Trustworthy Framework Instantiation

Prof. Dr. Uwe Aßmann
TU Dresden
Institut für Software– und Multimediatechnik
Lehrstuhl Softwaretechnologie
05-0.2, Jan 24, 2006
Obligatory Literature

The Framework Instantation Problem

- Frameworks are often hard to instantiate, because they have many extension and variation points and dependencies between them.
- Whitebox frameworks are often instantiated with non-conformant subclasses.
- Blackbox frameworks are often instantiated with non-fitting classes (multi-point dependencies).
- Some constraints cannot be checked statically.
Problem 1: A Car Configurator

- How to instantiate two 1-T-H hooks, if there are dependencies between them (multi-point constraints)?
- Static constraint, domain-specific
Problem 2: SalesPoint Framework

- Catalog and Stock hierarchies must be isomorphic
- Dynamic constraint; domain-specific
Problem 3: Parallel Hierarchies

- Window types must be varied parallely
- Static constraint, but technical
Problem 4: Dynamic Assumptions

► Null-checks
► Range checks
► Sortedness of ordered collections
► Other contract checks

► Dynamic technical constraints
## Classification of Instantiation Constraints

<table>
<thead>
<tr>
<th>Facet 2: Cause</th>
<th>Static</th>
<th>Dynamic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain-specific (analysis-related)</td>
<td>Car configurator multi-point constraint</td>
<td>SalesPoint isomorphic hierarchies of Catalogs and Stocks</td>
</tr>
<tr>
<td>Technical (design-related)</td>
<td>Windows parallel hierarchies</td>
<td>Dynamic assumptions Dynamic contracts</td>
</tr>
</tbody>
</table>
Remedies for Trustworthy Instantiation
Checking Mechanisms in All Phases of the Life Cycle

- Domain Model (Domain Ontology)
- Analysis Model
- Design Model
- Implementation Model
- Refactorings
- Framework Instantiation Languages
- Documentation

- Static checking of domain constraints
- Dynamic checking of domain constraints
- Testing of analysis constraints
- Dynamic checking of technical constraints
- Testing of technical constraints
- Static checking of technical constraints
- Run-Time Checks
- Acceptance Test
- System and Integration Test
- Unit Tests
Refactoring of Multi-Point Constraints

- Multi-point constraints can be refactored such that the constraint moves inside the framework
  - One point is removed
- Advantage: Framework can control itself
Static Verification of Static Constraints

- **UML collaborations** are appropriate to describe static (technical and domain-specific) instantiation constraints.
  - OCL specifies static invariants of the framework, instantiation preconditions and postconditions
  - OCL can reason over types, hence, instantiations or extensions of the framework can be analyzed and verified

![Diagram of UML collaborations and OCL expressions](image)
Framework Testing

Frameworks must be *negatively tested*

- Beyond functional tests (positive tests), censorious negative tests for the behavior in case of misinstantiation must be conducted
- Negative test cases have to be derived
  - specifying ill instantiation conditions
  - and the behavior of the framework
- Framework must react reasonably
  - NOT dump core
  - Handle exceptions appropriately
  - Emit comprehensible error messages, also to the end user
Misuse Diagrams

- Misuse diagrams specify misuse cases, dually to use case diagrams, which specify functional use cases.
- Used to describe system abuse (intrusion, fraud, security attacks)
- Coarse-grain technique to specify also framework misuse
From use case diagrams, usually test tables are derived
- A test table contains test case entries, describing one test case
  - Class of test case (positive, negative)
  - Input parameters of method
  - Output parameters
  - Reaction, state afterwards

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Testclass</th>
<th>Input</th>
<th>Output</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>String date</td>
<td>Date d1</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>day</td>
<td>month</td>
</tr>
<tr>
<td>1</td>
<td>positive</td>
<td>1. Januar 2006</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>positive</td>
<td>05/12/2008</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>positive</td>
<td>January 23, 2007</td>
<td>23</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>negative</td>
<td>Mak 44, 2007</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>negative</td>
<td>March 44, 2007</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Negative Test Case Entries for Misuse of Frameworks

- Input parameters must be refined
  - Dynamic constraints are tested as usual negative test cases, with input and output parameter specification
  - Static constraints, however, work on types. Hence, their test case entries are different. Negative test cases specify ill instantiations, framework error messages and exception handling

<table>
<thead>
<tr>
<th>Testcase</th>
<th>Testclass</th>
<th>Input</th>
<th>Reaction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>pos. static</td>
<td>QtMenu</td>
<td>QtButton</td>
</tr>
<tr>
<td>2</td>
<td>pos. static</td>
<td>GtkMenu</td>
<td>GtkButton</td>
</tr>
<tr>
<td>3</td>
<td>neg. static</td>
<td>QtMenu</td>
<td>GtkButton error „for multi-point, use parallel classes“</td>
</tr>
<tr>
<td>4</td>
<td>neg. static</td>
<td>GtkMenu</td>
<td>QtButton error „for multi-point, use parallel classes“</td>
</tr>
</tbody>
</table>
Derivation of JUnit Test Cases

- From every test table entry dealing with a dynamic constraint, a JUnit test case is derived ([www.junit.org](http://www.junit.org))
  - Test method or test class with test method, deriving from class `TestCase`
- From every test table entry dealing with a static constraint, a compilation test suite case is derived
  - Stored in a database
  - Sold with the framework to the customer of the framework
  - Helps the customer to instantiate right
- See course Softwaretechnologie II, summer semester
Framework Instantiation Languages

- Eclipse has demonstrated that a framework extension (instantiation) language can be beneficial
  - to type variability and extension points
  - to describe not only extension points for code, but also for other resources, such as GUI elements, business objects, etc.

- Eclipse language is based on XML, thus restricted on:
  - XML tree specifications
  - XML base types
Eclipse Extension Specs

**Example of a plugin**

```
<plugin name="extended">
  <extension-point id="example">
    <attribute name="id" type="string"/>
    <attribute name="name" type="string"/>
    <attribute name="schema" type="string"/>
  </extension-point>
</plugin>
```

**Example of an extension**

```
<extension-point id="example">
  <attribute name="id" type="string"/>
  <attribute name="name" type="string"/>
  <attribute name="schema" type="string"/>
</extension-point>
```

**Example of extending a plugin**

```
<extension point="example">
  <attribute name="point" type="string">
    <attribute name="class" type="string"/>
  </attribute>
</extension>
```

**Example of extending a plugin with a specific class**

```
<plugin name="extending">
  <extension point="example">
    <attribute name="class" type="string">
      <attribute name="package" type="string">
        package org.savga.Runner
      </attribute>
    </attribute>
  </extension>
</plugin>
```
Why A Framework Extension Language Should Be Based on Logic

- Beyond XML, logic can capture context-sensitive static constraints
  - also static multi-point framework instantiation constraints
- However, the logic must be enriched with domain-specific concepts, such as framework, hook, variation point, extension point, instantiation, etc.
- Good candidates are *typed logic languages*
  - Ontology languages OWL, SWRL
  - Frame logic (F-logic, on top of XSB)
  - OCL on UML class diagrams (UML collaborations)
Dynamic Contract Checking

- Dynamic multi-point constraints must be checked at run-time
  - Mainly, this amounts to *contract checking* of the framework
- Two best practices can be applied:
  - Framework contract layers
  - Contract aspects
Framework Contract Layers

- Best practice is to check a dynamic constraint (single- or multi-point) in a separate layer, encapsulating the *contract concern*

- The checking layer is called from outside (the application), but the inner layer from inside the framework. This is much faster than checking always!
  - When composing the framework with others, the contract layer can be wrapped around the resulting bigger framework (check effort reduced)

```java
class Collection {
    public boolean sorted() { ... /* sortedness predicate */ } 
    public Element searchBinaryBinary(ElementKey key) {
        // contract checking
        if(!sorted())
            sort();
        // calling the inner layer
        return searchBinaryInternal(key);
    }

    // inner layer
    protected Element searchBinaryInternal(ElementKey key) {
        .. binary search algorithm ...
    }
}
```
Contract Aspects

Once encapsulated in a layer, contract checks can be moved into a *contract aspect*:

- Tools such as Aspect/J can weave the contract in
- Here: methods of package *framework* that have a parameter of type *Menu* are checked on null value

Advantage: the aspect can easily be exchanged

- Reduces effort, in particular when the aspect is *crosscutting*

```java
before(Menu m): call(* framework.*.*(Menu)) && args(m) {
    if (m == null) {
        throw new Exception ("Null Menu parameter passed when " +
                           thisJoinPoint.getThis() + " was called ");
    }
}
```
What Have We Learned?

► Framework instantiation and extension is hard, because there are many constraints, both domain-specific and technical, to obey
► Multi-point constraints describe dependencies between two or several framework hooks
► Appropriate remedies against misinstantiations are:
  ▪ Thorough documentation (well, of course with the pyramid principle)
  ▪ Refactoring (removal) of multi-point constraints
  ▪ Negative testing with misuse diagrams and negative test table entries
  ▪ Using logic to verify static constraints
  ▪ Use contract layers and contract aspects to facilitate checking of dynamic constraints
Open Jobs

- Students searched for project B2-PDE, on design aspects of a big business framework (with COMARCH)
  - Framework testing (negative testing, misuse diagrams)
  - Framework extension languages
  - Framework documentation tools based on elucidative programming
  - Adapter generators
  - Contract layer generators
The End