Chapter 8, Object Design
Introduction to Design Patterns
Learning Design Patterns
Example OO application

Joe works for a company that makes a highly successful duck pond simulation game, *SimUDuck*. The game can show a large variety of duck species swimming and making quacking sounds. The initial designers of the system used standard OO techniques and created one Duck superclass from which all other duck types inherit.
Change Request Comes in

- In the last year, the company has been under increasing pressure from competitors... They need something really impressive to show at the upcoming shareholders meeting in Maui next week.

- The executives decided that flying ducks is just what the simulator needs to blow away the other duck sim competitors. And of course Joe’s manager told them it’ll be no problem for Joe to just whip something up in a week. “After all,” said Joe’s boss, “he’s an OO programmer... how hard can it be?”

I just need to add a fly() method in the Duck class and then all the ducks will inherit it. Now’s my time to really show my true OO genius.
Implementing Change Request

Joe, I'm at the shareholder's meeting. They just gave a demo and there were rubber duckies flying around the screen. Was this your idea of a joke? You might want to spend some time on Monster.com...
Joe failed to notice that not all subclasses of Duck should fly. When Joe added new behavior to the Duck superclass, he was also adding behavior that was not appropriate for some Duck subclasses. He now has flying inanimate objects in the SimUDuck program.

A localized update to the code caused a nonlocal side effect (flying rubber ducks)!
Easy Fix?

I could always just override the fly() method in rubber duck, the way I am with the quack() method...

But then what happens when we add wooden decoy ducks to the program? They aren't supposed to fly or quack...

<table>
<thead>
<tr>
<th>RubberDuck</th>
</tr>
</thead>
<tbody>
<tr>
<td>quack() { // squeak</td>
</tr>
<tr>
<td>display() { // rubber duck }</td>
</tr>
<tr>
<td>fly() {</td>
</tr>
<tr>
<td>// override to do nothing</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

Here's another class in the hierarchy; notice that like RubberDuck, it doesn't fly, but it also doesn't quack.

<table>
<thead>
<tr>
<th>DecoyDuck</th>
</tr>
</thead>
<tbody>
<tr>
<td>quack() { // override to do nothing</td>
</tr>
<tr>
<td>display() { // decoy duck }</td>
</tr>
<tr>
<td>fly() {</td>
</tr>
<tr>
<td>// override to do nothing</td>
</tr>
</tbody>
</table>
Interfaces?

I could take the fly() out of the Duck superclass, and make a Flyable() interface with a fly() method. That way, only the ducks that are supposed to fly will implement that interface and have a fly() method... and I might as well make a Quackable, too, since not all ducks can quack.

What do you think about this design?
Interfaces

We know that not all of the subclasses should have flying or quacking behavior, so inheritance isn’t the right answer. But while having the subclasses implement Flyable and/or Quackable solves part of the problem (no inappropriately flying rubber ducks), it completely destroys code reuse for those behaviors, so it just creates a different maintenance nightmare. And of course there might be more than one kind of flying behavior even among the ducks that do fly.

WHAT WOULD YOU DO IF YOU WERE JOE?
We have to deal with CHANGE

The one constant in software development

Okay, what’s the one thing you can always count on in software development?

No matter where you work, what you’re building, or what language you are programming in, what’s the one true constant that will be with you always?

CHANGE

(use a mirror to see the answer)

No matter how well you design an application, over time an application must grow and change or it will die.
DESIGN PRINCIPLE

Identify the aspects of your application that vary and separate them from what stays the same.

The first of many design principles. We’ll spend more time on these throughout the book.
Separating behaviors

The Duck class is still the superclass of all ducks, but we are pulling out the fly and quack behaviors and putting them into another class structure.

Now flying and quacking each get their own set of classes.

Various behavior implementations are going to live here.

Pull out what varies

Duck class

Flying Behaviors

Quacking Behaviors

Duck Behaviors
Separating Behaviours

**DESIGN PRINCIPLE**

Program to an interface, not an implementation.

```java
<<interface>>

FlyBehavior

fly()

FlyWithWings

fly()
// implements duck flying

FlyNoWay

fly()
// do nothing - can't fly!
```

I don't see why you have to use an interface for FlyBehavior. You can do the same thing with an abstract superclass. Isn't the whole point to use polymorphism?
Implementing Duck Behavior

FlyBehavior is an interface that all flying classes implement. All new flying classes just need to implement the fly() method.

<<interface>>
FlyBehavior

fly()

FlyWithWings
fly()
// implements duck flying

FlyNoWay
fly()
// do nothing - can't fly!

QuackBehavior

<<interface>>
QuackBehavior

quack()

Quack
quack()
// implements duck quacking

Squack
quack()
// rubber ducky squeak

MuteQuack
quack()
// do nothing - can't quack!

Same thing here for the quack behavior; we have an interface that just includes a quack() method that needs to be implemented.

Here's the implementation for all ducks that can't fly.

Quacks that really quack.

Quacks that squeak.

Quacks that make no sound at all.
Combining behaviors with the duck

The behavior variables are declared as the behavior INTERFACE type.

These methods replace fly() and quack().

Instance variables hold a reference to a specific behavior at runtime.

```
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior

performQuack()
swim()
display()
performFly()
// OTHER duck-like methods...
```
The Big Picture

Client makes use of an encapsulated family of algorithms for both flying and quacking.

Encapsulated fly behavior

Think of each set of behaviors as a family of algorithms.

Encapsulated quack behavior

These behaviors "algorithms" are interchangeable.
Implementation

NOTE

With this design, other types of objects can reuse our fly and quack behaviors because these behaviors are no longer hidden away in our Duck classes!

And we can add new behaviors without modifying any of our existing behavior classes or touching any of the Duck classes that use flying behaviors.

So we get the benefit of REUSE without all the baggage that comes along with inheritance.

```java
public class Duck {
    QuackBehavior quackBehavior;
    // more

    public void performQuack() {
        quackBehavior.quack();
    }
}
```

Each Duck has a reference to something that implements the QuackBehavior interface.

Rather than handling the quack behavior itself, the Duck object delegates that behavior to the object referenced by quackBehavior.
Duck Class

```java
public abstract class Duck {
    FlyBehavior flyBehavior;
    QuackBehavior quackBehavior;

    public Duck() {
    }

    public abstract void display();

    public void performFly() {
        flyBehavior.fly();
    }

    public void performQuack() {
        quackBehavior.quack();
    }

    public void swim() {
        System.out.println("All ducks float, even decoys!");
    }

    public interface FlyBehavior {
        public void fly();
    }

    public class FlyWithWings implements FlyBehavior {
        public void fly() {
            System.out.println("I'm flying!!!");
        }
    }

    public class FlyNoWay implements FlyBehavior {
        public void fly() {
            System.out.println("I can't fly");
        }
    }
}
```
Additional Design Heuristics

• Never use implementation inheritance, always use interface inheritance
• A subclass should never hide operations implemented in a superclass
• If you are tempted to use implementation inheritance, use delegation instead
Learning Design Patterns
Strategy Pattern

- **The Strategy Pattern** defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Strategy Pattern

- Different algorithms exists for a specific task
  - We can switch between the algorithms at run time
- Examples of tasks:
  - Different collision strategies for objects in video games
  - Parsing a set of tokens into an abstract syntax tree (Bottom up, top down)
  - Sorting a list of customers (Bubble sort, mergesort, quicksort)
- Different algorithms will be appropriate at different times
  - First build, testing the system, delivering the final product
- If we need a new algorithm, we can add it without disturbing the application or the other algorithms.
Strategy Pattern

Policy decides which ConcreteStrategy is best in the current Context.
Using a Strategy Pattern to Decide between Algorithms at Runtime

Client

Policy

TimelIsImportant
SpaceIsImportant

Database

SelectSortAlgorithm()
Sort()

SortInterface

* Sort()

Sort()

* Sort()

BubbleSort
Sort()

QuickSort
Sort()

MergeSort
Sort()
Supporting Multiple implementations of a Network Interface

Context = {Mobile, Home, Office}

Application

LocationManager

NetworkConnection

send()
receive()
setNetworkInterface()

NetworkInterface

open()
close()
send()
receive()

Ethernet

open()
close()
send()
receive()

WaveLAN

open()
close()
send()
receive()

UMTS

open()
close()
send()
receive()
A Game: Get-15

• Start with the nine numbers 1,2,3,4, 5, 6, 7, 8 and 9.
• You and your opponent take alternate turns, each taking a number.
• Each number can be taken only once: If you opponent has selected a number, you cannot also take it.
• The first person to have any three numbers that total 15 wins the game.

• Example:
  You:

  [1]  [5]  [3]  [8]

  Opponent:


  Opponent Wins!
Characteristics of Get-15

• Hard to play,
• The game is especially hard, if you are not allowed to write anything done.

• Why?
  • All the numbers need to be scanned to see if you have won/lost
  • It is hard to see what the opponent will take if you take a certain number
  • The choice of the number depends on all the previous numbers

  • Not easy to devise an simple strategy
Another Game: Tic-Tac-Toe

Source: http://boulter.com/ttt/index.cgi
A Draw Sitation

YOU ARE O
Strategy for determining a winning move
Winning Situations for Tic-Tac-Toe

Winning Patterns
**Tic-Tac-Toe is “Easy”**

Why? Reduction of complexity through patterns and symmetries.

**Patterns:** Knowing the following three patterns, the player can anticipate the opponents move.

\[
\begin{array}{c}
\text{H} \\
\text{F} \\
\end{array}
\]

**Symmetries:**
The player needs to remember only these three patterns to deal with 8 different game situations.

The player needs to memorize only 3 opening moves and their responses.
Get-15 and Tic-Tac-Toe are identical problems

• Any three numbers that solve the 15 problem also solve tic-tac-toe.
• Any tic-tac-toe solution is also a solution the 15 problem
• To see the relationship between the two games, we simply arrange the 9 digits into the following pattern

```
8 1 6
3 5 7
4 9 2
```
What is this?


This is a fianchetto!
In chess, the fianchetto (Italian: [fjaŋˈkɛtto] "little flank") is a pattern of development wherein a bishop is developed to the second rank of the adjacent knight fil
The fianchetto is a staple of many "hypermodern" openings, whose philosophy is to delay direct occupation of the center with the plan of undermining and destroying the opponent's central outpost.
The fianchetto is one of the basic building-blocks of chess thinking.
The diagram is from Reti-Lasker, New York 1924. We can see that Reti has allowed Lasker to occupy the centre but Rtei has fianchettoed both Bishops to hit back at this, and has even backed up his Bb2 with a Queen on a1!
Observations

• Many *problems recur*.
• Many problems have the *same solution structure*.
• Exact solution is dependent on the *context*.
• A more experienced person can solve new problems faster and better.
Problem Solving - Expert

Knows many problems.
Knows solutions to these problems.
Recalls generic solution when encountering new problems.
Knows various different contexts.
Knows how to apply this knowledge.

In short: Expert knows many patterns...
Making Patterns – Reusable

Engineers should aim to capture this valuable pattern knowledge and make this reusable in a catalog so that other engineers/designers can (re)use these patterns to build systems faster and with quality.
Discovering Patterns

Problems

Problem 1
Problem 2
Problem 3

Solution (Structure)

Context

Problem

Pattern
Describing Patterns

Name
  □ meaningful name
Problem
  □ statement of the intent/goal
Context
  □ preconditions and the pattern’s applicability
Forces
  □ description of relevant forces and constraints
Solution
  □ a structure
Example
  □ sample application of the pattern
Consequences
  □ state of the system after applying the pattern
Rationale
Related Patterns
  □ static and dynamic relations to other patterns

This is the pattern
Patterns are not designs

• Pattern is a template/blueprint
• Must be instantiated
  • make design decisions
  • evaluate tradeoffs
• combined with other patterns
• Patterns are reusable abstractions providing solutions for recurring problems
• Patterns are applied in mature engineering
• Patterns can be applied at various abstractions levels
  • architectural
  • design
  • programming
Electrical Engineering Patterns

Problem

pattern catalog (example)

Solution (Template)

push pull amplifier
Mechanical Engineering Patterns
Chemical Engineering Patterns

Problem

pattern catalog (example)

Solution (Template)
Software Engineering Patterns

Problem

pattern catalogs (examples)

Solution (Template)
Categorization of Software Patterns

Architectural Patterns (Styles)
Design Patterns
- provides schemes for refining the architecture.
Programming Patterns (Idioms)
- provides schemes for mapping the design to a specific programming language

(Other: Organizational, Analysis, UI Design etc.)
Software Architecture Patterns

Ex: Layers Pattern

Problem: A large system, which is characterized with a mix of low and high level issues, where high-level operations rely on low-level issues:

Solution: Structure your system into an appropriate number of layers and place them on top of each other;

express a fundamental structural organization schemes for software systems. provide a set of predefined subsystems specify their responsibilities and include rules and guidelines for organizing the relationships between them
Software Design Patterns

Design Patterns: Elements of Reusable Object-Oriented Software, Gamma et al., 1995

A design pattern provides a scheme for refining the architectural entities of a software system. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.

23 object-oriented design patterns
Design pattern community
  - http://hillside.net/patterns/
PLOP conferences/workshops
THE 23 GANG OF FOUR DESIGN PATTERNS

- Abstract Factory
- Adapter
- Bridge
- Builder
- Chain of Responsibility
- Command
- Composite
- Decorator
- Facade
- Factory Method
- Flyweight
- Interpreter
- Iterator
- Mediator
- Memento
- Proxy
- Observer
- Singleton
- State
- Strategy
- Template Method
- Visitor
- Prototype
Adapter Pattern

The British wall outlet exposes one interface for getting power.

The US laptop expects another interface.

The adapter converts one interface into another.
Can you think of a solution that doesn’t require YOU to write ANY additional code to integrate the new vendor classes? How about making the vendor supply the adapter class?
The Adapter Pattern Explained

1. The client makes a request to the adapter by calling a method on it using the target interface.
2. The adapter translates the request into one or more calls on the adaptee using the adaptee interface.
3. The client receives the results of the call and never knows there is an adapter doing the translation.
The adapter pattern uses inheritance as well as delegation:
- Interface inheritance is used to specify the interface of the Adapter class.
- Delegation is used to bind the Adapter and the Adaptee
Real life adapter in Java

Your new code still gets to use Iterators, even if there’s really an Enumeration underneath.

EnumerationsIterator is the adapter.

We’re making the Enumerations in your old code look like Iterators for your new code.

A class implementing the Enumeration interface is the adaptee.
Another Example

That’s a lot of classes, a lot of interactions, and a big set of interfaces to learn and use.
Watch a Movie

Pick out a DVD, relax, and get ready for movie magic. Oh, there’s just one thing—to watch the movie, you need to perform a few tasks:

1. Turn on the popcorn popper  
2. Start the popper popping  
3. Dim the lights  
4. Put the screen down  
5. Turn the projector on  
6. Set the projector input to DVD  
7. Put the projector on wide-screen mode  
8. Turn the sound amplifier on  
9. Set the amplifier to DVD input  
10. Set the amplifier to surround sound  
11. Set the amplifier volume to medium (5)  
12. Turn the DVD player on  
13. Start the DVD player playing
Watch a Movie

```java
popper.on();
popper.pop();

lights.dim(10);
screen.down();

projector.on();
projector.setInput(dvd);
projector.wideScreenMode();

amp.on();
amp.setDvd(dvd);
amp.setSurroundSound();
amp.setVolume(5);

dvd.on();
dvd.play(movie);
```

- Turn on the popcorn popper and start popping...
- Dim the lights to 10%...
- Put the screen down...
- Turn on the projector and put it in wide screen mode for the movie...
- Turn on the amp, set it to DVD, put it in surround sound mode and set the volume to 5...
- Turn on the DVD player... and FINALLY, play the movie!
But there’s more...

When the movie is over, how do you turn everything off? Wouldn’t you have to do all of this over again, in reverse?
Wouldn’t it be as complex to listen to a CD or the radio?
If you decide to upgrade your system, you’re probably going to have to learn a slightly different procedure.
So what to do? The complexity of using your home theater is becoming apparent!
Façade

1. Okay, time to create a Façade for the home theater system. To do this we create a new class HomeTheaterFacade, which exposes a few simple methods such as watchMovie().

2. The Facade class treats the home theater components as a subsystem, and calls on the subsystem to implement its watchMovie() method.

3. Your client code now calls methods on the home theater Facade, not on the subsystem. So now to watch a movie we just call one method, watchMovie(), and it communicates with the lights, DVD player, projector, amplifier, screen, and popcorn maker for us.

4. The Facade still leaves the subsystem accessible to be used directly. If you need the advanced functionality of the subsystem classes, they are available for your use.

A client of the subsystem facade.
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                Tuner tuner,
                DvdPlayer dvd,
                CdPlayer cd,
                Projector projector,
                Screen screen,
                TheaterLights lights,
                PopcornPopper popper) {
        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }

    public void watchMovie(String movie) {
        System.out.println("Get ready to watch a movie...");
        popper.on();
        popper.pop();
        lights.dim(10);
        screen.down();
        projector.on();
        projector.wideScreenWidthMode();
        amp.on();
        amp.setDvd(dvd);
        amp.setSurroundSound();
        amp.setVolume(5);
        dvd.on();
        dvd.play(movie);
    }

    public void endMovie() {
        System.out.println("Shutting movie theater down...");
        popper.off();
        lights.on();
        screen.up();
        projector.off();
        amp.off();
        dvd.stop();
        dvd.eject();
        dvd.off();
    }

    // other methods here

    // We're just about to fill these in...
}
Watch a Movie (The easy way)

```java
public class HomeTheaterTestDrive {
    public static void main(String[] args) {
        // instantiate components here

        HomeTheaterFacade homeTheater =
            new HomeTheaterFacade(amp, tuner, dvd, cd,
                                  projector, screen, lights, popper);

        homeTheater.watchMovie("Raiders of the Lost Ark");
        homeTheater.endMovie();
    }
}
```

---

Here we're creating the components right in the test drive. Normally, the client is given a facade; it doesn't have to construct one itself.

First you instantiate the Facade with all the components in the subsystem.

Use the simplified interface to first start the movie up, and then shut it down.
Facade Pattern

The Facade Pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Façade and Principle of Least Knowledge

The HomeTheaterFacade manages all those subsystem components for the client. It keeps the client simple and flexible.

We can upgrade the home theater components without affecting the client.

We try to keep subsystems adhering to the Principle of Least Knowledge as well. If this gets too complex and too many friends are intermingling, we can introduce additional facades to form layers of subsystems.

This client only has one friend: the HomeTheaterFacade. In OO programming, having only one friend is a GOOD thing!
Facade Pattern

- Provides a unified interface to a set of objects in a subsystem.
- A facade defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)

![Facade Pattern Diagram]
When should you use these Design Patterns?

• A façade should be offered by all subsystems in a software system who a services
  • The façade delegates requests to the appropriate components within the subsystem. The façade usually does not have to be changed, when the components are changed

• The adapter design pattern should be used to interface to existing components
  • Example: A smart card software system should use an adapter for a smart card reader from a specific manufacturer
Realizing an Opaque Architecture with a Facade

• The subsystem decides exactly how it is accessed.

• No need to worry about misuse by callers.

• If a façade is used the subsystem can be used in an early integration test.
  • We need to write only a driver.
Adapter / Façade Summary

- When you need to use an existing class and its interface is not the one you need, use an adapter.
- When you need to simplify and unify a large interface or complex set of interfaces, use a facade.
- An adapter changes an interface into one a client expects.
- A facade decouples a client from a complex subsystem.
- Implementing an adapter may require little work or a great deal of work depending on the size and complexity of the target interface.
- Implementing a facade requires that we compose the facade with its subsystem and use delegation to perform the work of the facade.
- You can implement more than one facade for a subsystem.
What is common between these definitions?

• Definition Software System
  • A software system consists of subsystems which are either other subsystems or collection of classes

• Definition Software Lifecycle:
  • The software lifecycle consists of a set of development activities which are either other activities or collection of tasks
Introducing the Composite Pattern

- Models tree structures that represent part-whole hierarchies with arbitrary depth and width.
- The Composite Pattern lets client treat individual objects and compositions of these objects uniformly.
Modeling a Software System with a Composite Pattern

![Composite Pattern Diagram]

- User
- Software System
- Class
- Subsystem
- *
- children
Graphic Applications also use Composite Patterns

- The *Graphic* Class represents both primitives (Line, Circle) and their containers (Picture)
Reducing the Complexity of Models

• To communicate a complex model we use navigation and reduction of complexity
  • We do not simply use a picture from the CASE tool and dump it in front of the user
  • The key is to navigate through the model so the user can follow it

• We start with a very simple model
  • Start with the key abstractions
  • Then decorate the model with additional classes

• To reduce the complexity of the model further, we
  • Look for inheritance (taxonomies)
    • If the model is still too complex, we show subclasses on a separate slide
  • Then we identify or introduce patterns in the model
    • We make sure to use the name of the patterns.
Example: A Complex Model

Basic Abstractions

Composite Patterns

Taxonomies

* Resource
  * Participant
    * Fund
      * Equipment
      * Facility
      * Organization
  * Schedule
  * Activity
    * Task
      * Work Package
        * Work Breakdown Structure
          * Resource
          * Work
            * Outcome
              * Set of Work Products
                * Work Product
                  * Project
                    * Deliverable
                      * Internal Work Product
                        * Project Function
                          * Organizational Unit
                            * Participant
                              * Staff
                                * Department
                                  * Team
Summary

• Composite, Adapter, Bridge, Façade, Proxy (Structural Patterns)
  • Focus: Composing objects to form larger structures
    • Realize new functionality from old functionality,
      • Provide flexibility and extensibility

• Command, Observer, Strategy, Template (Behavioral Patterns)
  • Focus: Algorithms and assignment of responsibilities to objects
    • Avoid tight coupling to a particular solution

• Abstract Factory, Builder (Creational Patterns)
  • Focus: Creation of complex objects
    • Hide how complex objects are created and put together
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 a lot of your software development problems
  - Pattern-oriented development
Chapter 8, Object Design: Design Patterns II
Recall: Why reusable Designs?

A design...
...enables flexibility to change (reusability)
...minimizes the introduction of new problems when fixing old ones (maintainability)
...allows the delivery of more functionality after an initial delivery (extensibility).
Definitions

- **Extensibility (Expandibility)**
  - A system is extensible, if new functional requirements can easily be added to the existing system

- **Customizability**
  - A system is customizable, if new nonfunctional requirements can be addressed in the existing system

- **Scalability**
  - A system is scalable, if existing components can easily be multiplied in the system

- **Reusability**
  - A system is reusable, if it can be used by another system without requiring major changes in the existing system model (design reuse) or code base (code reuse).
Command Pattern: Motivation

• You want to build a user interface
• You want to provide menus
• You want to make the menus reusable across many applications
  • The applications only know what has to be done when a command from the menu is selected
  • You don’t want to hardcode the menu commands for the various applications
• Such a user interface can easily be implemented with the Command Pattern.
**Command pattern**

- Client (in this case a user interface builder) creates a ConcreteCommand and binds it to an action operation in Receiver.
- Client hands the ConcreteCommand over to the Invoker which stores it (for example in a menu).
- The Invoker has the responsibility to execute or undo a command (based on a string entered by the user).
Comments to the Command Pattern

• The Command abstract class declares the interface supported by all ConcreteCommands.

• The client is a class in a user interface builder or in a class executing during startup of the application to build the user interface.

• The client creates concreteCommands and binds them to specific Receivers, this can be strings like “commit”, “execute”, “undo”.
  • So all user-visible commands are sub classes of the Command abstract class.

• The invoker - the class in the application program offering the menu of commands or buttons - invokes the concreteCommand based on the string entered and the binding between action and ConcreteCommand.
Decouples boundary objects from control objects

• The command pattern can be nicely used to decouple boundary objects from control objects:
  • Boundary objects such as menu items and buttons, send messages to the command objects (I.e. the control objects)
  • Only the command objects modify entity objects
• When the user interface is changed (for example, a menu bar is replaced by a tool bar), only the boundary objects are modified.
Command Pattern  Applicability

- Parameterize clients with different requests
- Queue or log requests
- Support undoable operations

Uses:
- Undo queues
- Database transaction buffering
Applying the Command Pattern to Command Sets

GameBoard «binds»

Match
  play()
  replay()

Move
  execute()

TicTacToeMove
  execute()

ChessMove
  execute()
Applying the Command design pattern to Replay Matches in ARENA

- **ReplayedMatch**
  - nextMove()
  - previousMove()

- **GameBoard**

- **Match**
  - play()
  - replay()

- **Move**
  - execute()

- **TicTacToeMove**

- **ChessMove**

- **<binds>**
Abstract Factory Pattern Motivation

• Consider a user interface toolkit that supports multiple looks and feel standards for different operating systems:
  • 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:
  • How can you write a single control system that is independent from the manufacturer?
Abstract Factory

Initiation Association:
Class **ConcreteFactory2** initiates the associated classes **ProductB2** and **ProductA2**
Applicability for Abstract Factory Pattern

- Independence from Initialization or Representation
- Manufacturer Independence
- Constraints on related products
- Cope with upcoming change
Example: A Facility Management System for a House

IntelligentHouse → HouseFactory

- createBulb()
- createBlind()

EIBFactory

- createBulb()
- createBlind()

LuxmateFactory

- createBulb()
- createBlind()

LightBulb → EIBBulb, LuxmateBulb

Blind → EIBBlind, LuxmateBlind
Applying the Abstract Factory Pattern to Games

- Tournament
- Game
  - createMatch()
  - createStatistics()
- TicTacToe
  - createMatch()
  - createStats()
- Chess
  - createMatch()
  - createStats()
- Match
- Statistics
- TTTMatch
- ChessMatch
- TTTStats
- ChessStats
Builder Pattern Motivation

• The construction of a complex object is common across several representations

• Example
  • Converting a document to a number of different formats
    • the steps for writing out a document are the same
    • the specifics of each step depend on the format

• Approach
  • The construction algorithm is specified by a single class (the “director”)
  • The abstract steps of the algorithm (one for each part) are specified by an interface (the “builder”)
  • Each representation provides a concrete implementation of the interface (the “concrete builders”)

Builder Pattern

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

ConcreteBuilderA
  BuildPart()
  GetResult()

ConcreteBuilderB
  BuildPart()
  GetResult()

Representation A

Representation B
Applicability of Builder Pattern

- The creation of a complex product must be independent of the particular parts that make up the product.
- The creation process must allow different representations for the object that is constructed.
Example: Converting an RTF Document into different representations

```java
while (t = GetNextToken()) {
    switch t.Type {
        CHAR: Builder->ConvertCharacter(t)
        FONT: Builder->ConvertFontChange(t)
        PARA: Builder->ConvertParagraph(t) }
}
```

Builder
- ConvertCharacter()
- ConvertFontChange()
- ConvertParagraph()

RTFReader
- Parse()

AsciiConverter
- ConvertCharacter()
- ConvertFontChange()
- ConvertParagraph()
- GetASCIIText()

AsciiText

TexConverter
- ConvertCharacter()
- ConvertFontChange()
- ConvertParagraph()
- GetTeXText()

TeXText

HTMLConverter
- ConvertCharacter()
- ConvertFontChange()
- ConvertParagraph()
- GetHTMLText()

HTMLText
Comparison: Abstract Factory vs Builder

- Abstract Factory
  - Focuses on product family
  - Does not hide the creation process
- 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
  - Hides the creation process from the user
- Abstract Factory and Builder work well together for a family of multiple complex products
Clues in Nonfunctional Requirements for the Use of Design Patterns

- **Text:** “manufacturer independent”, “device independent”, “must support a family of products”
  => Abstract Factory Pattern

- **Text:** “must interface with an existing object”
  => Adapter Pattern

- **Text:** “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”
  => Bridge Pattern

- **Text:** “must interface to existing set of objects”
  => Façade Pattern
Clues in Nonfunctional Requirements for use of Design Patterns (2)

- **Text:** “complex structure”, “must have variable depth and width”
  => Composite Pattern
- **Text:** “must be location transparent”
  => Proxy Pattern
- **Text:** “must be extensible”, “must be scalable”
  => Observer Pattern
- **Text:** “must provide a policy independent from the mechanism”
  => Strategy Pattern
Summary

• Composite, Adapter, Bridge, Façade, Proxy (Structural Patterns)
  • Focus: Composing objects to form larger structures
    • Realize new functionality from old functionality,
    • Provide flexibility and extensibility

• Command, Observer, Strategy, Template (Behavioral Patterns)
  • Focus: Algorithms and assignment of responsibilities to objects
    • Avoid tight coupling to a particular solution

• Abstract Factory, Builder (Creational Patterns)
  • Focus: Creation of complex objects
    • Hide how complex objects are created and put together
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 a lot of your software development problems
  - Pattern-oriented development