CPS122 Lecture: Quality Assurance Last revised March 22, 2023

Objectives:

1. To define key terms: "failure", "fault", and "error"
2. To introduce and distinguish between unit, integration, acceptance, and regression testing.
3. To introduce method preconditions, postconditions; class invariants.
4. To introduce and distinguish between black box and white (glass, clear) box testing.
5. To discuss the process of choosing test cases, including the notion of equivalence classes and path, edge, and node (statement) coverage.
6. To introduce and distinguish between top-down, bottom-up, and "sandwich" testing, and to introduce the notion of test "scaffolding".
7. To discuss testing, inspection, and formal verification as alternate and complementary means of attempting to ensure program correctness.
8. To introduce the process of reasoning from preconditions & invariant to postconditions and invariant.
9. To introduce the use of loop invariants and loop termination conditions

Materials

1. Projectables
2. Gries' coffee can problem - demo
3. Demo of old version of SimpleDate using a test driver main method
4. Handout for coffee can problem showing proof of `playOneRound`
5. Handout of proof of fast exponentiation

I. Introduction

A. In our study of the software lifecycle, we have seen that one of the major activities is quality assurance (or verification and validation).

1. While we are covering this as a distinct topic in the course, it is important to note that quality affects every aspect of the software lifecycle - it is not an add on at the end of the process. Most of what I

will say in this series of lectures pertains to quality assurance in general - regardless of where in the lifecycle it is actually done.

a) A key principle is that the earlier an error is discovered, the less expensive it is to fix - often by as much as an order of magnitude or more.

   For example, suppose the requirements for a piece of software contain an error. Fixing the defect in the requirements is clearly a lot easier than if the software, after being written to fulfill the erroneous requirements, now has to be changed to do what is should have done in the first place.

   (An example of this from a different realm - some time ago a contractor came out and put new gutters on my neighbors house. Alas - they had the wrong street address - she hadn't contracted for any work on her house by them! The result was that she got a free set of gutters!)

b) Nonetheless, it is common for extensive effort to be expended on verification and validation at the end of the software lifecycle, which is why we consider this topic here.

2. It is not uncommon for testing (and fixing the errors that testing uncovers!) to represent half or more of the time (and hence half or more of the cost) of initial development of a software product. Obviously, then, this aspect of program development is a prime target for savings.

   a) Unfortunately, in practice, savings are often achieved in this area by shortcuts that result in products being delivered that do not do what they are supposed to do, or produce "blue screens" or other unpleasant phenomena. Clearly, this is not an ethical or professional approach to developing software.

b) We can save testing and error correcting effort legitimately by approaching it in a correct way - which is the focus of this series of lectures.

B. Before going further, we need to be sure we are clear about several related terms that are often used interchangeably, but really have distinct meanings: “failure”, “fault”, and “error”.

1. Failure refers to an observable incorrect behavior of a system. A failure may be as minor as an appearance problem in a GUI, or it may be that the software causes they system to crash.

2. A fault is something that is wrong with the software. All failures are caused by faults, but not all faults manifest themselves as failures - at least during testing.

3. An error is the mistake the caused the fault. Faults exist because of errors - but it is possible for an error to not result in a fault. (E.g. most of us have made driving errors that have not resulted in collisions.)

   EXAMPLE: Consider the following java code, which is supposed to calculate the average of all the elements in an array of floats: x:

```
float sum = x[0];
for (int i = 0; i < x.length; i ++)
   sum += x[i];
System.out.println(“The average is “ + sum / x.length);
```

   PROJECT

   a) What’s wrong with this code segment?

      ASK

      For some reason, the programmer included x[0] in the sum twice. Either the initialization should have been sum = 0, or the for loop should have started at 1. If the programmer intended the latter, then the error was carelessness in typing the initial value for i in the loop.

b) In any case, the fault in this software is that it calculates the average incorrectly by counting x[0] twice.

c) The failure that would result from this fault is that an incorrect average is printed to System.out.

d) Would this code always result in a failure (an incorrect average being printed?)

ASK

No - not if x[0] happens to be zero! (or is so small, relative to the overall sum, that counting it twice doesn't produce a noticeably wrong result.)

Nonetheless, a fault is present, whether or not it results in a failure.

4. Note that we distinguish between errors that arise from how the software is used from faults in the software. Quality software anticipates things that can go wrong in "the outside world" and is prepared to cope with them.

a) Software must be prepared to deal with all sorts of external problems such as a communication link failure or another computer system being down or a file not existing on disk or a human error in typing input.

b) However, it is very bad practice to try to cover up exceptions that arise from faults in the software - e.g. code like the following might prevent a failure, but at the cost of hiding a fault:

```
try
{
    // Some code
}
catch(Exception e)  // Catches any exception
{ }                 // and simply ignores it
```

PROJECT

C. There are three broad categories of quality-assurance activities.

1. Execution-Based Testing

2. Inspections

3. Formal Proof of Correctness

D. All three need to take place at several points during and after the implementation process. **THEY SHOULD NOT BE LEFT TO THE END!** For starters, we will focus on execution-based testing.

1. One approach to testing (which I am not advocating) is called the "big bang approach". It goes like this: write all the code, and then when everything is written, test it

   a) This is a common method used by introductory CS students

   b) It usually leads to chaos when applied to large programs. When something goes wrong, where do you look?

   c) The other approaches are therefore much to be preferred!

2. Software of any size should be tested by using a combination of UNIT TESTING and INTEGRATION TESTING, followed eventually by ACCEPTANCE TESTING

   a) Unit testing means testing each piece of the software individually when it is coded to ensure that it performs its part

of the overall computation correctly.

Note that unit testing can (and should) take place as the software is being written. We have already looked at an approach to this using JUnit.

b) Integration testing means testing as the various units are put together into larger pieces or a complete program

(1)If every component has been successfully unit-tested, then any remaining errors should be interface errors - cases where a caller of a method made different assumptions about what the method does than the author of the method assumed.

EXAMPLE:

Suppose a certain method is required to compare two objects and return true or false based on their relative order in a sorted list. Assume this method is used by another method that actually sorts the objects.

Suppose the author of the method understands the expectation to be that the method returns true if the *first* object belongs *before the second* (i.e. they are in their correct relative order) but the author of a sorting method that uses it assumes that it returns true if the *second* object belongs *before the first* (i.e. their relative order needs to be switched.)

Both methods would test successful during unit testing (based on their author's understanding of the interface between them.) However, the error would show up in integration testing with the output being backwards!

(2)Integration testing is often a crisis time in software development - but need not be if the preconditions and postconditions of methods are spelled out carefully before coding begins. (As we discussed in connection with implementation.)

(3) In a large system, integration testing can occur at many levels, as larger and larger pieces of the final system are integrated with one another.

c) Acceptance testing is performed by the customer before "signing off" on the finished project.

Sometimes formal acceptance testing is preceded by "alpha testing" and "beta testing". (The cynic in me can wonder how much commercial software that one buys is really only at beta level - or below!)

3. Because fixing one problem with a piece of software often introduces another problem, there is also a need for REGRESSION TESTING. This involves repeating tests that have already succeeded after every change to be sure that the software still works correctly.

E. The author of a piece of software is not necessarily the best person to test it, for two reasons:

1. Psychology: Often people are reluctant to "break" their own code and hence may not test as thoroughly.

2. If the author of the software has misunderstood the requirements, she is unlikely to test for it. Having another person do the testing makes it more likely that a misunderstanding will be caught.

3. For this reason, many organizations have a separate "Software Quality Assurance Group" to do final testing. In this case

a) Software authors unit and integration test their software before releasing it to the QA group.

b) The QA group will do final testing based on the original specifications.

c) Doing things this way helps prevent the possibility that a software author didn't think about how the program would handle a certain piece of anomalous input, so his/her test data never tests it!

F. Quality assurance has, as its first goal, the prevention of errors. Given that this is not 100% possible, the second goal of quality assurance is to find and remove faults. Thus, quality assurance activities like inspection and testing are successful just when they find faults - not when they fail to find faults!

(Consider the process you go through of testing a program before you turn it in. The testing process is actually successful if the faults cause visible failures that you can fix. Having the software appear to work correctly - even though it contains faults - is not success at this point!)

II. Preconditions, Postconditions, and Invariants,

A. As a preliminary to planning a testing strategy, it is useful to think about the preconditions and postconditions for the various methods (and sometimes also for the system as a whole), and about invariants for the various classes (and sometimes for the system as a whole).

1. A precondition for a method is a statement of what must be true in order for the method to be validly called. A precondition for an overall system is what must be true of the input for the system to be used validly.

   EXAMPLE:

   As you may discover in lab, the `remove(int)` method of a List collection can be used to remove a specific element of a List. However, the method has a precondition that the specified element must exist - e.g. you can't remove the element at position 5 from a list that contains 3 elements, nor can you remove the element at position 0 (the first position) from an empty list.

   What happens if you fail to observe this precondition?

   ASK

2. A postcondition for a method or overall system is a statement of what the method/system will guarantee to be true - provided it is called/used with its precondition satisfied.

   EXAMPLE: The postcondition for the `remove(int)` method of a `List` collection is that the specified element is removed and all higher numbered elements (if any) are shifted down - e.g. if you remove element 2 from a `List`, then element 3 (if there is one) becomes the new element 2, element 4 (if there is one) becomes new element 3, etc.

   Note that a method is <u>not</u> required to guarantee its postcondition if it is called with its precondition not satisfied. (In fact, it's not required to guarantee anything!)

3. An invariant is a statement that is kept true of something. As an example, consider the following problem (a classic example first suggested by David Gries in 1981):

   "A coffee can contains some number of black beans and some number of white beans. The following process is repeated as long as possible: Randomly select two beans from the can.
   - If they have the same color, throw them out, but put another black bean in. (Enough extra black beans are available to do this.).
   - If they are of different colors, place the white bean back in the can and place the white bean back in the can and throw the black one away.

   Execution of this process always reduces the total number of beans in the can by one, so repetition of this process staring with a non-empty can must terminate with exactly one bean in the can, for then two beans cannot be selected.

   Question: by looking at the number of beans in the can initially, can you tell what the color of the final bean will be?

a) DEMO once.

b) It turns out that this question can be answered by discovering an invariant property of the can.

   DEMO again with prediction

   ASK Class to come up with ideas as to what the invariant is.

c) It turns out that the invariant is the "oddness" of the number of white beans in the can.

   - If we pick two beans of the same color and throw them away, then we either reduce the number of white beans by 2 or not at all - so the oddness of the number of white beans doesn't change.

   - If we pick two beans of different colors, putting the white one back in the can doesn't not change the number of white beans.

   - Thus, if the number of beans in the can to start with is odd, the final bean will be white; if it is even, the final bean will be black.

4. We can use invariants in several places:

   a) Sometimes, a system as a whole will have an invariant property.

      Example: If a person has two bank accounts and transfers money between them, the total amount of money the person has in the bank is invariant.

   b) A class invariant is a something that is held true for an object of the class regardless of what operations are performed on it.

Example: The coffee can program we just demonstrated uses a class called CoffeeCan that has an invariant that the "oddness" of the number of white beans stays unchanged.

c) A loop invariant is something that is held true for each time through the loop. We'll look at an example shortly.

5. An invariant has two essential properties: establishment and preservation.

a) Establishment: It is satisfied initially.

(1) In the case of a class invariant, a primary responsibility of each constructor in a class is to make sure that any newly-constructed object satisfies the class invariant.

(2) In the case of a loop invariant, it is the responsibility of the code just before the loop is entered to make sure the invariant is true to begin with.

b) Preservation: The invariant remains true after each operation.

(1) In the case of a class invariant, calling a public method of the class with the invariant true and the preconditions of the method satisfied results in the invariant remaining true (though it may temporarily become false during the execution of the method)

Therefore, a primary responsibility of any <u>public</u> method is to preserve the invariant.

Technically, <u>private</u> methods are not required to preserve the invariant - so long as public methods call them in such a way as to restore the invariant before the public method completes.

That is, the class invariant must be satisfied only in states which are visible to other classes. It may temporarily become false while a public method is being executed.

(2)In the case of a loop invariant, if the loop body is started with the invariant true, then the invariant remains true when the loop body completes.

Note that the loop invariant may become temporarily not true in the middle of the loop, as long as it becomes true again before the body completes.

6. Invariants can be used to show the correctness of code - especially code that is somewhat "tricky"

Example: consider the following code, which is an efficient way to raise a number to a power. (It takes time O(log exponent) rather than O(exponent))

```
/** Raise a number to a power
 *  @param base the number to raise to a power
 *  @param exponent the power to raise it to
 *  @return base raised ** exponent
 */
public static double power(double base, int exponent)
{
   double v = 1.0, b = base;
   int e = exponent;
   while (e > 0)
   {
        if (e % 2 == 0)
        {
             b *= b;
             e /= 2;
        }
        else
        {
             v *= b;
             b *= b;
```

```
            e /= 2;
        }|
    }
    return v;
}
```

PROJECT

To show the correctness of the code, we observe that the while loop has the invariant v * b ** e = base ** exponent.

a) Established initially since b is initially base and e is initially exponent

b) Preserved each time through the loop

(1) If e is even, b bis squared and e is divided by 2, which preserves b ** e and hence the invariant.

(2) If e was odd to begin with, we do the same thing, but also since e is an integer we need to multiply v by b to compensate for the fact that the division is not exact.

III. **Execution-Based Testing**

A. In this section, we will focus on what is sometimes called "execution-based testing" - i.e. running the software (or a portion of it) under controlled conditions and identifying failures. Execution-based testing, of course, can only take place during or after the implementation portion of the software lifecycle (though there has been some interest in the notion of "executable specifications" that would allow this kind of testing earlier in the process).

B. All too often, testing is approached as follows: we write code that we think is correct, and then we test it with data for which the correct results are known. When discrepancies are found, we locate their source and repeat the testing procedure until all tests produce correct results.

1. But does this process really mean that we have succeeded in producing correct software? - NO - it simply means we have succeeded in producing software that correctly handles all of the cases we have anticipated.

   a) Note that many serious errors have arisen precisely because the developers of a system did not anticipate some possible input when specifying and designing the system. It is not surprising that, in such cases, no test data showed the fault.

      EXAMPLES: Y2K; rising moon in NORAD's first missile warning system

   b) Of course, no system can totally prevent such situations from arising. The fact that they have occurred should keep us from overconfidence in software that "passed all the tests".

2. Does this process give us confidence in the correctness of our software? - NO - because each time we find an error we begin to wonder what others might be lurking out there undetected.

   a) Harlan Mills put it this way:

      "There is no foolproof way ever to know that you have found the last error in a program. So the best way to acquire confidence that a program has no errors is never to find the first one, no matter how much it is tested and used." (In 'How to Write Correct Programs and Know it').

   b) One textbook author spoke of what he called "the law of conservation of bugs":

      "The number of bugs remaining in a large system is proportional to the number of bugs already fixed. In other words, a fault-ridden system will always tend to remain a fault-ridden system. This is because a poorly-designed system will not only have more faults to start with, but will also be harder to fix correctly and

hence will suffer more strongly from the ripple effect." (The phenomenon that fixing one fault often introduces another.)

3. When testing is done by the same person who wrote the software, a third problem also comes into play: a psychological one. Since we really hope our software is correct (and we don't want to have to rework it again), it's easy for us to not test it as rigorously as we should.

4. As we noted earlier, waiting until the code is all written before testing it means that correcting some kinds of errors will be very costly.

   a) If the software specifications were wrong, and this isn't discovered until it is tested, then the whole piece of software may have to be rewritten virtually from scratch!

   b) Thus, effort is conserved if we try to ensure correctness at every step along the way, rather than waiting to test the finished product.

C. Choosing good test data is critical - but is often the hardest task of all.

1. Ideally, we would like to test our software EXHAUSTIVELY - i.e. by testing every possible input. But if we could do that, we probably wouldn't need to bother writing the program!

   EXAMPLES:

   If a program accepts a single integer as input, there are 4 billion possible cases that might need to be considered!

   Even if the values it accepts are limited, they can interact - e.g. a program that accepts 5 integers in the range 1..10 actually has 10^5 or 100,000 possible cases

2. We are therefore forced to use only a small subset of the possible inputs for testing. How we choose this subset depends in part on whether we are doing black box or white (glass, clear) box testing.

a) White box testing (also known as glass or clear box testing) is done by choosing cases based on the code, with an attempt to achieve some sort of "coverage" of all portions of the code.

b) Black box testing is done without regard to any knowledge of the internal structure of the software. Test cases are chosen only on the basis of the specifications.

c) Commonly, testing done by the program's authors during development uses a clear box approach, while final acceptance testing (often done by a separate QA group) is done using a black box approach.

3. In either case, we can use preconditions, postconditions, and invariants to guide our testing.

D. Choosing test cases for white/clear/glass box testing

1. There are several different levels of coverage we can attempt to achieve

a) Statement coverage - we ensure that every statement is tested by at least one test case. (The text author calls this “node coverage”.)

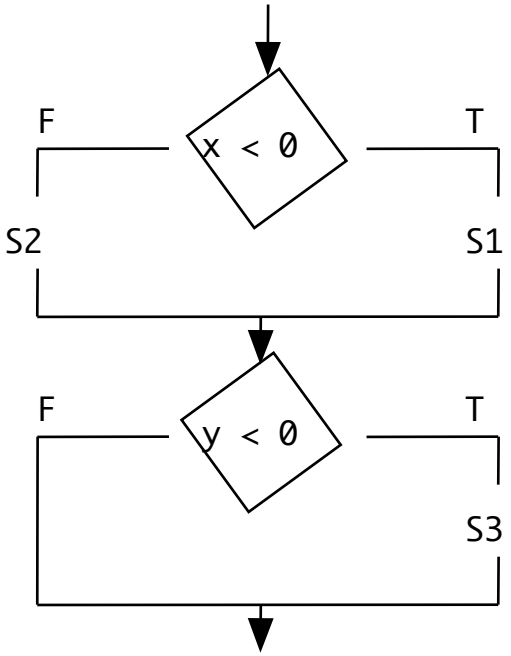
b) Path coverage - we ensure that every possible path through the program is taken. (This necessarily implies statement coverage) Of course, for programs containing loops, this is impossible, since there are infinitely many possible cases.

c) Edge coverage - we ensure that every possible edge in the program graph is taken at least once. (A practical alternative to path coverage.)

EXAMPLE: The following is a highly-simplified example to illustrate the difference. Consider the following block of code, where the “S”’s are arbitrary statements that don’t affect the value of x or y:

```
if (x < 0)
     S1;
else
     S2;
 if (y < 0)
     S3;
```

This corresponds to the following flow graph



PROJECT CODE AND F LOWCHART

(1)To achieve statement coverage, we need to ensure that each statement (S1, S2, S3 and the if tests) is executed at least once. This can be done with just two sets of test data. Can anyone see a way to do this? (Actually there are lots of possibilities)

ASK

(a)x = -1, y = -1 (tests S1, S3 and ifs)
(b)x = 1, y = -1 (adds S2)

PROJECT

(2)To achieve edge coverage, we need to ensure that each of the edges in the graph is traversed at least once. The above data set misses one edge. Which one?

ASK

The "false" edge out of the second test (s < 0).

We could achieve edge coverage by adding a data set like x = 1, y = 1. (For that matter, in this case we could change the second original data set to this as well)

(3)To achieve path coverage, we need to ensure that each possible path through the program is executed. How many possible paths are there?

ASK

Four (two possibilities for x) * (two possibilities for y)

For example, the following data would test all paths:

(a)x = -1, y = -1 (right, right)
(b)x = -1, y = 1 (right, left)
(c)x = 1, y = -1 (left, right)
(d)x = 1, y = 1 (left, left)

PROJECT

2. In practice, statement coverage and edge coverage are both possible - and edge coverage is preferable since it implies statement coverage. Path coverage, however, is often not practical, due to the large volume of cases that might be needed.

   Example: a program has 10 inputs, each of which selects one of two paths at a branch in the program. There are potentially 2^{10} (=1024) paths. With 20 variables, this jumps to over 1 million.

   A single loop can generate a near-infinite number of possibilities.

   Example:

```
int x;
while (x > 0) {
   if (x == 3)
        S1;
   else
        S2;
   x = x / 2;
}
```

Possible paths include the following, all of which differ:

x = 0 (nothing)
x = 1: S2
x = 2: S2, S2
x = 3: S1, S2
x = 4: S2, S2, S2
x = 6: S2, S1, S2
...

PROJECT

And so on for infinitely many possibilities (or at least as many as there are possible values of x represented by an int on the underlying machine)

E. Choosing test cases for black box testing.

1. In identifying test cases for black box testing, it is helpful to realize that the set of all possible data the program can handle (which is often large if not infinite) can be partitioned into equivalence classes.

   EXAMPLE:

   Suppose one input to a program must be an integer in the range 0..100. Suppose the program is supposed to do some computation with the valid integers, and to print an error message for an invalid integer.

The equivalence classes are: negative integers, integers between 0 and 100 (inclusive) and integers greater than 100.

2. For each equivalence class, we want to be sure that our test data includes

   a) A typical input from the class (e.g. 20 or 50 or 98 would be suitable for the class of integers in the range 0..100)

   b) If the class represents valid input, boundary inputs - values that lie right on the edges of the class - e.g. for integers in the range 0..100, the boundary values would be 0 and 100.

   c) If the class represents invalid input, values that are barely illegal (just on the wrong side of the boundary) - e.g. -1 for the class of negative integers and 101 for the class of integers > 100.

3. A common error I have noticed in student test plans is to expend a lot of effort on all sorts of illegal input (including really bizarre possibilities) while not testing correct input at all, or in only a very limited way! The result is that we have confidence that the program can handle the bizarre nicely, but no confidence that it will do the right thing in the normal case!)

F. One question that arises in this context is "how can I do unit testing or even integration testing of software that depends on other code that hasn't yet been written?"

1. We have to develop classes in some order, and therefore must be able to test one class while other classes have not yet been written. This complicates testing because we get into a "chicken and egg" problem.

   EXAMPLE:

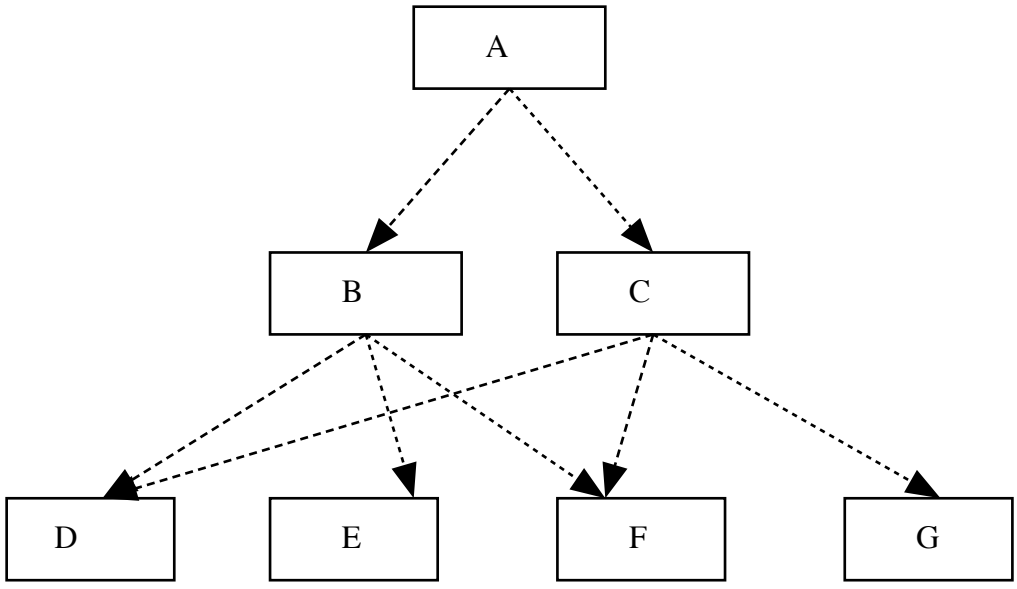
   Suppose a method mA of class A uses a method mB of class B. Suppose further - as will often be the case - that mA produces some output that is visible to the user (and thus can be examined during

testing), but mB produces no directly visible output. Which class should we implement and unit test first?

a) If we try to implement class A first, then when we try to test mA we run into a problem because it relies on mB, which has not yet been written.

b) If we try to implement class B first, then when we try to test mB we run into a problem because it produces no visible results - the results of mB are only visible when it is called by mA, which has not yet been written.

2. In general, there are two broad approaches to the question of "in what order do we implement the classes", called TOP-DOWN and BOTTOM-UP. We can see where these names come from if we arrange the classes hierarchically, so that each class only depends on (makes use of the methods of) classes below it in the hierarchy - so we get something like this:



PROJECT

(actually, this kind of strict hierarchical structure is more common in non-OO systems than in OO systems, which tend to have more

cooperating peers. Even so, though, it is often good practice to arrange the objects in an OO system in layers).

a) In a top down approach, we might implement and unit test modules in the order A, B, D, E, C, F, G.

b) In a bottom up approach , the order might be D, E, B, F, G, C, A.

c) Also possible (and useful in many cases) is a "sandwich" approach in which we first do the "bread" (A, D, E, F, G) and then the "filling" (B and C). (This approach is common if the highest layers are the GUI and the lowest layers provide utility services like access to a database, with the middle layers being the business logic.)

3. Whichever order of implementation and testing we choose, we are forced to create some extra code just to allow us to perform testing.

   This code is often called scaffolding - by analogy to building construction,, where temporary structures (scaffolds) are built to allow work on the target structure.

   a) Consider the matter of unit testing a method that relies on another method not yet written - as would happen in the above if we chose to develop A first (or for that matter to develop B or C before developing D, E, F, and G.) In this case, one must write a STUB - a piece of code that stands in for the as-yet unwritten method by doing something reasonable, like simply printing out the fact that it was called or returning a known "dummy" value.

      Note: In the skeleton code I gave you for the first registration system lab, all of the method prototypes were present, mostly with empty bodies. In the case of a method that returns a value to its caller, a dummy return statement (labeled STUB) was inserted to return an acceptable value (typically null) so the code would compile. These were not full-blown stubs, since a stub used for top-down testing would have to be able to produce a full range of possible results (which may mean interactively asking the user to

specify a particular result). E.g. in the registration system lab, a complete stub for the getCourse(String id) method of class Database would have to ask the user whether it should return null (as it now does) or an arbitrary valid course.

b) Now consider the matter of unit testing a method before its caller has been written - as would happen in the above example if we chose to develop D, E, F, or G first (or to develop B or C before developing A). To do this, it is necessary to write a piece of "throwaway" code called a TEST DRIVER to call the method under test with various inputs and report its results.

(1)One approach is to create a separate program that performs the testing by requesting input from the user (or providing some constant input), calls the method under test, and then either reports the result or checks the result against a known good value. In Java, this can often be incorporated into a class as a main method that serves to exercise that class. (A Java program can have main methods in any number of classes - one is designated as the main method for the program when it is run and the rest are ignored.). Thus, a Java class that is not a main class could still have a main method which serves as its test driver.

EXAMPLE: An earlier version of the SimpleDate.java class I gave you for the project incorporated the following test driver

(PROJECT)

```
public static void main(String [] args)
{
   SimpleDate today = SimpleDate.getToday();
   SimpleDate tomorrow = today.daysLater(1);
   SimpleDate aWeekFromNow = today.daysLater(7);

   System.out.println("Today is " + today);
   System.out.println("Tomorrow is " + tomorrow);
   System.out.println("A week from now is " +
        aWeekFromNow);

   System.out.println("A week from now is " +
        aWeekFromNow.daysAfter(tomorrow) + " days
```

```
        after tomorrow");

    System.out.println("Today equals tomorrow: " +
        today.equals(tomorrow));
    System.out.println("Today is after tomorrow:  " +
        today.isAfter(tomorrow));
    System.out.println("Tomorrow is after today:  " +
        tomorrow.isAfter(today));

    // Unless the program is started a few seconds
    // before midnight, the newly constructed object
    // should test the same as today

    try { Thread.sleep(10000); }
    catch(InterruptedException e)
    { }

    SimpleDate tenSecondsLater = new SimpleDate();
    System.out.println("Ten seconds later equals " +
        "today "+tenSecondsLater.equals(today));
    System.out.println("Ten seconds later is after "+
        "today  " + tenSecondsLater.isAfter(today));

    // Now test the code for diddling with today

    SimpleDate.changeTodayBy(5);
    System.out.println("After diddling with the " +
        "date, today is " + SimpleDate.getToday());
}
```

With this code in place, it was possible to run the class as if it were a standalone program, and then manually examine the output to see if all the methods were doing the right thing.

(DEMO)

(2)We have used an approach to unit testing called test-first development, and have made use of a facility available in the Java world called JUnit. (Similar facilities are available for other languages - e.g. pyunit we used in CPS121. In effect, what these tools do is facilitate the process of writing test drivers.

(3) Note that it is usually only necessary to develop one or the other type of scaffolding. Once we have tested a method that relies on stubs for as-yet unwritten methods it calls, we can use it to help us test those methods when they are written. Conversely, if we have tested a method using a test driver, we can then use it to help us test the methods that call it.

c) Object-oriented testing is a still a significant research area. A key question that is not yet fully answered is how best to test a class in isolation (as a unit) when classes are designed to work together.

G. When errors are uncovered during testing, we must, of course, locate and correct them - a task that is often easier said than done!

1. This task is made easier if we are disciplined about unit testing and integration testing, since the amount of code we must consider when looking for the error is kept down.

   a) During unit testing, we must of course only consider the unit being tested (or perhaps the test driver!)

   b) During integration testing, we can focus our attention on the INTERFACES between the previously tested units.

2. One tool that can often be used to help us here - is one that is commonly (but perhaps erroneously) called a SYMBOLIC DEBUGGER.

   (We won't be able to use this in CPS122, but you will encounter one in a later course if you continue in CS.)

H. Finally, it should be noted that effective testing calls for a TEST PLAN. A TEST PLAN must address issues like:

1. Scaffolding that needs to be built
2. Testing procedures - including the order in which various tests are done.

3. The test suite - the actual data to be used. Wherever possible, it is good to put the test suite into some form that it can easily be run again during integration testing - preferable an electronic form where applying the test suite can be automated; otherwise paper form.

IV. **Inspection**

A. While testing will always be one part of the process of attempting to ensure program correctness, best results are achieved when testing is used in conjunction with other methods. We will look at a number of methods that are based on disciplined READING of the code and REASONING about it:

B. Desk-checking: careful reading of our work ourselves, to try to uncover places where it is not correct - and (ultimately) to convince ourselves that it is correct once faults have been removed.

C. Inspections/reviews: having others read our work to help locate faults.

1. Because we know what we meant to do, we will sometimes fail to see places where we didn't actually SAY what we MEANT. Someone else may catch this for us.

2. Inspection and review can be done in many ways.

   a) Informally - by asking a colleague to review our work. (This approach can be used very productively when doing a project with a partner.)

   b) Various forms of Structured Walkthroughs are often used in industry. A small group of an author's peers will listen as he/she walks through the logic of his/her work with them, pointing out things they see that need correction or improvement.

   c) Formal reviews by management.

3. An interesting fusion of the these two approaches (desk-checking and reviews) that has arisen recently is the idea of PAIR PROGRAMMING, which is the idea behind the way we structure

labs in the course, and behind assigning team projects. However, having two people working independently on the same project is not pair programming; pair programming means two people ALWAYS WORKING ON EVERYTHING TOGETHER. (One types; the other thinks and interacts.)

V. **Basic Concepts of Formal Verification**

A. Formal verification means applying the methods of mathematical proof to programs in order to PROVE a program correct. Note that while testing only gives RELATIVE confidence in a program's correctness (unless of course you can test every possible input), a mathematical proof can give ABSOLUTE confidence - provided its logic is correct.

1. The method of program proof is not a total substitute for testing - since there can still be flaws in our proof logic. (There is a famous example in the software engineering literature concerning a short program that was "proved" correct several times - but each time it contained a fault which was not caught due to an error in the logic of the proof!)

2. But when formal verification is coupled with testing and inspections it can give a very high degree of confidence in program reliability.

3. Using the methods of mathematical proof while one is developing a program can help to prevent errors in the first place by helping us think clearly about what we are doing.

4. In practice, there are serious limitations to the usefulness of this approach due to the time and effort involved. However, in critical, tricky portions of a program - or systems where human lives are at stake (e.g. airplane flight control software) it can be a powerful aid.

B. By formal verification, we mean using the methods of mathematical proof to demonstrate that code MUST be correct.

1. The starting place of formal verification is the preconditions, postconditions, and class invariants we identified during detailed design and implementation.

2. Recall that:

   a) A precondition of a method is an assertion that describes conditions that the caller of a method must guarantee will hold when the method is called.

   b) A postcondition of a method is an assertion that describes conditions that a method guarantees will hold when it terminates, provided it was called with its preconditions satisfied. (No guarantees if the preconditions don't hold!)

   c) A class invariant is an assertion that holds:

      (1)When an instance of the class is first constructed.

      (2)After the COMPLETION of every public method (that is called legally - i.e. with its preconditions satisfied)

      (3)Note that the invariant can become false during the execution of a public method, so long as the method guarantees to make it true again before it completes.

      (4)The invariant need not hold for private methods, since these can only be called by public methods, which are allowed to make the invariant momentarily false as long as they finish with it true.

3. As a consequence, every public method of the class can assume that the class invariant holds when it is called, along with the preconditions.

C. In formal verification, we prove the correctness of each public method by showing all of the following:

1. If the method is called with its preconditions satisfied, and with the class invariant holding, then it will guarantee upon completion that its postconditions are satisfied.

   Symbolically::

   { pre & inv } M { post }

   PROJECT

2. If the method is called with its preconditions satisfied, and with the class invariant holding, then it will guarantee upon completion that the class invariant remains satisfied.

   Symbolically:

   { pre & inv } M { inv }

   PROJECT

3. If the method is called with its preconditions satisfied, and with the class invariant holding, then we must also ensure that every method it calls has its preconditions satisfied as well (and the class invariant as well if it is a public method)

D. Note that, in order to prove the correctness of a public method, we may also need to prove the correctness of any private methods it calls. A private method is proved true by showing that, if it is called with its preconditions satisfied, it guarantees the truth of its postconditions - with no assumptions about the class invariant unless explicitly stated in pre/post conditions.

E. In developing a proof of a method, we are allowed to rely on:

1. General properties of the various statements comprising its body.

2. The correctness of methods it calls - i.e. if our method can guarantee to satisfy the preconditions of any method it calls, then it can rely upon the postconditions of that method as part of establishing its own correctness.

F. Example: Coffee Can problem (discussed earlier).

1. DEMO again

2. Recall that we saw that they key to understanding the behavior of the program is to realize that the even/oddness of the number of white beans is invariant. Hence, if the can initially contains an odd number of white beans, the final bean will be white; it it initially contains an even number, the final bean will be black.

3. HANDOUT, discuss proof of `playOneRound()` and `chooseBean()`

   a) In the case of `playOneRound()`, we have

   { Pre & Inv } M { Post & Inv }

   b) Note, though, that the class invariant: w mod 2 == w_0 mod 2 is <u>not</u> satisfied when `chooseBean()` is called the second time in `playOneRound()` because the code just before may decrement w. This is OK, because `chooseBean()` is a private method and does not require this in its preconditions. The class invariant is restored before `playOneRound()` terminates.

G. Two more concepts we need to consider are a LOOP INVARIANT and LOOP TERMINATION.

1. A loop invariant is an assertion that is known to be true the first time the loop body is entered, and whose truth is not altered by a complete execution of the loop body.

   EXAMPLE: Fast exponentiation we did earlier

HANDOUT: correctness proof - review how part 1 formalizes what we discussed earlier

2. Termination of Loops

   a) When proving the correctness of a loop, there is one other item we must consider: TERMINATION. We need to show that the loop will eventually terminate.

   b) EXAMPLE:

      Discuss termination proof (part 2 of proof)

3. Together, the loop invariant and the proof that the loop terminates establish the postcondition of the method (part 3 of proof)