Cohesion and Coupling

Let's consider the boundaries between classes—what functionality should be in one class vs. in another.

We want to maximize *cohesion* and minimize *coupling*.

Outline for Week 11

I. Cohesion & coupling
   A. Maximizing cohesion
   B. Separation of responsibility
   C. Minimizing coupling
   D. The Law of Demeter
II. Creational patterns
   A. Factory Method
   B. Abstract Factory

Maximizing cohesion

The basic guideline of class design is,

> Every class should be responsible for doing one thing only and doing it well.

Readers should be able to understand the behavior of the class without reading the code.

The fact that all the behavior of a class is closely related is called "cohesion."

Another example is a "god" class that controls all the other objects in the program. The objects are reduced to mere data-holders.

A common problem is when code to check for a condition is littered throughout the system, so that to understand a class, the reader needs to read about several unusual conditions. Here is an example from Expertiza.

Separation of responsibility

It is not always clear which class should do what. Sometimes we need to consider the advantages and disadvantages of each assignment of responsibility.

Consider these examples.

Example 1: When an array of objects needs to be sorted, the objects need to be compared to each other.

- Should the objects know how to compare themselves to other objects with a method similar to **`String`**'s **`compareTo(Object)`** method, or
- should a separate object, such as the **`Comparator`**, be responsible for doing the comparing?

```
public class Comparator {
  public int compare(String o1, String o2) {
    return s1.compareTo(s2);
  }
  public int compare(Integer o1, Integer o2) {
    int i1 = o1.intValue();
    int i2 = o2.intValue();
    return i1 - i2;
  }

  ...compare methods for other types of data...
}
```

Submit your answer here.

For example, the **`Array`** class in the **`java.util`** package includes two methods that sort arrays of objects. One method uses the **`compareTo(Object)`** method of each object and the other uses a **`Comparator`** to do the comparing.

Example 2: Consider a **`LinkedList`** implemented from **`Nodes`**.

Each **`Node`** consists of data and a link (**`next`**).

When your program traverses a list, which object is responsible for keeping track of where it is?

- The *client* could keep a reference to the current node, and dereference **`next`** to move to the next node.
- The ***`LinkedList`*** object could keep a reference to the current node

  Then the client would ask the list—
  - for the data in the current node, and
  - to move to the next node (causing the list to update its **`current`** pointer).
- A *third object* could keep track of where the program is in the list.

  Then the client would ask the third object—
  - for the data in the current node, and
  - to move to the next node (causing the 3rd object to update its **`current`** pointer).

Which approach is best? Vote here.

- What, if anything, is wrong with the first one?
- What, if anything, is wrong with the second one?
- What, if anything, is wrong with the third one?

Guideline 1: Different responsibilities should be divided among different objects.

Guideline 2: Encapsulation. One class should be responsible for knowing and maintaining a set of data, even if that data is used by many other classes.

Corollary: Data should be kept in only one place. Cf. database normalization Cf. deadlines this semester (negative example)

One class should be chosen to manipulate a particular type of data.

Other classes must ask this class when they need to use or change the data.

Let's see what happens if this guideline is not followed.

For example, suppose you have an object of class **`Department`** that is responsible for maintaining a collection of **`Employee`** objects, held in an **`ArrayList`**.

Other objects may need to access the **`Employee`** objects.

The **`Department`** would have a **`getEmployees`** method that returns the **`ArrayList`** of **`Employee`** objects.

What is wrong with this approach? ? Clients could get at the ArrayList and corrupt it, e.g., by removing **`Employees`** who should be there or by adding other objects to the **`ArrayList`**, including non-**`Employee`** objects!
How can this risk be avoided?

1. Have the **`getEmployees`** method return an **`ArrayList`** of the Employees, but make it a new **`ArrayList`** that is a shallow clone of the Department's **`ArrayList`**.
2. Assuming that other objects rarely need all the **`Employee`** objects, have a "getter" method that finds and returns an employee specified by particular criteria.
3. Have the **`Department`** class manipulate the **`Employee`** objects. Clients have to ask the **`Department`** for any details regarding **`Employees`** that they need.
4. Replace the **`getEmployees`** method with an **`iterator`** method that returns an **`Iterator`** over the **`ArrayList`**.

The principle of encapsulation states that the **`Department`** should never let other classes see the actual **`ArrayList`**, but only the data in the **`ArrayList`**.

Guideline 3: *Information Expert pattern*. Assign a responsibility to the class that has the data needed to fulfill that responsibility.

"Ask not what you can do to an object; ask what the object can do to itself."

Guidelines 2 and 3 establish that data should not be manipulated in more than one place.

Similarly, code should not be duplicated in more than one place.

Guideline 4: *The DRY principle*. Code should not be duplicated. A given functionality should be implemented only in one place in the system.

Why is this a good guideline?

Minimizing coupling

Classes frequently need to be modified.

They should be written in such a way that changing one class is not likely to break other parts of the code.

Guideline 5: Design your classes so that they can handle change.

The idea is to define your variables and values to have the widest possible type rather than the narrowest type.

The widest possible type in Java is an interface that can be implemented by any number of classes.

Here is an example of an e-mail sender that comes in a high-coupling and low-coupling package.

Look at the difference between the two packages and answer these questions.

Guideline 6: Do not use class methods when instance methods will suffice.

What are some ways that class variables or methods can be used?

Class variables: Counting the # of objects of a class

Defining constants

```
public final static double PI = 3.14159265358979;
```

Providing a value used by all instances of a class (burnTime)
Can't be a constant, since it's read in by the UI

To implement the Singleton pattern (holding the only instance) … or in general, anytime the constructor is made private.

Class methods: Accessor methods for class variables

Methods that operate on elements of primitive classes, e.g., an **`isLeapYear`** method.

```
private static boolean isLeapYear(int year) {
    if (year % 4 != 0) {
        return false;
    }
    if (year % 400 == 0) {
        return true;
    }
    return (year % 100 != 0);
}
```

Math methods are another example: **`sin`**, **`cos`**, **`ceil`**, **`max`**

Methods that need to execute before any object is created, e.g., **`public static void main(…)`**

To perform operations on a group of objects of the same class

If the constructor is made private, but needs to be accessed from outside the class (e.g., the Singleton pattern).

Overuse of class methods increases coupling between classes.

Functionality that should be a method of one class is in another class, where no object is a receiver. This means that some of the responsibilities of the class are actually implemented in other classes.

The Law of Demeter

Long chains of method calls mean there is a large amount of coupling between classes.

Consider this approach to getting a bank balance:

```
Balance balance = atm.getBank(b).getBranch(r).
getCustomer(c).getAccount(a).getBalance();
```

Assume that **`b`**, **`r`**, **`c`**, and **`a`** are all strings.

How many classes does the calling class need to know about?

Another way to handle this would be to code,

```
Balance balance = atm.getBalance(b, r, c, a);
```

Now only the ATM class, not the caller, needs to worry about the existence of the branch, the customer account, etc.

The Law of Demeter says that a class should only send messages to

1. this object itself
2. this object's instance variables
3. the method's parameters
4. any object the method creates
5. any object returned by a call to one of this object's methods
6. the objects in any collection that falls into these categories

It should *not* send messages to objects that are returned by calls to other objects.

This is also a good organizational principle. Consider what used to happens when I wanted to retrieve or send back homework to off-campus students. I just phoned Eva Boyce and she took care of it …

Here is an exercise on the Law of Demeter.

Creational patterns

Factory Method design pattern

Factory Method is a *creational* pattern—that is, a pattern that is used for creating objects.

As we know, in o-o languages, objects can be created with some kind of **`new`** method.

Calling **`new`** is fine if the calling code knows what kind of object it wants to create. But a lot of times it doesn't.

Suppose, for example, that the code is copying a diagram, which consists of **`Shape`**s. If it gets in a loop and copies the shapes one by one, you'd need a big **`if`** statement to decide which kind of object to create.

Well, better to encapsulate that logic in a creation method, rather than to expose it at the call site.

Then the client only needs to get in a loop calling the **`getShape()`** method, and each time, the right kind of object will be created.

The method that creates the object could even be a **`static`** method of the class that returns an instance of that class. This has two advantages over using constructors:

1. The "new" object might in fact be a reused object that was previously created (think "buffer pool").
2. The object that is created might actually be a subclass object.

In any case, the client "is totally decoupled" from the code that creates the object.

Choose one of the following two exercises for Factory Method.

- An example on creating wifi or Bluetooth streaming connections.
- An example that creates postcodes for the US, India, and the UK.

The Abstract Factory design pattern

The Factory (or Factory Method) pattern is good for creating single objects of a specific type, as long as those objects don't have to "match" any other objects.

But a lot of times we do need objects to "match."

- User interfaces need all of their widgets to have the same "look and feel."
- Web (or print) pages have a certain style, and all the elements on the page need to match that style.
- A language run-time environment needs to make the appropriate calls to the operating system it is running on.

In all these cases, instantiated objects all need to be from the right "family."

My favorite example of this comes from Refactoring Guru. When a shop sells a set of furniture to a buyer, it's important that the set "match."

Let's look again at the entities in the pattern.

- The *client*—the code "using" the pattern—has-an Abstract Factory.
- The Abstract Factory is an interface that contains a method for creating each different "product."
- One or more Concrete Factories implement the Abstract Factory interface.
    - Hence, each concrete factory implements a method for creating each different "product."
- When the various `createProduct` methods of a particular Concrete Factory are invoked, they create objects that "match."
    - Specifically, the objects "match" because they implement the variant defined by the same concrete factory.

Now, to make sure you are following, answer these questions about the furniture example.

- What is a "modern sofa"? An abstract product, concrete product, etc.?
- What is a "chair"?
- What is a "Victorian furniture factory"?
- What does the method `createCoffeeTable` return?

Again, there are two choices of exercise:

- An example that builds applications in the Semantic or Angular framework
- An example that creates phone numbers and postcodes for the US or the UK.

Complete and consistent interfaces

Guideline 1: Give classes a complete interface.

A class should have a full set of methods so that it is as reusable in as many situations as possible. Suppose we have a GUI component that has a **`setSelected()`** method that highlights itself. What other method should it have?

Outline for Week 12

I. Complete & consistent interfaces
II. State pattern
III. Strategy pattern
IV. State vs. strategy
V. Visitor pattern

A **`setUnselected()`** method to remove the highlighting. Even better would be to have one method **`setSelected(boolean b)`** that highlights the component if **`b`** is true and unhighlights it if **`b`** is false.

The component should probably also have an **`isSelected()`** boolean function that tells whether the component is currently highlighted.

Guideline 2: A well-formed class has a consistent interface.

By "consistent," we mean that the methods that do similar things should be laid out similarly.

Suppose a class maintains and manipulates an indexed collection of people. It has

- a method that sets the *i*th person's name and
- a method that sets the *i*th person's age.

The two methods should have similar names and similar arguments in the same order.

Suppose one signature is **`setName(String name, int index)`**. What should the other be? **`setAge(int age, int index)`**. It would be very confusing if the parameters were in a different order in the two methods.

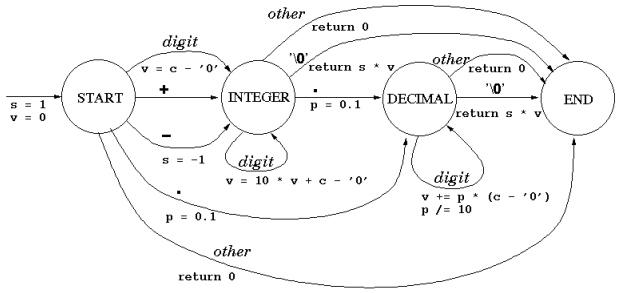
State pattern

We've been talking about bad uses of **case** statements in programs. What is one example? Testing objects' classes; instead use polymorphism.

Another way in which case statements are sometimes used is to implement finite-state machines.

An example: Horner's Rule

A finite-state machine can be used to convert an ASCII string of characters representing a real number to its actual numerical value.



Input symbols are written in **bold** above the transitions,
Symbol classes (such as *digit*) are written in *italic*.
Actions performed by the program are in courier below the transitions.
The variable c denotes the input character.

The letters shown in the FSM stand for the following:

`c` - current character
`s` - sign of the number
`v` - value of the number
`p` - power

Note that this FSM assumes that the string contains a valid floating-point number that

- starts with an optional + or –,
- has at least one digit, an optional decimal point,
- and any number (including 0) of digits before and after the decimal point.

A value of 0 is returned if an invalid string is encountered.

Table form of FSM:

State/Input	+ or –	.	digit	other
START	INTEGER	DECIMAL	INTEGER	ERROR
INTEGER	ERROR	DECIMAL	INTEGER	END
DECIMAL	ERROR	ERROR	DECIMAL	END

Using switch statements, this FSM can be coded as follows:

```
public class Parser {
   static double toDouble(String s) {
      double sign = 1; // sign of number (either 1 or -1)
      double value = 0; // current value of the number
      double power = 0.1; // current power of 10 for
                          // digits after decimal point
      int i = 0;
      final int START = 0;
      final int INTEGER = 1;
      final int DECIMAL = 2;
      final int ERROR = 3;
      int state = START;
      char ch; //current character in string

      while (state != ERROR && i < s.length()) {
         ch = s.charAt(i++);
         switch (state) {
            case START: if (ch == '.')
                  state = DECIMAL;
               else if (ch == '-') {
                  sign = -1.0;
                  state = INTEGER;
               }
               else if (ch == '+')
                  state = INTEGER;
               else if (Character.isDigit(ch)) {
                  value = ch - '0';
                  state = INTEGER;
               }
               else
                  state = ERROR;
               break;
            case INTEGER: if (ch == '.')
                  state = DECIMAL;
               else if (Character.isDigit(ch))
                  value = 10.0 * value + (ch - '0');
               else {
                  value = 0.0;
                  state = ERROR;
               }
               break;
            case DECIMAL: if (Character.isDigit(ch)) {
                  value += power * (ch - '0');
                  power /= 10.0;
               }
               else {
                  value = 0.0;
                  state = ERROR;
               }
               break;
            default: System.out.println("Invalid state: " + state);
         }
      }
      return sign * value;
   }

   public static void main(String[] args) {
      if (args.length == 1)
         System.out.println(toDouble(args[0]));
   }
}
```

This FSM can be represented more elegantly by the State pattern.

How can we code State in a more o-o fashion? *Hint:* We can make **`State`** an interface! Each state will implement this interface.

Horner's Rule: To use the State pattern for Horner's rule, the first step is to define a **`State`** interface. Consider the table form of the FSM.

- The rows of the table represent the different states.

- The columns of the table represent the different *behaviors* of each state.

Therefore, what *methods* should be defined in the **`State`** interface? Well, what do we need to test for each state?

Submit your **`State`** interface here.

```
public interface State {
    void onPoint();
    _____________;
    _____________;
    _____________;
    void onOther();
}
```

How should the **`State`**s be defined?

```
class _______ implements _____ { … }

class _______ implements _____ { … }

class _______ implements _____ { … }
```

In Java, we can define a class within another class. This is called an *inner class*.

Thus, our **`State`**s can be defined as inner classes of **`Parser`**.

Here is the code for the **`Parser`** class, minus its inner classes:

```
public class Parser {
    private final State start = new Start();
    private final State integer = new Integr();
    private final State decimal = new Decimal();
    private State state = start;
    double sign = 1; // sign of number (either 1 or -1)
    double value = 0; // current value of the number
    double power = 0.1; // current power of 10 for
                        // digits after decimal point
    char ch; //current character in string

    double toDouble(String s) {
       int i = 0;
       while (i < s.length()) {
```



```
          ch = s.charAt(i++);
          if (ch == '.') state.onPoint();
          else if (ch == '+') state.onPlus();
          else if (ch == '-') state.onMinus();
          else if (Character.isDigit(ch))state.onDigit();
          else state.onOther();
       }
       return sign * value;
    }

    public static void main(String[] args) {
     System.out.println(new Parser().toDouble("-
914.334"));
    }
}
```

Exercise: Choose one method to implement in all three classes. Submit your code here.

```
        void onMinus();
        void onPlus();
        void onDigit();
        void onOther();
```

If an illegal character is found, throw a **`NumberFormatException`**.

```
package statePattern;
```

Note: Put this code in hidden text in the students' version.

```
interface State {
      void onPoint();
      void onMinus();
      void onPlus();
      void onDigit();
      void onOther();
}


package statePattern;

public class Parser {

      private class Start implements State {
          public void onPoint() {
            state = decimal;
          }
          public void onMinus() {
            sign = -1.0;
            state = integer;
          }
```



```
          public void onPlus() {
            state = integer;
          }

            public void onDigit() {
                  value = ch - '0';
                  state = integer;
            }
            public void onOther() {
                  throw new NumberFormatException();
            }
      }

      private class Integr implements State {
          public void onPoint() {
                  state = decimal;
            }
          public void onMinus() {
                  throw new NumberFormatException();
          }
          public void onPlus() {
                  throw new NumberFormatException();
          }

            public void onDigit() {
                  value = 10* value + (ch - '0');
            }
            public void onOther() {
                  throw new NumberFormatException();
            }
      }

      private class Decimal implements State {
            public void onPoint() {
                  throw new NumberFormatException();
            }
          public void onMinus() {
                  throw new NumberFormatException();
          }
          public void onPlus() {
                  throw new NumberFormatException();
          }

            public void onDigit() {
                  value += power * (ch - '0');
                  power /= 10.0;
            }
            public void onOther() {
                  throw new NumberFormatException();
            }
      }

    private final State start = new Start();
    private final State integer = new Integr();
    private final State decimal = new Decimal();
    private State state = start;
    double sign = 1; // sign of number (either 1 or -1)
```



```
    double value = 0; // current value of the number
      double power = 0.1; // current power of 10 for
                          // digits after decimal point
      char ch; //current character in string

      double toDouble(String s) {
            int i = 0;
            while (i < s.length()) {
                  ch = s.charAt(i++);
                  if (ch == '.') state.onPoint();
                  else if (ch == '+') state.onPlus();
                  else if (ch == '-') state.onMinus();
                  else if (Character.isDigit(ch)) state.onDigit();
                  else state.onOther();
            }
            return sign * value;
      }

      public static void main(String[] args) {
            System.out.println(new Parser().toDouble("-914.334"));
      }
}
```

Strategy pattern

A related pattern is Strategy. This pattern helps when you need to choose an algorithm for a task depending on some "parameter" of the situation.

For example, consider quadrature (numerical integration) again. Each time you calculate the area of a region, you need to know what the function is that you are calculating the region underneath.

Or consider converting different file formats, e.g., .jpeg, .gif, .eps.

You *could* write a case statement whenever you needed to invoke one of the algorithms. Is this a good idea?

Consider extensibility and maintainability. Every time you add another file format, you need to add another case to every place in the program where you select a way of doing an operation (e.g., open, process, close).

But suppose there is only one case statement. Is it OK then? No, even in this case, the logic of which function to use is mixed in with the class that does the quadrature, etc. It is better to separate the

concerns (cf. Factory Method, which encapsulates the case statement in an external method).

Another situation might be where you are manipulating several geometric shapes, e.g., circles, squares, and composites of circles and squares. You need to—

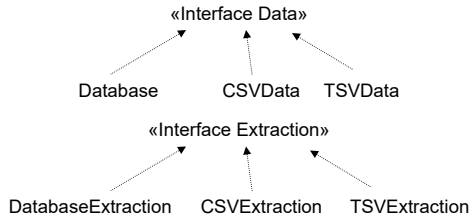
- draw the shapes on a display
- move them to a different location
- rotate them by a certain number of degrees.

These tasks will be performed differently for each shape. You *could* use a case statement everywhere you need to make the decision. But that violates the DRY pattern.

The Strategy pattern allows you to make the decision once when you begin to handle the shapes, and all of the other actions are performed accordingly.

Exercise: Another common situation is when you are working with various kinds of files. You need to open, close, and access them differently depending on the file type.

Our example looks like this.



Fill in the blanks to complete the pattern.

State vs. Strategy

A definition of Strategy (from *Head-First Design Patterns*) is,

> The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

Strategy allows clients to change algorithms at run time by using a different strategy object. This basically lets them appear to change class at run time.

Hmmm … that's interesting. Strategy lets objects appear to change class. Isn't that what State does?

What are the differences between Strategy and State?

- State requires objects to "change state"; that's the point of the pattern. In Strategy,
- The State pattern deals with *how* you switch between different states/implementations, while the Strategy pattern deals with *using* different implementations to implement different algorithms.
- In State, the behavior that's invoked typically depends both on the current state and an input. In Strategy, the behavior depends on the "state," but not on the input in any clear way.
- Martin Fowler: "*We encapsulate each algorithm into a class in strategy pattern, but we encapsulate each state into a class in state pattern*.

The Visitor pattern

Remember our discussion of overloading vs. overriding, from Week 9? At that time, we said,

> In summary, the compiler decides which overloaded method to call by looking at
>
> - the current type of the object being sent the message and
> - the declared types of the arguments to the method call.

> The method is chosen at runtime by dynamic method invocation using the actual value of the object being sent the message.
>
> The actual classes of the arguments to the method call do not play a role.
>
> This is very different from a language like CLOS, which uses the actual types of the arguments to decide which method to execute.

Suppose we *did* want the classes of the arguments to be used to determine which method to call.

My favorite example is "double-dispatching" in arithmetic expressions.

- If you add an integer and a floating-point number, what type should the result be? Floating-point
- Assuming you have a Fraction class, if you add an integer and a fraction, what type should the result be? Fraction
- If you add a floating-point number and a complex number, what type should the result be? Complex

Help answer these questions by filling in this table.

Should *either* the floating-point or complex number be able to be the receiver? Should either be able to be the argument?

So, the method called should depend both on the class of the receiver and the class of the argument. How do we achieve this effect?

Let's say that we implement the Sum method in all numeric classes—Integer, Floating Point, Fraction, and Complex.

So, if we're performing an addition, we invoke the Sum method of the receiver class.

Now, this Sum method knows that what it does actually depends on the class of its argument. How does it achieve this effect? It turns around and sends a message to the argument!

This method, e.g., in the Complex class, is called something like, SumFromFloatingPoint.

- What does it do? It adds a Complex number (originally the argument, now the receiver) to a Floating Point number.
- What does it return? A Complex number

OK, suppose that we have the four numeric classes mentioned above. How many Sum… methods do we need altogether?

4 + 16 = 20. We need a Sum method for each class, and a SumFrom method for each of the 4 classes in each class.

What is the sequence of calls?

- `mySum = myFloat.sum(myComplex)`
  - `return myComplex.sumFromFloatingPoint(myFloat)`

Here is how Visitor is structured.

- Define an interface or abstract class Visitor.
- Visitor contains a **`visit()`** method, which is implemented in each subclass of Visitor. (In our example, these are the **`sumFrom`** methods.)
- These methods are invoked from subclasses of the Element hierarchy. Each one of these classes has an **`accept()`** method, which takes an object of the Visitor hierarchy as a parameter.
- Each descendant of the Element class implements **`accept()`** by calling the **`visit()`** method on the Visitor object it was passed, with **`this`** as the only parameter.
- To perform an operation, the client creates a Visitor object, and calls **`accept()`** on the Element object, passing the Visitor object.

The Visitor pattern can be used to avoid tight coupling, as Bob Martin explains.