CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
DESIGN PATTERNS II

Autumn 2011

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
    - That is, the 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 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 (for example, without changing any Java code the program might call a different factory)

A brief aside: call graphs

- A call graph is a set of pairs describing, for a given program, which units (usually methods) call other units (usually methods)
- Eclipse, for example, has a **Call Hierarchy** view (where the **Callee Hierarchy** option is often best) that is at times useful in programming

```
{<main,readCatalog(String)>,
 <readCatalog(String),readCatalog(InputStream)>,
 …}
```

- This is a *static call graph* – analyze the program and return a call graph representing all calls that could happen in any possible execution of the program
  - (A *dynamic call graph* is one built by executing the program one or more times and returning all calls that did take place in those executions)
- Static call graphs are generally expected to be "conservative" – that is, there are no false negatives, meaning that every **<A,B>** that can ever be invoked over any execution is included in the call graph

Precision

- Of course, there's an easy algorithm to create a not-very-useful static call graph ☺

```
for (m : method)
  for (n : method)
    include <m,n> in call graph
```

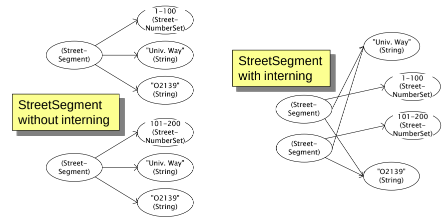
- A question is precision – how many false positives are included (**<A,B>** that are included to be conservative but that cannot ever be executed)?
- And inversion-of-control complicates this further – using the dependency injection pattern, for example, creates a static connection between **<client,Tricycle>** that would require quite complex analysis to report
  - In practice all or almost all inversion-of-control invocations are omitted in static call graphs
  - Even if a programmer is not using a static call graph, he or she is going through similar reasoning, and can also become confused or required to analyze in more detail in the face of inversion-of-control – so be thoughtful and careful about this issue!
  - This fuzzy connection can make it harder to understand and to change a program, although it can also make it easier to change a program – that's right, it can make it harder **and** easier to change at the same time

Sharing

- Interning: only one object with a particular (abstract) value exists at run-time
  - 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;
  }
}
```

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

`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

Structural patterns: Wrappers

- A *wrapper* translates between incompatible interfaces
- Wrappers are a thin veneer over an encapsulated class
  - modify the interface
  - extend behavior
  - restrict access
- The encapsulated class does most of the work

Wrapper Pattern	Functionality	Interface
Adapter	same	different
Decorator	different	same
Proxy	same	same

Adapter

- Change an interface without changing functionality
  - rename a method
  - convert units
  - implement a method in terms of another
- Example
  - Have the **Rectangle** class on the top right
  - Want to be able to use the **NonScaleableRectangle** class on the bottom right, which is not a **Rectangle**

```
interface Rectangle {
  // grow or shrink by the given factor
  void scale(float factor);
  ...
  float getWidth();
  float area();
}
class myClass {
  void myMethod(Rectangle r) {
    ...   r.scale(2);   ...
  }
}
class NonScaleableRectangle {
 void setWidth(float width) { ... }
  void setHeight(float height) { ... }
       // no scale method
  ...
}
```

Adapting via subclassing

```
class ScaleableRectangle1 extends NonScaleableRectangle
                          implements Rectangle {
  void scale(float factor) {
    setWidth(factor * getWidth());
    setHeight(factor * getHeight());
  }
}
```

Adapting via delegation: Forwarding requests to another object

```
class ScaleableRectangle2 implements Rectangle {
  NonScaleableRectangle r;
  ScaleableRectangle2(NonScaleableRectangle r) {
    this.r = r;
  }

  void scale(float factor) {
    setWidth(factor * r.getWidth());
    setHeight(factor * r.getHeight());
  }

  float getWidth() { return r.getWidth(); }
  float circumference() { return r.circumference(); }
  ...
}
```

Subclassing vs. delegation

- Subclassing
  - automatically gives access to all methods of superclass
  - built into the language (syntax, efficiency)
- Delegation
  - permits cleaner removal of methods (compile-time checking)
  - wrappers can be added and removed dynamically
  - objects of arbitrary concrete classes can be wrapped
  - multiple wrappers can be composed
- Some wrappers have qualities of more than one of adapter, decorator, and proxy

Decorator

- Add functionality without changing the interface
- Add to existing methods to do something additional (while still preserving the previous specification)
- Not all subclassing is decoration

Decorator: Bordered windows

```
interface Window {
  // rectangle bounding the window
  Rectangle bounds();
  // draw this on the specified screen
  void draw(Screen s);
  ...
}

class WindowImpl implements Window {
  ...
}
```

Bordered window implementations

```
class BorderedWindow1 extends WindowImpl {
  void draw(Screen s) {
    super.draw(s);
    bounds().draw(s);
  }
}
```

Subclassing

```
class BorderedWindow2 implements Window {
  Window innerWindow;
  BorderedWindow2(Window innerWindow) {
    this.innerWindow = innerWindow;
  }
  void draw(Screen s) {
    innerWindow.draw(s);
    innerWindow.bounds().draw(s);
  }
}
```

Delegation permits multiple borders, borders and/or shading, etc.

A decorator can remove functionality

- Remove functionality without changing the interface
- Example: **`UnmodifiableList`**
  - What does it do about methods like **`add`** and **`put`**?

Proxy

- Same interface and functionality as the wrapped class
- Control access to other objects
  - communication: manage network details when using a remote object
  - locking: serialize access by multiple clients
  - security: permit access only if proper credentials
  - creation: object might not yet exist (creation is expensive)
    - hide latency when creating object
    - avoid work if object is never used

Visitor pattern

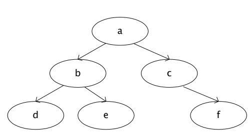
- Visitor encodes a traversal of a hierarchical data structure
- Nodes – objects in the hierarchy – accept visitors; visitors visit nodes
- **`n.accept(v)`** performs a depth-first traversal of the structure rooted at **`n`**, performing **`v`**'s operation on each element of the structure

```
class Node {
  void accept(Visitor v) {
    for each child of node {
      child.accept(v);
    }
    v.visit(this);
  }
}

class Visitor {
  void visit(Node n) {
    // perform work on n
  }
}
```

Sequence of calls to accept and visit

```
a.accept(v)
  b.accept(v)
    d.accept(v)
      v.visit(d)
    e.accept(v)
      v.visit(e)
    v.visit(b)
  c.accept(v)
    f.accept(v)
      v.visit(f)
    v.visit(c)
  v.visit(a)
```



- Sequence of calls to visit: `<d, e, b, f, c, a>`

Implementing visitor

- You must add definitions of **`visit`** and **`accept`**
- **`visit`** might count nodes, perform typechecking, etc.
- It is easy to add operations (visitors), hard to add nodes (modify each existing visitor)
- Visitors are similar to iterators: each element of the data structure is presented in turn to the visit method
  - Visitors have knowledge of the structure, not just the sequence

Next steps

- Assignment 3: due Sunday October 30, 11:59PM
- Lectures
  - M (Patterns III/GUI)
  - W (Midterm review, including example questions)
- Upcoming: Friday 10/28, in class midterm – open book, open note, closed neighbor, closed electronic devices