Introduction to Object Oriented Programming

Roy Schwartz, The Hebrew University (67125)

Lecture 5:

Intro to Design Patterns

Last Week

- Abstract Classes
- Interfaces

Lecture 5a: Overview

- Reuse Mechanisms
- Casting   עד כאן--->
- Intro to Design Patterns
- Façade Design Pattern

Reuse Mechanisms

- What is the right way to build reusable software?
- **Inheritance** provides a built-in mechanism for sharing code
  - Extending a class gives us access to all its **public** and **protected** data members and methods
  - This is considered by many one of the major reasons for using a class hierarchy
- However, there is an alternative mechanism to code reuse
  - **Object composition**

Reuse Mechanisms

Inheritance

- Define an implementation of one class in terms of another's
- Called **“white-box”** reuse since the internals of the parent are visible to its subclasses (i.e., **protected** elements)

Reuse Mechanisms

Inheritance example

```
public class B {
        protected void foo() { … }
}

public class A extends B {
      …
}
```

- Now, A gets the *foo*() method for free and can use it

Reuse Mechanisms

Composition

- An alternative to class inheritance
- New functionality is obtained by assembling (or composing) objects to get more complex functionality
- Requires the objects being composed to have well defined APIs
- Called **“black-box”** reuse because no internal details of objects are visible
  - Objects appear only as “black boxes”

Reuse Mechanisms

Composition example

```java
public class B {
        public void foo() { … }
}
```

```java
public class A {
        private B b;
        public A(B b) {
                this.b = b;
        }
        public anotherFoo(…) {
                …
                // A uses the foo() code by calling b.foo()
                this.b.foo();
                …
        }
}
```

Inheritance

Pros

- Straightforward to use (supported by the programming language)
- Enables **polymorphism**
- Defined statically at compile time
- Easy to modify the implementation being reused by overriding

Inheritance

Cons

- An implementation cannot be changed at runtime
- Breaks encapsulation
  - A subclass is exposed to details of the parent's implementation (**protected** fields/methods)
- Subclass implementation is bound to parent implementation
  - Any change in the parent forces the subclass to change
- A class can only extend a single parent-class

Object Composition

Pros

- Defined dynamically at runtime
  - The composed object can be replaced at runtime by another as long as it has the **same type**
- Encapsulation is not broken
- Substantially fewer implementation dependencies
- Helps keep each class encapsulated and focused on one task
- A class may compose as many objects as it wants

Object Composition

Cons

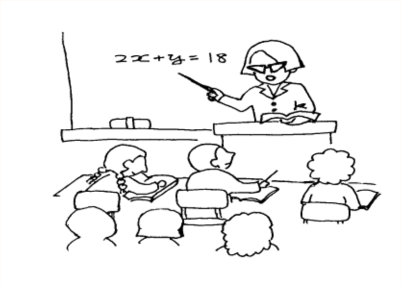
- A composition based design has more objects (if fewer classes)
- No support for **polymorphism**
- The system's behavior will depend on their interrelationships instead of being defined in one class
  - The structure of the program is more complex. A greater chance of having logical bugs

Reuse Mechanisms

Inheritance vs. Composition

- Use *inheritance* when:
  - A is **inherently** a B (a dog **is an** animal)
- Use *object composition* when:
  - You mainly want to **reuse code**
  - When it makes more sense that A extends another class C
- If you only need **polymorphism** (but an A **is not** a B), consider using *interfaces*
  - *Inheritance* and object *composition* **work together**

Interfaces Revised



Lecture 5b: Overview

- Reuse Mechanisms
- Casting
- Intro to Design Patterns
- Façade Design Pattern

Casting

- Casting is the operation in the heart of polymorphism: referring to a reference of one type with a different reference type
  - Animal a = **new** Cow();
  - This type of casting is called **up-casting**
- Up-casting is the setup when the reference *type* (i.e., the **left-hand** side) is a super class (or an interface) of the concrete object *type* (**right-hand** side)

Implicit vs. Explicit Casting

- Up-casting can be implicit
  - The compiler decided which type is the object
  - Animal myAnimal = **new** Dog();
- But it can also be explicit
  - Telling the compiler the exact casting type
  - Animal myAnimal = **(Animal)** **new** Dog();
- There is hardly any reason to use **explicit** up-casting

Down Casting

- A more complicated type of casting is **down-casting**
- Down-casting is the operation of assigning a reference with a super-class (or an interface) of the reference type
- Down casting is always **explicit**
  - Animal animal = …;
  - Cow c = **(Cow)** animal;
  - Cow c = animal; // Implicit down casting is illegal – compilation error.

Down-casting

Down Casting (2)

- Down-casting can sometimes succeed
  - if the right-hand side's **real** type is actually a sub-class (or implementing class) of the left-hand side's type
  - Animal animal = **new** Cow()
  - Cow c = **(Cow)** animal;
- But it can also fail
  - Animal animal = **new** Dog()
  - Cow c = **(Cow)** animal;
  - Cow c2 = **(Cow)** **new** Integer(5);

animal is actually a Dog. It cannot be

Cow is not a sub-class of **Integer**. This operation can never succeed. **Compilation error**

Down Casting (3)

- Down casting can only succeed after up-casting has been applied
  - Parent p = **new** Child(); // up-casting
  - Child c = (Child)p; // down-casting
- It can still fail though
  - Parent p = **new** AnotherChild(); // up-casting
  - Child c = (Child)p; // down-casting failed (runtime error)

We don’t Like Down Casting

- An reference of type C can potentially be cast to any class that **extends** / **implements** C
  - Such code will always compile
- However, casting can fail at **run-time**
- Run-time errors are **very expensive**
  - Time, reputation, money, …
- There is almost always a better alternative to down casting

We don’t Like Down Casting (2)

- Remember flexibility?
  - makeAnimalsSpeak(…) will continue to work even after writing a new Goat class
- Using down casting, this is no longer correct
  - We are hard-coding specific class names, thus making our code **specific to one class** and not **flexible**

```
public void makeAnimalSpeak(Animal animal) {
        animal.speak();
}
```

instanceof

- **instanceof** is a java operator that allows us to check whether an object is an instance of a given class
  - Animal animal = **new** Cow()
  - **if** (animal **instanceof** Cow) {

Returns **true**

- Supposedly, **instanceof** could be used as a safety measure against the runtime errors previously presented

We also don’t Like **instanceof**

- Using **instanceof** is generally considered **bad practice**
  - Although there are exceptions
- Using **instanceof** is still bug prone
- Also, **instanceof** code is inflexible

Bad instanceof Example

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
      if (animal instanceof Fish) {
            ((Fish)animal).swim();
      } else if (animal instanceof Horse) {
            ((Horse)animal).ride();
      }
      ...
    }
}
```

- What happens if we want to support a new animal (snake)?
  - Code has to change
  - Bugs may arise

instanceof Alternative

- Use a common API
  - Animal.move()

```
class AnimalMover{
    public void moveAnimal(Animal animal) {
            animal.move();
    }
}
```

- *Simpler, shorter, safer, easier to extend*

Casting Examples

a	c	d
Dog	Cow?	Dog

```
Animal a; Cow c; Dog d;
d = new Dog();          // OK
a = new Cow(5);         // OK (implicit up-casting)
a.speak();              // Returns "moo"
a = d;                  // OK (implicit up-casting)
a.speak();              // Returns "woff"
d = (Dog) a;            // OK  ([explicit] down-casting)
d = new Cow(3);         // Compile-time error (Cow is not a subclass of Dog)
d = a;                  // Compile-time error (implicit down-casting)
c = (Cow) a;            // Run-time error (incompatible down casting)
if (a instanceof Cow) {
      c = (Cow) a;       // OK (though not recommended)
}
```

Lecture 5c: Overview

- Reuse Mechanisms
- Casting <---עד כאן
- Intro to Design Patterns
- Façade Design Pattern

Design Patterns

"Each **pattern** describes a **problem** which occurs **over and over again** in our environment, and then describes the core of the solution to that problem, in such a way that **you can use this solution a million times** over and over, without ever doing it the same way twice" (Christopher Alexander, GoF, page 2)

- Alexander was an architect who studied ways to improve the process of designing buildings and urban areas

Design Patterns Properties

- Describes a proven approach to dealing with a common situation in programming / design
- Suggests what to do to obtain an elegant, modifiable, extensible, flexible & reusable solution
- Shows, at design time, how to avoid problems that may occur much later
- Is independent of specific contexts or languages

Design Pattern Motivation

Example

- Reminder: two main approaches for code reuse: **inheritance** and **composition**
- Problem: say we want to build a class B that has the exact same API as another class A
  - Use composition, but in an inheritance-like fashion
- Solution: delegation design pattern!

Delegation vs. Inheritance

```
public class A {
        public void foo() { … }
}
```

```
// Inheritance
public class C extends A {…}
```

C.foo() now calls A.foo()

requests to a

```
// Delegation
public class B {
        private A a;

        public B(A a) {
                this.a = a;
        }

        public void foo() {
                a.foo();
        }
}
```

B.foo() now also calls A.foo()!

Delegation

- Main advantage – easy to compose behaviors at runtime and to change the way they are composed
  - I.e., replace the A class object with a *D* ***extends*** *A* object
- Also, now the B class can extend another class
- The *delegation design pattern* (much like all design patterns) is **general** and applicable to **various settings and environments**

Essential Elements of a Design Pattern

1. **Design Pattern Name**
   – Having a concise, meaningful name for a pattern improves communication among developers
2. **Problem**
   – What is the problem and context where we would use this design pattern?
   – What are the conditions that must be met before this pattern should be used?

Essential Elements of a Design Pattern

3. **Solution**
   – A description of the elements that make up the design pattern
   – Emphasizes their relationships, responsibilities and collaborations
   – Not a concrete design or implementation; rather an abstract description

4. **Consequences**
   – The pros and cons of using the design pattern
   – Includes impacts on reusability, portability, extensibility

Design Patterns Types

- **Creational** patterns
  - Deal with creating objects (**instantiation**)
  - For example: **Factory, Singleton**
- **Structural** patterns
  - Deal with the objects' structure (**composition**)
  - For example: **Delegation, Façade, Decorator**
- **Behavioral** patterns
  - Handle the objects' behavior (**communication between objects**)
  - For example: **Iterator, Strategy**

Lecture 5d: Overview

- Reuse Mechanisms
- Casting
- Intro to Design Patterns
- Façade Design Pattern

Façade

- A **structural** design pattern
  - Deals with the objects’ structure (**composition**)
- The word “*façade*” comes from the world of architecture
  - A façade is one side of a building (usually the front)
- In OOD, a façade is an object that provides a simplified interface to a larger class or set of classes

Façade: The Problem

- Façade is useful when we have a large system with many classes that are independent of one another
  - A complex API
- In many cases, clients only need a small part of such APIs
  - Being exposed to the complex API makes it hard to use this system

Façade: The Solution

- Façade provides a simpler version of complex APIs
  - Façade does “all the dirty work” of handling the complex API
  - The client remains unaware of the complex API and is only required to know the simpler version
- Sketch of solution:
  - Build a new class (Façade)
  - Define the Façade class to have the desired (simpler) API
  - Implement this class using the larger, more complex system

Façade: Example
Making Pasta

```
public abstract class Item { … }

public class Pasta extends Item { … }

public class Salt extends Item { … }


public class SaltShaker {
        public Salt salt(int nTSpoons) { … }
}
```

```
public class Pantry {
        public Item get(String item) { … }
}


public class Pot {
        public void boil(int nLiters) { … }

        public void add(Item item) { … }
}
```

Façade: Example

Making Pasta

```
/** Chef class that implements the Façade
    Design Pattern */
public class Chef {
    private SaltShaker saltShaker;
    private Pantry pantry;
    private Pot pot;
    public Chef () {
        this.saltShaker = new SaltShaker();
        this.pantry = new Pantry();
        this.pot = new Pot();
    }
```

```
    public void makePasta() {
        pot.boil(2);
        Item pasta = pantry.get("pasta");
        pot.add(pasta);
        Salt salt = saltShaker.get(1);
        pot.add(salt);
        …
    }
}
```

Façade: Pros

- Pros
  - Clients only see the simple interface (substantially easier to learn and use it)
  - Clients remain **unaware** of **changes** in the internal system (assuming façade handles them correctly)
  - Sophisticated clients can use more advanced features by the original system (bypassing the façade)

More on Façade

- Facade does not add any new functionality
  - It’s just a simpler API for a complex system
- A Façade can also be used to give a complex system an API that matches other common (simpler) APIs

Façade: Example
Media Players

```
public class ComplexPlayer {
      public void play(String fileName,
                       int leftvolume,
                       int rightvolume,
                        double bass) { … }
      …
}
```

```
public interface SimplePlayer{
        public void play(String fileName);
}
```

```
public class ComplexPlayerFacade
                implements SimplePlayer {
      private ComplexPlayer player;
      public ComplexPlayerFacade () {
                this.player = new ComplexPlayer();
      }
      public void play(String fileName) {
             player.play(fileName, 50, 50, 0.5);
      }
}
```

So far…

- Reuse Mechanisms
  - Composition vs. Inheritance
- Casting
  - Up-casting vs. down-casing, instanceof

So far…

- Intro to Design Patterns
  - Good solutions to recurrent problems
- Façade Design Pattern
  - Simplified API for complex systems

Next Week

- Intro to Generics
- Java Collections