6.170 Lecture 15
Design Patterns

Michael Ernst
John Guttag
MIT EECS

Outline

- **Introduction to design patterns**
- **Creational patterns (constructing objects)**
- **Structural patterns (controlling heap layout)**
- **Behavioral patterns (affecting object semantics)**

What is a design pattern?

- a standard solution to a common programming problem
- a technique for making code more flexible by making it meet certain criteria
- a design or implementation structure that achieves a particular purpose
- a high-level programming idiom
- shorthand for describing certain aspects of program organization
- connections among program components
- the shape of a heap snapshot or object model

Example 1: Encapsulation (data hiding)

Problem: Exposed fields can be directly manipulated
Violations of the representation invariant
Dependences prevent changing the implementation

Solution: Hide some components
Permit only stylized access to the object

Disadvantages:
Interface may not (efficiently) provide all desired operations
Indirection may reduce performance

Example 2: Subclassing (inheritance)

This repetition is tedious, error-prone, and a maintenance headache.

Problem: Repetition in implementations
Similar abstractions have similar members (fields, methods)

Solution: Inherit default members from a superclass
Select an implementation via run-time dispatching

Disadvantages:
Code for a class is spread out, potentially reducing understandability
Run-time dispatching introduces overhead

Example 3: Iteration

Problem: To access all members of a collection, must perform a specialized traversal for each data structure
Introduces undesirable dependences
Does not generalize to other collections

The implementation has knowledge about the representation.

Solution:
The implementation performs traversals, does bookkeeping
Results are communicated to clients via a standard interface

Disadvantages:
Iteration order is fixed by the implementation and not under the control of the client

Example 4: Exceptions

Problem:
- Errors in one part of the code should be handled elsewhere.
- Code should not be cluttered with error-handling code.
- Return values should not be preempted by error codes.

Solution: Language structures for throwing and catching exceptions

Disadvantages:
- Code may still be cluttered.
- It may be hard to know where an exception will be handled.
- Use of exceptions for normal control flow may be confusing and inefficient.

When (not) to use design patterns

Rule 1: delay

Get something basic working first, then improve it once you understand it.

Design patterns can increase or decrease understandability
- Add indirection, increase code size
- Improve modularity, separate concerns, ease description

If your design or implementation has a problem, consider design patterns that address that problem

Canonical reference: the "Gang of Four" book
- *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley, 1995.

Another good reference for Java
- *Effective Java: Programming Language Guide*, by Joshua Bloch, Addison-Wesley, 2001.

Why should you care?

You could come up with these solutions on your own

You shouldn't have to!

A design pattern is a known solution to a known problem

Creational patterns

Factories
- Factory method
- Factory object
- Prototype

Sharing
- Singleton
- Interning
- Flyweight

Factories

Problem: client desires control over object creation

Factory method: put code in methods in client

Factory object: put code in a separate object

Prototype: put code in clone methods

Example: bicycle race

CreateRace is a factory method. It may seem strange that it appears in Race; we will see how to move it outside Race shortly.

```
class Race {

  Race createRace() {
    Frame frame1 = new Frame();
    Wheel front1 = new Wheel();
    Wheel rear1 = new Wheel();
    Bicycle bike1 = new Bicycle(frame1, front1, rear1);

    Frame frame2 = new Frame();
    Wheel frontWheel2 = new Wheel();
    Wheel rearWheel2 = new Wheel();
    Bicycle bike2 = new Bicycle(frame2, front2, rear2);

    ...
  }

}
```

Example: Tour de France

```
class TourDeFrance extends Race {

  Race createRace() {
    Frame frame1 = new RacingFrame();
    Wheel front1 = new Wheel700c();
    Wheel rear1 = new Wheel700c();
    Bicycle bike1 = new Bicycle(frame1, front1, rear1);

    Frame frame2 = new RacingFrame();
    Wheel frontWheel2 = new Wheel700c();
    Wheel rearWheel2 = new Wheel700c();
    Bicycle bike2 = new Bicycle(frame2, front2, rear2);

    ...
  }

}
```


Example: Cyclocross

```
class Cyclocross extends Race {

  Race createRace() {
    Frame frame1 = new MountainFrame();
    Wheel front1 = new Wheel26in();
    Wheel rear1 = new Wheel26in();
    Bicycle bike1 = new Bicycle(frame1, front1, rear1);

    Frame frame2 = new MountainFrame();
    Wheel frontWheel2 = new Wheel26in();
    Wheel rearWheel2 = new Wheel26in();
    Bicycle bike2 = new Bicycle(frame2, front2, rear2);

    ...
  }

}
```


Factory method

```
class Race {
  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear); }
  // Return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }
  Race createRace() {
    Bicycle bike1 = completeBicycle();
    Bicycle bike2 = completeBicycle();
    ...
  }
}
```


Code for specific races, using factory methods

```
class TourDeFrance extends Race {
  Frame createFrame() { return new RacingFrame(); }
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}

class Cyclocross extends Race {
  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new MountainBicycle(frame, front, rear);
  }
}
```


Factory objects encapsulate factory methods

Same code as before, but in a separate object

```
class BicycleFactory {
  Frame createFrame() { return new Frame(); }
  Wheel createWheel() { return new Wheel(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new Bicycle(frame, front, rear);
  }

  // return a complete bicycle without needing any arguments
  Bicycle completeBicycle() {
    Frame frame = createFrame();
    Wheel frontWheel = createWheel();
    Wheel rearWheel = createWheel();
    return createBicycle(frame, frontWheel, rearWheel);
  }
}
```


Specializations of the factory object

```
class RacingBicycleFactory {
  Frame createFrame() { return new RacingFrame(); }
  Wheel createWheel() { return new Wheel700c(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new RacingBicycle(frame, front, rear);
  }
}

class MountainBicycleFactory {
  Frame createFrame() { return new MountainFrame(); }
  Wheel createWheel() { return new Wheel26inch(); }
  Bicycle createBicycle(Frame frame, Wheel front, Wheel rear) {
    return new MountainBicycle(frame, front, rear);
  }
}
```

Use of the factory object

```
class Race {
  BicycleFactory bfactory;
  // constructor
  Race() { bfactory = new BicycleFactory(); }
  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }
}

class TourDeFrance extends Race {
  // constructor
  TourDeFrance() { bfactory = new RacingBicycleFactory(); }
}

class Cyclocross extends Race {
  // constructor
  Cyclocross() { bfactory = new MountainBicycleFactory(); }
}
```


Separate control over bicycles and races

```
class Race {
  BicycleFactory bfactory;
  // constructor
  Race(BicycleFactory bfactory) { this.bfactory = bfactory; }
  Race createRace() {
    Bicycle bike1 = bfactory.completeBicycle();
    Bicycle bike2 = bfactory.completeBicycle();
    ...
  }
}
// No special constructor for TourDeFrance or for Cyclocross
```

Now we can specify the race and the bicycle separately:

```
new TourDeFrance(new TricycleFactory())
```


Prototype

- Every object is itself a factory
- Each class contains a `clone` method that creates a copy of the receiver object

```
class Bicyle {
  Object clone() { ... }
}
```

Why is `Object` the return type of `clone`?

clone is declared in Object, and Java does not permit subclasses to change the return type of an overridden method.

Using prototypes

```
class Race {
  Bicycle bproto;
  // constructor
  Race(Bicycle bproto) { this.bproto = bproto; }
  Race createRace() {
    Bicycle bike1 = (Bicycle) bproto.clone();
    Bicycle bike2 = (Bicycle) bproto.clone();
    ...
  }
}
```

Again, we can specify the race and the bicycle separately:

```
new TourDeFrance(new Tricycle())
```


Sharing

- Singleton: only one object exists at runtime
- Interning: only one object with a particular (abstract) value exists at runtime
- Flyweight: separate intrinsic and extrinsic state, represent them separately, and intern the intrinsic state

Singleton

Only one object of the given type exists

```
class Bank {
  private static bank theBank;

  // constructor
  private Bank() { ... }

  // factory method
  public static getBank() {
    if (theBank == null) {
      theBank = new Bank();
    }
    return theBank;
  }
  ...
}
```

The second weakness of Java constructors

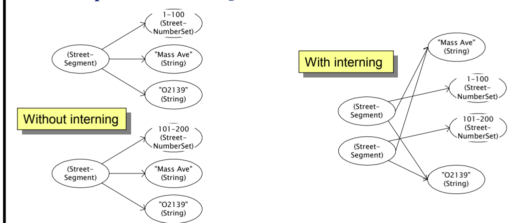
Java constructors always return a new object, never a pre-existing object

Interning

Reuse existing objects instead of creating new ones

Permitted only for immutable objects

Example: `StreetSegment`



Interning mechanism

Maintain a collection of all objects

If an object already appears, return that instead

```
HashMap segnames = new HashMap();  // why not a Set?
String canonicalName(String n) {
  if (segnames.containsKey(n)) {
    return segnames.get(n);
  } else {
    segnames.put(n, n);
    return n;
  }
}
```

Set supports `contains` but not `get`

Java builds this in for strings: `String.intern()`

Two approaches:

- create the object, but perhaps discard it and return another
- check against the arguments before creating the new object

Flyweight

Separate the intrinsic (same across all objects) and extrinsic (different for different objects) state

Intern the intrinsic state

Good when most of the object is immutable

Example: bicycle spoke

```
class Wheel {
  FullSpoke[] spokes;
  ...
}
class FullSpoke {
  int length;
  int diameter;
  bool tapered;
  Metal material;
  float weight;
  float threading;
  bool crimped;
  int location;   // rim and hub holes this is installed in
}
```

Typically 32 or 36 spokes per wheel, but only 3 varieties per bicycle.

In a 10,000-bike race, hundreds of spoke varieties, millions of instances

Alternatives to FullSpoke

```
class Spoke {
  int length;
  int diameter;
  boolean tapered;
  Metal material;
  float weight;
  float threading;
  boolean crimped;
}
```

This doesn't work: it's the same as FullSpoke

```
class InstalledSpokeFull extends Spoke {
  int location;
}
```

This does work, but there is a better solution

```
class InstalledSpokeWrapper {
  Spoke s;
  int location;
}
```

Original code to true (align) a wheel

```
class FullSpoke {
  // Tension the spoke by turning the nipple the
  // specified number of turns.
  void tighten(int turns) {
    ... location ...
  }
}

class Wheel {
  FullSpoke[] spokes;
  void align() {
    while (wheel is misaligned) {
      ... spokes[i].tighten(numturns) ...
    }
  }
}
```


Flyweight code to true (align) a wheel

```
class Spoke {
  void tighten(int turns, int location) {
    ... location ...
  }
}

class Wheel {
  Spoke[] spokes;

  void align() {
    while (wheel is misaligned) {
      ... spokes[i].tighten(numturns, i) ...
    }
  }
}
```


Flyweight discussion

What if `FullSpoke` contains a `wheel` field pointing at the `Wheel` containing it?

> `Wheel` methods pass this to the methods that use the `wheel` field.

What if `FullSpoke` contains a `boolean broken` field?

> Add an array of `booleans` in `Wheel`, parallel to the array of `Spokes`.

Flyweight is manageable only if there are very few mutable (extrinsic) fields.

Flyweight complicates the code.

Use flyweight only when profiling has determined that space is a *serious* problem.