ISSUE 36 | JULY 2010

SOCIETY OF ACTUARIES Technology Section

CompAct

ELECTRONIC NEWSLETTER

CONTENTS

QUICK LINKS

Technology Section Web site

Share Print Article Search Back issues

SOLID OBJECT ORIENTED DESIGN

by Andrew Chan

A lot of systems were developed using an Object Oriented (OO) programming language, e.g. C++, Java, C# or Visual Basic .Net. Why did so many system developers choose to use OO programming languages? What are the benefits of OO programming languages?

Without proper object oriented design (OOD), OO programming languages do not offer any significant advantages. To illustrate, consider the following scenario:

Let's say we had an old actuarial system that we developed using Visual Basic 6 that had 20 program files. At some point, we decided to migrate to the latest Visual Basic.Net and create 20 classes. Moreover, since we couldn't think of a good name for each class, we simply named them actuar01, actuar02 ... actuar20, the same name as each of the program files. We then copied the content of each program file into the corresponding class. In doing this, we created a system that uses OO programming language. But what benefits does our system have from this migration? Not a lot unfortunately!

In order to realize the true benefits of an OO programming language, we must understand the OO concepts and principles.

Object Oriented Concepts

There are a few object oriented concepts:

- Abstraction

- Encapsulation
- Inheritance
- Polymorphism

You can find some very good definitions and explanations from Wikipedia. I have posted the hyperlinks and highlighted the summary for your reference.

Abstraction

Abstraction is "the mechanism and practice of abstraction reduce and factor out details so that one can focus on a few concepts at a time." Since abstraction extracts key characteristics of an object and hides other immaterial complexity, your readers should easily understand and visualize what we want to discuss.

"I just bought a new Samsung 46-inch 1080p 120Hz LCD TV," my friend told once told me. Size, number of lines, refresh rate and type are the key characteristics of an HDTV. He can go on to tell me about its physical dimension, weight, power consumption, etc., but most people don't care, and would prefer being told "I just bought a new TV," abstracting the technical specifications to hide unnecessary details.

Encapsulation

Encapsulation is "the process of compartmentalizing the elements of an abstraction that constitute its structure and behavior; encapsulation serves to separate the contractual interface of an abstraction and its implementation."

Encapsulation allows programmers to use any method without understanding the details of implementation. This can drastically reduce the learning curve or maintenance effort of an actuarial system.

For example, how many actuarial programmers understand the implementation of ADO .Net? With ADO .Net, most of us can learn in a couple of hours how to write a simple function to retrieve data.

If your actuarial system consists of 1,000+ classes, do you want all your actuarial developers to understand every single class and every method within? It would take forever to train a new actuarial developer. Encapsulation will shorten the learning curve of actuarial programmers and will provide better system manageability and stability.

Inheritance

Inheritance is "a way to form new classes (instances of which are called objects) using classes that have already been defined. Inheritance is employed to help reuse existing code with little or no modification."

When you derive a new reserve calculator from its base class, you only have to implement the new features. It can save you enormous development, checking and testing time.

Inheritance also increases overall system stability and reduces quality assurance effort. It allows developers to reuse code, enhance and modify existing class.

Polymorphism

"Polymorphism in the context of object-oriented programming, is the ability of one type, A, to appear as and be used like another type, B."

Inheritance is required in order to achieve polymorphism.

Polymorphism can greatly simplify coding and allow extensibility for future enhancements. With polymorphism, you can call 10 different reserve calculators from a single line of code. When you want to add five more new reserve calculators, you don't need to modify the calling function.

SOLID Object Oriented Principles

- Single Responsibility Principle
- Open Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Single Responsibility Principle (SRP)

"There should never be more than one reason for a class to change."–Robert Martin, SRP paper linked from The Principles of OOD.

Simple is beautiful. Each class should have only one responsibility and focus to do one single thing.

Your reserve calculator classes are already very sophisticated. If you also implement policy projection, decrement calculation and cashflow projection within it, then it would be huge and all your actuarial programmers may always work on this big class together.

SRP would promote the reuse of code, clarity and readability. Your system would also be easier to test, enhance and maintained. Developers would also find less contention for source code files.

Open Closed Principle (OCP)

"Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification."–Robert Martin paraphrasing Bertrand Meyer, OCP paper linked from The Principles of OOD.

Most of us work on existing systems rather than build new systems from scratch. When you add new features to a system, you often feel more comfortable adding new functions than modifying an existing codebase. Why? You worry that your modifications would accidentally add new bugs to the systems, especially fragile systems. OCP recommends extending existing codebase, not modifying it.

If you already have 10 different reserve calculator classes, adding a new one should not modify any existing code.

Once you have followed SRP to build your system, it would be easier to implement OCP. Systems following OCP are often more stable because existing code does not change, and new changes are isolated. Deployment is also faster because existing features would not be accidentally modified.

Liskov Substitution Principle (LSP)

"Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."–Robert Martin, LSP paper linked from The Principles of OOD

This principle is just an extension of the Open Close Principle, and it means that we must make sure that new derived classes are extending the base classes without changing their behavior so that the derived classes must be completely substitutable for their base class. Otherwise the new classes can produce undesirable effects when they are used in existing program modules.

Below is a classic example of LSP:

```
public class Rectangle
{
protected int _width;
protected int _height;
public int Width
{
get { return _width; }
}

public int Height
{
get { return _height; }
}

public virtual void SetWidth(int width)
{
_width = width;
}

public virtual void SetHeight(int height)
{
_height = height;
}

} public class Square: Rectangle
{
public override void SetWidth(int width)
{
_width = width;
_height = width;
}

public override void SetHeight(int height)
{
_height = height;
_width = height;
}

}

[TestFixture]
public class RectangleTests
{
[Test]
public void AreaOfRectangle()
{
Rectangle r = new Square();

r.SetWidth(5);
r.SetHeight(2);

// Will Fail - r is a square and sets
// width and height equal to each other.
Assert.IsEqual(r.Width * r.Height,10);
}
}
```

Square class is derived from Rectangle class; so C++ allows a Square object to be cast into a Rectangle object. However, Square class has its own setter functions; so r.SetWidth and r.SetHeight would set width and height equal to each other. What do you expect r.Width * r.Height to be equal to? Is r a Rectangle or Square object?

LSP would make the system easier to test and provide a more stable design.

Interface Segregation Principle (ISP)

"Clients should not be forced to depend upon interfaces that they do not use."–Robert Martin, ISP paper linked from The Principles of OOD

Again, it is another "simple is beautiful" principle. We should have multiple slim interfaces rather than a giant interface. Each interface should serve one purpose only.

If you have both policy month and calendar month projections, put them in two separate interfaces. For example, use IPMCashflowProj and ICMCashflowProj rather than just one interface named ICashflowProj.

With ISP, design would be more stable and flexible; changes are isolated and do not cascade throughout the code.

Dependency Inversion Principle (DIP)

"A. High level modules should not depend upon low level modules. Both should depend upon abstractions.

B. Abstractions should not depend upon details. Details should depend upon abstractions."–Robert Martin, DIP paper linked from The Principles of OOD.

Low-level classes implement basic and primary operations, and high-level classes often encapsulate complex logic and rely on the low-level classes. It would be natural to implement low-level classes first and then to develop the complex high-level classes. This seems logical as the high-level classes consume low-level classes.

However, this is not a flexible design. What happens if we need to add or to replace a low-level class?

If your reserve classes (high level) contain cashflow classes (low level) directly, and you want to introduce a new cashflow class, you will have to change the design to make use of the new cashflow class.

In order to avoid such problems, we can introduce an abstraction layer between the high-level classes and the low-level classes. Since the high-level modules contain complex logic, they should not depend on the low-level modules. The new abstraction layer should not be created based on the low-level modules. The low-level classes are created based on the abstraction layer.

Once you implement DIP, your actuarial system will be significantly easier to extend, and deploying new features will take less time.

Conclusion

Please keep in mind that all of these principles are just guidelines that would make your system more stable, easier to maintain and enhance; but they are not ironclad rules. You must apply your own judgement and experience.

Andrew Chan can be contacted at chanpangchi@rogers.com.