

Lecture 13: Design Patterns

October 7, 2002

A design pattern is:

- a standard solution to a common programming problem. Some problems come up over and over in object-oriented programming. A design pattern represents a codified solution to a problem, an idiom that you can apply when you encounter it again.
- a shorthand for communicating design concepts. Design patterns provide a vocabulary for sharing software designs with other programmers, both verbally and in documentation and specifications. Rather than a wordy description like “this class steps through a collection and returns one element at a time”, you can just say “this class is an iterator.”
- a particular shape of object diagram, object model, or module dependency diagram. Many patterns reveal themselves this way, often by reducing the degree of coupling between components to make the design more amenable to change.

The bible for design patterns is the so-called “Gang of Four” book, *Design Patterns: Elements of Reusable Object-Oriented Software*, by Gamma, Helm, Johnson, and Vlissides, which is a recommended course text. Although the book is starting to show its age, with examples only in C++ and Smalltalk, it is still incomparable as a catalog of useful patterns, and deserves a place on your bookshelf.

We’ve already seen one example of a pattern in the poker game in Problem Set 2. Here’s the problem. One way you might represent suits and ranks uses integer constants, e.g.:

```
public static final int CLUBS = 1;
public static final int DIAMONDS = 2;
...
public static final int ACE = 1;
public static final int TWO = 2;
...
```

This approach is not *typesafe* — the compiler doesn’t stop you from using ACE where you should have used CLUBS, since both are simply integers. Subtle bugs can result, particularly if you can’t remember whether the constructor for a playing card should be called as `new Card(ACE, DIAMONDS)` or `new Card(DIAMONDS, ACE)`.

The poker game code solved this problem using the *typesafe enumeration* pattern. (Note that typesafe enumeration is not found in either the Gang of Four book or the course textbook by Liskov, but an excellent chapter about it can be found in the other recommended course text, *Effective Java* by Joshua Bloch.)

Here’s part of the code for `CardSuit` and `CardValue`, showing the typesafe enumeration pattern in action:

```

public class CardSuit {
    public static final CardSuit CLUBS = new CardSuit("club");
    public static final CardSuit DIAMONDS =
        new CardSuit("diamond");
    public static final CardSuit SPADES = new CardSuit("spade");
    public static final CardSuit HEARTS = new CardSuit("heart");

    private String suitName;

    private CardSuit(String suitName) {
        this.name = suitName;
    }

    public String toString () {
        return suitName;
    }
    ...
}

public class CardValue {
    public static final CardValue ACE =
        new CardValue ("Ace", "A");
    public static final CardValue TWO =
        new CardValue ("two", "2");
    ...
}

```

The typesafe enumeration pattern solves the type safety problem by defining a class for each enumeration (e.g., `CardSuit`) and creating a single fixed instance of the class for each value in the enumeration.

In a catalog of patterns, typesafe enumeration might appear as follows:

Typesafe Enumeration

Problem: Represent a small, fixed set of values with type safety.

Solution: Define an immutable class for the enumeration, with a public constant instance for each value of the enumeration, and private constructor to prevent clients from creating new values. Inherit the default implementation of `Object.equals`; equality of reference is OK because only one object exists for each value. Bloch also discusses some other tweaks to the pattern, such as defining an ordering on the enumerated values and allowing the enumeration to be extended with new values by subclassing.

Tradeoffs: Unlike the integer representation for enumerated values, you can't use a typesafe value in a `switch` statement. Also, your enumeration can be either ordered or extensible, but not both; see Bloch for more details.

13.1 When (and when not) to use design patterns

Design patterns can have several kinds of benefits:

- **More safety.** Code using the pattern is more robust, has more error-checking, and is easier to test and debug. The typesafe enumeration pattern is one example of a safety pattern; singleton is another.
- **Better performance.** The pattern improves the runtime performance of a system, often by increasing the degree of sharing among objects so that less storage is used. Examples of performance patterns include interning and flyweight.
- **More flexibility.** The pattern makes the system easier to change by weakening or eliminating dependencies among modules. Many design patterns improve flexibility; examples include factory, observer, and mediator.

Safety patterns like typesafe enumeration are almost always a good idea, because a safer program is faster to develop, requires less testing, and is generally more reliable. For performance or flexibility patterns, however, the first rule of design patterns is the same as the first rule of optimization: *delay*. Just as you shouldn't optimize prematurely, don't use design patterns prematurely. It may be best to first implement something and ensure that it works, then use the design pattern to improve weaknesses. This is especially true if you do not yet grasp all the details of the design. On the other hand, if you fully understand the domain and problem, it may make sense to use design patterns from the start, just as it would make sense to use a more efficient rather than less efficient algorithm from the start.

Design patterns may increase or decrease the understandability of a design or implementation. They can decrease understandability by adding indirection or increasing the amount of code. They can increase understandability by improving modularity, better separating concerns, and easing description.

Most people use design patterns when they notice a problem with their design (a change that ought to be easy but isn't) or implementation (performance). Examine the offending design or code. What are its problems, and what compromises does it make? What would you like to do that is presently too hard? Then check a design pattern reference. Look for patterns that address the issues you are concerned with.

Design patterns may seem abstract at first, or you may not be convinced that they address a significant problem. You will come to appreciate them as you build and modify larger systems — perhaps during your work on the final project.

13.2 Creational Patterns

Our first set of patterns concern the creation of objects. Suppose you are writing a class to represent a car race. Before starting, the race has to construct many cars:

```
public class Race {
    public Race () {
        car1 = new Car ();
        car2 = new Car ();
        ...
    }
    ...
}
```

You may want to specialize `Race` for other kinds of races using different kinds of cars:

```

public class Indy500 extends Race {
    public Indy500 () {
        car1 = new Formula1Car ();
        car2 = new Formula1Car ();
        ...
    }
    ...
}

public class Daytona500 extends Race {
    public Daytona500 () {
        car1 = new StockCar ();
        car2 = new StockCar ();
        ...
    }
    ...
}

```

The repeated code is tedious – we aren't able to reuse the car-creation code at all. There must be a better way. Creational design patterns provide an answer.

13.2.1 Factory Method

A factory method is a method that manufactures objects of a particular type. We can add a factory method for cars to the Race class:

```

public class Race {
    public Race () {
        car1 = makeCar ();
        car2 = makeCar ();
        ...
    }
    public Car makeCar () {
        return new Car ();
    }
    ...
}

```

Now subclasses can reuse the Race constructor without change, by overriding the factory method makeCar to produce cars of the appropriate type for the race:

```

public class Indy500 extends Race {
    public Car makeCar () {
        return new Formula1Car ();
    }
    ...
}

```

```

}

public class Daytona500 extends Race {
    public Car makeCar () {
        return new StockCar ();
    }
    ...
}

```

In Java, *static* factory methods are also useful, even though they can't be overridden by a subclass. Unlike a constructor, which always returns an object of the constructor's class, a static factory method can make a runtime decision about what class of object it should return, which can be any subclass of its declared return type. For example, Race might have a static factory method that creates a race appropriate to today's date:

```

public class Race {
    public static Race makeRaceForDate (Date date) {
        if (date.getMonth () == FEBRUARY)
            return new Daytona500 ();
        else if (date.getMonth () == MAY)
            return new Indy500 ();
        else
            return new Race ();
    }
    ...
}

```

Static factory methods fix the first *weakness of Java constructors*: a constructor can only return an object of the constructor's class. It can't return an object belonging to a subclass, even though that would be type-correct. A static factory method has no such restriction.

Static factory methods are common in Java programming. The `java.util.Collections` class includes quite a few examples.

13.2.2 Factory Object

If there are many kinds of object to construct, including the factory methods in each class can bloat the code and make it hard to change. Changing the class created by a factory method requires overriding the entire class, which is overkill if all you want to do is substitute a different kind of car, like Porsche instead of Formula 1.

A *factory object* is an object that encapsulates factory methods:

```

public interface CarFactory {
    public Car makeCar ();
}

```

We might create a different factory for each kind of car:

```

public class Formula1CarFactory implements CarFactory {
    public Car makeCar () {
        return new Formula1Car ();
    }
}

public class StockCarFactory implements CarFactory {
    public Car makeCar () {
        return new StockCar ();
    }
}

```

Then Race uses the factory object that we pass to it:

```

public class Race {
    public Race (CarFactory factory) {
        car1 = factory.makeCar ();
        car2 = factory.makeCar ();
        ...
    }
}

```

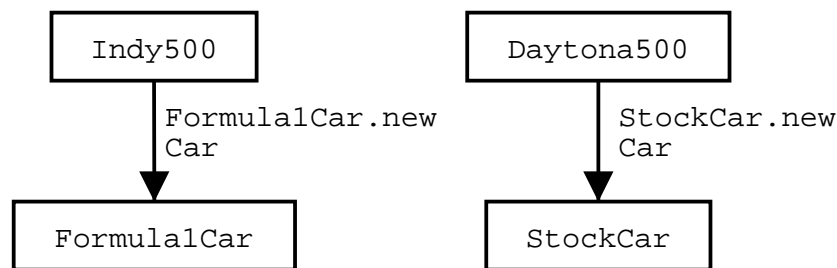
As a result, we can now create different kinds of races with different kinds of cars:

```

race1 = new Indy500 (new Formula1CarFactory ());
race2 = new Indy500 (new MonsterTruckFactory ());

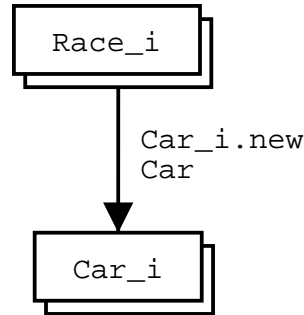
```

The factory object decouples the type of race from the type of car. This effect is easiest to see with module dependency diagrams, using the diagram syntax introduced in Lecture 9. The factory method pattern produces an MDD like this:

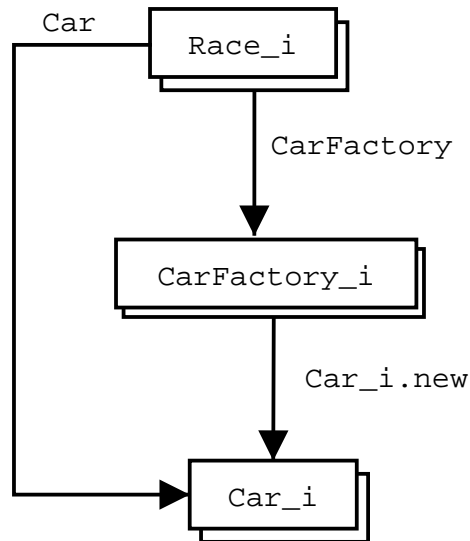


The labels on the arrows denote the assumptions that each race makes about the cars it uses. The factory method depends on calling a specific constructor (e.g. `StockCar.new`), but the rest of the race interacts with the car simply through the generic `Car` interface.

For convenience, we can combine these diagrams into a single MDD that collapses races and cars into stacked boxes representing a family of related classes:



The assumption `Car_i.new` indicates an undesirable coupling between races and cars. If we eliminated it, then races and cars could be changed independently. Introducing a factory object permits this:



13.2.3 Prototype

The prototype pattern provides another way to construct objects of arbitrary types. Rather than a `CarFactory` object, a `Car` object is passed in, as a model or *prototype* of the cars that should be used. New cars are created by punching out copies of the prototype, using either `clone()` or a class-specific copy method. The latter technique is used below:

```

public class Race {
    public Race (Car prototype) {
        car1 = prototype.copy ();
        car2 = prototype.copy ();
        ...
    }
}

public class Car {
    public Car copy () { ... }
    ...
}
  
```

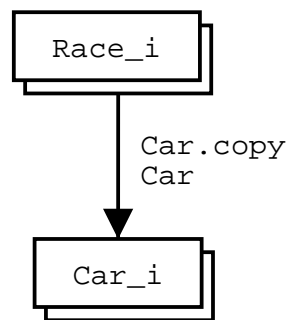
Effectively, each object is itself a factory specialized to making objects like itself.

Prototypes are particularly useful for avoiding subclassing, because you can create new “classes” at runtime by configuring different prototypes:

```
Car prototype1 = new MonsterTruck ();
prototype1.replaceTires (new BalloonTire ());
```

```
Car prototype2 = new Formula1Car ();
prototype2.repaint (Color.GREEN);
prototype2.removeRearSpoiler ();
```

The MDD for the prototype pattern is similar to factory method, but races have been decoupled from cars by having Race call a generic copy method, available on all cars, rather than a specific constructor:



There is no free lunch: the code to create objects of a particular class must go somewhere. Factory methods put the code in methods in the client. Factory objects put the code in methods of a factory object. Prototypes put the code in copy methods on the object itself.

13.3 Sharing Patterns

Several other patterns are related to object creation, in particular avoiding the creation of redundant or unnecessary objects by reusing (sharing) existing objects.

13.3.1 Singleton

The singleton pattern guarantees that only one object of a particular class ever exists. Consider the scheduler in a car-racing simulator, which is in charge of animating cars around the track. The scheduler class might look something like this:

```
class Animator {
    public void addCar (Car car) { ... }
    public void removeCar (Car car) { ... }
    public void timeStep () { ... }
    ...
}
```

Only one instance of the scheduler should exist; a program that instantiated multiple copies of the scheduler would have serious bugs. The singleton pattern prevents this by taking control over the creation of the scheduler object:

```
// private constructor, so clients can't create
private Animator () { ... }

// static reference to the one true Animator
private static Animator theInstance;

// factory method that returns the Animator
public static Animator getInstance () {
    if (theInstance == null)
        theInstance = new Animator ();
    return theInstance;
}
```

The singleton pattern fixes a second weakness of Java constructors: a constructor always returns a new object. The static factory method `getInstance` only ever creates one instance, and always returns a reference to that instance. Client code accesses the singleton by calling `getInstance`:

```
Animator animator = Animator.getInstance ();
animator.addCar (car1);
animator.addCar (car2);
...
```

In this case, the static factory method creates the singleton *lazily*; it delays construction of the singleton until the first time `getInstance` is called. Lazy construction is useful when the singleton is large or expensive to create, or when its creation depends on information that is not available until later than initialization time. If lazy construction is not necessary, the singleton can be created at initialization time and stored in a public static final variable:

```
public static final Animator theInstance = new Animator ();
```

The typesafe enumeration pattern uses this approach to create a singleton object for each value in the enumerated type. Clients can then access the instance variable directly (but they can't change it, because the variable is marked final):

```
Animator.theInstance.addCar (car1);
Animator.theInstance.addCar (car2);
```

The disadvantage of obtaining the singleton from a public variable rather than a public method is that you lose flexibility. If you used the public variable approach and you later need to create the singleton lazily, you would have to change all your clients to call a method instead.

Here's another question: since only a single `Animator` can exist, why not just use a collection of static procedures, like this:

```

class Animator {
    public static void addCar (Car car) { ... }
    public static void removeCar (Car car) { ... }
    public static void timeStep () { ... }
    ...
}

```

In this scenario, the scheduler's data would be stored in static fields, rather than instance fields, and clients would call the static procedures directly:

```

Animator.addCar (car1);
Animator.addCar (car2);

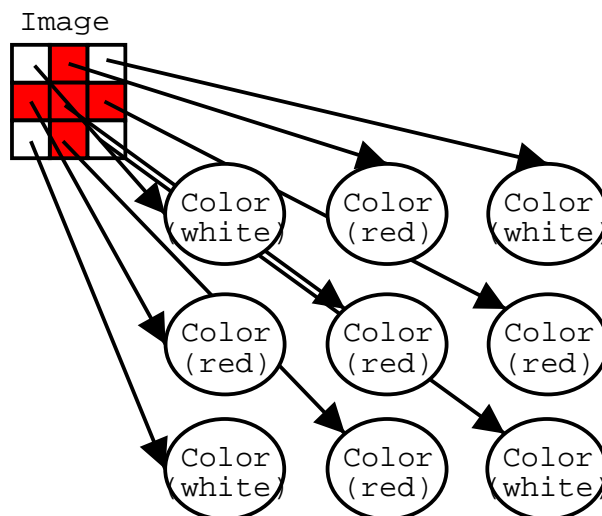
```

This approach has several disadvantages relative to the singleton pattern, all related to flexibility. First, at some point in the future you may *want* to have multiple schedulers in the system. If Animator is already implemented as a class with instance methods, it will be easier to make this change. Second, you may want to have different kinds of schedulers with different properties, or tuned to different hardware platforms. If Animator is implemented as a class, you can write subclasses that override its methods, and have `getInstance` decide at runtime which subclass to instantiate as the singleton. Static methods cannot be overridden, so the collection-of-procedures approach would be less amenable to this kind of change.

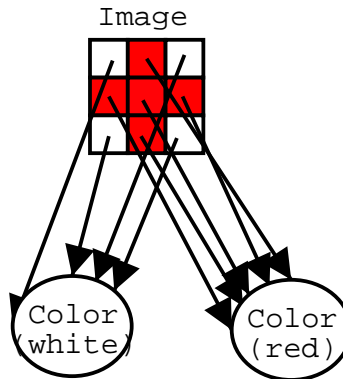
13.3.2 Interning

The interning design pattern reuses existing objects rather than creating new ones. If a client requests an object that is equal to one that already exists, then the preexisting one is returned instead. Interning is generally useful only for immutable objects.

For example, an image might be represented as an array of pixels, each of which is a Color. A large image may have many pixels in it, but only a few distinct Colors:



Representing every pixel with a unique Color object is unnecessarily wasteful of space. A better way would be to share the Colors:



Interning arranges for objects that are immutable to be reused. Interning requires a table of all the objects (of the given type) that have ever been created. If the table contains an object that is equivalent to the desired object, the table's object (the *interned* object) is returned instead. Otherwise, the desired object is created and stored in the table for future reference.

Here is a factory method that returns an interned Color, given its red, green, and blue values:

```
static Map map = new HashMap ();

public static Color fromRGB (float r, float g, float b) {
    Color color = new Color (r, g, b);
    if (map.containsKey (color)) {
        return (Color) map.get (color);
    } else {
        map.put (color, color);
        return color;
    }
}
```

The interning pattern necessarily uses a static factory method. We wouldn't be able to return an interned object from a Java constructor, because a constructor always returns a freshly minted object.

In this case, the table maps each interned Color to itself, so the method creates a trial Color object before checking whether the color has already been interned. When the interned objects are expensive to create, however, it may make more sense for the interning table to map the *content* of the object (in this case, its RGB value) to the interned object, in order to save the cost of construction when the object already exists.

Note that this code uses a Map rather than a Set, even though all we're really doing is storing a set of interned Colors. The reason is that Sets do not have a get operation, only contains. In other words, we'd be able to test whether the set of interned objects contains the color we want, but we'd be unable to actually obtain a reference to the interned object.

Interning for strings is so important that it is built into Java. `String.intern` returns an interned version of a string.

The Liskov text discusses interning in section 15.2.1, but calls the pattern "flyweight," which is different from standard terminology in the field.

13.3.3 Flyweight

The flyweight pattern is a generalization of interning. Interning is applicable only when an object is completely immutable and all of its state can be shared by many instances of the object. The more general flyweight pattern can be used when most (but not necessarily all) of an object is immutable, or when instances of the object share most but not all of their state.

Suppose you are modelling a tree, with a trunk, branches, twigs, and leaves. We'll focus on the twigs and the leaves:

```
class Twig {
    Leaf[] leaves;
    ...
}

class Leaf {
    Color color;
    Shape shape;
    Twig parent;
    int distance; // along twig
    int angle;   // around twig
    ...
}
```

The leaves in a tree are mostly identical. They differ only in their *context*, i.e., the twig to which the leaf is attached, and where it grows out of that twig. Properties inherent to the leaf make up its *intrinsic state*, in this case color and shape. Properties of the leaf's context are its *extrinsic state*; here, its parent, distance, and angle fields.

Although leaves seem like natural candidates for the interning pattern, because there are many of them and they are intrinsically identical, the presence of the extrinsic state in `Leaf` blocks us from using interning, because every `Leaf` object must have a different location.

The flyweight pattern solves this problem by removing the extrinsic state from the leaf object itself and relying on the leaf's context to supply it when needed. We'll define a new class `Leaflet` to represent just the intrinsic state of a leaf:

```
class Leaflet {
    Color color;
    Shape shape;
    ...
}
```

We can readily apply interning to `Leaflet`; all the leaves of one tree, even a whole forest of a single species, can be represented by a single object. A richer model of a tree might have different `Leaflets` in different stages of growth or under attack by caterpillars or disease, but this can be easily accommodated while still obtaining substantial sharing from interning.

When a `Leaflet` needs to access its extrinsic state, for instance to draw itself on the screen, the state must be passed in. For example, the twig might tell its leaves to draw themselves by computing their locations on the fly:

```

class Twig {
    Leaflet[] leaves;

    void draw () {
        for (int i = 0; i < leaves.length; ++i) {
            int distance = i * distanceIncrement;
            int angle = (i * angleIncrement) % 360;
            leaves[i].draw (this, distance, angle);
        }
    }
}

class Leaflet {
    Color color;
    Shape shape;

    void draw (Twig parent, int distance, int angle) {
        ...
    }
}

```

The main disadvantage of the flyweight pattern is that a reference to a `Leaflet` is no longer sufficient to identify a specific leaf and tell you everything you need to know about it. You have to know its context as well, namely the twig and the leaf's index on that twig, in order to access its extrinsic state. Often, however, flyweight can be used "under the covers" to improve the performance of a system, while from a client's point of view objects are still heavyweight. For example, if a client requests a reference to a specific leaf on a twig, `Twig` might return a full-fledged (but temporary) `Leaf` that wraps both the `Leaflet` and its context. More will be said about wrappers in a later design pattern lecture.

Remember that flyweight should only be considered after profiling has determined that space usage is a critical bottleneck in the program. Introducing such constructs into programs complicates them and presents many opportunities for error. It should be undertaken only in limited circumstances.