Week:
8 of 12

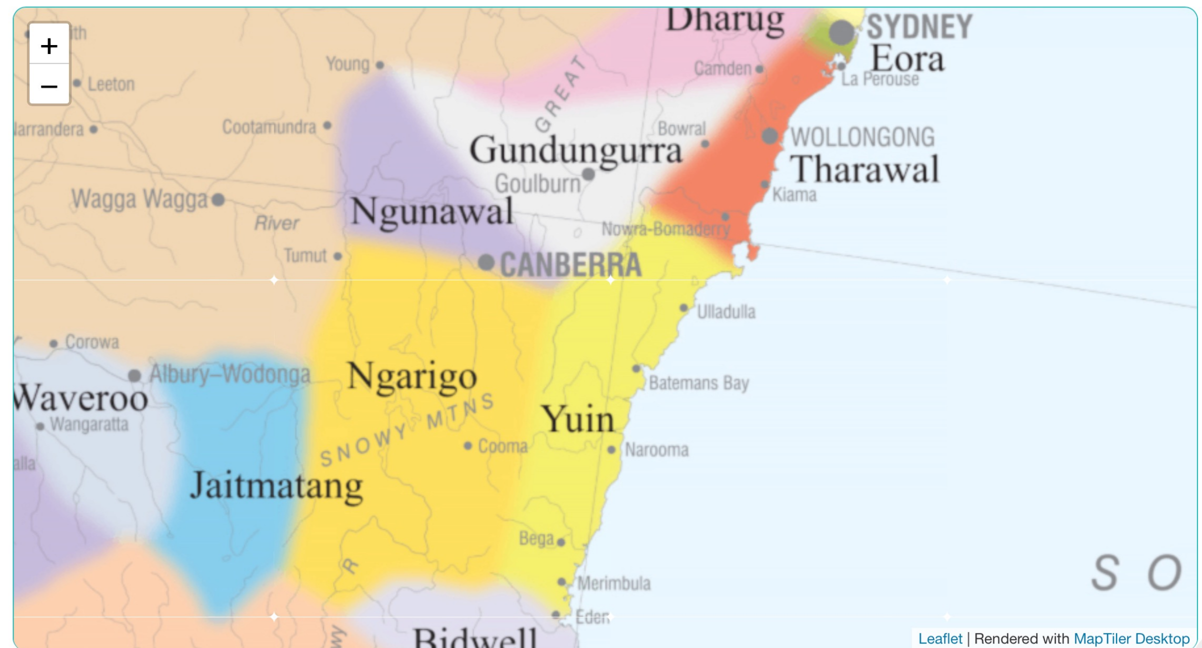
COMP 2120 / COMP 6120

TESTING

A/Prof Alex Potanin and Dr Melina Vidoni

ANU Acknowledgment of Country

"We acknowledge and celebrate the First Australians on whose traditional lands we meet, and pay our respect to the elders past and present."



https://aiatsis.gov.au/explore/map-indigenous-australia

CRICOS PROVIDER #00120C

The Story of Google Web Server

In Google's early days, engineer-driven testing was often assumed to be of little importance. Teams regularly relied on smart people to get the software right. A few systems ran large integration tests, but mostly it was the Wild West. One product in particular seemed to suffer the worst: it was called the Google Web Server, also known as GWS.

GWS is the web server responsible for serving Google Search queries and is as important to Google Search as air traffic control is to an airport. Back in 2005, as the project swelled in size and complexity, productivity had slowed dramatically. Releases were becoming buggier, and it was taking longer and longer to push them out. Team members had little confidence when making changes to the service, and often found out something was wrong only when features stopped working in production. (At one point, more than 80% of production pushes contained user-affecting bugs that had to be rolled back.)

To address these problems, the tech lead (TL) of GWS decided to institute a policy of engineer-driven, automated testing. As part of this policy, all new code changes were required to include tests, and those tests would be run continuously. Within a year of instituting this policy, the number of emergency pushes *dropped by half*. This drop occurred despite the fact that the project was seeing a record number of new changes every quarter. Even in the face of unprecedented growth and change, testing brought renewed productivity and confidence to one of the most critical projects at Google. Today, GWS has tens of thousands of tests, and releases almost every day with relatively few customer-visible failures.

The changes in GWS marked a watershed for testing culture at Google as teams in other parts of the company saw the benefits of testing and moved to adopt similar tactics.

CRICOS PROVIDER #00120C

Software testing

- Software testing is a process in which you execute your program using data that simulates user inputs.
- You observe its behaviour to see whether or not your program is doing what it is supposed to do.
  - Tests pass if the behaviour is what you expect. Tests fail if the behaviour differs from that expected.
  - If your program does what you expect, this shows that for the inputs used, the program behaves correctly.
- If these inputs are representative of a larger set of inputs, you can infer that the program will behave correctly for all members of this larger input set.

CRICOS PROVIDER #00120C

Program bugs

- If the behaviour of the program does not match the behaviour that you expect, then this means that there are bugs in your program that need to be fixed.
- There are two causes of program bugs:
  - ***Programming errors*** You have accidentally included faults in your program code. For example, a common programming error is an 'off-by-1' error where you make a mistake with the upper bound of a sequence and fail to process the last element in that sequence.
  - ***Understanding errors*** You have misunderstood or have been unaware of some of the details of what the program is supposed to do. For example, if your program processes data from a file, you may not be aware that some of this data is in the wrong format, so your program doesn't include code to handle this.

CRICOS PROVIDER #00120C

Types of testing

- ***Functional testing***
  Test the functionality of the overall system. The goals of functional testing are to discover as many bugs as possible in the implementation of the system and to provide convincing evidence that the system is fit for its intended purpose.

- ***User testing***
  Test that the software product is useful to and usable by end-users. You need to show that the features of the system help users do what they want to do with the software. You should also show that users understand how to access the software's features and can use these features effectively.

- ***Performance and load testing***
  Test that the software works quickly and can handle the expected load placed on the system by its users. You need to show that the response and processing time of your system is acceptable to end-users. You also need to demonstrate that your system can handle different loads and scales gracefully as the load on the software increases.

- ***Security testing***
  Test that the software maintains its integrity and can protect user information from theft and damage.

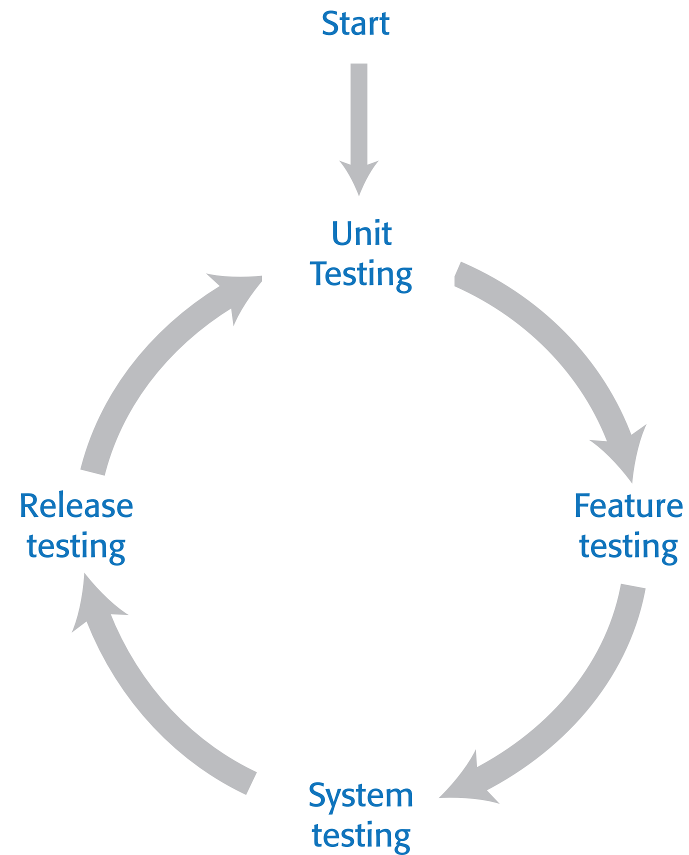
CRICOS PROVIDER #00120C

Functional testing

- Functional testing involves developing a large set of program tests so that, ideally, all of a program's code is executed at least once.
- The number of tests needed obviously depends on the size and the functionality of the application.
- For a business-focused web application, you may have to develop thousands of tests to convince yourself that your product is ready for release to customers.
- Functional testing is a staged activity in which you initially test individual units of code. You integrate code units with other units to create larger units then do more testing.
- The process continues until you have created a complete system ready for release.

CRICOS PROVIDER #00120C

Functional testing



CRICOS PROVIDER #00120C

Functional testing processes

- **Unit testing**
  The aim of unit testing is to test program units in isolation. Tests should be designed to execute all of the code in a unit at least once. Individual code units are tested by the programmer as they are developed.
- **Feature testing**
  Code units are integrated to create features. Feature tests should test all aspects of a feature. All of the programmers who contribute code units to a feature should be involved in its testing.
- **System testing**
  Code units are integrated to create a working (perhaps incomplete) version of a system. The aim of system testing is to check that there are no unexpected interactions between the features in the system. System testing may also involve checking the responsiveness, reliability and security of the system. In large companies, a dedicated testing team may be responsible for system testing. In small companies, this is impractical, so product developers are also involved in system testing.
- **Release testing**
  The system is packaged for release to customers and the release is tested to check that it operates as expected. The software may be released as a cloud service or as a download to be installed on a customer's computer or mobile device. If DevOps is used, then the development team are responsible for release testing otherwise a separate team has that responsibility.

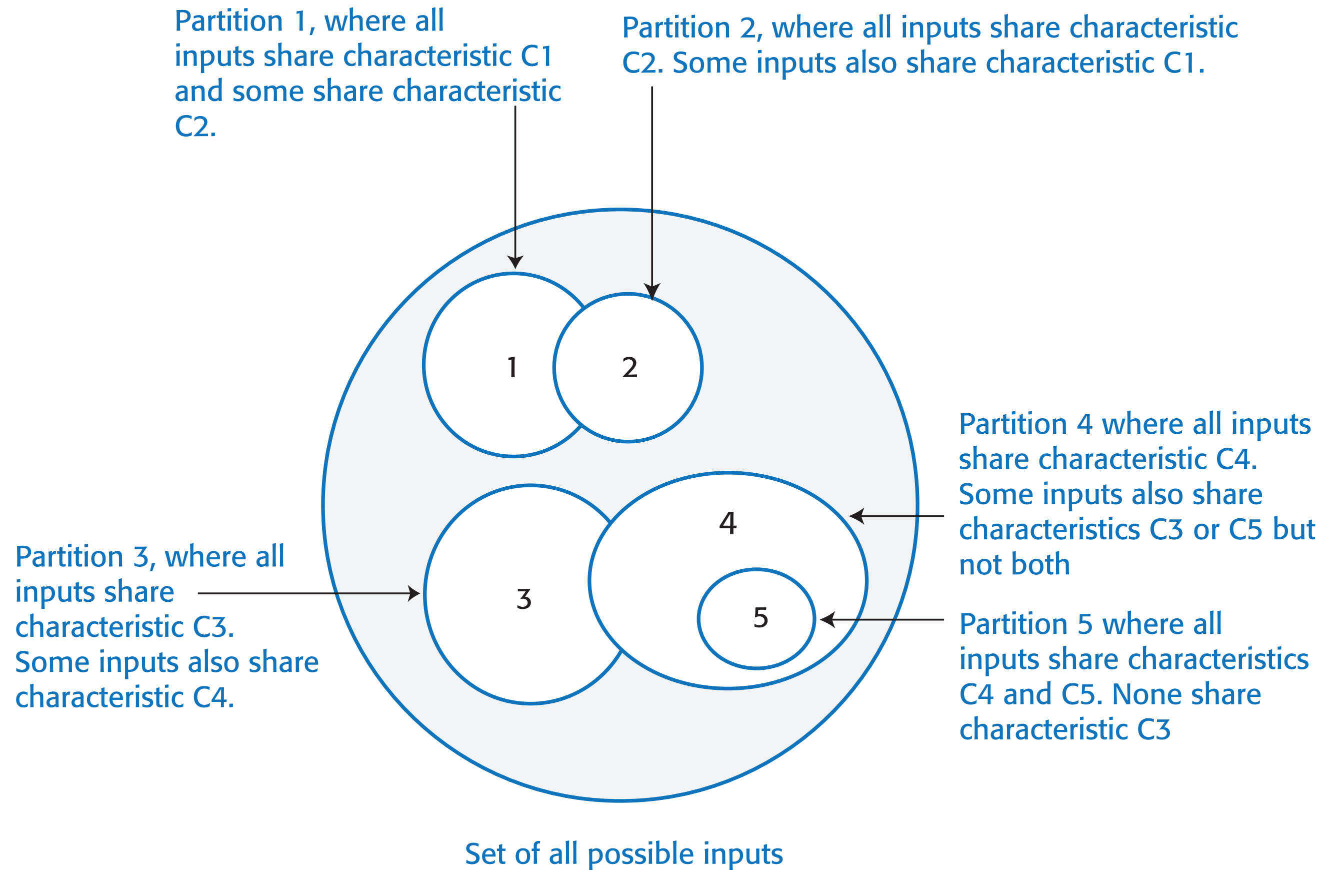
CRICOS PROVIDER #00120C

Unit testing

- As you develop a code unit, you should also develop tests for that code.
- A code unit is anything that has a clearly defined responsibility. It is usually a function or class method but could be a module that includes a small number of other functions.
- Unit testing is based on a simple general principle:
  - If a program unit behaves as expected for a set of inputs that have some shared characteristics, it will behave in the same way for a larger set whose members share these characteristics.
- To test a program efficiently, you should identify sets of inputs (equivalence partitions) that will be treated in the same way in your code.
- The equivalence partitions that you identify should not just include those containing inputs that produce the correct values. You should also identify 'incorrectness partitions' where the inputs are deliberately incorrect.

CRICOS PROVIDER #00120C

Equivalence partitions



CRICOS PROVIDER #00120C

A name checking function

```
def namecheck (s):

    # Checks that a name only includes alphabetic characters, - or
    # a single quote. Names must be between 2 and 40 characters long
    # quoted strings and -- are disallowed

    namex = r"^[a-zA-Z][a-zA-Z-'][{1,39}$"
    if re.match (namex, s):
        if re.search ("'.*'", s) or re.search ("--", s):
            return False
        else:
            return True
    else:
        return False
```

CRICOS PROVIDER #00120C

Equivalence partitions for the name checking function

- ***Correct names 1***
  The inputs only includes alphabetic characters and are between 2 and 40 characters long.
- ***Correct names 2***
  The inputs only includes alphabetic characters, hyphens or apostrophes and are between 2 and 40 characters long.
- ***Incorrect names 1***
  The inputs are between 2 and 40 characters long but include disallowed characters.
- ***Incorrect names 2***
  The inputs include allowed characters but are either a single character or are more than 40 characters long.
- ***Incorrect names 3***
  The inputs are between 2 and 40 characters long but the first character is a hyphen or an apostrophe.
- ***Incorrect names 4***
  The inputs include valid characters, are between 2 and 40 characters long, but include either a double hyphen, quoted text or both.

CRICOS PROVIDER #00120C

Unit testing guidelines (1)

- ***Test edge cases***
  If your partition has upper and lower bounds (e.g. length of strings, numbers, etc.) choose inputs at the edges of the range.
- ***Force errors***
  Choose test inputs that force the system to generate all error messages. Choose test inputs that should generate invalid outputs.
- ***Fill buffers***
  Choose test inputs that cause all input buffers to overflow.
- ***Repeat yourself***
  Repeat the same test input or series of inputs several times.

CRICOS PROVIDER #00120C

Unit testing guidelines (2)

- ***Overflow and underflow***
  If your program does numeric calculations, choose test inputs that cause it to calculate very large or very small numbers.
- ***Don't forget null and zero***
  If your program uses pointers or strings, always test with null pointers and strings. If you use sequences, test with an empty sequence. For numeric inputs, always test with zero.
- ***Keep count***
  When dealing with lists and list transformation, keep count of the number of elements in each list and check that these are consistent after each transformation.
- ***One is different***
  If your program deals with sequences, always test with sequences that have a single value.

CRICOS PROVIDER #00120C

But, remember…

The Importance of Maintainability

Imagine this scenario: Mary wants to add a simple new feature to the product and is able to implement it quickly, perhaps requiring only a couple dozen lines of code. But when she goes to check in her change, she gets a screen full of errors back from the automated testing system. She spends the rest of the day going through those failures one by one. In each case, the change introduced no actual bug, but broke some of the assumptions that the test made about the internal structure of the code, requiring those tests to be updated. Often, she has difficulty figuring out what the tests were trying to do in the first place, and the hacks she adds to fix them make those tests even more difficult to understand in the future. Ultimately, what should have been a quick job ends up taking hours or even days of busywork, killing Mary's productivity and sapping her morale.

Here, testing had the opposite of its intended effect by draining productivity rather than improving it while not meaningfully increasing the quality of the code under test. This scenario is far too common, and Google engineers struggle with it every day. There's no magic bullet, but many engineers at Google have been working to develop sets of patterns and practices to alleviate these problems, which we encourage the rest of the company to follow.

CRICOS PROVIDER #00120C

Feature testing

- Features have to be tested to show that the functionality is implemented as expected and that the functionality meets the real needs of users.
  - For example, if your product has a feature that allows users to login using their Google account, then you have to check that this registers the user correctly and informs them of what information will be shared with Google.
  - You may want to check that it gives users the option to sign up for email information about your product.
- Normally, a feature that does several things is implemented by multiple, interacting, program units.
- These units may be implemented by different developers and all of these developers should be involved in the feature testing process.

CRICOS PROVIDER #00120C

Types of feature test

- Interaction tests
  - These test the interactions between the units that implement the feature. The developers of the units that are combined to make up the feature may have different understandings of what is required of that feature.
  - These misunderstandings will not show up in unit tests but may only come to light when the units are integrated.
  - The integration may also reveal bugs in program units, which were not exposed by unit testing.
- Usefulness tests
  - These test that the feature implements what users are likely to want.
  - For example, the developers of a login with Google feature may have implemented an opt-out default on registration so that users receive all emails from a company. They must expressly choose what type of emails that they don’t want.
  - What might be preferred is an opt-in default so that users choose what types of email they do want to receive.

CRICOS PROVIDER #00120C

User stories for the sign-in with Google feature

- ***User registration***
  As a user, I want to be able to login without creating a new account so that I don't have to remember another login id and password.
- ***Information sharing***
  As a user, I want to know what information you will share with other companies. I want to be able to cancel my registration if I don't want to share this information.
- ***Email choice***
  As a user, I want to be able to choose the types of email that I'll get from you when I register for an account.

CRICOS PROVIDER #00120C

Feature tests for sign-in with Google

- ***Initial login screen***
  Test that the screen displaying a request for Google account credentials is correctly displayed when a user clicks on the 'Sign-in with Google' link. Test that the login is completed if the user is already logged in to Google.
- ***Incorrect credentials***
  Test that the error message and retry screen is displayed if the user inputs incorrect Google credentials.
- ***Shared information***
  Test that the information shared with Google is displayed, along with a cancel or confirm option. Test that the registration is cancelled if the cancel option is chosen.
- ***Email opt-in***
  Test that the user is offered a menu of options for email information and can choose multiple items to opt-in to emails. Test that the user is not registered for any emails if no options are selected.

CRICOS PROVIDER #00120C

System and release testing

- System testing involves testing the system as a whole, rather than the individual system features.
- System testing should focus on four things:
  - Testing to discover if there are unexpected and unwanted interactions between the features in a system.
  - Testing to discover if the system features work together effectively to support what users really want to do with the system.
  - Testing the system to make sure it operates in the expected way in the different environments where it will be used.
  - Testing the responsiveness, throughput, security and other quality attributes of the system.

CRICOS PROVIDER #00120C

Scenario-based testing

- The best way to systematically test a system is to start with a set of scenarios that describe possible uses of the system and then work through these scenarios each time a new version of the system is created.
- Using the scenario, you identify a set of end-to-end pathways that users might follow when using the system.
- An end-to-end pathway is a sequence of actions from starting to use the system for the task, through to completion of the task.

CRICOS PROVIDER #00120C

Choosing a holiday destination

- Andrew and Maria have a two year old son and a four month old daughter. They live in Scotland and they want to have a holiday in the sunshine. However, they are concerned about the hassle of flying with young children. They decide to try a family holiday planner product to help them choose a destination that is easy to get to and that fits in with their childrens' routines.
- Maria navigates to the holiday planner website and selects the 'find a destination' page. This presents a screen with a number of options. She can choose a specific destination or can choose a departure airport and find all destinations that have direct flights from that airport. She can also input the time band that she'd prefer for flights, holiday dates and a maximum cost per person.
- Edinburgh is their closest departure airport. She chooses 'find direct flights'. The system then presents a list of countries that have direct flights from Edinburgh and the days when these flights operate. She selects France, Italy, Portugal and Spain and requests further information about these flights. She then sets a filter to display flights that leave on a Saturday or Sunday after 7.30am and arrive before 6pm.
- She also sets the maximum acceptable cost for a flight. The list of flights is pruned according to the filter and is redisplayed. Maria then clicks on the flight she wants. This opens a tab in her browser showing a booking form for this flight on the airline's website.

CRICOS PROVIDER #00120C

End-to-end pathways

1. User inputs departure airport and chooses to see only direct flights. User quits.
2. User inputs departure airport and chooses to see all flights. User quits.
3. User chooses destination country and chooses to see all flights. User quits.
4. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User quits.
5. User inputs departure airport and chooses to see direct flights. User sets filter specifying departure times and prices. User selects a displayed flight and clicks through to airline website. User returns to holiday planner after booking flight.

CRICOS PROVIDER #00120C

Release testing

- Release testing is a type of system testing where a system that's intended for release to customers is tested.

- The fundamental differences between release testing and system testing are:

  - Release testing tests the system in its real operational environment rather than in a test environment. Problems commonly arise with real user data, which is sometimes more complex and less reliable than test data.

  - The aim of release testing is to decide if the system is good enough to release, not to detect bugs in the system. Therefore, some tests that 'fail' may be ignored if these have minimal consequences for most users.

- Preparing a system for release involves packaging that system for deployment (e.g. in a container if it is a cloud service) and installing software and libraries that are used by your product. You must define configuration parameters such as the name of a root directory, the database size limit per user and so on.

CRICOS PROVIDER #00120C

Test automation

- Automated testing is based on the idea that tests should be executable.
- An executable test includes the input data to the unit that is being tested, the expected result and a check that the unit returns the expected result.
- You run the test and the test passes if the unit returns the expected result.
- Normally, you should develop hundreds or thousands of executable tests for a software product.

CRICOS PROVIDER #00120C

Test methods for an interest calculator

```
# TestInterestCalculator inherits attributes and
methods from the class

# TestCase in the testing framework unittest


class TestInterestCalculator (unittest.TestCase):

    # Define a set of unit tests where each test tests
one thing only

    # Tests should start with test_ and the name should
explain what is being tested

    def test_zeroprincipal (self):

        #Arrange - set up the test parameters

        p = 0; r = 3; n = 31

        result_should_be = 0

        #Action - Call the method to be tested

        interest = interest_calculator (p, r, n)

        #Assert - test what should be true

        self.assertEqual (result_should_be, interest)
```

```
def test_yearly_interest (self):

        #Arrange - set up the test parameters

        p = 17000; r = 3; n = 365

        #Action - Call the method to be tested

        result_should_be = 270.36

        interest = interest_calculator (p, r, n)

        #Assert - test what should be true

        self.assertEqual (result_should_be, interest)
```

CRICOS PROVIDER #00120C

Automated tests

- It is good practice to structure automated tests into three parts:
  - **Arrange** You set up the system to run the test. This involves defining the test parameters and, if necessary, mock objects that emulate the functionality of code that has not yet been developed.
  - **Action** You call the unit that is being tested with the test parameters.
  - **Assert** You make an assertion about what should hold if the unit being tested has executed successfully. In program on the previous slide, we use assertEquals, which checks if its parameters are equal.
- If you use equivalence partitions to identify test inputs, you should have several automated tests based on correct and incorrect inputs from each partition.

CRICOS PROVIDER #00120C

Executable tests for the namecheck function

```
import unittest

from RE_checker import namecheck


class TestNameCheck (unittest.TestCase):


    def test_alphaname (self):

        self.assertTrue (namecheck ('Sommerville'))


    def test_doublequote (self):

        self.assertFalse (namecheck ("Thisis'maliciouscode'"))


    def test_namestartswithhyphen (self):

        self.assertFalse (namecheck ('-Sommerville'))


    def test_namestartswithquote (self):

        self.assertFalse (namecheck ("'Reilly"))

    def test_nametoolong (self):

        self.assertFalse (namecheck 
('Thisisalongstringwithmorethen40charactersfrombeginningtoend'))


    def test_nametooshort (self):

        self.assertFalse (namecheck ('S'))


    def test_namewithdigit (self):

        self.assertFalse (namecheck('C-3PO'))


    def test_namewithdoublehyphen (self):

        self.assertFalse (namecheck ('--badcode'))
```

CRICOS PROVIDER #00120C

Executable tests for the namecheck function

```
def test_namewithhyphen (self):
        self.assertTrue (namecheck ('Washington-Wilson'))

def test_namewithinvalidchar (self):
        self.assertFalse (namecheck('Sommer_ville'))

def test_namewithquote (self):
        self.assertTrue (namecheck ("O'Reilly"))

def test_namewithspaces (self):
        self.assertFalse (namecheck ('Washington Wilson'))

def test_shortname (self):
        self.assertTrue ('Sx')

def test_thiswillfail (self):
        self.assertTrue (namecheck ("O Reilly"))
```

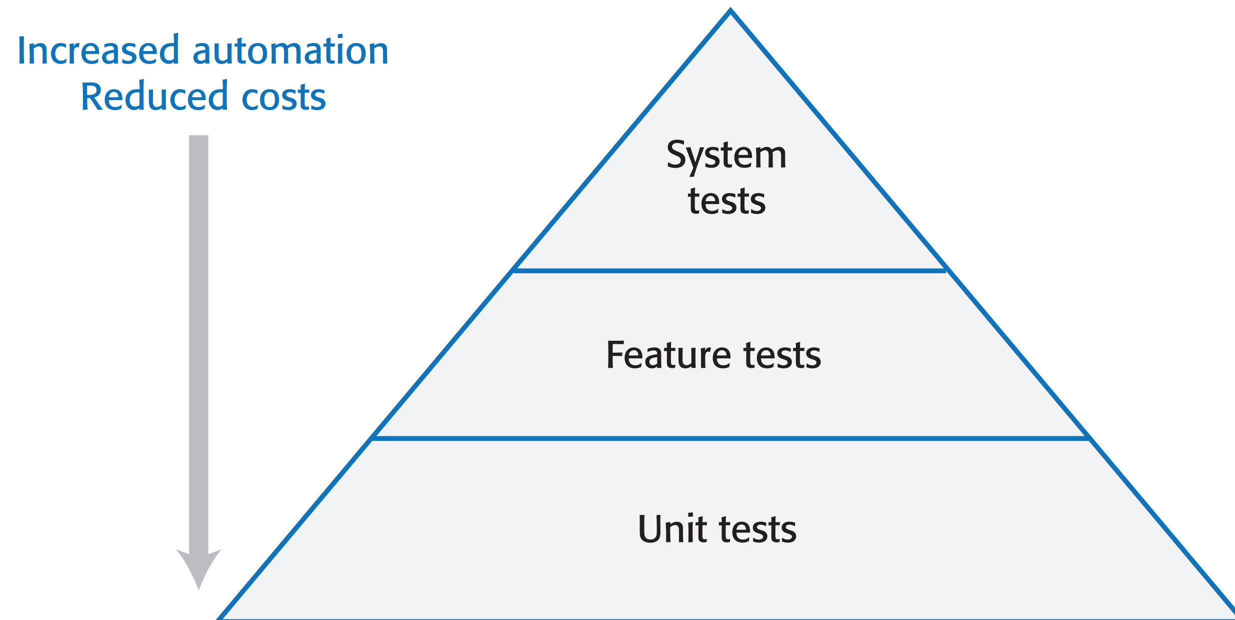
CRICOS PROVIDER #00120C

Code to run unit tests from files

```
import unittest

loader = unittest.TestLoader()

#Find the test files in the current directory

tests = loader.discover('.')

#Specify the level of information provided by the test runner

testRunner = unittest.runner.TextTestRunner(verbosity=2)
testRunner.run(tests)
```

CRICOS PROVIDER #00120C

The test pyramid



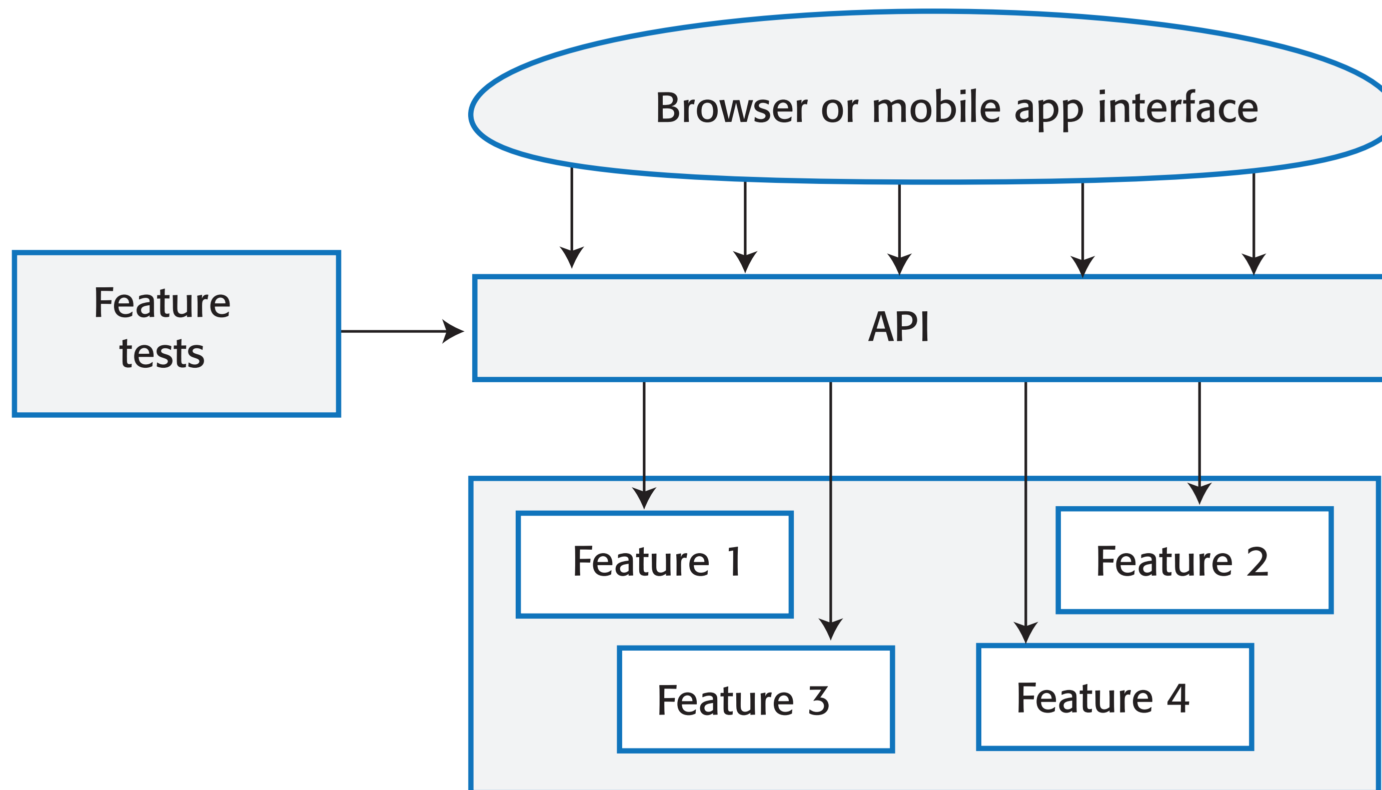
CRICOS PROVIDER #00120C

Automated feature testing

- Generally, users access features through the product's graphical user interface (GUI).
- However, GUI-based testing is expensive to automate so it is best to design your product so that its features can be directly accessed through an API and not just from the user interface.
- The feature tests can then access features directly through the API without the need for direct user interaction through the system's GUI.
- Accessing features through an API has the additional benefit that it is possible to re-implement the GUI without changing the functional components of the software.

CRICOS PROVIDER #00120C

Feature editing through an API



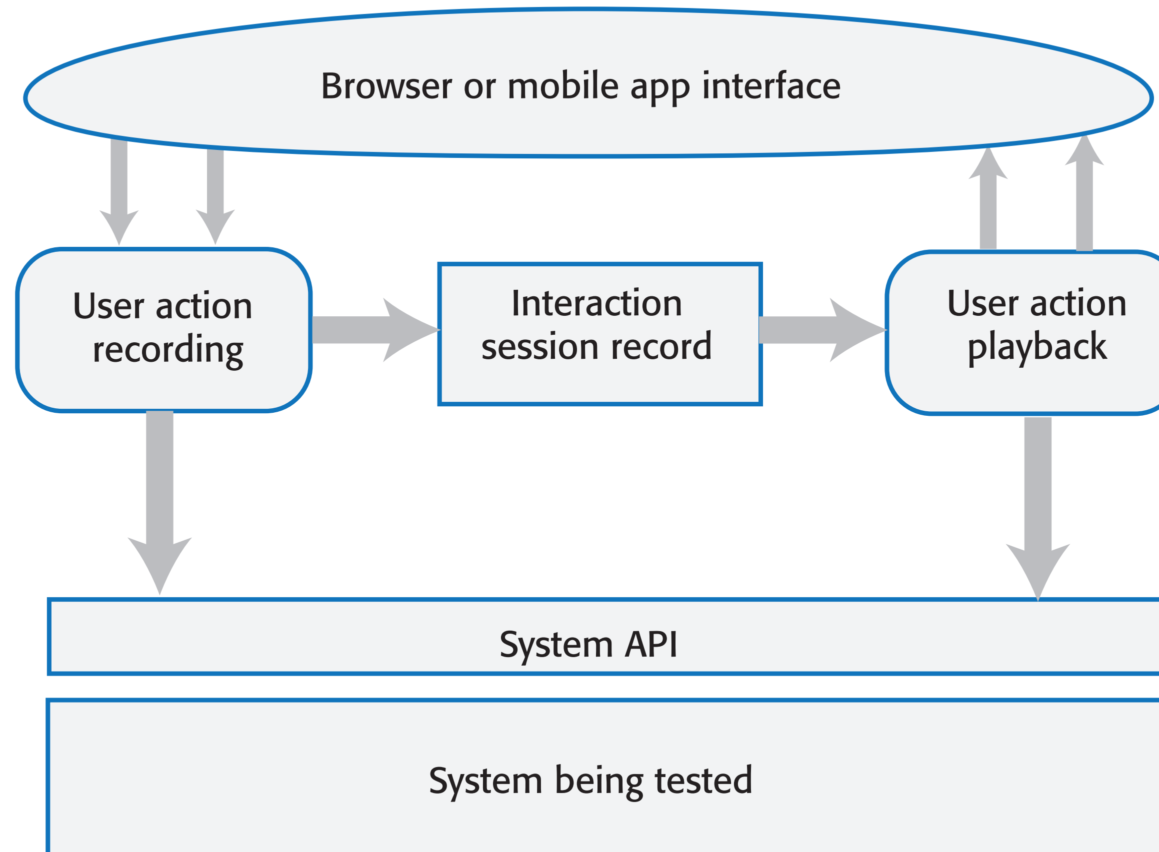
CRICOS PROVIDER #00120C

System testing

- System testing, which should follow feature testing, involves testing the system as a surrogate user.
- As a system tester, you go through a process of selecting items from menus, making screen selections, inputting information from the keyboard and so on.
- You are looking for interactions between features that cause problems, sequences of actions that lead to system crashes and so on.
- Manual system testing, when testers have to repeat sequences of actions, is boring and error-prone. In some cases, the timing of actions is important and is practically impossible to repeat consistently.
  - To avoid these problems, testing tools have been developed that can record a series of actions and automatically replay these when a system is retested

CRICOS PROVIDER #00120C

Interaction recording and playback



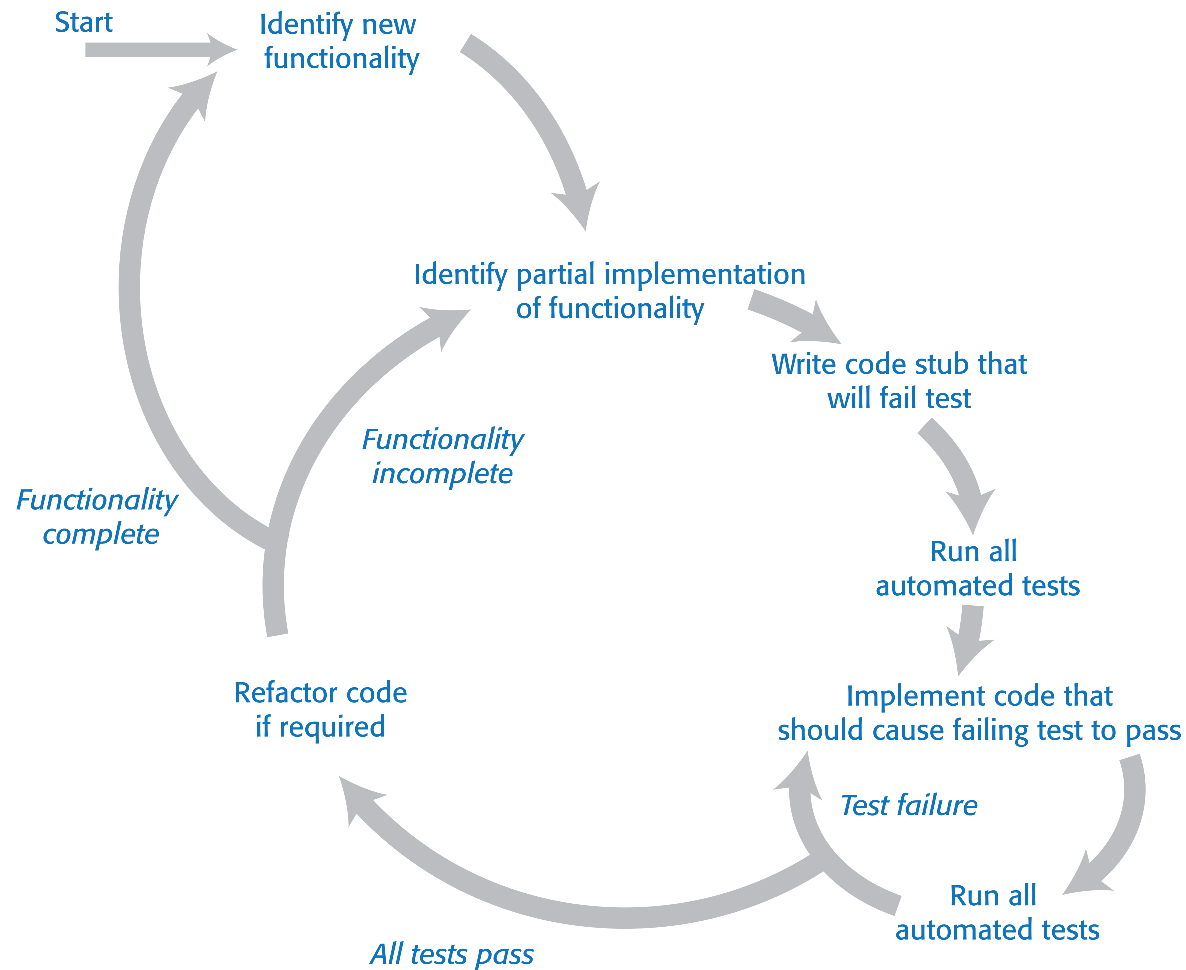
CRICOS PROVIDER #00120C

Test-driven development

- Test-driven development (TDD) is an approach to program development that is based around the general idea that you should write an executable test or tests for code that you are writing before you write the code.
- It was introduced by early users of the Extreme Programming agile method, but it can be used with any incremental development approach.
- Test-driven development works best for the development of individual program units and it is more difficult to apply to system testing.
- Even the strongest advocates of TDD accept that it is challenging to use this approach when you are developing and testing systems with graphical user interfaces.

CRICOS PROVIDER #00120C

Test-driven development



CRICOS PROVIDER #00120C

Stages of test-driven development (1)

- ***Identify partial implementation***
  Break down the implementation of the functionality required into smaller mini-units. Choose one of these mini-units for implementation.
- ***Write mini-unit tests***
  Write one or more automated tests for the mini-unit that you have chosen for implementation. The mini-unit should pass these tests if it is properly implemented.
- **Write a code stub that will fail test**
  Write incomplete code that will be called to implement the mini-unit. You know this will fail.
- ***Run all existing automated tests***
  All previous tests should pass. The test for the incomplete code should fail.

CRICOS PROVIDER #00120C

Stages of test-driven development (2)

- ***Implement code that should cause the failing test to pass***
  Write code to implement the mini-unit, which should cause it to operate correctly
- ***Rerun all automated tests***
  If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.
- ***Refactor code if necessary***
  If all tests pass, you can move on to implementing the next mini-unit. If you see ways of improving your code, you should do this before the next stage of implementation.

CRICOS PROVIDER #00120C

Benefits of test-driven development

- It is a systematic approach to testing in which tests are clearly linked to sections of the program code.
  - This means you can be confident that your tests cover all of the code that has been developed and that there are no untested code sections in the delivered code. In my view, this is the most significant benefit of TDD.
- The tests act as a written specification for the program code. In principle at least, it should be possible to understand what the program does by reading the tests.
- Debugging is simplified because, when a program failure is observed, you can immediately link this to the last increment of code that you added to the system.
- It is argued that TDD leads to simpler code as programmers only write code that's necessary to pass tests. They don't over-engineer their code with complex features that aren't needed.

CRICOS PROVIDER #00120C

Sommerville’s reasons for not using TDD

- ***TDD discourages radical program change***
  I found that I was reluctant to make refactoring decisions that I knew would cause many tests to fail. I tended to avoid radical program change for this reason.
- ***I focused on the tests rather than the problem I was trying to solve***
  A basic principle of TDD is that your design should be driven by the tests you have written. I found that I was unconsciously redefining the problem I was trying to solve to make it easier to write tests. This meant that I sometimes didn’t implement important checks, because it was difficult to write tests in advance of their implementation.
- ***I spent too much time thinking about implementation details rather than the programming problem***
  Sometimes when programming, it is best to step back and look at the program as a whole rather than focusing on implementation details. TDD encourages a focus on details that might cause tests to pass or fail and discourages large-scale program revisions.
- ***It is hard to write ‘bad data’ tests***
  Many problems involving dealing with messy and incomplete data. It is practically impossible to anticipate all of the data problems that might arise and write tests for these in advance. You might argue that you should simply reject bad data but this is sometimes impractical.

CRICOS PROVIDER #00120C

Security testing

- Security testing aims to find vulnerabilities that may be exploited by an attacker and to provide convincing evidence that the system is sufficiently secure.
- The tests should demonstrate that the system can resist attacks on its availability, attacks that try to inject malware and attacks that try to corrupt or steal users' data and identity.
- Comprehensive security testing requires specialist knowledge of software vulnerabilities and approaches to testing that can find these vulnerabilities.

CRICOS PROVIDER #00120C

Risk-based security testing

- A risk-based approach to security testing involves identifying common risks and developing tests to demonstrate that the system protects itself from these risks.
- You may also use automated tools that scan your system to check for known vulnerabilities, such as unused HTTP ports being left open.
- Based on the risks that have been identified, you then design tests and checks to see if the system is vulnerable.
- It may be possible to construct automated tests for some of these checks, but others inevitably involve manual checking of the system's behaviour and its files.

CRICOS PROVIDER #00120C

Examples of security risks

- Unauthorized attacker gains access to a system using authorized credentials
- Authorized individual accesses resources that are forbidden to them
- Authentication system fails to detect unauthorized attacker
- Attacker gains access to database using SQL poisoning attack
- Improper management of HTTP session
- HTTP session cookies revealed to attacker
- Confidential data are unencrypted
- Encryption keys are leaked to potential attackers

CRICOS PROVIDER #00120C

Risk analysis

- Once you have identified security risks, you then analyze them to assess how they might arise. For example, for the first risk two slides earlier (unauthorized attacker) there are several possibilities:
  - The user has set weak passwords that can be guessed by an attacker.
  - The system's password file has been stolen and passwords discovered by attacker.
  - The user has not set up two-factor authentication.
  - An attacker has discovered credentials of a legitimate user through social engineering techniques.
- You can then develop tests to check some of these possibilities.
  - For example, you might run a test to check that the code that allows users to set their passwords always checks the strength of passwords.

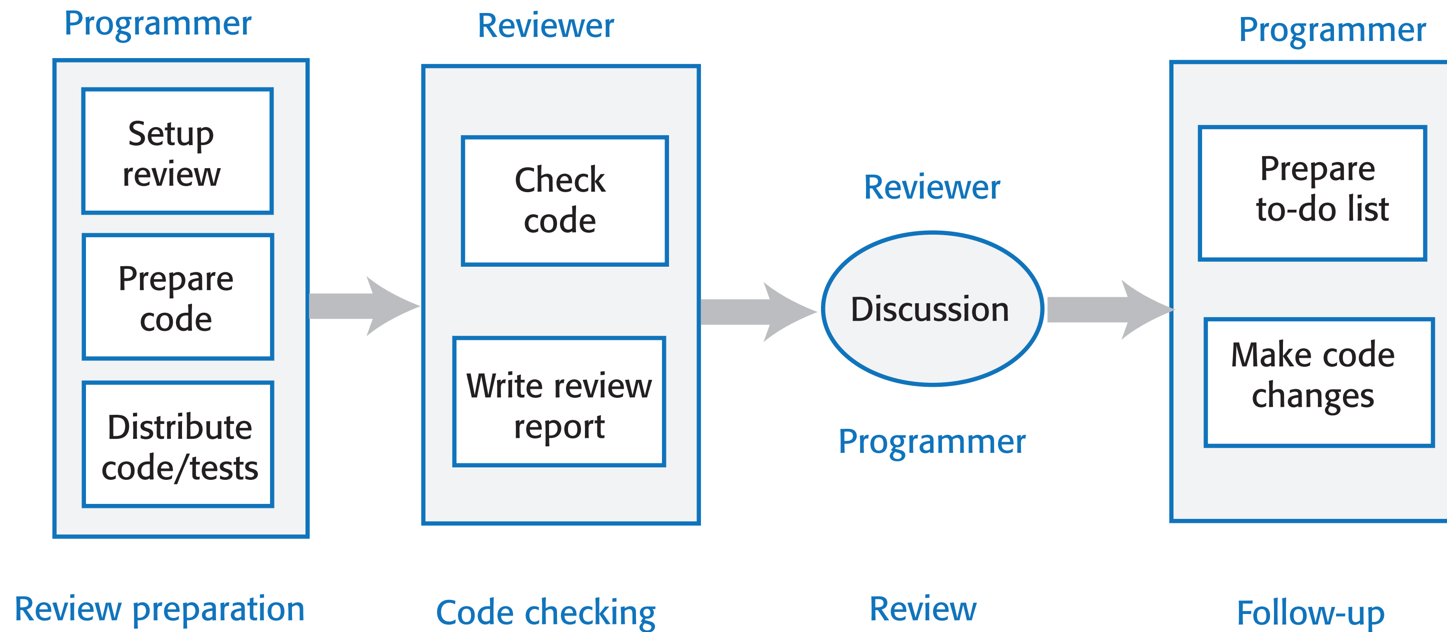
CRICOS PROVIDER #00120C

Remember Code Reviews?

- Code reviews involve one or more people examining the code to check for errors and anomalies and discussing issues with the developer.
- If problems are identified, it is the developer's responsibility to change the code to fix the problems.
- Code reviews complement testing. They are effective in finding bugs that arise through misunderstandings and bugs that may only arise when unusual sequences of code are executed.
- Many software companies insist that all code has to go through a process of code review before it is integrated into the product codebase.

CRICOS PROVIDER #00120C

Code reviews



CRICOS PROVIDER #00120C

Code review activities (1)

- ***Setup review***
  The programmer contacts a reviewer and arranges a review date.
- ***Prepare code***
  The programmer collects the code and tests for review and annotates them with information for the reviewer about the intended purpose of the code and tests.
- ***Distribute code/tests***
  The programmer sends code and tests to the reviewer.
- ***Check code***
  The reviewer systematically checks the code and tests against their understanding of what they are supposed to do.
- ***Write review report***
  The reviewer annotates the code and tests with a report of the issues to be discussed at the review meeting.

CRICOS PROVIDER #00120C

Code review activities (2)

- ***Discussion***
  The reviewer and programmer discuss the issues and agree on the actions to resolve these.
- ***Make to-do list***
  The programmer documents the outcome of the review as a to-do list and shares this with the reviewer.
- ***Make code changes***
  The programmer modifies their code and tests to address the issues raised in the review.

CRICOS PROVIDER #00120C

Part of a checklist for a Python code review

- ***Are meaningful variable and function names used? (General)***
  Meaningful names make a program easier to read and understand.

- ***Have all data errors been considered and tests written for them? (General)***
  It is easy to write tests for the most common cases but it is equally important to check that the program won't fail when presented with incorrect data.

- **Are all exceptions explicitly handled? (General)**
  Unhandled exceptions may cause a system to crash.

- **Are default function parameters used? (Python)**
  Python allows default values to be set for function parameters when the function is defined. This often leads to errors when programmers forget about or misuse them.

- ***Are types used consistently? (Python)***
  Python does not have compile-time type checking so it it is possible to assign values of different types to the same variable. This is best avoided but, if used, it should be justified.

- ***Is the indentation level correct? (Python)***
  Python uses indentation rather than explicit brackets after conditional statements to indicate the code to be executed if the condition is true or false. If the code is not properly indented in nested conditionals this may mean that incorrect code is executed.

CRICOS PROVIDER #00120C

Mini Break in Monday Lecture

CRICOS PROVIDER #00120C

INTRO TO QA AND TESTING (TAKE 2 ☺)

CRICOS PROVIDER #00120C

What is Testing???

- What is testing?
  - Execution of code on sample inputs in a controlled environment
- Principle goals:
  - Validation: program meets requirements, including quality attributes.
  - Defect testing: reveal failures.
- Other goals:
  - Reveal bugs (main goal)
  - Assess quality (hard to quantify)
  - Clarify the specification, documentation
  - Verify contracts

CRICOS PROVIDER #00120C

What is Testing???

- What can we test for? (Software quality attributes)
  - What can we not test for?
- Why should we test? What does testing achieve?
  - What does testing not achieve?
- When should we test?
  - And where should we run the tests?
- What should we test?
  - What CAN we test?
- How should we test?
  - How many ways can you test the sort() function?
- How good are our tests?
  - How to measure test quality?

CRICOS PROVIDER #00120C

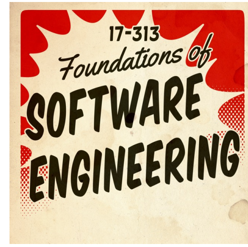
WHAT CAN WE RUN (AUTOMATED) TESTS FOR? (SOFTWARE QUALITY ATTRIBUTES)

CRICOS PROVIDER #00120C

WHAT CAN WE NOT (EASILY) TEST FOR? (SOFTWARE QUALITY ATTRIBUTES)

CRICOS PROVIDER #00120C

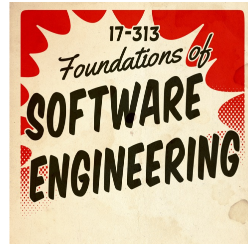
Things we might try to test



- **Program/system functionality:**
  - **Execution space (white box).**
  - **Input or requirements space (black box).**
- The expected user experience (usability).
  - GUI testing, A/B testing
- The expected performance envelope (performance, reliability, robustness, integration).
  - Security, robustness, fuzz, and infrastructure testing.
  - Performance and reliability: soak and stress testing.
  - Integration and reliability: API/protocol testing

CRICOS PROVIDER #00120C

Software Errors



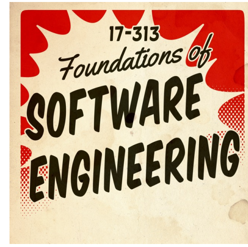
- Functional errors
- Performance errors
- Deadlock
- Race conditions
- Boundary errors
- Buffer overflow
- Integration errors
- Usability errors
- Robustness errors
- Load errors
- Design defects
- Versioning and configuration errors
- Hardware errors
- State management errors
- Metadata errors
- Error-handling errors
- User interface errors
- API usage errors
- …

CRICOS PROVIDER #00120C

WHY SHOULD WE TEST?
(WHAT DOES TESTING HELP US ACHIEVE?)

CRICOS PROVIDER #00120C

Value of Testing



- [Low bar] Ensure that our software meets requirements, is correct, etc.
- Preventing bugs or quality degradations from being accidentally introduced in the future
- Helps uncover unexpected behaviors that can't be identified by reading source code
- Increased confidence in changes ("will I break the internet with this commit?")
- Bridges the gap between a declarative view of the system (i.e., requirements) and an imperative view (i.e., implementation) by means of redundancy.
- Tests are executable documentation; increases code maintainability
- Forces writing testable code <-> checks software design

CRICOS PROVIDER #00120C

WHAT ARE THE LIMITATIONS OF TESTING? (WHAT DOES TESTING NOT ACHIEVE?)

CRICOS PROVIDER #00120C

Limitations of Testing

"Testing shows the presence, not the absence of bugs.”

-Edsger W. Dijkstra

- Testing doesn’t really give any formal assurances
- Writing tests is hard, time consuming
- Knowing if your tests are good enough is not obvious
- Executing tests can be expensive, especially as software complexity and configuration space grows
  - Full test suite for a single large app can take several days to run

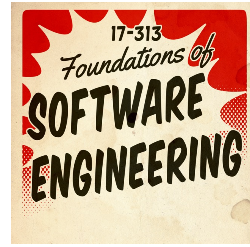
CRICOS PROVIDER #00120C

WHEN SHOULD WE TEST?
(AND WHERE SHOULD WE RUN THE TESTS?)

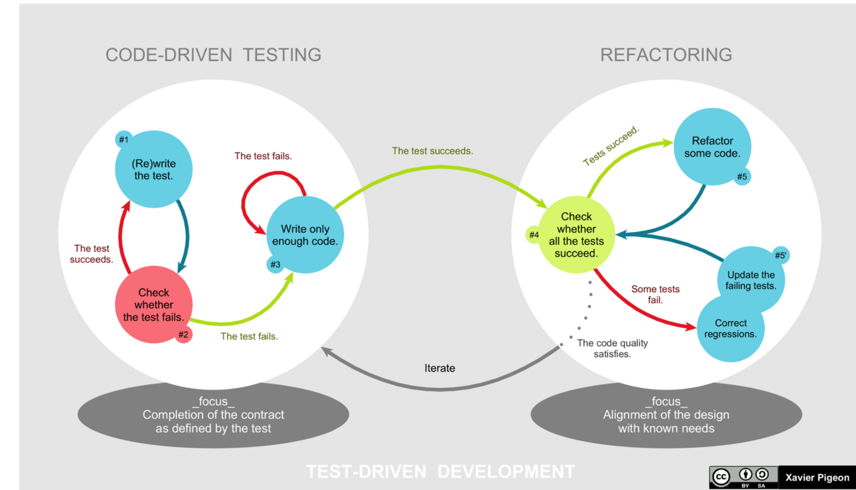
CRICOS PROVIDER #00120C

CRICOS PROVIDER #00120C

Test Driven Development (TDD)

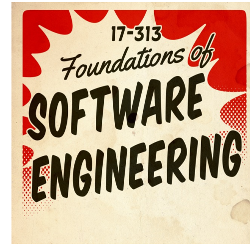


- Tests first!
- Popular agile technique
- Write tests as specifications before code
- Never write code without a failing test
- Claims:
  - Design approach toward testable design
  - Think about interfaces first
  - Avoid unneeded code
  - Higher product quality
  - Higher test suite quality
  - Higher overall productivity



CRICOS PROVIDER #00120C

Common bar for contributions



Chromium

- **Changes should include corresponding tests.** Automated testing is at the heart of how we move forward as a project. All changes should include corresponding tests so we can ensure that there is good coverage for code and that future changes will be less likely to regress functionality. Protect your code with tests!

Firefox

Testing Policy

Everything that lands in mozilla-central includes automated tests by default. Every commit has tests that cover every major piece of functionality and expected input conditions.

Docker

Conventions

Fork the repo and make changes on your fork in a feature branch:

- If it's a bugfix branch, name it XXX-something where XXX is the number of the issue
- If it's a feature branch, create an enhancement issue to announce your intentions, and name it XXX-something where XXX is the number of the issue.

Submit unit tests for your changes. Go has a great test framework built in; use it! Take a look at existing tests for inspiration. Run the full test suite on your branch before submitting a pull request.

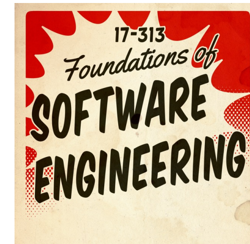
CRICOS PROVIDER #00120C

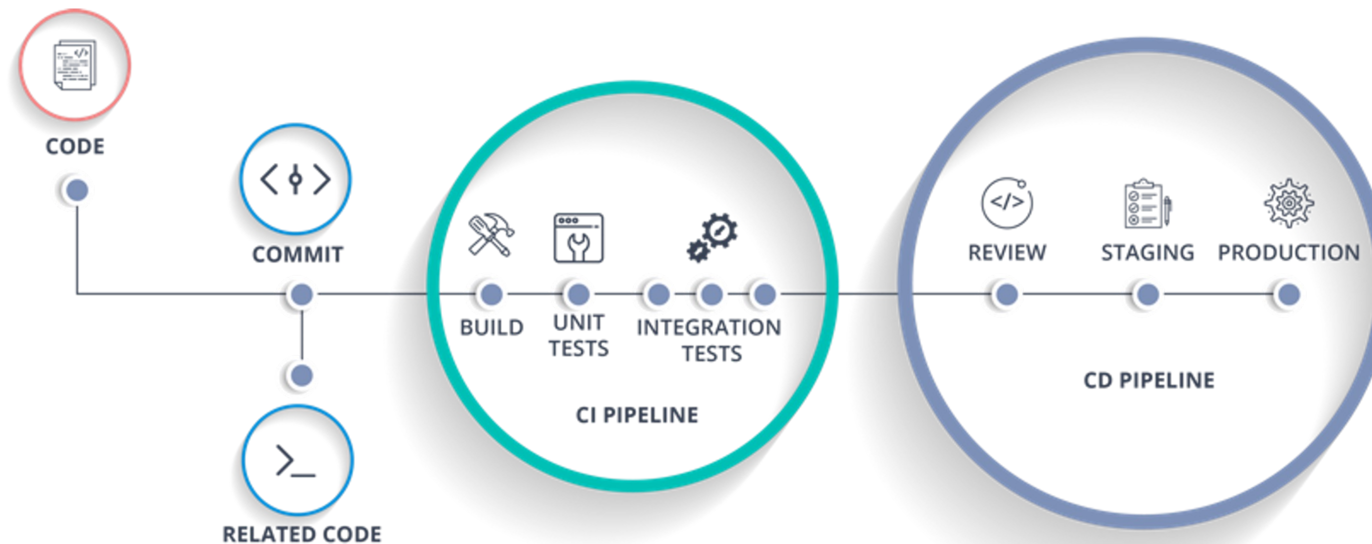
Regression testing

- Usual model:
  - Introduce regression tests for bug fixes, etc.
  - Compare results as code evolves
    - **Code1** + **TestSet** → **TestResults1**
    - **Code2** + **TestSet** → **TestResults2**
  - As code evolves, compare **TestResults1** with **TestResults2**, etc.
- Benefits:
  - Ensure bug fixes remain in place and bugs do not reappear.
  - Reduces reliance on specifications, as **<TestSet,TestResults1>** acts as one.

CRICOS PROVIDER #00120C

Continuous Integration





CRICOS PROVIDER #00120C

WHAT SHOULD WE TEST?
(WHAT CAN WE TEST?)

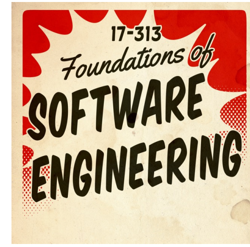
CRICOS PROVIDER #00120C

Testing Levels

- **Unit testing**
- **Integration testing**
- **System testing**

CRICOS PROVIDER #00120C

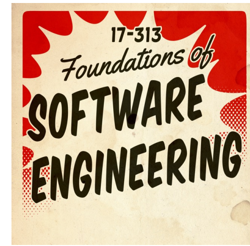
Testing Levels



- **Unit testing**
  - Code level, E.g. is a function implemented correctly?
  - Does not require setting up a complex environment
- **Integration testing**
  - Do components interact correctly? E.g. a feature that cuts across client and server.
  - Usually requires some environment setup, but can abstract/mock out other components that are not being tested (e.g. network)
- **System testing**
  - **Validating the whole system end-to-end (E2E)**
  - Requires complete deployment in a staging area, but fake data
- **Testing in production**
  - **Real data but more risks**

CRICOS PROVIDER #00120C

What's a good distribution of test levels?



E2E
Integration
Unit

E2E
Integration
Unit

CRICOS PROVIDER #00120C

HOW GOOD ARE OUR TESTS?
(HOW CAN WE MEASURE TEST QUALITY?)

CRICOS PROVIDER #00120C

Code Coverage

- Line coverage
- Statement coverage
- Branch coverage
- Instruction coverage
- Basic-block coverage
- Edge coverage
- Path coverage
- …

CRICOS PROVIDER #00120C

Code Coverage

LCOV - code coverage report

Current view: top level - test
Test: coverage.info
Date: 2018-02-07 13:06:43

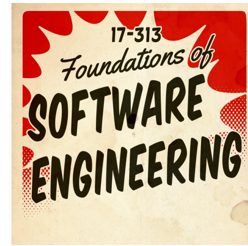
	Hit	Total	Coverage
Lines:	6092	7293	83.5 %
Functions:	481	518	92.9 %

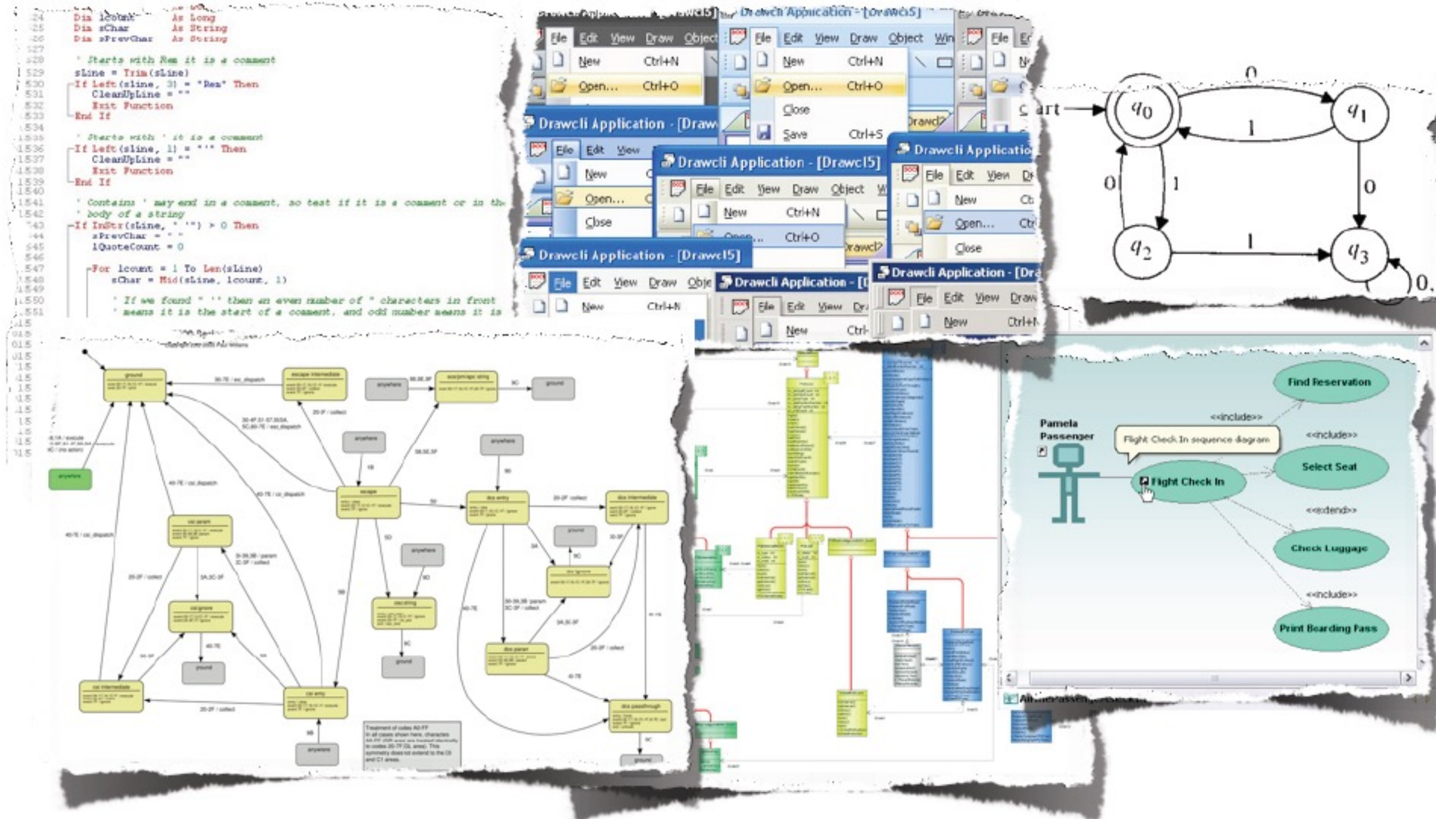
Filename	Line Coverage			
asn1_string_table_test.c	58.8 %	20 / 34		
asn1_time_test.c	72.0 %	72 / 100		
bad_dtls_test.c	97.6 %	163 / 167		
bftest.c	65.3 %	64 / 98		
bio_enc_test.c	78.7 %	74 / 94		
bntest.c	97.7 %	1038 / 1062		
chacha_internal_test.c	83.3 %	10 / 12		
ciphername_test.c	60.4 %	32 / 53		
crltest.c	100.0 %	90 / 90		
ct_test.c	95.5 %	212 / 222		
d2i_test.c	72.9 %	35 / 48		
danetest.c	75.5 %	123 / 163		
dhtest.c	84.6 %	88 / 104		
drbgtest.c	69.8 %	157 / 225		
dtls_mtu_test.c	86.8 %	59 / 68		
dtlstest.c	97.1 %	34 / 35		
dtlsv1listentest.c	94.9 %	37 / 39		
ecdsatest.c	94.0 %	140 / 149		
enginetest.c	92.8 %	141 / 152		
evp_extra_test.c	100.0 %	112 / 112		
fatalerrtest.c	89.3 %	25 / 28		
handshake_helper.c	84.7 %	494 / 583		
hmactest.c	100.0 %	71 / 71		
ideatest.c	100.0 %	30 / 30	100.0 %	4 / 4
igetest.c	87.9 %	109 / 124	100.0 %	11 / 11
lhash_test.c	78.6 %	66 / 84	100.0 %	8 / 8
mdc2_internal_test.c	81.8 %	9 / 11	100.0 %	2 / 2
mdc2test.c	100.0 %	18 / 18	100.0 %	2 / 2
ocspapitest.c	95.5 %	64 / 67	100.0 %	4 / 4
packettest.c	100.0 %	248 / 248	100.0 %	24 / 24

```
 98     0 /   1:                 goto fail;
 99           :     }
100           :     else {
101           :         /* DSA, ECDSA - just use the SHA1 hash */
102     0 /   1:         dataToSign = &hashes[SSL_MD5_DIGEST_LEN];
103     0 /   1:         dataToSignLen = SSL_SHA1_DIGEST_LEN;
104           :     }
105           :
106     1 /   1:     hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
107     1 /   1:      hashOut.length = SSL_SHA1_DIGEST_LEN;
108     1 /   1:      if ((err = SSLFreeBuffer(&hashCtx)) != 0)
109     0 /   1:          goto fail;
110           :
111     1 /   1:      if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
112     0 /   1:          goto fail;
113     1 /   1:      if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
114     0 /   1:          goto fail;
115     1 /   1:      if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
116     0 /   1:          goto fail;
117     1 /   1:      if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
118     0 /   1:          goto fail;
119     1 /   1:          goto fail;
120           :      if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
121           :          goto fail;
122           :
123           :     err = sslRawVerify(ctx,
124           :                        ctx->peerPubKey,
125           :                        dataToSign,                /* plaintext */
126           :                        dataToSignLen,             /* plaintext le
127           :                        signature,
128           :                        signatureLen);
129           :     if(err) {
130           :         sslErrorLog("SSLDecodeSignedServerKeyExchange: sslRawVerify "
131           :                     "returned %d\n", (int)err);
132           :         goto fail;
133           :     }
134           :
135           : fail:
136     1 /   1:      SSLFreeBuffer(&signedHashes);
137     1 /   1:      SSLFreeBuffer(&hashCtx);
138     1 /   1:      return err;
139           :
140     1 /   1: }
141           :
```

CRICOS PROVIDER #00120C

We can measure coverage on almost anything





A. Zeller, Testing and Debugging Advanced course, 2010

CRICOS PROVIDER #00120C

Beware of coverage chasing

- Recall: issues with metrics and incentives
- Also: Numbers can be deceptive
  - 100% coverage != exhaustively tested
- "Coverage is not strongly correlated with suite effectiveness"
  - Based on empirical study on GitHub projects [Inozemtseva and Holmes, ICSE'14]
- Still, it's a good low bar
  - Code that is not executed has definitely not been tested

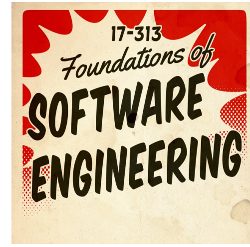
CRICOS PROVIDER #00120C

Coverage of what?

- Distinguish code being tested and code being executed
- Library code >>>> Application code
  - Can selectively measure coverage
- All application code >>> code being tested
  - Not always easy to do this within an application

CRICOS PROVIDER #00120C

Coverage != Outcome



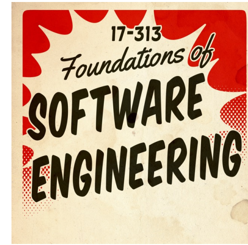
- What’s better, tests that always pass or tests that always fail?
- Tests should ideally be *falsifiable.* Boundary determines specification
- Ideally:
  - Correct implementations should pass all tests
  - Buggy code should fail at least one test
  - Intuition behind *mutation testing*
- What if tests have bugs?
  - Pass on buggy code or fail on correct code
- Even worse: flaky tests
  - Pass or fail on the same test case nondeterministically
- What’s the worst type of test?

CRICOS PROVIDER #00120C

HOW SHOULD WE TEST?

CRICOS PROVIDER #00120C

JUnit



- Popular unit-testing framework for Java
- Easy to use
- Tool support available (Maven, Gradle, etc.)
- Can be used as design mechanism

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

import java.util.*;

public class Tester {
    @Test
    public void testSort() {
        int[] input = {8, 16, 15, 4, 42, 23};
        int[] output = {4, 8, 15, 16, 23, 42};
        assertArrayEquals(sort(input), output);
    }

    int[] sort(int[] args) {
        List<Integer> in = new ArrayList();
```

CRICOS PROVIDER #00120C

Basic Elements of a Test

- Tests usually need an *input* and *expected output.*

```
@Test
public void testSort() {
    int[] input = {8, 16, 15, 4, 42, 23};
    int[] output = {4, 8, 15, 16, 23, 42};
    assertArrayEquals(sort(input), output);
}
```

- More generally, a *test environment,* a *test harness*, and a *test oracle*
  - **Environment**: Resources needed to execute a family of tests
  - **Harness**: Triggers execution of a test case (aka *entry point*)
  - **Oracle**: A mechanism for determining whether a test was successful

CRICOS PROVIDER #00120C

Test Design principles

- Use public APIs only
- Clearly distinguish inputs, configuration, execution, and oracle
- Be simple; avoid complex control flow such as conditionals and loops
- Tests shouldn't need to be frequently changed or refactored
  - Definitely not as frequently as the code being tested changes

CRICOS PROVIDER #00120C

Anti-patterns

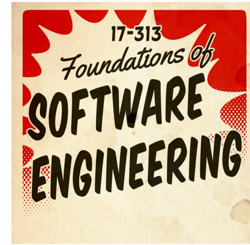
- Snoopy oracles
  - Relying on implementation state instead of observable behavior
  - E.g. Checking variables or fields instead of return values
- Brittle tests
  - Overfitting to special-case behavior instead of general principle
  - E.g. hard-coding message strings instead of behavior
- Slow tests
  - Self-explanatory (beware of heavy environments, I/O, and `sleep()`)
- Flaky tests
  - Tests that pass or fail nondeterministically
  - Often because of reliance on random inputs, timing (e.g. `sleep(1000)`), availability of external services (e.g. fetching data over the network in a unit test), or dependency on order of test execution (e.g. previous test sets up global variables in certain way)

CRICOS PROVIDER #00120C

TEST STRATEGIES

CRICOS PROVIDER #00120C

Basic Unit Test for Sort

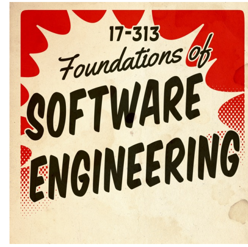



```java
@Test
public void testSort() {
    var input = Arrays.asList(1, 3, 2);
    var output = Arrays.asList(1, 2, 3);
    Collections.sort(input);
    assertEquals(input, output);
}
```

- What are some interesting values to test?
  - List tuples <input, output, reason>

CRICOS PROVIDER #00120C

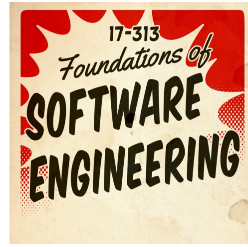
Black-box & Specification-Based Testing



- Test cases are often designed based on behavioral equivalence classes.
  - *Assumption*: if test passes for one value => test will pass for all values in the equivalence class.
- Systematic tests can be drawn from specification.
  - For example: A year is a *leap year* if:
    - the year is divisible by 4;
    - and the year is not divisible by 100;
      - except when the year is divisible by 400
  - Tests:
    - assert isLeapYear(1945) == false
    - assert isLeapYear(1944) == true
    - assert isLeapYear(1900) == false
    - assert isLeapYear(2000) == true

CRICOS PROVIDER #00120C

Boundary-Value Testing



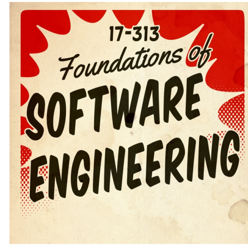
- *Aim*: Test for cases that are at the "boundary" of equivalence classes in the specification.
  - Small change in input moves it from one class to another.
  - Example: Testing a function *divide*(int a, int b)
    - One boundary may be at `a == b`

- *Edge case:* One of many parameters are at the boundary
  - E.g. for *divide*: a=0, b=42 or a=42, b = 0
  - E.g. for *sort*: list contains duplicates, list is empty

- *Corner* case: Combination of parameters are at the boundary
  - E.g. for *divide*: a=0, b=0

CRICOS PROVIDER #00120C

White-box or Structural Testing



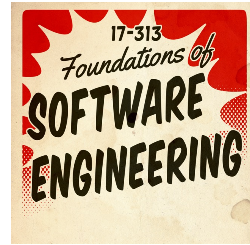
- *Aim*: Test for cases that exercise various program elements (e.g. functions, lines, statements, branches)

- *Key idea*: If you don't execute some code, you can't find bugs in that code. So, let's execute all the code.

- Which one do you think is harder: black-box boundary-value testing or white-box structural testing?

CRICOS PROVIDER #00120C

Coverage of the Basic Unit Test




```
alex@kanga TestingExamples % ./run-pytest.sh
================================================ test session starts ================================================
platform darwin -- Python 3.10.4, pytest-7.1.3, pluggy-1.0.0
rootdir: /Users/alex/Dropbox/Teaching/COMP2120/2022S2/TestingExamples
collected 8 items

bubble_sort.py ..                                                                                            [ 25%]
insertion_sort.py ..                                                                                         [ 50%]
merge_sort.py ..                                                                                             [ 75%]
tim_sort.py ..                                                                                               [100%]

================================================= 8 passed in 0.01s =================================================
Name                 Stmts   Miss  Cover   Missing
--------------------------------------------------
bubble_sort.py          14      0   100%
insertion_sort.py       16      0   100%
merge_sort.py           31      0   100%
tim_sort.py             64     33    48%   6-7, 21-51, 63-67
--------------------------------------------------
TOTAL                  125     33    74%
alex@kanga TestingExamples %
```

CRICOS PROVIDER #00120C

But the basic unit test worked well for Merge and otherSort….

Coverage != Completeness

CRICOS PROVIDER #00120C

Mutation Testing

- *Key idea*: Inject bugs in the program by *mutating* the source code.
- *Ideally*: at least one test should fail on the mutated program (= catch bug).
  - If this happens, the mutant is said to be “killed”.
  - If all tests continue to pass under the mutated program, then the mutant is said to “survive”.
  - Mutation score = (mutants killed) / (total mutants). This is a better predictor of bug-finding capability than coverage.
- *Competent programmer assumption*: programs are mostly correct, except for very small errors.
  - Shows that tests are falsifiable at the boundary of implementation (as opposed to boundary of specification).

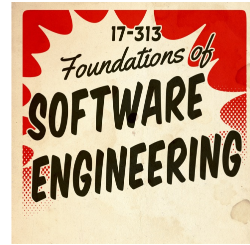
CRICOS PROVIDER #00120C

Mutation Testing

- Sample mutations include:
  - Change ‘a + b’ to ‘a – b’
  - Change ‘if (a > b)’ to ‘if (a >= b)’ or ‘if(b > a)’
  - Change ‘i++’ to ‘i—’
  - Replace integer variables with 0
  - Change ‘return x’ to ‘return True’ (or some other constant)
  - Delete lines containing void method calls (e.g. ‘x.setFoo(1)’)
  - … and many more
- Over time, standard list of mutators curated by researchers
- Pitest is a popular mutation testing tool for Java (pitest.org)

CRICOS PROVIDER #00120C

Mutation Testing



- Nice idea but has several limitations:
  1. *Equivalent* mutations: Modifications that do not affect program semantics (e.g. affecting the pivot in Quicksort).
  2. Needs a pretty *complete* test oracle: Otherwise, some genuine bugs may never be caught. We'll come back to this point later.
  3. Expensive to run. N mutants require N test executions. Program testing costs scale quadratically (because N also grows with size).

```
private static <T extends Comparable<T>> int partition(T[] array, int left, int right) {
  int mid = (left + right) >>> 1;
  T pivot = array[mid];

  while (left <= right) {
    while (less(array[left], pivot)) {
      ++left;
    }
    while (less(pivot, array[right])) {
      --right;
    }
```

CRICOS PROVIDER #00120C

Test Oracles

- Obvious in some applications (e.g. “sort()”) but more challenging in others (e.g. “encrypt()” or UI-based tests)
- Lack of good oracles can limit the scalability of testing. Easy to generate lots of input data, but not easy to validate if output (or other program behavior) is correct.
- Fortunately, we have some tricks.

CRICOS PROVIDER #00120C

Property-Based Testing

- Intends to validate invariants that are always true of a computed result.
  - E.g. if testing a list-reversing function called `rev`, then we have the invariant: `rev(rev(list)).equals(list)`
- Key idea: Can now easily scale testing to very large data sets, either hand-written or automatically generated, without the need for hard-coding expected outputs completely.

```
@Property
public void testSameLength(List<Integer> input) {
    var output :List<Integer> = sort(input);
    // Check length
    assert output.size() == input.size() : "Length should match";
}
```

CRICOS PROVIDER #00120C

Differential Testing

- If you have two implementations of the same specification, then their output should match on all inputs.
  - E.g. `timSort(x).equals(quickSort(x))` → should always be true
  - Special case of a property test, with a free oracle.
- If a differential test fails, at least one of the two implementations is wrong.
  - But which one?
  - If you have N > 2 implementations, run them all and compare. Majority wins (the odd one out is buggy).
- Differential testing works well when testing programs that implement standard specifications such as compilers, browsers, SQL engines, XML/JSON parsers, media players, etc.
  - Not feasible in general

CRICOS PROVIDER #00120C

Regression Testing

- Differential testing through time (or *versions*, say V1 and V2).

- Assuming V1 and V2 don't add a new feature or fix a known bug, then f(x) in V1 should give the same result as f(x) in V2.

- *Key Idea*: Assume the current version is correct. Run program on current version and log output. Compare all future versions to that output.

CRICOS PROVIDER #00120C

Google’s TotT

https://testing.googleblog.com/search/label/TotT

CRICOS PROVIDER #00120C

Key Points

- The aim of program testing is to find bugs and to show that a program does what its developers expect it to do.
- Four types of testing that are relevant to software products are functional testing, user testing, load and performance testing and security testing.
- Unit testing involves testing program units such as functions or class methods that have a single responsibility. Feature testing focuses on testing individual system features. System testing tests the system as a whole to check for unwanted interactions between features and between the system and its environment.
- Identifying equivalence partitions, in which all inputs have the same characteristics, and choosing test inputs at the boundaries of these partitions, is an effective way of finding bugs in a program.
- User stories may be used as a basis for deriving feature tests.
- Test automation is based on the idea that tests should be executable. You develop a set of executable tests and run these each time you make a change to a system.

CRICOS PROVIDER #00120C

Key Points

- The structure of an automated unit test should be arrange-action-assert. You set up the test parameters, call the function or method being tested, and make an assertion of what should be true after the action has been completed.
- Test-driven development is an approach to development where executable tests are written before the code. Code is then developed to pass the tests.
- A disadvantage of test-driven development is that programmers focus on the detail of passing tests rather than considering the broader structure of their code and algorithms used.
- Security testing may be risk driven where a list of security risks is used to identify tests that may identify system vulnerabilities.
- Code reviews are an effective supplement to testing. They involve people checking the code to comment on the code quality and to look for bugs.

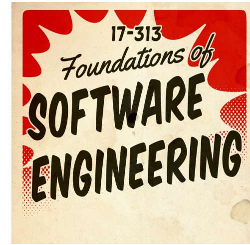
CRICOS PROVIDER #00120C

Key Points

- Most tests that you will write will be muuuuuuch more complex than testing a sort function.
- Need to set up environment, create objects whose methods to test, create objects for test data, get all these into an interesting state, test multiple APIs with varying arguments, etc.
- Many tests will require mocks (i.e., faking a resource-intensive component).
- General principles of many of these strategies still apply:
  - Writing tests can be time consuming
  - Determining test adequacy can be hard (if not impossible)
  - Test oracles are not easy
  - Advanced test strategies have trade-offs (high costs with high returns)

CRICOS PROVIDER #00120C

Key Points



- Identify the scope and limitations of software testing
- Appreciate software testing as a methodology to use automation in improving software quality
- Estimate the costs of testing and discuss trade-offs of running tests at different times in the software development lifecycle
- Measure the quality of software tests and define test adequacy criteria
- Enumerate different levels of testing such as unit testing, integration testing, system testing, and testing in production
- Describe the principles of test-driven development
- Outline design principles for writing good tests
- Recognize and avoid testing anti-patterns

CRICOS PROVIDER #00120C

Key Points

- Enumerate various strategies for picking test cases, such as:
  - Specification-based testing
  - Boundary-value testing
  - Structural testing
  - Property testing
  - Regression testing
  - Differential testing
  - Property-based testing
  - Mutation testing

CRICOS PROVIDER #00120C