Quality Assurance in Construction

As discussed earlier, verification and validation occur throughout the software engineering life-cycle and developers can't (and shouldn't) try to isolate these kinds of activities. Nonetheless, from a conceptual standpoint, there are verification and validation activities prominent during the construction phase. We discuss these activities in terms of the role of programmers.

Programmers must accomplish many things while they are programming. First, they must write software that is syntactically correct. That is, they must write software that obeys the grammatical rules of the language they are using. Second, they must write software that is consistent with the standards sanctioned by the project and organization. Third, they must write software that implements algorithms and data structures in accord with design specifications. The verification and validation activities that are conceptually part of the construction phase are concerned with these three objectives.

Some verification and validation activities that occur during construction are automated (and referred to as *code analysis*), and some are not (and referred to as *code review*). Some analysis techniques involve programs while they are running (and are referred to as *dynamic analysis*), while others examine software when it is not running (and are referred to as *static analysis*).

Code Reviews

Code reviews are a specific example of the general category of reviews discussed in the chapter on software quality assurance. As with other kinds of reviews, they differ in their formality. Desk checking software differs from desk checking other work products in that the author of the code executes it by hand when desk checking, while with other work products, the author merely reads the work product carefully. During construction, sub-programs should be desk-checked in isolation. In other words, there is no reason to (and it is very difficult to) desk check a large software system. Code walkthroughs also need not wait for the entire program or system to be completed, but usually involve fairly substantial sub-programs or libraries.

Software work product inspections were originally done for code, and they have been well developed and studied. Code inspections are are regarded as **formal reviews** because each participant has a well-defined role, the activity follows a well-defined process, and the activity is guided by a checklist. We discussed the formal inspection process in some detail in the software quality assurance chapter; here we simply add a few details particular to code inspections to that discussion.

Formal code review preparation and the review meetings themselves are quite taxing. If they are done too quickly, they are usually not done well. Consequently, preparation rates should not exceed about 200 lines of code per hour, and team inspection rates should not exceed about 150 lines of code per hour.

The code inspection checklist should include specific defects that inspectors should look for during preparation. The following are some examples from a Java code inspection checklist.

- All variables and constants are named in accord with naming conventions.
- There no variables or attributes with confusingly similar names.
- Every variable and attribute is correctly typed.
- Every method returns the correct value at every return point.
- All methods and attributes have appropriate access modifiers (private, protected, public).

- No nested if statements be converted into a switch statement.
- All exceptions handled appropriately.

Ideally, the inspector makes a pass through the work product looking for a handful of defects, then another pass looking for another handful, and so forth; looking for all defects at once in a single pass is usually not effective.

Static Analysis

Static analysis involves the automated verification of software before it is executed. Because static analysis techniques do not involve execution, they can be performed on sub-programs.

Syntax Checking

The simplest kind of static analysis is syntax checking. A syntax check ensures that the software obeys the grammatical rules of the language it is written in. Today, syntax checking is most frequently performed by a language-sensitive editor.[1] That is, the editor (i.e., the software product that the author uses to create and edit code) knows the syntax of the language being used and, while the author types, checks to make sure that everything is syntactically correct. The author is informed of syntactic mistakes using some kind of symbology (either color-coding, icons, or underlining). The process is quite similar to the spell checking and grammar checking that is performed by many word processors. However, many editors also can suggest syntactically correct completions of partially entered statements or automatically complete statements.

One important thing to be aware of is that syntax checkers (including compilers) do not always correctly identify the root cause of the problem. For example, consider the following fragment in C/Java/JavaScript.

```
if (hour <= 0) || (hour > 12))
{
   invalid = true;
}
else if (hour === 12)
{
  hour = 0;
}
```

A syntax checker might suggest that the `||` operator in the first line was misplaced, might suggest that there was an extra `)` in the first line, or might suggest that the `else` was misplaced. In fact, the problem is a missing `(` before `hour` in the first line. As another example, consider the following fragment in HTML.

```
<p.>
I never think of the future. It comes soon enough.
</p>
```

A syntax checker might suggest that there is a missing `</p.>`, or might suggest that there is an unexpected `</p>`. In fact, the first line should be `<p>`.

1 In the early days of programming, syntax checking was typically performed by the compiler or interpreter. In the days of punch cards and batch processing, this meant that the author might have to wait hours to learn of a syntax error.

Style Checking

Most projects require programmers to follow a language-specific style guide. Style guides include standards related to typography (such as the use of uppercase and lowercase letters), naming (such as descriptive names for variables), formatting, including indentation and other white space issues (for example, the size of tabs, the location of curly brackets), control structure usage (for example, always having a default case in switch statements), module sizes (for example, the maximum lines in a method or class), language features usage (for example, always declaring variables in languages that do not require this), and so forth.

One of the first style checkers was a tool called *lint*. It was developed at Bell Laboratories in the 1970s to find the "undesirable fiber and fluff" in C programs. (Many current style checkers include "lint" in their name out of respect.) For example, lint checks to see if any variables are declared and not used. While this is not a problem either syntactically or logically, it is bad style.

Idiom and Usage Checking

Though lint (for C) is often described as a style checker, it is actually much more than that. For example, it identifies statements that are unreachable, logical expressions that have constant values, and calls to functions that return values in some places and not in others. Hence, lint is also an idiom and usage checker.

Usage checkers usually look for three different things: suspicious or error-prone constructs (e.g., uninitialized variables, the use of the division operator), non-portable constructs (e.g., constructs that might give rise to range problems), and memory allocation inconsistencies. Idiom checkers, on the other hand, are much more varied.

Idiom and usage checkers tend to be language specific. For example, in C an idiom and usage checker might flag functions that are passed different numbers of arguments or arguments of different types, whereas in Java the compiler would detect this. As another example, in JavaScript, an idiom and usage checker might flag arrays that are constructed using `new Array()` rather than `[]` (for example, `grades = new Array()` rather than `grades = []`, which is purely an idiomatic matter peculiar to JavaScript).

There are exceptions, however. For example, many languages have both for and while loops, and an idiom and usage checker might flag determinate while loops or indeterminate for loops. As another example, some languages do not have block scope, and an idiom and usage checker might flag variables that are declared in a block (that, between curly brackets) as misleading.

Formal Methods

The term **formal methods** refers to the class of static analysis tools that rely on mathematical models. They can be categorized as follows.

Model checking is the process of automatically determining if a program or sub-program satisfies certain requirements. The inputs to a model checker are the program or sub-program and a formal specification of requirements. Most often, requirements are specified using logical expressions.

Data flow analysis is the process of enumerating the set of possible values calculated at various points in a program or sub-program using ideas from graph theory.

Symbolic evaluation is the process of automatically tracing the execution of a program or sub-program using symbolic values rather than numeric values. It is used to identify the values that will cause different statements to be executed.

Despite the rigor of these methods, they are not perfect. It has been proven that no automated process can always determine if an arbitrary program will execute correctly.[2] In addition, these tools tend to be very difficult to use, so they are used only in specialized circumstances.

Unit Testing

As discussed in the chapter on quality assurance, **unit testing** is the testing of individual units or sub-programs in isolation. An individual unit test case consist of one value for each of the sub-program arguments and one corresponding expected value for each of the outputs. The goal of unit testing is to identify all of the faults in a sub-program. An incomplete test suite will give rise to a **false negative** (the conclusion that there are no faults). An incorrect expected value (say, as a result of a mistake made when calculating the value by hand) in a test will give rise to a **false positive** (the conclusion that there is a fault when there isn't).[3]

Developing Tests

There are two broad strategies for testing that differ in what is presumed to be known about the component, sub-system, or product being tested. In **black box** testing the processing details of the component, system, or product are presumed to be unknown. That is, the person developing the tests only knows what can go into the tested unit and what should come out. In **clear box** testing (sometimes also called **white box** or **open box** testing), the person developing the tests knows exactly how the component, sub-system, or product is built and how it should operate. Though it might seem counterintuitive, a combination of the two is generally required for effective testing.

Clear box testing tends to focus on the **coverage** of the tests, which can be loosely defined as the proportion of the tested unit that is exercised. However, it is important to understand that coverage can be measured in a variety of different ways, and many of the measures used are based on a graph-theoretic representation of the component, sub-system, or product. Most commonly, the graph theoretic representation is the *control flow graph (CFG),* but it is also possible to use activity diagrams for this purpose.

In a control-flow graph, an **action node** represents a piece of code with one entry point and one exit point. For example, an assignment statement can be represented by an action. A **decision node**, on the other hand, is used to represent the start of a piece of code with one entry point and multiple exit points. For example, the conditional expression in an if statement is represented by a decision node. Nodes are connected by arrows representing possible flows of execution through the nodes.

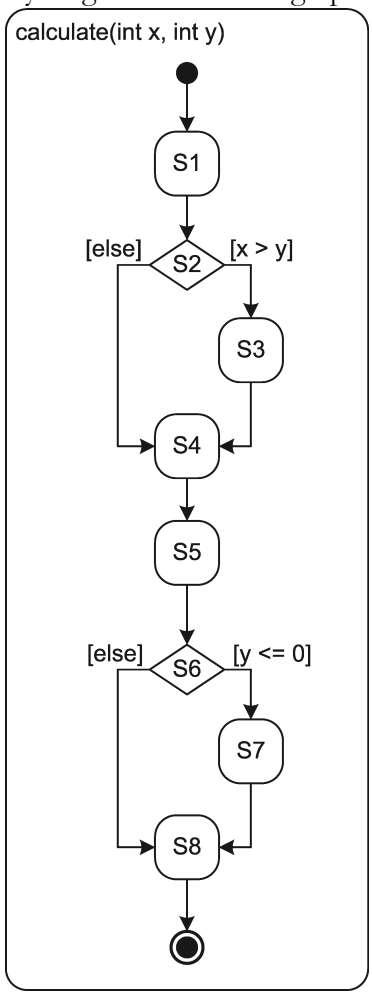
The easiest way to understand the difference is with the following example (in which the statement numbers are included as comments at the end of each executable line). For this example, test case input consists of a values for the `calculate()` function parameters `x` and `y`, which we write as the ordered pair (x, y). (Since we are only concerned now with code coverage, we will not worry about test case outputs.)

2 This result is related to the halting problem for Turing machine.

3 The terms false negative and false positive can be confusing but are used in the way they are in medical testing where a positive result indicates the condition is present (often a bad thing from the standpoint of the patient) and a negative result indicates the condition is not present (often a good thing from the standpoint of the patient). The terms type I error and type II error from statistical hypothesis testing are also sometimes used in software testing, bet their interpretation depends on the statement of the hypothesis (e.g., there are no faults).

```
int calculate(int x, int y)
{
   int        a, b;

   a = 1;                    // S1

   if (x > y)                // S2
   {
      a = 2;                 // S3
   }

   x++;                      // S4
   b = y * a;                // S5

   if (y <= 0)               // S6
   {
     b++;                    // S7
   }

   return b;                 // S8
}
```

This example can be illustrated ina flow-graph, which shows all alternative flows of control through a piece of code. The following activity diagram is the flow graph for the code above.



We can use this flow-graph to calculate various kinds of coverage metrics.

Statement coverage is the percentage of statements exercised when a set of test cases is executed. To measure statement coverage, one identifies the nodes in the activity diagram that have and have not been executed by a set of test cases. For example, consider the test case with input (1, 2). Tracing execution using either the code or the activity diagram, it is easy to see that the statements S1, S2, S4, S5, S6, and S8 are executed while S3 and S7 are not. Hence, the statement coverage for this test is 75% (six statements were covered out of eight total statements).

Branch coverage is the percentage of branch directions taken when a set of test cases is executed. In branch coverage, one identifies the outbound edges from decision nodes that have and have not been traversed. For example, again consider the test case with input (1, 2). Since 1 is not greater than 2, the [else] edge out of S2 will be traversed; since 2 is not less than or equal to 0, the [else] edge out of S6 will be traversed. Hence, the branch coverage for this test is 50% (two outbound edges from decision nodes out of four total).

Path coverage is the percentage of all execution paths taken when a set of test cases executes. In path coverage, one identifies the paths that have and have not been traversed from the initial node to the final node. In this example, there are four possible paths from S1 to S8. The first is S1, S2, S4, S5, S6, S8; the second is S1, S2, S3, S4, S5, S6, S8; the third is S1, S2, S4, S5, S6, S7, S8; and the fourth is S1, S2, S3, S4, S5, S6, S7, S8.[4] Since the test case input (1, 2) causes traversal of only one path, it has a path coverage of 25% (one path out of four).

Of course, one test is never enough, and one must consider the coverage of entire test suites (sets of test cases). Letting $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ denote the set of test case inputs in a test suite (where *n* denotes the number of tests in the suite, x_i denotes the first parameter of `calculate()` and y_i the second, consider coverage for the following three different test suites.

First, consider the test suite input set {(5, -2)}. S1 would be executed, S2 would be executed, S3 would be executed because 5 is greater than -2, S4 would be executed, S5 would be executed, S6 would be executed, S7 would be executed because -2 is less than or equal to 0, and S8 would be executed. Thus, this test suite has statement coverage of 100% (even though it consists of only one test and there are eight statements).

Note, however, that not every branch is executed. In particular, x>y is covered but x<=y is not covered. Similarly, y<=0 is covered, but y>0 is not. It turns out that all branches can be covered with the test suite input set {(5, 2), (-2, -1)}. The first test covers x>y and y>0, and the second test covers x<=y and y<=0. Hence, this test suite has branch coverage of 100% (even though it consists of only two tests and there are four branches).

While this test suite covers all of the branches, it does not cover all four of the paths: it only covers two. However, the test suite input set {(5, 2), (-2, 5), (-1, -2), (-2, -1)} covers all four paths. In particular, the test input (5, 2) covers the path S1, S2, S3, S4, S5, S6, S8; the test input (-2, 5) covers the path S1, S2, S4, S5, S6, S8; the test input (-1, -2) covers the path S1, S2, S3, S4, S5, S6, S7, S8; and the test input (-2, -1) covers the path S1, S2, S4, S5, S6, S7, S8.

Obviously, regardless of the measure being used, a test suite with greater coverage is preferred to a test suite with lesser coverage. However, large test suites take longer to execute than small test suites. Hence, it is important to think about the relationship between the number of tests and coverage. In particular, notice that even though there are eight statements, it is possible to have 100% statement

4 Note that, because each edge joins a unique pair of nodes, paths can be identified either by their edges or by their nodes. Since every node is labeled and every edge is not, it is more convenient to use the nodes in this case.

coverage with only one test. Also, notice that even though there are four branches, it is possible to have 100% branch coverage with only two tests. However, there is no way to have 100% path coverage (i.e., cover all four paths) with fewer than four tests.

A point to note about path coverage is that in most programs it is impossible to achieve 100% path coverage with a reasonable test suite. Consider a sub-program with a loop that may execute as many as a million times. The number of paths through this sub-program is at least one million (bypassing the loop, going once through the loop, going twice through the loop, etc.). It would require at least one million test cases to cover all these paths. The definition of path coverage can be modified to take account of loops in various ways and so generate higher path coverage metrics, but when counting *all* paths through programs, 100% path coverage is generally unachievable.

Furthermore, while 100% path coverage is laudable, it is not a guarantee of correctness. To see why, consider the following specification for a tax calculator.

> The tax due must equal to 1% of the assessed value but must not exceed $1000.

Now, suppose that this functionality is provided (incorrectly) by the following function (in Python).

```
def tax(value):
   result = 0.01 * value
   if (value > 1000):
      result = 1000
   return result
```

Further, suppose that this function is tested with the suite input set {100, 200000} which has anticipated (correct) output values of {1, 1000}. This test suite has 100% statement coverage, 100% branch coverage, and 100% path coverage and the function will produce correct output for both tests. Nonetheless, this function contains a fault. Hence, though unit testing would seem to imply that this implementation is free of faults, in fact, it contains a major fault.

The functionality can be provided correctly by the following function that contains a subtle but important change in the condition of the if statement.

```
def tax(value):
   result = 0.01 * value
   if (result > 1000):
      result = 1000
   return result
```

This implementation produces exactly the same output for the two test cases and also has 100% coverage using all three measures.

Ideally, the fault in the first implementation would be identified during verification. However, because it is subtle, it might be missed. This observation, and others, has led some people to advocate a process known as **test driven development** (TDD) in which tests are developed as the code is and for very small increments in functionality [1]; we discuss test-driven development below.

Some tests should be based on the requirements and, ideally, developed before the code. This helps with the analysis phase of construction (it helps the programmer understand the requirement) and, to some extent, documents the code. Other tests should be developed after the code, and should focus on coverage. Because the increments in functionality are small, the creation of these tests is not overly burdensome.

Tests developed based on requirements (before code is written) are, by their very nature, black box tests, and there are two obvious "brute force" ways to proceed. The first is to use complete enumeration and the second is to use random sampling.

Complete enumeration (or **exhaustive** testing) creates a test suite that contains all possible inputs. For example, suppose you need to create a black box test suite for a sub-program that takes a letter grade as its only input and calculates a corresponding numeric grade. It is relatively easy to create a test suite that contains all possible inputs. At first glance, it even seems like complete enumeration will work in all cases. There are, after all, a finite number of both integer values (i.e., 2^{32} in a 32-bit system) and even floating point values (i.e., 2^{64} in a 64-bit system) in a binary representation. However, remember that a test case consists of both the input and the expected output. Hence, without an **oracle** (i.e., another sub-program that is known to output the correct answers), complete enumeration is very tedious. In addition, most sub-programs have more than one input, and the size of the completely enumerated test suite is the product size of the sets of individual inputs. So, for example, consider a sub-program that receives an array of 10 integers as input. A completely enumerated test suite would have $2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} \cdot 2^{32} = 2^{3210} = 10^{320}$ test cases, which is a very large number.

An alternative "brute force" approach is **randomization**, which can be done in either a structured or unstructured fashion. The idea here is to start with (conceptually) the completely enumerated test suite and then select tests from that suite in some probabilistic fashion. In theory, if the distribution of failure trigger conditions is known (or can reasonably be assumed), then one can use probabilistic reasoning to determine the expected number of faults that will remain undetected. In practice, the distribution of triggers is unlikely to be known and the need to generate the expected result for each case remains.

It is important to remember, however, that black-box testing is not "completely uninformed" testing. Specifically, the requirements provide a great deal of information about the kinds of tests that are likely to be failure triggers. One particularly important way to use requirements to generate black-box tests is **boundary value analysis** which is predicated on the idea that failure triggers tend to be found at the "boundaries" of the sets of inputs and outputs. For example, if an input must be in a certain range, test cases should be created using values just below, at, and just above the minimum and maximum limits of the range. Similarly, if an output must be in a certain range, test cases should be created that generate values around the minimum and maximum limits of the output range.

In the tax example above, the requirement itself would naturally lead to inclusion of test cases that would, based on the 1% rule alone, lead to a tax of slightly less than 1000 (for example, an input value of 90000), exactly 1000 (input of 100000), and slightly more than 1000 (for example, input of 110000). The original implementation would fail the first of these tests (producing an output of 1000 when the correct output is 900).

Not surprisingly, there are many other common heuristics. Some popular value-based heuristics for individual parameters include the following.

- The number 0 (because of problems caused by dividing by 0)
- Very large and very small (in both actual value and absolute value) numbers (because of underflow and overflow problems)
- Character or string versions of digits and numbers

Some popular value-based heuristics for groups of parameters include the following.

- Equal values for the parameters
- Different relative values of the parameters (for example, for two parameters x and y, have one test with x greater than y and one with x less than y)

Some popular array and collection heuristics include the following.

- Very small and very large arrays or collections
- Arrays or collections of length 0 and 1
- Arrays or collections that are unsorted, ascending, and descending
- Arrays or collections with duplicated values, and those without duplicates

In some situations it is possible to partition test cases into sets in which each test case in the set is expected to trigger the same failure. Such a set is called an **equivalence class**. The important property of equivalence classes is transitivity[5], which, in the context of three test cases *a*, *b*, and *c*, means that if *a* triggers the same failure as *b*, and *b* triggers the same failure as *c*, then *a* triggers the same failure as *c*. Obviously, if one can identify equivalence classes, then there is no reason to use more than one test case from each equivalence class. To understand how this works in practice, consider a sub-program that is passed an integer representation of the day of the week, with 0 denoting Sunday, 1 denoting Monday, etc. There are three partitions associated with this range—the valid values $\{0, 1, 2, 3, 4, 5, 6\}$, the invalid values that are less than 0, $\{x: x < 0\}$, and the invalid values that are greater than 6, $\{x: x > 6\}$. Similarly, an input that is a single numeric value has three associated partitions (the value itself and both sides of the value), an input that is a Boolean has two associated partitions (true and false), and an input that is an element of a set has two associated partitions (the set and the complement of the set).

Regression Testing

Four things can occur when a developer attempts to correct a fault, as illustrated I the table below.

	No New Fault Introduced	**New Fault Introduced**
Fault Corrected	Good	Bad
Fault Not Corrected	Bad	Very Bad

Project data shows that

- As many as 30% of software changes result in one of the three bad outcomes,
- On average, bad outcomes occur about 10% of the time,
- Faults introduced during bug fixes are more difficult to identify and remove than others.

These facts suggest that programmers should thoroughly test a fix to ensure that it really does correct the defect they intend to correct. Furthermore, because of the likelihood of introducing a new fault, programmers should rerun all tests to try to detect this occurrence. Running *all* tests over again is called **regression testing**. Regression testing should be done after *any* software change, even apparently innocuous changes such as editing comments.

5 An equivalence relation is also reflexive and symmetric.

It used to be that regression testing was an arduous and time-consuming activity. Nowadays unit testing and integration testing tools automate regression testing, so there is no reason whatever not to do frequent regression testing. We discuss such tools next.

Unit Testing Tools

In the early days of software engineering, it was common for testers to to create a custom **test driver** or **test harness** that either contained hard-coded test cases or read test case inputs and corresponding expected outputs from a file, ran the tested unit on the inputs, and compared the results with the expected outputs. Now, however, it is common to use a standard testing framework to scaffold the process.

In most unit testing frameworks, a single test consists of a small block of code that includes an identifier for the test, the sub-program to call, the input values to use, the output values to expect, and an **assertion** that relates actual output to the expected output. Assertions are often stated as equalities (that is, the actual output must equal the expected output). For example, in QUnit (a testing framework for JavaScript), an individual test case for a body mass index (BMI) calculator might look something like the following.

```
test("Underweight Test", function() {
      equal(18.2, calculateBMI(123, 69).toFixed(1));
});
```

The identifier for this test is `"Underweight Test"` and the assertion is that the result of calling `calculateBMI()` with the inputs 123 and 69 (the weight in pounds and height in inches, respectively) must equal 18.2. This example also illustrates an important detail of unit testing that is important to remember—when working with floating point numbers it is important to use either an "approximately equals" assertion or to specify a specific number of significant digits (as is done in this case).

One of the main advantages of using a unit testing framework is robustness. In *ad hoc* testing frameworks it is difficult to isolate different tests. In other words, the test suite may terminate at the first failure or individual tests may change the state of the system and, hence, not be independent. Well-designed unit testing frameworks make it easy to construct isolated tests.

Another big advantage of unit testing frameworks is the way they report results. In general, the reports that are generated are easy to read and highlight the failed tests rather than the successes.

Many unit testing frameworks also include, or can be supplemented with, code coverage tools. These tools provide clear, easy to understand reports about the coverage of individual tests and test suites. In general, these tools require no additional effort on the part of the tester. Continuing with the BMI calculator example from above, after running 13 tests that included inputs for three important partitions, underweight, normal weight, and overweight, the output of a code coverage tool might report the following.

Statements	*Covered*	*% Covered*
9	7	77.78%

```
function calculateBMI(weight, height)
{
      "use strict";
      var bmi;

      bmi = weight / (height * height) * 703;

      if (bmi < 15){
            bmi = 15;
      }else if (bmi > 60){
            bmi = 60;
      }

      return bmi;
}
```

This indicates that the test suite neglected to include important output-related boundary cases and, as a result, did not cover the two highlighted statements (i.e., these two statements were never executed).

Debugging

The discussion of software quality assurance included definitions of the terms defect, failure, and fault. To understand the debugging process fully it is also important to define the terms error and symptom.

A **symptom** is a characteristic of a failure that can be observed.

An **error** (of omission or commission) is something that a person does that gives rise to a fault.

The best way to understand the difference between these concepts is with an example of a very simple software product. The requirements for this product are as follows: it must read an integer from the keyboard, divide 10 by that integer (using integer division), and display either the result or an error message if the user-entered value is 0. The following pseudo-code is (supposed to be) an implementation of that product

```
a ← 10
b ← readInt()
c ← a/b
writeInt(c)
```

If a user executes this fragment and enters the value 20, it will produce the correct output. On the other hand, if the user enters the value 0, the product will "crash" (stop executing). The value 0 is, thus, a trigger condition that causes the product to fail (in particular, to not display an error message). The symptom of this failure is the crash. The fault is the omission of statements that checks the user-supplied value and display an error message.

Now consider the following implementation.

```
a ← 10
b ← readInt()
if b ≠ 0 then
       c ← a/b
       writeInt(c)
endif
```

In this case if a user executes the fragment and enters 0, nothing will be displayed. So, the failure is the same (the error message is not displayed), the trigger condition is the same (the value 0), but the symptom is different (there is no output) and the fault is different (only the statements to display the error message are omitted).

It's important to remember that a failed test case results in one symptom and one corresponding trigger conditions of a failure. Debugging is the process of using trigger conditions to identify and correct faults. It consists of the following steps.

1. **Stabilize**—Understand the symptom and the trigger condition identified by the test so that the failure can be reproduced.
2. **Localize**—Locate the fault.
   - 2.1 Examine the sections of code that are likely to be influenced by the trigger condition.
   - 2.2 Form a hypothesis about the fault.
   - 2.3 Instrument the relevant sections of code.
   - 2.4 Execute the code using the instrumentation to monitor the state of the likely problematic sections of code.
   - 2.5 Prove or disprove the hypothesis. If proven, continue to step 3; if disproven return to step 2.
3. **Correct**—Fix the fault.
4. **Verify**—Test the fix. It is advisable to run all tests to help ensure that the fix did not introduce a new fault with different symptoms.
5. **Globalize**—Look for and fix similar defects in other parts of the system.

This process obviously can be applied to code (that is, the realization of algorithms and data structures). However, it can also be applied to data. Consider, for example, an HTML document. A language-sensitive editor will find all syntax errors, but a structural fault (for example, a `p` element that is incorrectly declared to be an `li` element) may not become apparent until the document is rendered in a browser. The process above would then be used to debug the document. Similarly, consider a CSS document. Again, syntax errors should be found by an editor, but logical mistakes (for example, assuming that all `p` elements are contained in `section` elements) may not be apparent until it is applied to an HTML document, at which point the process above could be used.

Stabilization and verification involve careful use of the testing tools and techniques discussed earlier. Correction and globalization also involve nothing new. Localization, on the other hand, including both the instrumentation and the execution steps, are new and warrant additional discussion. In particular, they involves the use of debug code, a debugger, or both.

Debug Code

Debug code includes both temporary output statements that can be used to monitor state information and temporary input statements that can be used to pause the execution of a component.

When including debug output, it is important to include output identification information along with the output data. One mistake that beginning software engineers make is to include debug code like the following (in Java).

```
for (int i=0; i<n; i++)
{
   System.out.println(i);
   System.out.println(n);
   total = total + sales[i];
   System.out.println(sales[i]);
   System.out.println(total);
}
```

When executed, this kind of debug code generates a screen full of numbers that is virtually impossible to interpret. Instead, one should certainly include information to identify the values, like the following.

```
for (int i=0; i<n; i++)
{
   System.out.println("i: " + i + "(n: " + n + ")");
   total = total + sales[i];
   System.out.println("sales[i]: " + sales[i]);
   System.out.println("total:    " + total);
}
```

In addition, one may want to include debug output statements that indicate that a block, function, or method has been entered, and perhaps even indent the output accordingly, as in the following example.

```
System.out.println("Entering the loop.");
for (int i=0; i<n; i++)
{
    System.out.println("    i: " + i + "(n: " + n + ")");
    total = total + sales[i];
    System.out.println("    sales[i]: " + sales[i]);
    System.out.println("    total:    " + total);
}
System.out.println("Exited the loop.");
```

In some cases, it is convenient to put the debug output in the same stream as the normal output (for example, in a Java console application, use `System.out` for both). This can make it easier to follow the flow of execution. In other cases, this can generate too much output and it is better to put the debug output in a different stream (for example, `System.err` in one or more files).

Experienced developers use a wide variety of techniques to facilitate the use of debug code. Some "comment out" debug code when it is not needed (and some editors provide this capability). Some include debug code in conditional statements and use environment variables or parameters to control which debug statements are executed. Sometimes this even involves multiple "debug levels."

Despite the clunkiness of debug code, it is used quite commonly in practice. Its biggest disadvantages are that it can obscure the code it is meant to debug, it can take a significant amount of time to add, and it can be difficult to know what to output. Hence, its use often involves a trial and error process.

Debuggers

A **debugger** is a tool that can be used to trace the execution of a code without having to modify it in any way (in particular, without having to add debug code). Some debuggers are command-line tools (like gdb) and some have a GUI (for example, front ends for gdb like DDD, and the Java debugger in NetBeans). Regardless, they provide the following functionality.

1. They allow you to set **breakpoints** to pause the execution of a program. That is, they allow you to start executing a program but pause execution when a particular statement is reached.
2. They allow you to execute statements after a breakpoint one at a time. At the lowest level of abstraction, the user can **step into** each statement which, if the statement is a call to a sub-program, enter the sub-program and execute each of its statements one at a time. At a higher level of abstraction, the user can **step over** a sub-program call treating it as if it is a single statement (ignoring its internal details). When executing code in a sub-program, the user can **step out** of the sub-program, continuing execution of its code until it returns, then pausing again after the return.
3. They allow you to use **watches** to monitor state information. For example, as you execute individual statements you can monitor the values contained in different variables to see how they change. In some debuggers, it is even possible to set alerts for certain conditions.

The advantage of debuggers is that they allow you to trace the execution of a piece of code in all of its gory detail. This is, unfortunately, also their downfall. It can take a very long time to localize a fault. Hence, it is quite common for debuggers and debug code to be used in concert.

Performance Optimization/Tuning

For some products, though certainly not all, performance is an important measure of quality. When constructing such products (and some others as well), it sometimes becomes necessary to *optimize* or *tune* the performance of individual sub-programs and the system as a whole.

Some performance tuning amounts to little more than being careful and precise. For example, consider the following fragment in C that determines if an array named `data` contains the value named `target`.

```
found = false;
for (int i=0; i<n; i++) {
   if (data[i] == target) found = true;
}
```

This implementation loops over all `n` arguments whether or not `target` is present in the array. One could, instead, use a **sentinel** as in the following implementation.

```
found = false;
for (int i=0; !found && i<n; i++) {
   if (data[i] == target) found = true;
}
```

Another example of this is the elimination of unnecessary operations in a loop. The following fragment in Python scales x- and y-coordinates to fill a window.

```
for i in range(len(x)):
        x[i] = x[i] * (width / windowWidth)
        y[i] = y[i] * (height / windowHeight)
```

Both the readability and the efficiency of this calculation can be improved by pre-computing the scaling factors as follows.

```
xScale = (width / windowWidth)
yScale = (height / windowHeight)
for i in range(len(x)):
        x[i] = x[i] * xScale
        y[i] = y[i] * yScale
```

On the other hand, some performance tuning is based on empirical evidence, which involves the collection of data. For example, nested conditions (for example, in nested if statements) can be ordered to reduce the number that need to be evaluated based on the frequency of different data values. Other times this involves the collection of data about execution times. In general, a small percentage of the lines of code in a program are responsible for a large percentage of its execution time. Of course, the "offending" lines of code must be identified, and that can be done with a tool called a *profiler*.

Profiling is the process of measuring the amount of time (or space) used by sub-programs and statements during a particular execution of a program. Since it occurs while the program is executing, it is a form of dynamic analysis. A profiler does not provide any information about why the offending lines of code require as much time as they do or how improvements can be made; it just collects and presents the data.

The following is an example of the report generated by profiling the execution of an *n*-dimensional minimization algorithm (the Cyclic Coordinates Algorithm) that makes repeated use of a 1-dimensional minimization algorithm (the Golden Section Algorithm).

Function	Calls	Percent ▼	Own Time	Time	Avg	Min	Max
nD_to_1D/<	7856	39.22%	87.975ms	132.862ms	0.017ms	0.014ms	0.345ms
argminGoldenSection	92	29.1%	65.274ms	220.458ms	2.396ms	1.951ms	9.082ms
anonymous	7857	20.02%	44.894ms	44.894ms	0.006ms	0.004ms	0.097ms
equals	4020	9.7%	21.757ms	21.757ms	0.005ms	0.004ms	0.058ms
CCapp.prototype.solve	1	0.7%	1.569ms	224.164ms	224.164ms	224.164ms	224.164ms
argminCyclicCoordinates	1	0.64%	1.44ms	222.588ms	222.588ms	222.588ms	222.588ms
nD_to_1D	92	0.31%	0.69ms	0.69ms	0.008ms	0.005ms	0.054ms
midpoint	92	0.25%	0.565ms	0.565ms	0.006ms	0.004ms	0.098ms
registerListeners/<	1	0.06%	0.126ms	224.291ms	224.291ms	224.291ms	224.291ms

It shows that 39% of the time is used to evaluate the function being minimized and 29% of the time is used to solve the 1-dimensional problems. Little can be done about the first, but it may be possible to tune the 1-dimensional minimization algorithm or replace it with an alternative.

Refactoring

Refactoring is changing code without altering its behavior. Refactoring is done to improve the structure, presentation, or performance of code. Because refactoring does not change the behavior of code, there must be some mechanisms to help insure this. Hence a thorough test suite in a regression testing framework must be in place before refactoring can be done with confidence.

Refactoring should be done when any of the following conditions obtain.

Duplication—Some portion of the code is duplicated. This may occur with sequences of statements (suggesting a loop, sub-program, or super-class may be needed), expressions (suggesting a function may be needed), and manipulations or declarations of variables (suggesting that collection may be needed), data (suggesting that data structures may be combined or eliminated).

Lack of Clarity—The code is hard to understand. The over-arching aim is to simplify the code as much as possible, and adding more hints for readers. This typically requires renaming variables or sub-programs, changing control structures, reorganizing control structures, adding comments, and so forth.

Code Smells—The term **code smell** is used by the refactoring community for a broad category of code defects. Code smells include comments that duplicate code, classes that do nothing but hold data, failure to hide information, tight coupling, low cohesion, bloated classes, lazy classes (that are too small), long methods, and reliance on switch statements instead of classes.

There are several common refactorings, some of which are supported in IDEs. The following are examples of common refactorings.

Rename—Give a variable or sub-program a new name. The point of this is to make the code more readable. Sometimes choosing a better name can make a comment unnecessary.

Introduce explaining variable—If an expression is long and difficult to understand, introducing a temporary variable with a good name can often help make the code more meaningful. For example, consider the following code fragments.

```
(a) if ( 0 <= (b*b - 4*a*c)) { … }

(b) var quadratic_has_real_roots = (0 <= (b*b - 4*a*c));
    if (quadratic_has_real_roots) { … }
```

The temporary variable explains what is going on in this code without a comment.

Inline temporary variable—This refactoring is the opposite of the previous one. Sometimes introducing a temporary variable can help readability, but usually these variables are not necessary and just add the cognitive load of program readers. Consider the following code fragments.

```
(a) var fraction = x%1;
    if (0 !== fraction) { … }

(b) if (0 !== x%1) { … }
```

In this case the expression is so simple that the extra variable only adds complexity to the code.

Extract sub-program—Decompose a sub-program into two or more sub-programs. There are many occasions to use this refactoring. The most obvious is when a sub-program is very long and complicated—it should be broken into simpler sub-programs. Another is when code is duplicated either in several sub-programs, or in the same sub-program, perhaps with minor differences that can be captured by sub-program parameters.

Inline sub-program—This is the opposite of the last refactoring. This refactoring should be used when a sub-program is simply too short to warrant being a separate entity. Obviously, there is a lot of judgement (and disagreement) over when this is the case.

Replace a conditional with polymorphism—A conditional that takes different actions depending on the type of an object can be replaced with code that processes objects of sub-classes of the original object, each with an overridden method that encapsulates the specialized behavior.

Make template method—Move the essentials of methods in sub-classes to a super-class, and customize the method so it uses features of the sub-classes.

Etcetera—There are many more common refactorings. Martin Fowler's book *Refactoring* [3] lists over 70 of them.

The refactoring process encourages making only small changes and then immediately rerunning all tests. This ensures that the behavior of the code is not changed. Here "small changes" means changing only a few lines of code at a time. The overall refactoring process proceeds a follows.

1. Run tests to make sure they all pass. Fix the code so it passes all tests.
2. Identify a refactoring.
3. Make a small change that gets the code closer to the refactored code.
4. Run tests and fix whatever is broken.
5. If the refactoring is not complete, goto step 3.
6. If the code needs further refactoring, go to step 2.

Test Driven Development

Traditional testing views all testing, even unit testing, as an activity distinctly separate from coding. The paradigm is that developers write code, then write tests to try to get code to fail. Treating testing as a separate activity is probably a good approach for various forms of system testing because system testing focusses on determining whether a product meets requirements. Testing a system against its requirements demands a more-or-less fully functional product to test, and it accommodates a black-box testing approach. Hence it is perfectly appropriate (and probably better) for a separate testing organization to system test a complete product.

But unit and integration testing are different. The test cases should be based on clear-box criteria, not black-box criteria. In unit testing particularly, small pieces of code should be exercised comprehensively to help ensure correctness. None of this can be done properly by anyone but the coders themselves. If the coders treat testing in a traditional manner as an after-the-fact activity, then the following things tend to happen.

- Few tests are written because the coders regard their job as finished already—testing is just a perfunctory activity to confirm what has already been accomplished.

- Poor tests tend to be written because the coders are writing to the code that they have just completed. For example, if a coder has just written code to check a parameter, then a test with a check for a bad parameter value is written. The problem is that the test will assume that the parameter will be bad in just the way that the coder is testing for, so this test is not really likely to fail.

Few and poor unit and integration tests will find few defects.

Traditional approaches to testing also do not assume testing frameworks. People used to write all tests as custom pieces of code. This was a huge amount of work. Also, regression testing was more difficult because the test code could easily become unusable as the product code evolved. Hence the entire process was slow, cumbersome, and expensive. The introduction of standard and easy-to-use testing frameworks, in particular the *xUnit* family of testing frameworks, has changed all this radically.

Test-driven development (**TDD**) is a style of coding in which writing tests is integral to the coding process itself. The TDD process results in many tests that check behavior in ways that do not depend on the code (because they are written before the code). The result is many tests of generally high quality. TDD also assumes a testing framework, so it is easy to write and execute tests, and they can be saved and rerun frequently—in fact constant regression testing is part of TDD.

The following are the essential features of TDD.

A testing framework is assumed. The framework must allow suites of tests to be written separately from product code. The framework must provide facilities to rerun all tests with almost no programmer effort and to report failures with details about expected and actual test results.

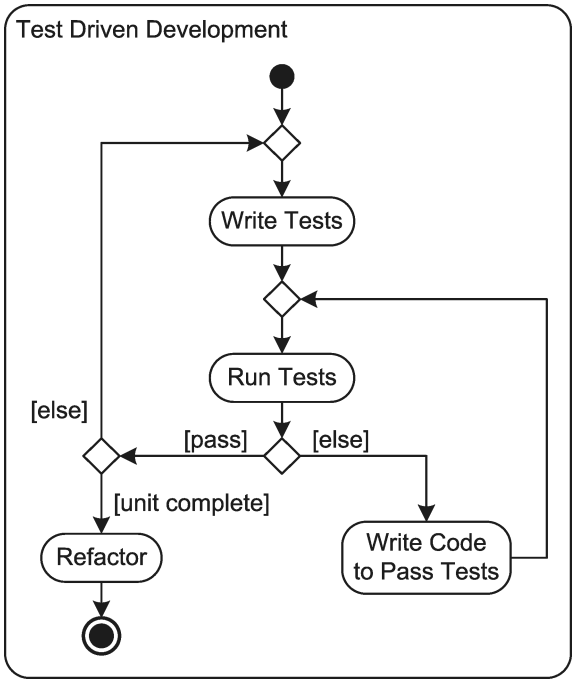
Tests are written before code. The coder first writes tests that define the expected behavior of the code that needs to be written. Then the coder writes code to pass the tests. This helps keep coders from writing poor tests that simply check the conditions that the programmer has written into the code.

Tests and code are written incrementally. In implementing a program unit, a few tests are written to check part of the function of the unit. Then code is written to pass the tests. Then a few more tests are written and code to pass them is written. This encourages coders to write extensive and thorough tests.

Code is only written to pass tests. No code is written unless doing so will help pass a test. If a coder wants to add a feature to a program unit, then she must first write a test for it. But if there is no need for the feature, then there will be no motivation to write the test. The idea of this caveat is to keep programmers from adding unnecessary features and complexity to code. Agile methodologists call this "doing the simplest thing that could possibly work."

Refactoring is expected. When code is added incrementally to enable a unit to pass more and more tests, the result may be very poorly structured. TDD encourages refactoring code to improve its structure, readability, and so forth. Note how well this approach supports refactoring: a thorough suite of tests is needed for refactoring, and that is exactly what TDD produces prior to refactoring.

The diagram below illustrates the TDD process.



As this diagram shows and as we have stressed, TDD begins with writing tests, then running them, then writing code to pass them. Once code meets all tests for the unit, then refactoring takes place using the now complete unit test suite.

References

1. Beck, K. *Test Driven Development: By Example*, Addison-Wesley, 2002.
2. Beizer, Boris. *Software Testing Techniques*, Van Nostrand, 1990.
3. Fowler, Martin. *Refactoring: Improving he Design of Existing Code*, Addison-Wesley, 2000.