COMP 210 - Lecture Notes - 06 - Containment vs. Extension

January 24, 2017

In these notes we examine a case study in containment vs extension in order to better understand when to use object containment vs class extension.

Square

Let's say we want to add a *Square* class to the shape hierarchy that we've been developing over the past several sets of lecture notes. Clearly squares and rectangles are related and we'd like to leverage that relationship in the design and implementation of the Square class. When trying to leave our existing code as is, we're left with two options: Squares contain rectangles or Squares extend rectangles. A third option would require some refactoring: Rectangles extend Squares.

The first gut check we can perform is against the "is-a" relationship. If we cannot say that a subclass "is-a" variant of the superclass, then extension is not the way to go. This question really isn't a programming question, it's a question against the logic of our application or problem domain. Mathematically, "a square is a rectangle with width equal to height". The other direction doesn't really work. You never her someone refer to a rectangle as a square because rectangles violate parts of the the definition of a square and not the other way around. At this point we should be happy, as the only reasonable extension option is the one the extends what we've already done: a Square *is-a* Rectangle.

Once we're satisfied that an *is-a* relationship makes sense in the problem domain, we need to test the relationship in the programming domain. In programming we require that subclasses be a true extension of the superclass. This means that the behavior, the interface, of the subclass must be at least that of the superclass. Class extension is an additive property. You can override existing behavior or define new behaviors or attributes but you cannot/should not hide behaviors. This is codified in the **Liskov Substitution Principle**[1] which states that instances of the superclass should be replaceable with the subclass without breaking the expected characteristics of the program.

[1] https://en.wikipedia.org/wiki/Liskov_substitution_principle

When we check the Square/Rectangle relationship against Liskov's principle we find a problem. Rectangles have two side length attributes and we'd expect the usual set of getters, setters, and constructors with respect to those two attributes. Squares on the other

hand have a single side length and should really only have half as many getters and setters. It would also be redundant and fraught with problems if the Square class allowed a constructor that initialized two side lengths. So while the problem logic implies class extension, the programming logic doesn't agree. The solution to our problem must be containment.

By using object containment we are committing to designing and implementing the Square as an ABSTRACT DATA TYPE. The rectangle provides all the implementation we need just not in the form we need it. By hiding it away as a private data field, we can map it's interface to the expected interface of a Square. The next question we must address is where should Square fit in out hierarchy. Certainly it should implement Shape, but should it extend AbstractShape?

Extending AbstractShape is problematic. The Square already has access to a pin location through the contained rectangle and extending AbstractShape adds a second pin location. This redundancy could be the cause for error and it's aesthetically unpleasant. On top of this, we added the abstract class as an implementation detail and not to establish a taxonomy of classes. The interface serves the later purpose and we've committed to implementing it with Square. Square simply don't directly need the implementation provided by AbstractShape because it's captured it indirectly through containment. If we're committed to leaving the existing design intact, then we should simply implement Shape with Square and leave AbstractShape out of it.

Our new Shape hierarchy is shown in figure 1. We've lifted the location accessor and mutator up to the interface as it just seems to make sense.

Implementing the Square Class

Implementing the Square ADT class is pretty straight forward. All of the interface methods can be implemented with a call to the equivalent method for the contained rectangle. Figure 2 illustrates this with the *isWithin* method[2].

[2] recall *asrec* is the contained Rectangle

The remaining methods are only tricky in that with the exception of *equals* and *hashcode*, they cannot be auto generated by Eclipse because they're not based on the actual fields of the Square. Figure 3 shows the class constructors. Figure 4 gives the side length accessor and mutator. Finally, figure 5 provides an implementation of *toString* that makes use of the class' own abstract interface rather than working through the Rectangle directly. This final strategy makes the implementation of *toString* more or less independent of the choice of representation of the Square.

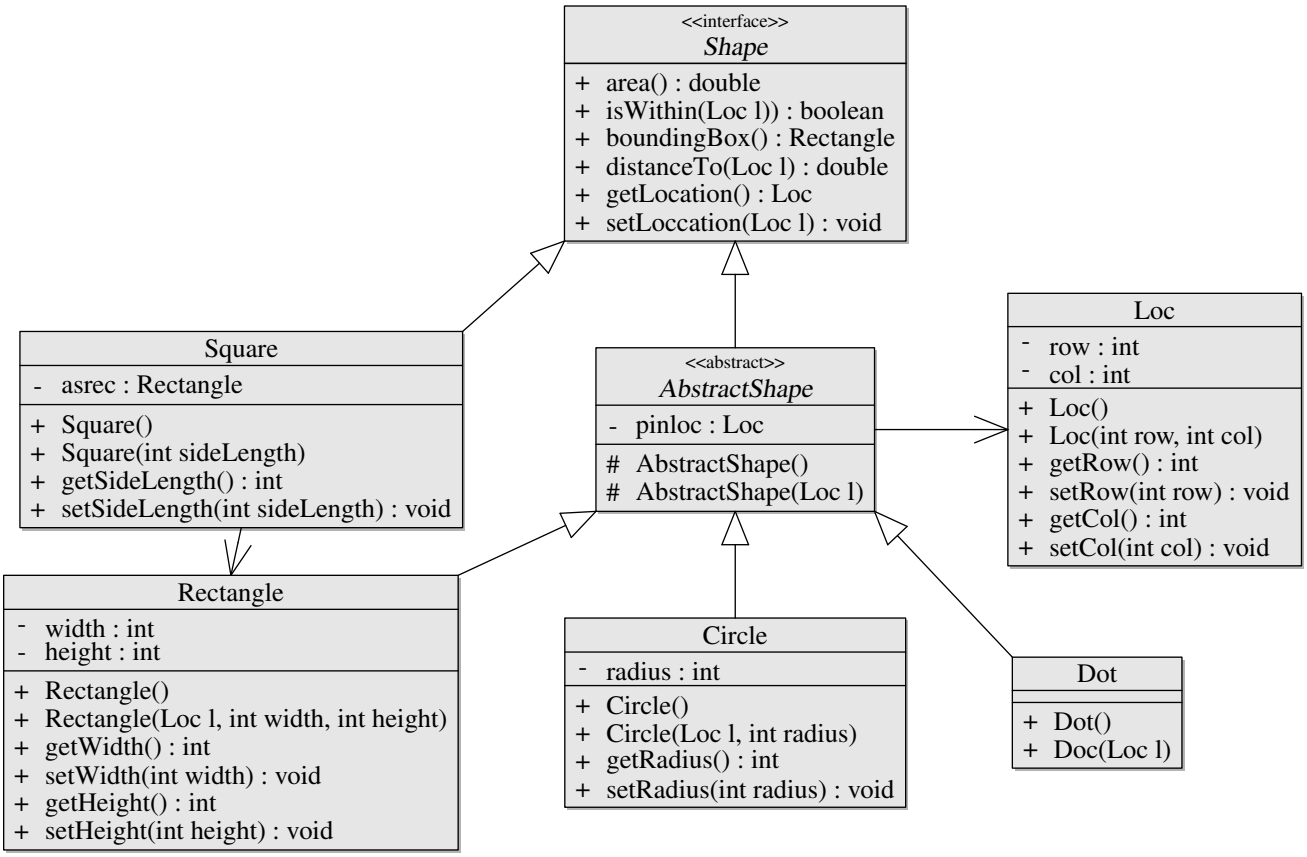

Figure 1: Shape Hierarchy with Square Class

```
public boolean isWithin(Loc l) {
  return this.asrec.isWithin(l);
}
```

Figure 2: The *isWithin* implementation for the Square class

Recap

The desire to reuse existing code and create more generalized, and thereby reusable, code often drive program design. In OOP we get access to a tool that gives us both: class extension. This example illustrates that class extension is not the silver bullet of reusable code design and that it isn't always the right choice, even with the problem domain implies it might be. When extending classes we must always remember that we're not just reusing code, we're building and extending hierarchies of types. Liskov's substitution principle tells us that subclasses must be replaceable with their superclasses. Adhering to this principle means that class extension truly is an extension, it's an additive process. You can add fields, you can add methods, you

```
public Square(){
  // unit square at the origin
  this.asrec = new Rectangle();
}

public Square(Loc pin, int sideLength){
  this.asrec = new Rectangle(pin,sideLength,sideLength);

}
```

Figure 3: The Square class constructors

```
public int getSideLength(){
  return this.asrec.getLength();
}

public void setSideLength(int sideLength){
  this.asrec.setLength(sideLength);
  this.asrec.setWidth(sideLength);
}
```

Figure 4: The Square side length getter and setter

```
public String toString() {
  return String.format("Square [sideLength=%s, location=%s]",
      this.getSideLength(),this.getLocation());
}
```

Figure 5: The Square class toString

can override methods, but you cannot hide or remove methods or fields.

Hiding implementation is the bread and butter of class containment and data abstraction. Initially we use object containment to capture the logical structure of a class. Shapes contain locations because all shapes "have-a" location. In implementing the Square class we utilized a Rectangle not because a square "has-a" rectangle, but because we could implement the entirety of the square interface with a restricted set of the rectangle behavior.