Thoughts on Testing

Like the design process, testing offers us two views of our system: one from the outside and one from the inside. When we begin a design, we take an outside, top-level view of the system. That view (must be) is through the customer's eyes. We then move progressively to the inside and to an increasingly detailed view as we develop the system. When we test, we do the opposite. We begin on the inside with a detailed view and move to the outside, top-level view.

Design should be inherently top down; testing should be inherently bottom up. We make certain that the lower level, building block modules work and are tested first. We then integrate them into working subsystems and test again. We repeat these steps until all of the system modules and subsystems are integrated and tested.

When we are testing, our goal for the inside view is to try to ensure that we are designing a safe, reliable, and robust system that meets all of its specifications. Our goal for the outside view is to make certain that we did a good job with the inside view.

Testing is perhaps one of the more neglected yet most important parts of design. Testing should not begin, nor should it be done, as an afterthought when a design is completed. It should be an integral part of every phase of the development cycle. It should be considered as part of the formulation of the original requirements, incorporated into the formal specification, and implemented during the detailed design.

An Inside View

In this class, we do not have the time to be able to execute either the outside or inside views of the test process with the same thoroughness that we would in industry. None-the-less, this does not diminish their importance. As a result, in the scope of our work, we will limit our focus to several basic aspects of the test process. For our inside view we will incorporate simple *type conformance*, ensure the proper functionality of the design over the *range* of and at the *boundary values* for the input data, and manage dynamic memory allocation.

Type Conformance

In the context of this discussion, we are interpreting *type conformance* to mean ensuring that the types of variables entered as inputs to stand alone functions or to class member functions are consistent with and compatible with the types specified in the function's signature – the number, *types*, and order of its arguments and with the implemented function body. For our present work, we will relegate most of the testing to our compiler. Among other things, this means not setting the warning threshold of the compiler so high that it never bothers us. With the proper levels set, when or if warnings are generated, we must analyze them, identify the cause of each, then either correct the problem(s) or determine that it (they) can be safely ignored….this does not mean just saying "oh yeah, I always get a bunch of warnings".

The compiler can be a very powerful tool. Take advantage of it. Remember, what we do here is only the tip of the iceberg.

When we study class inheritance, we will learn that we can pass child class instances as arguments to functions that may have been written with a parent class type as one of the arguments. While syntactically correct and legal, we must ensure that when doing so, the intended behaviour of such a function is consistent with and valid for the designed behaviour of the child.

Range and Boundary Values

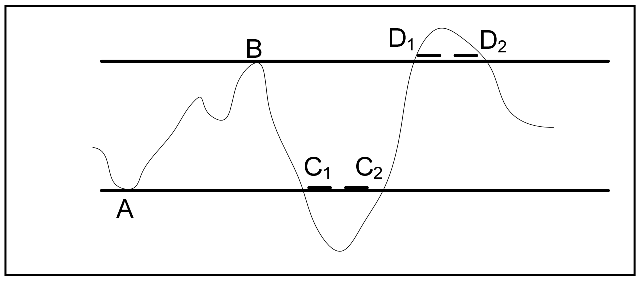
One of the essential steps during the early phases of a design is defining the interface inputs and outputs for and amongst the system functions then for the comprising modules. This process begins when we develop the use cases and extends through the definition of the public interfaces for each of the classes. It involves specifying the *type* and the *range* of each input or output variable. As each of those decisions is made, we must further consider and evaluate how the system will behave / respond if the input data either does not match the expected type or exceeds the specified range. In addition to deciding what to do if the data value exceeds the specified range, we must also consider the proper course of action if (and when) the data magnitude returns to the proper values.

The problem can be addressed from during the design process and at runtime. As part of the design, we must ensure that the input data values are in bounds prior to using them in any calculations or before making any irrevocable decisions – the test code for ensuring that such constraints are met is incorporated at design time and used at runtime. Further, we must decide how to treat out of bounds values should they occur at runtime.

Under no circumstances should an out of bounds piece of data ever be mapped into one that is in bounds or valid without warning. That said, the boundary tests must incorporated into the runtime code to confirm that input data has values (and types) that are within the specified upper and lower bounds. Further, we need to decide what to do if such bounds are exceeded. Alternatives include,

- Hold at max or min value
- Alarm
- Combination

The following diagram illustrates a typical variable and a specified range or limits on the values of such a variable. At points A and B, the signal is just at the low and high boundaries respectively; at these two points, the variable's value remains exactly in range.



At C_1, the value exceeds the specified range in the negative direction and continues decreasing. It reaches a negative peak and begins increasing, crossing the lower bound at C_2. The increase continues until the upper bound is crossed at D_1. The behavior at the lower bound is repeated at the upper and eventually the signal crosses back into the specified range at D2.

The variable's values at the points A and B are valid and must be accepted as proper inputs. The design and subsequent tests should confirm that behavior. The values at points C and D, however, exceed the specifications. At both sets of points, we have several choices:

- Ignore the out of range values.
- Detect the out of range values, issue a warning, and continue operating. When or if the value returns to the proper range, continue operation with annunciation of the original fault.
- Detect the out of range values, issue a warning, throw an exception, then try to correct the problem.
- Detect the out of range values, issue a warning, and terminate operation.

There are also several choices as to what values to assign to the data at these points.

- Accept the actual value of the data
- Map the actual value into the maximum or minimum value but do not use the value other than for annunciation.

Whatever the choice, the system must be thoroughly tested at such values to ensure proper operation according to specification. Testing at a typical or a single value of the input is *not* sufficient.

As an example, consider a design that utilizes a linked list as a container. There are five positions within a linked list that are important: the three in bounds values (position 0 – the head of the list, position (n-1) – the tail, and any position between the head and the tail) and the two out of bounds values (a negative position and a position beyond the tail of the list). The design must test for and properly handle attempts to insert, delete, and search for both of the out of bounds indices and function properly for the in bounds cases. Neither of the out of bounds cases should ever be mapped into a valid in bounds value.

Working with Dynamic Memory

In the C++ language, we allocate memory dynamically using one of the forms of the new operator. If the allocation succeeds, new will return the address of the newly allocated memory; if it fails, it will either return a NULL or throw an exception. Either of these failure cases, must be handled.

For this class, we will not be working with exceptions; we will cover these in later classes. However, this does not mean that we do not have to worry about memory allocation failures. Thus, each time that we use the new operator, we must invoke it with the *nothrow* constant, for example.

```
char* pptr = new (nothrow) char;
```

This will ensure that if the allocation fails, an exception will not be thrown and that a NULL will be returned instead. Before proceeding, whenever we use the new operator, we must then check for NULL as follows …

First, include the library cassert into your program...

```
#include <cassert>
```

then, elaborating on the above example, write either

```
    char* pptr = assert (new (nothrow) char);
```

or simply

```
    char* pptr = new (nothrow) char;
    assert (pptr);
```

If the argument to assert *does not* evaluate to NULL, program flow will continue undisturbed; otherwise the program will terminate.

Bear in mind, the assert macro is a debugging tool, we do not want to use it in production code...right now, we are only using it temporarily until we learn exceptions.

An Outside View

As we noted earlier, our goals for the outside view of testing are two: first and foremost, to make certain that our design meets the customer's specifications and secondly to make certain that we did a good job in satisfying the goals of the inside view. As with the inside view, time constraints limit the scope of coverage of the testing process that we can incorporate into this certificate. The following, then, comprise a simple set of guidelines that we'll follow.

First, at the highest level, we want to separate the test code from the implementation code. That said, as we discussed in the tutorial on debugging, we can use the preprocessor to control what goes into our system build. Thus, we can selectively incorporate test code into the implementation suite to facilitate testing. We should make certain, however, that none of the test code is included in the release build.

Separation of Powers

When architecting your program, the main() routine should serve as a high-level container for the two main pieces of functionality: the test function and the implementation function. Let's examine the structure for testing a dynamic linked list type container.

```
#include files

global declarations

/*
 *  header describing program and tests
 */

int main(void)
{
    local declarations
    testerInvocation();

    return 0;
}
```

We begin at the top…

Then the implementation and test files implementation files first. For each file…that way →

```
# include system header files
# include linkedList class and function
definitions etc.

/*
 *  header describing program
 */
 member function implementations
```

…now the test suite… that way ↓

```
# include system header files
# include header containing function definitions etc

/*
 *  header describing program and tests
 */
void testerInvocation(void)
{
      Local declarations and definitions and initializations
      //    description of normal tests and expected behaviour

      1.    create initial instance of the linked list
            need to verify that each constructor functions properly

      2.    test normal behaviour
            insert into empty list
            insert at tail
            insert several links in middle

            insert at head
            insert at tail

            remove from head
            remove from tail
            remove from middle

            display list

            test any other member functions in public interface for expected behaviour

            delete populated list

      3.  test boundary behaviour
            //  description of boundary tests and expected behaviour

            create a working list

            insert before head
            insert after tail

            remove before head
            remove after tail
            remove from empty list

            display empty list
            delete empty list

            test any other member functions in public interface for boundary behaviour

      return
}
```

The following should be written to a test results file

1. Name of the test being run.

2. Identification of test vector(s) or data being used.
3. Expected results of the test.
4. Actual results of the test.

Note that "*test passed*" or "*test failed*" is not a presentation of the results of the test…neither is 12 or myName with no other descriptive annotation.

Understand that the test and results files and the items being tested will be different for each program being tested. The examples above are just that, simple representative examples. Your tests and test file must reflect the design and important issues in your program.