CSE 331
Software Design & Implementation

Hal Perkins

Winter 2013

Design Patterns I

(Slides by Mike Ernst and David Notkin)

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 design or implementation structure that achieves a particular purpose
- a high-level programming idiom

A technique for making code more flexible

- reduce coupling among program components

Shorthand for describing program design

- a description of connections among program components (static structure)
- the shape of a heap snapshot or object model (dynamic structure)

A few simple examples….

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)

Problem: Repetition in implementations

Similar abstractions have similar components (fields, methods)

Solution: Inherit default members from a superclass

Select an implementation via run-time dispatching

Disadvantages:

Code for a class is spread out, and thus less understandable

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

Solution:

- The implementation performs traversals, does bookkeeping
  - The implementation has knowledge about the representation
- Results are communicated to clients via a standard interface (e.g., **`hasNext(), next()`**)

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

Example 5: Generics

Problem:

Well-designed data structures hold one type of object

Solution:

Programming language checks for errors in contents

`List<Date>` instead of just **`List`**

Disadvantages:

More verbose types

Why design patterns?

Advanced programming languages like Java provide lots of powerful constructs – subtyping, interfaces, rich types and libraries, etc.

By the nature of programming languages, they can't make everything easy to solve

To the first order, design patterns are intended to overcome common problems that arise in even advanced object-oriented programming languages

They increase your vocabulary and your intellectual toolset

When (not) to use design patterns

Rule 1: delay

- Get something basic working first
- 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

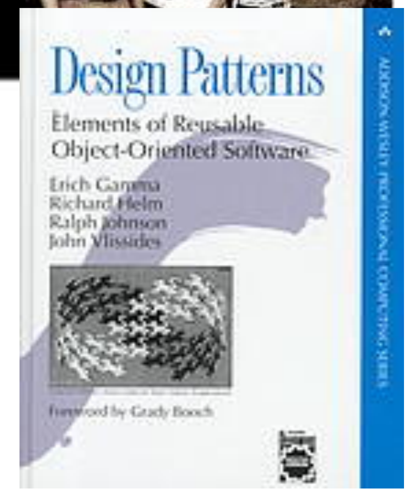
Why should you care?

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

Whence design patterns?



The Gang of Four (GoF) – Gamma, Helm, Johnson, Vlissides

Each an aggressive and thoughtful programmer

Empiricists, not theoreticians

Found they shared a number of “tricks” and decided to codify them – a key rule was that nothing could become a pattern unless they could identify at least three real examples

Patterns vs. patterns

The phrase “pattern” has been wildly overused since the GoF patterns have been introduced

“pattern” has become a synonym for “[somebody says] **X** is a good way to write programs.”

> And “anti-pattern” has become a synonym for “[somebody says] **Y** is a bad way to write programs.”

A graduate student recently studied so-called “security patterns” and found that very few of them were really GoF-style patterns

GoF-style patterns have richness, history, language-independence, documentation and thus (most likely) far more staying power

An example of a GoF pattern

Given a class C, what if you want to guarantee that there is precisely one instance of C in your program?

And you want that instance globally available?

First, why might you want this?

Second, how might you achieve this?

Possible reasons for Singleton

One **`RandomNumber`** generator

One graph model object

One **`KeyboardReader`**, etc…

Make it easier to ensure some key invariants

Make it easier to control when that single instance is created – can be important for large objects

…

Several solutions

```
public class Singleton {
  private static final Singleton instance
      = new Singleton(); // Private constructor prevents
                         // instantiation from other classes
  private Singleton() { }
  public static Singleton getInstance() {
    return instance;
  }
}
```

Eager allocation of instance

```
public class Singleton {
  private static Singleton instance;
  private Singleton() { }
  public static synchronized Singleton getInstance() {
    if (instance == null) {
      instance = new Singleton();
    }
    return instance;
  }
}
```

Lazy allocation of instance

GoF patterns: three categories

Creational Patterns – these abstract the object-instantiation process

Factory Method, Abstract Factory, Singleton, Builder, Prototype, …

Structural Patterns – these abstract how objects/classes can be combined

Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy, …

Behavioral Patterns – these abstract communication between objects

Command, Interpreter, Iterator, Mediator, Observer, State, Strategy, Chain of Responsibility, Visitor, Template Method, …

Blue = ones we've seen already

Creational patterns

Constructors in Java are inflexible
- Can't return a subtype of the class they belong to
- Always return a fresh new object, never re-use one

Problem: client desires control over object creation

Factory method
- Hides decisions about object creation
- Implementation: put code in methods in client

Factory object
- Bundles factory methods for a family of types
- Implementation: put code in a separate object

Prototype
- Every object is a factory, can create more objects like itself
- Implementation: put code in **`clone`** methods

Motivation for factories: Changing implementations

Supertypes support multiple implementations

```
interface Matrix { ... }
class SparseMatrix implements Matrix { ... }
class DenseMatrix implements Matrix { ... }
```

Clients use the supertype (**`Matrix`**)

- Still need to use a **`SparseMatrix`** or **`DenseMatrix`** constructor
- Switching implementations requires code changes

Use of factories

Factory

```
class MatrixFactory {
  public static Matrix createMatrix() {
    return new SparseMatrix();
  }
}
```

Clients call **`createMatrix`**, not a particular constructor

Advantages

- To switch the implementation, only change one place
- Can decide what type of matrix to create

Example: bicycle race

```
class Race {

  // factory method for bicycle race
  Race createRace() {
    Bicycle bike1 = new Bicycle();
    Bicycle bike2 = new Bicycle();
    ...
  }

}
```

Example: Tour de France

```
class TourDeFrance extends Race {

  // factory method
  Race createRace() {
    Bicycle bike1 = new RoadBicycle();
    Bicycle bike2 = new RoadBicycle();
    ...
  }

}
```

Example: Cyclocross

```
class Cyclocross extends Race {

  // factory method
  Race createRace() {
    Bicycle bike1 = new MountainBicycle();
    Bicycle bike2 = new MountainBicycle();
    ...
  }

}
```

Factory method for Bicycle

```
class Race {
  Bicycle createBicycle() { ... }
  Race createRace() {
    Bicycle bike1 = createBicycle();
    Bicycle bike2 = createBicycle();
    ...
  }
}
```

Use a factory method to avoid dependence on specific new kind of bicycle in **`createRace`**()

Code using Bicycle factory methods

```
class Race {
  Bicycle createBicycle() { ... }
  Race createRace() {
    Bicycle bike1 = createBicycle();
    Bicycle bike2 = createBicycle();
    ...
  }
}

class TourDeFrance extends Race {
  Bicycle createBicycle() {
    return new RoadBicycle();
  }
}

class Cyclocross extends Race {
  Bicycle createBicycle(Frame) {
    return new MountainBicycle();
  }
}
```

Factory objects/classes encapsulate factory methods

```
class BicycleFactory {
  Bicycle createBicycle() { ... }
  Frame createFrame() { ... }
  Wheel createWheel() { ... }
  ...
}

class RoadBicycleFactory extends BicycleFactory {
  Bicycle createBicycle() {
    return new RoadBicycle();
  }
}

class MountainBicycleFactory extends BicycleFactory {
  Bicycle createBicycle() {
    return new MountainBicycle();
  }
}
```

Using a factory object

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

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

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())
```

DateFormat factory methods

DateFormat class encapsulates knowledge about how to format dates and times as text

- Options: just date? just time? date+time? where in the world?
- Instead of passing all options to constructor, use factories.
- The subtype created doesn't need to be specified.

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(DateFormat.FULL,
   Locale.FRANCE);
Date today = new Date();
System.out.println(df1.format(today)); // "Jul 4, 1776"
System.out.println(df2.format(today)); // "10:15:00 AM"
System.out.println(df3.format(today)); // "juedi 4 juillet 1776"
```

Prototype pattern

Every object is itself a factory

Each class contains a **`clone`** method that creates a copy of the receiver object

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

Often, **`Object`** is the return type of **`clone`**

- **`clone`** is declared in **`Object`**
- Design flaw in Java 1.4 and earlier: the return type may not change covariantly in an overridden method
  - i.e., return type could not be made more restrictive
  - This is a problem for achieving true subtyping

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())
```

Dependency injection

Change the factory without changing the code

With a regular in-code factory:

```
BicycleFactory f = new TricycleFactory();
Race r = new TourDeFrance(f)
```

With external dependency injection:

```
BicycleFactory f
    = ((BicycleFactory)
          DependencyManager.get("BicycleFactory"));
Race r = new TourDeFrance(f);
```

plus an external file:

```
<service-point id="BicycleFactory">
  <invoke-factory>
    <construct class="Bicycle">
      <service>Tricycle</service>
    </construct>
  </invoke-factory>
</service-point>
```

+ Change the factory without recompiling
- Harder to understand
- Easier to make mistakes

Sharing

Recall the second weakness of Java constructors

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

Singleton: only one object exists at runtime

Factory method returns the same object every time (we've seen this already)

Interning: only one object with a particular (abstract) value exists at runtime

Factory method returns an existing object, not a new one

Flyweight: separate intrinsic and extrinsic state, represent them separately, and intern the intrinsic state

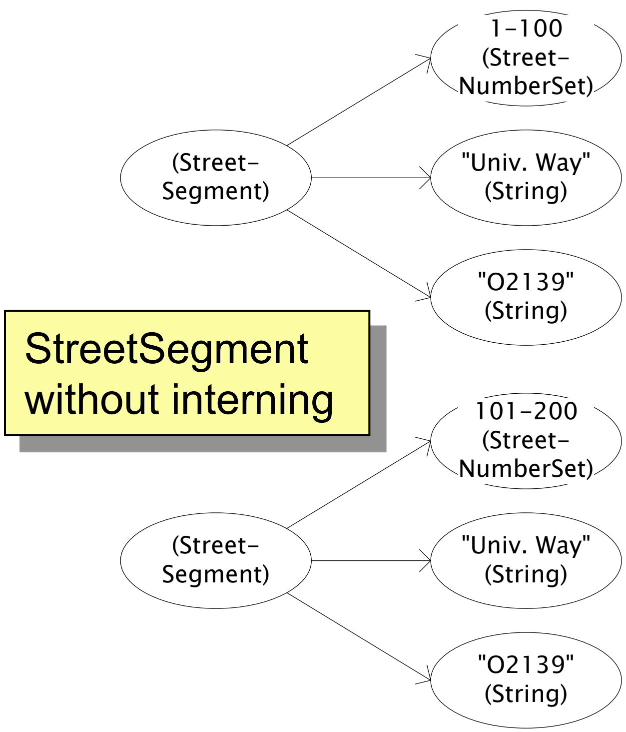
Implicit representation uses no space

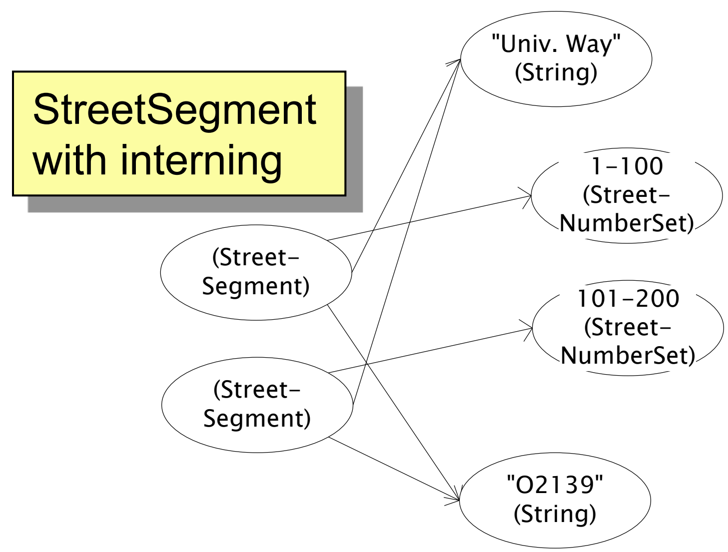
Interning pattern

Reuse existing objects instead of creating new ones

- Less space
- May compare with == instead of `equals()`

Permitted only for immutable objects





Interning mechanism

Maintain a collection of all objects

If an object already appears, return that instead

```
HashMap<String, String> segnames;   // why not Set<String>?
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

java.lang.Boolean does not use the Interning pattern

```
public class Boolean {
  private final boolean value;
  // construct a new Boolean value
  public Boolean(boolean value) {
    this.value = value;
  }

  public static Boolean FALSE = new Boolean(false);
  public static Boolean TRUE = new Boolean(true);

  // factory method that uses interning
  public static valueOf(boolean value) {
    if (value) {
      return TRUE;
    } else {
      return FALSE;
    }
  }
}
```

Recognition of the problem

Javadoc for `Boolean` constructor:

Allocates a `Boolean` object representing the value argument.

Note: It is rarely appropriate to use this constructor. Unless a new instance is required, the static factory `valueOf(boolean)` is generally a better choice. It is likely to yield significantly better space and time performance.

Josh Bloch (JavaWorld, January 4, 2004):

The `Boolean` type should not have had public constructors. There's really no great advantage to allow multiple `true`s or multiple `false`s, and I've seen programs that produce millions of `true`s and millions of `false`s, creating needless work for the garbage collector.

So, in the case of immutables, I think factory methods are great.

Flyweight pattern

Good when many objects are mostly the same
- Interning works only if objects are entirely the same (and immutable!)

Intrinsic state: same across all objects
- Technique: intern it (interning requires immutability)

Extrinsic state: different for different objects
- Represent it explicitly
- Advanced technique: make it implicit (don’t even represent it!)
  - Making it implicit requires immutability (or other properties)

Example without flyweight: 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 bike race, hundreds of spoke varieties, millions of instances

Alternatives to FullSpoke

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

This doesn't save space: it's the same as FullSpoke

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

This saves space

```
class InstalledSpokeWrapper {
  IntrinsicSpoke s;     // refer to interned object
  int location;
}
```

… but flyweight version uses even less space

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 ...    // location is a field
  }
}

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

What is the value of the **location** field in **spokes[i]**?

Flyweight code to true (align) a wheel

```
class IntrinsicSpoke {
  void tighten(int turns, int location) {
    ... location ...    // location is a parameter
  }
}

class Wheel {
  IntrinsicSpoke[] spokes;

  void align() {
    while (wheel is misaligned) {
      // tension the ith spoke, which affects the wheel
      ... spokes[i].tighten(numturns, i) ...
    }
  }
}
```

Flyweight discussion

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

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

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

Add an array of **`boolean`**s in **`Wheel`**, parallel to the array of **`Spoke`**s.

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.