Modellbasierte Softwareentwicklung

Wilhelm Schäfer
FG Softwaretechnik
Raum ZM1.02-09
wilhelm@uni-paderborn.de
Sprechstunde: Dienstags 14:00 – 15:00 Uhr
Recap: Reasons for Failure

- RACE Conditions, Concurrency Problems (Therac25)
- Wrong or Incomplete Requirements (AIRBUS Crash)
- Wrong or Incomplete Environment Assumptions (Lake Constance)
Main Idea of Model Driven Development

- Code generation from a detailed design model
  - (in UML: PSM – Plattform Specific Model)
- Reduces complexity, abstracts from implementation details
- Enables advanced formal (model) checking techniques, e.g. High Level Petri Nets
- Facilitates changes
- Similar to the step from assembler to current programming languages
Bidirectional Associations

Model (Class diagram)

Registrar

1

Shuttle

* 1

class Registrar {
    private Set shuttles = new HashSet();
    public void remove(Shuttle s) {
        shuttles.remove(s);
    }
}

class Shuttle {
    private Registrar registrar;
    public void remove() {
        registrar = null;
    }
}

(Insufficient or even Incorrect) Implementation
Bidirectional Associations (continued)

```java
import de.upb.tools.fca.*;

public class Registrar {
    private FHashSet shuttle;
    public boolean removeFromShuttle(Shuttle value) {
        boolean changed = false;
        if ((this.shuttle != null) && (value != null)) {
            changed = this.shuttle.remove(value);
            if (changed) {
                value.setRegistrar(null);
            }
        }
        return changed;
    }
    ...
}
```

```java
import de.upb.tools.fca.*;

public class Shuttle {
    private Registrar registrar;
    public boolean setRegistrar(Registrar value) {
        boolean changed = false;
        if (this.registrar != value) {
            if (this.registrar != null) {
                Registrar oldValue = this.registrar;
                this.registrar = null;
                oldValue.removeFromShuttle(this);
            }
            this.registrar = value;
            if (value != null) {
                value.addToShuttle(this);
            }
            changed = true;
        }
        return changed;
    }
    ...
}
```

Correct Implementation
Kinds of Models

- **Structure oriented models** (behaviour added later)
  - e.g. graphs, Class diagrams, Design Patterns, architectural styles

- **Behaviour oriented models**
  - Operational models (e.g. Statecharts, Message Sequence Charts, Graph Grammars)
  - Descriptive (declarative) models
Testing and Model Checking

- Testing
  - White-box and black-box testing
  - Model-based testing
  - Regression testing

- Model Checking
  - Labeled transition systems (Kripke-structures) and
  - Logic (linear time logic, computational tree logic)
Table of Contents

1. Introduction
2. Model Construction
   2.1 Design Patterns and Architecture
   2.2 State Machines
   2.3 Timed Automata and RTSC
   2.4 Graph Grammars
   2.5 Z
3. Code Generation
4. Testing
5. Model Checking
References

Gamma et.al.: Design Patterns, Addison-Wesley
Zündorf: Habilitation and some papers (see Web)
Ghezzi: Fundamentals of Software Engineering
        Addison Wesley
G. Berard et.al.: System and Software Verification,
        Springer
Börger, Stärk: Abstract State Machines: A Method for High-
        Level System Design and Analysis

for more references see Webpage
2.1 Design Patterns and Architecture
Overview

- In Softwareentwurf: Architectures
  - Good, approved ways to structure programs
    - Model-View-Controller, Layered Architectures, SOA

- Here: Design Patterns
  - Good, approved solutions to recurring problems
Patterns for Software?

- The first concepts for employing patterns in the software development have been inspired by Christopher Alexander’s architecture book „A Pattern Language“:
  
  „Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.“ [Alexander+1977]

- Since 1990 a number of workshops about software architecture also discuss design patterns.

- The book „Design Patterns“ by Gamma, Vlissides, Johnson und Helm (GoF) makes patterns popular ([Gamma+1994])

- Since 1994 a specific annual conference named “Pattern Languages of Program Design” (PLoP) is held (later also EuroPLoP (1996) and ChiliPLoP (1997)).
Advantages of Design Patterns?

- A design pattern documents existing and proven design experience that can be reused.
- They support the design and documentation of complex and heterogeneous software architectures.
- Design patterns support design qualities such as evolvability, extendibility, and reusability.
- Design patterns can also be employed when reengineering or analysing an existing software.
Problem: Maintaining Consistency

- Example: NewsChannel

Define a one-to-many dependency between a NewsChannel and several NewsReader so that consistency between the NewsChannel and its NewsReaders is guaranteed. Each time the NewsChannel receives a new News its related NewsReaders should be notified and automatically update their knowledge of the channel.
Design Problem & Solution

- **Forces:** Decoupled objects while keep their states consistent. Manage a dynamically changing number of observers.

- **Solution:** Changes of the subject are propagated to the observers. The observers register themselves at the subject.
A First OO Design (1/3)

- Define a one-to-many dependency between the classes so that the **consistency** between the one **channel** and its **readers** is guaranteed. Each time the channel is modified its observing readers are notified and automatically update their knowledge.

```
NewsChannel
- notify();
+ addObserver();
+ removeObserver();
+ addNews();
+ getLatestNews();

NewsReader
+ update();
```

1   *
Collaborations:

- **NewsChannel**
  - addNews()
  - notify()
  - update()
  - getLatestNews()

- **NewsReader**
  - update()
  - getLatestNews()
A First OO Design (3/3), Discussion

- **Coupling**
  - Strong type coupling with known disadvantages
- **Extendibility**
  - Only one fixed update relation
- **Evolvability**
  - Both types have to evolve somehow coordinated
- **Reusability**
  - Result and code is specific for each similar notification relation
What is a Design Pattern?

- A description of an object-oriented design technique which names, abstracts and identifies aspects of a design structure that are useful for creating a reusable object-oriented design.

The design pattern identifies classes and instances, their roles, collaborations and responsibilities. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in the presence of other design constraints, and the consequences and trade-offs of its use.

(see [Gamma+1994])
Describing Design Patterns

(1/2)

Name: Clear meaningful name
Intent: What problem is solved?
Also Known As: Other well-known names
Motivation: Scenario illustrating a problem and how it is solved using the pattern
Applicability: When can the pattern be applied?
Structure: UML model of the classes involved
Participants: Classes/objects involved and their responsibilities
Collaborations: How the participants collaborate to achieve their common goal
Consequences: Trade-offs and achieved flexibility
Constraints: Facts that may restrict their usage
Implementation: What pitfalls or techniques are relevant for the pattern
Sample Code: Code fragments illustrating the implementation (Java)
Known Uses: Domain and situation the problem occurs
Related Patterns: Differences and useful combinations with other patterns
Design Pattern Solution: Observer Pattern (1/7)

- **Intent**: Define a one-to-many dependency between objects so that *consistency* between the one *subject* and its *observers* is guaranteed. Each time the subject state is changed its observers are notified and automatically update their knowledge of the subject.

- **Also Known As**: Dependents, Publish-Subscribe

Observer Pattern (2/7)

- **Motivation**: The state of multiple objects (observer) depends on another object (subject). It is important to keep them consistent. All changes of the subject should be made observable by the other objects. A single subject may have multiple observers. However, tight coupling between the classes has to be avoided, because that reduces their reusability.
Observer Pattern (3/7)

- **Applicability**: use in the following situations:
  - Abstraction with **two aspects** decomposed in this manner can be reused independently
  - **Propagate** required changes between an unknown number of objects
  - When other objects should be **notified** without making assumption who they are
Observer Pattern (4/7)

**Subject**

+ attach(o : Observer)
+ detach(o : Observer)
+ notify()

```java
notify() {
  for (Observer o : observers) {
    o.update()
  }
}
```

**ConcreteSubject**

- subjectState : State

```java
+ getState()
+ setState(state : State)
```

```java
return subjectState;
```

**Observer**

+ update()

```java
for (Observer o : observers) {
  o.update()
}
```

**ConcreteObserver**

- observerState : State

```java
+ update()
```

© Fachgebiet Softwaretechnik, Heinz Nixdorf Institut, Universität Paderborn

MBSE - 2.1 Design Patterns and Architecture

HEINZ NIXDORF INSTITUT
Universität Paderborn
Softwaretechnik
Prof. Dr. Wilhelm Schäfer

17.10.2011

25
Observer Pattern (5/7)

- **Participants:**
  - **Subject:**
    - Knows its observers
    - Provides an interface to attach and detach observers
  - **Observers:**
    - Defines an interface for update
  - **ConcreteSubject:**
    - Store state of interest to ConcreteObserver objects
    - Send notification to observers when state changes
  - **ConcreteObserver:**
    - Maintain reference to ConcreteSubject
    - Store state copy of subject
    - Implement Observer interface for updates
Observer Pattern (6/7)

Collaborations:

- :ConcreteSubject
  - setState()
  - notify()
    - update()
  - getState()
- :ConcreteObserver
  - update()
- :ConcreteObserver
  - update()
  - getState()
Observer Pattern (7/7)

- **Consequences:**
  - Avoid tight coupling between subject and observer classes
  - Support for broadcast communication
  - Because we abstract from the concrete update processing unexpected costly cascades of updates may result
Characteristics of Patterns

- A pattern is a detectable form in a specific **context**
- Patterns provide a scheme to **detect, compare and create** pattern instances
- Patterns are the **essence of the experience** and analysis for a **recurring** problem
- Patterns have an inner **structure and dynamics**
- Patterns can be flexible **combined**.
- Patterns provide a **vocabulary** for designing a system and therefore simplify communication
<table>
<thead>
<tr>
<th>Scope</th>
<th>Class</th>
<th>Purpose</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Creational</td>
<td>Structural</td>
<td></td>
</tr>
<tr>
<td>Class</td>
<td>Factory Method</td>
<td>Adapter</td>
<td>Interpreter Template Method</td>
</tr>
<tr>
<td>Object</td>
<td>Abstract Method Builder Prototype Singleton</td>
<td>Adapter Bridge Composite Decorator Facade Flyweight Proxy</td>
<td>Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor</td>
</tr>
</tbody>
</table>

**Purpose**

- **Creational**
  - Factory Method
- **Structural**
  - Adapter
  - Chain of Responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Visitor
Redesign

- Redesign is time-consuming and thus expensive
- Design pattern can avoid redesign in some cases
- The main causes for redesign can be avoided.
Causes for Redesign (1/2)

1. Creating an object by specifying a class explicitly
   **Design patterns**: Abstract Factory, Factory Method, Prototype

2. Dependence on specific operations
   **Design patterns**: Chain of Responsibility, Command

3. Dependence on hardware and software platform
   **Design patterns**: Abstract Factory, Bridge

4. Dependence on object representations or implementation:
   **Design patterns**: Abstract Factory, Bridge, Memento, Proxy
Causes for Redesign (2/2)

5. Algorithmic dependencies
   **Design patterns**: Builder, Iterator, Strategy, Template Method, Visitor

6. Tight coupling
   **Design patterns**: Abstract Factory, Bridge, Chain of Responsibility, Command, Facade, Mediator, Observer

7. Inability to alter classes conveniently
   **Design patterns**: Adapter, Decorator, Visitor
Different Kinds of Pattern

- Common:
  - Idioms (class level)
  - Design pattern (multiple classes)
  - Architecture pattern (overall architecture)

- Other areas:
  - Analysis pattern
  - Process pattern (e.g., for tests, reviews, …)
  - Organisational patterns

- New derived form:
  - Anti patterns: describe often chosen wrong solutions for a specific problem and suggest means to improve them
Patterns in UML

- Parameterized collaborations

**Definition:**

- **Structure:** classes, inheritance, associations, shared aggregation, composition
- **Dynamics:** collaborations, sequence diagrams, abstract code

**Usage:**

- **Pattern:**
- **Assign roles:** Class\(<n>\)