Short Introduction to Design Patterns

Lecture 6b

Software Engineering
TDDC88/TDDC93
autumn 2013

Slides by David Broman and Kristian Sandahl
Department of Computer and Information Science
Linköping University, Sweden
kristian.sandahl@ida.liu.se
A Software Life-cycle Model
Which part will we talk about today?

- **Requirements**
- **System Design** (Architecture, High-level Design)
- **Module Design** (Program Design, Detailed Design)
- **Implementation** of Units (classes, procedures, functions)
- **Unit testing**
- **System Testing** (Integration testing of modules)
- **Module Testing** (Integration testing of units)
- **Acceptance Test** (Release testing)

- Validate Requirements, Verify Specification
- Verify System Design
- Verify Module Design
- Verify Implementation

- Project Management, Software Quality Assurance (SQA), Supporting Tools, Education
- Maintenance
Christopher Alexander
Create alternating areas of light and dark throughout the building, in such a way that people naturally walk toward the light, whenever they are going to important places: seats, entrances, stairs, passages, places of special beauty, and make other areas darker, to increase the contrast.
A Design Pattern is a standard solution for a standard design problem in a certain context.

Goal: reuse design information
Example: Facade
Example: Facade

Façade

Diagram showing the façade pattern, with the façade module connected to various modules.
How to describe design patterns?
Intent

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

Motivation

Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. … example …
Facade

Applicability

Use the Facade pattern when:

- you want to provide a simple interface to a complex subsystem. This makes subsystems more reusable and easier to customize.
- there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from other subsystems, thereby promoting subsystem independence and portability.
- you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.
Consequences

The Facade pattern offers the following benefits:

1. It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making subsystem easier to use.

2. It promotes weak coupling between subsystem and its clients. Weak coupling lets you vary the components of the subsystem without affecting its clients.

3. It doesn't prevent applications from using subsystem classes if they need to.
Facade

- Structure
- Participants
- Collaborations
- Implementation
- Sample Code
- Known Uses
- Related Patterns
a = 10%
b = 30%
c = 40%
Applicability

- When an abstraction has two aspects, one dependent on the other.
- When a change to one object requires changing others.
- When an object should be able to notify other objects without making assumptions about who these objects are.
Observer, structure

Subject
- attach(Observer)
- detach(Observer)
- notify()

ConcreteSubject
- subjectState
- getState()
- setState()

Observer
- update()

ConcreteObserver
- observerState
- update()
Observer, collaborations

The diagram illustrates the Observer pattern, where a concrete subject communicates changes in state to its observers. The sequence of events includes:

1. The subject sends a notification to its observers.
2. Each observer updates its state based on the notification.
3. The subject gets its state, followed by the observer getting its state.

The objects involved are:
- AConcreteSubject
- AConcreteObserver
- AnotherConcreteObserver
Observer, consequences

- Abstract coupling between Subject and Observer
- Support for broadcast communication
- Unexpected updates
**Strategy**

**Name:** Strategy  
**Also known as:** Policy

**Problem:**
- Need to use different variants of the same algorithm in a class
- Different algorithms will be appropriate at different time.
- It is hard to add new algorithms and to change existing ones.

**Example:**

```
Input (Plain Text)  
| Cryptographic Module | Output (cipher text) |
```

**Algorithms:**
- AES
- DES
- 3DES
- RC5

**Intent (from GoF):**

"Define a family of algorithms, encapsulate each one and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it."
Strategy

Structure:

- Reference to a strategy type
- Abstract
- In example: e.g. class `EncryptAlg`

Context

+contextInterface()

Strategy

-*strategy

+algorithmInterface()

ConcreteStrategyA

+algorithmInterface()

ConcreteStrategyB

+algorithmInterface()

ConcreteStrategyC

+algorithmInterface()

In example: Part of crypto module. Holds data, keys etc.

In Example: Implements e.g. algorithm AES

E.g. AlgDES

E.g. AlgRC5