Implementing an Object-Oriented Design

Learning Goals

You must be able to:

- ✓ **Express a UML class diagram as a Java program, including expressing associations, compositions and aggregations.**
- ✓ **Extract and explain an object-reference graph from a part of an object-oriented program.**
- ✓ **Ensure a Java class can be used correctly with the Java Collections Framework by correctly overriding appropriate methods (e.g., equals, hashCode).**

Implementing a UML Class Diagram

An object-oriented software design guides and constrains how an object-oriented program should work. Some parts of an object-oriented design translate directly to an object-oriented implementation. Other parts of a design require care to translate them correctly. A UML class diagram describes the structure of an object-oriented program. The Object-Oriented Design I reading described the purpose of a UML class diagram and provided a pointer to a reading about the fundamental parts of a class diagram. This reading assumes you have a solid understanding of how to read a UML class diagram from those readings.

In this reading, we will consider how to implement a given UML class diagram in Java. When we translate a UML class diagram to Java, we are focusing on the structure of the program: the classes, methods and fields needed for the program. A complete implementation requires adding more details, such as the implementations of all methods.

We will continue with the example of a simple drawing editor to illustrate the implementation of a UML class diagram. The drawing editor we will work with allows the placement of rectangles and circles of fixed size on a drawing and the movement of those

figures around the drawing. Figure 1 provides the basic UML diagram for this drawing editor. The UML class diagram is not complete and focuses on aspects needed to illustrate points for this reading. This diagram uses the multiplicity * as an abbreviation for 0..*.

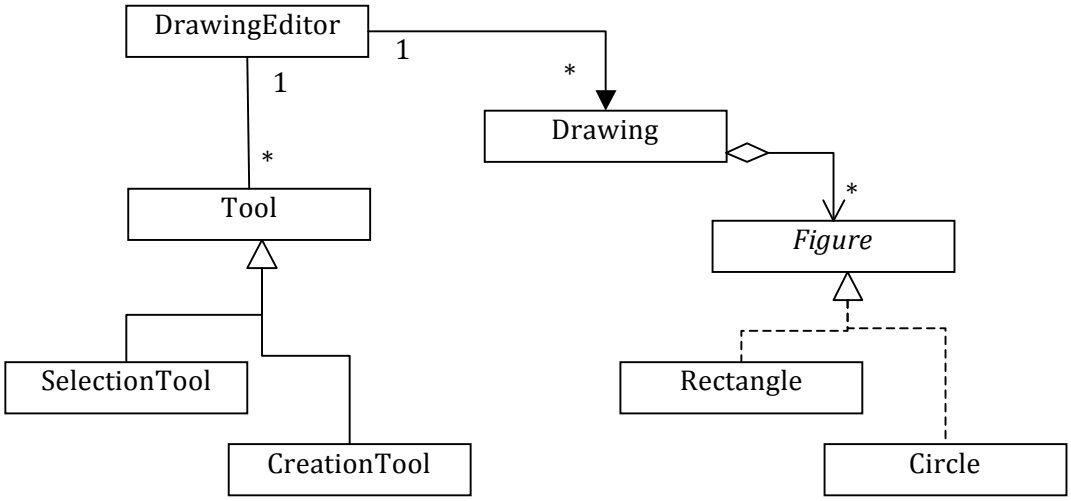

Figure 1 UML Class Diagram for Drawing Editor

Expressing Classes, Interfaces and Inheritance

The first step in implementing a UML class diagram is to express the classes and interfaces described in the diagram as Java classes and interfaces. For the Drawing Editor, we need to express seven classes (`DrawingEditor`, `Tool`, `SelectionTool`, `CreationTool`, `Drawing`, `Rectangle` and `Circle`) and one interface (`Figure`).

We can also read directly off the UML class diagram to express inheritance relationships. For instance, we express `SelectionTool` and `CreationTool` as subclasses of `Tool` using the Java `extends` mechanism, as in:

```
public class SelectionTool extends Tool …

public class CreationTool extends Tool …
```

We capture the relationship between the classes `Rectangle` and `Circle` and the `Figure` interface using the Java `implements` mechanism, as in:

```
public class Rectangle implements Figure …

public class Circle implements Figure …
```

Expressing Associations

When we implement an association, we need to consider both directionality and cardinality. The Drawing Editor design includes two associations: 1) a uni-directional association that indicates that a `DrawingEditor` object can access multiple associated `Drawing` objects and 2) a bi-directional association that indicates that a `DrawingEditor` object is associated with multiple `Tool` objects and that each `Tool` object can access its associated `DrawingEditor`.

Expressing a uni-directional association

Let's start by looking at the uni-directional association between `DrawingEditor` and `Drawing`. If the `DrawingEditor` object needs to be able to access multiple `Drawing` objects, this fact suggests that the `DrawingEditor` needs to have a field that can remember multiple `Drawing` objects. As we have seen earlier in the course, we can use the Java Collections Framework (JCF) to collect multiple objects into a single field. Thus, we start our implementation by creating a field of the Java Collections Framework type `Collection`:[1]

```
public class DrawingEditor {

    private Collection<Drawing> drawings;
    …
}
```

Note that the `Collection` interface is *generic*; that is, it accepts a type, in this case `Drawing`, which specifies the kind of objects that will be placed into the collection. As you have seen in the code that we have read, when we initialize the `drawings` field and create an object of the `Collection` type to hold the objects (i.e., in the constructor) we will have to choose the kind of collection to create, such as a `List` or a `Set`. The UML class diagram does not provide enough information directly for us to make this choice. Our choice has to depend upon the problem we are trying to solve. As we show in Appendix A, for now, we have chosen a list representation for the collection (see the implementation of the constructor for the `DrawingEditor` class shown in Appendix A). We come back to the possible choices for a collection and how one might choose between the possibilities later in this reading. Declaring the field `drawings` in `DrawingEditor` is not sufficient to

[1] In general, it is helpful to declare variables with the most general type possible. However, as you determine more specific requirements for the data structure used to implement the Collection you might refine the type used to declare the variable. For instance, declaring the variable to be of type List will enable positional access to items in the collection.

implement the bi-directional association even from the viewpoint of `DrawingEditor`. We also need to provide support for adding drawings to the association. We refer to the method `addDrawing` (shown in Appendix A) as a *setter* method because it helps to set the `drawings` field in `DrawingEditor`. At this point in our implementation, we may not yet know which object will be responsible for calling `addDrawing`; we thus leave the visibility of the method as the default visibility for now, refining whether the method is public, private or default as we iteratively implement the design.

Take a look at the partial implementations of `DrawingEditor` and `Drawing` in Appendix A.

Expressing a bi-directional association

Let us now consider the other association in our design, the bi-directional association between `DrawingEditor` and `Tool`. As with the uni-directional association, our DrawingEditor object needs to access multiple `Tool` objects, this fact suggests that the `DrawingEditor` needs to have a field that can remember multiple `Tool` objects. We will again use a `Collection` in `DrawingEditor` to store our `Tool`s.

```
public class DrawingEditor {
   …
   private Collection<Tool> tools;
   …
}
```

As we show in Appendix B, for now, we have chosen a set representation for the collection (see the implementation of the constructor for the `DrawingEditor` class shown in Appendix B) because it's unlikely that we want to have two similar tools (ie, two rectangle drawing tools) associated with our Drawing Editor application. We also need a setter for this end of the association; this setter method, called `addTool`, is shown in Appendix B.

So far, we have expressed one end of the bi-directional association between `DrawingEditor` and `Tool`s. We still need to capture that each `Tool` has knowledge of which `DrawingEditor` with which it is associated. This requirement suggests that the `Tool` class also needs a field to remember the `DrawingEditor` it is associated with. Since the cardinality of the bi-directional association between the two classes is one, the field can simply be declared to be of the type `DrawingEditor` as in:

```
public class Tool {

   private DrawingEditor editor;

   …
}
```

As we did for the `DrawingEditor` end of the association, we need a setter for this end of the association as well; this setter method, called `setEditor`, is shown in Appendix B.

Take a close look at the implementations of `addTool` and `setEditor`. You might have expected the implementations to be much simpler. For instance, you might have expected `addTool` to be similar to `addDrawing` and look something like this:

```
void addTool( Tool newTool ) {
   tools.add(newTool);
}
```

If we had implemented `addTool` as shown above, we would be requiring the code calling `addTool` to somehow ensure that the newly added tool has had its editor field appropriately set. Consider the following code:

```
public class Driver {

   public static void main(String[] args) {
        DrawingEditor editor = new DrawingEditor();
        editor.addTool(new CreationTool());
        // Point A
   }
   …
}
```

At Point A in the code above, assuming the `addTool` method just above, we would have only one half of the bi-directional association in place. If we instead use the implementations of `addTool` and `setEditor` in Appendix B and we have the code below:

```
public class Driver {

   public static void main(String[] args) {
        DrawingEditor editor = new DrawingEditor();
        editor.addTool(new CreationTool(editor));
        // Point A
   }
   …
}
```

then we always have the constraints of the bi-directional association met.

Note that both the uni-directional and bi-directional associations we looked at implementing had one-to-many cardinalities. A one-to-one association is simpler to implement as it only requires a field of the associated type in a class rather than a `Collection`.

Expressing Aggregations

We express an aggregation in the same way that we express an association because an aggregation is just a special kind of association. The class with the open diamond attached is the composing class and an object of that class will compose multiple objects of the class at the other end of the aggregation. When implementing such an aggregation, the composing class has a field with a `Collection` type that can manage the multiple composed objects.

You might then be wondering why we distinguish aggregations from associations on a class diagram. The aggregation can provide useful information that the composing class is a "whole" that contains various "parts", namely objects of the composed class.

Expressing Compositions

[Note we will not be distinguishing between aggregations and compositions. We left this part of the reading intact in case you are interested in going deeper into the subject. You can skip this section and start reading again at "Maintaining Constraints on an Association"]

In an aggregation, the lifetimes of the whole, or containing object, and the parts, the contained objects, are not linked. In a composition (the black diamond on a UML class diagram), the lifetimes of the container and contained objects are linked. If the container ceases to exist, the contained objects also must cease to exist.

There is one composition shown in the UML class diagram for the Drawing Editor example: a `Drawing` composes multiple `Figure`s. We implement this composition similar to how we implemented the associations with one-to-many cardinality. The `Drawing` class has a `Collection` type that manages multiple objects that correspond to the `Figure` interface (see Appendix C). Similar to associations, a part may be added to the whole after the whole is created, so the `Drawing` class has an `addFigure` method.

How do we ensure that the `Figure`s contained inside a `Drawing` are deleted if the `Drawing` is deleted? In some object-oriented languages (e.g., C++), the programmer must write code that ensures such deletions happen. In Java, the destruction of objects is handled by the Java virtual machine's garbage collector. Here is the basic idea of a garbage collector. When you create an object, Java allocates memory for the object and gives back a reference to the object. As long as the program maintains access to that object reference through a variable or a field, the object remains accessible. If the program does not maintain access to an object, the object becomes garbage. Every so often, the Java garbage collector is automatically invoked by the Java virtual machine as your program runs. Its job is to go through memory to find objects that are no longer referred to by your program and collect that garbage.

Consider the following piece of code:

```
public static void main(String args[]) {
    String s = new String("a");
    // Point A
    s = new String("b");
    // Point B
}
```

When execution reaches Point A in the code above, the variable `s` refers to the `String` object in which the string is "a". The next statement after Point A creates a new `String` object in which the string is "b". At Point B, the variable `s` refers to the "b" `String` object and there is no reference in the program to the "a" `String` object. If the garbage collector runs at Point B, it will determine there is no reference to the "a" `String` object and will collect this garbage.

We can thus ensure that a composition association has the right semantics for deletion if we can ensure that only the composing object has a reference to the composed objects. When the garbage collector determines that the composing object, say `Drawing`, is ready to be collected as garbage, the collector will also automatically collect the composed `Figure` objects.

Consider the implementation of the `Drawing` class below. This example demonstrates what **not** to do if you want to ensure `Drawing` composes `Figure`s.

```
public class Drawing {

  private Collection<Figure> figures;

  …

  public boolean addFigure(Figure newFigure) {
    return figures.add( newFigure );
  }

  // Return the first figure in the drawing containing the
  // specified point
  public Figure getFigureContainingPoint(Point p) {
     for (Figure f: figures) {
       if (f.contains(p))
          return f;
     }
  }

}
```

The problem with the code above is that the method `getFigureContainingPoint` can leak out a reference to the composed figure. Instead of the `Drawing` being the only object containing a reference to one of its composed `Figure` objects, another variable or object may now maintain a reference. If the `Drawing` is deleted, the composed `Figure` object that has been leaked will not be collected if a reference to it is maintained by another object or variable.

Instead, to maintain the semantics of the composition, if we need a method like `getFigureContainingPoint`, we should implement it as follows:

```
public Figure getFigureContainingPoint(Point p) {
    for (Figure f: figures) {
      if (f.contains(p))
         return new Figure(f);
    }
 }
```

In this code, a copy of the `Figure` object is returned instead of the original `Figure` object. This code ensures that a reference to the composed figure cannot leak out of the composition. On the other hand, one would question this code to ask whether or not having a copy of the composed figure referenced elsewhere in the system is a good idea. This question can only be answered by understanding more of the functionality of the entire application.

Maintaining Constraints on an Association

The examples above focus on one-to-many associations (aggregations and compositions) where the definition of many is zero to n. If an association is one-to-many where the definition of many is one to n, then the constructor for the class at the one end of the association must ensure that at least one instance of the object at the other end of the association is available.

The examples above show how to add an object into an association, but not how to delete an object from the association. If the association needs to support deletion of objects from the association, appropriate methods must be added to remove an object similar to the methods we have shown for adding an object into an association. The same holds for compositions or aggregations.

Choosing a Collection from the Java Collections Framework

When you implement an association (or an aggregation or a composition) that includes a cardinality greater than one, you must choose a type for the field that will hold multiple objects. One option is to define a new class (type) to hold the multiple objects. As Java has extensive support for different types for holding collections of objects in the Java Collections Framework (JCF), more likely you will reuse an existing type from the JCF. The types available in the JCF provide different capabilities and make different trade-offs between processing speed and memory use. As we have already seen in examples in class, a `List` data type allows duplicate objects and allows objects in the list to be accessed based on their numerical position in the list. A `Set` data type, on the other hand, ensures there are no duplicate objects in the set.

The Java Tutorials provides a good overview of the JCF. Read the sections listed below.

- The Collection Interface (http://java.sun.com/docs/books/tutorial/collections/interfaces/collection.html), you can skip "Collection Interface Bulk Operations" and "Collection Interface Array Operations".

- The Set Interface (http://java.sun.com/docs/books/tutorial/collections/interfaces/set.html), see below for a discussion of hashing, equals and hashCode which are referred to in the Set interface description. You can also skip the description of the generic method removeDups, the "Set Interface Bulk Operations" and "Set Interface Array Operations".
- The List Interface (http://java.sun.com/docs/books/tutorial/collections/interfaces/list.html), you can skip "Comparison to Vector", focus on the part of "Positional Access and Search Operations" up to, but not including the shuffle description. You can also skip "Range-View Operation" and "List Algorithms".
- We will talk about the Queue interface in class. Read The Queue Interface after class (http://java.sun.com/docs/books/tutorial/collections/interfaces/queue.html).
- We will also talk about the Map interface in class. Read The Map Interface after class (http://java.sun.com/docs/books/tutorial/collections/interfaces/map.html), skipping "Comparison to Hashtable", "Map Interface Bulk Operations", "Fancy Uses of Collection View: Map Algebra" and "Multimaps".

To be an effective Java programmer, you will need to be able to select the right data type for a problem you are trying to solve and you will need to ensure the objects you are putting in a JCF type meet any constraints imposed by the type. We will look at some best practices for using the JCF types in the next section.

Best Practices for Using the Java Collections Framework

For the JCF to correctly manage objects of a class you write, there are two rules from Joshua Bloch's Effective Java [1] that you must follow.

- "Obey the general contract when overriding equals" [1, Item 8] and
- "Always override hashCode when you override equals" [1, Item 9].

In addition, if you use any of the JCF types that manage a collection of objects in sorted order, you may need to also follow Bloch's recommendation to:

- "Consider implementing Comparable" [1, Item 12].

We will consider brief meanings of each of these rules below. See [1] for complete details.

Overriding equals

The Java `Object` class, which is a superclass to all Java classes, defines a number of methods, including `equals`, which supports the determination of whether one object equals another. Remember that variables and fields hold references to objects. So, if you try to compare whether one object is equal to another with the == operator, you are checking whether two variables (or fields) refer to the same object but not whether two objects have equivalent contents (or state). Consider the following code.

```
String s = new String("a");
String t = new String ("a");
s == t; // is false
s.equals(t); // is true
```

We want the return value of the last statement in the code above to be true (and it is) because the two objects referred to by the variables `s` and `t` have the same contents.

It is important to ensure the `equals` method works correctly for the objects you place in JCF types in case you use the facilities of those types to search for or find an object with particular contents in a collection. Ensuring the `equals` method works correctly typically means overriding `equals` in the class you have defined. When you override `equals`, it is critical to adhere to the general contract of the `equals` method defined in Java's `Object` class, which states that

"the `equals` method implements an equivalence relation on non-null object references:

- It is *reflexive* for any non-null reference value, `x.equals(x)` should return true.
- It is *symmetric* for any non-null reference values `x` and `y`, `x.equals(y)` should return true if and only if `y.equals(x)` returns true.

- It is transitive for any non-null reference values, `x`, `y` and `z`, if `x.equals(y)` returns true and `y.equals(z)` returns true, then `x.equals(z)` should return true.
- It is consistent: for any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return true or consistently return false, providing no information used in `equals` comparisons on the objects is modified.
- For any non-null reference value `x`, `x.equals(null)` should return false."

[Java Object equals documentation]

For examples of why each of these parts of the contract matter, see [1].

When you override `equals`, there is a basic formula you can follow to help ensure the implementation you write meets the necessary contract. We'll explain the formula by writing an `equals` method for an expanded version of the `Rectangle` class in our Drawing Editor example. The formula is described in the comments for the `equals` method.

```
public class Rectangle implements Figure {

   private Point topLeftCorner;
   private int length;
   private int width;

   @Override
   public boolean equals(Object obj) {
      // Step One. Return false if object to compare to is null
      if ( obj == null )
        return false;

      // Step Two. Make sure the two objects to compare are
      // of the same type. We use "getClass" to do this check.
      if ( getClass() != obj.getClass() )
        return false;

      // Step Three. Treat obj as a value of type Rectangle,
      // by casting it. We know it's ok to treat it as a
      // Rectangle because we checked its type in Step Two
      Rectangle other = (Rectangle) obj;

      // Step Four. Check that the values of all fields in both
      // objects are equivalent. Note because other is of the
      // same type as the type containing the declaration of
      // this equals method, we can access private fields
      return topLeftCorner.equals( other.topLeftCorner ) &&
             width == other.width &&
             height == other.height;
   }
}
```

Override hashCode

Implementations of some JCF types use the concept of hashing, which provides a way to quickly locate an object within the collection by using a hash function that represents an object as an integer value. We'll look at how hashing helps improve the performance of collection types in class. In this reading, we'll focus on how you produce an integer value for an object by overriding the `hashCode` method defined on Java `Object`'s class. The first rule you must follow is that if you override `equals`, you must override `hashCode`. The second rule you must follow when defining `hashCode` is that equal objects must have equal hash codes. A trivial way to meet this constraint is to have hashCode return the same number for all objects as in :

```
@Override
public int hashCode() { return 1; }
```

This `hashCode` definition meets all constraints but is not very useful because it won't help speed up finding objects. Instead, we want to have `hashCode` tend to return unequal hash codes for unequal objects. Below is a formula you can follow to define `hashCode` that results in reasonable hash codes. As before, we present this formula in the context of defining `hashCode` for the `Rectangle` class.

```
public class Rectangle implements Figure {

   private final static int HASH_MULTIPLER = 11; // a prime num

   @Override
   public int hashCode() {
      // Make the code out of the components of the
      // Rectangle
      int code = length;
      code = code * HASH_MULTIPLIER + width;
      code = code * HASH_MULTIPLIER + topLeftCorner.hashCode();
      return code;
   }
}
```

Although this `hashCode` method meets the desired constraints, it does have some problems. If the value of `hashCode` is used to store and retrieve a `Rectangle` object in a collection and the rectangle is moved so that the top left corner point changes value and/or the height and width of the rectangle are modified, the `hashCode` value used to store and retrieve the object will no longer be valid. As a result, this `hashCode` definition may not be desirable for `Rectangle`. Instead, a definition that is based on other less changing fields of `Rectangle` or a static integer value may be preferred.

The article, "Java theory and practice: Hashing it out" (http://www.ibm.com/developerworks/java/library/j-jtp05273.html) provides another perspective on `equals` and `hashCode`.

References and Further Reading

Material in this reading is based on:

[1] Effective Java by Joshua Bloch, Addison Wesley.

Appendix A

This appendix contains partial implementations of `DrawingEditor` and `Drawing` to illustrate how to implement a uni-directional association.

DrawingEditor class

```
public class DrawingEditor {

   private Collection<Drawing> drawings;

   public DrawingEditor() {
       drawings = new ArrayList<Drawing>();
   }

   void addDrawing(Drawing newDrawing) {
       drawings.add( newDrawing ) );
   }

}
```

Drawing class

```
public class Drawing {

  …

}
```

Appendix B

This appendix contains partial implementations of `DrawingEditor` and `Tool` to illustrate how to implement a bi-directional association in addition to the partial implementations provided in Appendix A.

DrawingEditor class

```
public class DrawingEditor {

   private Collection<Drawing> drawings;

   private Collection<Tool> tools;

   public DrawingEditor() {
      drawings = new ArrayList<Drawing>();
       tools = new HashSet<Tool>();
   }

   void addDrawing(Drawing newDrawing) {
       drawings.add( newDrawing ) );
   }

   void addTool(Tool newTool) {
       if ( tools.add( newTool ) )
          newTool.setEditor( this );
   }

}
```

Tool class

```
public class Tool {

    private DrawingEditor editor;

    public Tool() {}

    void setEditor(DrawingEditor theEditor) {
        if ( editor != theEditor ) {
           editor = theEditor;
           editor.addTool(this);
        }
    }

}
```

Drawing class

```
public class Drawing {

   …

}
```