Chapter 10, Object Design II: Patterns and Design Principles

Object-Oriented Software Construction

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher
(based on Bruegge & Dutoit)
Outline of the Lecture

♦ Review of design pattern concepts
  ♦ What is a design pattern?
  ♦ Modifiable designs
  ♦ Design Principles

♦ More patterns
  ♦ Observer: Provide publisher/subscribe mechanism
  ♦ Strategy: Support family of algorithms, separate of policy and mechanism
  ♦ Command: Encapsulate control flow
  ♦ Abstract Factory: Provide manufacturer independence
  ♦ Builder: Hide a complex creation process
  ♦ Proxy: Provide Location transparency
A design pattern is...

...a template solution to a recurring design problem

- Look before re-inventing the wheel just one more time
- variations of the problem and their driving forces

...reusable design knowledge

- Higher level than classes or data structures (link lists, binary trees...)
- Lower level than application frameworks
- They provide a shared vocabulary to designers

...an example of *modifiable* design
Why are modifiable designs important?

A modifiable design enables...

...an iterative and incremental development cycle

- flexibility to change (changing requirements)
- concurrent development
- risk management

...to minimize the introduction of new problems when fixing old ones

- avoid fragility
  - Even small changes can cause cascading effects
  - Code breaks in unexpected places
- locality of change

...to deliver more functionality after initial delivery
What makes a design modifiable?

♦ Low coupling and high cohesion

♦ Clear dependencies

♦ Explicit assumptions
Design Principles, Robert C. Martin, 1996

- Dependency Inversion Principle (DIP)
  - Depend upon Abstraction, Do NOT Depend upon Concretions.

- Acyclic Dependencies Principle (ADP)
  - The dependency structure for a released component must be a Directed Acyclic Graph (DAG). Dependencies must not form cycles.

- Stable Dependencies Principle (SDP)
  - The dependencies between components in a design should be in the direction of stability. A component should only depend upon components that are more stable than it is.

- Stable Abstractions Principle (SAP)
  - The abstraction of a package should be proportional to its stability! Packages that are maximally stable should be maximally abstract. Instable packages should be concrete.
More Reusable Class

Useless Class

Unmaintainable Class

Less Reusable Class

proper dependency

improper dependency

main sequence

no cycles!
DIP/SDP (Remember these colors! 😊)

- More Reusable Class
- Useless Class
- Unmaintainable Class
- Less Reusable Class

proper dependency
Towards a Pattern Taxonomy

♦ Structural Patterns
  ♦ Adapters, Bridges, Composite, Facades, and Proxies are variations on a single theme:
    ♦ They reduce the coupling between two or more classes
    ♦ They introduce an abstract class to enable future extensions
    ♦ They encapsulate complex structures

♦ Behavioral Patterns
  ♦ Concerned with communication between objects
  ♦ Characterize complex control flows that are difficult to follow at runtime
  ♦ Assignment of responsibilities between objects
    ♦ Who does what?

♦ Creational Patterns
  ♦ provide a simple abstraction for a complex instantiation process.
  ♦ make the system independent from the way its objects are created, composed and represented
On to More Patterns!

- Behavioral pattern
  - Observer
  - Strategy
  - Command

- Creational Patterns
  - Abstract Factory
  - Builder

- Structural pattern
  - Proxy
A Pattern Taxonomy

Pattern

Structural Pattern

Behavioral Pattern

Command
Observer
Strategy

Creational Pattern

Abstract Factory
Builder Pattern

Adapter
Bridge
Facade
Proxy
Review: Model View Controller Framework

- Multiple views can exist at the same time
- New views can be added without any changes to the model
- architectural pattern
Design patterns inside MVC

♦ Propagate change: **Observer Pattern**
  - Also used in client / server development for the notification of clients

♦ Nested Views: **Composite Pattern**
  - A View contains other views, but is used like a simple view
  - E.g. used for GUI widgets or parse trees in compiler implementations

♦ Reactions on events in the controller: **Strategy Pattern**
  - Controllers can be changed at runtime
  - E.g. used for different GUI controllers (view activated / deactivated)
Observer pattern

♦ “Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.”

♦ Also called “Publish and Subscribe”

♦ Forces / Applicability
  ♦ A change to one object requires changes of others
  ♦ Notification of other objects of unknown structure
  ♦ Maintaining consistency across redundant state
  ♦ Optimizing batch changes to maintain consistency
Observer pattern (continued)

Observers

Subject

Object Design 2 (Patterns).ppt

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher (based on Bruegge & Dutoit) Object-Oriented Software Construction 17
Observer pattern (cont’d)

- The **Subject** represents the actual state, the **Observers** represent different views of the state.
- **Observer** can be implemented as a Java interface.
- **Subject** is a super class (needs to store the observer set) **not** an interface.
Sequence diagram for scenario: Change filename to “foo”

- aFile
- anInfoView
- aListView

Attach()
setState(“foo”)
notify()
update()

Subject goes through all its observers and calls update() on them, asking for the new state is decoupled from the notification.
Animated Sequence diagram

- aFile
  - Attach()
  - setState(“foo”)
  - notify()
  - update()
  - getState()
  - “foo”

- anInfoView

- aListView
  - Attach()
  - setState( foo )
  - notify()
  - update()
Observer pattern implementation in Java

// import java.util;

public class Subject extends Object {
    private HashSet<Observer> observers = new HashSet();
    public void addObserver(Observer o) { observers.add(o); }
    public void deleteObserver(Observer o) {...}
    public void notifyObservers() { forall: o in: observers do: ... }
    public void notifyObservers(Object arg);
}

public abstract interface Observer {
    public abstract void update(Observable o, Object arg);
}

public class ConcreteSubject extends Subject {
    public void setState(String filename);
    public string getState();
}

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher (based on Bruegge & Dutoit) Object-Oriented Software Construction
Observer enforces Design Principles

Motivation

Temperature Sensor notifies Temperature Display.

Application
We replaced the "bad" dependency with a common "good" dependency.
Observer variations

- Push / pull model
- Specifying interest explicitly
  - Extend subject's registration interface
  - Specify interest for specific events
  - Improves update efficiency
- Change-Manager
  - Example
  - Operation involves changes to several independent objects
  - Change-Manager postpones notification after all changes have been committed (transaction)
A Pattern Taxonomy

Pattern

Structural Pattern
- Adapter
- Bridge
- Facade
- Proxy

Behavioral Pattern
- Command
- Observer
- Strategy

Creational Pattern
- Abstract Factory
- Builder Pattern
Strategy Pattern - Forces

- Many different algorithms exist for the same task
- The different algorithms will be appropriate at different times
- Examples:
  - Breaking a stream of text into lines
  - Sorting a list of customers
- We don’t want to support all the algorithms if we don’t need them
- If we need a new algorithm, we want to add it easily without disturbing the application using the algorithm
**Strategy Pattern**

Policy decides which Strategy is best given the current Context.
Applying a Strategy Pattern in a Database Application

Database

<table>
<thead>
<tr>
<th>Search()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sort()</td>
</tr>
</tbody>
</table>

Strategy

* Strategy

Sort()

BubbleSort

Sort()

QuickSort

Sort()

MergeSort

Sort()
Applicability of Strategy Pattern

♦ Many related classes differ only in their behavior
  ♦ configure a single class with one of many behaviors
♦ Different variants of an algorithm are needed that trade-off space against time
  ♦ Implement all variants as a hierarchy of algorithms
Command Pattern: Motivation

♦ You want to build a user interface
♦ You want to provide menus
♦ You want to make the user interface reusable across many applications
  ♦ You cannot hardcode the meanings of the menus for the various applications
  ♦ The applications only know what has to be done when a menu is selected.
♦ Such a menu can easily be implemented with the Command Pattern
Command pattern

- **Client** creates a **ConcreteCommand** and binds it with a **Receiver**.
- **Client** hands the **ConcreteCommand** over to the **Invoker** which stores it.
- The **Invoker** has the responsibility to do the command ("execute" or "undo").
“Encapsulate a request as an object, thereby letting you
- parameterize clients with different requests,
- queue or log requests, and
- support undoable operations.”

Uses:
- Undo queues
- Database transaction buffering
- decouple gui and backend
- Composite commands (macros)
Abstract Factory Motivation

♦ 2 Examples

♦ Consider a user interface toolkit that supports multiple looks and feel standards such as Gnome, Windows XP or Aqua OS X.
  ♦ How can you write a single user interface and make it portable across the different look and feel standards for these window managers?

♦ Consider a facility management system for an intelligent house that supports different control systems such as Siemens’ Instabus or Johnson & Control Metasys.
  ♦ How can you write a single control system that is independent from the manufacturer?
Abstract Factory

Client

AbstractFactory

CreateProductA
CreateProductB

ConcreteFactory1

CreateProductA
CreateProductB

ConcreteFactory2

CreateProductA
CreateProductB

AbstractProductA

ProductA1
ProductA2

AbstractProductB

ProductB1
ProductB2

Initiation Association:
Class ConcreteFactory2 initiates the associated classes ProductB2 and ProductA2

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher (based on Bruegge & Dutoit) Object-Oriented Software Construction 38
Applicability for Abstract Factory Pattern

♦ Independence from Initialization or Representation:
  ♦ The system should be independent of how its products are created, composed or represented

♦ Manufacturer Independence:
  ♦ A system should be configured with one family of products, where one has a choice from many different families.
  ♦ You want to provide a class library for a customer ("facility management library"), but you don’t want to reveal what particular product you are using.

♦ Constraints on related products
  ♦ A family of related products is designed to be used together and you need to enforce this constraint

♦ Cope with upcoming change:
  ♦ You use one particular product family, but you expect that the underlying technology is changing very soon, and new products will appear on the market.
Example: A Facility Management System for the Intelligent Workplace

- **Facility Mgt System**
  - IntelligentWorkplace
    - InitLightSystem
    - InitBlindSystem
    - InitACSystem
  - SiemensFactory
    - InitLightSystem
    - InitBlindSystem
    - InitACSystem
  - Metasys
    - InitLightSystem
    - InitBlindSystem
    - InitACSystem
  - LightBulb
    - InstabusLight
    - ZumbobelLight
  - Blinds
    - InstabusBlind
    - ZumtobelBlind
Abstract Factory enforces Design Principles

Abstract

Blind

Bulb

Concrete

IntelligentWorkspace

InstabusLight

InstabusBlind

Facility Mgt System

Stable

UNSTABLE

ABSTRACT

CONCRETE

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher (based on Bruegge & Dutoit)

Object-Oriented Software Construction
Abstract Factory enforces Design Principles

**Abstract**

Blind | Bulb
---|---

**Concrete**

WorkspaceFactory

IntelligentWorkspace

InstabusLight

InstabusBlind

SiemensFactory

Facility Mgt System

**Stable**

**Unstable**
A Pattern Taxonomy

Pattern

Structural Pattern

Behavioral Pattern

Command

Observer

Strategy

Creational Pattern

Abstract Factory

Builder Pattern

Adapter

Bridge

Facade

Proxy

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher (based on Bruegge & Dutoit) Object-Oriented Software Construction
Builder Pattern Motivation

♦ Conversion of documents

♦ Motivation: A reader for RTF format
  ♦ Convert RTF to many text formats (Tex, HTML, MS Word 2003, Open Office 2.0, …)
    ♦ Problem: The number of conversions is open-ended.
Builder Pattern

♦ Applicability / Forces
  ♦ Algorithm for creating objects should be independent of the parts
  ♦ Construction process must allow different representation for the constructed objects

♦ Solution
  ♦ Configure the RTF Reader with a “builder” object that specializes in conversions to any known format and can easily be extended to deal with any new format appearing on the market
Builder Pattern

For all objects in Structure {
    Builder->BuildPart()
}

ConcreteBuilderA
BuildPart()
GetResult()

ConcreteBuilderB
BuildPart()
GetResult()

Director
Construct()

Builder
BuildPart()

Representation A

Representation B

Armin B. Cremers, Tobias Rho, Holger Mügge, Daniel Speicher (based on Bruegge & Dutoit) Object-Oriented Software Construction 46
Example

```
While (t = GetNextToken()) {
    switch (t.Type) {
        CHAR: builder->ConvertCharacter(t.Char);
        FONT: builder->ConvertFont(t.Font);
        PARA: builder->ConvertParagraph(t.Para);
    }
}
```
When do you use the Builder Pattern?

- The creation of a complex product must be independent of the particular parts that make up the product.
  - In particular, the creation process should not know about the assembly process (how the parts are put together to make up the product).
Comparison:
Abstract Factory vs Builder

♦ Abstract Factory
  ♦ Focuses on product family
    ♦ The products can be simple (“light bulb”) or complex (“engine”)
  ♦ Does not hide the creation process
    ♦ The product is immediately returned

♦ Builder
  ♦ The underlying product needs to be constructed as part of the system, but the creation is very complex
  ♦ The construction of the complex product changes from time to time
  ♦ The builder patterns hides the creation process from the user:
    ♦ The product is returned after creation as a final step

♦ Abstract Factory and Builder work well together for a family of multiple complex products
Proxy Pattern: Motivation

♦ Imagine you want to set up an internet connection via a GPRS phone to access a news site. Typically news sites are packed with advertising - static pictures and flash animations which take a while to load over a slow connection and make your visit expensive if you pay your connection per KB.

♦ What can you do?
Proxy Pattern

♦ What is expensive?
  ♦ Object Creation
  ♦ Object Initialization

♦ Defer object creation and object initialization to the time you need the object

♦ Proxy pattern:
  ♦ Reduces the cost of accessing objects
  ♦ Uses another object ("the proxy") that acts as a stand-in for the real object
  ♦ The proxy creates the real object only if the user asks for it
  ♦ Maybe a local representative for an object in a different address space.
Interface inheritance is used to specify the interface shared by **Proxy** and **RealSubject**.

Delegation is used to catch and forward any accesses to the **RealSubject** (if desired).

Proxy patterns can be used for lazy evaluation and for remote invocation.

Proxy patterns can be implemented with a Java interface.
Proxy Applicability / Forces

♦ Remote Proxy
  ♦ Local representative for an object in a different address space
  ♦ Caching of information: Good if information does not change too often

♦ Virtual Proxy
  ♦ Object is too expensive to create or too expensive to download
  ♦ Proxy is a stand-in

♦ Protection Proxy
  ♦ Proxy provides access control to the real object
  ♦ Useful when different objects should have different access and viewing rights for the same document.
  ♦ Example: Grade information for a student shared by administrators, teachers and students.
Smart reference
- Replacement for bare references
- Performs additional actions when object is accessed

Examples
- Reference counting ("smart pointers")
- Loading persistent object into memory
- Synchronization
Virtual Proxy example

- **Images** are stored and loaded separately from text.
- If a **RealImage** is not loaded a **ProxyImage** displays a grey rectangle in place of the image.
- The client cannot tell that it is dealing with a **ProxyImage** instead of a **RealImage**.
- A proxy pattern can be easily combined with a **Bridge**.
Parteien lehnen Verrechnung von Krankheitstagen ab

Die Grünen reden von "Schwächselohn", die SPD von "Unsinn" – auch Union und FDP lehnen Urlaubskürzungen nach Krankheit ab. Die Forderung des Zentralverbandes des Deutschen Handwerks steht auf breite politische Ablehnung, mehr...

- Wirtschaftsexperte: Alternierende Arbeitnehmer sollen weniger verdienen
- Abgaben: Eiche: erteilt Absage an höhere Mehrwertsteuer

Angriff auf die Agrarlobby

Nach dem Großbritanniens Premier Tony Blair sich für drastische Kürzungen der Agrarhilfen ausgesprochen hat, fordern auch deutsche Politiker eine Umstrukturierung des AG-Haushalts. Derzeit gibt Europa nach 50 Milliarden Euro für Landwirtschaftssubventionen aus – doch die Agrarlobby dürfte Probleme bekommen, mehr...

- Währungsturbulenzen: EU-Streit belastet Euro
- Finanzstreit: Kommissionsvöllner fordert EU-Sondergipfel
- "EU-Krise": Schroeder gerät in die Schusslinie
- Interview mit SPD-Generalsekretär: "Europa ist sicher nicht am Ende"
Controlling Access
Hier klicken!

---

**URLAUBSKÜRZUNGEN**

**Parteien lehnen Vorruhestand von Krankheitstagen ab**

**Vote** Die Grünen reden von "Schwachsinn", die SPD von "Unsi..." - auch Union und FDP lehnen Urlaubsnachleistungen nach Krankheit ab. Die Forderung des Zentralverbandes des Deutschen Handwerks stößt auf breite politische Ablehnung. mehr...

- **Wirtschaftsexperte**: Ältere Arbeitnehmer sollen weniger verdienen
- **Abgaben**: Eichel erteilt Absage an höhere Mehrwertsteuer

---

**UMSTRITTENER EU-HAUSHALT**

**Angriff auf die Agrarlobby**

Nachdem Großbritanniens Premier Tony Blair sich für drastische Kürzungen der Agrarhilfen ausgesprochen hat, fordern auch...
Summary

♦ Structural Patterns
  ♦ Focus: How objects are composed to form larger structures
  ♦ Problems solved:
    ♦ Realize new functionality from old functionality,
    ♦ Provide flexibility and extensibility

♦ Behavioral Patterns
  ♦ Focus: Algorithms and the assignment of responsibilities to objects
  ♦ Problem solved:
    ♦ Too tight coupling to a particular algorithm

♦ Creational Patterns
  ♦ Focus: Creation of complex objects
  ♦ Problems solved:
    ♦ Hide how complex objects are created and put together
Open-Closed Principle (OCP)
(Robert C. Martin)

♦ Classes should be *open* for extension
  ♦ through inheritance, delegation, observer, …

♦ Classes should be *closed* for modification
  ♦ functionality (once set) should not be changed
    ♦ protect correct, bug free code

♦ Sounds contradictory?
  ♦ Examples: Observer, Abstract Factory, Decorator,…

♦ But: Do not apply OCP everywhere!
  ♦ Can lead to complex, hard to understand code
  ♦ New levels of abstraction adds complexity
  ♦ Wasteful and unnecessary
Conclusion

♦ Design patterns
  ♦ Provide solutions to common problems.
  ♦ Lead to extensible models and code.
  ♦ Can be used as is or as examples of interface inheritance and delegation.
  ♦ Apply the same principles to structure and to behavior.

♦ Design patterns solve all your software engineering problems 😊
Recommendations

  ♦ well written introduction
  ♦ comprehensible examples
  ♦ concentrates on ten patterns