Software Engineering

Lesson Design Pattern
General
v1.0

Uwe Gühl

Winter 2007/ 2008

Contents

- Introduction
- Overview
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns
- Reuse

Introduction

- Basic [GHJ+95]:
  - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, „Design Patterns - Elements of Reusable Object-Oriented Software“, 1995
- Practical Reference [Coo98]
  - Practical reference with Java Example Code from James W. Cooper: „The Design Patterns Java Companion“, http://www.patterndepot.com/put/8/JavaPatterns.htm, 1998
- More: [AIS+77], [CV02], [Joh92], [JZ91]

Introduction

- Meanwhile exist more pattern collections:
  - Analysis Patterns
  - Process Patterns
  - Architecture Patterns
  - Test Patterns
  - Anti Patterns
- „One of the ways that I measure the quality of an object-oriented system is to judge whether or not its developers have paid careful attention to the common collaborations among its objects“ (Grady Booch)

Introduction

- Design Patterns
  - describe successful applied solutions for perseverative problems
  - were described first by C. Alexander concerning architectural problems [AIS+77]
  - are found and not invented

Introduction

- Design Patterns
  - improve communication
    - “We use the Decorator Pattern to be able to represent different options of our product”
    - Discussion in a higher abstract level, not too much discussion about details
  - improve code
    - `public class Espresso extends Decorator`
    - `public class Results implements Observable`
    - `// We use Proxy here to ...`

Introduction

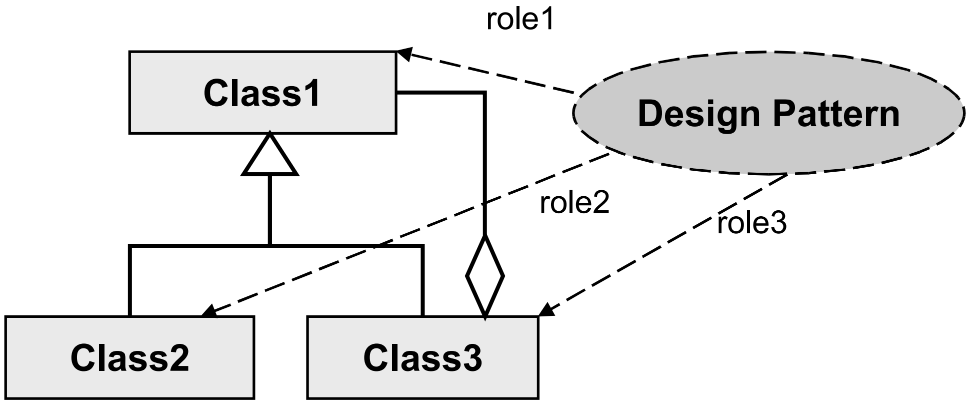
- Difference by size
  - Architectural pattern
    Solutions for preliminary design (Example: Multi level architecture)
  - (Ordinary) Design Pattern
    Solutions for problems in detailed design, independent from programming languages
  - Idioms
    Programming language depending solutions (Do‘s and Don‘ts)

Introduction

- Elements of a Design Pattern
  - Pattern name (for efficient Communication)
  - Problem description - problem to be solved by the design pattern
  - Problem context – to describe when the pattern should be used (and when not!)
  - Solution of the problem
  - Consequences (Pros and cons)

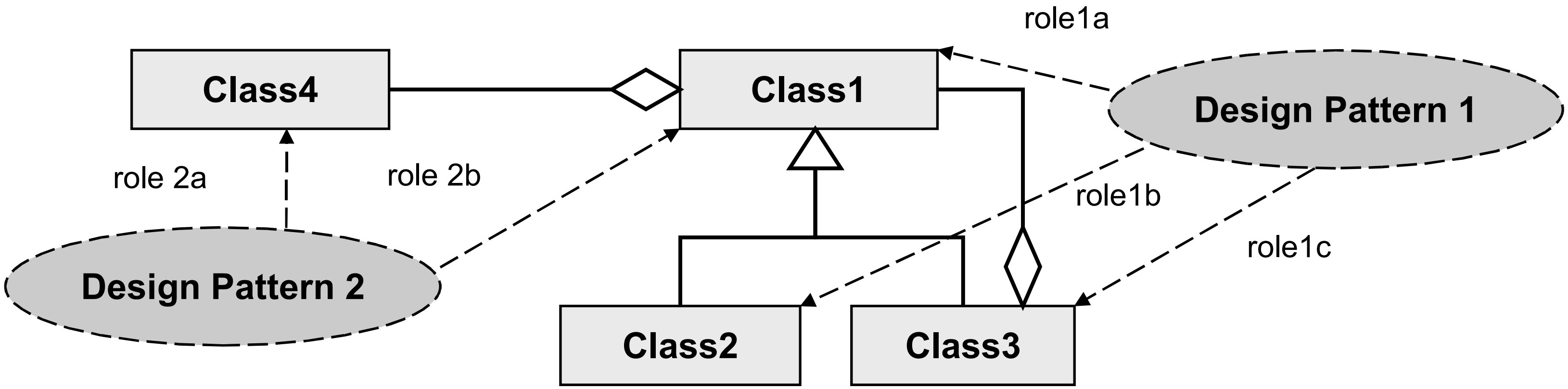
Introduction

- Description in UML
  - Design Patterns describe roles, which could be assigned by a concrete implementation of corresponding classes
  - A concrete class could play different roles in different Design Patterns at the same time



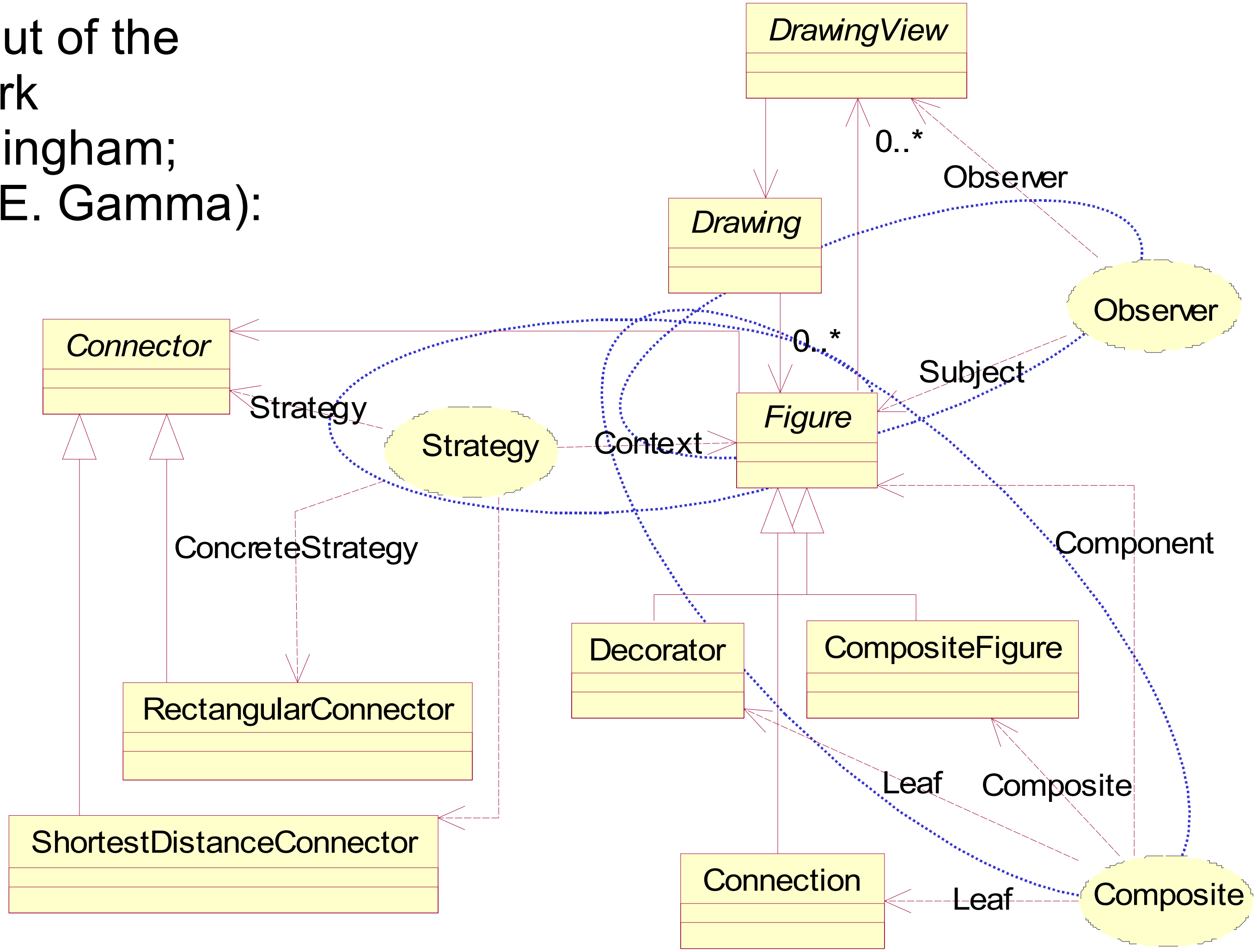
Introduction

- Description in UML – Example



Introduction

Simplified extract out of the HotDraw Framework (K. Beck / W. Cunningham; Java-Version from E. Gamma):



The Class **Figure** is the *Subject* of the **Observer-Pattern**, a *Component* of the **Composite-Pattern** and the *Context* of a **Strategy-Pattern** at the same time.

Introduction

- For what?
  Design Patterns solve Design Problems like
  - finding„right“ objects
  - determination of the granularity
  - specification of interfaces
  - implementation aspects (inheritance)
  - consideration of reuse
  - determination of performance
  - maintainability

Introduction

- How to find?
  I have a problem and I am looking for a Design Pattern to help me solving it
  - Read the overview of individual pattern
  - Study the interaction of the pattern
  - Examine patterns of the same category
  - Reflect, what could be reasons for redesign
  - Think about what should vary in the design
  - Read the description of an interesting design pattern to get an overview

Introduction

- How to find?
  - Understand structure, participants and the collaboration between the participants
  - Study example code
  - Determine names for pattern participants, which are important in the implementation context
  - Define classes
  - Find implementation specific names for methods in the pattern
  - Implement methods to realize responsibilities and interrelationships in the pattern

Overview

- [GHJ+95] describes 23 Patterns, organized in three categories
  - Creational Patterns
    discuss the process of object generation
  - Structural Patterns
    concern about the arrangement of classes
  - Behavioral Patterns
    describe, how objects work together and share responsibility

Overview

- [GHJ+95] depicts most patterns like this
  - Intent
  - Motivation
  - Applicability
  - Structure including Participants and Collaboration
  - Example
  - Consequences
  - Implementation, and
  - Known Uses

Overview

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (Class)	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (Object) Bridge Composite Decorator Facade Flyweigth Proxy	Chain of Reponsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Overview
Creational Pattern

- Creational Patterns
  - deal with the process of object generation
- Scope „Classes”
  - Factory Method
- Scope „Objects“
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton

Overview
Creational Pattern

- **Abstract Factory** ✓
  - defines an interface to generate families of related or dependent objects without specifying their concrete classes
- **Builder** (✓)
  - helps to separate the construction process of a complex object from its representation, so that the same process could create different representations

Overview
Creational Pattern

- **Factory Method** ✓
  - defines a common interface for object generation. Delegates the decision, which concrete class to be instantiated to the subclasses
- **Prototype**
  - specifies the objects, which could be used as a prototypical instance, and creates new objects by copying this prototype

Overview
Creational Pattern

- **Singleton** ✓
  - ensures that a specific class has only one instance and enables a global access to it

Overview
Structural Pattern

- **Adapter** ✓
  - converts the interface of a class, so that a collaboration of classes is possible even with incompatible interfaces
- **Bridge** (✓)
  - decouples an abstraction from its implementation so that both can vary independently

Overview
Structural Pattern

✓

- **Composite**
  - composes objects into tree structures to represent part-whole hierarchies
  - a client could access objects and composites of objects in the same way

Overview
Structural Pattern

- **Decorator** ✓
  - adds additional responsibilities to a specified object instead of all objects of a class dynamically
- **Facade** ✓
  - defines a simplified interface to a larger body of code for a component

Overview
Structural Pattern

- **Flyweight**
  - supports the efficient, cooperative use of a large number of small objects
- **Proxy** ✓
  - A proxy is a class functioning as a placeholder to another object like a network connection or a large object in memory to control access to it

Overview
Behavioral Pattern

- **Chain of Responsibility**
  - used to pass responsibility for handling a request to another class in a chain
- **Command** ✓
  - A command object encapsulates an action and its parameters, supports Undo operations

Overview
Behavioral Pattern

- **Interpreter**
  - as a particular design pattern proposes to implement a specialized computer language to rapidly solve a defined class of problems
- **Iterator** (✓)
  - provides a way to access the elements of an aggregate object step by step without exposing its underlying representation

Overview
Behavioral Pattern

- **Mediator** (✓)
  - defines an object to encapsulate the interaction of a set of corresponding objects
- **Memento** (✓)
  - extracts the state of another object without violating its encapsulation

Overview
Behavioral Pattern

- **Observer** ✓
  - defines a 1:n relationship, so that if one object is changed all dependent objects could be informed and updated automatically
- **State** ✓
  - allow an object to change its behaviour when its internal state changes

Overview
Behavioral Pattern

- **Strategy** ✓
  - defines a family of algorithms, encapsulate each one, and make them interchangeable, so algorithms could vary independently from clients using it
- **Template Method** ✓
  - defines the skeleton of an algorithm in an operation, deferring some steps to subclasses

Overview
Behavioral Pattern

- **Visitor**
  - defines a way of separating an algorithm from an object structure.
    New operations could be added to existing object structures without modifying those structures.

Reuse

- Goal:
  Development of flexible reusable Software
- Design Patterns help to achieve this goal!

Reuse

- Aspects of reusability
  - Inheritance and composition
  - delegation
  - Inheritance and parametrized types
  - Designing for Change
  - Internal Reuse – with loose coupling
  - Toolkits – e. g. lists, stream library
  - Frameworks
    - content often concrete special examples of Design Pattern

Reuse

- Extract: What is the difference between Design Pattern and Frameworks?
  - Design Patter are abstract descriptions of solutions, so many different implementations are possible
  - Frameworks could not implement all combinations of design pattern, so frameworks content some realized examples of design pattern
  - Code generators could support the use of design pattern

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Generation of an object by specifying a class explicitly
  - Future Changes are complicated to be realized
  - Idea: Create objects indirectly
  - **Abstract Factory, Factory Method, Prototype**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- **Dependence on specific operations**
  - Specifying a concrete operation gives only one way to satisfy a request
  - Idea: Avoid hard-coded requests
  - **Chain of Responsibility, Command**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Dependence to Hardware and Software platform
  - platform independent software is difficult to port and to maintain
  - Idea: Limit platform dependency
  - **Abstract Factory, Bridge**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Dependence on object representations or implementations
  - If Clients have to „know too much“ about objects, a cascade of changes have to be done if one object is going to be changed
  - Idea: „Information hiding“
  - **Abstract Factory, Bridge, Memento, Proxy**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Algorithm dependencies
  - New, better, and faster algorithm should be usable easily during development
  - Idea: Isolation of algorithms from using
  - **Builder, Iterator, Strategy, Template Method, Visitor**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Tight coupling
  - Tight coupled classes could not be reused in isolation. An update or deletion of such a class is very expensive
  - Idea: Loose Coupling
  - **Abstract Factory, Bridge, Chain of Responsibility**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Extending functionality by subclassing
  - Dependencies in the class hierarchy make extensions difficult
  - Idea: Flexible Extension with composition
  - **Bridge, Chain of Responsibility, Composite, Decorator, Observer, Strategy**

Reuse

Possible reasons for a redesign [pp. 24 GHJ+95]

- Inability to alter classes conveniently
  - Classes are in a commercial library, but modification is necessary
  - Idea: „Workaround“
  - **Adapter, Decorator, Visitor**