



24. Trustworthy Framework Instantiation

Prof. Dr. Uwe Aßmann
TU Dresden

Institut für Software- und
Multimediatechnik

Lehrstuhl Softwaretechnologie
11-0.1, 23.12.11

- 1) The framework instantiation problem
- 2) Remedies

Obligatory Literature

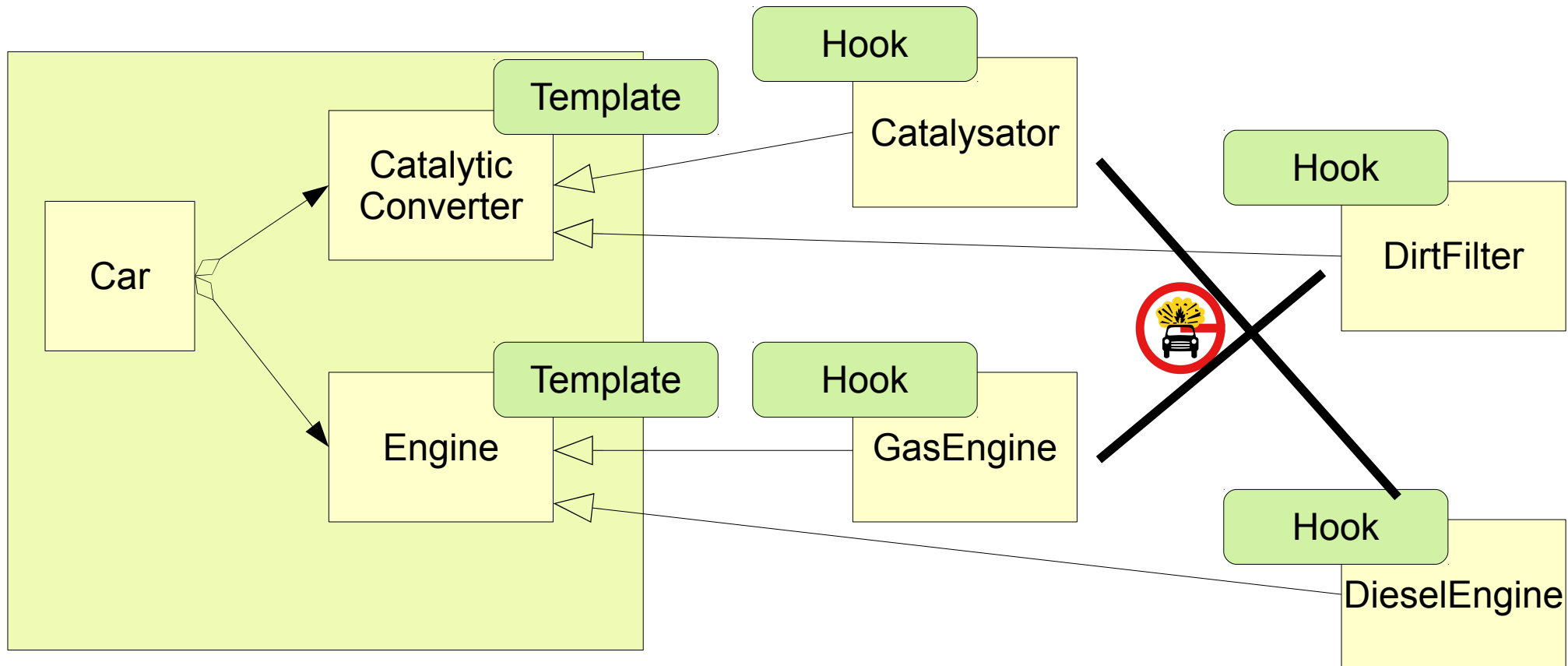
- ▶ Uwe Aßmann, Andreas Bartho, Falk Hartmann, Ilie Savga, Barbara Wittek. Trustworthy Instantiation of Frameworks. In *Trustworthy Components*, Reussner, Ralf and Szyperski, Clemens (ed.), Jan. 2006. LNCS 3938, Springer. Available at <http://www.springerlink.com/index/104074p5h8581115.pdf>

24.1 The Framework Instantiation 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

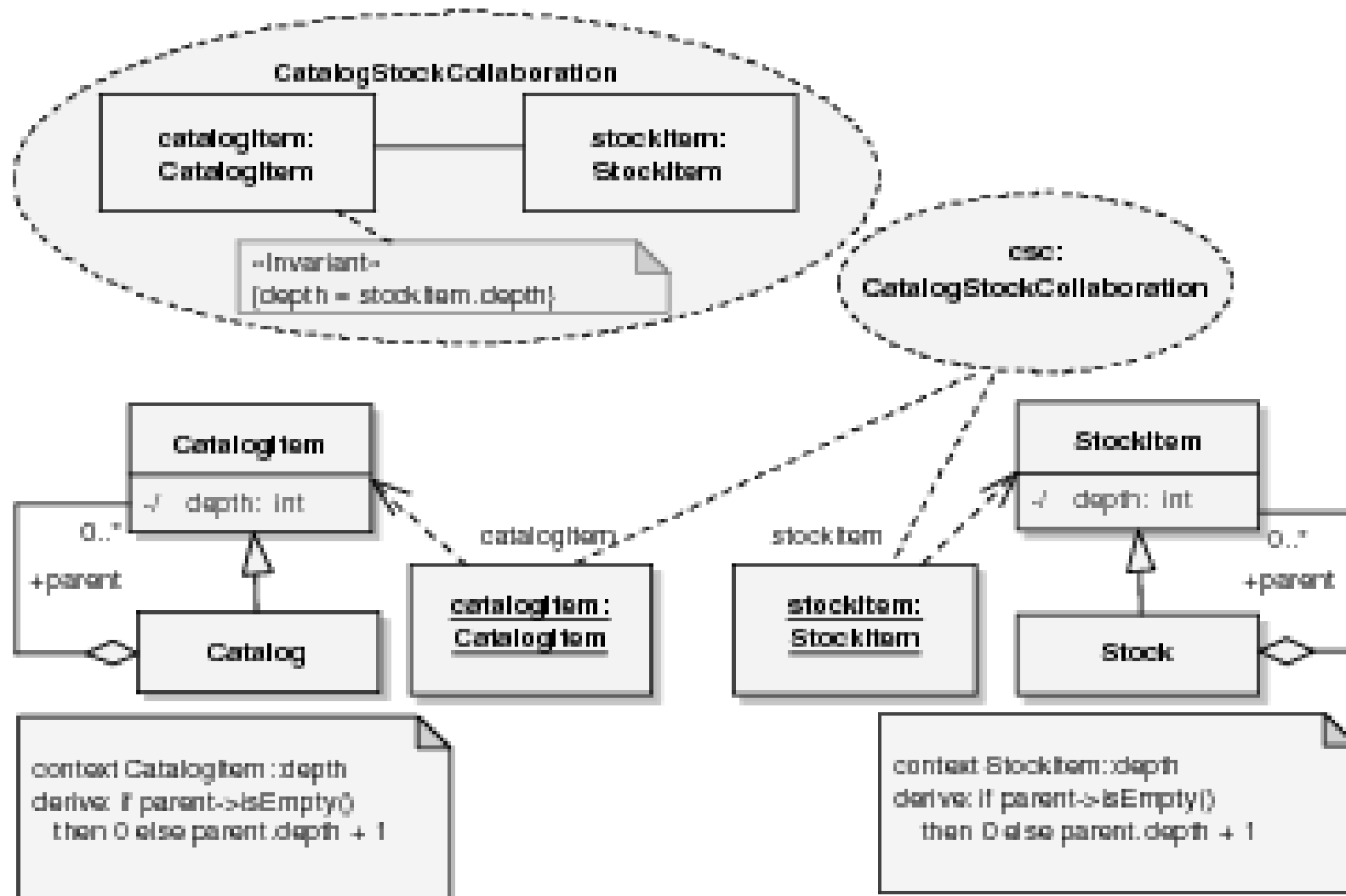
Problem 1: A Car Configurator

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



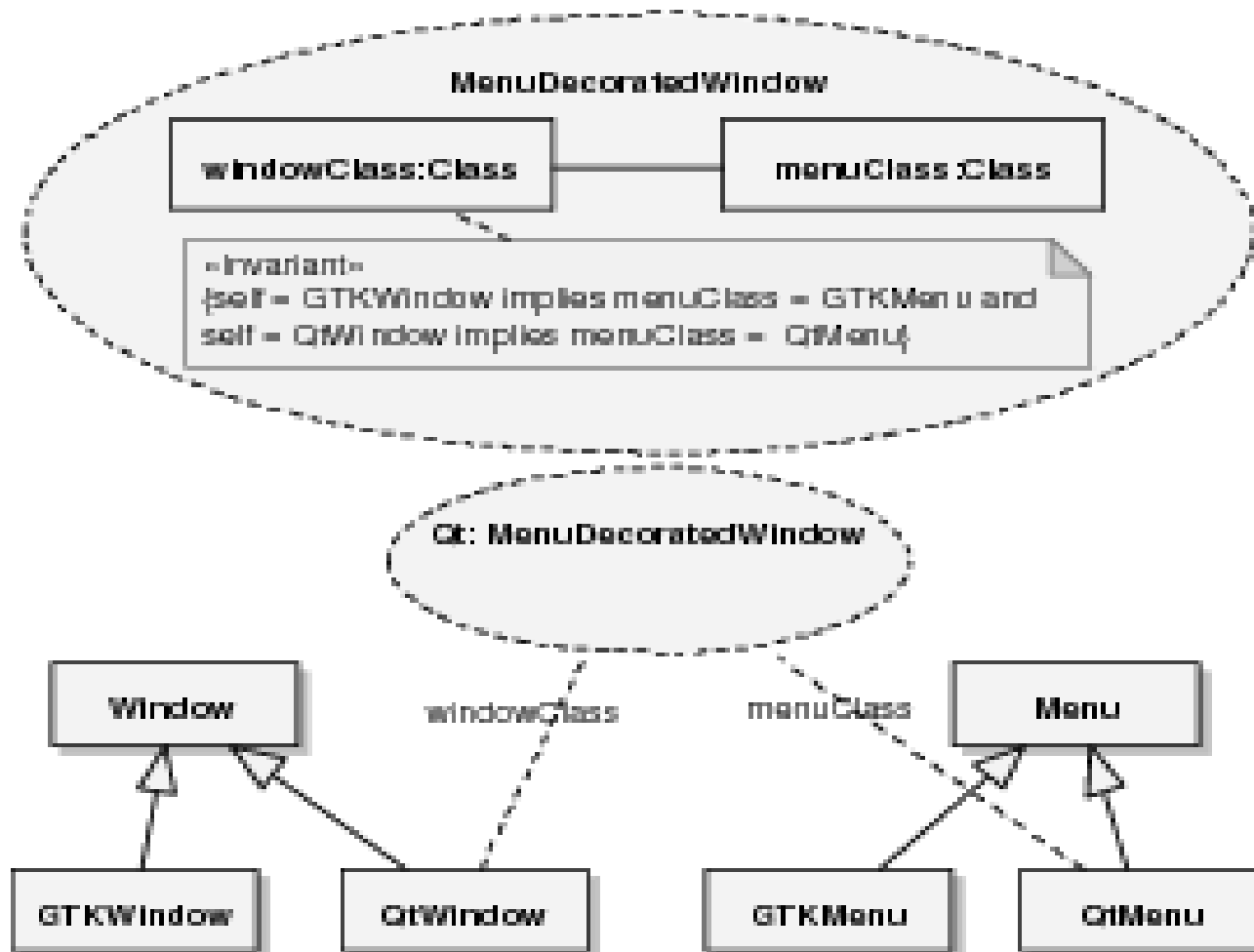
Problem 2: SalesPoint Framework

- ▶ Catalog and Stock hierarchies must be isomorphic
- ▶ Dynamic constraint; domain-specific



Problem 3: Parallel Hierarchies

- ▶ Window types must be varied parallelly
- ▶ Static constraint, but technical



Problem 4: Dynamic Assumptions

- ▶ Other dynamic contract checks

Null-checks
Range checks
Sortedness of ordered collections

Dynamic technical constraints

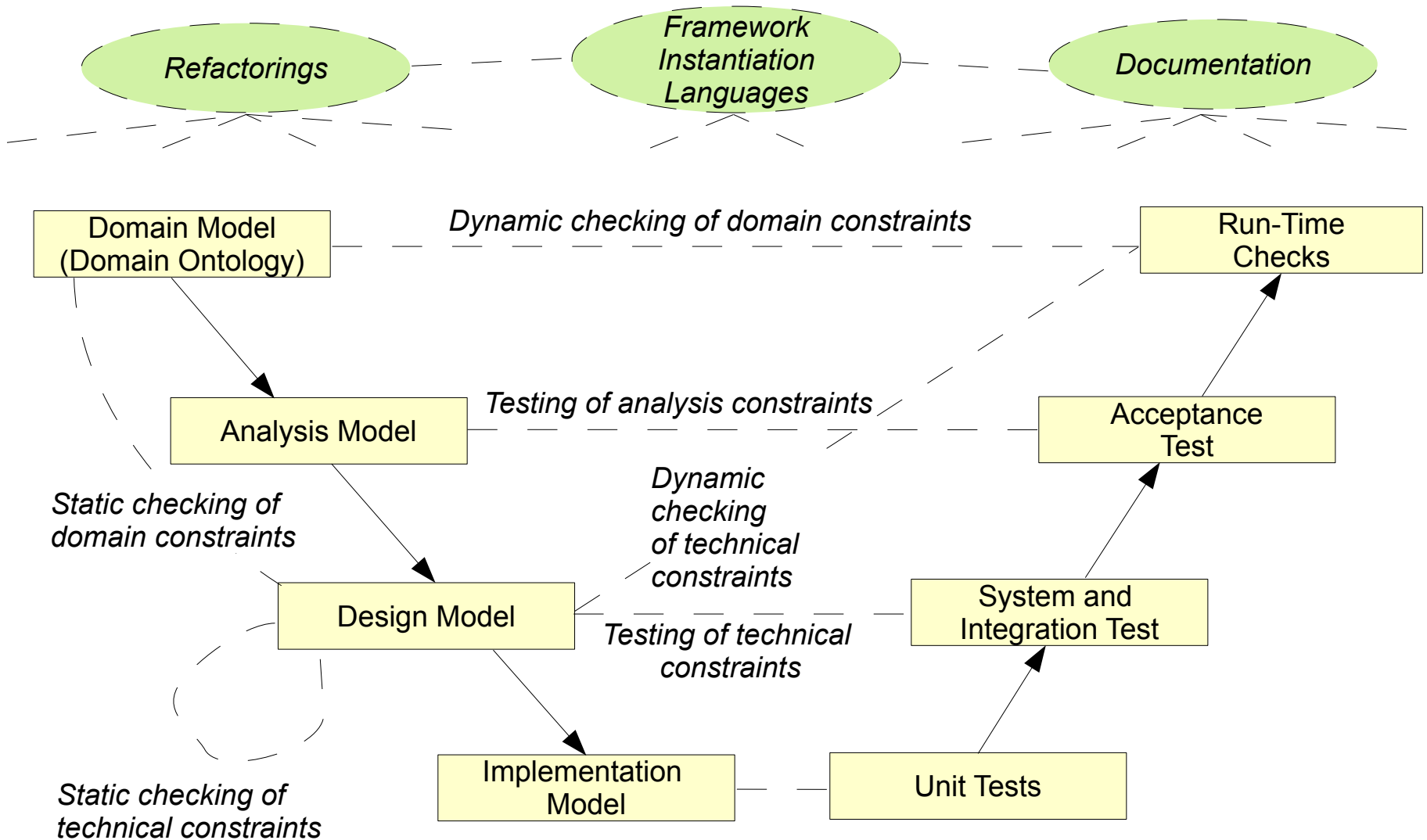
Classification of Instantiation Constraints

Facet 1: Stage	Static	Dynamic
Facet 2: Cause		
Domain-specific (analysis-related)	Car configurator multi-point constraint	SalesPoint isomorphic hierarchies of Catalogs and Stocks
Technical (design-related)	Windows parallel hierarchies	Dynamic assumptions Dynamic contracts



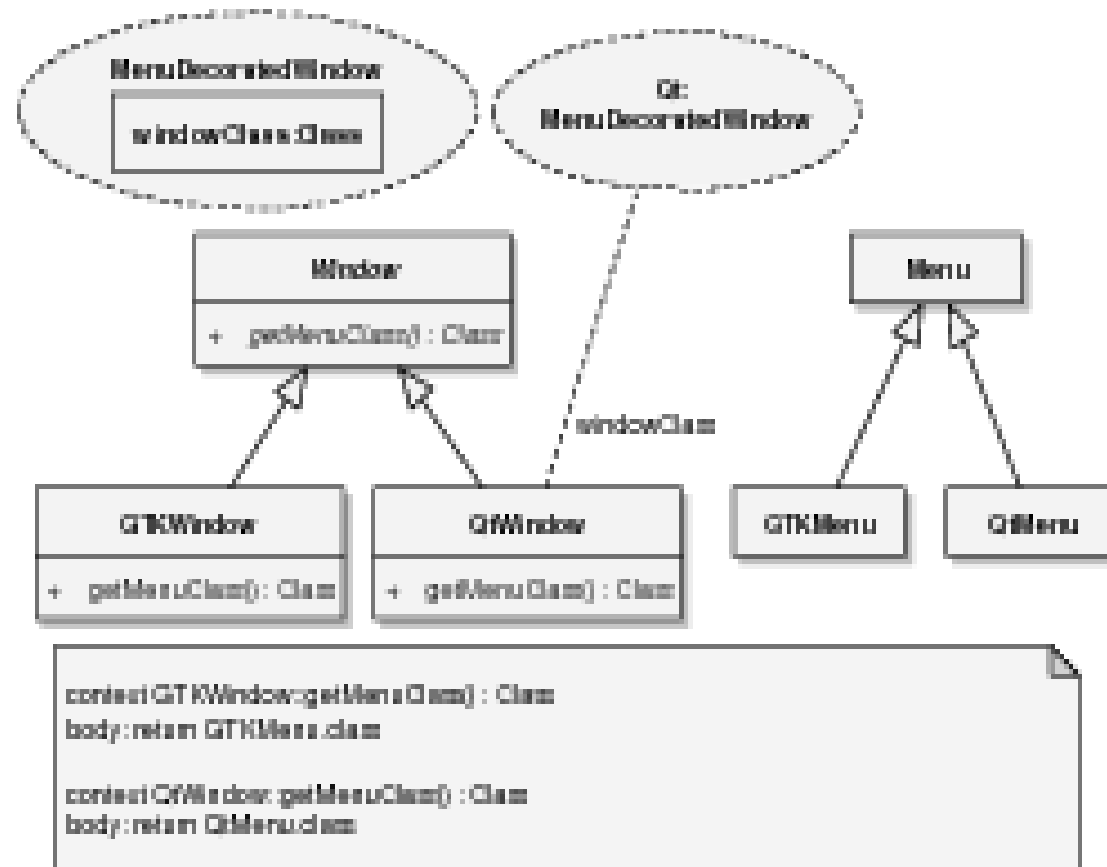
24.2 Remedies for Trustworthy Instantiation

Checking Mechanisms in All Phases of the Life Cycle



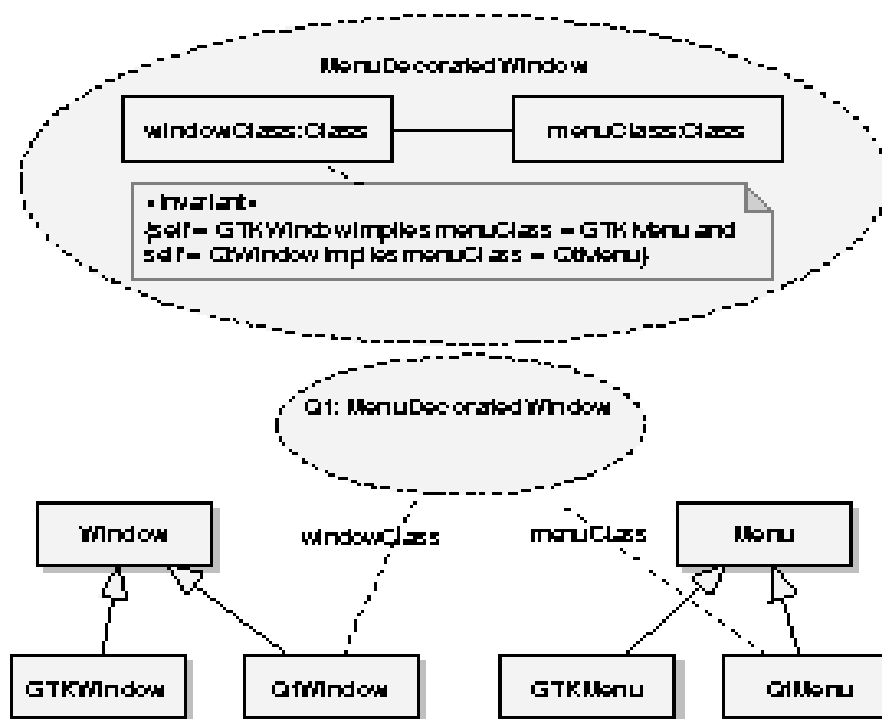
Remedy 1: Refactoring of Multi-Point Constraints

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



Remedy 2: 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

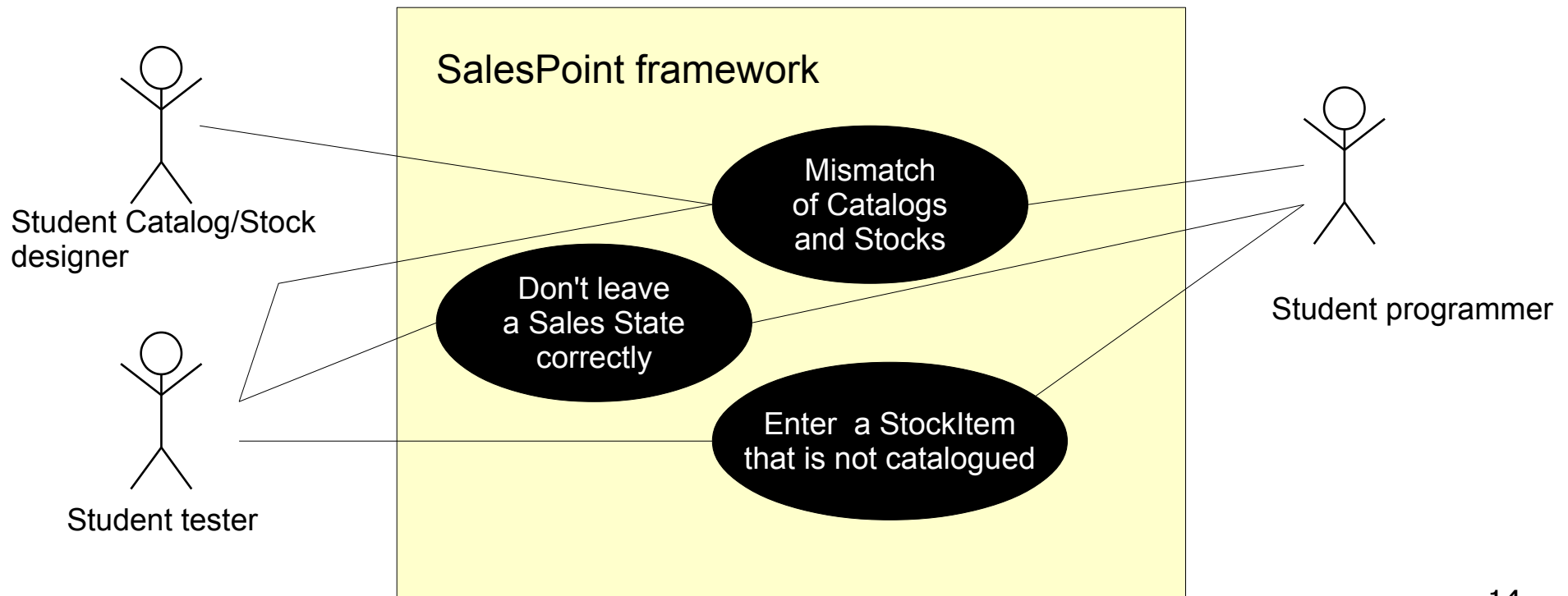


Remedy 3: Framework Testing

- ▶ Frameworks must be *negatively tested*
 - Beyond functional tests (positive tests), censorious negative tests for the behavior in case of misinstantiaton 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
- ▶ [Sindre, G., Opdahl, A.L. Eliciting security requirements with misuse cases. Requirements Engineering 10 (2005) 34–44]
- ▶ Used to describe system abuse (intrusion, fraud, security attacks)
- ▶ Coarse-grain technique to specify also *framework misuse*



Negative Test Table Entries

- ▶ 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)
 - Onput parameters of method
 - Output parameters
 - Reaction, state afterwards

Testcase	Testclass	Input	Output	Reaction
		String date	Date d1	
			day month year	
1	positive	1. Januar 2006	1 1 2006	
2	positive	05/12/2008	5 12 2008	
3	positive	January 23, 2007	23 1 2007	
4	negative	Mak 44, 2007		failure
5	negative	March 44, 2007		failure

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

Testcase	Testclass	Input		Reaction	
		hook 1	hook 2		
1	pos. static	QtMenu	QtButton		
2	pos. static	GtkMenu	GtkButton		
3	neg. static	QtMenu	GtkButton	error „for multi-point, use parallel classes“	
4	neg. static	GtkMenu	QtButton	error „for multi-point, use parallel classes“	

Derivation of JUnit Test Cases

- ▶ From every test table entry dealing with a dynamic constraint, a JUnit test case is derived (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

Remedy 4:

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

plugin.xsd

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

example.exsd

```
<schema ...>
<element name = "extension">
  <attribute name="point" type="string"
    use="required" />
  <attribute name="class" type="string"/>
</element>
</schema>
```

<<instance-of>>

<<instance-of>>

plugin.xml (extended)

```
<plugin name="extended" ...>
<extension-point
  id="example"
  name="example"
  schema="example.exsd"/>
</plugin>
```

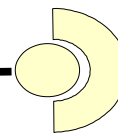
plugin.xml (extending)

```
<plugin name="extending" ...>
  <extension point="example"
    class="package org.savga.Runner"/>
</plugin>
```

<<refers-to>>

<<instance-of>>

<<extends>>



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)

Remedy 5: 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

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

Remedy 6: 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*

```
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



The End