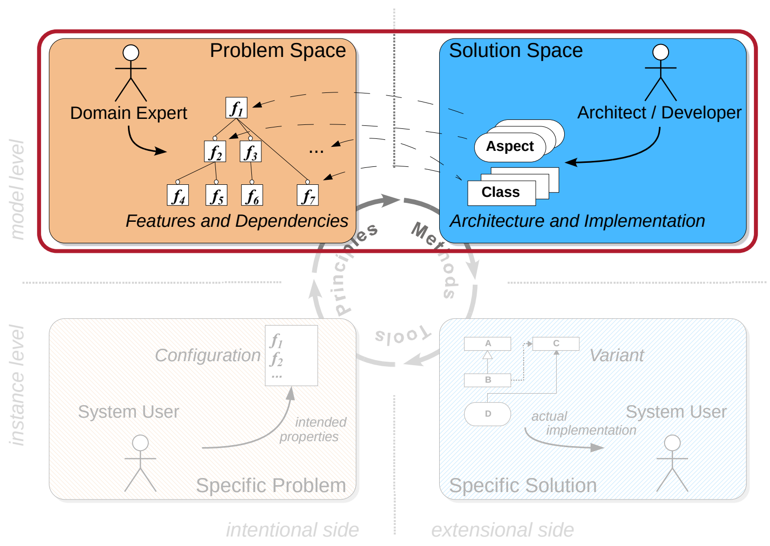
Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 13 – 2013-04-25

http://www4.informatik.uni-erlangen.de/Lehre/SS13/V_KSS

About this Lecture



Agenda

Agenda

Model Car Industry: Variety of an BMW X3

- Roof interior: **90000** variants available
- Car door: **3000** variants available
- Rear axle: **324** variants available

> *“Varianten sind ein wesentlicher Hebel für das Unternehmensergebnis”*
> Franz Decker (BMW Group)

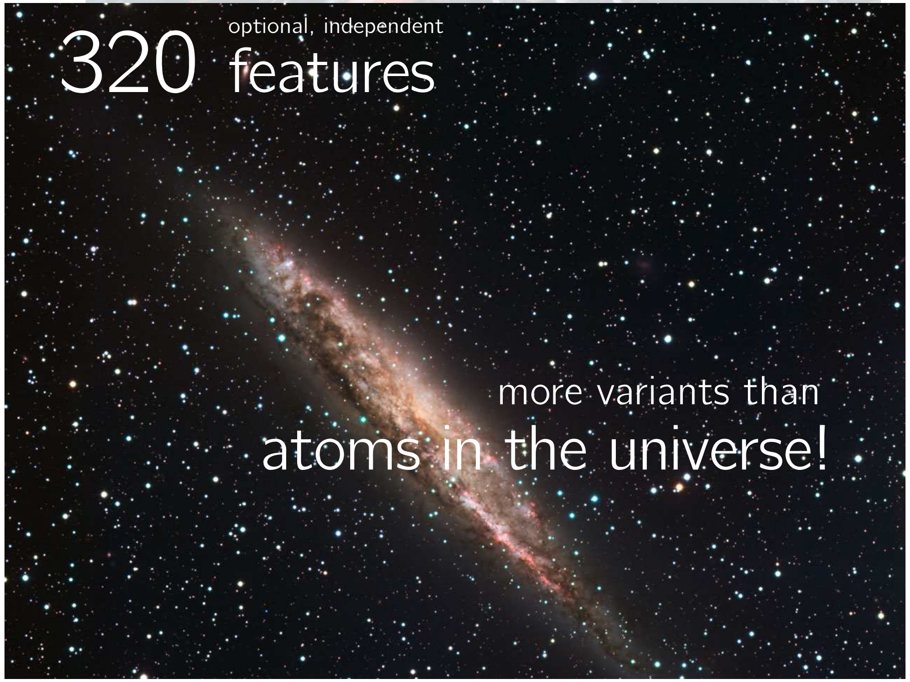
Model Car Industry: Variety Increase

- **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

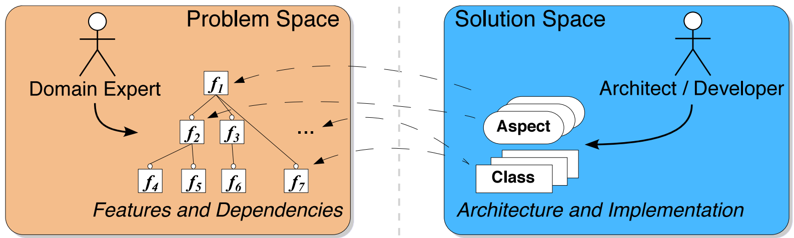
↪ **Product lines** with fully automated assembly

33 optional, independent features

one individual variant
for each human being



Challenges



❶ How to **identify** the actually desired variability?

❷ How to **express** the intended variability?

❸ How to **implement** this variability in the code?

❹ How to **map** variability options to the code?

Agenda

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**. ”*

Withey 1996: *Investment Analysis of Software Assets for Product Lines* [12]

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. ”*

Northrop and Clements 2001: *Software Product Lines: Practices and Patterns* [8]

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**. ”*

Withey 1996: *Investment Analysis of Software Assets for Product Lines* [12]

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. ”*

Northrop and Clements 2001: *Software Product Lines: Practices and Patterns* [8]

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]

The Emperors New Clothes?

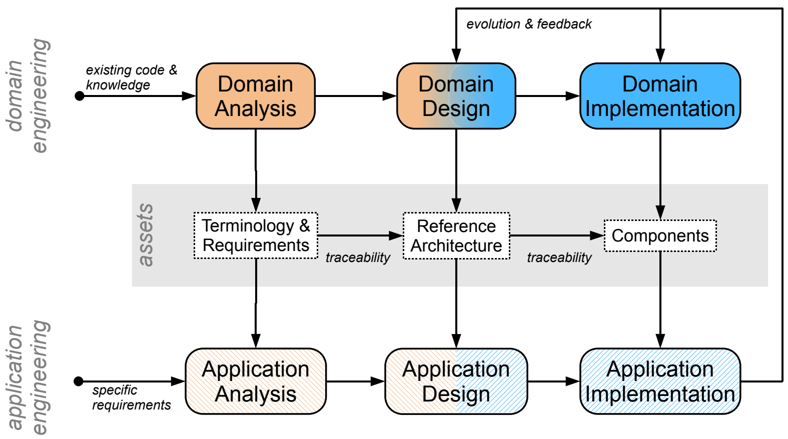
Program Family (Definition 4)

“ 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. ”

Parnas 1976: *“On the Design and Development of Program Families”* [10]

- 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 $\rightleftharpoons$ Software Product Line
- However, according to the definitions, the viewpoint is different
  - Program family: defined by similarity between programs $\mapsto$ Solutions
  - SPL: defined by similarity between requirements $\mapsto$ Problems
  - $\Rightarrow$ A program family implements a software product line
- In current literature, however, both terms are used synonymously
  - Program Family $\Longleftrightarrow$ Software Product Line

SPL Development Reference Process [1]



application engineering $\mapsto$ **tailoring**

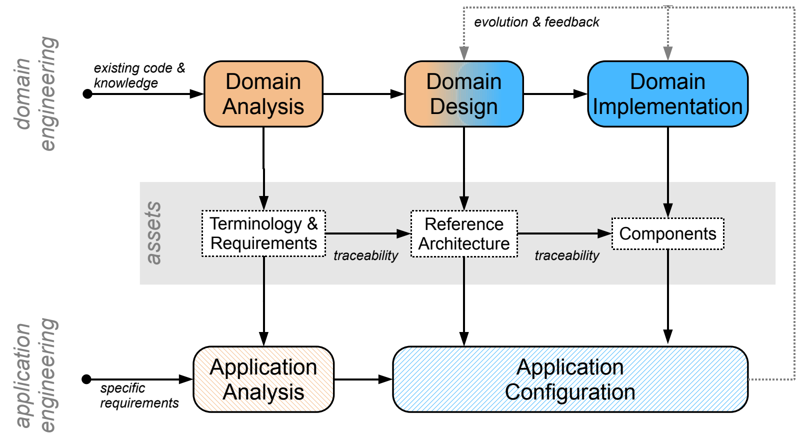
Our understanding: Configurable System Software

Configurability (Definition 5)

Configurability 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, CiAO, ...), ecosConfig (eCos)
    - configuration by an interactive configuration editor
    - formal model of configuration space, hierarchical features
    - implicit/explicit validation of constraints

Configurable SPL Reference Process

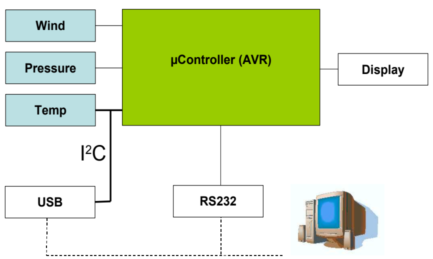


application engineering ↦ **configuring**

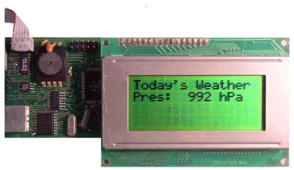
Agenda

The i4WeatherMon Weather Station [7]

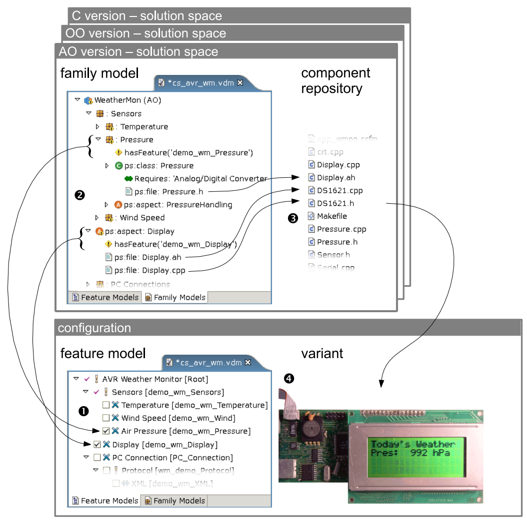
- A typical embedded system
  - Several, optional sensors
    - Wind
    - Air Pressure
    - Temperature
  - Several, optional actuators (here: output devices)
    - LCD
    - PC via RS232
    - PC via USB



- To be implemented as a product line
  - Barometer: Pressure + Display
  - Thermometer: Temperature + Display
  - Deluxe: Temperature + Pressure + Display + PC-Connection
  - Outdoor: *<as above>* + Wind
  - …

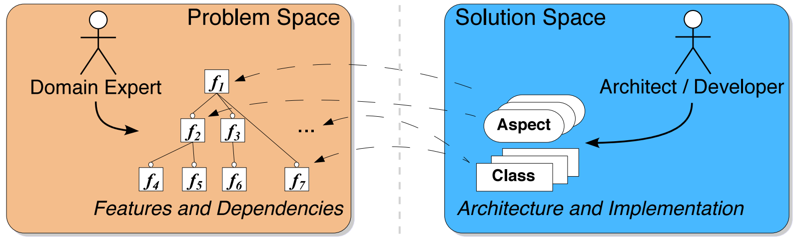


The i4WeatherMon Software Product Line [7]



Agenda

Challenges



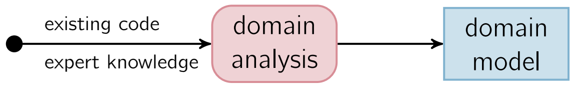
❶ How to **identify** the actually desired variability?

❷ How to **express** the intended variability?

❸ How to **implement** this variability in the code?

❹ How to **map** variability options to the code?

Domain Analysis



- 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]

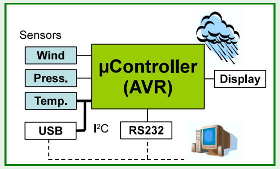
Elements of the Domain Model

- 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

I4WeatherMon: Domain Model (simplified)

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, PC, outdoor, ... shall be supported.



- The i4WeatherMon controller software is shipped in the flash memory of the μC and shall not be changed after delivery.
- The i4WeatherMon shall be usable with all versions of the PC Weather client software.
- ...

I4WeatherMon: Domain Model (simplified)

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 i4WeaterMon hardware that processes weather data (such as: LCD).
- **XML Protocol:** XML-based data scheme for the transmission of arbitrary weather data over a PC Connection.
- **SNG Protocol:** Binary legacy data scheme for the transmission of wind, temperature and air pressure data only over a PC Connection. The data scheme is used by versions < 2.0 of PC Weather.
- ...

I4WeatherMon: Domain Model (simplified)

Concept Models: i4WeatherMon

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

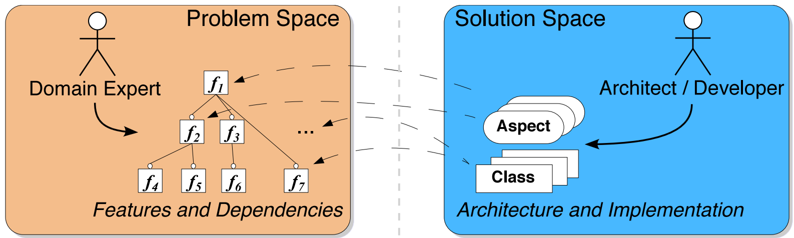
```
<!ELEMEMENT weather ...> ...
```

- **SNG Protocol:** Wind, temperature and air pressure data are encoded into 4 bytes, sequentially transmitted as a 3-byte datagram over a PC Connection as follows:

```
...
```

- **PC Connection** ...
- ...

Challenges



❶ How to **identify** the actually desired variability?

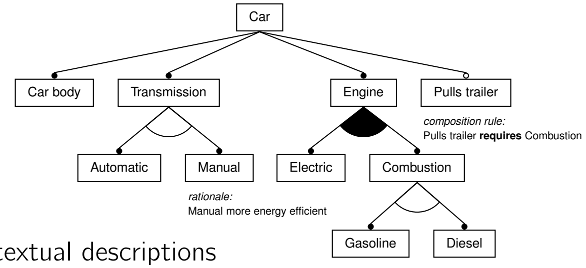
❷ How to **express** the intended variability?

❸ How to **implement** this variability in the code?

❹ 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

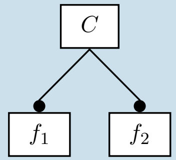


- Complemented by textual descriptions
  - Definition and rationale of each feature
  - Additional constraints, binding times, …

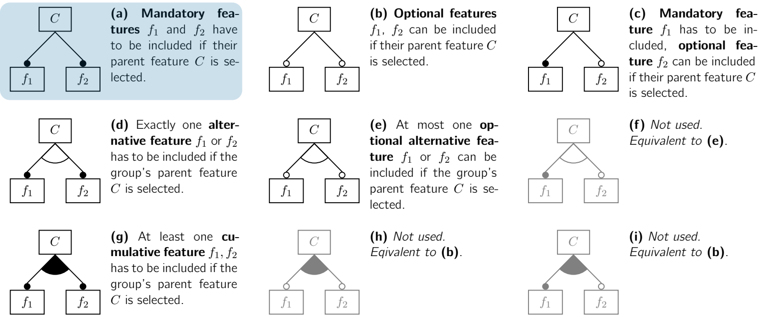
Feature Diagrams – Language [3]

Syntactical Elements

The filled dot • indicates a mandatory feature:
$\mathcal{V} = \{(C, f_1, f_2)\}$



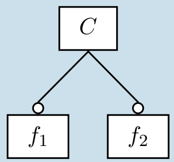
(a) Mandatory features f_1 and f_2 have to be included if their parent feature C is selected.



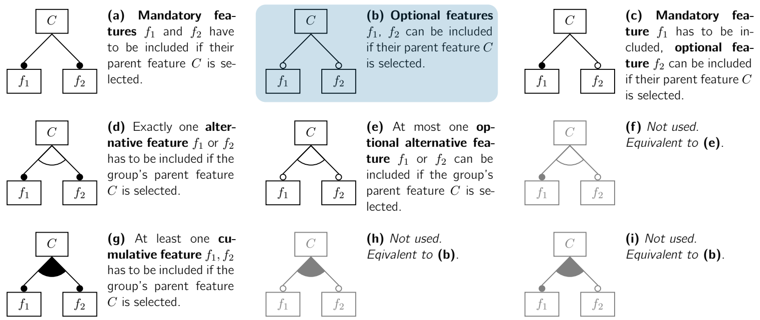
Feature Diagrams – Language [3]

Syntactical Elements

A shallow dot ∘ indicates an optional feature:
$\mathcal{V} = \{(C), (C, f_1), (C, f_2), (C, f_1, f_2)\}$



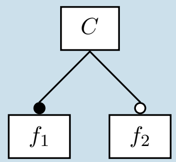
(b) Optional features f_1, f_2 can be included if their parent feature C is selected.



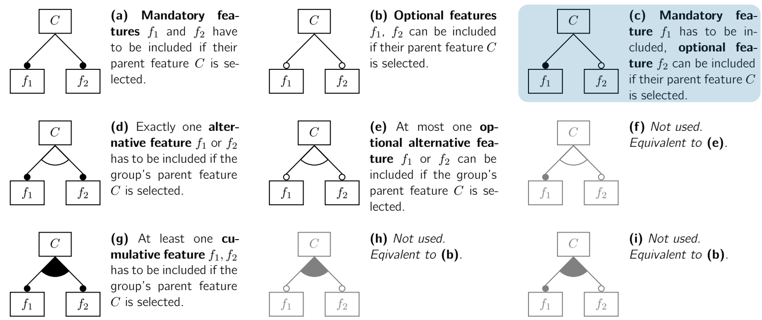
Feature Diagrams – Language [3]

Syntactical Elements

Of course, both can be combined:
$\mathcal{V} = \{(C, f_1), (C, f_1, f_2)\}$



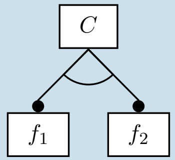
(c) Mandatory feature f_1 has to be included, **optional feature** f_2 can be included if their parent feature C is selected.



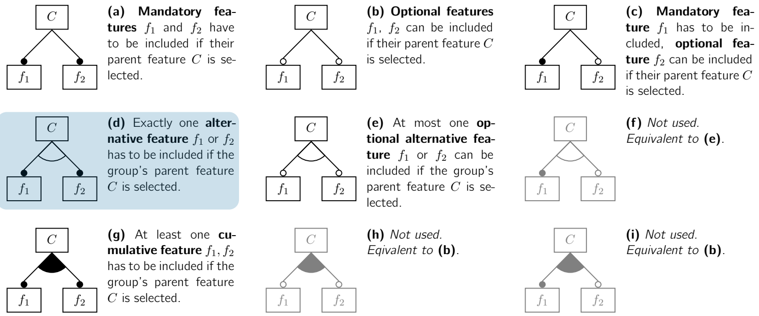
Feature Diagrams – Language [3]

Syntactical Elements

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



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



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

(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, **optional feature** f_2 can 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.

(e) At most one **optional alternative feature** f_1 or f_2 can be included if the group's parent feature C is selected.

(f) *Not used. Equivalent to* **(e)**.

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

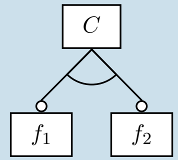
(h) *Not used. Eqivalent to* **(b)**.

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

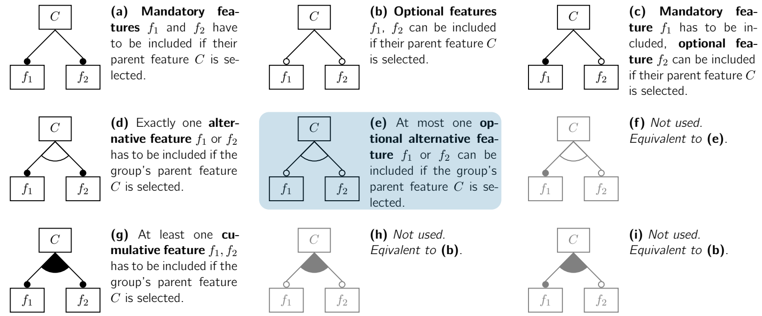
Feature Diagrams – Language [3]

Syntactical Elements

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



(e) At most one **optional alternative feature** f_1 or f_2 can be included if the group's parent feature C is selected.



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

(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, **optional feature** f_2 can 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.

(e) At most one **optional alternative feature** f_1 or f_2 can be included if the group's parent feature C is selected.

(f) *Not used. Equivalent to* **(e)**.

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

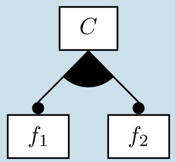
(h) *Not used. Eqivalent to* **(b)**.

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

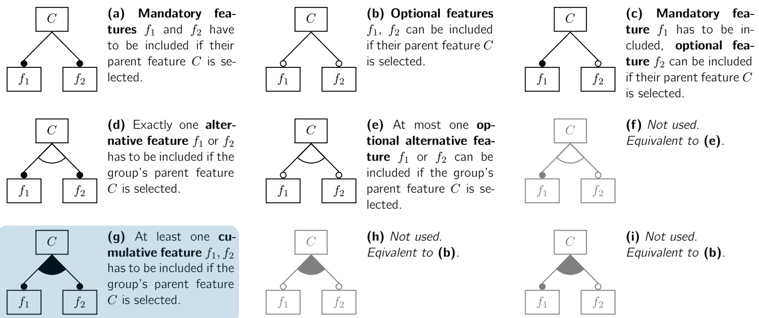
Feature Diagrams – Language [3]

Syntactical Elements

The filled arc depicts a group of cummulative features: $\mathcal{V} = \{(C, f_1), (C, f_2), (C, f_1, f_2)\}$



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



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

(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, **optional feature** f_2 can 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.

(e) At most one **optional alternative feature** f_1 or f_2 can be included if the group's parent feature C is selected.

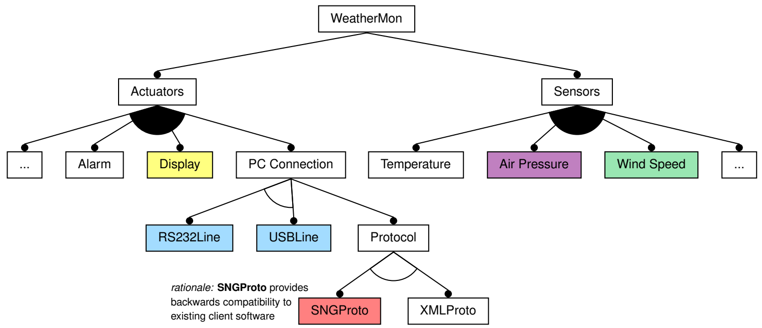
(f) *Not used. Equivalent to* **(e)**.

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

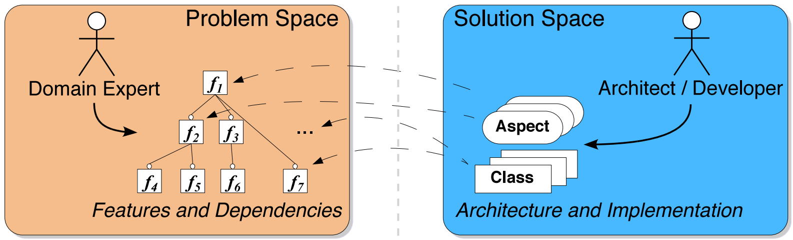
(h) *Not used. Eqivalent to* **(b)**.

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

I4WeatherMon: Feature Model



Challenges



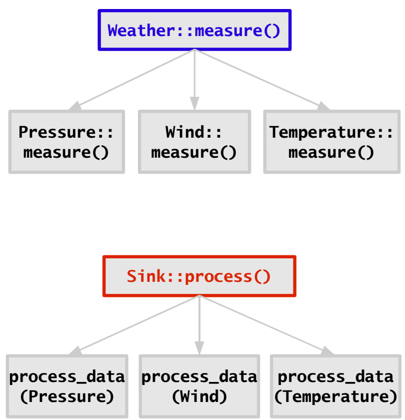
1. How to **identify** the actually desired variability?
2. How to **express** the intended variability?
3. How to **implement** this variability in the code?
4. How to **map** variability options to the code?

Agenda

I4WeatherMon: Reference Architecture

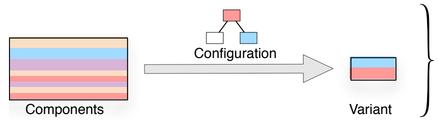
Functional decomposition (structure and process):

```
int main() {
  Weather data;
  Sink    sink;

  while(true) {

    // aquire data
    data.measure();

    // process data
    sink.process( data );

    wait();
  }
}
```



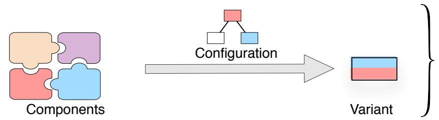
Implementation Techniques: Classification

- Decompositional Approaches



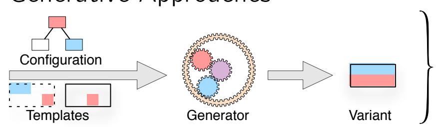
  - Text-based filtering (untyped)
  - Preprocessors

- Compositional Approaches



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

- Generative Approaches



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

Implementation Techniques: Goals

General

❶ Separation of concerns (SoC)

❷ Resource thriftiness

Operational

❸ **Granularity** Components should be fine-grained. Each artifact should either be mandatory or dedicated to a single feature only.

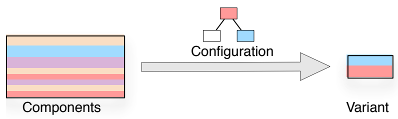
❹ **Economy** The use of memory/run-time expensive language features should be avoided as far as possible. Decide and bind as much as possible at generation time.

❺ **Pluggability** 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”.

❻ **Extensibility** The same should hold for new optional features, which may be available in a future version of the product line.

Implementation Techniques: The C Preprocessor

- Decompositional Approaches



  - Text-based filtering (untyped)
  - Preprocessors (**CPP**)

- Conditional compilation with the C Preprocessor (CPP) is *the* standard approach to implement static configurability [6]
  - 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)

I4WeatherMon (CPP): Implementation (Excerpt)

```
struct Weather {

  #ifdef cfWM_WIND
        UInt16  _w;
  #endif

  #ifdef cfWM_PRESSURE
        UInt16  _p;
  #endif

  #ifdef cfWM_TEMPERATURE
    Int8 _t1;
    UInt8 _t2;
  #endif

  #ifdef cfWM_STACK
        unsigned int _maxstack;
  #endif
};
```

Sensor integration cross-cuts the central data structure, an interaction with a mandatory feature.

I4WeatherMon (CPP): Implementation (Excerpt)

```
// sensor processing
inline void init_sensors() {
  #ifdef cfWM_STACK
      stack_init();
  #endif
  #ifdef cfWM_WIND
      wind_init();
  #endif

  #ifdef cfWM_PRESSURE
      pressure_init();
  #endif

  #ifdef cfWM_TEMPERATURE
      temperature_init();
  #endif
}
```

Sensor (and actuator) integration both crosscut the structure of the main program, an interaction with a mandatory feature.

I4WeatherMon (CPP): Implementation (Excerpt)

```
inline void XMLCon_process() {
    char val[ 5 ];

    Serial::send ("<?xml version=\"1.0\"?>\n" "<weather>\n");

  #ifdef cfWM_WIND
    wind_stringval( val );
    XMLCon_data ( wind_name(), val );
  #endif

  #ifdef cfWM_PRESSURE
    pressure_stringval( val );
    XMLCon_data ( pressure_name(), val );
  #endif

  #ifdef cfWM_TEMPERATURE
    temperature_stringval( val );
    XMLCon_data ( temperature_name(), val );
  #endif

  #ifdef cfWM_STACK
    stack_stringval( val );
    XMLCon_data ( stack_name(), val );
  #endif

    Serial::send ("</weather>\n");
}
```

Sensor integration also crosscuts actuator code, an interaction between optional features!

I4WeaterMon (CPP): Evaluation

General

❶ Separation of concerns (SoC) ✘

❷ Resource thriftiness ✔

Operational

❸ Granularity (✔)

- Components implement only the functionality of a single feature, but contain integration code for other optional features.

❹ Economy ✔

- All features is bound at compile time.

❺ Pluggability ✘

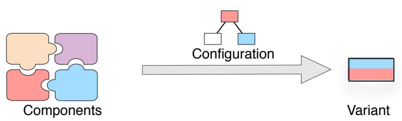
- Sensor integration crosscuts main program and actuator implementation.

❻ Extensibility ✘

- New actuators require extension of main program.
- New sensors require extension of main program and existing actuators.

Implementation Techniques: OOP

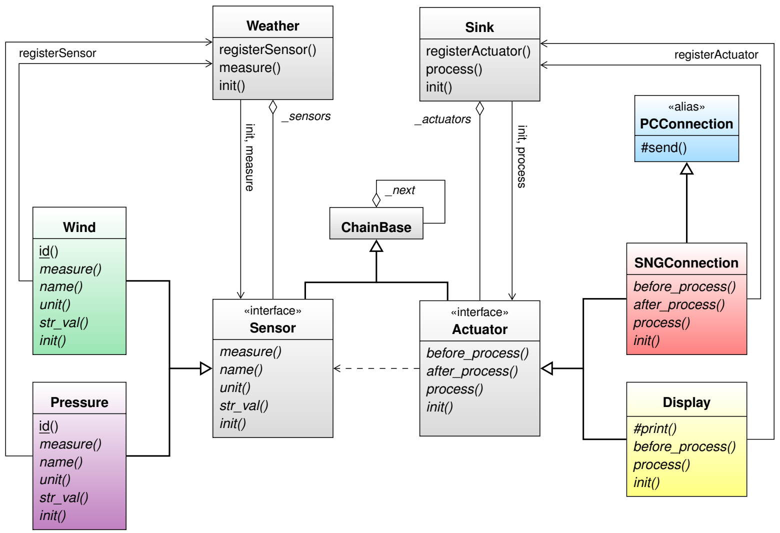
- Compositional Approaches



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

- Object-oriented programming languages provide means for loose coupling by generalization and OO design patterns
  - Interfaces
    ⇝ type substitutability (optional/alternative features)
  - Observer-Pattern
    ⇝ quantification (cumulative feature groups)
  - Implicit code execution by global instance construction
    ⇝ self integration (optional features)

I4WeatherMon (OOP): Design (Excerpt)



I4WeaterMon (OOP): Evaluation

General

❶ Separation of concerns (SoC) ✔

❷ Resource thriftiness ?

Operational

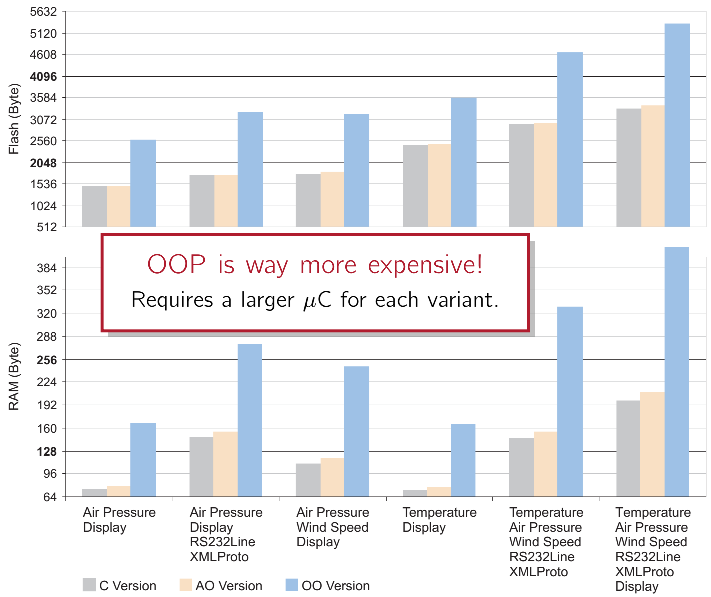
❸ Granularity ✔
- Every component is either a base class or implements functionality of a single feature only.

❹ Economy (✔)
- Run-time binding and run-time type information is used only where necessary to achieve SoC.

❺ Pluggability ✔
- Sensors and actuators integrate themselve by design patterns and global instance construction.

❻ Extensibility ✔
- "Plug & Play" of sensor and actuator implementations.

I4WeaterMon: CPP vs. OOP – Footprint



I4WeaterMon: CPP vs. OOP – Footprint

variant	version	text	data	bss	stack	= flash	= RAM	time (ms)
Air Pressure, Display	C	1392	30	7	34	1422	71	1.21
	AO	1430	30	10	38	1460	78	1.21
	OO	2460	100	22	44	2560	166	1.29
Air Pressure, Display,	C	1578	104	7	34	1682	145	60.40
RS232Line, XMLProto	AO	1622	104	12	38	1726	154	59.20
	OO	3008	206	26	44	3214	276	60.80
Air Pressure, Wind Speed,	C	1686	38	14	55	1724	107	2.96
Display	AO	1748	38	18	61	1786	117	2.96
	OO	3020	146	33	65	3166	244	3.08
Temperature, Display	C	2378	28	8	34	2406	70	1.74
	AO	2416	28	11	38	2444	77	1.73
	OO	3464	98	23	44	3562	165	1.82
Temperature, Wind Speed,	C	2804	90	17	35	2894	142	76.40
Air Pressure, RS232Line,	AO	2858	90	23	41	2948	154	76.40
XMLProto	OO	4388	248	39	41	4636	328	76.40
Temperature, Wind Speed,	C	3148	122	17	57	3270	196	79.60
Air Pressure, RS232Line,	AO	3262	122	24	63	3384	209	77.60
XMLProto, Display	OO	5008	300	44	67	5308	411	80.00

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

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.

© dl KSS (VL 2 | SS 13) 2 Software Product Lines | 2.5 Solution Space 2–44

Referenzen

[1] Günter Böckle, Peter Knauber, Klaus Pohl, et al. *Software-Produktlinien: Methoden, Einführung und Praxis*. Heidelberg: dpunkt.verlag GmbH, 2004. isbn: 3-80864-257-7.

[2] Fred Brooks. *The Mythical Man Month*. Addison-Wesley, 1975. isbn: 0-201-00650-2.

[3] Krysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000. isbn: 0-20-13097-77.

[4] Edsger Wybe Dijkstra. "The Structure of the THE-Multiprogramming System". In: *Communications of the ACM* 11.5 (May 1968), pp. 341–346.

[5] Arie Nicolaas Habermann, Lawrence Flon, and Lee W. Cooprider. "Modularization and Hierarchy in a Family of Operating Systems". In: *Communications of the ACM* 19.5 (1976), pp. 266–272.

[6] Jörg Liebig, Sven Apel, Christian Lengauer, et al. "An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines". In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE '10)*. (Cape Town, South Africa). New York, NY, USA: ACM Press, 2010. doi: `10.1145/1806799.1806819`.

Referenzen (Cont'd)

[7] Daniel Lohmann, Olaf Spinczyk, and Wolfgang Schröder-Preikschat. "Lean and Efficient System Software Product Lines: Where Aspects Beat Objects". In: *Transactions on AOSD II*. Ed. by Awais Rashid and Mehmet Aksit. Lecture Notes in Computer Science 4242. Springer-Verlag, 2006, pp. 227–255. doi: `10.1007/11922827_8`.

[8] Linda Northrop and Paul Clements. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. isbn: 978-0-201-70332-0.

[9] David Lorge Parnas. "On the Criteria to be used in Decomposing Systems into Modules". In: *Communications of the ACM* (Dec. 1972), pp. 1053–1058.

[10] David Lorge Parnas. "On the Design and Development of Program Families". In: *IEEE Transactions on Software Engineering* SE-2.1 (Mar. 1976), pp. 1–9.

[11] David Lorge Parnas. *Some Hypothesis About the "Uses" Hierarchy for Operating Systems*. Tech. rep. TH Darmstadt, Fachbereich Informatik, 1976.

[12] James Withey. *Investment Analysis of Software Assets for Product Lines*. Tech. rep. Pittsburgh, PA: Carnegie Mellon University, Software Engineering Institute, Nov. 1996.