2.1 Design Patterns and Architecture (continued)

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
Reminder:
What are Design Patterns?

- Approved solutions to recurring software engineering tasks
- Reusable design ideas (class structures, modularization, collaboration, …)

- Patterns have an intent (e.g. making an algorithm easily exchangeable and simplify future changes) as well as benefits and drawbacks (some future changes become easy, others may become more difficult)

- Correctly applied, patterns increase software quality by increasing flexibility and reducing maintenance effort

- Most famous design patterns:

  [Gamma et al., Design Patterns – Elements of Reusable Object-oriented Software, 1995]
The 23 Gang-of-Four (GoF) Design Patterns

- Gang-of-Four = authors of the design patterns book: Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- In this lecture, we will talk about the highlighted patterns:

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Creational</th>
<th>Structural</th>
<th>Behavioral</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Class</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object</td>
<td><strong>Abstract Factory</strong></td>
<td><strong>Object Adapter</strong></td>
<td><strong>Chain of Responsibility</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Builder</strong></td>
<td><strong>Composite</strong></td>
<td><strong>Command</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Prototype</strong></td>
<td><strong>Decorator</strong></td>
<td><strong>Iterator</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Singleton</strong></td>
<td><strong>Facade</strong></td>
<td><strong>Mediator</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Flyweight</strong></td>
<td><strong>Memento</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Proxy</strong></td>
<td><strong>Observer</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>State</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Strategy</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td><strong>Visitor</strong></td>
</tr>
</tbody>
</table>
Example: Developing a Class Diagram Editor

Editor preview:
Eclipse

- Open source IDE framework (Integrated Development Environment) originally developed by IBM in 2001
- Built to be extended with new editors (plug-ins)
- Modular, flexible architecture built by application of numerous design patterns

*Erich Gamma, Kent Beck, Contributing to Eclipse – Principles, Patterns, and Plug-ins, 2004*
Eclipse

- Offers common functionality for editors
- Good starting point for our class diagram editor
How could a class diagram editor be structured?

Typically, there are several design patterns in an implementation of such an editor. Let’s look at some examples.
Model-View-Controller (MVC) Paradigm / Architectural Style

- **Goal:**Decouple the view from the model to easily add new views without having to modify the model.

MVC is not really a design pattern, but an architectural style. MVC uses several patterns, e.g., Observer & Command.
**Observer Pattern in MVC**

- On modification of the model, the model notifies the controller about the change (e.g. changed class name) and the controller redraws the view.

**Diagram:**

- **Subject**
  - Subject
  - notify() : void
  - ConcreteSubject
  - has some accessible state
  - subject

- **Observer**
  - Observer
  - update() : void
  - ConcreteObserver
  - uses subject’s state

- **Classes (Model)**
  - register

- **EditParts (Controller)**
  - call update() on each observer

- **Figures (View)**
  - Track
    - nr: Long
    - length: Integer
  - RailCab
    - maxSeats: Integer
    - member
  - Convoy
Command Pattern

Example in MVC:

What is the idea behind the pattern?
Command Pattern – Motivation

- Editor’s palette provides different operations

Problems:
- How can the operations be handled uniformly?
- How can these operations be undone, re-done?
- How can the operations be parameterized, e.g. with coordinates?
**Command Pattern**

- Basic idea: encapsulate each operation and its parameters in an object and provide a uniform interface.

```
+ execute()
+ undo()
+ redo()
```

```
Command
```

```
CreateClassCommand
+ ...
```

```
CreateAttributeCommand
+ ...
```
General Structure: Command

Consequences

- Adding new operations is easy
- All operations are handled uniformly
- Each operation can undo its action
- Parameters and internal state saved in Command object

```
execute() {
    ...
    receiver.action();
    ...
}
```
Design Pattern: Command

- **Intent:**
  - Encapsulate actions in (parameterized) objects to ease adding actions and provide general undo-mechanism

- **Also known as:**
  - Action, Transaction

- **Motivation**
  - GUI elements (e.g. buttons) represent operations performed by the user
  - Functionality not implemented in the GUI
  - Command objects encapsulate user operations and can be undone

- **Related Pattern:**
  - Composite can be used to implement combined commands
Composite Pattern – Motivation

Graphical elements in a class diagram are hierarchical and should be drawn uniformly

- Class figure contains
  - Attribute figures
  - Method figures

Naive solution: class diagram editor stores references to all figures and draws each of them by running through the hierarchy and calling their draw operations

Drawbacks:
- Each type of figures has to be handled differently
- Draw operations have to be called explicitly on each figure
### Composite Pattern

- All figures have the same interface
- Some figures can have contained subfigures
- All figures are drawn uniformly
- Each figure is responsible for drawing itself and its child figures

```java
public void drawFigure() {
    for(Figure f : getChildren()) {
        f.drawFigure();
    }
}
```

**Figure**

- `+ drawFigure() : void`

**AttributeFigure**

- `+ drawFigure() : void`

**ClassFigure**

- `+ drawFigure() : void`
- `+ add(child : Figure) : void`
- `+ remove(child : Figure) : boolean`
- `+ getChild(index : int) : Figure`

```java
public void drawFigure() {
    for(Figure f : getChildren()) {
        f.drawFigure();
    }
}
```
**General Structure: Composite**

- **Client**
  - `+ operation()`

- **Component**
  - `+ operation()`
  - `+ add(component : Component) : Void`
  - `+ remove(component : Component) : Boolean`
  - `+ getChild(index : int) : Component`
  - `+ operation()`

- **Leaf**
  - `+ operation()`

- **Composite**
  - `operation() { for(Component c : children) { c.operation(); } }`
**Implementation Variants**

**Clean Variant:**
- Distinguish leaves from composites (only composites can have children)

**Efficient Variant**
- Completely uniform treatment of all elements, but leaves could have children

```java
Client
+ operation()

Component
+ operation()
children

Leaf
+ operation()

Composite
+ add(component : Component) : Void
+ remove(component : Component) : Boolean
+ getChild(index : int) : Component
+ operation()

Client
+ operation() { for(Component c : children) { c.operation(); } }

Component
+ add(component : Component) : Void
+ remove(component : Component) : Boolean
+ getChild(index : int) : Component
+ operation()

Leaf
+ operation()

Composite
+ operation()

Client
+ operation() { for(Component c : children) { c.operation(); } }
```

![Diagram of Clean and Efficient Variants](image)
**Design Pattern: Composite**

- **Intent:**
  - Compose objects into a tree to represent part-whole hierarchies where the composition and its parts are treated uniformly

- **Related Patterns**
  - Often the Decorator Pattern (see assignments) is used combined with the Composite Pattern.
  - The Iterator Pattern can be used to traverse composite structures.
  - The Visitor Pattern (see assignments) can be used to encapsulate behaviour that would otherwise be distributed across the composites and leafs.
Strategy Pattern – Motivation

- Outline view in Eclipse lists all elements of a class diagram, e.g. classes, attributes, and methods.
  - List can be sorted differently sorted
    - Alphabetically
    - Order of appearance
    - Other sorting algorithms could be added in future

- Naive solution: implement sorting algorithms directly in outline view.
  - Drawbacks: adding or replacing the algorithms is complicated.
Strategy Pattern

OutlineContentProvider

+ sortTree(...)

sortTree(...) {
  ...
  sortingStrategy.sort(...);
  ...
}

Sorting

+ sort(...)

OutlineContentProvider

+ sortTree(...)

sortingStrategy

Sorting

+ sort(...)

AlphaNumSort

+ sort(...)
General Structure: Strategy

Context
+ request()

Strategy
+ algorithm()

ConcreteStrategy
+ algorithm()

```java
request() {
    ...
    strategy.algorithm();
    ...
}
```
Design Pattern: Strategy

- Intent:
  - Encapsulation of a family of dynamically interchangeable algorithms

- Also Known As:
  - Policy

- Problem:
  - “Best" algorithm can often not be uniquely determined due to trade-offs:
    - Runtime <=> Memory
    - Runtime <=> Quality
  - Best solution depends on problem characteristics: e.g. for sorting: data structure, stability, problem size, user needs,...

- Related Pattern:
  - Strategy objects often make good flyweights
Basic Concept in Patterns: Delegation

- Used, e.g. in the Strategy pattern
- An objects delegates a task to another object

Another example:

<table>
<thead>
<tr>
<th>Window</th>
<th>Rectangle</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ getPosition() : Point</td>
<td>- width : int</td>
</tr>
<tr>
<td></td>
<td>- height : int</td>
</tr>
<tr>
<td></td>
<td>- x : int</td>
</tr>
<tr>
<td></td>
<td>- y : int</td>
</tr>
<tr>
<td></td>
<td>+ getLocation() : Point</td>
</tr>
<tr>
<td></td>
<td>+ ...</td>
</tr>
</tbody>
</table>

Point getPosition() {
    return bounds.getLocation();
}

Point getLocation() {
    return new Point(x, y);
}

Advantages:

- Loose coupling, Rectangle class can be replaced at runtime and Window does not have to subclass Rectangle (it is not a rectangle)
Abstract Factory Pattern – Motivation

- Class diagram editor is to open two different formats of class diagrams
- Class models differ, but are conceptionally similar
- Naive solution: direct call of the constructor for each type of class and its attributes, etc.
- Drawbacks: all types of diagram elements have to be treated differently, adding new types of diagrams is difficult

**Diagram:**

```
UMLClass
- identifier: String
- visibility: int

RRClass
- name: String
- visibility: String

UMLAttribute
- identifier: String
- visibility: int

RRAttribute
- name: String
- visibility: String
```
Abstract Factory Pattern

- Basic idea: introduce an interface for similar objects and an interface for the creation of a family of similar objects
- Creation operations defined in a factory
- Factory subclasses determine what to create (products)
- Ensures consistent creation of product families and type secure programs
General Structure: Abstract Factory

AbstractFactory

+ createProductA() : AbstractProductA
+ createProductB() : AbstractProductB

ConcreteFactory1

ConcreteFactory2

AbstractProductA

ConcreteProductA1

ConcreteProductA2

AbstractProductB

ConcreteProductB1

ConcreteProductB2

Client

MBSE - 2.1 Design Patterns and Architecture
**Design Pattern: Abstract Factory**

- **Intent:**
  - Structure creation of related objects

- **Problem:**
  - How to manage the creation of families of related or dependent object?

- **Solution:**
  - Encapsulate object creation within one class
  - Redefine in subclasses as required

- **Related Patterns:**
  - Often a concrete factory is also a singleton
  - Often realized using the Factory Method Design Pattern
Adapter Pattern – Motivation

- The implementation of the different class models to be opened with your editor cannot be changed, the source code is not under your control, i.e. you cannot introduce a common interface for class model elements.

- E.g. you want to open Ecore class models in your editor and cannot introduce a common interface for UMLClass and EClass objects (you cannot modify EClass).

<table>
<thead>
<tr>
<th>UMLClass</th>
<th>EClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifier: String</td>
<td>name: EString</td>
</tr>
<tr>
<td>visibility: int</td>
<td>abstract: EBoolean</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>UMLAttribute</td>
<td>EAttribute</td>
</tr>
<tr>
<td>identifier: String</td>
<td>name: String</td>
</tr>
<tr>
<td>visibility: int</td>
<td></td>
</tr>
</tbody>
</table>
Adapter Pattern

Basic idea: convert the available interface into the required interface by means of an adapter (wrapper) object and delegation.

```
public String getClassName() {
    return original.getName();
}
```
General Structure: Adapter

Client

Target

+ request()

Adapter

+ request()

Adaptee

+ specificRequest()

```
request() {
    adaptee.specificRequest();
}
```
Design Pattern: Adapter

- **Intent:**
  - Transform a call to another call-format needed by other class
  - Adapt external interfaces (libraries, modules, ...) to the required internal interfaces

- **Also Known As:**
  - Wrapper

- **Related Patterns:**
  - The Bridge is used to separate the interface from its implementation while the adapter is used to change the interface to an existing object
  - Proxies are representatives for other objects, but do not change the interface
  - Decorator keeps the interface and adjusts functionality while the Adapter adjust the interface and keeps the functionality
Bridge Pattern – Motivation

- Variance both, on a conceptional and implementation level:
  - Conceptional variance
    - Warning dialog
    - Info dialog
    - Error dialog
  - Implementation variance
    - Platform: e.g. Windows vs. Linux
    - GUI framework: e.g. AWT vs. Swing vs. SWT
- Naive solution: Represent both dimensions of variance in one class hierarchy.
- Drawback: Abstraction and implementation cannot be modified or extended independently.
Bridge Pattern

- Basic idea: separate concepts and implementations in separate class hierarchies, use the different implementation via delegation
Bridge Pattern

Consequences

- concepts and implementations can vary quite independently
- Adding or changing implementations for different platforms / frameworks without changing in the rest of the program

```
Bridge

Abstraction
--- Dialog
  - drawRect(…)
  - drawText(…)

Implementation
--- DialogImpl
  - drawLine(…)
  - drawText(…)

impl
  - impl.drawLine(…)
  - impl.drawLine(…)
  - impl.drawLine(…)
  - impl.drawLine(…)

WarningDialog
  - drawOkButton()

DecisionDialog
  - drawYesButton()
  - drawNoButton()

WinDialogImpl
  - drawLine(…)
  - drawText(…)

LinuxDialogImpl
  - drawLine(…)
  - drawText(…)
```

General Structure: Bridge

Abstraction
+ operation()  

impl

Implementor
+ operationImpl()

operation() {
    implementation.operationImpl();
}

RefinedAbstraction

ConcreteImplementor
+ operationImpl()
Design Pattern: Bridge

- Intent:
  - Decouple abstraction and implementation to vary them independently
- Also known as:
  - Handle / Body
- Motivation:
  - Hierarchy of classes with similar functionality
  - Separate abstract class hierarchy from implementation class hierarchy
  - Achieve platform independence using abstract super classes and platform specific subclasses
- Related Patterns:
  - An Abstract Factory may create and configure a Bridge
Singleton Pattern – Motivation

- Class diagram editor is realized as a plug-in
- Each plug-in in Eclipse is only allowed to exist once in a running Eclipse workbench
- Plug-ins should be accessible by other plug-ins

- Naive solution: global variable to hold the single plug-in instance
- Drawback: no one ensures, there is only one plug-in instance
Singleton Pattern

Basic idea:

- Single access point to the plug-in instance
- Private constructor
- Static method provides access and ensures to create only one instance

Exemplary implementation in Java:

```java
public class ClassDiagramPlugin {
    private static ClassDiagramPlugin instance;

    public static ClassDiagramPlugin get() {
        if (instance == null) {
            instance = new ClassDiagramPlugin();
        }
        return instance;
    }

    private ClassDiagramPlugin() {
        ...
    }
}
```
**General Structure: Singleton**

<table>
<thead>
<tr>
<th>Singleton</th>
</tr>
</thead>
<tbody>
<tr>
<td>- uniqueInstance : Singleton</td>
</tr>
<tr>
<td>- data : SingletonData</td>
</tr>
<tr>
<td>+ instance() : Singleton</td>
</tr>
<tr>
<td>+ getData() : SingletonData</td>
</tr>
<tr>
<td>+ singletonOperation() : void</td>
</tr>
</tbody>
</table>

... return uniqueInstance;
**Design Pattern: Singleton**

- **Intent:**
  - Ensure a class has only one instance

- **Problem:**
  - How to manage the creation and access to this object?

- **Solution:**
  - Static access to the instance
  - Create instance if not present
  - **But!** Depending on the platform or programming language it is difficult to always ensure, there is only one instance, e.g. in Java, if concurrent execution or multiple virtual machines are involved, another implementation of the pattern is needed*.  

- **Related Patterns:**
  - Often a concrete factory is also a singleton

*See e.g.: http://www.ibm.com/developerworks/java/library/j-dcl/index.html
Creational Design Patterns

- Motivation
  - Abstract from instantiation process
  - Make system independent of object creation
- Details:
  - What class? Who creates? How? When?
- Problems:
  - Class creation spread over the code (no single point of change)
- Basic themes:
  - Encapsulate knowledge about concrete classes
  - Hide how objects are created and composed
- Presented here:
  - Singleton and Abstract Factory
Structural Patterns

- **Motivation:**
  - Compose objects to realize new functionality
  - Flexible structures that can be changed at run-time

- **Problems:**
  - Fixed class for every composition is required at compile time otherwise

- **Presented here:**
  - Composite Pattern, Bridge Pattern, Adapter Pattern
Behavioral Patterns

Motivation:
- Assign responsibilities to different objects to encapsulate operations and make the operations replaceable and extendable

Problems:
- How to change the behaviour in different classes at runtime?

Presented here:
- Observer Pattern (last week)
- Strategy Pattern
- Command Pattern
Discussion & Summary

- Design patterns enable **reuse** of design experience at the design level!
- Each design pattern application should be guided by analyzing the trade-offs
- Design patterns are **domain-independent** concepts, but their application in a specific context is guided by the domain-specific requirements (e.g. time, space)
- Design patterns generalize concepts that have been successfully employed in real-world applications and therefore are **"bullet proofed"**
- Design patterns characterize “good” designs (well-defined class hierarchies and object interactions)

**Beware!** Design patterns are no silver bullets!
**Only use them when it makes sense.**