Design Patterns

Bibliography

- Thinking in Patterns (online version)
  Bruce Eckel
- Design Patterns, Elements of Reusable Object-Oriented Software
  Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
  Addison-Wesley
- Patterns in Java, Volume 1
  Mark Grand
  Wiley

What is it?

- “Design patterns help you learn from others' successes instead of your own failures.” (Mark Johnson)
- A design pattern is an especially clever and insightful way of solving a particular class of problems
- A design pattern has already been proved good in several different situations

The goal

- The goal is to write programs that are easy to extend and to reuse
- For this purpose, you must isolate things that can change in your program

Classifying Design Patterns

- Creational: how an object can be created
  Factory, singleton
- Structural: how objects ou classes are connected
  Decorator, composite, adapter, proxy
- Behavioral: concern behavior of objects
  Observer, iterator, strategy, command
- ...

Advantages of design patterns

- Helps you to write good, tested and extensible programs
- Gives you a common vocabulary when you speak to another computer scientist: "did you tried a factory method? "
- Adds another level of abstraction to design and understand code

Standard Description of a pattern

- Pattern name and aliases
- Intent: a short description of what the pattern do, what problem does it address
- Motivation: an example of the problem and how the pattern can help
- Applicability: situations where the pattern can be applied
- Structure: static structure (class diagram) of the classes involved in the pattern

Description of a pattern (cont.)

- Participants: description and responsibilities of the classes and objects
- Collaborations: dynamic description of the collaboration between objects
- Consequences: trade-offs and consequences if you use this pattern
- Implementation: techniques for implementations; pitfalls and hints
- Sample code: code fragments to illustrate the implementation

Description of a pattern (cont.)

- Known uses: examples of uses of the pattern in real systems
- Related patterns: other patterns related to this pattern; other patterns that this pattern can use or in which this pattern can be used

An Example of Design Pattern

Factory

- Probably the more often used pattern
- Also called factory method or virtual constructor
- Intent: define an interface for creating an object; the actual class of the created instance is decided at runtime, according to the context

Factory

- Motivation: a program reads data in a file in order to create shapes that are rectangles or circles
- The place to read the data is an AbstractShape
- AbstractShape reads data and delegates the creation of the shapes to the concerned subclass of AbstractShape (classes `Rectangle` or `Circle`)

Using AbstractShape

```
BufferedReader br =
  new BufferedReader(
    new FileReader("shapes"));
while ((shape = AbstractShape.read(br))
       != null) {
  // processing the read shape
  . . .
}
```


AbstractShape

```
public Shape read(BufferedReader br)
       throws IOException {
  String line = br.readLine();
  if (line == null) return null;
  StringTokenizer st =
      new StringTokenizer(line, "|");
  String type = br.nextToken();
  if (type.equals("circle"))
    return Circle.read(st);
  else if (type.equals("rectangle"))
    return Rectangle.read(st);
  else
    return null;
}
```

What is the problem?

Not extensible

Factory

- A factory returns an instance of the interface **Shape**; the actual class of the instance can be changed without modifying code that uses the factory
- You isolate what can change: the instantiation of the shape whose type can change

Factory: sample code

```
public class ShapeFactory {
  public Shape create(String type,
                      StringTokenizer st) {
    if (type.equals("circle"))
      return Circle.read(st);
    else if (type.equals("rectangle"))
      return Rectangle.read(st);
    else
      return null;
  }
}
```


Class Circle

```
public class Circle {
  . . .
  public static Circle read(StringTokenizer st)
  {
    Circle circle = new Circle();
    circle.initialize(st);
    return circle;
  }
  public void initialize(StringTokenizer st) {
    this.xCenter =
      Double.parseDouble(st.nextToken());
    . . .
  }
```


AbstractShape With the Factory

```
public Shape read(BufferedReader br)
       throws IOException {
  String line = br.readLine();
  if (line == null) return null;
  StringTokenizer st =
      new StringTokenizer(line, "|");
  String type = br.nexToken();
  return ShapeFactory.create(type, st);
}
```

Extending the factory

```
public class ShapeFactory2
        extends ShapeFactory {
  public Shape create(String type,
                      StringTokenizer st) {
    if (type.equals("polygon"))
      return Polygon.read(st);
    else
      return super.create(type, st);
  }
}
```


A difficulty

- You must indicate to the program which factory to use
- For example, by a setFactory method

Variants

- Chains of factories instead of using inheritance
- Map of factories; the key is the type of the objects and the value is the associated factory (polymorphic factory)

Polymorphic Factory

```
public abstract class ShapeFactory {
  private static
    Map factories = new HashMap();
  /** To be implemented by concrete factories
   *  that create particular shapes */
  protected Shape create(String type,
                         StringTokenizer st);

  public static final
    void addFactory(String type,
                    ShapeFactory factory) {
      factories.add(id, factory);
  }
```


Polymorphic Factory (cont.)

```
  /** Static method that will be called
   * from other classes */
  public static final
   Shape createShape(String type,
                     StringTokenizer st) {
   return (ShapeFactory)
     factories.get(id).create(type, st);
  }
}
```


Using a Polymorphic Factory

```
// Concrete factories that create
// particular shapes.
// They are added to the polymorphic factory
ShapeFactory sf1 = new ShapeFactory1();
ShapeFactory.add("Circle", sf1);
ShapeFactory.add("Rectangle", sf1);
ShapeFactory sf2 = new ShapeFactory2();
ShapeFactory.add("Polygon", sf2);
. . .
// Using the abstract polymorphic factory
ShapeFactory.create(type, st);
```

ShapeFactory1

```
public class ShapeFactory1
        extends ShapeFactory {
  // Implements the abstract method of ShapeFactory
  protected Shape create(String type,
                         StringTokenizer st) {
    if (type.equals("circle"))
      return Circle.read(st);
    else if (type.equals("rectangle"))
      return Rectangle.read(st);
    else
      return null;
  }
}
```


ShapeFactory2

```
public class ShapeFactory1
        extends ShapeFactory {
  protected Shape create(String type,
                         StringTokenizer st) {
    if (type.equals("polygon"))
      return Polygon.read(st);
    else
      return null;
  }
}
```


AbstractShape With the Factory

```
public Shape read(BufferedReader br)
        throws IOException {
  String line = br.readLine();
  if (line == null) return null;
  StringTokenizer st =
      new StringTokenizer(line, "|");
  String type = br.nexToken();
  return ShapeFactory.createShape(type, st);
}
```


A variant

- Each shape class creates its factory as a private inner class and add it to the ShapeFactory (in a static initializer)

Factory: sample code

```
public class Circle {
  . . .
  private static class Factory
                extends ShapeFactory {
    protected Shape create(StringTokenizer st){
      return read(st);
    }
  }
  static {
    ShapeFactory.addFactory("circle",
                            new Factory());
  }
  private static Shape read(StringTokenizer
  st){ . . . }
```


Structure