CITS5501 Software Testing and Quality Assurance
Introduction to testing; unit testing

Lecturer: Arran Stewart

Highlights

- Testing concepts
- Documenting code
- APIs
- Unit testing
- Testable documentation
- Property-based testing

Highlights

- Documentation and APIs: how do we work out what the correct behaviour of a piece of software *is* so that we can test it?
- Unit testing: What is unit testing, what is the terminology, and how do we write unit tests?

Testing concepts (cont'd)

Faults and failures

Last lecture, we were talking about *faults* and *failures* (and *erroneous* or *inconsistent states*).

In normal English, we might not make much distinction between them.

But in software engineering, it can be useful to distinguish whether we're talking about

- the *behaviour* of a running system (what we can observe about what the system does)
- the *static artifacts* from which the system is produced (e.g. files of source code, or data files in formats like HTML or CSS)
- the *runtime state* of the system (i.e., what's currently stored in memory)

(Aside: terminology)

We will use terminology that's roughly consistent with:

- Ammann & Offutt,[1] and
- Bruegge & Dutoit[2]

but you may find other sources that use different terminology.

(In particular, sources vary greatly in what they consider "error" to mean.)

[1] Amman, P. & J. Offut, *Introduction to Software Testing* (2nd edn, 2017)

[2] Dutoit, B.B. and A.H. Dutoit., *Object-Oriented Software Engineering: Using UML, Patterns, and Java* (3rd edn, 2014).

Failures

Recall that:

- A *failure* is any deviation of the observed behaviour of a program or system from the specification.

It describes the system's *behaviour*.

(Can we say a file of source code contains a "failure"? No. But can we say a failure occurs when some program is run? Yes.)

Failure examples

If a program should save a document when the user types "`ctrl-s`", but instead crashes when the filename contains a space – that would be a failure.

- The program would be failing to meet a *functional* requirement

Failure examples, cont'd

If a program should always respond to user input within 0.1 seconds, but instead starts "lagging" when more than 5 documents are open – that's also a failure.

- This time, the program would be failing to meet a *non*-functional requirement, that the system meet particular standards for *responsiveness*.

Failure examples, cont'd

If an electronic voting booth should accurately record votes cast, but due to a cosmic ray flipping a bit in memory, 4096 additional votes are counted for one candidate – that's also a failure.

Some questions:

- In this case – what sort of requirement has the system likely failed to meet: functional or non-functional?
- Is there any corresponding defect?
- And is there any reasonable way of preventing or avoiding the failure?

Faults

Also called *bugs* or *defects*.

- A *fault* is something in the static artifacts of a system that (when executed or otherwise encountered) will cause a failure.

Faults

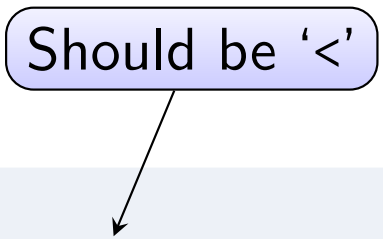
Also called *bugs* or *defects*.

- A *fault* is something in the static artifacts of a system that (when executed or otherwise encountered) will cause a failure.

For an example of a fault, consider the following Java code, intended to iterate over an array of book titles and print them out:

```
for(int i = 0; i <= book_titles.length; i++) {
  System.out.println(book_titles[i]);
}
```

Fault example




```
for(int i = 0; i <= booktitles.length; i++) {
  System.out.println(booktitles[i]);
}
```

Broad definition of "fault"

Some sources will use "fault" more broadly to mean the *cause* of a failure besides just defects in the code – e.g. perhaps cosmic rays – but we'll mostly be concerned with problems in the code.

Not every failure can be traced back to a *single* spot in the code: failures of security, scalability, performances etc. are global properties of the system artifacts.

Erroneous state

In the textbooks I've mentioned, when the authors use the phrase "error" or "erroneous state", they mean the situation at runtime, where some fault has become reflected in the system's runtime state.

- So you can have a *fault* in the code (e.g. off-by-one Java loop error we saw), but if we execute the program and (at least this time round) that bit of code doesn't happen to get run, we don't get a corresponding erroneous state.

Invariants

In many cases (for failures of functional requirements, at least), the failure arises because some *invariant* (e.g. a *class* or *program invariant*) has been violated.

An invariant is just a statement – a proposition – which can be true or false about the runtime state.

E.g.: "The value of `i` is always greater than or equal to zero, and less than `book_titles.length`" (and thus is a valid index into the array).

Let's see an example of a class invariant.

Class invariant example – a stack

```
/** A Stack data type, implemented using an array */
class ArrayStack implements Stack {
  int elements[];
  /* class invariant: topOfStack always points to current top item
   * (or −1 if empty)*/
  int topOfStack;
  public push(int value) {
     topOfStack += 1;
     if (topOfStack >= 20) { /* throw exception */ }
     elements[topOfStack] = value;
  }
  public int pop() {
     if (topOfStack < 0) { /* throw exception */ }
     return elements[topOfStack];
  }
  // ... other methods ...
}
```

topOfStack is never decremented!

System invariant example – databases

Suppose our system has a database, with records representing *students*, *units*, and *enrolments*.

An enrolment is a `(studentId, unitId)` pair, e.g.: `(23456789, CITS5501)`.

It's a (plainly sensible) rule of our system that an enrolment record must not contain a student ID for a student that doesn't exist, nor a unit ID for a unit that doesn't exist.

If this rule is breached (perhaps a unit gets removed, and the corresponding enrolments don't), then our system is now in an inconsistent (or erroneous) state.

Invariants

We will look at invariants more in the lab/workshops.

Reliability

- The *reliability* of a system is the degree to which its observed behaviour conforms to its specification.

Testing

Definition

We define testing as a systematic attempt to find faults in a software system in a planned way.

(Adapted from Bruegge & Dutoit)

What sorts of things can we test?

We can classify tests by the "level" of component or system they work with:

- We can test some code which is a *small part* of a *larger system* – for instance: a single procedure, method or function. (How small can it be?) This is called *unit testing*.
- We can test how units, classes, modules or other components of a system work together – this is called *integration testing*
- We can test an entire system – this is called *system testing* (or "system-level testing").

  We can test whether the system meets its specifications (system testing proper), and whether it properly executes some scenario in an appropriate environment (end-to-end testing).

What sorts of things can we test?

We can also classify them by the purpose of the test, or when in the software development lifecycle the testing activity occurs:

- After making a change to some component (an enhancement, or bug fix, or re-factoring): we can check whether it still passes all relevant tests. This is called *regression testing*.
- On delivery of a system: we can 'test' whether a system meets a customer's expectations – this is called *acceptance testing*

Testing

- Testing requires a different mind-set from construction: when constructing (or designing) software, we usually focus on what it will do when things go *right*;
  when testing, we focus on *finding faults* – occasions when things do wrong.
- Programmers do often refer to tests as "failing" – but when a test indicates a bug, that's actually example of it *succeeding* in its purpose (i.e., showing the presence of a fault)

Unit testing and unit specifications

- We'll start by looking at the "lowest" level of tests, unit tests.
- When we test a unit of code, we aim to (by finding faults and removing them) increase our confidence that it meets its specifications.
  If we don't *have* any specifications for it, that obviously makes life difficult.
- So in general, we aim to *document* the intended behaviour of any externally accessible unit.

  (Some units might be purely internal – "protected" or "private", for instance, in Java – it is usually good practice to document those as well.)

Requirements

Concept revision

Recommended prior study for this unit is CITS4401 Software Requirements and Design, from which you should already be familiar with different ways of expressing clients' expectations of a system, and the distinction between *functional* and *non-functional* requirements.

By way of revision, we look a little at these concepts now.

For more information, it's recommended you review the Pressman text.

Requirements

For our purposes, we will assume that requirements are statements in natural language which describe the services a system (or some part of it) provides, and any constraints it should satisfy.

If we don't have requirements, then ultimately it is not possible to write tests, because tests are intended to tell us whether the subject under test satisfies required behaviour.

Functional vs non-functional requirements

Functional requirements:

- Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
- May state what the system should *not* do

Non-functional requirements:

- *Constraints* on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
- Often apply to the system as a whole rather than individual features or services.

Functional requirement example

Example

The system shall allow users to search for books by an author's surname.
Details:

- Input: The user selects the "Author Query" option and enters an author's surname (e.g., "Lawrence").
- Output: The system displays a list of all books by authors with that surname, including the book title and publication year. If no books are found for the specified surname, the system shall display the message: "No books found for this author."

This requirement describes particular functionality of a library catalog system.

Stating functional requirements

- Functional requirements are natural language sentences; they typically will start with "The system shall . . . " or "The system must . . . ".
- They will typically contain verbs like "allow," "enable," "provide," "calculate," "store," "retrieve," "validate," "display", "generate", etc. which clearly describe what the system must do.
- Usually, they will mention clear inputs, outputs, and conditions under which particular behaviour will occur.

Properties of good functional requirements

Software engineering texts will provide you with details of the properties a good requirement should have.

Some of the most important properties for our purposes are that a requirement should be

- **Verifiable** - it should be possible to objectively measure whether the requirement has been satisfied or not
- **Unambiguous** - someone reading the requirement should be able to interpret it unambiguously – it should not be capable of multiple interpretations
- **Consistent** - the requirement should not conflict with other requirements
- **Complete** - the requirement should not leave "gaps", where it is unclear what will happen in particular circumstances

Documenting code

Documenting units

- Most modern languages provide some way of documenting the specification of units *inline* (that is, in the body of the code, rather than in a separate reference manual) and extracting that documentation for use by developers.
- For instance:
  - Java provides the Javadoc tool
  - Python provides the Pydoc tool
- For languages which do not have such a tool, applications such as Doxygen allow units to be documented and the documentation extracted.

Javadoc example

Consider the task of finding the position of the first occurrence of some character in a string.

In Java, the `String.indexOf()` method will do this for us.

Its signature is:

```
int indexOf(int ch)
```

That is, it takes an `int` and returns an `int`. (Why not a `char`? For historical reasons we won't get into.)

Javadoc example, cont'd

If we look up the Java documentation for the `indexOf` method, we will get the following information:

> *Returns the index within this string of the first occurrence of the specified character. If a character with value ch occurs in the character sequence represented by this String object, then the index (in Unicode code units) of the first such occurrence is returned. For values of ch in the range from 0 to 0xFFFF (inclusive), this is the smallest value k such that:*
> `this.charAt(k) == ch`
> *is true. For other values of ch, it is the smallest value k such that:* `this.codePointAt(k) == ch` *is true. In either case, if no such character occurs in this string, then -1 is returned.*
>
> ***Parameters:***
> *ch - a character (Unicode code point).*
>
> ***Returns:***
> *the index of the first occurrence of the character in the character sequence represented by this object, or -1 if the character does not occur.*

Javadoc example, cont'd

Key points from the Javadoc documentation:

When we call `someString.indexOf(ch)`:

- If `ch` is *not* in `someString`, the method returns -1
- If `ch` *is* in `someString`, the method returns "the smallest value k such that $>$ `this.codePointAt(k) == ch`"

  (Or: "The first position in `someString` where the character `ch` appears." Which of the two is easier to understand? Which is easier to write a test for?)

Javadoc example, cont'd

Key points from the Javadoc documentation:

When we call `someString.indexOf(ch)`:

- If `ch` is *not* in `someString`, the method returns -1
- If `ch` *is* in `someString`, the method returns "the smallest value k such that $>$ `this.codePointAt(k) == ch`"

  (Or: "The first position in `someString` where the character `ch` appears." Which of the two is easier to understand? Which is easier to write a test for?)

(Are there any other possibilities not covered by the documentation?)

Javadoc example, cont'd

The documentation for the `indexOf` method is produced from a specially written *comment* which looks something like this:

```
/** Returns the index within this string of the first occurrence of the
 * specified character. If a character with value ch occurs in the character
 * sequence represented by this String object, then the index (in Unicode code
 * units) of the first such occurrence is returned. For values of ch in the range
 * from 0 to 0xFFFF (inclusive), this is the smallest value <i>k</i> such that:
 * this.charAt(<i>k</i>) == ch
 * is true. For other values of ch, it is the smallest value <i>k</i> such that:
 * this.codePointAt(<i>k</i>) == ch
 * is true. In either case, if no such character occurs in this
 * string, then -1 is returned.
 *
 * @param   ch   a character (Unicode code point).
 * @return  the index of the first occurrence of the character in the
 *          character sequence represented by this object, or
 *          -1 if the character does not occur.
 */
```

Javadoc conventions

The Javadoc comment is normally placed just before the method it documents, and begins with a double asterisk ("`/**`")

- It describes what the method does, what parameters should be passed in, and what result will be returned.
- It uses the `@param` markup to describe each parameter.
- It uses the `@return` markup to describe the return value.

APIs

- The specification for all the externally accessible classes, methods and so on in a module make up what is called the *API*[3] of the module – the "Application Programming Interface".

[3] See further "Who invented the API?", https://nordicapis.com/who-invented-the-api/

APIs

- The specification for all the externally accessible classes, methods and so on in a module make up what is called the *API*[3] of the module – the "Application Programming Interface".
- The name derives from the idea that if we write a re-usable component of some sort (like a library), then other developers will want to use this in their application programming, and we should document the public *interface* to that component.

[3] See further "Who invented the API?", https://nordicapis.com/who-invented-the-api/

APIs

- The specification for all the externally accessible classes, methods and so on in a module make up what is called the *API*[3] of the module – the "Application Programming Interface".
- The name derives from the idea that if we write a re-usable component of some sort (like a library), then other developers will want to use this in their application programming, and we should document the public *interface* to that component.
- (Actually, the other developers might not be writing an *application* per se – they might be writing another library – but the name has stuck.)

[3] See further "Who invented the API?", https://nordicapis.com/who-invented-the-api/

APIs as contracts

- We can think of the API for a function (or other procedural unit) as constituting a *contract* between the developer of the function, and the client code using it.[4]
- In effect, the documentation says "If you, the client code, pass me arguments which meet the following criteria, I promise to do the following thing: . . . "

[4] See further "Design by contract", https://en.wikipedia.org/wiki/Design_by_contract; Meyer, Bertrand. "Applying 'design by contract'." Computer 25.10 (1992): 40-51.

APIs, cont'd

- The "following thing" – the behaviour of the function – will usually be to return some sort of value, or to cause some sort of "side effect".
- (A *side effect* is anything the function does to alter the current or subsequent behaviour of the system or its interaction with external systems, other than returning a value.
  For example, writing a file to disk, or sending an email, or changing the value of a global variable.)

API example

- If you have used Java, you most likely at some point will have written something like `System.out.println("some string ...")`

API example

- If you have used Java, you most likely at some point will have written something like `System.out.println("some string ...")`
- What does the `println` method promise to do?

API example

- If you have used Java, you most likely at some point will have written something like `System.out.println("some string ...")`
- What does the `println` method promise to do?
- The Javadoc says:

API example

- If you have used Java, you most likely at some point will have written something like `System.out.println("some string ...")`
- What does the `println` method promise to do?
- The Javadoc says:

  `void println(String x)`
  Prints a String and then terminate the line.

API example

- If you have used Java, you most likely at some point will have written something like `System.out.println("some string ...")`
- What does the `println` method promise to do?
- The Javadoc says:

  `void println(String x)`
  Prints a String and then terminate the line.

- Does `println()` return a value? If not, then what does it promise to do?

APIs – specification vs implementation

The API documentation does not normally say *how* the function is to be implemented – just what its return value and effects are.

This means that if the library developer decides to reimplement the function in another way (for instance, to improve efficiency), they can, without changing the API.

(Functions are a form of abstraction. They allow a developer to call the function, without knowing how it is implemented.)

APIs – specification vs implementation

- This means that, as a user of an API, you can think of the API as a "black box" –
  - You don't know (nor do you need to know) *how* it works – just that it will do the job its documentation describes

Specification vs implementation in Java

- Q. In Java, does the API tell us *how* `String.indexOf(ch)` is implemented? How would you implement it?

Specification vs implementation in Java

- Q. In Java, does the API tell us *how* `String.indexOf(ch)` is implemented? How would you implement it?
- A. It does not. The *plausible* way to do it is to test each possible index from 0 to (length-of-string - 1), see if matches the character we're looking for, and if it does, return the index we're currently at.

Specification vs implementation in Java

- Q. In Java, does the API tell us *how* `String.indexOf(ch)` is implemented? How would you implement it?
- A. It does not. The *plausible* way to do it is to test each possible index from 0 to (length-of-string - 1), see if matches the character we're looking for, and if it does, return the index we're currently at.
- But there's nothing in the *specification* to stop us from implementing it in other ways . . .

Specification vs implementation in Java

- Q. In Java, does the API tell us *how* `String.indexOf(ch)` is implemented? How would you implement it?
- A. It does not. The *plausible* way to do it is to test each possible index from 0 to (length-of-string - 1), see if matches the character we're looking for, and if it does, return the index we're currently at.
- But there's nothing in the *specification* to stop us from implementing it in other ways ...
- e.g. Generate a random number from 0 to (length-of-string - 1), call it k. Check and see if we've hit all positions from 0 to $k-1$ yet; if we have, and `inputString.charAt(k)` equals the character we're after, return the current index.[5]

[5] See also Bogosort, https://en.wikipedia.org/wiki/Bogosort

Specification vs implementation – "illities"

Sometimes specifications for units will describe not just what the unit *returns* or *does*, but *how it does it*.

For instance, if we implement `String.indexOf(ch)` in the "generate a random number" way we described, it would be *extremely* slow.

Specification vs implementation – "illities"

The specification of `String.indexOf(ch)` could rule out "silly" implementations like this, by saying something like "the `indexOf` method shall provide guaranteed $O(n)$ time cost, where n is the length of the string".

(If you have not done a data structures and algorithms course, don't worry too much about what "$O(n)$" means – it roughly means that the time to execute `indexOf` will increase in proportion to the length of the string.)

Specification vs implementation – "illities"

If you look at the documentation for Java's `TreeMap` class, in fact, you will see that the implementation provided by Oracle promises to provide guarantees about how long particular methods will take to run:

> *This implementation provides guaranteed log(n) time cost for the* `containsKey`, `get`, `put` *and* `remove` *operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's Introduction to Algorithms.*

APIs, summary

So:

- The API describes the *expected behaviour* of a module (or larger system).
- The code constitutes a particular *implementation* of that API.

APIs, cont'd

What should go in the API documentation?

- The *preconditions* – any conditions which should be satisfied by the parameters or the system state when the function is called.
- The *postconditions* – the return value of the function, and any *changes* the function makes to the system state (the "side effects" discussed earlier)

Sometimes the postconditions will vary, depending on the arguments passed:

- "*IF* a valid email address is supplied, *THEN* the `emailMyResignationLetter()` method will email a resignation letter. But if not, then a `MalformedEmailAddress` exception will be thrown."

We need to make sure we cover all circumstances, so that users of an API will know what to expect.

APIs, cont'd

If the preconditions are *not* satisfied, then the behaviour is **undefined**. This means the user has failed to live up to their part of the "bargain", and has NO guarantees about what the system might do.

By way of example, we'll consider Java's `binarySearch` method in `java.util.Arrays`:

- https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#binarySearch(byte%5B%5D,%20byte)

APIs, cont'd

Once we know the preconditions and postconditions for a function, we can write tests for it.

(They needn't be spelled out formally or mathematically – but it is best if they are clear, consistent and unambiguous.)

References

- Bruegge and Dutoit, *Object-Oriented Software Engineering Using UML, Patterns, and Java* (Pearson, 2010)
- Martin Fowler, "Mocks Aren't Stubs" (https://martinfowler.com/articles/mocksArentStubs.html)
- Gerard Meszaros, *xUnit Test Patterns: Refactoring Test Code* (Addison-Wesley Professional, 2007)
- Claessen and Hughes, "QuickCheck: a lightweight tool for random testing of Haskell programs." *ACM Sigplan Notices* 46.4 (2011): 53-64.
- Kristopher Sandoval, "Who Invented the API?", Sept 20 2018 (https://nordicapis.com/who-invented-the-api/).