COMP 303 Study guide

Francis Piché

December 11, 2018

Contents

Part I
Introduction

1 Disclaimer

These notes are curated from Professor Joseph Vybihal's COMP303 lectures at McGill University. They are for study purposes only. They are not to be used for monetary gain.

2 About This Guide

I make my notes freely available for other students as a way to stay accountable for taking good notes. If you spot anything incorrect or unclear, don't hesitate to contact me via Facebook or e-mail at `http://francispiche.ca/contact/`

Part II
Writing Good Code

Good code is:

- Optimal (Time complexity and memory)
- Simple
- Correct
- Robust (Few crashes, good error handling etc.)
- Easy to read
- Is well documented
- Uses accepted engineering techniques.

3 Strategies for Writing Good Code

3.1 Optimality

To minimize memory usage we could:

- Encode long strings of data (compressing)
- Re-calulate instead of store values (trade-off with speed)
- Check library overhead!

- Data structure overhead (graph has a lot more pointers than a linked list!)

To minimize time complexity:

- Come up with a better algorithm (Improve big O)
- Store more things in RAM (rather than in a DB or over a network)
- Avoid nesting of pointers
- Avoid deep or unnecessary recursion

3.2 Simplicity

Strategies for simplicity:

- Good variable names (avoid single letters except for array index's)
- Don't use lots of variables when an array is more appropriate
- Use simpler data structures if possible
- Reduce the volume of code
- Limit line length
- Modularize the program
- Use algorithms that are well known if possible

3.3 Correctness

The only way to be somewhat sure that your program is correct is through rigorous testing. Some things are easier to test than others, and theres entire QA departments that specialize in testing. But with the rise of Agile, more and more standard Devs need to be good at testing.

Some general ideas on testing:

- 1 test per 1 function
- Test valid inputs
- Test invalid inputs
- Test edge cases
- Validate that the output is correct

There's a ton of material online about testing and I definitely recommend taking some time to learn about more in-depth testing methods than what is covered in this course. It's required in almost all dev jobs and it's a very valuable skill to have.

3.4 Readability

This comes from good indentation, spacing and general style of code. Most people know this implicitly so I won't go into detail. For examples see the lecture slides from Lecture 2.

3.5 Comments as documentation

The first place documentation happens is on the code level. This is where you can remove any ambiguity or sources of confusion from your code for other developers (or even future you). While real projects require real documentation, and excessive comments can be a detriment to readability, more comments are generally better than not enough. Break down your complex algorithms into comment separated steps, or add side-notes on anything that could be confusing to another developer.

4 Well Designed Objects

A well designed object should be one that does not cause any ”wut?” moments amongst a team of developers. Specifically:

- Single class per file
- Single purpose
- Expose only essential information
- Support an API structure
- Follow appropriate inheritance methods

4.1 Single Purpose

A class should be used to represent an idea, concept or object (pun-intended), and nothing more. A student should only contain ideas directly relevant to a student. A `Car` class should not have information about busses or trucks.

This ensures that someone using your class can quickly find out everything there is to know about your object in an intuitive way. If your Student class has information about apples buried inside somewhere, someone on your team would have a hard time finding that out.

4.2 Restriction of Information

This is done to ensure that the class will be used correctly. There should be a decent amount of thought put into which parts of the code are ”internal” and ”external” to the class, ie: what parts of the class does the program need easy access to?

Any time you use the `public` keyword, you should be thinking twice about if it's truly needed.

4.3 Encapsulation

When you take the concepts of single purpose and restriction of information and put them together, you get *encapsulation*. Essentially what this means is that your class should be it's own complete bubble. No code outside the class should do the same things as inside your class, nor should things outside affect the class in an uncontrolled way.

4.4 Inheritance

To avoid duplication of code, we use inheritance to relate objects. This way classes can share common elements and simplify our lives. For details on inheritance see my COMP250 guide.

Note that private information from a parent is not visible t the child.

To design inheritance well:

- A parent is more general
- A child is more specific

4.5 Pre & Post Conditions

To keep things coherent, if a parent object imposes a condition on data, the child should maintain this condition. For example if a parent object has the condition that `salary` $>$ `0` then the child should not violate this by overriding the condition with say, negative values. It could, however, override it with `salary` $>$ `10000`.

This is known as a **rich** starting point. Another example of a rich starting point is using a library, or some sample code.

THis might be an issue since you may be inheriting, importing or implementing more features than are really needed.

4.6 Extending vs Wrapping vs Interfaces

Extending is when a new class contains the parent but adds extra methods and variables. Allows for polymorphism (see 250 guide for more on that).

Interfaces are used when the classes implementing are not necessarily related, but share common methods. For example the `Iterable` interface can be implemented by say, a `Degree` class, which contains a list of courses. But also a `Queue` class which has nothing to do with degrees or courses conceptually. So this allows for polymorphism across multiple inheritance trees.

Wrapping is not a formal construct in a language, but it is the idea of placing objects inside a "wrapper" class, to put objects together in a more abstracted way.

A special case is `abstract base classes`. These cannot be instantiated and contain both implemented and non-implemented methods. They provide the inheritance properties of extending, with the templating of interfaces.

5 Object Identity & Lifecycle

All objects have an idenity,(a reference or anonymous) and a lifecycle. Their lifecycle depends on the language and how they are handled (in C/C++ you have to manually free them from memory, vs garbage collection)

5.1 Referenced Obects

This is when an object is created by assigning its instance to a variable. `Object o = new Object()`. In this way, when no variables remain that point to this object, it is garbage collected (in Java) or causes a memory leak (C/C++).

5.2 Anonymous Objects

These are objects without references. For example `fn(new Object())`. It is then assigned a reference in the scope of the function, or in the scope of the Object to which it was passed into. For example if you pass an object into the constructor of another object, if it is then assigned to a variable in the second object, it will die with the object it is inside of.

5.3 Static vs Instance

A static object (or variable) can exist only once. That is, it cannot be instantiated. There can be no direct communication between static and instance structures.

The `C` version of this would be in the two different ways of creating structs. One where you give your struct a name at the end, vs if you instantiate it afterwards.

6 Object Class and Class Class

All objects in Java extend the Object class. I went in-depth with this in my 250 guide, but I'll summarize here.

There are a few important methods that the Object class has.

- `toString()`
- `equals(Object o)`
- `hashCode()`
- `clone()`

6.1 Object.toString()

Automatically called by many functions such as print statements. Overwrite this to change the string representation of your object. By default, it stores the name of the object, and a unique hexedecimal identifier.

6.2 Object.equals()

Since the `==` operator only checks if the addresses are the same (for reference types), using it with reference types often has strange results. It is therefore necessary to override the `Object.equals()` method to specify what it means for two objects to be equal. Are their attributes equal?(This is the default) Etc. Note that the default will not look at nested Objects, so you must specify an equals method for complicated data structures.

6.3 Object.hashCode()

Hash data structures are only as good as their hash function. So if the default for your data type is really bad, so you'd have to implement your own hash function (override the hashCode() method.)

6.4 Object.clone()

The default clone makes a shallow copy by only cloning the references. (The contents of the original can be changed by the clone!!). A deep copy would be making a clone of all references and make them point to actual independent locations. This can be done with the clone() method.

6.5 Type Inquiry

This is the idea of testing what type a variable is. This is done using the `instanceof` operator. For example: `x isinstance Shape`.

However, it does not check if it is exactly that type, it instead checks if `x` is part of the inheritance tree of Shape. We can use this to check if a cast will be successful.

We can check for exact class matching with the `Object.getClass()` method. It returns an Object containing the type information for a class. This object contains the name, and a pointer to the superclass object. (The class object of it's parent).

The Class class has some other useful methods such as `isArray()` and `getComponentType()` which 1) tests if the class is array, and 2) if it is, get the type of the components of the array.

7 Reflection

This is the practice of a program analyzing itself.

The Class class we looked at is useful for reflection.

We also have:

- Class
- Package
- Field
- Method
- Constructor
- Array

Using these, we can get all the features of a class.

```
Class super = Rectangle.class.getSuper();

Class[] interfaces = Rectangle.class.getInterfaces();

Package p = String.class.getPackage();

Field[] fields = Math.class.getDeclaredFields();
for (Field f: fields){
  if(Modifier.isStatic(f.getModifiers()))
}

Contstructor[] constructors = Rectangle.class.getDeclaredConstructors();

for(Constructor c: constructors){
  Class[] params = c.getParameterTypes(); // can be used on methods too!
Method m = PrintStream.class.getDeclaredMethod("println", String.class);
m.invoke(System.out, "Hello, world!"); //invoke is only for public methods
}
```

Note that we might get some errors with `getDeclaredMethod()` namely, `NoSuchMethodException`, `IllegalAccessException` , and `InvocationTargetException`

We can then use the `setAccessible(bool tf)` method on Fields to set the field to be able to modified. We can then use `set()` and `get()` on the fields.

In this example, we'll resize an array:

```
Object newA = Array.newInstance(a.getClass().getComponantType(),
    2*Array.getLength(a) + 1);

System.arraycopy(a, 0, newA, 0, Array.getLength(a));

a = newA;
```

8 Generic Types

A generic type is instantiated when an actual type is substituted for the type-variable.

```
public class ArrayList<E>{
   public E get(int i){...}
}
```

If a type is used, it must always be the same for the same type variable. So if you give `E` a String, all references to `E` must be with a String.

We can take this further, with extending. For example:

```
public static <E, F extends E> void append(ArrayList<E> a, ArrayList<F> c){...}
```

So this would only allow types E and F if F extends E.

Going *even further* we can use wildcards like this:

```
public static <E> void append(ArrayList<? super F> a, ArrayList<F> b, int
    count){...}
```

To say that anything which is a parent of F is valid.

It's worth noting that all this stuff with generics is really just for the compiler and is an abstraction. The hardware doesn't care, and actually once the code is compiled, everything is an Object. Due to this, all Generics are just Objects when you try to do type inspection on them at runtime.

Generics can't:

- Throw or catch generic types.
- Use primitives
- Be static

- Be instantiated
- Mess up inheritance

9 Specifications

A specification is a written document from the designer to the programmer specifying the exact requirements for the project.

They usually include:

- User interface
- Input and Output
- Files and databases
- Objects, data structures, algorithms
- Features/functionality
- Menus, GUI's etc.

They are hard to write. They have to cover all cases, be clear, be what the client wants, and limiting assumptions.

How can we use the compiler to enforce specifications?

9.1 Design by Contracts

An interface can be used as a contract to enforce specifications.

Note that a class can implement multiple interfaces, and an interface can extend another interface.

However this doesn't contain any information about how the methods should be implemented, this is what the document is for.

When we do this, we could have something like `SomeInterface x = new SomeClassThatImplemented()`. The type of the variable dictates which methods can be called. (Methods that are in the class but not the interface cannot be accessed).

10 Anonymous Objects and Classes

As mentioned in a previous section, we can create objects without assigning them to a variable. This is useful for passing it into a method, which forces it to be garbage collected at the end of the method. Or, for passing into the constructor of another object, so that when the new object is destroyed, both will be destroyed.

We can also write an anonymous class directly in the instantiation of an object!

```
  Example<Object> ex = new Example<>(){
    public int compare(Example e1, Example e2){
      return e1.compareTo(e2)
    }
  }
```

When doing this, the class is tied to exactly one object. When the object is garbage collected, so is the class.

Even crazier, we can make a method, which will generate new anonymous classes!

```
public class Country{
  public static Comparator<Country> comparatorByName(){
    return new Comparator<Country>(){
      public int compare(Country c1, Country c2){
        return c1.getName().compareTo(c2.getName());
      }
    };
  }
}
```

Part III
Aside: Swing GUI Programming

I'll refrain from copy pasting the code from the slides here, but I'll do my best to provide the explanation that Prof. Vybihal gave in class.

11 Basic Structure

The basic element of a GUI in swing as a `JFrame` object. This will cause the window to be displayed, and can be populated with elements.On this you can set a size, set a layout, (there are layout objects in Swing too, for example `GridLayout`). You can then add a listener to this frame to specify close, resize, minimize behaviors. (By implementing interfaces in an anonymous class).

Next is the `JPanel` object. This is like the window, but is instead a canvas onto which elements are placed. This will be placed within the window. You can also give layouts to a panel,

To add buttons, we need to add listeners to our button elements. This is done by implementing the `actionPerformed` method. This method take an `ActionEvent` object which has information about the event.

12 Object Truth & Writing Good Classes

Classes and objects represent a logical entity, and so must always stay consistent with the context of that entity. Put another way, they are an analogy to real things, and so must be consistent with the analogy. If we have a class Student, we expect it to behave and contain things that a Student would have.

Truth is the idea that the objects instantiated from a class are consistent with the analogy of the class.

If we use setters, getter and mutators, we must ensure that they preserve truth.

A class is**invariant** when all objects of the class are always in a valid state.

This may be broken for a moment in time **during** a ”fix” to maintain this valid state.

This idea is similar to that of **cohesion**, in which all methods of a class should belong to the same abstraction or analogy. If the class is for Students, it doesn't have anything else.

We also need to find a balance with convenience, since too much cohesion can make the classes hard to use. For example, `pop()` is not only a pop. It also returns a value, so it's really a `peek()` and a `remove()`.

We should also name and program in a way that is intuitive and consistent.

13 Defensive Programming

Always assume people are going to break your stuff. We must then validate all inputs and outputs.

We do this using a **precondition** and **postcondition**. We use the precondition to test the inputs, do some work assuming everything is fine, and then validate the result at the end. Usually we can just use one, (usually preconditions because it saves work of doing the method).

We throw exceptions when the expectation of the result is not obeyed. (If one of the conditions fails.) We only return if the result and inputs were valid.

Part IV
Tools That Aid Good Design

14 Unit Testing

A unit is a single ”functioning unit”, that is, it has no specific size or shape. It could be one method, a class, or collection of classes. We then cover this unit by handling all possible cases.

We've already covered writing unit tests without tools in the earlier part of the course.

15 JUnit

JUnit is a java framework for unit testing. It basically looks for classes and methods with the name ”test” in them, and runs all your tests.

You would normally create a test class `DayTest` for example, that extends `TestCase`, a JUnit parent class. This parent class contains all the JUnit ”stuff”.

16 Javadoc

The importance of documentation is self-evident. Javadoc is a tool that auto-generates an HTML file based on specially formatted comments in your code. They always begin with `/**` (rather than just /*). We then use the @ symbol to specify attributes. (for example: @author Francis Piché, @return int)

Since it gets converted to HTML, you can actually inject html into the comments to add custom formatting!

Part V
Modeling

Software design is the idea of planning before coding. Abstractions and architecture are used to create a full model before beginning the implementation.

The development (which would be done by the programmers) is controlled by the designer by using effective communication, strong OO constructs, and unit testing. The requirements are described by contracts, generalized code and UML (or some other modeling method).

17 UML Class Diagrams

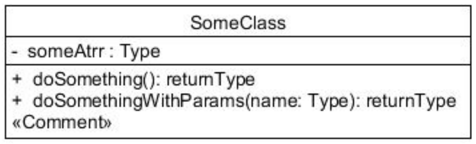
This diagram describes all classes, data structures, objects and what they contain.

You could exhaustively go and model every possible attribute, method etc. But this is really hard to do since it's almost impossible for a designer to forsee every method a programmer would need, without writing anything himself.

In this class, we will make minimal diagrams, where we specify only the minimal variables and methods.

You could also make hybrid responsibility diagrams, where instead of specifying methods you would write a "responsibility" description. (Not done in this course)

A basic class would look like:


SomeClass
- someAtrr : Type
+ doSomething(): returnType
+ doSomethingWithParams(name: Type): returnType
«Comment»

Where the + means public and the - means private.

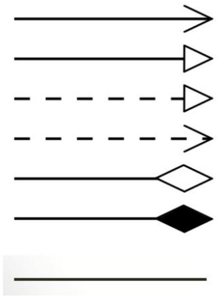
17.1 Relations

Since classes often depend and use each other, we need a way to depict this.

- Association : class A calls a method in class B
- Inheritance : A extends B
- Realization / Implementation : A implements B
- Dependency : A assumes something about B but doesn't use it directly.
- Aggregation : see below
- Composition : see below
- Undirected : Relation in an undefined way.

Aggregation and composition are a little weird to understand. (At least for me!) Class A aggregates class B, if class B can exist without the existence of class A. Meanwhile, Class A composes class B if class B cannot exist without being within class A. `https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/`

These relationships are expressed with these arrows: (in order)

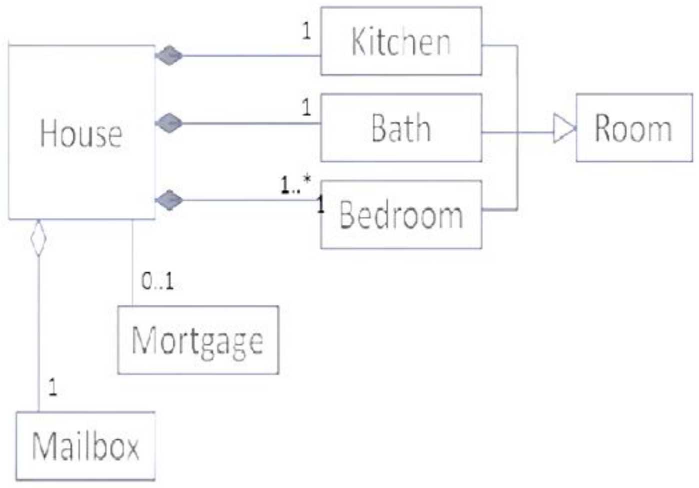


Note that having TONS of arrows is probably a good indication of bad design.

We can also express multiplicity by adding $n..m$ or $0..*$ etc. To show there should be between n to m instances, or 0 to ∞ instances.

17.2 Examples

17.2.1 House



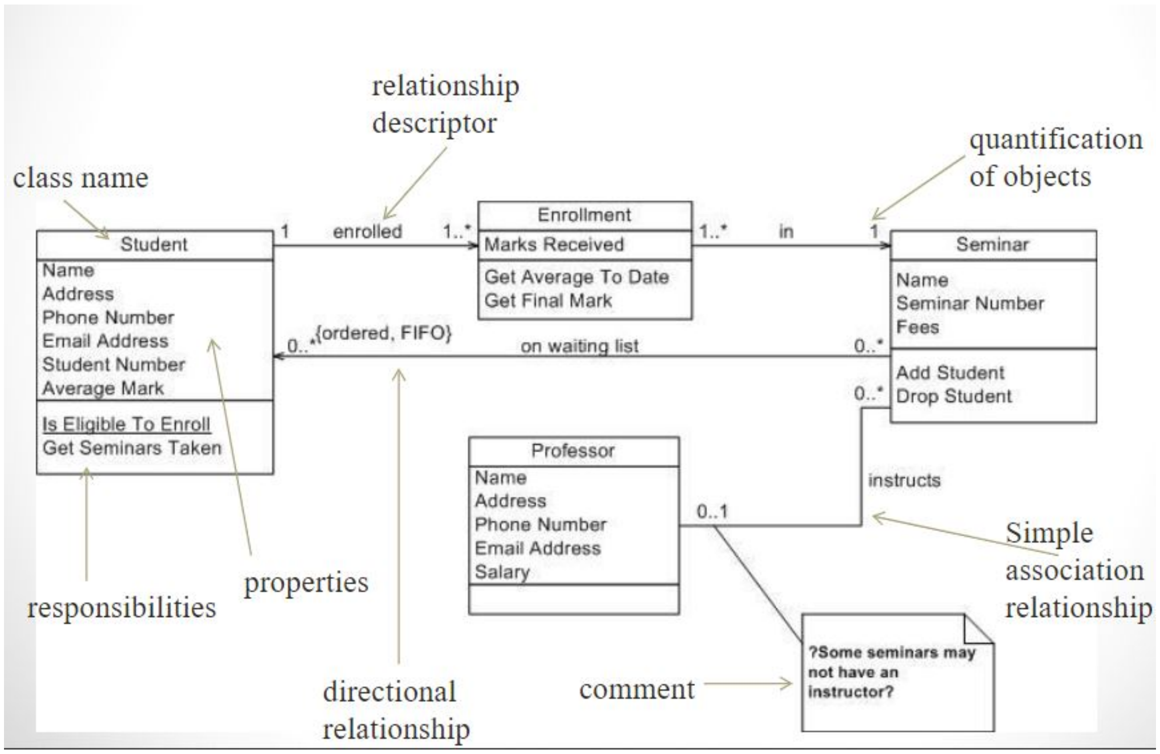
The main method is most likely inside House, since most instantiations are coming out of house.

In this picture, a House must have exactly 1 Kitchen, 1 Bathroom, and 1 to many Bedrooms. All of those extend Room. A Mortgage is somehow related, and can exist or not. A Mailbox is given to the house, but not necessarily part of it. (aggregation).

There is actually a problem with this. Is Mailbox static? If not, then it must be instantiated by something. But this picture says it must be static, but it's also associated to the House using an aggregation, which means it must be instantiated.

We could make this correct by adding another Main class which instantiates Mailbox (and possibly House if we don't want it to be static). **Always think of what needs to be instantiated!!!**

17.2.2 University



Here we are adding labels to the associations. These describe what the relationship actually is.

Notice that Professor has no methods, since they aren't important and we are doing a hybrid responsibility here (even though we said we wouldn't do those!). Meanwhile, Enrollment has some methods since they are important to understanding the situation and requirements.

18 Sequence Diagram

This is a use-case view of the system. Here we look at (in detail) what one part of the system will do in order. Very narrow and explicit.

A class diagram does not explain the order (sequence) in which the program executes.

In these we have a "timeline" representing execution flow, from left to right. The dotted lines coming down vertically break up the timeline into pieces by class. If the line is dotted the whole way down, the class is not executing, if there is a box, the class is executing. Solid arrows mean a method is being called, dotted arrows mean a value is being returned.

We can add actors, who initiate the use cases.

Loops and alternates are used to represent ”for, while” and ”or” (do not think about it as an if statement, since its not specified what the condition is.)

19 Activity Diagram

Is essentially a flow chart of that the program should do. It can be for a single method, or a use case.

This is more abstract and general than the sequence diagram, since it does not contain actual return values and methods, just plain words to describe whats happening.

Note that there can be only one starting point, and can be many end points.

We can also make a multi-actor and object diagram, where we break up the diagram into columns of groups, and then individual actors. All actions regarding that actor fall within its column.

Part VI
Design Patterns

The point of design patterns is to have a way of documenting language agnostic solutions to common software engineering problems. Often this is represented in UML.

A design pattern contains:

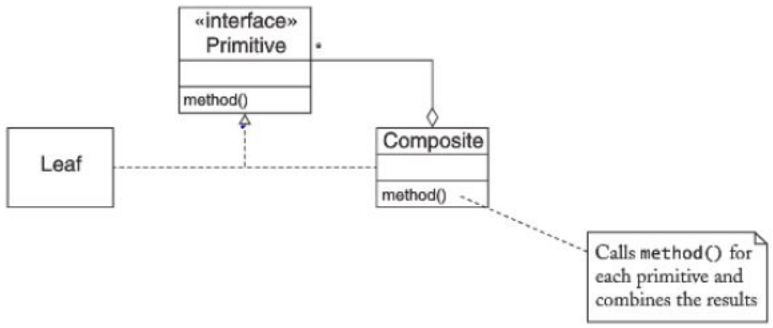
- Title
- Long description of the problem
- Full description of the solution using UML

20 Composite Pattern

From the wiki:

”A part-whole hierarchy should be represented so that clients can treat the part and whole objects uniformly.”

Basically, one object contains a list of others which implement a common interface so that they can be called together.



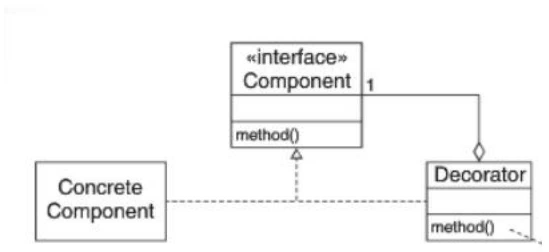
Here, both Leaf and Composite implement the Primitive interface. The method() of Composite simply calls the method() of all the Leaves which it contains.

This can be used to allow for individual Leaf objects to call method() by themselves, or be part of some greater Composite, and be called together.

Example:

```
class Team implements CompositeInterface{
   //Arraylist of CompositeInterface means we don't necessarily need all members
       of team to be same type
   //The deduct hitpoint could be different for each type of teammember
   private ArrayList<CompositeInterface> team = new ArrayList<>();

   //Call the deductHitPoint for each member of the team
   public void deductHitPoint(int n){
      for(T member : this.team){
         member.deductHitPoint(n);
      }
   }
}

//Same interface as the composite object (the Team)
class Soldier implements CompositeInterface{
   private float hp;
   public void deductHitPoint(int n){
      this.hp -= n;
   }
}

interface CompositeInterface{
   public void deductHitPoint(int n);
}
```

21 Decorator Pattern



21.1 When is it used?

The Decorator pattern is used when you have an object that cannot or should not be modified, but we want to add extra functionality to.

You might notice that the UML is very similar to the Composite, but there is only 1 Component in the Decorator composite object.

21.2 How does it work?

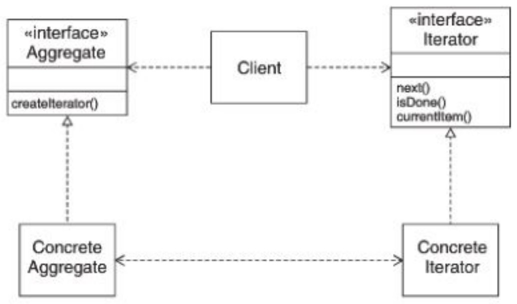
```
class ExampleRunner{
   public static void main(){
      ConcreteComponent c = new ConcreteComponent();
      Decorator d = new Decorator();

      double random = Math.random()*10;
      if(random > 5){
         d.method();
      }else{
         c.method();
      }
   }

}

interface Component{
   public void method();
}

class ConcreteComponent implements Component{

   //Empty constructor for example
   public ConcreteComponent(){}
   //Implement the interface method
   public void method(){
      //Do stuff
   }
}

class Decorator implements Component{

   //Composed of exactly one Component
   Component c;

   //Notice the interface type being used, so we could decorate decorators easily
       too!
   Decorator(Component c){
      this.c = c;
   }

   //Decorator only works if you need not modify intermediate steps inside the
       method() of the ConcreteComponent. Must be able to just use end result.
   public void method(){
      this.c.method();
      //Do extra stuff
   }
}
```

21.3 Discussion

We could also combine both the Decorator and Composite together. By having a 3rd class which implements the common interface, it could be composed of several Decorated (and non decorated) objects!

Also, note that using inheritance instead of interfaces is not the same thing. In inheritance, we would only allow decorators in the same inheritance tree. We could not decorate two objects with the same decorator if those two objects are not in the same inheritance tree. This would lead us to only add functionality through evolution down the tree. We would also lose the reference to objects further along the inheritance tree, since we wouldn't have a direct reference to the base object.

22 Iterator



22.1 When is it used?

The Iterator can be used whenever we want to traverse across an object, independent of what the object is. It could be a LinkedList, ArrayList, Tree, Graph, etc. The iterator would work for all of them. So it is useful in cases when you don't necessarily know the datatype of what you'll be iterating over, or if you want to specify custom iteration behavior (for example BFS vs DFS on a Tree).

22.2 How does it work?

The client depends on both the Iterator interface and the Aggregate interface. They do not depend on eachother, but the client makes them work together through some control structure. The iterator interface always has the following methods:

- next()
- hasNext()
- currentItem()

The Aggregate interface contains a `getIterator()` method. We then need two classes. One implementing the Iterator interface, the other implementing the Aggregate.

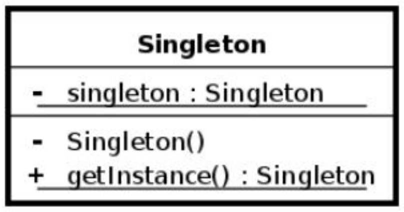
Often, the iterator class will be nested and private inside the Aggregate class, then the way they are accessed from the client is by calling the getIterator() method from the Aggregator, and calling next() until hasNext() returns false.

```
public interface Iterator{
    public Object next();
    public boolean hasNext();
    public Object currentItem();
}
```

```
public interface Aggregate{
   public Iterator getIterator();
}

public class ConcreteAggregate{
   //Some datastructure, doesn't matter which.
   private Tree<String> data = //Some data

   public Iterator getIterator(){
      return new ConcreteIterator();
   }

   //Private class to encapsulate the iterator.
   private class ConcreteIterator{
      TreeNode currentNode = data.root;

      //Just pseudocode
      @Override
      public boolean hasNext(){
         if(currentNode.children != null){
            return true;
         }
         return false;
      }

      @Override
      public TreeNode next(){
         if(this.hasNext()){
            //you get the idea
         }

         return null;
      }
   }
}

//Now we can easily iterate through without caring about the details.
public ClientExample{
   public static void main(String[] args){
      ConcreteAggregate example = new ConcreteAggregate();

      for(Iterator iter = example.getIterator(); iter.hasNext;){
         //Do stuff
         TreeNode next = iter.next();
      }

      /*Note that in Java 8, you can use the enhanced for loop on anything
```

```
        that implements the iterator interface (built into Java).
      for(TreeNode t : example.getIterator())*/
    }
  }
```

23 Singleton



23.1 When is it used?

The Singleton is used when we want to ensure that only one instance of the class ever exists at one time.

23.2 How does it work?

To understand the Singleton, it's important to understand the `static` keyword in Java. By now you probably know that Java is built on C. Essentially, objects boil down to structs in C. The difference, then, between static and non-static is that static allows the thing to be created compile time only. (Cannot be malloced).

We use this to ensure that only one instance of the singleton is created.

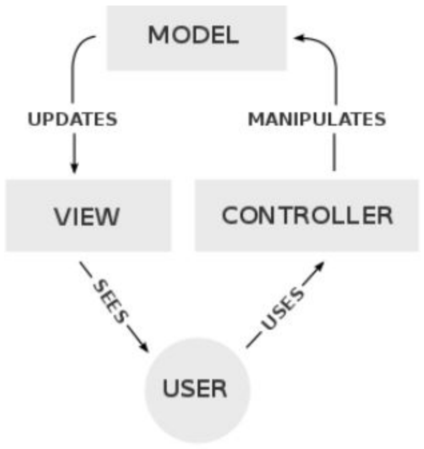
```
public class Singleton{

  //Notice the private static
  private static SingleTon instance = null;

  //Hide the constructor
  private Singleton(){}

  //Controlled static access to the singleton
  public static Singleton getInstance(){

    //Only create the instance if it's the first time this is called.
    if(this.instance == null){
      this.instance = new Singleton();
    }
```

```
        return this.instance;
    }

}
```

So now, the compiler will give an error if anyone tries to do `Singleton singleton = new Singleton()` from outside the singleton class, since the constructor is private.

We can also use this to wrap other objects to make them singletons as well, by having them as a private class in the Singleton file.

24 MVC



The Model View Controller is a **style** or set of similar design patterns that provide a solution to the same use-case. The idea is that most applications can be divided into three parts. The controller, model, and view. The controller is the interface between the User and the Model that, well, controls how data is updated. The Model is all the data, and the View is the visual representation of the data seen by the user.

24.1 When is it used?

This is used whenever the above described behavior is what describes your system. There can be many views, many controllers and many models. The ratio's between them are not always 1:1. We use this style when we want all of these things to be updated simultaneously. For example, a use presses a button. The controller updates the model data, and the view is updated to match the data. There could be multiple views for one model (think bar chart and pie chart for the same data), data may be manipulated by several controllers.

The canonical example of this is the Java Swing library. ller will have an array list of

24.2 How does it work?

Basically, the idea is just to separate the concepts into different classes. Your Controller class should only be responsible for:

- Handling user input
- Updating the Model(s)
- Displaying the View(s)

To do this, normally the controller keeps a reference to the Model(s) and the View(s), and has a method for updating and displaiyng.

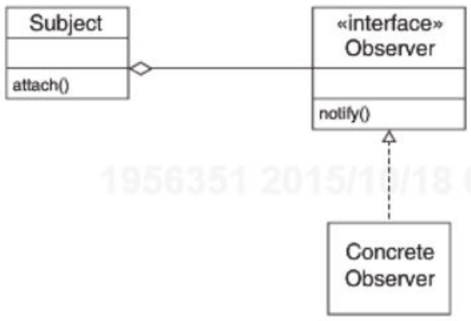
```
class Model {
   //Data stuff

   //Could be different/more complex, this is just an example for simplicity.
   void update(){
   }
}

class View {
   //Visual stuff (printing, GUI, etc)

   void display(){
      //Do all the displaying
   }
}

class Controller{
   private Model m;
   private View v;


   //Update is the only method that the developer will have to call. This can
       take parameters if needed.
   void update(){
      m.update();
      v.display();
   }
}
```

It is also common to see the controller and view together in one class, this is known as MVC1 (as opposed to MVC2 described previously). In this case, it's more of a ”interactive window” idea.

Multiple models and views can be implemented using **registration**. The idea there, is that the Controller has instead a list of Models/ Views. ”Registering” a view/model is just adding one to the Controller's list.

We can also modify this so that the Model can notify the View of changes, rather than the controller managing this. This is known as MVC with **Notification**. In this way, the Controller only updates the Model, which then tells all relevant views to repaint themselves. In this one, the Views need to be registered with the Model, not the Controller. This way is often done with the Observer design pattern.

25 Observer



25.1 When is it used?

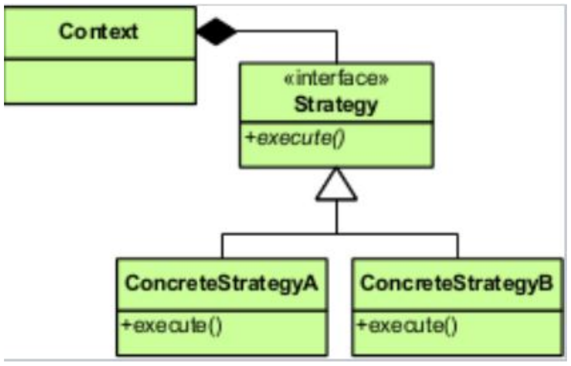
The Observer pattern is used whenever there is a one to many relationship in which a change in one object is to cause a notification in the relevant objects automatically.

25.2 How does it work?

The Subject class keeps a list of Observer objects (objects that implement Observer interface), and calls notify on all of them when a change is made to the Subject class. It also is responsible for handling attaching and detaching Observers from itself. Sometimes the Observers are an abstract class with a reference to the Subject, so they can attach themselves. Otherwise this is handled outside the pattern.

This is essentially what is happening with the MVC design pattern with notification.

26 Strategy



26.1 When is it used?

Suppose you have a class that could benefit from having several different implementations of similar logic.You want to do this in such a way that the implementation logic can be swapped out dynamically, with good encapsulation.

Now, one could have a bunch of if-statements or a switch-statement to handle which to use, or we could use the Strategy pattern to implement better code.

26.2 How does it work?

Essentially, we have a Strategy interface which will be implemented by the different "Strategies" for solving a problem. In a LayoutManager, these would be the exact layouts, BorderLayout, FormLayout etc.

Then, the Context class determines which one of these strategies is used, and how.

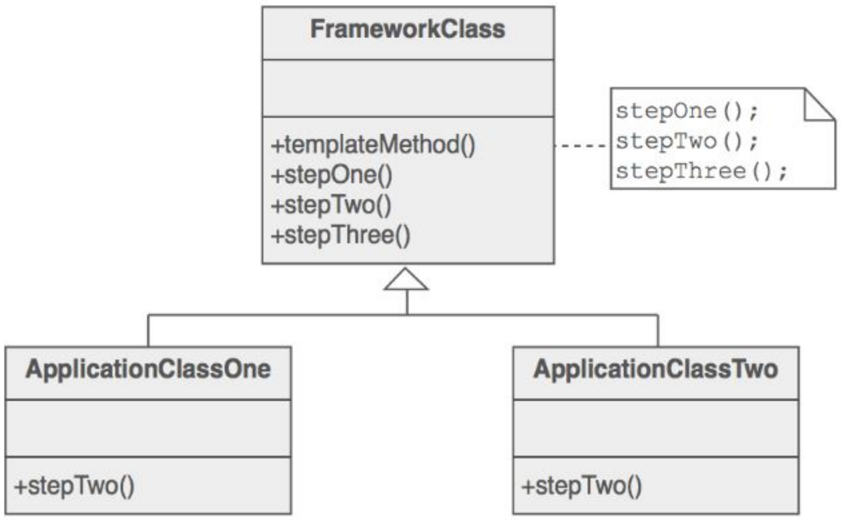
For example, we could have an avatar which uses two different AI techniques (strategies) to implement it's decision making. These can be changed depending on the context.

We could also combine the Strategy design pattern with itself, by having multiple interfaces that extend eachother.

```
interface Behavior{
   void doAction();
}

interface Thinking extends Behavior{
   Behavior think();
}

class Strategy implements Thinking{
   Behavior b;

   Behavior think(){
      //Decide which behavior to do
      return b;
   }

   void doAction(){
      //Do crazy stuff
   }
}

class Avatar {
   Thinking thinker = new Strategy();
   public void doThink(){
      Behavior b = thinker.think();
      b.doAction();
   }
}
```

So there, the Strategy decides which Behavior to use, (assume there are several), and then this can be used by the avatar to complete the action.

27 Template



27.1 When is it used?

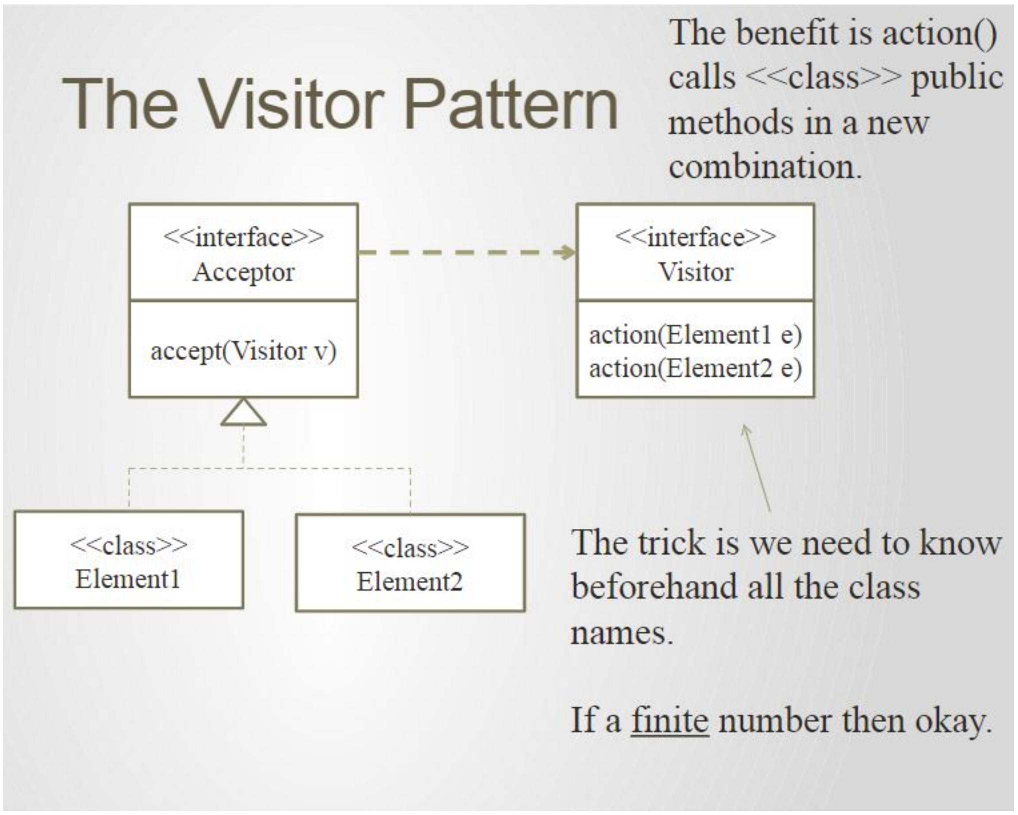
Say you have an algorithm that has many similar steps, but some that may vary in their implementation. How can we avoid duplicate code?

27.2 How does it work?

By using an abstract class, we can implement a template method, in which each step may or may not be unique. If it is unique to the implementation, then it should be declared abstract. Otherwise, it can be implemented in the abstract class.

```
abstract class ShoppingCart{
   bool makePayment(double amount){
      String info = paymentInfo();
      int ptr = connect(info);
      bool answer = pay(ptr, amount);
      return answer;
   }

   abstract String paymentInfo();
   abstract connect(String connectionInfo);
   abstract bool pay(int connectID, double amount);
}
```

28 Visitor


The Visitor Pattern
The benefit is action() calls <<class>> public methods in a new combination.
<<interface>> Acceptor
accept(Visitor v)
<<interface>> Visitor
action(Element1 e)
action(Element2 e)
<<class>> Element1
<<class>> Element2
The trick is we need to know beforehand all the class names.
If a finite number then okay.

28.1 When is it used?

Suppose we want to modify (add methods) to classes of many different types without having to alter those classes too much.

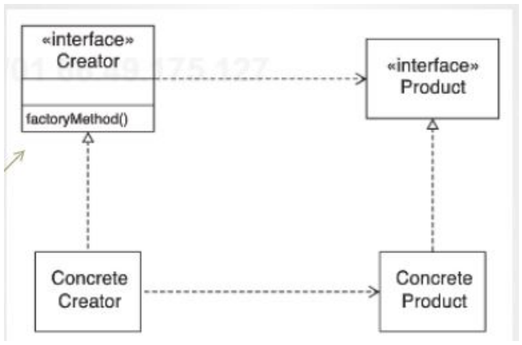
Easy! The Visitor pattern allows you to add a method (or several) to several classes by making those classes implement the Acceptor interface. Then, you specify the behavior of all your methods in the a class which extends visitor.

28.2 How does it work?

```
interface Visitor{

   public double visit(SomeClass1 c1);
   public double visit(SomeClass2 c2);
   public double visit(SomeClass3 c3);
}

interface Acceptor{
   //This is the only change that will need to be made to your classes to make
       them work with visitor design pattern
   public double accept(Visitor v);
}

class ConcreteVisitor implements Visitor{
   public double visit(SomeClass1 c1){
      //Do something specific to SomeClass1
   }
   public double visit(SomeClass2 c2){
      //Do something specific to SomeClass2
   }
   public double visit(SomeClass3 c3){
      //Do something specific to SomeClass3
   }
}

class SomeClass1 extends Acceptor{

   //Some class stuff

   public double accept(Visitor visitor){
      visitor.visit(this); // Pass reference to yourself so the Visitor class
          knows what to do
   }


}

class SomeClass2 extends Acceptor{

   //Some class stuff
   public double accept(Visitor visitor){
      visitor.visit(this); // Pass reference to yourself so the Visitor class
          knows what to do
   }

}
```

The problem with this is that we need to know all of the classes which will need to extend the Acceptor interface. If there is a finite number of them then fine, but when we add a new class we need to remember to add it to the interfaces!

29 Factory



29.1 When is it used?

Suppose we want to create an object at runtime, but we want to randomize the type of that object within a hierarchy. So if you have a bunch of Enemy classes that extend Enemy, say, and you want to generate a random type of enemy. The Factory is used to decide which enemy is created.

Alternatively, you can generate classes which implement a common interface as well, rather that within a hierarchy. This can be useful when separating logical components, and when using in conjunction with other patterns. (Maybe generating a Strategy for example)

29.2 How does it work?

```
interface Factory{
   public Product factoryMethod();
}

interface Product{
   //Some interface, abstract class or class
}

class Prod1 implements Product{
   //Some stuff
}

class Prod2 implements Product{
   //Some other stuff
}

class EnemyFactory implements Factory{
   public Product factoryMethod(){
    double random = Math.random();
    return (random > 0.5)? new Prod1() : new Prod2();
  }
}
```

30 Adapter

NOT COVERED FOR FINAL

31 Command

NOT COVERED FOR FINAL

Part VII Threads

Java threads in 15 minutes or less. LETS DO IT.

32 Threads in General

Parallel execution my dudes. Threads do that. They share all the same variables and memory though so be careful! Make sure you use proper exclusion techniques to elegantly deal with race conditions.

Race condition: Something that can go bad if theres two things modifying it at the same time. Note that something as seemingly harmless as `i++` can lead to race conditions. (Think what is i++ doing at the assembler level. It's made up of a read, and a write. So if one thread writes while another tries to read, boom your i is corrupt).

32.1 Mutual Exclusion

Dealing with race conditions always involves using locks.

Basically, locks will allow one (or more, depending on how you set them up), threads to enter a section of code, but will block all others from entering until the lock is released.

How this is done in Java is below:

33 Java Threads

Apparently Java is nice for multithreading. Not that we would know, we never got practice in this course.

In Java, if you want your class to be able to run as a thread, you have to extend the Runnable interface. Then, you implement the run() method in your class.

When you're ready to start your program, call start(). It's good practice to have a try catch block in your run() method since the thread could get interrupted, so you want to catch InterruptedException. It's also good practice to call Thread.sleep(DELAY); to give other threads a chance to run (since Java threads have no preemption, if you know what that is. Otherwise don't worry about it).

You can kill your threads by calling t.interrupt(). This could cause an interrupted exception if the thread was inside a wait(), sleep() or join().

33.1 Run vs Start

Calling run() is usually a mistake, since it won't actually create a new thread of execution (ie, your program won't speed up). However, if for some reason you want to execute the code you wrote inside run() inside the current thread of execution, go for it ;).

33.2 Java Locks

The simplest way to lock a critical section is to use the ReentrantLock() in Java.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try{

}catch{//handle some error maybe
}finally{
   lock.unlock();
}
```

33.3 Conditions

Conditions are a way of extracting the Object monitor methods (if you don't know what monitors are then don't worry about it). Basically, they can be used to have conditional waits.

```
class BoundedBuffer{
   pubic int size = 0;

   private Lock queueLock = new ReentrantLock();
   private Condition full = queueLock.newCondition();
   private Condition empty = queueLock.newCondition();

   public void add(E newValue){
      queueLock.lock();
      try{
         while(size == elemnts.length){
            full.await(); //Block any threads that try to add
         }
         elemnts.add(newValue);
         size++;
      }finally{
         queueLock.unlock();
      }
   }

   public void remove(){
      queueLock.lock();
      try{
         while(size == 0){
            empty.await(); //Block any threads that try to add
         }
         full.signal(); //Wake up one sleeping thread.
         elemnts.add(newValue);
         size--;
      }finally{
         queueLock.unlock();
      }
   }
}
```

Not used in the above example is signalAll(); which wakes up ALL threads which are waiting on a condition.

33.4 Synchronized

The Synchronized keyword in Java will implicitly lock your method so that no two threads can execute it at the same time.

```
public void synchronized superFastProgram(){ //Non-static so only this instance
    is synchronized
   //sensitive stuff

}
```

```
public static void synchronized superFastProgram(){

} //Static so ALWAYS synchronized

public void doThing(){

   //Can synchronize single blocks
   synchronized(this){

   }
}
```

33.5 join()

Join is often really badly explained so lemme try to make it simple.

In short, t.join() makes the thread which executed that line to wait until t.run() completes.

So if you do:

```
t1.start(); //Prints some stuff
System.out.println("banana");
```

You may end up seeing banana printed before the code of t1 can execute. To fix this:

```
t1.start();

try{
   t1.join();
}catch(InterruptedException e){
}
System.out.println("banana");
```

This forces t1 to finish before the print can happen.

This does not affect other threads started in the same level. Say you have two threads: t1, t2.

```
t1.start();
t2.start();

try{
   t1.join()
}catch blaba

System.out.println("banana");
```

The banana gets printed after t1 finishes, but leaves no guarantee that t2 has finished.