# *COMP 210 - Lecture Notes - 04 - Basic Classes and Hierarchies*

*January 18, 2016*

> These notes are more or less a reorganization and retelling of content from *How to Design Classes* chapters 1-4 and 10-14. Through a problem with geometric shapes we'll explore the design and implementation of OO code using Object Containment and Class Unions.

## *Some 2D Geometry*

Let's think about the following problem[1]:

[1] See page 27 and 155 in HtDC

> You've been tasked to develop some libraries for managing basic shapes laid out on a graphical canvas. The canvas is just a whole-valued, positive only coordinate system addressed in row then column order with the origin, $(0, 0)$ in the upper left hand corner.[2]. For starters, your library should support three shapes: *Circles*, *Rectangles*, and *Dots*. All shapes have an associated location. For circles that location is the center of the circle. For rectangles, that location is the upper left hand corner of the rectangle. For dots, that location is, essentially, the dot. Circles also have an associated radius. Similarly, rectangles have a length and width where length is the number of rows covered by the rectangle and width is the number of columns. All shapes must have the following functionality: compute and return its area, determine if a given point is within the bounds of that shape, compute the distance of the shapes "pin" location to a given location, modify the location of the shape, and compute and return the bounding box of a shape as a Rectangle.

[2] The standard coordinate system for 2D computer graphics

## *The Objects for Consideration*

The first design question to ask your self is, "What objects are implied by the problem description, what are their classes, and how are the represented?" In this problem we see 4 concrete classes:

1. 2D Cartesian Points for *row* and *column* coordinates
2. Dots which are represented by a location
3. Rectangles which are represented by a location, width, and length
4. Circles which are represented by a location and radius

Right away we see the need for object containment: every one of the shape classes will contain an instance of the Cartesian Point class. You might be tempted to drop the point class all together and simply use a row and column field in each shape type. You'd be wrong. For starters, there is little reason to flatten the problem

logic like this. You're likely to think and talk about "locations" and "points" and there's not reason to not reflect that thinking in your design. Additionally, encapsulating location specific functionality and logic, creates code independence between the shapes and the implementation of their locations. *As long as the interface to a Cartesian Point never changes, you're free to tinker around with the implementation of some or all of it's behaviors without breaking the Shape's own implementation.*[3]. Finally, by moving location logic into a separate class we get to reuse code. If every Shape does it's own location logic, then you'll end up repeating some logic verbatim within each shape class.

[3] this is referred to as *Separation of Concerns*

## *Some of these things are a lot like the others*

Dots, rectangles, and circles are not unrelated. In fact, we keep referring to them generically as *shapes*. A shape is not an object type in the same sense as a Rectangle. The later is tangible, concrete, while the former is abstract and conceptual. We can understand and think in terms of the Shape abstraction, but have not yet seen how to encode this abstract relationship into our programs.

If a Shape isn't a concrete object, then there can't be a Shape class. What shape really describes is the CLASS UNION[4] of the Dot, Rectangle, and Circle classes. Any object of type Dot, Rectangle, or Circle should also be viewed as an instance of the Shape type. The question we must ask now is, "what makes a shape a shape?"

[4] union here is the same union discussed in the mathematics of sets

Once again we turn to behavioral abstraction. We'll define shapes not by specific data, or physical properties by by behavior. In programming terms, we're talking about a COMMON PUBLIC **interface**. The old saying, "walks like a duck, quacks like a duck, must be a duck" is appropriate here. It defines ducks by what they do, not by the presence of a beak, feather, webbed feet, etc.

Our problem statement clearly lists the expected behavior for shapes.

- Shapes can compute their own area
- Shapes can determine if a given point falls within the shape
- Shapes can determine the distance from the shape's pin location to another point
- Shapes can compute their own bounding boxes
- Shape can change their location

Notice that two of these behaviors could be restated as physical properties and envisioned as data fields, not methods:

- Shapes have an area
- Shapes have a bounding box

This way of thinking is restricting. Both properties can be computed from other inescapable properties of a Shape. More importantly, committing to representation via data means committing to a specific implementation. If area is something returned by a shape, then we have not committed ourselves to any one means of achieving that return value. This same principle underlies the usage of basic getters and setters. By restricting access to data fields to methods, to abstractions, then you're not committing to any one representation of that data.

What we're seeing is our first CLASS HIERARCHY. When a set of classes can be viewed as subsets of some larger, logical class, then we can create a CLASS UNION by identifying the expected behavioral interface for all members of that union.

## *From Hierarchy Design to Java Implementation*

A class union defined solely in terms of object behavior[5] is defined using a Java *interface*. An interface provides a series of method declarations and documentation. The intent is quite clear: define the type by it's expected interface. Concrete instances of that type, i.e. the subclasses of the union, then declare that they *implement* that interface[6]. Classes can implement any number of interfaces in Java. Once a class declares an interface, then it must either implement that interface or declare unimplemented interface methods as abstract using the *abstract* keyword. This later option forces the class itself to be abstract. We'll learn about abstract methods and classes later. For now, all our unions will be represented by their interface.

[5] methods

[6] Interface-based types can also be subtypes of a union. In this case the subtype interface *extends* its supertype

### *UML Diagram*

Before we get to the coding, we'll sketch out a visual diagram of our Class hierarchy using **UML**[7]. This lets us write down everything but actual method implementation details. In doing so we see all the "what" details of our design without getting bogged down by the "how". This is nice in part because Java doesn't typically let us separate this like we did in C++[8]. It also let's you devise an implementation plan. Eclipse can do a lot of the coding grunt work for us, but we need to be able to direct it. It can help you implement a design, but I won't do the design work for you. By diagramming your hierarchy, you're writing down your ideas in such a way that doesn't commit you to any code.

[7] Unified Modeling Language

[8] Header + Implementation

In UML, all classes and interfaces are designated with a box. At the top of the box is the class/interface name. Interfaces are labeled as such to avoid confusion with concrete classes. Object containment relationships are diagrammed using an arrow with an arrowhead shaped point that points from container to containee. Subclasses of a class union have an arrow with a triangular point that points from subtype to supertype. All interface methods are declared under the interface name. Methods are declared in the following way:

```
name(parameter list) : return type
```

Public and private methods can be designated as such using + for public and − for private. For classes, we first list the fields under the class name using the following syntax:

```
name : type
```

Once again, a + or − can be used to designate a field public versus private. Class methods are listed below the fields with the same syntax as interface methods. However, we do not have to restate any methods inherited from a superclass or interface. It behooves you to work top-down[9].

[9] Abstract to Concrete. Supertype to subtype

Here's what the diagram for our Shape hierarchy looks like:

TODO

Figure 1: Shape Hierarchy in UML

There's a lot more to UML diagramming. We're just employing some basics in order to clearly state the core details of our class design.

### *Implementation in Java*

Eclipse has several features that help you to get Classes and Class Hierarchies setup quickly:

- Automatic stubs for inherited methods by declaring interfaces[10] in the new class wizard.
- Launching the new Class/Interface wizard from unresolved type errors as a Quick fix option.
- Automatic generation of Class getters and setters based on declared fields
- Automatic generation of Class constructors based on declared fields
- Automatic generation of *equals*, *hashCode*, and *toString* overrides based on class fields

[10] and super classes

- Automatic generation of stubs for undeclared methods

Basically, most of your boilerplate can be automatically generated for you and the methods that all objects inherit from the *Object* class can be automatically overridden using accepted Java best practices.

To make the best use of Eclipse's automatic code generation features, we'll work the implementation of our hierarchies from most to least abstract. This typically means interface on down.

1. Complete interfaces by declaring and documenting all their methods.

   Whenever you need to reference a Class or Interface that is a part of your program but is not yet implemented, just go ahead and state the type. Eclipse will highlight it as an error. Ignore it until the interface is complete.

2. For any undeclared interfaces or classes in your completed interface, use the *Create Class* or *Create Interface* quick fix[11] to have Eclipse stub out the Class/Interface. Be sure to add interfaces to the classes from within the wizard so that eclipse can stub out the methods for you.

3. Repeat the above steps until you have rough stubs for the complete hierarchy. All that should be left at this point are classes[12] of contained objects. Repeat the above stubbing process for those classes/hierarchies as well.

4. For each class, add the data field declarations at the top of the class, above stubbed out methods.[13].

5. For each class use the *Source* menu to auto generate:

   - A no-arg/default constructor and a field initialization construction[14]
   - Getters and Setters
   - An *equals* and *hashCode* method
   - A *toString* method

   Eclipse lets you specify the entry point for all the code it generates. For our Java classes, we want all the above stubs and generated code listed in the following order: fields, constructors, getters and setters, equals, hashCode, toString, and finally inherited methods.

6. All the boilerplate and stubs for all the methods in all the classes should now be in place and we can now generate JUnit Test cases for each class and begin writing your tests.

[11] hover the mouse over the error

[12] or hierarchies

[13] We'll list fields then methods

[14] one argument per field

In practice you should test any method that is public. Constructors can be tested implicitly through the methods[15]. We may loosen our testing practices up for auto-generated code eventually, but to start out, writing tests is an excellent way to familiarize yourself with the methods and their functionality.

[15] if it behaves the way its supposed to, then it was constructed correctly

7. Verify your auto-generated code then begin implementing your core methods.

   If, while implementing your design, you decide you need a helper method, go ahead and call it. Eclipse generates an undeclared method error with a quick fix option of adding that method to the appropriate class. If you're confident in the helper's purpose, then finish the method that caused you to need the helper. If you're unsure about the helper, then stop work on the current method in order to document, write tests for, and implement the helper.

It often makes sense, and is convenient, to run the tests for multiple classes at once. For example, you might like to run all the tests for all the classes that implement a particular interface. This is accomplished by a *JUnit Tests Suite*. To create one go to *New* > *Other* > *JUnit* > *Test Suite* and use the wizard to create a suite.

## *Polymorphic Method Dispatch*

While you cannot instantiate a Shape object[16]. You can have variables of that type.That variable can take on values[17] of any of the classes that implement it. We call this variable POLYMORPHIC as its actual value takes on many different forms, or types. There is often significant advantage in doing this. It's important to understand how computation proceeds when working with these abstractly typed variables.

[16] nor any Interface type

[17] reference objects

Consider the following test: The type of *aShape* is *Shape*. The Shape

```
// A 5x5 square at row 14, column 10
Shape aShape = new Rectangle(new Loc(14,10),5,5);
assertEquals(aShape.area(),25.0,0.000001);
```

Figure 2: Testing a Rectangle with a Shape Variable

type is an interface and has no concrete implementation for area. So, determine how to compute the area of *aShape* cannot be as simple as, "run the method implementation for the type of the variable."

The key, of course, is that the type of the variable is less important than the type of the value. In our example, the value stored in *aShape* is of type *Rectangle*. This type has a clear and unambiguous

implementation for area. The only question is how does the computer make this connection? The answer is SUBSTITUTION. First the computer evaluates the variable itself and effectively substitutes the name for the value. In our example this produces the unambiguous statement[18]:

[18] the assigned expression is the value

```
assertEquals(new Rectangle(new
    Loc(14,10),5,5).area(),25.0,0.000001);
```

Figure 3: Name-Value Substitution to resolve ambiguity

What if this came next?

```
aShape = new Dot(new Loc(0,0));
assertEquals(aShape.area(),0.0,0.0000001);
```

Figure 4: Assigning a new value to *aShape*

Now the exact same expression[19] produces the execution of different code.

[19] aShape.area()

```
assertEquals(new Dot(new Loc(0,0)).area(),25.0,0.000001);
```

Figure 5: Once again, object type, not variable type, determines method call

This is an entirely new phenomenon for us. The reason is clear, the expression maybe the same but the type of the object referenced by aShape has changed and it's that type that determines which implementation of area is executed. This feature of OOP and class hierarchies is called POLYMORPHIC METHOD DISPATCH[20]. Recall that the metaphor of operation is message passing. Calling *area* is done by passing a message, or dispatching a message, to the appropriate object which then executes the code. The dispatch described by this expression can takes on many different forms[21] within a single program based on the type of the object referenced by *aShape*.

[20] aka Dynamic Dispatch

[21] polymorphic