Software Engineering

Testing: Functional testing, Test automation, Test-driven development, and Code reviews

Min-Yuh Day
Associate Professor
Institute of Information Management, National Taipei University

https://web.ntpu.edu.tw/~myday
2020-12-21
<table>
<thead>
<tr>
<th>週次 (Week)</th>
<th>日期 (Date)</th>
<th>內容 (Subject/Topics)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2020/09/15</td>
<td>軟體工程概論 (Introduction to Software Engineering)</td>
</tr>
<tr>
<td>2</td>
<td>2020/09/22</td>
<td>軟體產品與專案管理：軟體產品管理，原型設計 (Software Products and Project Management: Software product management and prototyping)</td>
</tr>
<tr>
<td>3</td>
<td>2020/09/29</td>
<td>敏捷軟體工程：敏捷方法、Scrum、極限程式設計 (Agile Software Engineering: Agile methods, Scrum, and Extreme Programming)</td>
</tr>
<tr>
<td>4</td>
<td>2020/10/06</td>
<td>功能、場景和故事 (Features, Scenarios, and Stories)</td>
</tr>
<tr>
<td>5</td>
<td>2020/10/13</td>
<td>軟體架構：架構設計、系統分解、分散式架構 (Software Architecture: Architectural design, System decomposition, and Distribution architecture)</td>
</tr>
<tr>
<td>6</td>
<td>2020/10/20</td>
<td>軟體工程個案研究 I (Case Study on Software Engineering I)</td>
</tr>
</tbody>
</table>
週次 (Week)  日期 (Date)  內容 (Subject/Topics)
7  2020/10/27  基於雲的軟體：虛擬化和容器、軟體即服務
            (Cloud-Based Software: Virtualization and containers,
             Everything as a service, Software as a service)
8  2020/11/03  雲端運算與雲軟體架構
            (Cloud Computing and Cloud Software Architecture)
9  2020/11/10  期中報告 (Midterm Project Report)
10 2020/11/17  微服務架構：RESTful服務、服務部署
            (Microservices Architecture: RESTful services,
             Service deployment)
11 2020/11/24  軟體工程產業實務
            (Industry Practices of Software Engineering)
12 2020/12/01  安全和隱私 (Security and Privacy)
週次 (Week) 日期 (Date)  內容 (Subject/Topics)
13 2020/12/08  軟體工程個案研究 II
   (Case Study on Software Engineering II)
14 2020/12/15  可靠的程式設計 (Reliable Programming)
15 2020/12/22  測試：功能測試、測試自動化、
   測試驅動的開發、程式碼審查
   (Testing: Functional testing, Test automation,
    Test-driven development, and Code reviews)
16 2020/12/29  DevOps和程式碼管理：
   程式碼管理和DevOps自動化
   (DevOps and Code Management:
    Code management and DevOps automation)
17 2021/01/05  期末報告 I (Final Project Report I)
18 2021/01/12  期末報告 II (Final Project Report I)
Software Engineering and Project Management

- **Analyze**
  - Requirements definition

- **Design**
  - System and Software design

- **Build**
  - Implementation and unit testing

- **Test**
  - Integration and system testing

- **Deliver**
  - Operation and maintenance

---

**Project Management**
Product management concerns

- Business needs
- Technology constraints
- Customer experience

Technical interactions of product managers

Product vision management
Product backlog management
User stories and scenarios
Acceptance testing
Customer testing
User interface design

Software Development Life Cycle (SDLC)

The waterfall model

- Requirements definition
- System and Software design
- Implementation and unit testing
- Integration and system testing
- Operation and maintenance

Plan-based and Agile development

Plan-based development

- Requirements engineering
- Requirements specification
- Design and implementation

Requirements change requests

Agile development

- Requirements engineering
- Design and implementation

The Continuum of Life Cycles

- **Incremental**
  - Low Degree of Change
  - High Frequency of Delivery

- **Agile**
  - High Degree of Change
  - High Frequency of Delivery

- **Predictive**
  - Low Degree of Change
  - Low Frequency of Delivery

- **Iterative**
  - High Degree of Change
  - Low Frequency of Delivery

Predictive Life Cycle

Analyze → Design → Build → Test → Deliver

Iterative Life Cycle

1. Analyze
2. Analyze Design
3. Build Test
4. Deliver

Prototype \[\rightarrow\] Refine

A Life Cycle of Varying-Sized Increments

- Analyze
- Design
- Build
- Test
- Deliver

# Iteration-Based and Flow-Based Agile Life Cycles

## Iteration-Based Agile

<table>
<thead>
<tr>
<th>Requirements Analysis</th>
<th>Requirements Analysis</th>
<th>Requirements Analysis</th>
<th>Requirements Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Analysis</td>
<td>Analysis</td>
<td>Repeat as needed</td>
</tr>
<tr>
<td>Design</td>
<td>Design</td>
<td>Design</td>
<td></td>
</tr>
<tr>
<td>Build</td>
<td>Build</td>
<td>Build</td>
<td>Repeat as needed</td>
</tr>
<tr>
<td>Test</td>
<td>Test</td>
<td>Test</td>
<td></td>
</tr>
</tbody>
</table>

Repeat as needed ...

## Flow-Based Agile

<table>
<thead>
<tr>
<th>Requirements Analysis</th>
<th>Requirements Analysis</th>
<th>Requirements Analysis</th>
<th>Requirements Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analysis</td>
<td>Analysis</td>
<td>Analysis</td>
<td>Repeat as needed</td>
</tr>
<tr>
<td>Design</td>
<td>Design</td>
<td>Design</td>
<td>Repeat as needed</td>
</tr>
<tr>
<td>Build</td>
<td>Build</td>
<td>Build</td>
<td>Repeat as needed</td>
</tr>
<tr>
<td>Test</td>
<td>Test</td>
<td>Test</td>
<td>Repeat as needed</td>
</tr>
</tbody>
</table>

Repeat as needed ...

the number of features in the WIP limit

From personas to features

1. **Personas**
   - A way of representing users

2. **Scenarios**
   - Natural language descriptions of a user interacting with a software product

3. **Stories**
   - Natural language descriptions of something that is needed or wanted by users

4. **Features**
   - Fragments of product functionality

Service-oriented Architecture

Client 1
Client 2
Client 3
Client ...

Web Server

Service gateway

S1
S2
S3
S4
S5
S6

Services

Everything as a service

Software as a service (SaaS)

Platform as a service (PaaS)

Infrastructure as a service (IaaS)

Cloud data center

Photo editing

Cloud management

Monitoring

Storage

Network

Logistics management

Database software development

Computing virtualization

Software as a service

Software customers

Software provider

Cloud provider

Software services

Cloud Infrastructure

Microservices architecture - key design questions

What are the microservices that make up the system?

How should data be distributed and shared?

How should the microservices in the system be coordinated?

How should microservices communicate with each other?

How should service failure be detected, reported and managed?
Types of security threat

**Availability threats**

An attacker attempts to deny access to the system for legitimate users

Distributed denial of service (DDoS) attack

**Integrity threats**

An attacker attempts to damage the system or its data

Virus

Ransomware

**Confidentiality threats**

An attacker tries to gain access to private information held by the system

Data theft

Software product quality attributes

1. Reliability
2. Availability
3. Resilience
4. Maintainability
5. Responsiveness
6. Usability
7. Security

A refactoring process

1. Identify code ‘smell’
2. Identify refactoring strategy
3. Make small improvement until strategy completed
4. Run automated code tests

Testing: Functional testing, Test automation, Test-driven development, and Code reviews
Outline

• Software testing
• Functional testing
• Test automation
• Test-driven development
• Code reviews
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.

### 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: ‘off-by-1’ error
  - **Understanding errors**
    - You have misunderstood or have been unaware of some of the details of what the program is supposed to do.

# Types of testing

<table>
<thead>
<tr>
<th><strong>Functional testing</strong></th>
<th>Test the functionality of the overall system.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User testing</strong></td>
<td>Test that the software product is useful to and usable by end-users.</td>
</tr>
<tr>
<td><strong>Performance and load testing</strong></td>
<td>Test that the software works quickly and can handle the expected load placed on the system by its users.</td>
</tr>
<tr>
<td><strong>Security testing</strong></td>
<td>Test that the software maintains its integrity and can protect user information from theft and damage.</td>
</tr>
</tbody>
</table>

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

• 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.

Functional testing

Start

1. Unit Testing
2. Feature Testing
3. System Testing
4. Release Testing

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
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.

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.

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.

- **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.

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.
Feature testing

• 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.

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.

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.

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.

Feature tests for sign-in with Google

• **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.

System and release testing

- **System testing** involves testing the **system as a whole**, rather than the individual system features.

System testing

- 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.

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.

Choosing a holiday destination

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.

Release testing

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

• 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.

Release testing and System testing

• 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.

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.

Automated testing

Files of executable tests

Code being tested → Test runner → Testing framework → Test Report

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_zero_principal(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)
Automated tests

• It is good practice to **structure automated tests into three parts:**

  1. **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.

  2. **Action**
     • You call the unit that is being tested with the test parameters.

  3. **Assert**
     • You make an assertion about what should hold if the unit being tested has executed successfully.

  **AssertEquals**: checks if its parameters are equal.

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_name-toolong (self):
        self.assertFalse (namecheck ('Thisisalongstringwithmorethen40charactersfrombeginningtoend'))

    def test_name-too-short (self):
        self.assertFalse (namecheck ('S'))
def test_namewithdigit (self):
    self.assertFalse (namecheck('C-3PO'))

def test_namewithdoublehyphen (self):
    self.assertFalse (namecheck ('--badcode'))

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'))
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)
The test pyramid

- Unit tests
- Feature tests
- System tests

Increased automation
Reduced costs
Feature editing through an API

Interaction recording and playback

Browser or mobile app interface

User action recording → Interaction session records → User action playback

System API

System being tested

Test-driven development (TDD)

- 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.

Test-driven development (TDD)

1. Identify new functionality
2. Identify partial implementation of functionality
3. Write code stub that will fail test
4. Run all automated test
5. Implement code that should cause failing test to pass
6. Run all automated test
7. Refactor code if required

Functionality complete
Functionality incomplete

All tests pass
Test failure

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.

2. **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.

3. **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.

4. **Run all existing automated tests**
   All previous tests should pass. The test for the incomplete code should fail.

5. **Implement code that should cause the failing test to pass**
   Write code to implement the mini-unit, which should cause it to operate correctly

6. **Rerun all automated tests**
   If any tests fail, your code is probably incorrect. Keep working on it until all tests pass.

7. **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.
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.
• 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.
• 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.

Reasons for not using TDD

- TDD discourages radical program change
- I focused on the tests rather than the problem I was trying to solve
- I spent too much time thinking about implementation details rather than the programming problem
- It is hard to write ‘bad data’ tests

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.

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.

Risk analysis

- Once you have identified security risks, you then analyze them to assess how they might arise.
  - 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.

- 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.
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.

Code reviews

Programmer
- Setup review
- Prepare code
- Distribute code/tests

Reviewer
- Check code
- Write review report

Discussion

Reviewer

Programmer
- Prepare to-do list
- Make code Changes

Follow-up

Review preparation

Code checking

Review

Follow-up

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.
Summary

• **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.

• 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.
Summary

• 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.

Summary

• **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.

References

• Titus Winters, Tom Manshreck, and Hyrum Wright (2020), Software Engineering at Google: Lessons Learned from Programming Over Time, O'Reilly Media.