Design Patterns

MEEC

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 description of a software design:

- Well-known terminology improves communication/ documentation.
- Makes it easier to “think to use” a known technique.

Example 1: Encapsulation (data hiding)

Problem: Exposed fields can be directly manipulated.

Solution: Hide some components.

- Constrain ways to access the object.

Disadvantages:

- Interface may not (efficiently) provide all desired operations to all clients.
- Indirection may reduce performance.

Example 2: Subclassing (inheritance)

Problem: Repetition in implementations.

- Similar abstractions have similar components (fields and 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.
- Hard to design and specify a superclass.

Example 3: Iteration

Problem: To access all members of a collection, one must perform a specialized traversal for each data structure.

- Introduces undesirable dependences.
- Does not generalize to other collections.

Solution:

- Implement an **`Iterator`** to perform traversals.
- Results are communicated to clients via a standard interface (e.g., **`hasNext()`**, **`next()`**).

Disadvantages:

- Iteration order 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.
- Hard to remember and deal with code not running if an exception occurs in a callee.
- It may be hard to know where an exception will be handled.

Example 5: Generics

Problem:

- Well-designed (and used) 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 (more) design patterns?

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

- But it's not enough to "know everything in the language".
- Still many common problems not easy to solve.

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

- For high-level design, not specific "coding tricks".

They increase your vocabulary and your intellectual toolset.

Do not overuse them!

- Not every program needs the complexity of advanced design patterns.
- Instead, consider them to solve reuse/modularity problems that arise as your program evolves.

Why should you care?

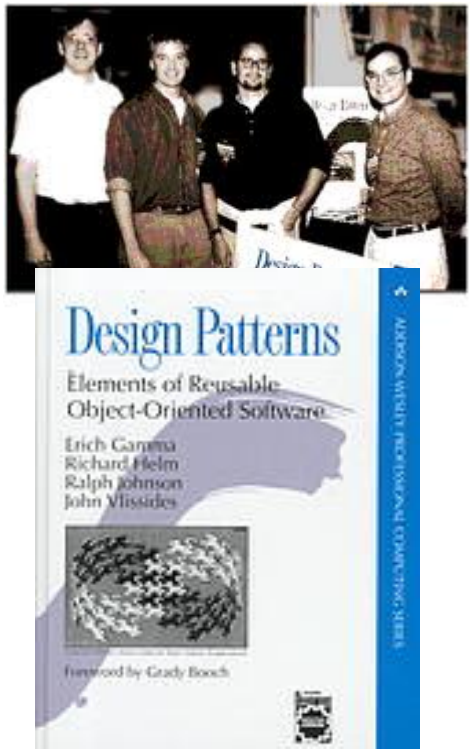
You could come up with these solutions on your own.

- You shouldn't have to!

A design pattern is a known solution to a known problem.

- A concise description of a successful “pro-tip”.

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.
- Done for object-oriented programming.
  - Some patterns more general; others compensate for OOP shortcomings.
  - But any “paradigm” should have design patterns.

An example of a GoF pattern

For some class `C`, guarantee that at run-time there is exactly one instance of `C`.

- And that the instance is globally visible.

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…
- Have an object with fields that are “like public, static fields” but you can have a constructor decide their values.
- Make it easier to ensure some key invariants.
  - There is only one instance, so never mutate the wrong one.
- Make it easier to control when that single instance is created.
  - If expensive, delay until needed and then don’t do it again.

How: multiple approaches

```
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

```
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.

- Always return a fresh new object, never re-use one!

Factories:

- Patterns for code that you call to get new objects other than constructors.
- Examples: Factory method, Factory object, Prototype and Dependency injection.

Sharing:

- Patterns for reusing objects (to save space, for example).
- Examples: Singleton, Interning, Flyweight.

Motivation for factories

Supertypes support multiple implementations:

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

Clients use the supertype (**`Matrix`**) but still need to use a **`SparseMatrix`** or **`DenseMatrix`** constructor.

- Must decide concrete implementation somewhere.
- Don't want to change code to use a different constructor.
- 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.

Advantages:

- To switch the implementation, change only *one* place.
- **`createMatrix`** can do arbitrary computations to decide what kind of matrix to make (unlike what's shown above).

`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!

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

Date today = new Date();

String s1 = df1.format(today);// "Jul 1, 1990"
String s2 = df2.format(today);// "10:15:00 AM"
String s3 = df3.format(today);// "jeudi 1 juillet 1790"
```

Example: Bicycle race

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

- No factories yet!
- Objective: factories for the bicycles to get flexibility and code reuse…

Example: Tour de France

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

The problem: We are reimplementing the constructor in every **`Race`** subclass just to use a different subclass of **`Bicycle`**.

Example: Cyclocross

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

The problem: We are reimplementing 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();
    ...
  }
}
```

Use a factory method to avoid dependence on specific new kind of bicycle in the constructor.

- Call the factory method instead!

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(); }
}
```

- Foresight to use factory method in superclass constructor.
- Then dynamic dispatch to call overridden method.
- Subtyping in the overriding methods (covariant returns type is ok).

Next step

- But **`createBicycle`** was just a factory method!

- Now let's move the method into a separate class.
  - So it's part of a factory object…

- Advantages:
  1. Can group related factory methods together.
     - Not shown: **`repairBicycle`**, **`createSpareWheel`**, …
  2. Can pass factories around as objects for flexibility.
     - Choose a factory at runtime.
     - Use different factories in different objects (e.g., races).

Factory **objects/classes**

Factory objects/classes encapsulate factor methods…

```
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();
  }
}
```

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.

The subclasses

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

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

- 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())`**.

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

Prototype pattern

- Every object is itself a factory.
- Each class contains a **`clone`** method that creates a copy of the receiver object:

```
class Bicycle {
  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)

Using prototypes

```
class Race {
  Bicycle bproto;

  public Race(Bicycle bproto) {
    this.bproto = bproto;
    Bicycle bike1 = (Bicycle) bproto.clone();
    Bicycle bike2 = (Bicycle) bproto.clone();
    ...
  }
}
```

Again, we can specify the race and the bicycle separately:

```
new Race(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
- External file is essential part of program

Factories: summary

Problem: 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:

- Bundles factory methods for a family of types.
- Can store object in fields, pass to constructors, etc.

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.

Sharing

Recall the second weakness of Java constructors:

Java constructors always return a new 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.

Interning pattern

- Reuse existing objects instead of creating new ones:
  - Less space.
  - May compare with `==` instead of **`equals()`**.

- How does the mechanism work?
  - Maintain a collection of all objects.
  - If an object already appears, return that instead.
  - Java builds this in for strings: **`String.intern()`**.

https://docs.oracle.com/javase/8/docs/api/java/lang/String.html#intern--

Space leaks

- Interning can waste space if your collection:
  - Grows too big.
  - With objects that will never be used again.

- Not discussed here: The solution is to use weak references.
  - This is their canonical purpose!

    https://en.wikipedia.org/wiki/Weak_reference

- Do not reinvent your own way of keeping track of whether an object in the collection is being used:
  - Too error-prone.
  - Gives up key benefits of garbage-collection.

`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 Boolean 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.**

https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html#Boolean-boolean-

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: Independent of object's "context".

- Often same across many objects and immutable.
- Technique: intern it.

Extrinsic state: different for different objects; depends on "context".

- Have clients store it separately, or better:
  - Make it implicit (clients *compute* it instead of represent it).
  - Saves space.

http://www.tutorialspoint.com/design_pattern/flyweight_pattern.htm