ISSUE 34 | JANUARY 2010

SOCIETY OF ACTUARIES Technology Section

CompAct

ELECTRONIC NEWSLETTER

CONTENTS

QUICK LINKS

Technology Section Web site

Council

Links of Interest

Share Print Article Search Back issues

COMMON SOFTWARE DESIGN ISSUES

by Andrew Chan

When I developed my first business system 20 years ago, I was very proud that I took a business process needing multiple man-months to run and managed to migrate it to an automated computer process requiring only a few key strokes to execute.

However, my first business application had a lot of software design issues which made the system difficult to debug and maintain. Improper software design could reduce your development team's productivity and your actuarial system could become difficult to adapt to competitive market and statutory requirements.

I am going to discuss some of the more common design issues such as: blob class, copy–and–paste programming, spaghetti code, and Swiss army knife classes. I'll review their characteristics, problems and explain how you can avoid them.

Blob Class

The blob class is also known as the God class. It is a huge class that contains many attributes and operations. It is not a cohesive object and it often tries to serve multiple purposes. Its existence eliminates all of the object oriented design benefits, so it is difficult to reuse, modify and test. It can also compromise the scalability and performance of your system since it is huge and consumes a lot of system resource even for simple operations.

Copy–and–Paste Programming

This was one of my favorites 20 year ago, and describes the process

Fiction Contest

Howard Callif, Editor

SOA Staff
Meg Weber, Staff Partner

Sue Martz,
Section Specialist

Sam Phillips, Staff Editor

POLL

What do you think of the new CompAct format?

[✓ I love it!]
○ It's better than the printed version
○ The print version was better / more convenient
○ I'm not sure - what's CompAct?

View Results
Share This
vote

of copying code from one project to another as a form of "reuse." It is simple and easy to operate this way until you want to add a new feature or fix a bug. You have to spend a lot more effort to get the same work done across the whole system (i.e. identify the bugs, add new features, code review, and test), because you need to modify each system independently. Not to mention inserting useful comments in the right places every time—providing you can find all the pasted code!

Spaghetti Code

Spaghetti code has many forms, I will just go through some of the common ones:

- Long function—if you have the opportunity to read a function with more than 1,000 lines (believe me, they do exist), by the time you are half way through you may forget what it is supposed to do.
- Mystery variable name or condition—x1, x2, x3 or if (prem > prem1 && prem2 + prem3 > prem4). I wonder if anyone, including the original developer, can remember what they represent.
- Global variables–which seem very convenient! You can use or change a global variable anywhere, anytime. However, when you have to maintain or debug your system, and you have to find out why its value is suddenly changed and where the change comes from, the problem becomes clear. You may have similar problems if member variables in your classes are public. Note that this is also not scalable for classes that are used in Web systems.
- GOTO/BREAK—it is a developer's nightmare that the code suddenly jump from one place to another or just stop in an loop or break out from the function.

Spaghetti code is very difficult to reuse and update. Future maintenance can be costly, and fixing a bug may generate more bugs. In the end, you may find your development team spends more time fixing bugs than adding new features. It would eventually reach the point of diminishing returns to update or add to a system.

Swiss Army Knife

Swiss Army Knife class typically has a lot of data, functions and interfaces. It tries to provide a solution to every possible use of the class. Again, whenever something is big and complex, it becomes difficult to understand, modify and debug.

Solution

There is no simple solution. You have to understand SOLID object oriented (OO) design principles.

A class should be cohesive and have a single, clearly stated responsibility. If it has more than 60 attributes and operations then it is time to examine your class and refactor it. Simplicity is beauty!

It is every developer's job to constantly review and optimize the code. Refactor the code once a defect is identified. Eliminate global variables and write accessor functions for member variables. Remove obsolete code. Classes, functions, data types and variables must have meaningful names.

Every time you read or modify a piece of code, you have to understand what it does and ask yourself if it makes sense. Comments in code should explain what is being done, and why.

I will discuss SOLID OO design in more detail and demonstrate how modern development tools can help you refactor the code in another paper.

Andrew Chan is an independent consultant, and can be reached at andrew.chan@actuariallink.com