Module 4

This is a single, concatenated file, suitable for printing or saving as a PDF for offline viewing. Please note that some animations or images may not work.

Module 4 Study Guide and Deliverables

Module Topics:	• Topic 1: Implementation • Topic 2: Testing
Readings:	• Online lecture notes • Braude, Parts VI and VII (or related chapters in other textbooks)
Discussions:	• Weekly Group Meeting
Assignments:	• Lab 3 due **Tuesday, October 3 at 6:00 AM ET**
Assessments:	• Quiz 2 due **Tuesday, October 3 at 6:00 AM ET**
Live Classrooms:	• **Tuesday, September 26 from 7:00-9:00 PM ET** • **Thursday, September 28 from 7:00-8:00 PM ET**

Learning Outcomes

By the end of this module, you will be able to do the following:

- Describe different code smells and identify some of those smells from their code.
- Describe different refactoring techniques and apply these techniques to get rid of code smells.
- Describe coding principles revealed by those smells and apply these principles to avoid these smells in coding.
- Explain the definition, importance and different types of testing.
- Explain the relationship between refactoring and testing, and the importance of refactoring in Agile.
- Explain the work flow of TDD and BDD and how to apply them in your group project.
- Explain how to design test cases.
- Write unit test code and develop test cases for your group project.

Introduction

The final product of a software-development project is the working software. Agile values working software over comprehensive documentation. The key is to translate the design (whether in writing or not) into code correctly. Unlike the traditional waterfall model, coding is done after completing the comprehensive design documentation, and testing is done after finishing all coding. Coding is tightly coupled with designing and testing in Agile. In fact, design ideas are usually directly detailed in the production code or testing code, without additional documentation. Testing code is written side by side with or even earlier than the production code. In this module, we will discuss several techniques and practices used in implementation and testing activities, as well as how design principles discussed in the previous model guide these techniques.

Topic 1: Implementation

Programming is not equal to software engineering, but is the biggest part of it. In fact, the majority of time in the software-development life cycle (usually more than 60%) is spent on the construction of software, which is mainly programming or coding. Be aware that detailed design and testing are often intermingled with coding and cannot be clearly separated. Particularly in Agile development, detailed design is often not a separate phase with additional detailed design documents as output. Instead, it is directly reflected in the code. Detailed design documents (such as class/object documents or diagrams) may be generated automatically from the code. This helps improve productivity and avoid inconsistency. Constant refactoring is essential to refine the design of existing code before adding new code to accommodate new features.

The Choice of Programming languages and Frameworks

The first decision to make in software construction is what programming language(s) to use. The choice will affect productivity and code quality. In general, programmers are more productive using a familiar language than an unfamiliar one. Programmers working with high-level languages achieve better productivity and quality than those working with low-level languages. Other factors also affect the choice, such as the industry standards, language characteristics and support, and the compatibility with legacy systems. The learning curves for different languages should also be considered, particularly for student group projects when not all students are familiar with the same programming language.

There are so many different programming languages in use today. When developing user-level applications, commonly used, high-level programming languages include Java, C++, C#, Python, JavaScript, PhP, and Ruby. New programming languages—such as Swift, Go, and Kotlin—are also gaining popularity.

When developing system software—such as operating systems, system utility programs, and compilers—C is often used to support low-level operations, such as manipulating memory addresses and individual bits. Even assembly languages may be used sometimes to control the hardware directly. C is the second-most popular programming language, according to the Tiobe Index as of February 2020, with Java ranked first and Python third.

There are also other programming languages designed more for specific purposes, such as SQL for databases, R for statistics, MATLAB for mathematical models, and Arduino for embedded devices.

Most students choose to develop web or mobile applications as their group projects in this class. Java, Python, JavaScript, and SQL are the most popular languages used in these projects.

Besides proper programming languages, the choice of frameworks also affects software projects. A software framework provides a reusable software environment in which to build applications in a standard way. It provides generic functionality and enables developers to write additional, specific code. It may include support programs, compilers, code libraries, tool sets, and APIs to support software development. Here are a few popular frameworks in use today:

- .NET, for Windows applications
- ASP .NET, for web applications using Microsoft's server-side technologies
- Spring, for Java enterprise applications
- Django, Flask, for web applications in Python
- Ruby on Rail, for web applications in Ruby
- Express, for web applications using Node.js
- Node.js, a server-side JavaScript execution environment
- Angular, a mobile and web development framework using JavaScript
- MEAN, a full-stack tool kit consisting of MongoDB, Express JS, AngularJS, and Node.js
- React, a cross-platform UI development framework using JavaScript

The basic principle of using a framework is to avoid reinventing the wheel. Thus, we can focus on application-specific business logic rather than common tasks. While using a framework is not absolutely necessary, it is very useful for providing high quality at lower cost and within industry standards. The popular frameworks have gone through extensive testing, and most are of better quality than the code you write yourself. While using a framework can improve productivity, the initial learning cost needs to be considered when selecting an unfamiliar framework.

Coding Standards

Besides choosing a proper programming language and framework, applying coding standards is also very important for improving software quality and productivity. It improves the readability, reusability, and maintainability of the code, and helps reduce complexity. It promotes good programming practices and increases the efficiency of the programmers. Some coding standards are common among different languages. Others vary from language to language. For example, using meaningful names is a common practice, but naming conventions may be different in different languages. Java uses camel-case lettering (e.g., localVar), whereas PEP8 for Python uses lower-case letters with underscores (e.g., local_var). It is very important to predefine the coding standards that will be used in the project across the teams, usually including guidelines on naming conventions, indentation, comments, etc. Here are some existing standards for Java and Python. You can directly adopt these standards or base your own customized set on these. Each application framework may also define additional conventions or standards.

- Java Code Conventions by Oracle
- Google Java Style Guide
- PEP 8 – Style Guide for Python Code
- Google Python Style Guide

Meaningful Names

As mentioned previously, using meaningful names is an important coding practice. Uncle Bob spent one chapter on meaningful names (Martin, 2008). Some basic rules follow:

- Use intention-revealing names (e.g., *fileSize* instead of s).
- Avoid disinformation (e.g., avoid the letters *l* and *O*, as they look very like the numbers *1* and *0*).
- Use searchable names, avoiding single letter numbers and numeric constants.
- Use noun or noun-phrase names for classes and objects (e.g., *Student*), and verb or verb-phrase names for methods (e.g., *submit()*).

- Avoid different words for the same purpose and same words for different purposes.
- Add meaningful context, using solution or problem domain names.

For example, compare the following two code snippets (Martin, 2008), which implement the same functionality. The first one is very hard to understand with those meaningless names, such as *getThem, list1, x*, and *theList*. The second one uses meaningful names and is self-explanatory after changing to meaningful names.

```
public List<int[]> getThem()
        {
                List<int[]> list1 = new ArrayList<int[]>();
                for (int[] x: theList)
                        if (x[0] == 4)
                                list1.add(x);
                return list1;
        }


public List<Cell> getFlaggedCells()
        {
                List flaggedCells = new ArrayList();
                for (Cell cell: gameBoard)
                        if (cell.isFlagged())
                                flaggedCells.add(cell);
                return flaggedCells;
        }
```

Test Yourself

What programming language and coding standard do you use in your group project? What framework, if any, do you use in your group project? What are some pros and cons of using that framework?

Code Smells and Refactoring Part 1

Code Smells

No matter which languages or frameworks you choose, there are some common principles and practices to help you write better code. Bad code should be avoided or constantly refactored. Martin Fowler (1999) summarized 22 bad smells in code that can be easily identified in his book *Refactoring: Improving the Design of Existing Code*:

1. Duplicated code
2. Long method
3. Large class
4. Long parameter list
5. Divergent change
6. Shotgun surgery
7. Feature envy
8. Data clumps

9. Primitive obsession
10. Switch statements
11. Parallel inheritance hierarchies
12. Lazy class
13. Speculative generality
14. Temporary field
15. Message chains
16. Middle man
17. Inappropriate intimacy
18. Alternative classes with different interfaces
19. Incomplete library class
20. Data class
21. Refused bequest
22. Comments

You can read his book for more details. You can also find more detailed information about these bad smells and refactoring at Sourcemaking. It classifies bad smells into five categories (with one different smell from the above list):

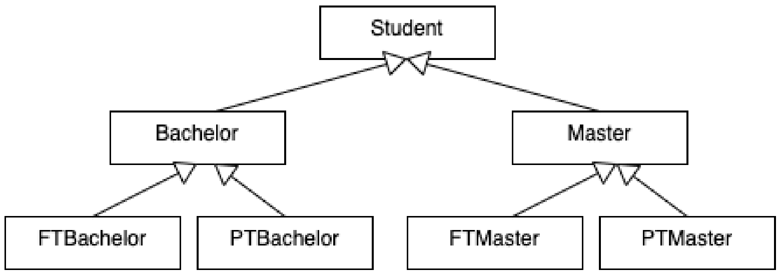
Bloaters **(increased complexity)**	1. Long Method 2. Large Class 3. Primitive Obsession 4. Long Parameter List 5. Data Clumps
Object-Orientation Abusers **(incorrect use or against OO principles)**	1. Switch Statements 2. Temporary Field 3. Refused Bequest 4. Alternative Classes with Different Interfaces
Change Preventers **(hard to make change)**	1. Divergent Change 2. Shotgun Surgery 3. Parallel Inheritance Hierarchies
Dispensables **(unnecessary)**	1. Comments 2. Duplicate Code 3. Lazy Class 4. Data Class 5. Dead Code 6. Speculative Generality
Couplers **(excessive coupling or delegation)**	1. Feature Envy 2. Inappropriate Intimacy 3. Message Chains 4. Middle Man

Some smells are easily understandable. For example, **duplicate code** means that there are the same or similar code segments in different methods or classes. It is a very common smell, as developers like to copy and paste a lot. It is against the DRY principle and easily causes inconsistency. Duplicate code segments may be expression or logic blocks. You should always get rid of duplicate code blocks by extracting duplicate code into separate methods or classes.

Interestingly, **comments** can be smells, too. They are not always good. When the code is highly readable and self-explanatory, we should not need very many comments. Extra comments may imply poor readability of the code. In addition, comments are not testable and may not be changed (or maintained) together with code changes. Inaccurate comments can be far worse than no comments at all. Truth can only be found in code. It is important to minimize unnecessary comments. It can improve code readability. We can get rid of unnecessary comments by using meaningful names and avoiding long methods and large classes. Some comments are necessary or beneficial, including legal comments, explanations of intentions or consequences, TODO comments, and comments to generate API documents.

Smaller is better. **Long methods and large classes** are not only hard to understand, but also more prone to errors and harder to test. Each method or class should also just focus on one thing and do it well. This is the single-responsibility principle. Otherwise, it will be changed in different ways for different reasons. This smell is called **divergent change**. **Shotgun surgery** is an opposite smell: Every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. This can happen after an overzealous application of divergent change. In this case, a single responsibility has been split up among a large number of classes.

The **parallel inheritance hierarchies** smell occurs when you need to create similar sub-subclasses for subclasses. For example, we have bachelor students and master students in our department. Both undergraduate and graduate students can be either full time or part time. Then we may have the following hierarchies. If we need to add PhD students, we will need to add another parallel inheritance hierarchy. This can make changes harder and harder.



Parallel Inheritance–Hierarchy Class Diagram

We cannot always avoid **switch statements**, but we should make sure that each switch statement is buried in a low-level class and never repeated. When the same switch statements are scattered in different places, they should be replaced with polymorphism and interface inheritance. State or strategy patterns are usually used.

Refused bequest occurs when the subclass simply wants to use some methods in the superclass, but is not really in common with the superclass. An example is to use List as the superclass of Stack, introduced in the last module. The Liskov substitution principle should be followed when using the inheritance relationship.

In the last module, we also discussed another important design principle: minimizing coupling and maximizing coherence. This applies at the class level, too. We should try to group related fields and methods in a single class instead of spreading them among multiple classes. **Feature envy** occurs when a method accesses the data of another object more than its own data, and **inappropriate intimacy** occurs when a class uses the internal fields and methods of another class.

The **message chains** and the **middle man** are two opposite smells associated with improper use of delegation. It is important to use delegation properly. When the client depends on navigating along the class structure, resulting in a long message chain, it is preferred to introduce a middle man to hide delegates. On the contrary, overly aggressive delegate hiding can cause the middle-man smell, meaning that it is hard to see where the functionality is actually occurring.

Just delete any unused code, such as unused methods, fields, and classes; avoid commenting them out. Also delete any unnecessary fields, methods, or classes, as well as inline methods or classes that are unnecessarily delegated.

Refactoring

One should try to avoid these bad smells when writing new code, and constantly refactor the code to get rid of any existing bad smells. A **refactoring** is a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior. To refactor is to restructure software by applying a series of refactorings without changing its observable behavior. The main purpose of code refactoring is to make the code more efficient and maintainable. While refactoring itself doesn’t remove bugs, it can help prevent them with much cleaner code.

Martin Fowler introduced a number of refactoring techniques to help get rid of bad smells. He classified these techniques into seven categories:

Composing Methods	1. Extract method 2. Inline method 3. Inline temp (remove a temporary variable) 4. Replace temp with query/function 5. Introduce explaining variable (to replace complicated expression) 6. Split temporary variable (that is used for multiple purposes) 7. Remove assignments to parameters (introduce temp vararariable) 8. Replace method with method object 9. Substitute algorithm
Moving Features between Objects	1. Move method/field 2. Extract class/inline class 3. Hide delegate/remove middle man 4. Introduce foreign method/introduce local extension (for library class)
Organizing Data	1. Self-encapsulate field 2. Replace data value with object (for when basic types aren't enough) 3. Change value to/from reference 4. Replace array with object 5. Duplicate observed data 6. Change bidirectional association to/from unidirectional 7. Replace "magic numbers" with symbolic constants 8. Encapsulate field/collection (accessor methods for private variables) 9. Replace record with data class (hide database implementation) 10. Replace type code with class/subclass 11. Replace type code with state/strategy 12. Replace subclass with fields

Simplifying conditional expressions	1. Simplifying Conditional Expression 2. Decompose/consolidate conditional expression 3. Consolidate duplicate conditional fragments 4. Remove control flag (use break/return) 5. Replace nested conditional with guard clauses (which either returns or throws an exception) 6. Replace conditional with polymorphism 7. Introduce null object 8. Introduce assertion
Making Method Calls Simpler	1. Rename method 2. Add/remove parameter 3. Parameterize method/replace parameter with explicit methods 4. Preserve whole object/introduce parameter objects 5. Replace parameter with method (receiver invokes the method) 6. Remove setting method 7. Separate query from modifier 8. Hide method 9. Replace constructor with factory method 10. Encapsulate downcast 11. Replace error code with exception/replace exception with test
Dealing with generalization	1. Dealing with generalization 2. Pull up/push down field 3. Pull up/push down method 4. Pull up constructor body and replace with *super()* 5. Extract subclass/superclass 6. Extract interface 7. Collapse hierarchy 8. Form template method 9. Replace inheritance with delegation and vice versa
Big Refactoring	1. Tease apart inheritance 2. Convert procedural design to objects 3. Separate domain from presentation 4. Extract hierarchy

Code Smells and Refactoring Part 2

Composing Methods

The extract method and the inline method are opposite techniques. In general, when a method is long or there are repeated code blocks in different methods, we need to extract methods. When a method is very short (e.g., a couple of lines), we should simply inline it.

For simple expressions, we should remove unnecessary temporary variables, particularly if they are not used elsewhere. But for complicated expressions, we may want to introduce additional, explaining variables for parts or the results of the expressions.

In general, each variable should be used just for one purpose with a meaningful name to avoid confusion. Splitting temporary variables and removing assignments to parameters address this issue.

Sometimes, we need to transform a long method into a separate class when it deals with a number of intertwined variables that are hard to separate. In an extreme case, the whole method may be rewritten using a different algorithm.

Simplifying Method Calls

Several techniques in this group address the long-parameter and code-duplication smells, such as preserving the whole object, replacing parameters with explicit methods, and introducing parameter objects.

A factory method can be used to replace the general constructor method when we need more than basic initialization of fields in the constructor. In the old procedure programming, like using C, a method usually returns a special value to indicate an error. However, in modern programming, error handling is usually through exceptions. However, sometimes a simple test, instead of throwing an exception, is enough.

Organizing Data

The refactoring techniques in this group address data-handling issues. Properly using these techniques requires a clear understanding of the following concepts:

- **Data vs. objects**—When data is associated with certain operations, use an object instead.
- **Array vs. object**—An array contains save type of data, and an object can contain different types of data.
- **Value vs. reference**—Use value objects for infrequently changed data. Reference objects are easier to change.
- **Unidirectional vs. bidirectional**—When both classes need to be aware of each other, use bidirectional association. Otherwise, use unidirectional.
- **Single value vs. collection**—When dealing with collections, use *add()* or *remove()* methods instead of simple getter or setter methods.

Always use literal constants instead of magic numbers for better readability and easy modification. Use type code with caution. Instead, use subclasses or a separate state/strategy object. This is usually associated with the switch-statement smell.

Simplifying Conditional Expressions

Replace conditional with polymorphism" is often used with "replace the type code with subclass or state/strategy object" to get rid of switch-statement smell. Using specification inheritance and polymorphism makes the code more flexible to changes. For example, when adding a new type/case, we don't need to change all switch statements. Instead, a new subclass can be easily added without changing the existing code. Here, both the "program to interface" and "open-closed" principles are applied. There are also other techniques to simplify nested or duplicate conditions.

Moving Features between Objects

The refactoring techniques in this group address how to move functionality between classes and hide implementation details. The single-responsibility principle should be used to properly distribute functionality among different classes. The smells addressed by the techniques in this group are diverge change, shotgun surgery, message chains, and middle man.

Dealing with Generalization

The refactoring techniques in this group focus on proper use of inheritance. The fields or methods may be moved up to the superclass or down to the subclass. The key principles introduced in the previous module follow:

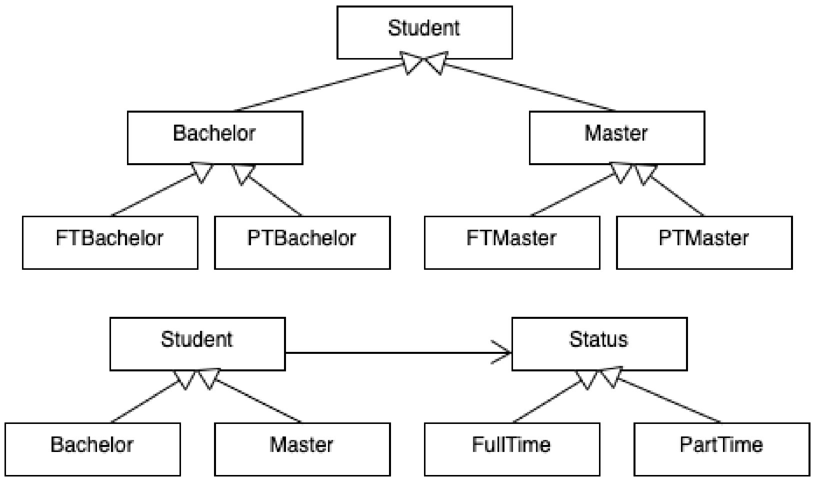
- Program to interface
- Favor composition over inheritance for implementation reuse
- Liskov substitution principle
- DRY

These principles can guide us to make proper choices between delegation and inheritance, between interfaces, superclasses and subclasses.

Big Refactoring

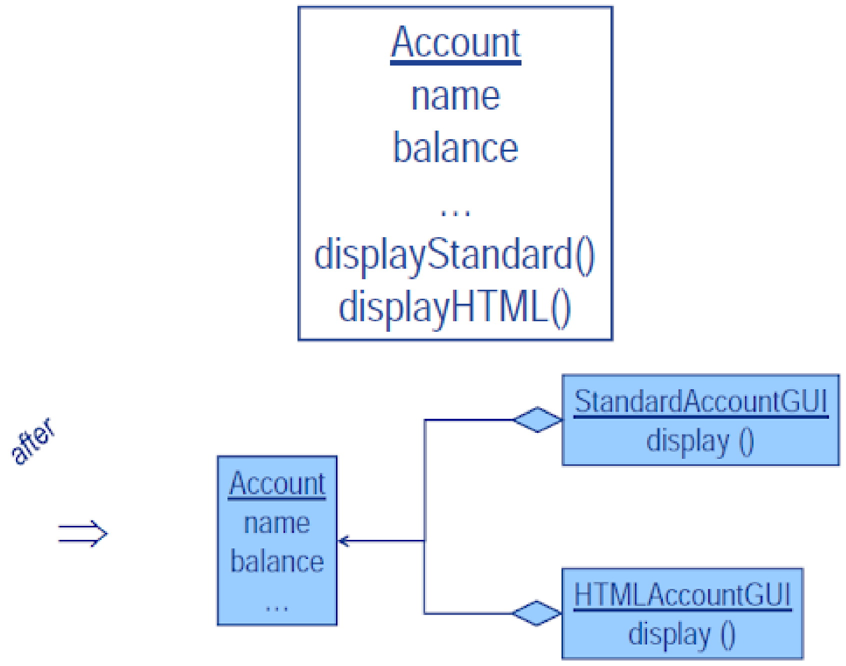
Big refactorings involve the whole team and set the direction for many changes. Four examples are listed in the book:

- **Tease apart inheritance**—Deal with the parallel inheritance hierarchies smell. The main idea is to decouple different jobs and create a multidimensional inheritance. A single Student class hierarchy can be refactored into two separate hierarchies, as shown below:



Tease Apart Inheritance before and after Class Diagrams

- **Extract hierarchy**—When a class is overly complex, we should refactor it into a group of subclasses.
- **Separate domain from presentation**—This is the key idea of MVC or MVWhatever. Always separate domain data and logic from the UI code. For example, we should extract the *displayStandard()* and *displayHTML()* from the *Account* class, as these methods are mainly for presentation purposes.



Separate Domain from Presentation before and after Class Diagrams

- **Convert procedural design to objects**—This big-refactoring technique converts the original procedural design to object-oriented design by introducing classes to procedural functions. This is applied sometimes to the legacy code.

Test Yourself

Test Yourself

Refactoring is to change ________ without changing ________.

the internal structure, the external behavior

Test Yourself

List at least three principles used in the refactoring techniques discussed in this module.

Test Yourself

It is always better to add more comments to code.

True

False

General feedback.

MATCH THE FOLLOWING TWO COLUMNS

By filling in the boxes below with their corresponding letter

1. Many changes are made to a single class.
2. Same switch structured code is scattered in different places in a program.
3. Different parts of the code contain identical groups of variables.
4. One class uses the internal fields and methods of another class unnecessarily.
5. The client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.
6. A class accesses the data from another class more than its own.
7. Always use primitive instead of creating a new class.
8. Need to create a subclass for multiple classes.
9. A class does nothing but delegating work to another class.
10. A class rarely does anything.
11. A single change is made to multiple classes simultaneously.

a. Primitive Obsession Surgery
b. Data Clumps
c. Switch Statements
d. Divergent Changes
e. Shotgun
f. Parallel Inheritance Hierarchies
g. Lazy Class
h. Feature Envy
i. Inappropriate Intimacy
j. Message Chains
k. Middle Man

Check Your Answers

Topic 2: Testing

Refactoring cannot be done without testing. Testing should be done as early as possible and as frequently as possible, at every reasonable opportunity. Testing cannot be separated from implementation; the two should be done side by side. Testing plays a much more critical role than documentation in model software development.

Regression Testing

Refactoring should be performed only on the working code. The code should continue to work after refactoring. One of the reasons that developers tend to avoid refactoring is that they are afraid of introducing new bugs or breaking the original working code. To guarantee that the code still works, it is critical to perform testing before and after refactoring. We need to rerun all tests that are related to changes and make sure that the software passes these tests after refactoring. This type of testing is called regression testing.

Regression testing is not only performed with refactoring, but also whenever there are changes in code. However, manually rerunning all tests constantly after refactoring would be very time consuming and costly. Automation is essential for regression testing to reduce the cost, as automated testing code can be run over and over again at no additional cost once created.

Why Testing, and What Is Testing?

Testing is the most popular quality-assurance activity. Testing validates that the system meets different functional and nonfunctional requirements, such as performance, reliability, security, and usability. It ensures that we are building the right system. Another important purpose of testing is to detect defects. Everyone can make mistakes. Testing can reveal mistakes by provoking failures in a planned way.

Professor Cem Kaner from FIT (Florida Institute of Technology) defines software testing as “an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test.”

Is Complete Testing Possible?

As an empirical, technical activity, testing cannot prove that the system in question is correct. It can only reveal the mistakes. As Dijkstra said, “Testing can only show the presence of bugs, not their absence.”

Even if no bugs are revealed through testing, it doesn’t guarantee that the system is free of bugs. We have to understand that it is impossible to completely test any nontrial module or system. The practical limitations of time and cost prohibit complete testing. Theoretically, we can prove that complete testing is impossible by converting it into a halting problem.

Since complete testing is not possible, the questions are, to what extent we should perform testing, and when should we stop testing? Testing is not free. The costs are both manpower and machine power to develop and execute tests. We need to balance the value of testing with its cost. Various principles and techniques have been proposed and developed to address this issue. For example, the “open to addition and close to modification” principle we introduced previously helps reduce the regression tests by minimizing the change. Tests of risky use cases and frequent use cases should have higher priority. Automated testing frameworks and tools are widely used to reduce the cost and make constant testing possible.

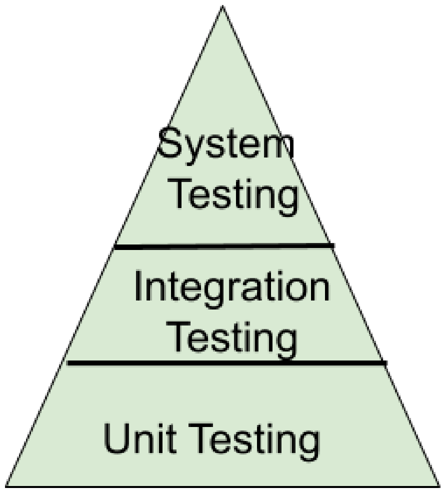
Types of Testing

There are many different testing techniques and various taxonomies to classify them.

Examples, which depend on the targeted testing components, follow:

- **Unit testing**—Testing individual components (units). A unit is usually a class or a method. Sometimes it can be a group of classes or methods. The goal is to confirm that the individual component is correctly coded and carries out the intended functionality. It is usually performed by developers and executed automatically.
- **Integration testing**—Testing groups of units, such as a subsystem or a collection of subsystems, and eventually the entire system. It is usually done in addition to unit testing, in order to test the interfaces among the subsystems and the interactions of the subsystems.
- **System testing**—Testing the entire system, including its integration with other software and hardware systems. The goal is to determine whether the system meets functional and nonfunctional requirements. Nonfunctional requirements such as performance and security cannot be tested at the lower level of integration. They need to be tested at the level of the whole system.

The following testing pyramid shows that most tests should be done at the unit level.



Testing Pyramid

That said, functional and nonfunctional testing are usually separated.

- **Functional testing**—Evaluating the system to determine whether the functions specified by the requirements are performed properly by the whole system.
- **Nonfunctional testing**—Evaluating the system to determine whether nonfunctional requirements are satisfied. Depending on what nonfunctional requirements are defined, this testing can take the following forms:
    - Performance testing
    - Volume testing
    - Scalability testing
    - Usability testing
    - Load testing
    - Stress testing
    - Compliance testing
    - Portability testing
    - Disaster-recovery testing

The above tests are usually run by developers or testers on the development environments. Customers perform additional testing, including the following:

- **Acceptance testing**—Performed by the customers to determine whether their requirements are met. This may involve executing typical transactions on-site on a trial basis. The goal is to demonstrate that the system meets the requirements and is ready to use. In Agile, every user story can only be accepted after passing the acceptance test.
- **Beta testing**—It is performed by "real users" of the software application in a "real environment" and can be considered a form of external user-acceptance testing. The beta version of the software is released to a limited number of end users of the product to obtain feedback on the product quality. Beta testing reduces product-failure risks and increases the quality of the product through customer validation.

Due to the separation of development and operation environments, tests that have already been passed in the development environment may fail in the operation environment. The new DevOps practices address this issue by unifying the development and operating environments.

Depending on whether testing can be performed repeatedly without human intervention, there are the following test types:

- **Automated testing**—Test cases are developed in code and can be automatically executed using certain tools.
- **Manual testing**—Test cases are developed in documents and performed manually by people.

Depending on whether test cases are based on the internal structure of the target system to be tested, there are the following test types:

- **White-box testing**—Testing the internal structure (or code) of the target system. Unit testing is also white-box testing.
- Black-box testing—Testing the external behavior of the system without (using) the knowledge of its internal structure.
- **Grey-box testing**—A combination of white- and black-box testing.

There are also other types of testing:

1. **API testing**—Testing application programming interfaces (APIs) directly and as part of integration testing.
2. **Installation testing**—Verifies whether the software system is successfully installed and working as expected.
3. **Smoke testing**—A subset of tests to ensure that the most important functions work and decide whether a build is stable enough to proceed through further testing.
4. **Release testing**—To verify that the software can be released.

Test Plan

Every developer agrees that testing is imperative. The question is how to conduct testing. Here are some questions to be answered:

- What to test?
- How to test?
- When to test?
- Who should test?
- On what date to test?
- How many resources are needed?

A test plan is a detailed document that addresses the above questions and guides one through the testing process. It should describe the strategy, objectives, schedule, estimation, deliverables, and resources required for testing.

1. **Determine the scope of testing: what to test and what not to test.**
   For example, what components of the systems should be tested? Should the platform and third-party libraries be tested? What features should be tested? Should nonfunctional requirements be tested?
2. **Determine what types of testing will be performed.**
   Usually, both unit and integration testing should be performed. Continuous integration is a common practice now, and continuous integration testing should be performed, as well. In addition, both functional and nonfunctional testing should be performed at the system level. API and UI testing are widely used, too. Acceptance testing by customers should always be performed in order to accept user stories. Regression testing should always be performed following changes. For unit testing, what constitutes a unit needs to be clearly defined.
3. **Decide who will test what and when, using what tools.**
   For example, is each individual developer responsible for unit-testing his or her own code? What types of testing should independent QA teams perform, and when? Do we need to involve any third parties?
   Usually, both the developers and additional testers will perform testing. The developers generally will write the test scripts and perform automated testing using white-box techniques, and the QA team will perform automated or manual testing using black-box techniques. Third-party testers may also be involved. As automated testing is essential in model software development, proper automated testing frameworks and tools should be used.
4. **Determine to what extent the testing should be performed.** Do not just “test until time expires.” Clearly define the testing objectives and criteria. Prioritize tests so that the most important ones are definitely performed.
   Since it is impossible to perform complete testing, the extent of testing should be clearly defined in advance. For example, should every user story, feature, class, and method be tested? A checklist may be used to ensure that all defined tests are covered, and the stopping criteria should be established.
5. **Determine how and where to get test input and oracle.** For example, we may need to use specific test data for certain domain-specific applications. To test a compiler, we may use well-known benchmark programs. We also need to determine the expected output (oracle). It may not be trivial. We may need to compute the output manually or use the output from other applications.
6. **Estimate resources, using historical data if available.** Resources include personnel—such as a QA engineer, a QA manager, and system admins—and hardware and software, such as the server, tools, infrastructure, and network. The cost will be mostly in person-hours.
7. **Identify the metrics to be collected.** Possible metrics include time, defect count, type, and source. Define the metrics and process to collect and analyze the data.
8. **Decide what documents will be produced, and develop the document templates.**
   There are three types of documents:

   1. Test-plan document.
   2. Test-cases specification. This may be in the code format, instead of in separate documents.
   3. Test reports. These include test results, errors and execution logs, and defect reports, as well as all of the resulting metrics.

   A test-case report should include the following information:

   - Test-case ID and name
   - Test items (what you test)
   - Test priority (high/medium/low)
   - Dependencies (on other test cases/requirements, if any)
   - Preconditions
   - Input data
   - Testing steps
   - Postconditions

- Expected output
- Actual output
- Pass or fail
- Bug ID/link (this may link to your GitHub issue ID)
- Additional notes

9. **Risk analysis is also important here.** Similar types of risks are involved in both the testing and the whole project, such as the following:
    - **Personnel risk**—No specific QA engineer.
    - **Technical risks**—Team members lack the required testing skills. Developers are not familiar with automated unit testing.
    - **Management risks**—Poor management skills; managers do not value testing enough.
    - **Wrong budget estimate and cost overruns**—No clear stopping criteria.

A Test Case and Test Code

No matter what types of testing or techniques you use, each test case should include several components:

- Setup or precondition
- Input data or action
- The oracle that can determine the expected output
- The real execution output, and whether it is the same as the expected output. If they are the same, the test has been passed. Otherwise, this test has been failed.

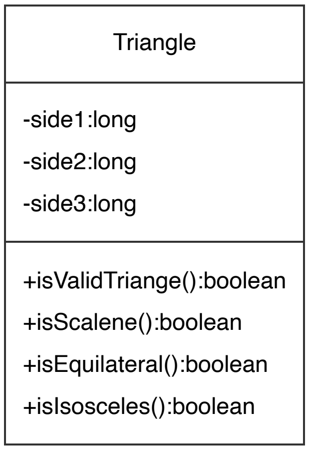
A test driver and/or a stub may be needed for a test case, particularly at the unit-testing level, to separate the test target from the rest of the system. A test driver simulates the part of the system that calls the component being tested. It passes the test inputs to the component and displays the results. A test stub simulates the component that is called by the tested component. The stub component may be a partial implementation of the original component, or it may simply return some fake values. However, it must provide the same interface as the simulated component and return a value compliant with the result type of the simulated one.

Most automated testing frameworks will provide APIs for drivers and stubs. There are many available testing frameworks, especially at the unit-testing level. Examples follow:

- JUnit for Java: the latest version is JUnit 5
- TestNG for Java
- unittest for python: a python version of JUnit
- NUnit for all .Net language
- Jasmine for JavaScript
- Mocha for Node.js
- Selenium: a portable software-testing framework for web applications

Example

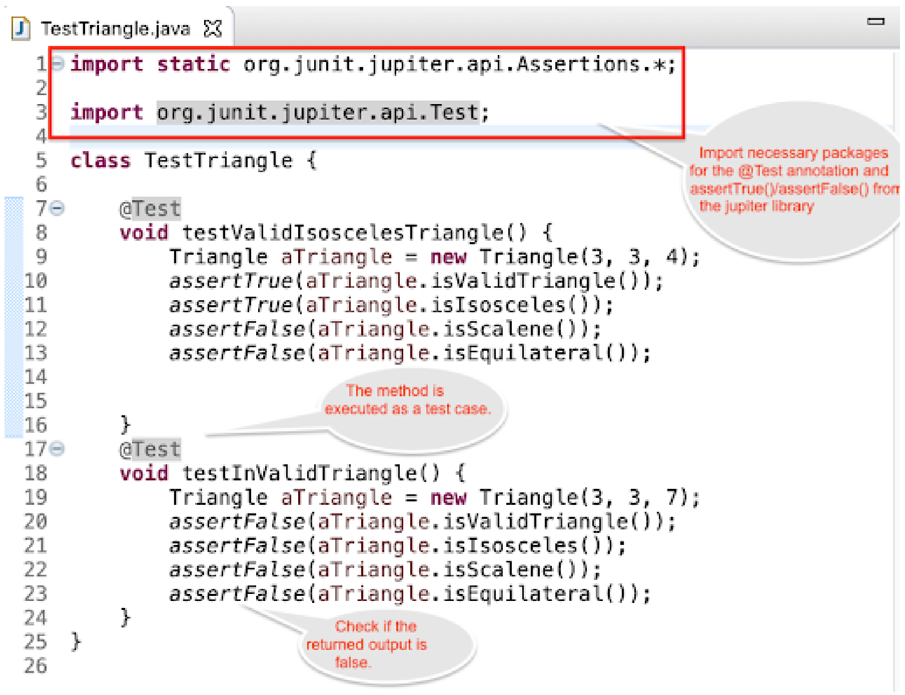
The triangle problem is a classic software-testing problem introduced in 1969. The program reads three numbers as the sides of a triangle and states whether the triangle is scalene, equilateral, or isosceles. So, suppose we have the following *Triangle* class:



Triangle
-side1:long -side2:long -side3:long
+isValidTriange():boolean +isScalene():boolean +isEquilateral():boolean +isIsosceles():boolean

Triangle Class Diagram

Here are two examples of unit test cases in Java:


TestTriangle.java
Import necessary packages for the @Test annotation and assertTrue()/assertFalse() from the jupiter library
The method is executed as a test case.
Check if the returned output is false.

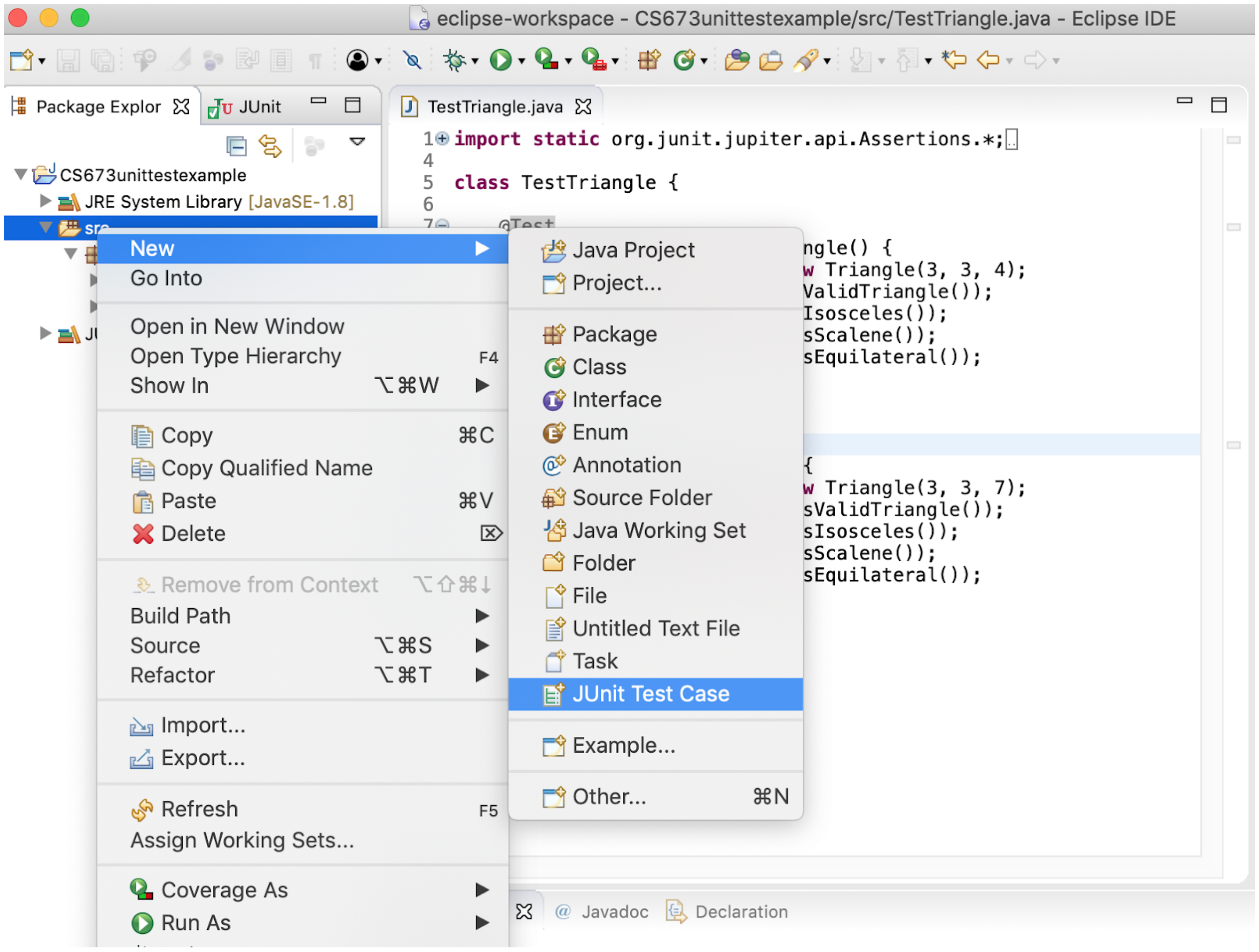
```java
import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestTriangle {

    @Test
    void testValidIsoscelesTriangle() {
        Triangle aTriangle = new Triangle(3, 3, 4);
        assertTrue(aTriangle.isValidTriangle());
        assertTrue(aTriangle.isIsosceles());
        assertFalse(aTriangle.isScalene());
        assertFalse(aTriangle.isEquilateral());


    }
    @Test
    void testInValidTriangle() {
        Triangle aTriangle = new Triangle(3, 3, 7);
        assertFalse(aTriangle.isValidTriangle());
        assertFalse(aTriangle.isIsosceles());
        assertFalse(aTriangle.isScalene());
        assertFalse(aTriangle.isEquilateral());
    }
}
```

TestTriangle.Java Screenshot

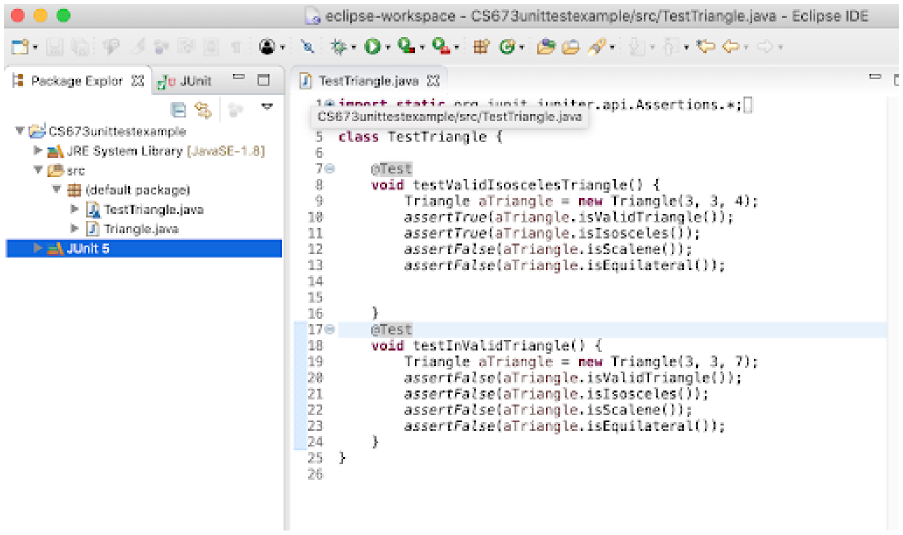
Here is the TestTriangle.java code. Can you implement the *Triangle* class and add more unit test cases?

Here, we use Eclipse as the IDE for this simple Java program. We use JUnit, the current version of which is JUnit 5. When you try to create a new source file, choose the JUnit Test Case and then New JUnit Jupiter Test. Eclipse will create a Java file with JUnit 5 added into your project path, and proper import.

JUnit uses @Test annotation to indicate that the public void method to which it is attached can be executed as a test case. In the method, we simply create a triangle with three sides and call these methods to check what type of triangle it is. The input data here are the three sides of the triangle. The output is a boolean value. The *assertTrue()* or *assertFalse()* methods simply compare the real output with the expected output.


eclipse-workspace - CS673unittestexample/src/TestTriangle.java - Eclipse IDE

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

class TestTriangle {

    @Test
                          ngle() {
                          w Triangle(3, 3, 4);
                          ValidTriangle());
                          Isosceles());
                          sScalene());
                          sEquilateral());

                          {
                          w Triangle(3, 3, 7);
                          sValidTriangle());
                          sIsosceles());
                          sScalene());
                          sEquilateral());
```


eclipse-workspace - CS673unittestexample/src/TestTriangle.java - Eclipse IDE

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

class TestTriangle {

    @Test
    void testValidIsoscelesTriangle() {
        Triangle aTriangle = new Triangle(3, 3, 4);
        assertTrue(aTriangle.isValidTriangle());
        assertTrue(aTriangle.isIsosceles());
        assertFalse(aTriangle.isScalene());
        assertFalse(aTriangle.isEquilateral());


    }
    @Test
    void testInValidTriangle() {
        Triangle aTriangle = new Triangle(3, 3, 7);
        assertFalse(aTriangle.isValidTriangle());
        assertFalse(aTriangle.isIsosceles());
        assertFalse(aTriangle.isScalene());
        assertFalse(aTriangle.isEquilateral());
    }
}
```

Create Unit Test Cases in Eclipse Screenshots

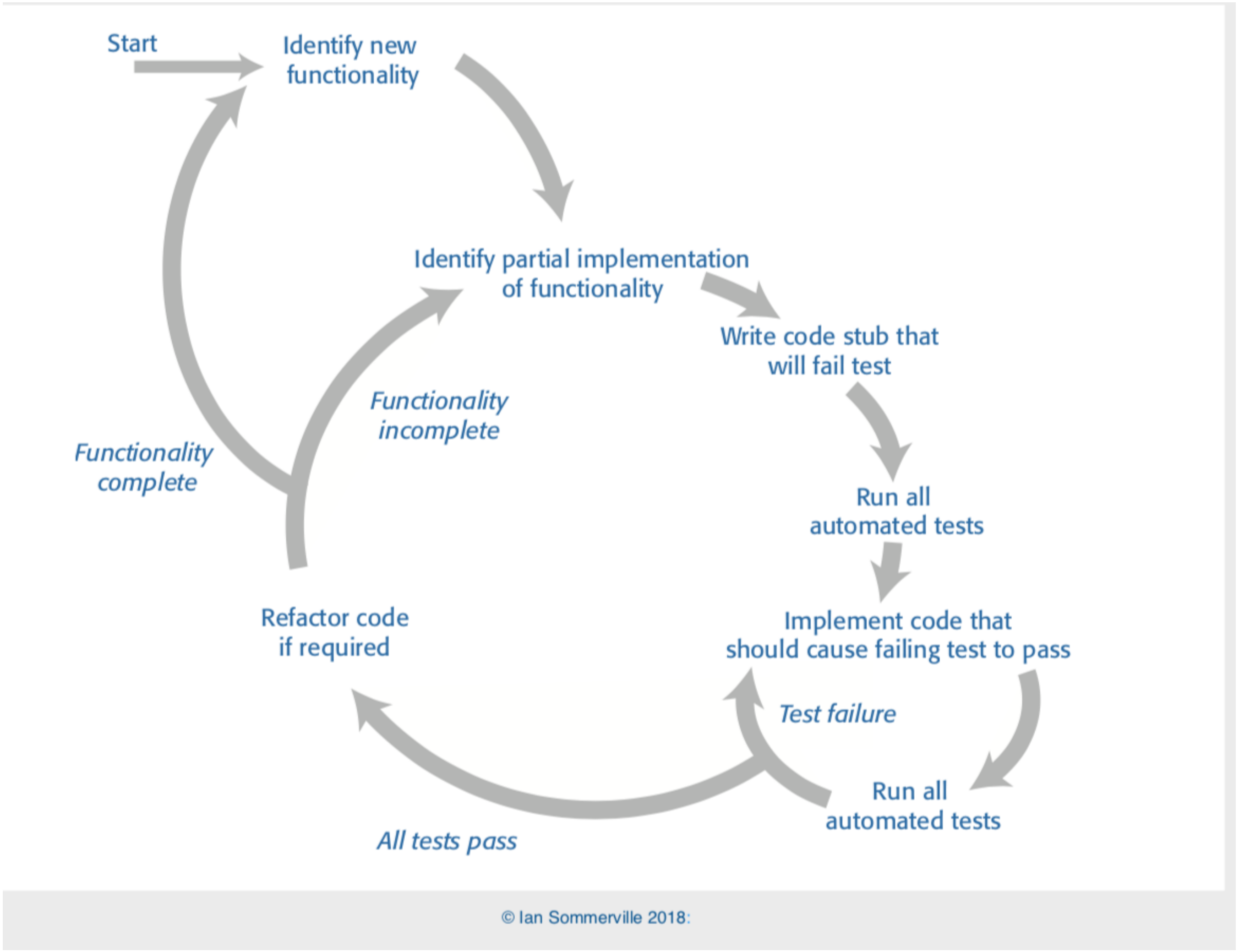
Test-Driven Development

In the traditional, sequential software-development process, testing is performed after the software is implemented. However, in modern, iterative software-development processes such as Agile, testing should be considered from the very beginning and in all activities. For example, acceptance tests should be developed along with user stories when performing requirement analysis in each iteration. Test plans should be developed before the system is implemented, and test cases should also be developed before or while the system is implemented. Test cases should be executed while the system is implemented and whenever there are changes.

In Agile, developing test cases (or writing test code) is part of the design and implementation process. Test-driven development was initially started in XP (eXtreme Programming) and evolved over time. The main idea is to test before implementation. Before writing any production code, we write the test code. Actually, the design idea is reflected in the test code before the production code. Instead of using the traditional “design -> implement -> test” sequence, TDD promotes the “test -> implement -> test” sequence, with a red-green-refactor work flow:

- **Red**—Understand the requirements and create automated test cases that define the expected output. The new test will fail because the requirements haven't been implemented.
- **Green**—Write code (experiment/prototype/build) to pass the test. The code is only designed to pass the test.
- Refactor—Clean up the code, remove duplicates (DRY: do not repeat yourself), and change the code structure without affecting the existing functionality. Make the code more readable, maintainable, etc. Rerun all tests, and the code shall pass all.
- After refactoring, always perform regression testing.

The following diagram shows the iterative process with the “red -> green -> refactor” work flow (Sommerville, 2018):


Start
Identify new functionality
Identify partial implementation of functionality
Write code stub that will fail test
Run all automated tests
Implement code that should cause failing test to pass
Test failure
Run all automated tests
All tests pass
Refactor code if required
Functionality incomplete
Functionality complete
© Ian Sommerville 2018:

Agile Testing Work Flow

Uncle Bob proposed three laws about TDD (Martin, 2008):

- You may not write production code until you have written a failing unit test.
- You may not write more of a unit test than is sufficient to fail, and not compiling is failing
- You may not write more production code than is sufficient to pass the current failing test.

Javier Saldana refactored the above three laws into two points:

- Write only enough of a unit test to fail.
- Write only enough production code to make the failing unit test pass.

Basically, the idea of TDD is to write tests first and need no extra code. Refactoring is critical to TDD, too. Here, you can find a very introductory video by James Shore on TDD:

Why TDD?

Strictly following TDD, a cycle may only take 30 seconds, and we can write dozens of tests every day. Gradually, we may have tests that cover all of our production code, and there may be more testing code than production code itself. More testing code will encourage more refactoring as you don’t need to be afraid that the changes will break the system. This can improve software quality by generating cleaner code and better design. Consequently, it decreases debugging time and development cost even though additional time is needed to write the test.

Here is a great talk presented in GopherCon SG 2017: “TDD for those who don't need it”

However, successful TDD requires keeping the tests clean. Test code is as important as the production code. As Uncle Bob said, having dirty tests is equivalent to, if not worse than, having no tests. As the production code evolves, the tests may need to change, too. Maintaining test suites is as important as maintaining the production code. Test code should be simple, succinct, and expressive, though it may not need to be as efficient as production code, since it will not be executed in the production environment. Test code should also follow the single-responsibility principle to be more maintainable. Uncle Bob proposed the "FIRST" criteria for clean tests:

- **Fast**—Run quickly
- **Independent**—Not dependent on each other. Otherwise, a failure can cause a cascade of downstream failures, making diagnosis difficult and hiding downstream defects.
- **Repeatable in any environment**—Tests should not just be able to run on your own machine.
- **Self validating**—The result should be either pass or fail, so that it is easier to
- **Timely**—Written in a timely fashion, just before the production code.

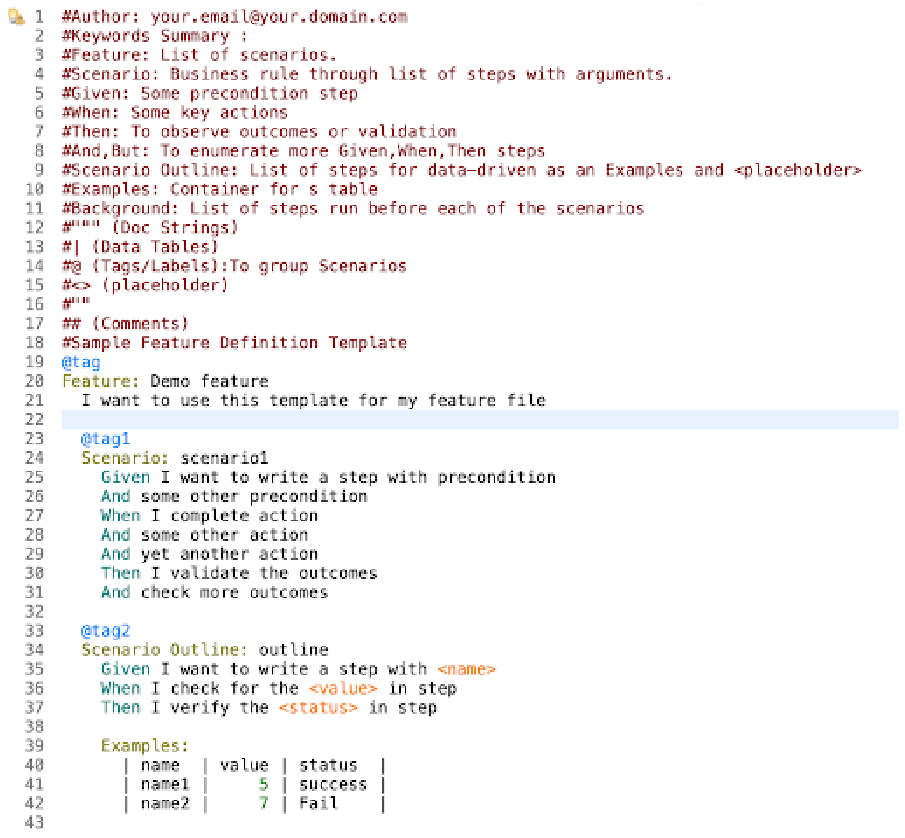
Without clean tests, new tests are hard to add or change. Without test code, the developers cannot perform refactoring constantly, due to fear of adding new bugs. Consequently, the code becomes messier and produces more bugs. With clean test code and extensive test coverage, you can improve the architecture and design through refactoring without fear.

As the development starts with test code, TDD can help you develop more testable code. Testability is also an important software quality. It is defined as the degree of ease with which a software system exposes its faults through execution-based testing. High testability is desired in a software system, as it allows the developers to find and fix bugs faster.

BDD

BDD (behavior-driven development) is another idea that pushes TDD to a high level. TDD is mostly at the implementation-code level, and the testing code is mostly unit test code. BDD also emphasizes the "test first" idea, but focuses on testing the actual behavior of the system, from the end user's perspective. David Heinemeier Hansson, the author of *Ruby on Rail*, posted the article "TDD Is Dead. Long Live Testing." to argue the issue of TDD. With constant refactoring of the production code, unit-test code is also subject to change. For example, if we move a method from one class to another, the original unit tests are not usable anymore and need to be changed. This can greatly increase the cost and frustration. Kent Beck and Martin Fowler, advocates of TDD, responded to David and had a great discussion with him. They emphasized some basic principles and relaxed some logistics of how TDD should be executed. Anyway, BDD seems to be a good solution. Some argue that BDD provides the best of test-driven development to help create better software with self-documenting, executable tests that bring users and developers together with a common language.

Cucumber is a tool that supports BDD. It was originally developed for Ruby, but now supports most mainstream programming languages, including Java and JavaScript. It enables the running of automated acceptance tests written in a logical language, Gherkin, that is close to natural language and can be understood by customers. Here is an example of a Cucumber test template:



```
#Author: your.email@your.domain.com
#Keywords Summary :
#Feature: List of scenarios.
#Scenario: Business rule through list of steps with arguments.
#Given: Some precondition step
#When: Some key actions
#Then: To observe outcomes or validation
#And,But: To enumerate more Given,When,Then steps
#Scenario Outline: List of steps for data-driven as an Examples and <placeholder>
#Examples: Container for s table
#Background: List of steps run before each of the scenarios
#""" (Doc Strings)
#| (Data Tables)
#@ (Tags/Labels):To group Scenarios
#<> (placeholder)
#""
## (Comments)
#Sample Feature Definition Template
@tag
Feature: Demo feature
  I want to use this template for my feature file

  @tag1
  Scenario: scenario1
    Given I want to write a step with precondition
    And some other precondition
    When I complete action
    And some other action
    And yet another action
    Then I validate the outcomes
    And check more outcomes

  @tag2
  Scenario Outline: outline
    Given I want to write a step with <name>
    When I check for the <value> in step
    Then I verify the <status> in step

    Examples:
      | name  | value | status  |
      | name1 |     5 | success |
      | name2 |     7 | Fail    |
```

BDD Script Screenshot

Test Yourself

_______ testing should always be performed with refactoring

Regression

Test Yourself

Complete the triangle-problem production code and test code using TDD.

Test Yourself

The TDD workflow is ___, ___ and ____.

red, **green**, and refactoring

Test Yourself

What are similarities and differences between TDD and BDD in your own words?

Test Yourself

If an application passed all test cases, then it is bug free.

MATCH THE FOLLOWING TWO COLUMNS

By filling in the boxes below with their corresponding letter

1. Testing individual components.	a. Performance Testing
2. Testing groups of components.	b. Beta Testing
3. Testing the functional requirements (features) of the system.	c. Functional Testing
4. Testing the system performance such as throughput.	d. Smoke Testing
5. Tested by the customers to determine if requirements are met and if the user story can be accepted.	e. Unit Testing
6. Tested by a small subset of real customers in a “real environment.”	f. Acceptance Testing
7. To verify if the software system is successfully installed and it is working as expected after installation.	g. Installation Testing
8. A subset of tests to ensure that the most important functions work, in order to decide if a build is stable enough to proceed with further testing.	h. Integration Testing

Check Your Answers

Test Yourself: Refactoring

Here are some examples from Fowler's refactoring book. Please try to refactor using the given techniques:

Before Refactoring

//Please Extract the printDetails method from the following method:

```
void printOwing(double amount) {
            printBanner();

            //print details
            System.out.println ("name:" + _name);
            System.out.println ("amount" + amount);
}
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please inline the method moreThanFiveLateDeliveries() into getRating()

```
int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
        return _numberOfLateDeliveries > 5;
}
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please inline temp var basePrice

```
double basePrice = anOrder.basePrice();
       return (basePrice > 1000)
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please replace Temp with Query (extract the expression into a method.)

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please introduce explaining variable

//(Put the result of expression into a temp variable)

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
   (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 ) {
         // do something
}
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please Split temp variable

//make a separate temp var for each assignment

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Remove assignments to parameters

//use a temp var instead of assign to a parameter

//confusing on passing by value

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    ...
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Replace method with method object:

//class Order...

```
double price() {
    double primaryBasePrice;
    double secondaryBasePrice;
    double tertiaryBasePrice;
    //long computation;
    ...
}
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please substitute algorithm (with a better one)

```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            return "Don";
        }
        if (people[i].equals ("John")){
            return "John";
        }
        if (people[i].equals ("Kent")){
            return "Kent";
        }
    }
    return "";
    }
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please introduce foreign method

//Create a method in the client class with an instance of the server class as its first argument.

```
 Date newStart = new Date (previousEnd.getYear(),
        previousEnd.getMonth(), previousEnd.getDate() +
1);
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please replace this array with an object

```
 String[] row = new String[3];
   row [0] = "Liverpool";
   row [1] = "15";
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Please change the unidirectional association to bidirectional

//add reference to order in Customer class

```
    class Order {
     Customer getCustomer() {
         return _customer;
     }
     void setCustomer (Customer arg) {
         _customer = arg;
     }
     Customer _customer;
    ...
    }
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Encapsulate Collection

//Change set method to remove & add method

```
class Course...
{
    public Course (String name, boolean isAdvanced) {...};
    public boolean isAdvanced() {...};
    ...
}

class Person {
    public Set getCourses() {
        return _courses;
    }
    public void setCourses(Set arg) {
        _courses = arg;
                   }
    private Set _courses;
...
}
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//remove control flag, replace it with break or return

```
void checkSecurity(String[] people) {
       boolean found = false;
       for (int i = 0; i < people.length; i++) {
           if (!found) {
              if (people[i].equals ("Don")){
                sendAlert();
                found = true;
              }
              if (people[i].equals ("John")){
                sendAlert();
                found = true;
              }
           }
       }
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//Replace nested conditional with guard clauses

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
            if (_isRetired) result = retiredAmount();
            else result = normalPayAmount();
        };
    }
  return result;
  }
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Before Refactoring

//change the error code with exception

```
 int withdraw(int amount) {
     if (amount > _balance)
```

```
            return -1;
        else {
            _balance -= amount;
            return 0;
        }
    }
```

Use the text box to type in the text above, refactor using the given technique, and then click the arrow to check your answer.

▶ Click to see code after refactoring

Please identify possible smells in the following code snippets and refactor them.

```
class UserValidator {
        private Cryptographer cryptographer;

        public boolean checkPassword(String userName, String password) {
                User user = UserGateway.findByName(userName);
                if (user != User.NULL) {
                        String codePhrase = user.getPhraseEncodedByPassword();
                        String phrase =
cryptographer.decrypt(codePhrase,password);
                        if ("Valid.password".equals(phrase)) {
                                Session.initialize();
                                return true;
                        }
                }
                return false;
        }
}
```

Use the text box to type in the text above, identify code smells, and refactor.

```
public class Customer {
        private Vector _rentals = new Vector();
        public Customer(String name) {
        _name = name;
    }
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + name() + "\n";
        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            //determine amounts for each line
            switch (each.tape().movie().priceCode()) {
                case Movie.REGULAR:
                    thisAmount += 2;
                    if(each.daysRented() > 2)
                        thisAmount += (each.daysRented() - 2) * 1.5;
                    break;
                case Movie.NEW_RELEASE:
                    thisAmount += each.daysRented() * 3;
                    break;
                case Movie.CHILDRENS:
                    thisAmount += 1.5;
                    if (each.daysRented() > 3)
                        thisAmount += (each.daysRented() - 3) * 1.5;
                    break;

            }
            totalAmount += thisAmount;

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.tape().movie().priceCode() == Movie.NEW_RELEASE) && each.daysRented() > 1)
                frequentRenterPoints ++;
            //show figures for this rental
            result += "\t" + each.tape().movie().name()+ "\t" + String.valueOf(thisAmount) + "\n";
        }
        //add footer lines
        result +=  "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) + " frequent renter points";
        return result;

    }
```

Use the text box to type in the text above, identify code smells, and refactor.

Conclusion

In this module, we have introduced several techniques and practices used in implementation and testing activities. In particular, we addressed how to write clean code by identifying the bad code smells and applying refactoring techniques. Testing is the key to ensuring that the code implements the right functionality, and is critical to ensuring that refactoring can be applied. We introduced some basic concepts about testing, and different types of testing. Finally, we discussed TDD and BDD techniques and tools used in modern application development.

Bibliography

- McConnell, S. (2004) *Code complete: A practical handbook of software construction* (2nd ed.).
- Martin, R. C. (2008). *Clean code: A handbook of Agile software craftsmanship by (Uncle Bob).*
- Fowler, M. (1999). *Refactoring: Improving the design of existing code.*

Boston University Metropolitan College