Interface Based Programming!

Venkat Subramaniam
venkats@durasoftcorp.com
http://www.durasoftcorp.com

Abstract

When I ask people why they want to use the object paradigm, invariably one of the top answers is *Reusability*. Yet, how much of the code we write is reusable? Are we writing code that lends itself to reusability? In this article, we will talk about how interface based programming truly helps us to develop reusable and extensible code.

An assignment for me

Let's say I am asked to write a program that will monitor a door. If the door is left unattended for a period of 10 seconds, the program needs to raise an alarm. This door may be used in a secure facility like a Bank or an office where it should be monitored very closely.

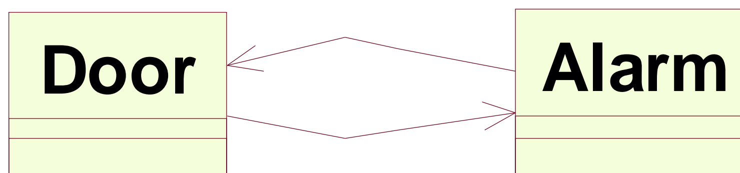
This should not be a difficult task. Throw-in some quick Java code with multithreading, and the job is done. Here is a sample of the code that will do just that.

```
//Door.java
public class Door
{
        private boolean closed = true;

        public void open()
        {
                closed = false;
                // Interface with door API at this point.

                Alarm anAlarm = new Alarm(this, 10);
        }
        public void close()
        {
                closed = true;
                // Interface with door API at this point.
        }

        public boolean isClosed() { return closed; }
}

// Alarm.java
public class Alarm
{
        public Alarm(final Door aDoor, final int seconds)
        {
                Thread monitoringThread = new Thread(
                        new Runnable()
                        {
                                public void run()
                                {
                                        try
```

```
                                        {
                                                Thread.sleep(
                                                        seconds * 1000);
                                                if (!aDoor.isClosed())
                                                        raiseAlarm();
                                        }
                                        catch(Exception e)
                                        {
                                                // return if thread
                                                // interrupted. Other
                                                // actions may be
                                                // necessary, however...
                                                // not the focus of
                                                //problem here.
                                        }
                                }
                        }
                        );
                monitoringThread.start();
        }

        public void raiseAlarm()
        {
                // Real code to play alarm goes here.
                // As a sample to illustrate, I am poping a dialog.
                javax.swing.JOptionPane.showMessageDialog(null,
                                        "Door left open, alarm sounded");
        }
}

// Tester.java
import javax.swing.*;
import java.awt.event.*;

public class Tester extends JFrame
{
        private JButton doorButton;
        private Door theDoor;

        public void frameInit()
        {
                super.frameInit();

                theDoor = new Door();
                doorButton = new JButton("Open Door");

                getContentPane().add(doorButton);

                doorButton.addActionListener(
                                new ActionListener()
                                {
                                        public void
                                                actionPerformed(
                                                  ActionEvent actionEvent)
                                        {
                                                if (theDoor.isClosed())
                                                {
```

```
                                                    theDoor.open();
                                                   doorButton.setText(
                                                        "Close Door");
                                            }
                                            else
                                            {
                                                    theDoor.close();
                                                   doorButton.setText(
                                                         "Open Door");
                                            }
                                    }
                            }
                            );
        }

        public static void main(String[] args)
        {
                final JFrame frame = new Tester();

                frame.setBounds(100, 100, 500, 400);
                frame.setVisible(true);

                frame.setDefaultCloseOperation(DISPOSE_ON_CLOSE);

                frame.addWindowListener(
                        new WindowAdapter()
                        {
                                public void windowClosed(WindowEvent e)
                                {
                                        System.exit(0);
                                }
                        }
                        );
        }
}
```

This program will display a Dialog with a Door Open button. Click on it and wait for a little over 10 seconds, then you will see an alarm message being displayed. This code works and we can always tune it to the specific door access API that may be available for us to use. However, let’s take a minute to look at the class diagram for this code.



The first observation is that there is a cyclic dependency between the two classes. Further, the code fails Open-Closed Principle[2] as well. A couple of months later, let’s say someone working within the same organization or group is asked to monitor say a Reaction (some thing like a chemical or nuclear reaction). If the reaction is not under control within certain duration of time, then an alarm needs to be raised. Wouldn’t it be nice if this person can reuse my Alarm?

We should be able to give this Alarm code to any developer who has a need for it. However, this developer will then have to modify the code of the Alarm to make it work with the Reaction. Of course, while doing this, if this developer calls me and says there is some problem with the code, I would find that her code is different from my code. She changed my code, sorry no support for her!

Another alternative is to use the Alarm class without changing the code. This may be achieved by inheriting the Reaction from the Door! That hurts. It is very poor modeling, some thing that is purely done in order to fit what is available in the existing code.

When I first set out to write the code (and when you first set out to read it), it should have occurred to me that after all an Alarm is monitoring the Door, and it is interested in a small part or detail of the door and not the door in its entirety. If the coupling of the Alarm to the Door can be removed, then the Alarm would become more reusable. **Higher coupling leads to lesser reusability in a system.**

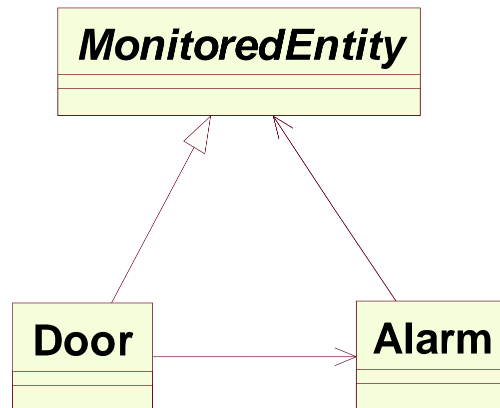
Alarm application another shot

Let's try to break the coupling between the Alarm and the Door. One way to do that is to abstract out, into an interface, the details that the Alarm is interested in the Door. How about creating an interface called MonitoredEntity for this purpose. A MonitoredEntity will simply abstract and represent an object that can be monitored by an alarm. Here is the code after this change.

```
//MonitoredEntity.java
public interface MonitoredEntity
{
	public boolean isOK();
}

// Door.java
public class Door implements MonitoredEntity
{
	private boolean closed = true;

	public void open()
	{
		closed = false;
		// Interface with door API at this point.

		Alarm anAlarm = new Alarm(this, 10);
	}
	public void close()
	{
```

```
		closed = true;
		// Interface with door API at this point.
	}

	public boolean isClosed() { return closed; }

	public boolean isOK() { return isClosed(); }
}

// Alarm.java
public class Alarm
{
	public Alarm(final MonitoredEntity theMonitored, final int seconds)
	{
		Thread monitoringThread = new Thread(
			new Runnable()
			{
				public void run()
				{
					try
					{
						Thread.sleep(seconds * 1000);
						if (!theMonitored.isOK())
								raiseAlarm();
					}
					catch(Exception e)
					{
						//…
					}
				}
			}
			);
		monitoringThread.start();
	}

	public void raiseAlarm()
	{
		//…
	}
}
```

The result of this code change is shown in the following class diagram:

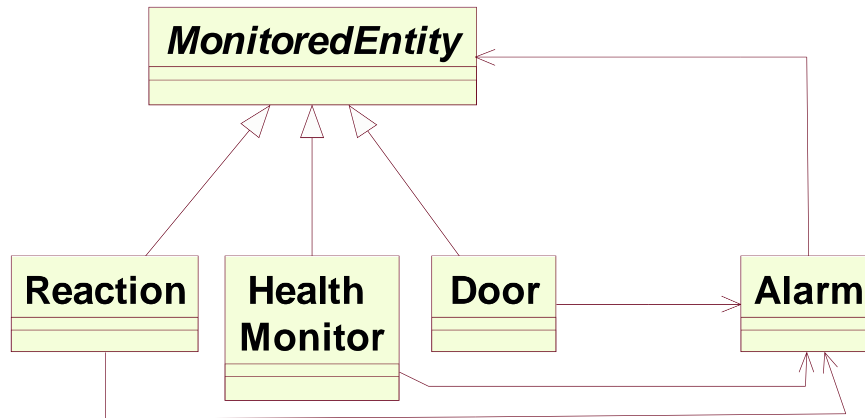


Note that the cyclic dependency has been removed now in this case. While the Door depends on the Alarm, the Alarm depends on some thing that is abstract and not concrete. Instead of the Alarm depending on the concrete class Door, both the concrete classes (Door and Alarm) now depend on some thing abstract (MonitoredEntity). This concept is addressed in the Dependency-Inversion Principle[3].

If some one wants to use the Alarm in their application, then we now have to give them only two files – the Alarm.class and the MonitoredEntity.class (or one jar with these two files in it). The source code does not have to be distributed, and so there is no risk of that being modified. The developer who wants to use the Alarm can simply implement the MonitoredEntity interface for the Reaction class. Now, Reaction is a MonitoredEntity, which sounds a lot better and more sensible than Reaction is a Door.

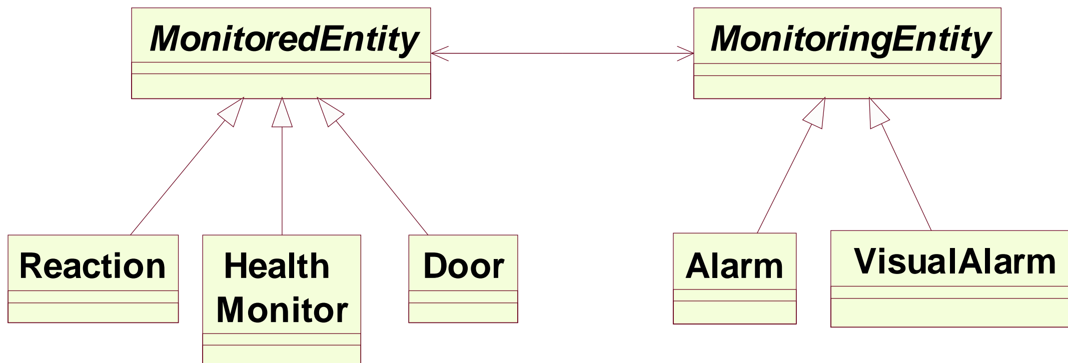
One logical question here is what if the Reaction is an existing class and you can't modify it to implement the MonitoredEntity interface? The Adapter Pattern[1] comes to your rescue here. You may write an object adapter that will implement the MonitoredEntity interface and it will route the call isOK() to the Reaction class (the adaptee). Similarly, the adapter can take care of properly invoking the Alarm when needed.

The UML class diagram in this case looks like this:



Alarm application final shot

Now that we have removed the cyclic dependency and made the Alarm more reusable, what can we do further? The Door (and the Reaction) depends on the Alarm. What if we decide to use another kind of alarm, say a visual alarm instead of an audio alarm? One way to fix that is to let the Door depend on some thing abstract as well. Let's call this a MonitoringEntity. The resulting model will look like the following:



Of course, originally the Door was creating an object of the Alarm class. Now we have both Alarm and VisualAlarm. How would the Door then create an object of the Alarm, if it does not know which one to use? This can be solved using the Abstract Factory Pattern[1]. It is even easier to achieve this in Java than in C++. The concept of reflection allows us to create an object of a class, given its class name. Here is how the Door's open method may be implemented:

```
public void open()
{
        closed = false;
        // Interface with door API at this point.
```

```
		try
		{
			Class theMonitoredEntityClass =
				Class.forName(
					System.getProperty("TheMonitoringEntity"));
			MonitoringEntity monitor =
				(MonitoringEntity) theMonitoredEntityClass.newInstance();
			monitor.setMonitoredEntity(this);
			monitor.setMonitoringTime(10);
			monitor.startMonitor();
		}
		catch(Exception e)
		{
			// Handle this one as appropriate
		}
	}
```

Note that the Door is getting the name of the Alarm class (like VisualAlarm, AudioAlarm, Buzzer, etc.) from a system property called TheMonitoringEntity. It then creates an object of that type using the newInstance() method of the java.lang.Class class.

Is there a limit – a word of caution

One tendency for people who are getting into object modeling is to over abstract the system. They start building a hierarchy of classes and over do it. The result is a system that is very complicated and difficult to maintain. Surely, the objective of OO paradigm is not to complicate the system and render it difficult to work with. One has to apply one's judgment in developing an object model. Providing extensibility and building a hierarchy should come from anticipating the changes in a system. One has to evaluate the likely hood of such changes in the requirements or use/reuse of classes. Planning for extensibility without these in mind would lead to inefficiency as well.

Conclusion

Reusability does not come from merely using an object-oriented programming language. It comes from better modeling. A better system is one that has a layer of abstraction and a layer of concreteness. Dependencies should generally run vertically from the concrete layer to the abstract layer, and not horizontally between concrete classes.

References

1. Erich Gamma, et. al., *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, Boston, MA, 1994.
2. Robert C Martin, *The Open-Closed Principle*, C++ Report, 1996. http://www.objectmentor.com/resources/articles/ocp.pdf.
3. Robert C Martin, *The Dependency-Inversion Principle,* C++ Report, 1996. http://www.objectmentor.com/resources/articles/dip.pdf.