CSIS 3701:Advanced Object-Oriented Programming

Object-Oriented Unit Testing

Introduction

In the previous section, we put a great deal of emphasis on **decomposing** a complex project into simpler modules, each of which can be implemented by an individual programmer/group, with a minimum of communication after the initial design. Of course, an important part of implementing an individual module is **testing** it – your individual modules should be **completely debugged *before*** integrating it with the other modules. This is called **unit testing**.

Unit Tests

Glass Box Testing

Glass box testing is based on the **structure** of the methods. These are the kind of tests that you probably performed in 2610, such as:

- At least one test case for each *branch*.
- Several tests for each *loop*.

These kinds of tests are often also based on your **internal representation**. For example, if you are testing methods in the `Student` class, and you are using some sort of list to store the courses, then you should probably test case where there are *no* courses in the list, *one* course in the list, and *many* courses in the list.

These test cases will be created by the **developer**, as they are based on the **internal representation** of the methods and class.

Black Box Testing

Black box testing is based on the **requirements** for the class – that is, what the class methods are required to do by **other classes**.

The best source of back box tests are the **scenarios** created during the design and specifications. There should be at least one test case for each scenario in which the method takes part.

For example, the following types of tests should be applied to the `addCourse` method of the `Student` class:

Purpose	State	Call	Desired Result
successful add	course 0632 open, not in list	`addCourse(0632)`	0632 added to list nothing returned or thrown
course closed	course 0633 closed, not in list	`addCourse(0633)`	CourseClosedException thrown, no state change
already in course	course 0632 open, in list	`addCourse(0632)`	AlreadyInCourseException thrown, no state change
no such course	no such course as 0001	`addCourse(0001)`	NoSuchCourseException thrown, no state change

These tests are often developed by the **users** of the class. Since they are the ones who depend on those methods, they are often the best to determine when all of the requirements are met. These tests are often developed during the **design and specification** stage, often directly from the ***scenarios*** created during that stage.

Test Harnesses

As mentioned in the section on Design and Debugging, we often have to write code to test our classes (since they generally can't run on their own). These pieces of code are often referred to as *test harnesses*. We have already seen a couple of these:

- **Debugging inspectors**, which print the *state of the object*, usually to standard output.
- **Class drivers**, which execute all constructors and methods, under as many circumstances as possible.

When using any type of debugger, it is a good idea to make a list of tests (like that above) in order to make sure that all requirements are tested. It is also necessary **to retest all test cases after any modification** in order to make sure the modification had no adverse affect on any of the code (this is called *regression testing*).

Stub Classes

Another major problem is how to test methods which contain calls to methods **in other classes** being created by **other developers** – methods we currently have *no access* to. For example, consider the `Student` class again, this time from the point of view of the developer of the **`StudentInterface`** class. Their code will contain calls to the methods `addCourse` and `getCourses`. How can they test (or even compile) their code, given that some other developer is currently writing those methods?

In order to debug code that contains calls to methods which are not available, we will need to create "temporary" classes and methods to take their place. These are usually refered to as "**stubs**".

Stub *functions* are often used in place of functions called by a function we are currently testing. For example, in C++ if we are testing the function `f1`, which includes a call to the (currently unavailable) function `f2`:

```
void f1(int x) {
   ...
   f2();
   ...
   }
```

We might test it by creating a stub version of `f2` which does nothing:

```
void f2() {}  // stub function
```

Most of the time, however, we have it print *diagnostic messages* to help with the debugging -- in particular, to confirm that `f1` correctly calls `f2`:

```
void f2() {cout << "f2 called\n";}
```

If `f2` takes any **input parameters**, we usually print them as well:

```
void f2(int a, int b) {
   cout << "f2 called with  " << a << " and " << b << "\n";
   }
```

And finally, if `f2` **returns** or **throws** anything, we can either return a dummy value, or **prompt** for the value to be returned:

```
int f2(int a, int b) {
   cout << "f2 called with  " << a << " and " << b << "\n";
   int c;
   cout "What value should f2 return: ";
   cin >> c;
   return c;
   }
```

In order to debug a *class* which sends messages to other objects, we will need to create a **stub class** for those objects, in order to create "temporary" objects for the purpose of testing. Such a stub class must contain:

- A (very simple) constructor.
- Stubs for each method that will be invoked by our class.

This stub class will probably *not* contain any state variables.

For example, consider once more the `StudentInterface` class. Suppose that we wished to test and debug our code for it, but had no actual code for the `Student` class. We will have to create a stub class which takes its place, giving stub methods for the `addCourse` and `getCourses` methods.