Konfigurierbare Systemsoftware (KSS)

VL 2 – Software Product Lines

Daniel Lohmann

Lehrstuhl für Informatik 4
Verteilte Systeme und Betriebssysteme

Friedrich-Alexander-Universität
Erlangen-Nürnberg

SS 16 – 2016-04-18

http://www4.informatik.uni-erlangen.de/Lehre/SS16/V_KSS
About this Lecture

Problem Space

Domain Expert

Features and Dependencies

Solution Space

Architect / Developer

Class

Architecture and Implementation

instance level

System User

Specific Problem

intentional side

configuration

intended properties

Variant

actual implementation

System User

Specific Solution

extensional side

System User

Variant

actual implementation

System User

Specific Solution
About this Lecture

Problem Space
- Domain Expert
- Features and Dependencies

Solution Space
- Architect / Developer
- Architecture and Implementation

Features and Dependencies
- $f_1$
- $f_2$
- $f_3$
- $f_4$
- $f_5$
- $f_6$
- $f_7$

Domain Expert

Architect / Developer

Architecture and Implementation

Configuration
- $f_1$
- $f_2$
- ...
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
Agenda

2.1 Motivation: The Quest for Variety
Model Car Industry
Challenges

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: 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
optional, independent features

320 features

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

5000 features

14000 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)**

“*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)**

“*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)**

“*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]
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 $\Rightarrow$ 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 $\iff$ Software Product Line
SPL Development Reference Process

Domain engineering:
- Domain Analysis
- Domain Design
- Domain Implementation

Application engineering:
- Application Analysis
- Application Design
- Application Implementation

Assets:
- Terminology & Requirements
- Reference Architecture
- Components

Traceability:
- Evolution & Feedback

Application engineering → tailoring
**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

- **Domain Engineering**
  - Domain Analysis
  - Domain Design
  - Domain Implementation

- **Application Engineering**
  - Application Analysis
  - Application Configuration

**Assets**
- Terminology & Requirements
- Reference Architecture
- Components

**Evolution & Feedback**

**Application Engineering** → **Configuring**


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

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

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.

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, 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.

- ...
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 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.

- ...
Concept Models: i4WeatherMon

- **XML Protocol**: The following DTD specifies the format used for data transmission over a PC Connection:
  ```xml
  <!ELEMENT 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

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
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.

(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. Equivalent to (b).

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

A shallow dot ⋄ indicates an optional feature:

\[ V = \{(C), (C, f_1), (C, f_2), (C, f_1, f_2)\} \]

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

(b) **Optional features** \( f_1, f_2 \) 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.** Equivalent to (b).

(i) **Not used.** Equivalent to (b).
Feature Diagrams – Language

Syntactical Elements

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

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

(b) Optional features \( f_1 \) and \( f_2 \) 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. Equivalent to (b).

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

The shallow arc $\triangledown$ depicts a group of alternative features:

$$\mathcal{V} = \{(C, f_1), (C, f_2)\}$$

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

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

(b) Optional features $f_1$, $f_2$ can be included if their 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.

(c) Mandatory feature $f_1$ has to be included, optional feature $f_2$ can be included if their 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. Equivalent to (b).

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

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

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

(b) Optional features \(f_1, f_2\) 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. Equivalent to (b).

(i) Not used. Equivalent to (b).
Feature Diagrams – Language

Syntactical Elements

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

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

(b) Optional features \( f_1 \) and \( f_2 \) 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. Equivalent to (b).

(i) Not used. Equivalent to (b).
I4WeatherMon: Feature Model

WeatherMon

Actuators
- Alarm
- Display
- PC Connection

Sensors
- Temperature
- Air Pressure
- Wind Speed

Protocols
- SNGProto
- XMLProto

rationale: SNGProto provides backwards compatibility to existing client software
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?
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
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
## Implementation Techniques: Goals

### General
1. **Separation of concerns (SoC)**
2. **Resource thriftiness**

### Operational
3. **Granularity**
   - Components should be fine-grained. Each artifact should either be mandatory or dedicated to a single feature only.
4. **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.
5. **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”.
6. **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

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”)

- Text-based filtering (untyped)
- Preprocessors (CPP)
I4WeatherMon (CPP): Implementation (Excerpt)
I4WeatherMon (CPP): Implementation (Excerpt)

```c
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.
# sensor processing

```cpp
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.
inline void XMLCon_process() {
    char val[5];
    Serial::send("<?xml version="1.0"?>
    "<weather>
    
    #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>
    
inline void XMLCon_init() {
#endif // __XMLConnection_ah__

inline void wind_measure() {
    // measure wind
    _wind_counter = CiAO::TimerCounter::value();
    XMLCon.data(wind_name(), val);
    wind_stringval(val);
}

inline void pressure_measure() {
    // measure pressure
    // Get sensor data
    data._w = _wind_counter;
    XMLCon.data(pressure_name(), val);
    pressure_stringval(val);
}

inline void stack_measure() {
    // measure stack
    // Get sensor data
    XMLCon.data(stack_name(), val);
    stack_stringval(val);
}

}
I4WeaterMon (CPP): Evaluation

General
① Separation of concerns (SoC) ✗
② Resource thriftiness ✔

Operational
③ Granularity
- Components implement only the functionality of a single feature, but contain integration code for other optional features. (✔)
④ Economy
- All features is bound at compile time. ✔
⑤ Pluggability
- Sensor integration crosscuts main program and actuator implementation. ✗
⑥ 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)

```
// «interface» Sensor
measure()  // id
name()     // id
unit()     // id
str_val()  // id
init()     // id

// «interface» Actuator
before_process() // id
after_process()  // id
process()        // id
init()           // id

// «alias» SNGConnection
#send()

// ▼ registerSensor -> ▲ registerActuator

Weather:
registerSensor()
measure()
init()

Sink:
registerActuator()
process()
init()

ChainBase:
_sensors
_init, measure

Wind:
id()  // id
measure()  // id
name()     // id
unit()     // id
str_val()  // id
init()     // id

Pressure:
id()  // id
measure()  // id
name()     // id
unit()     // id
str_val()  // id
init()     // id

Display:
#print()
before_process()
process()
init()

PCConnection:
«alias»
before_process()
after_process()
process()
init()

Colors reflect the respective features → [2–30]
```
I4WeaterMon (OOP): Evaluation

General

1. Separation of concerns (SoC) ✔
2. Resource thriftiness ?

Operational

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

4. Economy
   - Run-time binding and run-time type information is used only where necessary to achieve SoC.

5. Pluggability
   - Sensors and actuators integrate themselves by design patterns and global instance construction.

6. Extensibility
   - “Plug & Play” of sensor and actuator implementations.
OOP is way more expensive!
Requires a larger μC for each variant.
### I4WeaterMon: CPP vs. OOP – Footprint

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

leadsto AOP as an alternative.


