COM6120/3210 - Software Measurement and Testing
Assumed Background Material on Software Testing

This material explains the background to software testing, which for undergraduate students has been covered at level one. It will be overviewed quickly at the start of the course, but students who either have not covered this (such as MSc students), or who have forgotten it, should read it carefully and ensure that they understand it.

Motivation for Testing

The reason why testing is necessary is because **errors** are unavoidable. Mechanical failures, such as faulty memory devices or noisy network cables lead to errors. Similarly, humans are fallible, and make errors. Mostly the errors are due to the way human memory works; they cannot be avoided by "trying harder" or by punishing people who make them. The topic of how the memory works, or at least why it works unreliably, and how this affects software development, is sometimes described as the **psychology of testing**. Its basis is that the memory is organised on three levels.

Short term memory. This is used for remembering pieces of information you are actually working on at a given moment. Unfortunately it is very small (hence the significance of the formula "seven plus or minus two"), and operates like a stack. So, when you need to remember a new item in short term memory, you forget one of the older items. Worse still, there is no reliable overflow flag, so you don't get any warning that this has happened. Typically, this leads to what are usually called "trivial" errors, meaning simply ones that can't easily be attributed to any other cause. These include typographical errors, wrong variables, array indexes out by one, etc.

Medium term memory. This is used during "problem solving", and it contains all the information relevant to the task in hand, including the method or the line of reasoning you are using. Not knowing the answer to a problem leads to a build up of mental tension, whereas finding an answer results in a "peak experience", and a release of the tension. The subconscious desire for this peak experience means that you are likely to jump to any plausible looking solution without fully reasoning it out. Also, once you have found a solution, you will be reluctant to start thinking about it again, or to try and adopt a new method.

Long term memory. This holds your "World view", and so it supplies relevant information for problems that you are working on, together with potential solutions. Unfortunately, though, your World view is almost certainly inconsistent, and in ways of which you are unaware. In particular, you tend to see what you already know rather than what is actually there (since it is much easier to use your existing world view than it is to extend or alter it).

These psychological aspects lead to the conclusion that errors are here to stay, and that we cannot hope to totally eliminate them. Moreover, it won't do any good just to try harder, or to penalise people for making them (they'll just try to cover them up). Instead, software developers have to plan for errors, aiming to spot when an error has been made so that remedial action can be taken as soon as possible.

Errors in Software Processes

During a software project, errors can occur in any part of the development process. For instance, during requirements capture errors can be made by an analyst, due to misunderstandings between the analyst and the clients, which reflect the limitations of long term memory. Similarly, errors can be made in the design process in choosing and evaluating solutions that meet the requirements, and in planning the development process, and these reflect the limitations of both long term and medium term memory. Then, during the implementation process, errors can be made in writing code that fulfills the design, and these will reflect the limitations of short and medium term memory.

Any error that is made during the development process is likely to manifest itself as a **fault** in the product of that process. For instance, such a fault could be:

- a misstated or misemphasised or missing attribute requirement;
- an incomplete or inconsistent specification;
- an inadequate design; or
- a buggy piece of code.

The end results of any such fault are likely to be **failures** in the system's operation. In other words, if some aspect of the system's behaviour differs from that which the end user expects, then this is a failure. Of course, there is a weakness in this definition, in that the end user might have unreasonable expectations, so it may need to be tempered in various ways: eg "a reasonable end user ought to be prepared to accept".

In addition, one can also have faults that exist in a software product, but have not yet resulted in a failure occurring. These are called **latent faults**.

The approach of defining errors, faults and failures in this way then leads to the following definition of testing, which is based on the one given by Myers, but updates the terminology. "**Testing** concerns the selection of inputs for a system with the intention of causing failures in the system, and thereby revealing the presence of faults." There are two important features of this definition. One is that it takes a positive view of testing, as the process of finding out something about the behaviour of a system, rather than the negative process of trying to damage or crash it. The other is that it emphasises the part played by selecting suitable sets of inputs, which (as described below) distinguishes testing from other forms of validation or verification.

At face value, this definition might suggest that testing is something that can only be done once the system is complete. Actually this is far from the case, but before discussing it we need to summarise the formal definitions for some of the above terms, on which this definition depends. The following definitions abbreviate the ones given in the IEEE standard 729 of 1983, and are consistent with the more recent definitions in BS7925 Part 1 of 1998.

- A **failure** occurs when the output from a system is in some way unsatisfactory.
- A **fault** is something in the system state that can lead to failures.
- A fault that has not yet led to any failures is **latent**.
- Faults are (usually) the result of **errors**, which are mistakes made (accidentally) by people.

The Activities of Testing

There are a number of basic concepts that are fundamental to the activities involved in testing a system, and these concepts need to be defined. The following are based on BS7925 Part 1 of 1998.

A **test case** is one particular (set of) inputs to the system, together with the expected (correct) outputs that should result from using those inputs.

A **test set** is a collection of test cases.

A **test script** is a collection of test cases or sets together with the commands needed to carry them out.

Test execution consists of running the system on each of the inputs described in a suite (set), and noting the actual outputs.

Test analysis consists of comparing the actual outputs with the expected output.

A **test suite** is not defined formally, but the term is sometimes used to refer to a (collection of) test sets which is designed to fully' test (some aspect of) a system, and so relates to a particular version of a system, the purpose of the tests, the set up required, and the means of executing the tests.

Testing is defined formally as the “process of exercising software to verify that it satisfies specified requirements and to detect errors”, but informally the term is also used to refer to any of the related activities. These include: constructing test cases, sets or suites; executing tests; analysing test results; constructing or using tools to help with any of these; planning any of them. Implicitly, testing is distinguished from other forms of verification or validation by being based on the selection and use of individual test cases, or sets of test cases.

A Brief History of Software Testing

The following history is organised into a set of periods, as identified in a paper by Gelperin & Hetzel[1]. Note that these periods correspond to their assessment of the state of the art (as defined by some key publications, the details of which are not included here), as compared with standard practice in the software industry, which typically was some way behind this. Note also that their periods typically lasted for less than ten years, so we have probably completed at least one more since their paper was published, but as yet nobody has made a significant attempt to update their analysis.

Debugging oriented period: up to 1956. In the early days of digital computing, most problems were in the hardware, and so this is where the emphasis lies in all the published work on testing from this period. A programmer would construct a a program (a laborious and error prone process in itself), and would then "check it out" to see if it behaved "correctly". Consequently, there was no clear distinction between testing, debugging and program check-out.

Demonstration oriented period: 1957-1978. In this period, program check-out was divided into two distinct activities: debugging the program (ie ensuring that it ran without crashing), and testing the program (ie ensuring that it solved the problem). It was during this period that computers came into widespread use, and many large complex (and therefore expensive) systems were developed. Testing was therefore directed towards detecting problems before product delivery.

[1] D. Gelperin & W. Hetzel, The Growth of Software Testing, *Communications of the ACM*, **31**.6 (June 1988), pp 687-695.

Destruction oriented period: 1979-1982. This marked a change of emphasis, in that previously, successful tests were ones that gave the correct answers, and so revealed no faults. During this period, the goal of testing became instead to try to cause failures, and so to reveal faults (as described by Myers). Along with this, the notion of debugging also changed, and it became instead the process by which the faults detected by testing were located and corrected.

Evaluation oriented period: 1983-1987. Testing became integrated with other forms of validation and verification, such as document review, and appropriate techniques were applied at the end of every stage of the software life-cycle, so as to provide product evaluation information. This approach is typical of quality assurance methodologies, as required by government bodies etc., such as Defence Standard 00 - 55, the US National Bureau of Standards NBS FIPS 101 (1983), Prince, and others.

Prevention oriented period: 1988 onwards. The emphasis is on using testing methods to try to avoid faults, rather than just on detecting or measuring them.

Modern Software Testing

Instead of the approach that testing simply means test execution, and so could not begin until at least some of the system was complete, the prevention oriented approach is based on a process of developing tests, running in parallel with the process of developing the system. Its basis is described by Hetzel as the realisation that "*one of the most effective ways of specifying something is to describe (in detail) how you would recognise (test) the something if someone ever gave it to you*". Developing this gives rise to testing activities that can usefully be carried out at **every** stage in a system's development. For example:

- specifying attribute requirements (ie requirements for non-functional properties, such as performance, or ease of use) in terms of measurable scales gives the basis for acceptance tests for a system;
- applying black-box testing techniques to a specification can identify inconsistencies and ambiguities before they turn into design faults; and
- writing test scripts before writing code can help prevent many faults from being included in the code.

This modern approach therefore uses testing at all stages of the software development process so as to try to manage the risks in the process. That is, modern testing can be used to prevent problems (ie errors, faults and failures) by good practice, to evaluate capabilities (so as to avoid overloading development organisations or their systems), to detect problems (eg by document review, or actual product testing), to demonstrate performance and to predict performance. For the purposes of this module, however, the main emphasis will be on the practical techniques that are used in the more traditional approaches to testing, rather than on the broader issues of project management and quality assurance.

Before leaving these broader issues, though, it is important to note that what the modern approach to testing tries to achieve is very similar to any other forms of verification and validation, namely to show:

- that the system being developed is the one which the client wants (ie validation); and
- that this system is being built correctly (ie verification).

What is distinctive about testing, as opposed to these other forms of v&v, is that it relies on the selection of specific test cases to demonstrate correctness, rather than trying to produce more general arguments or demonstrations.

Performing Testing

The most important issue in actually performing testing, whether for requirements documents, specifications, designs, individual programs or complete systems, is that of how test sets can be constructed for these artefacts. In theory, the ideal way to test any sort of system would be to conduct an **exhaustive test**: that is, to use a test set which contained every possible test case. In practice, for even the most trivial system the number of test cases in the test set for this would have to be so large as to be totally unmanageable. Therefore, a **testing method** is required: that is, a method which given some information about some system, will construct the test set for it, in such a way that the resultant test set possesses some desirable properties.

Properties of Testing Methods

Most of the desirable properties of testing methods can be described quite simply at an intuitive level, but actually defining them formally is a much more complicated task. They can be described as follows, but these should not be taken as precise definitions of terminology.

- The constructed test set needs to be of a realistic size, and it should be possible to control this size. What constitutes a realistic size will, of course, depend on the complexity of the system to be tested, and could be anywhere from tens to thousands of test cases.
- For any given system, if a bigger test set is constructed then this should somehow be better than a smaller one, in some measurable way. This makes it possible to conduct a more thorough test if the nature of the system requires a higher level of reliability to be achieved.

- Any constructed test set should cover realistic test cases, where realistic here means that these cases will cover aspects of the system that users would regard as more important than the aspects that would be covered by other cases not in the test set.
- Any constructed test set should be a representative one. This is based on the idea that an individual test case is a representative one if there is some other set of test cases such that one can reasonably assume that if any of these cases would identify some particular fault, then the representative one will as well. A complete test set will then be representative of the union of these individual sets of test cases: the larger this overall set, the more representative that test set is.
- Any constructed test set of a certain size should be adequate, according to one or more adequacy criteria. Each adequacy criterion is intended to relate to some particular class of faults, in such a way that if a test set for a system is adequate according to that criterion, then the set is guaranteed to find any faults of that class in the system.

The basic problems in trying to formalise these notions arise because the links between errors, faults, failures and tests that will identify them are almost impossible to model at a formal level: indeed, trying to determine how faults might affect the operation of a system (ie whether it will fail or not) is in many cases a problem for which the solution is not computable. This makes it very difficult to define meaningful adequacy criteria, and most of the ones that are used in current practice have been developed by observing experimentally properties that seem to be useful in finding particular classes of faults, rather than from rigorous theory.

Also, one can only assert that one test case is representative of another by asserting that there can not be any faults in the system which might cause it to exhibit a failure for one case but not the other. Consequently, the whole notion of test cases or test sets being representative depends on assumptions about the kinds of faults which might be present in a system, whereas part of the purpose of the testing may have to be to establish whether those assumptions are actually true in practice.

Types of Testing Methods

It is convenient to divide the various testing methods that are available into four main groups.

Random Testing. These methods rely on generating test cases at random, which (because they ignore the fact that most systems require their input to have a fairly precisely defined structure) will usually not be either realistic or representative. On their own, therefore, these methods are of little practical use, but from the theoretical point of view they can provide a way of characterising the minimum baseline of performance, which any useful method should be able to improve on, and this property can be regarded as significant.

Specification-Based Testing. These methods treat the system as a "black box", and so construct the test sets solely from the information given in whatever "specification" (which may or may not be a formal one) is supposed to define the behaviour of this black box (ie the relationship between its inputs and its outputs).

Implementation-Based Testing. These methods treat the system as a "white box" or “glass box”, meaning one whose contents are known, and so they construct the test sets from information about the internal structure of the implementation of the system.

Hybrid Testing. These methods try to combine these approaches, and they split into two subgroups. One subgroup combines the black box and white box approaches by using information about both the specification and the internal structure of a system to construct the test sets. The other subgroup is the statistical methods, which use information about the specification of a system to guide and structure the process of generating random test sets.

For the purposes of this module, the methods that will be considered are the specification-based ones (ie black-box methods, often called **functional** testing methods). Consideration of the other kinds of methods is deferred to more advanced courses.

The reason for this is that testing methods which are based on the specification of a system have several advantages over those based on its implementation. In particular, functional methods allow the design of test cases to begin earlier (ie as soon as an initial specification is available), with the benefit that the act of generating test cases leads to a better understanding of the system, and can be used to make improvements at the specification stage. Also, the test set that is constructed will be based on what the system is supposed to do rather than on what it actually does, so that faults of omission (such as not implementing bits of the requirements) are more likely to be detected.

Concepts of Specification-Based Methods

There are three basic concepts that are fundamental to most functional testing techniques. One is the notion of a system as providing a set of functions (ie use cases), which in principle can be tested separately. In practice,

though, there is often some sort of ordering between the different functions, which arises because some functions (which will get data into the system) will need to be working before others can even be run at all. Such an ordering will therefore affect the way in which the system is tested, in that these "input" functions or use cases will need to be tested first, to make sure that they are working correctly, before it is possible to go on to test the other functions which use the data that they have input.

Separating out the functions in this way is quite an important step in simplifying the problem of constructing test sets, because it means that one can concentrate on testing one function at a time. The other two concepts then apply to the way in which one analyses the behaviour of an individual function. One of them is the concept of **equivalence partitioning**, which involves the identification and use of representative values for the inputs to that function (where representative has the meaning described above). The other is **boundary analysis**, which is the targeting of areas of the problem where faults are most likely to occur, and to result in failures.

Any data set that is used as input to or output from a function can be **partitioned** into subsets of elements that "are the same" in some important way. These subsets are referred to as **equivalence classes**, even where (as is quite common) it consists of a single value. A single test case from each equivalence class should then act as a representative value for that class. For any system, the following data sets can be partitioned:

- The set of possible inputs (which can obviously be partitioned into invalid and valid inputs);
- the set of possible outputs; and
- any hidden internal data sets (where the fact that they are internal means that there may not be much known about them at the requirements or specification stages, although they will probably appear in the data model for the specification).

Between the equivalence classes of any data set are **boundaries**. Errors are especially likely to occur at boundaries, and so choosing test cases at or "close" to the boundary makes it easier to detect the faults that result from these. Thus, the basic steps in applying boundary analysis are:

1. Decide where the boundary should be, denoted by b.
2. At the very least, choose a test case corresponding to the value b (ie this single value becomes an equivalence class).
3. If more concentrated testing of this boundary is required, then work out the smallest amount the boundary could be wrong by and denote it by ε.
4. Choose additional test cases (ie equivalence classes) corresponding to the values $b+\varepsilon$ and $b-\varepsilon$.

Of course, in doing this b and ε will not necessarily be numbers on a linear scale - for example, for many data sets they could be vectors, or other structures - and so the interpretations of + and - will need to be adapted to suit.

Identifying Equivalence Classes

The basic problem in specification-based testing is to identify the sets of data values that need to be partitioned into equivalence classes, and then to decide which of those possible equivalence classes are important enough to warrant testing separately. In the case of simple variables the solution is usually straightforward: for instance, if an item of data is a single integer variable which must lie in the range min to max, then there are really only three possibilities. The most basic one is to treat the whole range as a single equivalence class, but this is unlikely to test the system adequately. A much better solution is to have three equivalence classes, covering the following ranges for the value v of this variable:

- $v = min$;
- $min < v < max$;
- $v = max$.

If it is considered necessary to test the boundaries of this range thoroughly, then the full boundary value analysis can be used, and this would give five equivalence classes, as follows.

- $v = min$;
- $v = min + 1$;
- $min + 1 < v < max - 1$;
- $v = max - 1$;
- $v = max$.

For data that has a more complex structure, then there will usually be several properties that could be used as a basis for identifying possible partitions. For instance, if an object has several attributes, then one might well want to treat each individual attribute as a separate property, and partition the values of that attribute into equivalence classes. Or, if the data has a structure such as a list of items, then the length of the list would be an obvious property to use, and there may be other properties related to individual items in the list.

As an example of this, suppose that a mail-order firm wants to use a computer system to handle customer accounts. In particular, at the end of each month, they require the system to print out a statement for each customer, detailing their purchases during the month together with their current credit rating. The system must also send out invoices to customers with outstanding payments, and should not send anything to customers who bought nothing in the past month, and owe nothing for earlier items.

For this example there are several obvious properties that could be used to partition the set of customers, and each of these has some boundary cases associated with it.

Size partition: Since there is no obvious upper limit to the size of the set of cutomers, it could be partitioned into three cases: no customers (ie an empty set); one customer; and many customers.

Balance partition: Assuming that there are some customers, they can be partitioned according to their balance of payments at the end of a month: those who are in credit (balance $\geq$ 1p); those who are owing (balance $\leq$ -1p); and those who are exactly balanced (-1p < balance < 1p).

Number of orders partition: Again, assuming that there are some customers, they can be partitioned according to how many orders they placed in the past month, and the number of these for which details will fit on one sheet of paper: no orders placed; up to one sheet of orders; over one sheet of orders.

Address partition: Again, assuming that there are some customers, they can be partitioned according to their addresses: within UK (normal postage); outside UK, but within EC; rest of the world.

This therefore gives four different ways to partition the customers, although for one of these (the size partition) there is one of the equivalence classes (the empty set) for which there is only one possible test case. For the other two partitions, the equivalence classes are completely independent of those produced by the other ways of paritioning the data, and these different ways also give quite independent sets of equivalence classes (three classes for each). Thus, in principle, all the different combinations of these should be tested, meaning that there are $2 \times 3 \times 3 \times 3 = 54$ different combinations of equivalence classes, and so one would expect to have to construct a representative test case for each of these, plus one for the empty set case. In practice, though, one might be able to cut this down, since the fact that every customer in the set is to be processed means that it may be sufficient to do a single test for the equivalence class "one customer in the set", rather than trying to repeat all 27 combinations that could occur for this class.

This example serves to illustrate the kind of problems that arise in identifying the different equivalence classes, and then constructing the sensible combinations of them. What is therefore required is a systematic method for doing this. There are two important specification-based methods that have been developed for doing this: one is known as the category-partition method, and the other as the cause-effect graphing method. These will be described in detail later in the course.

The Testing Process

The key issue in actually doing testing is that the methods described above can be applied at different levels of detail within the structure of a software system. Typically, the following are commonly identified.

Unit Testing is carried out at the level of small units of code, such as individual methods in a class. There are now tools available (such as JUnit) to assist in constructing such a test suite for a class, to make it easy for the suite to be rerun each time that any change is made to the code of the class. Typically the test cases will be constructed from design documentation, such as a CRC card for the class.

Integration Testing is carried out at the level of a subsystem, which in terms of a system written in Java could mean anything from a small group of a couple of classes, through a complete package, to a complete program within a system that consists of a number of programs. Typically the test cases are constructed from a specification (possibly formal) of the subsystem, or from a subset of the system requirements document.

System Testing is carried out at the level of the complete system, so that the test cases are typically constructed from either the system requirements document or a formal specification of the system. It is often organised in two stages: **alpha testing** is done by the developers, and **beta testing** by selected users.

Exploratory Testing is the form of system testing that testers have to use initially, if they do not have a requirements document on which to base any other form of testing.

Non-Functional Testing is usually also carried out at the level of the complete system, but whereas the above kinds of testing are all concerned with whether functional requirements are met correctly, non-functional testing is concerned with how well requirements for performance, quality, etc are met.

Stress Testing is a particular kind of non-functional testing, which focuses on whether the system can meet particular extremes of non-functional requirements.

Acceptance Testing is system testing done by the clients, involving any agreed combination of functional, non-functional or stress testing, in order to determine whether the developers have done enough for the clients to accept the system (and pay for it).