Creating a Class Hierarchy

Suppose we want to create a program that does high school geometry? We first need to identify the types of things that must be modelled in the program:

lines, points, area, polygons, perimeter, triangles, squares, circles, circumference, etc..

How do we organize this into classes?

1. is-a rule: If A is-a B then we make A and B classes where A extends B.

Examples: Square is-a Rectangle.

Square is-a Polygon

Rectangle is-a Polygon

Triangle is-a Polygon

Circle is-a Ellipse

Rectangle is-a Shape

So, the is-a relation suggests the following hierarchy:

Shape

/ \

Polygon Ellipse

/ \ |

Rectangle Triangle Circle

|

Square

2. has-a rule: If A has-a B, then make B a method/methods in A.

Examples: Square has-a area

Polygon has-a numSides

Circle has-a area

Circle has-a circumference

To use inheritance well, we want to move each attribute as high up the hierarchy as we can.

Since all the above classes have an area, we will move area up the hierarchy so that Shape

defines area. If the other classes need to change how area is defined, they can "override" the method.

Likewise, we will move perimeter/circumference up to the top because all shapes have a perimeter.

numSides does not belong in Shape because ellipse and circle does not have sides. Instead, the highest we will move that attribute is into Polygon.

We will place length and height into Rectangle and base and height into Triangle. It would be nice to move them up to a common ancester as well, but

Polygon may not be the right choice. What does length and height mean for an arbitrary polygon?

Abstract Classes:

We do not want to be able to create an instance of "Shape" without specifying what type of Shape.

So we can make Shape be an abstract class.

An abstract class is a class that cannot be instantiated. It is used to enforce common behavior of all its child classes.

In this case, we want to be able to say that polygons, circles, rectangles, triangles, and squares are all "shapes", and all shapes have area.

To get the last behavior, we make area a method of the abstract class.

However, we do not know how to compute the area without knowing what type of shape it is, so we make area an "abstract method".

An abstract method is a method with no body. All classes inherit the abstract method, but unless the class is abstract, it must override the abstract method to give it a body.

(Otherwise, we could call the method without defining what it does!)

public abstract class Shape {

public abstract double area();

public abstract double perimeter();

public boolean isLargerThan(Shape s) {

return this.area() > s.area();

}

An abstract class can containt -everything- that a normal class can (including constructors!), plus it can contain abstract methods.

Note in the above example, we included a non-abstract method isLargerThan. It will work on all shapes because it only accesses the area method of Shape.

Every class that is a Shape (i.e. it extends Shape) will have an area method.

Now, let us make the Polygon class also abstract to demonstrate an abstract class with a constructor. The constructor will set the number of sides to the polygon.

public abstract class Polygon extends Shape {

private int numSides;

public Polygon(int numSides) {

this.numSides = numSides;

}

public int getNumSides() {

return numSides;

}

}

Notice that Polygon does not need to override the area method. It inherits the abstract method, and because Polygon is abstract, it is allowed to have abstract methods.

Now, we will create the Rectangle class. Note that Rectangle is required to have a constructor because the Polygon class does not have a constructor that takes no input (remember how constructors work!).

Since we have to create a constructor, it makes sense to have it also set the length and height.

be able to compute the area correctly. Because Rectangle is not abstract, we must override any abstract methods Rectangle inherits. In this case, it inherits area.

public class Rectangle extends Polygon {

private double width;

private double height;

public Rectangle(double width, double height) {

super(4);

this.base = base;

this.height = height;

}

/\* include getter and setter methods for base and height \*/

@Override

public double area() {

return getWidth() \* getHeight();

}

@Override

public double perimeter() {

return 2 \* (getWidth() + getHeight());

}

}

Note: why did we use "this.width = width" instead of "setWidth()" in the constructor? Because "setWidth()", or rather "this.setWidth()" may not call the setWidth method in Rectangle. It will call the setWidth method

of the true type. As a result, the width field might not get set. To make sure the width and height fields are always set, we need to put the assignment statements in the constructor and not call methods that could

be overridden.

Note that we are using the getter methods inside the body for area. Doing so will make our work for Square (and any other class that extends Rectangle) a lot simpler. Square will not have to change how area is

computed because Square will simply override the getter and setter methods to make sure that a square has the same width as height.

public class Square extends Rectangle {

public Square(double length) {

super(length, length);

}

/\* need to override setBase and setHeight to make sure that this object is always has equal width and height (i.e. it is always a square) \*/

}

There are two ways to do this:

Version 1: Override the setWidth and setHeight methods so that both the width and height are always set the same

@Override

public void setWidth(double width) {

super.setWidth(width);

super.setHeight(width);

}

@Override

public void setHeight(double height) {

this.setWidth(height);

}

Version 2: We will use only the width Rectangle to determine both the width and height of Square

@Override

public void setHeight(double height) {

this.setWidth(height);

}

@Override

public double getHeight() {

return this.getWidth();

}

Which method is better? Neither! They both work the same. However, note that if we had Rectangle's area method use "width \* height" instead of "getWidth() \* getHeight()", then Version 2 would not work!

That does not mean Version 1 is better because there are other things we could have done in Rectangle using fields instead of methods that could have broken Version 1 as well.

IMPORTANT RULE: Outside of the constructors, always use the getter/setter methods instead of the fields when accessing values of an object. This will allow any classes that extend your class to easily override methods to specify necessary behavior.