Effective Use of Abstract Classes and Interfaces

Source	http://javalive.com/modules/articles/article.php?id=17

Imagine you are the Transport Department who is responsible for the cars in the city. You come out with a specification of a car by stating its behavior and properties. I won't specify all the attributes but only two. Car manufacturers can use this specification to manufacture their own versions of cars. You define a car as:

A car is a vehicle where the driver can accelerate and where he can decelerate by braking.

To specify the car you define an interface:

```
interface Car {
    public void accelerate( double amt );
    public void brake( double amt );
    public double getSpeed();
}
```

By your interface definition, the driver can speed up the car by using the `accelerate()` method. The `amt` parameter specifies by what amount the speed has to be accelerated. Then the driver can slow down the car by using the `brake()` method. Again, the `amt` parameter specifies the amount of braking to be applied to slow down. Finally the `getSpeed()` method allows the driver to realize the speed at which he is driving.

After reading this specification, the manufacturer, Porsche comes out with a car that meets the specification you provided as follows:

```
class PorscheCar implements Car {
    private double _speed;           // stores the speed of the car

    public void accelerate( double amt ) {
        if( ( _speed + amt ) > 700.0 ) {
            _speed = 700.0;
            return;
        }
        _speed += amt;
    }

    public void brake( double amt ) {
        if( ( _speed - amt ) < 0.0 ) {
            _speed = 0.0;
            return;
        }
        _speed -= amt;
    }

    public double getSpeed() {
        return _speed;
    }
}
```

The car made by Porsche has a top speed of 700 mph and the car does not go beyond that. The braking also does not take place if the speed drops below 0.0 mph. Porsche meets your car specification at the moment, but they need not follow any rules. For example they could no braking to the car or no speed indication by providing an empty implementation for the **brake()** and **getSpeed()** methods. They can easily violate your specification.

When you are writing code, it is always safe NOT to trust any third-party implementation. You must always be aware that he could easily find ways to misuse your class or bypass your class's invariants to his own advantage. Hence you must take all possible precautions to prevent misuse.

So after Porsche started selling its cars, a large number of accidents are reported. All of them are due to over-speeding. To bring some sort of control and regulation you decide to extend the specifications of a car into two varieties:

1. Civilian car - must have a top speed limit of 200.0 mph.
2. Sports car - must have a top speed limit of 400.0 mph.

You also specify that both cars must have compulsory braking and must have a way for the driver to know its speed. You also may impose higher tax on sports cars and also make stringent rules about giving licenses to sports car drivers. But I am not going to get into those real-world details.

With your new specifications, you bring some invariants into the picture. The invariant for civilian cars are that their speed cannot exceed 200 mph and they must have compulsory braking. and that sports car can have top speed up to 400 mph and also must have compulsory braking.

So how do u enforce these invariants on the manufacturers? Use interfaces?

No, interfaces are only contractual obligations. They do not enforce anything on the implementation. Hence Porsche or any other manufacturer still can violate your specifications. To prevent this from happening, you should use an abstract class. Therefore you specify your new civilian and sports car specifications as follows.

First you define an abstract class which enforces compulsory braking and speed indication:

```
abstract class GenericCar implements Car {
   // stores the speed, notice the use of protected modifier
   protected double _speed;

    // acceleration is still abstract
    public abstract void accelerate( double amt );

    // implement compulsary braking,
    // notice the use of final modifier to prevent
    // overriding.
    public final void brake( double amt ) {
        if( ( _speed - amt ) < 0.0 ) {
            _speed = 0.0;
            return;
        }
        _speed -= amt;
```

```
    }

    // implement compulsory speed indication
    public final double getSpeed() {
        return _speed;
    }
}
```

To analyze the **GenericCar** class, we enforce the compulsory speed indication and braking by providing a default, non-overridable(final) implementation of braking and speed indication. Preventing overriding is important here as we don't want manufacturers to find a loop hole in the specification.

Next, you implement your civilian car specification

```
abstract class CivilianCar extends GenericCar {
    // define a public constant for max speed
    public final static double MAX_SPEED = 200.0;

    /* implement the speed limit invariantfor acceleration to 200 mph
     * see the use of the pre-condition check for the invariant */
    public final void accelerate( double amt ) {
        // the precondition check
        if( ( _speed + amt ) > MAX_SPEED ) {
            accelerateImpl( MAX_SPEED - _speed );    // accelerate to max
speed
            return;
        }
        accelerateImpl( amt );
    }

    // we want manufacturers to implement their own acceleration technique
    // hence this method
    protected abstract void accelerateImpl( double amt );
}
```

This is an interesting class. Here we are implementing the **accelerate()** method but only to enforce our pre-condition of speed limit. We don't want them to override the accelerate method but, we still want manufacturers to implement their own method of acceleration hence, we define the abstract **accelerateImpl()** method which needs to be implemented by subclasses to perform the actual acceleration.

Since the **CivilianCar** extends the **GenericCar** class, the compulsary braking and speed indication invariants are inherited and hence we dont define them again.

Now lets define our sports car specification...

```
abstract class SportsCar extends GenericCar {
    // define a public constant for max speed
    public final static double MAX_SPEED = 400.0;

    /* implement the speed limit invariantfor acceleration to 400 mph
     * see the use of the pre-condition check for the invariant */
    public final void accelerate( double amt ) {
        // the precondition check
```

```
        if( ( _speed + amt ) > MAX_SPEED ) {
            accelerateImpl( MAX_SPEED - _speed );    // accelerate to max
speed
            return;
        }
        accelerateImpl( amt );
    }

    // we want manufacturers to implement their own acceleration technique
    // hence this method
    protected abstract void accelerateImpl( double amt );
}
```

The **SportsCar** class is very similar to our **CivilianCar** class, except that it increases the speed limit to 400.0 mph.

With these two specifications defined, we (i.e. the Transport Department) can now tell the public that they must specify what kind of car they wish to purchase. This choice of theirs could have a bearing on the taxation and pre-requisites to acquire the license to drive the car.

And now our car manufactures must also choose the type of car they make. So lets redefine our **Porsche** car definition to suit the new specification of a sports car:

```
class NewPorscheCar extends SportsCar {

    // notice how we implement the acceleration
    public void accelerateImpl( double amt ) {
        if( ( _speed + amt ) > SportsCar.MAX_SPEED ) {
            _speed = SportsCar.MAX_SPEED;
            return;
        }
        _speed += amt;
    }
}
```

So you see how the invariants are enforced on the new **NewPorscheCar**. Porsche is a good manufacturer hence they have their own check for the speed limit. But a bad manufacturer, who wants to take advantage of our specifications, may intentionally not do so. For example:

```
class BadSportsCar extends SportsCar {

    public void accelerateImpl( double amt ) {
        // observe there are no checks
        _speed += amt;
    }
}
```

The **BadSportsCar** accelerates the car without checking the speed. This attempt of the class will be defeated by its superclass, i.e. **SportsCar** because even if the **BadSportsCar** does not check the speed, the superclass will. Lets see how. I am writing a test program to test our specification:

```
public class TestCarSpecs {
    public static void main( String[] args ) {
        Car newPorscheCar = new NewPorscheCar();
        Car badSportsCar = new BadSportsCar();

        System.out.println( "Now testing NewPorscheCar..." );
        testCar( newPorscheCar );

        System.out.println();

        System.out.println( "Now testing BadSportsCar..." );
        testCar( badSportsCar );
    }

    private static void testCar( Car car ) {
        double accelerateBy = 100;
        for( int ctr = 0; ctr < 10; ctr++ ) {
            car.accelerate( accelerateBy );
            System.out.println( "Car's speed is: " + car.getSpeed() );
        }
    }
}
```

Now when you run this program, you will see that both **NewPorcheCar** and **BadSportsCar** do not exceed 400 mph as per the specifications of a sports car. Even the badly designed **BadSportsCar** obeys the specification speed limits. You may see an output like:

Quote:

```
Now testing NewPorscheCar...
Car's speed is: 100.0
Car's speed is: 200.0
Car's speed is: 300.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0

Now testing BadSportsCar...
Car's speed is: 100.0
Car's speed is: 200.0
Car's speed is: 300.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
Car's speed is: 400.0
```

This is how we can use interfaces and abstract classes to our advantage and make safe designs possible. Here is a list recommendations I strongly suggest you consider when designing your projects:

1. Interfaces must be re-inforced by abstract classes.
2. All implementations MUST be accessed via their interfaces.
3. Special care must be taken to recognize and enforce type invariants when designing the class hierarchy.
4. Never trust third-party implementations. Always safe-guard your types.
5. Always remember to clearly document details about type invariants and how they should be enforced via JavaDocs.

~~~ End of Article ~~~