CSE 331
Software Design & Implementation

Kevin Zatloukal

Summer 2016

Design Patterns, Part 1

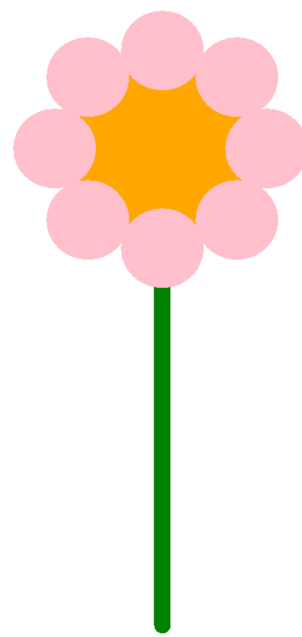
(Based on slides by Mike Ernst, Dan Grossman, David Notkin, Hal Perkins, Zach Tatlock)

Announcements

- Course evaluation: https://uw.iasystem.org/survey/163511
  - we can’t see the results until the course is over
- HW9 is due next Wednesday at 11pm, but…
  - (voluntary) demos during Wednesday lecture
  - need to finish by Tuesday night to participate
- Last reading quiz is due tonight at 11pm

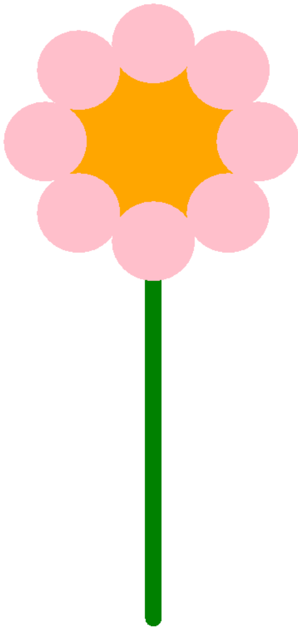
Review

Graphics Example 1



`Flower.java`

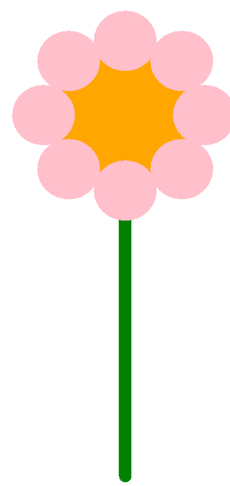
Graphics Example 2



`FlowerTimer.java`

Graphics Example 3

- Easier to do the drawing work in a drawing program
  - (if no timer) just save the drawing to an image
  - draw the image in `paintComponent`



`FlowerImage.java`

Example: Chat Client GUI

- We now can fully understand the chat client
  - uses BorderLayout and a couple of nested panels
    - one is a scrolling panel for the message area
  - uses two event loops: one for UI, one for networking

`ChatClientGUI.java`

Design Patterns

What is a design pattern?

A standard solution to a common programming problem

- a high-level programming idiom

Often a technique for making code more flexible

- reduces coupling among program components (at some cost)

Shorthand description of a software design

- well-known terminology improves communication
- makes it easier to think of using the technique

A couple *familiar* examples….

Example 1: Observer

Problem: other code needs to be called each time state changes
but we would like the component to be reusable

- can't just hard-code calls to everything that needs to be called

Solution:

- object maintains a list of observers with a known interface
- calls a method on each observer when state changes

Disadvantages:

- need extra code to add each observer
- potentially wastes memory by maintaining a list of objects that are known a priori (and are always the same)

Example 2: Iteration

Problem: accessing all members of a collection requires performing a specialized traversal for each data structure

- (makes clients strongly coupled to that data structure)

Solution:

- the *implementation* performs traversals, does bookkeeping
- results are communicated to clients via a standard interface (e.g., **`hasNext()`**, **`next()`**)

Disadvantages:

- iteration order fixed by the implementation (not the client)

Why (more) design patterns?

Design patterns are intended to capture common solutions / idioms, name them, make them easy to use to guide design

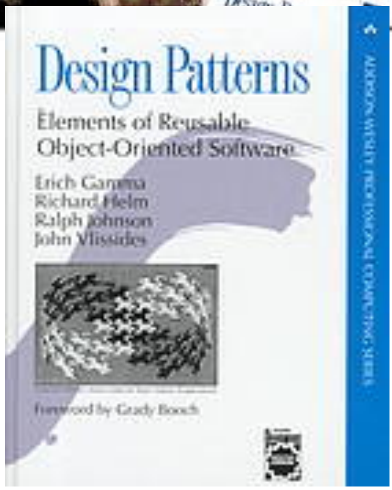
- they are high-level designs, not specific "coding tricks"

They increase your vocabulary and your intellectual toolset

Do not **overuse** them

- introducing new abstractions to your program has a cost
  - it makes the code more complicated
  - it takes time
- don't fix what isn't broken
  - wait until you have strong evidence that you will run into the problem that pattern is designed to solve

Origin of term



The “Gang of Four” (GoF)

– Gamma, Helm, Johnson, Vlissides

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 [different] examples
– for object-oriented programming
  - some patterns more general
  - others compensate for OOP shortcomings

Patterns vs patterns

The phrase *pattern* has been overused since GoF book

Often used as "[somebody says] **X** is a good way to write programs"

- and "anti-pattern" as "**Y** is a bad way to write programs"

These are useful, but GoF-style patterns are more important

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

An example GoF pattern

For some class `C`, guarantee that at run-time there is exactly one (globally visible) instance of `C`

First, *why* might you want this?

- what design goals are achieved?

Second, *how* might you achieve this?

- how to leverage language constructs to enforce the design

A pattern has a recognized *name*

- this is the *Singleton* pattern

Possible reasons for Singleton

- One **`RandomNumber`** generator
- One **`KeyboardReader`**, **`PrinterController`**, etc…
- One **`CampusPaths`**?

- Have an object with fields / methods that are "like public, static fields / methods" but have a **constructor** decide their values
  - e.g., have **`main`** decide which files to give **`CampusPaths`**
  - but rest of the code can just assume it exists

- Other benefits in certain situations
  - could delay expensive constructor until actually needed

How: multiple approaches

```java
public class Foo {
  private static final Foo instance = new Foo();
  // private constructor prevents instantiation outside class
  private Foo() { … }
  public static Foo getInstance() {
    return instance;
  }
  … instance methods as usual …
}
```

Eager allocation of instance

```java
public class Foo {
  private static Foo instance;
  // private constructor prevents instantiation outside class
  private Foo() { … }
  public static synchronized Foo getInstance() {
    if (instance == null) {
      instance = new Foo();
    }
    return instance;
  }
  … instance methods as usual …
}
```

Lazy allocation of instance

GoF patterns: three categories

Creational Patterns are about the object-creation process

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

Structural Patterns are about how objects/classes can be combined

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

Behavioral Patterns are about communication among objects

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

Green = ones we've seen already

Creational patterns

Constructors in Java are inflexible

1. Can't return a subtype of the class
2. Can’t reuse an existing object

Factories: patterns for how to create new objects

- Factory method, Factory object / Builder, Prototype

Sharing: patterns for reusing objects

- Singleton, Interning

Motivation for factories: Changing implementations

Supertypes support multiple implementations

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

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

BUT still call **`SparseMatrix`** or **`DenseMatrix`** constructor

- must decide concrete implementation *somewhere*
- might want to make the decision in one place
  - rather than all over in the code
- factory methods put this decision behind an abstraction

Use of factories

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

Clients call **`createMatrix`** instead of a particular constructor

- better version might take some parameters in order to decide whether sparse or dense is a better choice

Advantages:

- to switch the implementation, change only *one* place

`DateFormat` factory methods

`DateFormat` class encapsulates how to format dates & times

- options: just date, just time, date+time, w/ timezone, etc.
- instead of passing all options to constructor, use factories
- the subtype created by factory call need not be specified

```
DateFormat df1 = DateFormat.getDateInstance();
DateFormat df2 = DateFormat.getTimeInstance();
DateFormat df3 = DateFormat.getDateInstance(
                     DateFormat.FULL, Locale.FRANCE);

Date today = new Date();

df1.format(today);   // "Jul 4, 1776"
df2.format(today));  // "10:15:00 AM"
df3.format(today));  // "jeudi 4 juillet 1776"
```

Example: Bicycle race

```
class Race {
  public Race() {
    Bicycle bike1 = new Bicycle();
    Bicycle bike2 = new Bicycle();
    … // assume lots of other code here
  }
  …
}
```

Suppose there are different types of races

Each race needs its own type of bicycle…

Example: Tour de France

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    Bicycle bike1 = new RoadBicycle();
    Bicycle bike2 = new RoadBicycle();
    …
  }
  …
}
```

The Tour de France needs a road bike…

Example: Cyclocross

```
class Cyclocross extends Race {
  public Cyclocross() {
    Bicycle bike1 = new MountainBicycle();
    Bicycle bike2 = new MountainBicycle();
    …
  }
  …
}
```

And the cyclocross needs a mountain bike.

Problem: have to override the constructor in every **`Race`** subclass just to use a different subclass of **`Bicycle`**

Factory *method* for Bicycle

```
class Race {
  Bicycle createBicycle() { return new Bicycle(); }
  public Race() {
    Bicycle bike1 = createBicycle();
    Bicycle bike2 = createBicycle();
    ...
  }
}
```

Solution: use a factory method to avoid choosing which type to create

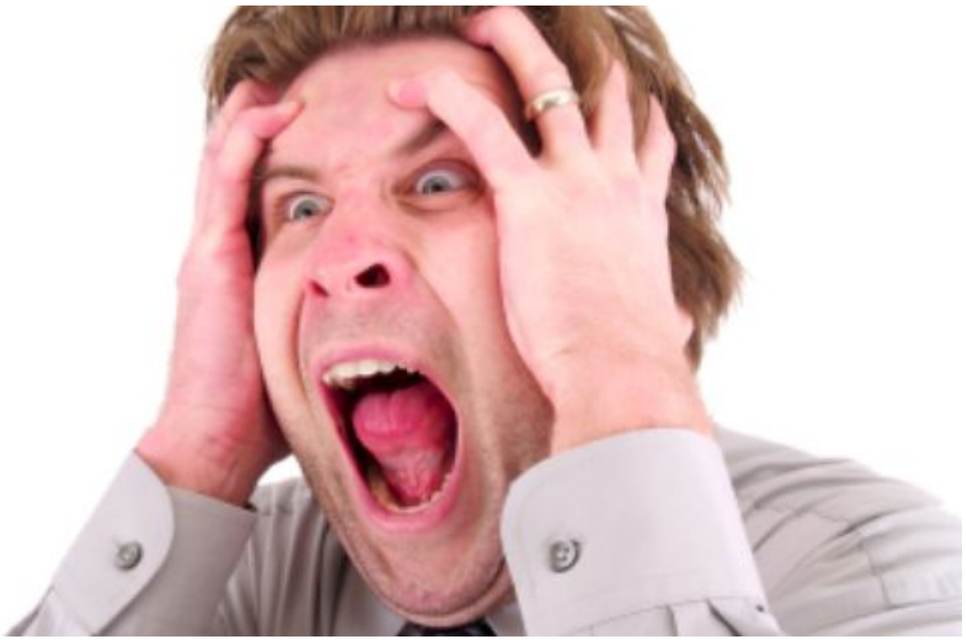
- let the subclass decide by overriding **`createBicycle`**

Subclasses override factory method

```
class TourDeFrance extends Race {
  Bicycle createBicycle() {
    return new RoadBicycle();
  }
  public TourDeFrance() { super(); }
}
class Cyclocross extends Race {
  Bicycle createBicycle() {
    return new MountainBicycle();
  }
  public Cyclocross() { super(); }
}
```

- Requires foresight to use factory method in superclass constructor
- Subtyping in the overriding methods!
- Supports other types of reuse (e.g. **`addBicycle`** could use it too)

A Brief Aside



Did you see what that code just did?

- it called a subclass method from a *constructor!*
- factory methods should usually be **static** methods

Factory objects

- Let's move the method into a separate class
  - so it's part of a *factory object*

- Advantages:
  - no longer risks horrifying bugs
  - can pass factories around around at runtime
    - e.g., let **`main`** decide which one to use

Factory ***objects*/classes** encapsulate factory method(s)

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

These are returning subtypes

Using a factory object

```
class Race {
  BicycleFactory bfactory;
  public Race(BicycleFactory f) {
    bfactory = f;
    Bicycle bike1 = bfactory.createBicycle();
    Bicycle bike2 = bfactory.createBicycle();
    …
  }
  public Race() { this(new BicycleFactory()); }
  …
}
```

Setting up the flexibility here:

- Factory object stored in a field, set by constructor
- Can take the factory as a constructor-argument
- But an implementation detail (?), so 0-argument constructor too
  - Java detail: call another constructor in same class with `this`

The subclasses

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    super(new RoadBicycleFactory());
  }
}

class Cyclocross extends Race {
  public Cyclocross() {
    super(new MountainBicycleFactory());
  }
}
```

Voila!

- Just call the superclass constructor with a different factory
- `Race` class had foresight to delegate “what to do to create a bicycle” to the factory object, making it more reusable

Separate control over bicycles and races

```
class TourDeFrance extends Race {
  public TourDeFrance() {
    super(new RoadBicycleFactory()); // or this(…)
  }
  public TourDeFrance(BicycleFactory f) {
    super(f);
  }
  …
}
```

By having factory-as-argument option, we can allow arbitrary mixing by client: **`new TourDeFrance(new TricycleFactory())`**

Less useful in this example (?): Swapping in different factory object whenever you want

Reminder: Not shown here is also using factories for creating *races*

Prototype pattern

- Each object is itself a factory:
  - objects contain a **`clone`** method that creates a copy

- Useful for objects that are created via a process
  - Example: java.awt.geom.AffineTransform
  - create by a sequence of calls to translate, scale, and rotate
  - easiest to make a similar one by copying and changing
    - saves the work of repeating all the common operations

Factories: summary

Goal: want more flexible abstractions for what class to instantiate

Factory method

- call a method to create the object
- method can do any computation and return any subtype

Factory object (also Builder)

- Factory has factory methods for some type(s)
- Builder has methods to describe object and then create it

Prototype

- every object is a factory, can create more objects like itself
- call `clone` to get a new object of same subtype as receiver

Dependency Injection

- put choice of subclass in a file to avoid source-code changes or even recompiling when decision changes
  - (not usually a big problem)