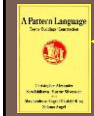


Published in 1994

Today's amazon.com stats

Inspired in part by

Amazon Best Sellers Rank: #2,069 in Books (See Top 100 in Books)
#1 in Books > Computers & Internet > Computer Science > Software Engineering > **Design Tools & Techniques**
#1 in Books > Computers & Internet > Programming > Software Design, Testing & Engineering > **Software Reuse**
#3 in Books > Nonfiction > Foreign Language Nonfiction > **French**

CSE 331
SOFTWARE DESIGN & IMPLEMENTATION
DESIGN PATTERNS I

Autumn 2011

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)

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

---

From a colleague

- FML. Today I got to write (in Java):

```
import java.util.Set;
import com.google.common.base.Function;
import com.google.common.collect.DiscreteDomains;
import com.google.common.collect.Iterables;
import com.google.common.collect.Ranges;

final int x = ...;
Set<Integer> indices =
   Ranges.closed(0, size).asSet(DiscreteDomains.integers());
Iterable<Coord> coords =
  Iterables.transform(indices, new Function<Integer,Coord>(){
    public Coord apply (Integer y) {
      return new Coord(x, y);
    }
  }
);
```

when I wanted to write (in Scala):

```
val x = ...;
val coords = 0 to size map(Coord(x, _))
```

No programming language is, or ever will be, perfect.

Extra-language solutions (tools, design patterns, etc.) are needed as well.

Perlis: "When someone says 'I want a programming language in which I need only say what I wish done,' give him a lollipop."

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

My first experience with patterns at Dagstuhl ⓘ with Helms and Vlissides

---

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 `Restaurant`, one `ShoppingCart`
- 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 (null == _instance) {
      _instance = new Singleton();
    } return _instance;
  }
}
```

Lazy allocation of instance

And there are more (in EJ, for 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`

Creational patterns: Factory method

- 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
  - Implementation can decide what type of matrix to create

Example: bicycle race

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

CreateRace is a factory method – why is it in Race?

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 Code using that method

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

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

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

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())
```

A semi-aside: *inversion of control*

- A number of modern design techniques – including many design patterns – exploit a notion mentioned in an earlier lecture: *inversion of control*
- In conventional flow-of-control, methods are called or invoked by name

```
double area = rectangle1.height() * rectangle1.width()
```

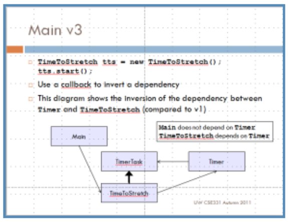
- The intent is to have the called method perform an action that the client needs to work properly – almost always, the result of the call is material to the post-condition of the caller either directly or indirectly
- This is true even if the exact method to be called is less clear due to overloading and/or overriding

Conventional flow-of-control

- For method **A** to call method **B**, **A** needs to know the name of **B** – usually, **B**'s class is imported
- This is vaguely like a telephone call – you can only call person **P** if you know his or her phone number **N**
  - A phone book gives you a way to find out the association between people and numbers
  - Kind of like the JDK gives you a way to find the association between computations you want and which methods perform those computations

Inversion of control

- At times, it is beneficial to be able to have method **A** invoke method **B** without knowing the name of **B**
- Like from several lectures ago – **`Timer`** can invoke **`TimeToStretch`** without **`Timer`** knowing its name
- **`Timer`** knows that something is invoked, but doesn't care what in the sense that **`Timer`**'s post-condition does not depend on any information computed by or returned by "whatever" is invoked
- Sometimes referred to as Hollywood's principle: "Don't call us, we'll call you"



Main v3

- `TimeToStretch tts = new TimeToStretch();`
  `tts.start();`
- Use a callback to invert a dependency
- This diagram shows the inversion of the dependency between `Timer` and `TimeToStretch` (compared to v1)

`Main` does not depend on `Timer`
`TimeToStretch` depends on `Timer`

`invokes` doesn't coincide with `names`

- In inversion of control, the `invokes` relation (which methods call which other methods) does not coincide with the `names` relation (which methods know the names of which other methods)
- Like the phone analogy, this is vaguely similar to radio or TV broadcasting – the broadcasting station doesn't know the names of the listeners, even though it is providing content to them
  - However, the listeners know the name (the frequency or the channel) of the station
- This allows some kinds of valuable flexibility in programs – for example, the actual task invoked by the **`Timer`** can be changed without modifying **`Timer`**, which increases the ease of reusing it
  - And **`TimeToStretch`** may also be more reusable due to more constrained dependences

But wait!

- Notkin said this class would focus on correctness far more than anything else (including performance, ease of change, etc.)
- But inversion of control at its core is intended to add flexibility, making things easier to change
- Well, yes… but …
  - Allowing programs to change in a more disciplined way serves correctness by leaving more components unchanged
  - There can be a clearer distinction between invocations that require some specific behavior vs. those that require much simpler properties of the invoked (but unnamed) methods
- At the same time, inversion of control can also make some aspects of correctness more complicated – and this is one reason that the disciplined use of it in design patterns is a plus

Next steps

- Assignment 3: due Sunday October 30, 11:59PM
- Lectures: F (Design Patterns)
- Upcoming: Friday 10/28, in class midterm – open book, open note, closed neighbor, closed electronic devices

Characteristic problems

- Representation exposure problem
  - Violate the representation invariant; dependences complicate changing the implementation
  - Hiding some components may permit only stylized access to the object
  - This may cause the interface to
- Disadvantages:
  - Interface may not (efficiently) provide all desired operations
  - Indirection may reduce performance