What is a Design Pattern?

A design pattern is a general solution to a commonly encountered problem in object-oriented design. Here’s an analogy to bend your brain...

The Pattern of Myths: "Hero with a thousand faces" In his 1948 book, "Hero with a thousand faces" Joseph Campbell discusses how myths from various cultures share a common structure: the **monomyth**:

"A hero ventures forth from the world of common day into a region of supernatural wonder: fabulous forces are there encountered and a decisive victory is won: the hero comes back from this mysterious adventure with the power to bestow boons on his fellow man."

Important roles are filled by archetypes: characters that adhere to particular patterns:

- The Hero (e.g. Frodo, Luke Skywalker)
- Shadows (e.g. Sauron, Darth Vader)
- Mentors (e.g. Gandalf, Yoda)
- etc...
What is a Design Pattern?

- A design pattern is like the monomyth. It consists of a set of classes with particular interactions. These classes fill certain roles (archetypes).
- Popularized by the following book from the "Gang of Four": Design Patterns, Elements of Reusable Object-Oriented Software, by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley Professional, 1995
- You have probably employed some design patterns already, without even knowing it.
- Recognizing your existing use of a pattern helps to document it and adds clarity to your design.
- Introducing patterns into your design will help to alleviate design smells and adhere to the design principles.

### Iterator

**Intent:** Provide a way to access the elements of an aggregate object [e.g. a list] sequentially without exposing its underlying representation.

You already know this pattern (from 4892)! Consider the code for a list of int's that automatically resizes...

```java
public class IntVect {
    private int sz;
    private int[] vect;
    public IntVect(int capacity) {
        this.vect = new int[capacity];
        this.sz = 0;
    }
    public int size() { return sz; }
    public void add(int e) {
        if (sz >= vect.length) {
            int[] t = new int[2*sz];
            for(int i = 0; i < vect.length; i++) {
                t[i] = vect[i];
            }
            vect = t;
        }
        vect[sz] = e;
        sz++;
    }
    public int get(int index) { return vect[index]; }
    public void set(int index, int e) { vect[index] = e; }
}

public class IntVectIter {
    private IntVect intVect;
    public IntVectIter(IntVect intVect) {
        this.intVect = intVect;
        this.next = 0;
    }
    public boolean hasMore() {
        if (next < intVect.size()) return true;
        else return false;
    }
    public int nextElement() {
        if (next >= intVect.size()) { throw new RuntimeException("no more elements"); }
        int v = intVect.get(next);
        next++;
        return v;
    }
}
```
Using the Iterator

```java
public class TestIntVect {
    public static void main(String[] args) {
        IntVect vec = new IntVect(5);
        vec.add(10);
        vec.add(20);
        vec.add(30);
        vec.add(40);
        assert vec.size() == 4;

        // Create the iterator.
        IntVectIter iterator = new IntVectIter(vec);

        // Iterate!
        while (iterator.hasMore()) {
            int value = iterator.nextElement();
            System.out.println("value : " + value);
        }
    }
}
```

What is a Design Pattern?

- Iterator
- Strategy
- Factory
- Singleton
- Facade
- Composite

Aside: Container Classes (i.e. Data Structures)

For general purpose containers, use classes from the Java Collections Framework. For an automatically resizing array use `ArrayList`. Better yet, use `ArrayList<Type>`:

```java
import java.util.ArrayList;
public class TestGenerics {
    public static void main(String[] args) {
        ArrayList<Integer> vec = new ArrayList<>();
        vec.add(10);
        vec.add(20);
        vec.add(new String("asdf"));
    }
}
```

If you don’t use the Generics feature (i.e. `ArrayList` instead of `ArrayList<Type>`) then you lose the compile-time type check for what goes in the ArrayList.

Different data structures will require different Iterator implementations. Therefore, we can apply the DIP and abstract out the type of iterator required.

- Notice here that the type of List data structure is also made abstract. The Factory pattern is used here in `CreateIterator` (we will cover this pattern soon)
- This figure is from [1] which came out prior to UML. Therefore, the notation is slightly different.

Here is the general structure of Iterator from [1]:

- **Aggregate**: Creates `Iterator`
- **Client**: Can use `Iterator`
- **ConcreteAggregate**: `CreateIterator`
- **ConcreteIterator**: Can implement `Iterator`

This figure is from [1] which came out prior to UML. Therefore, the notation is slightly different.
What is a Design Pattern?

Iterator
Strategy
Factory
Singleton
Facade
Composite

Strategy

This is another pattern you've already seen. Recall in assignment 1 that the AI interface was implemented by two different concrete strategies:

- **Iterator**
- **Strategy**
- **Factory**
- **Singleton**
- **Facade**
- **Composite**

The idea of the Strategy pattern is to define a set of interchangeable algorithms. The algorithms can change but those changes are insulated from the client code.

Here is the general structure of Strategy:

![Strategy UML Diagram]

**Participants:**
- **Strategy**: Declares common interface for this family of algorithms
- **ConcreteStrategyX**: Implements one algorithm
- **Context**: Creates a ConcreteStrategyX but only refers to it as a Strategy (see next slide)

UML Diagrams Lie!

The UML diagram previously shown for the AI interface suggests that Game knows nothing about SmartAI or DumbAI. Yet here is a segment of the code for Game:

```java
public class Game {
    private AI ai; // ...

    public Game(boolean playerIsX, boolean challenging) {
        // ...
        if (challenging)
            ai = new SmartAI(!playerIsX);
        else
            ai = new DumbAI(!playerIsX);
    }

    public void aiPlacePiece() {
        // ...
        board = new Board(board, ai.chooseMove(board));
    }
}
```

Game creates either a SmartAI or a DumbAI. So Game actually does have a dependency on these classes not shown in the UML!

These previously unshown dependency arrows show that the design violates the DIP! But...

- The dependency is restricted to one small section in the constructor where either SmartAI or DumbAI is created
- Afterwards, we refer to the AI functionality only through the AI interface
- So its a white lie... a small omission of information—If we added the dependencies above it would be more accurate, but would also impair the clarity of the design
What is a Design Pattern? Iterator Strategy Factory Singleton Facade Composite

- This pattern was dubbed "Factory Method" by [1]. However, following [2] we will just call it "Factory".

- In the example above the creation of a SmartAI or DumbAI had to be made by Game.

- Can this creation logic be separated out from the client class?

- Yes. The Factory pattern allows us to create concrete objects through an abstract “factory” interface.

Define a ShapeFactory interface and an underlying implementation to do the actual creation. SomeApp now just calls makeSquare or makeCircle.

```
interface ShapeFactory {
    Shape makeCircle();
    Shape makeSquare();
}
```

```
public class ShapeFactoryImplementation implements ShapeFactory {
    public Shape makeCircle() {
        return new Circle();
    }
    public Shape makeSquare() {
        return new Square();
    }
}
```

Figure 21-1 An app that violates the DIP to create concrete classes

This example is very similar to the tic-tac-toe case. SomeApp will refer to the Shapes it creates only though the Shape interface, but at some point it has to create concrete instances of Shape.
Having individual `makeSquare`, `makeCircle` in the ShapeFactory class means that we still have a sort of dependency between SomeApp and the concrete Shape classes. To correct this we can refactor ShapeFactory to have only one `make` method:

```java
interface ShapeFactory2 {
    Shape make(String shapeName) throws Exception;
}

public class ShapeFactoryImplementation2 implements ShapeFactory2 {
    public Shape make(String shapeName) throws Exception {
        if (shapeName.equals("Circle"))
            return new Circle();
        else if (shapeName.equals("Square"))
            return new Square();
        else
            throw new Exception("Cannot create " + shapeName);
    }
}
```

### Singleton

The Singleton class is suitable when we need exactly one object, with global access. Examples:

- An OS has only one file system
- A ship has only one Captain
- A program may have only one configuration file
- A running Java application has only one run-time environment

```java
public class Singleton {
    private static Singleton theSingleton = null;
    // Called only within this class!
    private Singleton() {
    }

    public static Singleton getSoleInstance() {
        if (theSingleton == null) {
            theSingleton = new Singleton();
        }
        return theSingleton;
    }
}
```

### Example from the Java API

```java
import java.util.Calendar;
import java.util.Locale;

public class PrintDate {
    public static void main(String[] args) {
        // get a calendar based on the local environment
        Calendar cal = Calendar.getInstance();
        System.out.println(cal.getTime());
        System.out.println("First weekday: " + cal.getFirstDayOfWeek());

        // get a calendar for the french environment
        Calendar frCal = Calendar.getInstance(Locale.FRENCH);
        System.out.println("First weekday: " + frCal.getFirstDayOfWeek());
    }
}
```

In North America the first day of the week is Sunday, but in France it is Monday.
What is a Design Pattern?
- Iterator
- Strategy
- Factory
- Singleton
- Facade
- Composite

Here is the UML diagram for Singleton:

```
//theSingleton
Singleton
+ getSingleton()
```

Remember that the filled diamond indicates composition. The lifetime of the Singleton and itself are the same (D-uh!).

---

Example from the Java API: The Runtime class:
```java
class RuntimeDemo {
    public static void main ( String[] args ) {
        // Get the singleton!
        Runtime rt = Runtime.getRuntime();
        // System.out.printf("No. of processors %d\n", rt.availableProcessors());
        // run the garbage collector
        rt.gc();
        // Add a bit of code to run on shutdown
        rt.addShutdownHook( new Thread() {
            public void run() {
                System.out.println("Shutting down");
            }
        });
        try {
            while( (char)System.in.read() != 'q' ) {
                // loop until user enters 'q'
            }
        } catch ( java.io.IOException ex ) {} 
    }
}
```

Now for a more useful example (Runtime should be used sparingly!). The java.util.logging package allows logging messages to be stored from your programs. The main class, Logger is not a true Singleton since you can create more than one, but it is quite similar.

```
import java.io.*;
import java.util.logging.*;
public class LoggingTest {
    public static void main ( String[] args ) {
        Logger logger = Logger.getLogger("mylogger");
        try {
            logger.setLevel(Level.FINE);
            FileHandler handler = new FileHandler("log.xml", true);
            logger.addHandler(handler);
            try {
                logger.setLevel(Level.FINE);
            } catch (SecurityException e) {
                System.err.println("Could not create log file" + e);
                System.exit(1);
            }
        } catch (IOException e) {
            System.err.println("Could not create log file" + e);
            System.exit(1);
        }
    }
}
```
// Try the different logging levels
// not all messages will be logged
logger.severe("a severe msg");
logger.warning("a warning msg");
logger.info("a info msg");
logger.config("a config msg");
logger.fine("a fine msg");
logger.finer("a finer msg");
logger.finest("a finest msg");
}
}

What is a Design Pattern?
Iterator Strategy Factory Singleton Facade Composite

Facade

Good object-oriented designs tend to yield more and smaller classes. Yet it may be difficult for clients to understand and use your design. Facade provides a simple interface to a complex subsystem.

System is a Facade because it provides a simplified interface to a large and complex system.

e.g. java.lang.System

The System class in java.lang provides a wide-array of useful fields and methods. Here are a few:

```java
public class SystemExample {
    public static void main(String[] args) throws Exception {
        long time = System.currentTimeMillis();
        System.out.println("Whaddya at world!");
        if ((char)System.in.read() == 'q') {
            System.err.println("You want to quit already!");
            System.exit(0);
        }
        long elapsed = System.currentTimeMillis() - time;
        System.out.println("elapsed (secs): " + elapsed / 1000);
    }
}
```

System is a Facade because it provides a simplified interface to a large and complex system.

Singleton is not without its detractors. One criticism is that it introduces global state into an application. It is possible for two seemingly unrelated classes to communicate through a Singleton. Thus, some have referred to Singleton as an anti-pattern. So use it, but try not to overuse it.

What is a Design Pattern?
Iterator Strategy Factory Singleton Facade Composite
What is a Design Pattern?

Notice that System is not a proper Singleton. Instead it has only static methods and fields. This may seem equivalent but Singleton offers some advantages:

1. You can specify arguments to control the initialization of your Singleton (not possible for an all-static class)
2. You can sub-class a Singleton

These advantages don’t apply to java.lang.System which is a rather special class.

The Composite pattern allows clients to treat individual objects and compositions of objects uniformly. Consider the following example:

```java
interface Shape {
    void draw();
}
import java.util.ArrayList;
public class CompositeShape implements Shape {
    private ArrayList<Shape> shapes =
        new ArrayList<Shape>();
    public void add(Shape s) {
        shapes.add(s);
    }
    public void draw() {
        for (Shape s : shapes)
            s.draw();
    }
}
```

You can call draw on a simple Shape such as Circle or Square, but you can also draw a whole collection of shapes.

e.g. Evaluation Items in Courses

Consider a course such as this one. There are evaluation items such as assignments, tests, and maybe a project. These items have a mark and some weight in the grading scheme. A composition of items (e.g. the composition of all assignments) also has a mark and some weight. The whole course can be considered a composition (with a weight of 100).
interface EvaluationItem {
    double getWeight();
    double getMark();
}

public class Item implements EvaluationItem {
    private String name;
    private double earned, maximum, weight;
    public Item(String name, double earned, double maximum, double weight) {
        this.name = name;
        this.earned = earned;
        this.maximum = maximum;
        this.weight = weight;
    }
    public double getWeight() {
        return weight;
    }
    public double getMark() {
        return weight * earned / maximum;
    }
}

public class TestEvaluation {
    public static void main(String[] args) {
        EvaluationGroup assigns = new EvaluationGroup("Assigns");
        assigns.add(new Item("A1", 60, 100, 10));
        assigns.add(new Item("A2", 70, 100, 10));
        assigns.add(new Item("A3", 80, 100, 10));
        print(assigns);
        EvaluationGroup tests = new EvaluationGroup("Tests");
        tests.add(new Item("Mid-term", 80, 100, 20));
        tests.add(new Item("Final", 80, 100, 50));
        print(tests);
        EvaluationGroup course = new EvaluationGroup("Course");
        course.add(assigns);
        course.add(tests);
        print(course);
    }
    private static void print(EvaluationGroup group) {
        System.out.println(group.getName() + ":	weight : " +
            group.getWeight() + "\tmark : " + group.getMark());
    }
}

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class EvaluationGroup implements EvaluationItem {
    private String name;
    private List<EvaluationItem> items =
        new ArrayList<EvaluationItem>();
    public EvaluationGroup(String n) { name = n; }
    public void add(EvaluationItem item) { items.add(item); }
    public String getName() { return name; }
    public double getWeight() {
        double totalWeight = 0;
        for (EvaluationItem item : items)
            totalWeight += item.getWeight();
        return totalWeight;
    }
    public double getMark() {
        double finalMark = 0;
        for (EvaluationItem item : items)
            finalMark += item.getMark();
        return finalMark;
    }
}

public class Item implements EvaluationItem {
    private String name;
    private double earned, maximum, weight;
    public Item(String name, double earned, double maximum, double weight) {
        this.name = name;
        this.earned = earned;
        this.maximum = maximum;
        this.weight = weight;
    }
    public double getWeight() {
        return weight;
    }
    public double getMark() {
        return weight * earned / maximum;
    }
}

import java.util.ArrayList;