

6.170 Lecture 17 Design Patterns



Michael Ernst
MIT EECS

Outline

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

Michael Ernst/John Guttag

Spring 2001

Slide 2

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, object model, or MDD

Michael Ernst/John Guttag

Spring 2001

Slide 3

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

Michael Ernst/John Guttag

Spring 2001

Slide 4

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

Michael Ernst/John Guttag

Spring 2001

Slide 5

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 extend to other collections.

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

The implementation has knowledge about the representation.

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

Michael Ernst/John Guttag

Spring 2001

Slide 6

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.

Michael Ernst/John Guttag Spring 2001 Slide 7

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.

Michael Ernst/John Guttag Spring 2001 Slide 8

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

Michael Ernst/John Guttag Spring 2001 Slide 9

Creational patterns

Factories
Factory method
Factory object
Prototype

Sharing
Singleton
Interning
Flyweight

Michael Ernst/John Guttag Spring 2001 Slide 10

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

Michael Ernst/John Guttag Spring 2001 Slide 11

Example: bicycle race

```
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);  
  
        ...  
    }  
}
```

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

Michael Ernst/John Guttag Spring 2001 Slide 12

6.170 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);  
  
        ...  
    }  
}
```

Michael Ernst/John Guttag Spring 2001 Slide 13

6.170 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);  
  
        ...  
    }  
}
```

Michael Ernst/John Guttag Spring 2001 Slide 14

6.170 Factory method

```
class Race {  
    Frame createFrame() { return new RacingFrame(); }  
    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();  
        ...  
    }  
}
```

Michael Ernst/John Guttag Spring 2001 Slide 15

6.170 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);  
    }  
}
```

Michael Ernst/John Guttag Spring 2001 Slide 16

6.170 Factory objects encapsulate factory methods

Same code as before, but in a separate object

```
class BicycleFactory {  
    Frame createFrame() { return new RacingFrame(); }  
    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);  
    }  
}
```

Michael Ernst/John Guttag Spring 2001 Slide 17

6.170 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);  
    }  
}
```

Michael Ernst/John Guttag Spring 2001 Slide 18

6.170 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 (); }
}
```

Michael Ernst/John Guttag Spring 2001 Slide 19

6.170 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 ())
```

Michael Ernst/John Guttag Spring 2001 Slide 20

6.170 Prototype

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

```
class Bicycle {
    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.

Michael Ernst/John Guttag Spring 2001 Slide 21

6.170 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 ())
```

Michael Ernst/John Guttag Spring 2001 Slide 22

6.170 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

Michael Ernst/John Guttag Spring 2001 Slide 23

6.170 Singleton

Only one object of the given type exists

```
class Bank {
    private static Bank theBank;

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

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

Michael Ernst/John Guttag Spring 2001 Slide 24

6.170 The second weakness of Java constructors

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

Michael Ernst/John Guttag Spring 2001 Slide 25

6.170 Interning

Reuse existing objects instead of creating new ones
Permitted only for immutable objects
Example: **StreetSegment**

Michael Ernst/John Guttag Spring 2001 Slide 26

6.170 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)) { Set supports contains but not get
        return segnames.get(n);
    } else {
        segnames.put(n, n);
        return n;
    }
}
    
```

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

Michael Ernst/John Guttag Spring 2001 Slide 27

6.170 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

Michael Ernst/John Guttag Spring 2001 Slide 28

6.170 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

Michael Ernst/John Guttag Spring 2001 Slide 29

6.170 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;
}

Michael Ernst/John Guttag Spring 2001 Slide 30

6.170 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) ...
        }
    }
}
```

Michael Ernst/John Guttag Spring 2001 Slide 31

6.170 Flyweight code to true (align) a wheel

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

class Wheel {
    FullSpoke[] spokes;

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

Michael Ernst/John Guttag Spring 2001 Slide 32

6.170

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.

Michael Ernst/John Guttag Spring 2001 Slide 33