CZ3002 - Advanced Software Engineering

Software Testing: Test Driven Development, Test then Code, Junit and Ant

Faculty	:	Dr Althea Liang
School	:	School of Computer Science and Engineering
Email	:	qhliang@ntu.edu.sg
Office	:	N4-02c-107

Introduction to this Lesson

At the end of the lesson, you should be able to:

- Explain the **problems** faced for testing
- Explain the **emergence** of Test Driven Development (TDD)
- Describe the Test Driven Development (TDD) **life cycle, benefits and process**
- Apply the technique of test then code with JUnit
- Use Ant to automate all tasks

Introduction to this Lesson: Examples Preview

Example	Slides
C	A Comparison
JU	Test then code with Junit

Software Engineers' Impressions on Testing

"Demanding that Testers locate every fault: *This is totally a Developer's job, because developers have the needed skills. Testers generally don't have these skills, though at times, they may have useful hints..*"

-- Jerry Weinberg

"So we get beautiful code by some people and junk by others, and the junk will kill the program. Just kill it. And so they run into all these problems, months of testing. Well, their product -- they never could get it to work.."

-- Watts Humphrey

"*Testing is an infinite process of comparing the invisible to the ambiguous in order to avoid the unthinkable happening to the anonymous.*"

-- James Bach

The problem faced for testing

- Every programmer knows they should write tests for their code. A few do.
- The universal response to "Why not?" is "I'm in too much of a hurry."
- This quickly becomes a vicious cycle.
  - The **more pressure** you feel, the **fewer tests** you write.
  - The fewer tests you write, the **less productive** you are and the **less stable** your code becomes.
  - The less productive and accurate you are, the **more pressure** you feel.

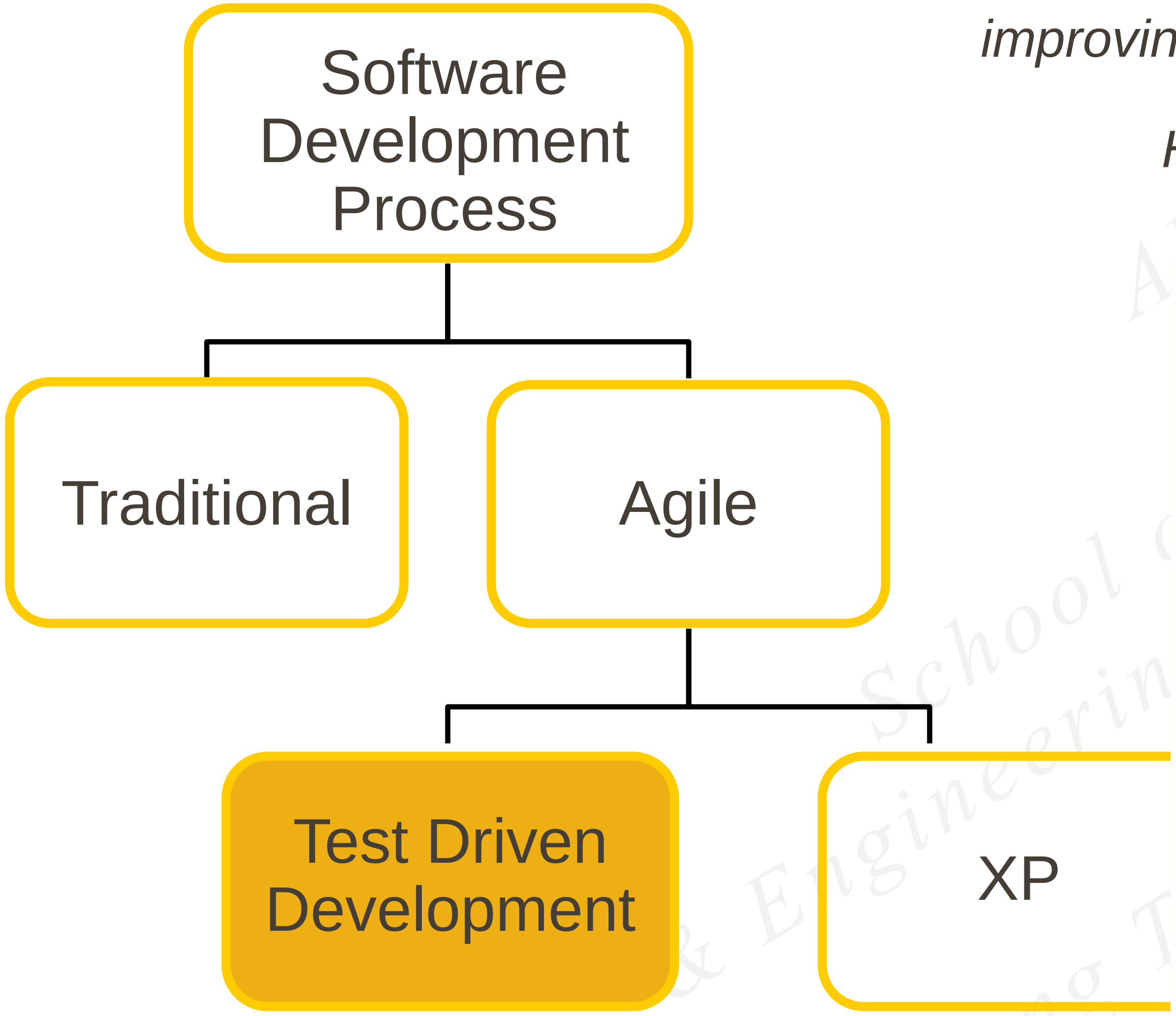
Emergence of Test then Code

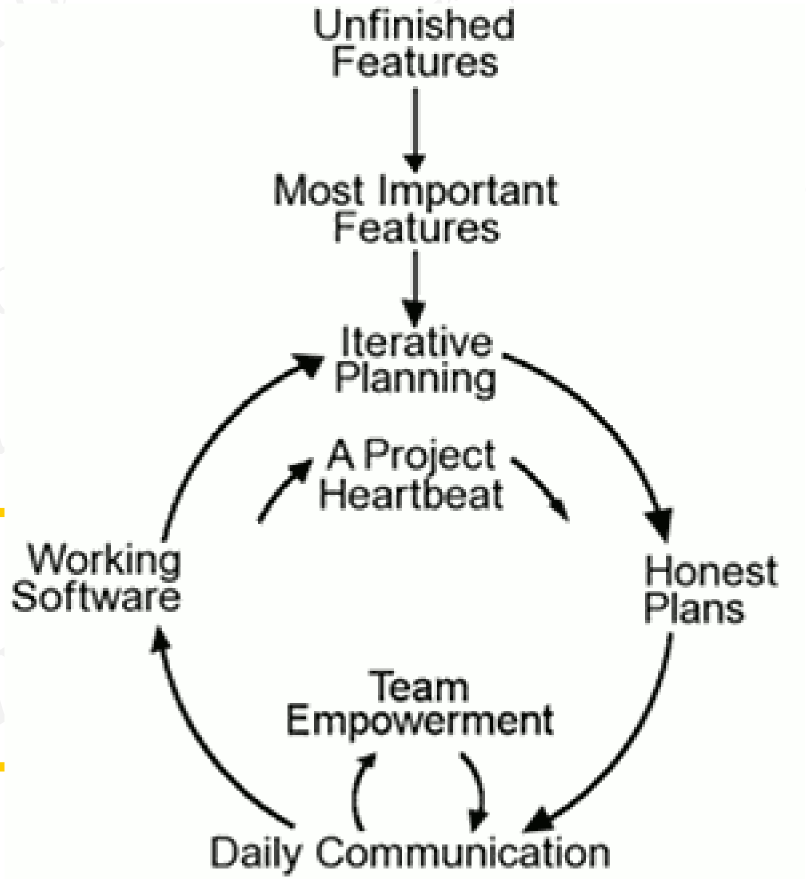
- Historically, the software development cycle places the **testing** stage **after** the **coding** stage.
- Yet, it is possible to write **black box tests** of each requirement **before** writing the code that will meet that requirement.
- There are several advantages to this strategy.
  - The suite (list) of tests becomes something that can be **run automatically** as the program is developed.
  - These tests are run every time the program is **compiled** to ensure that new additions to the code do not break earlier functionality.

Can Software Testing Play a Bigger Role?

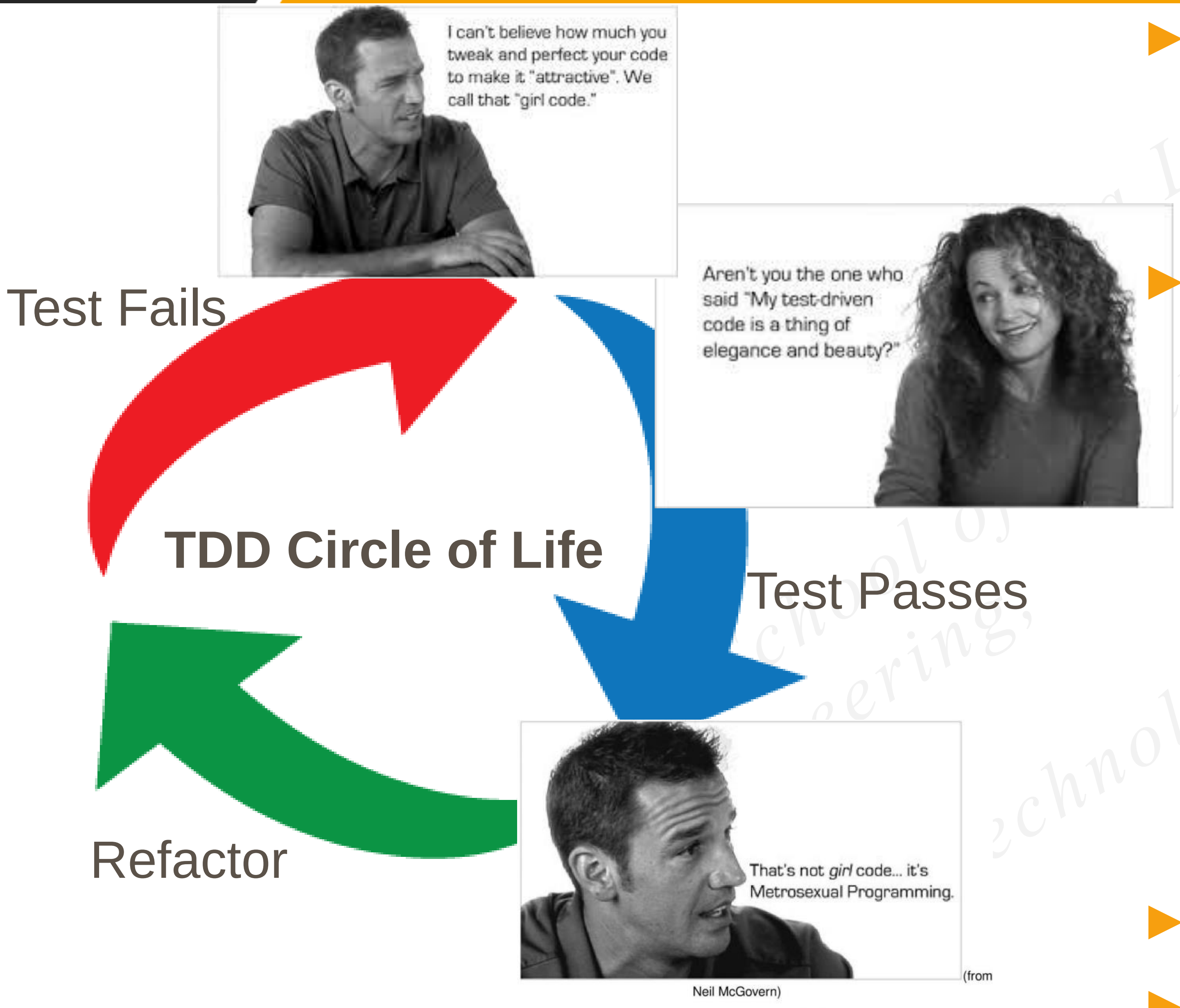
FROM PREVIOUS LESSON*: Evaluating product quality and for improving it, by identifying defects and problems.*

HOW ABOUT A BIGGER ROLE?





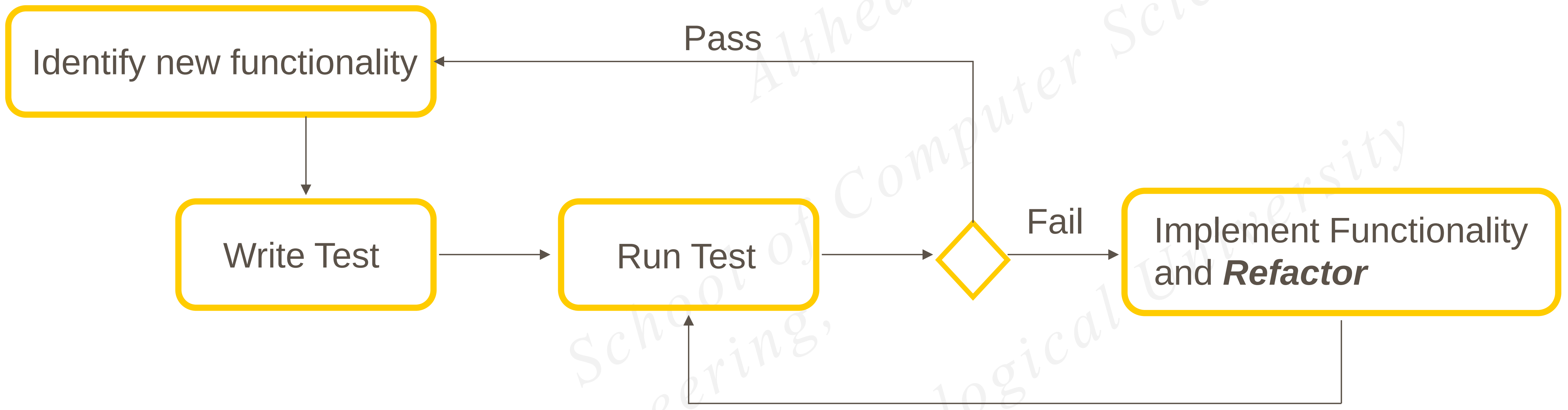
Test Driven Development (TDD) and Circle of Life


I can't believe how much you tweak and perfect your code to make it "attractive". We call that "girl code."
Aren't you the one who said "My test-driven code is a thing of elegance and beauty?"
That's not *girl* code... it's Metrosexual Programming.
Test Fails
TDD Circle of Life
Test Passes
Refactor
ETH
Chair of Software Engineering

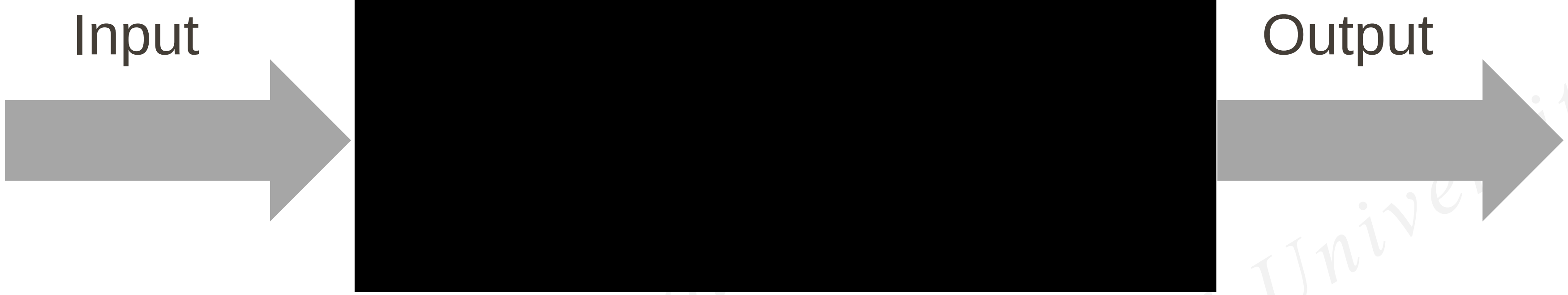
(from Neil McGovern)

- Differentiating TDD
  - SDLC and length
  - Automating
- Development
  - Function selection
  - Test Case
  - Requirement
  - Validation
  - Phase 1: fails-> simple writing
  - Phase 2: passes-> reviewing
  - Phase 3: refactor ->completes
- Maintenance
- Driving

What Exactly is TDD (Kent Beck)?


Identify new functionality
Pass
Write Test
Run Test
Fail
Implement Functionality and ***Refactor***

What is the Suitable Testing Technique?



Black Box

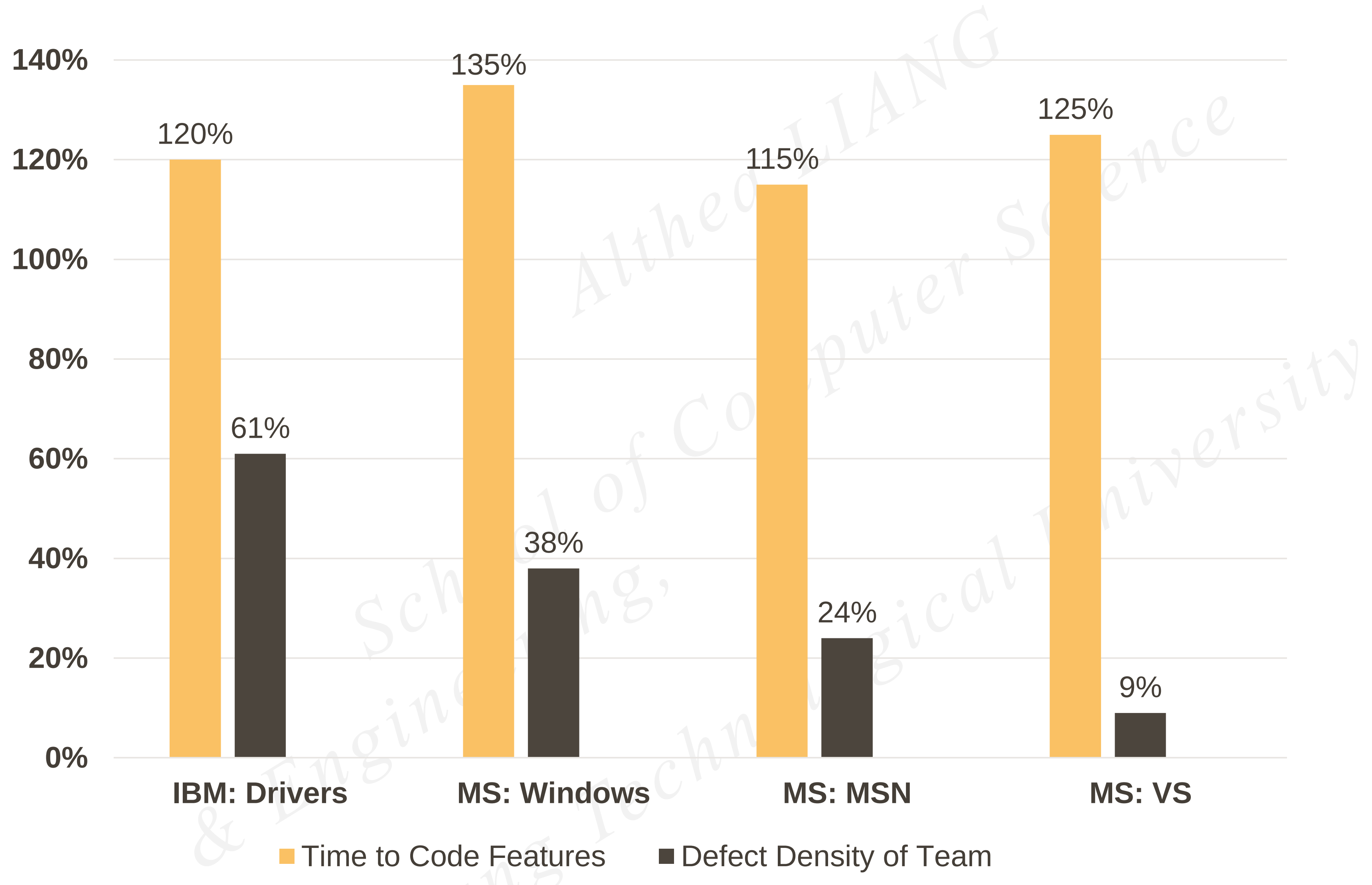
Bringing in Big Benefits

- Every programmer knows they should write tests for their code. Few do. The universal response to "Why not?" is "I'm in too much of a hurry." This quickly becomes a vicious cycle.

  Break the vicious cycle

  - Never leave testing **behind** because of a hurry to success or a hurry to get it done?
- The more **pressure** you feel, the fewer tests you write.
  - Less pressure for **smaller** things?
- The fewer tests you write, the less productive you are and the less stable you code becomes.
  - **Effective** test cases to prove?
- The less productive and accurate you are, the more pressure you feel.
  - **Improve** them!

A Comparison (Example C)



***From:** Benefit From Unit Testing in THE REAL WORLD!*

Test First Guideline

- The tests should drive you to write the code, the reason you write code is to get a test to succeed, and you should only write the minimal code to do so.

```
[illegible] item = el->FirstChildElement[illegible]
GroupScene::ElementDesc elDesc;
std::string sp_name = item->Attribute( [illegible]
std::string spritename = item->Attribute( [illegible]
float x = boost::lexical_cast<float>( [illegible]
float y = boost::lexical_cast<float>( [illegible]
float offset = boost::lexical_cast<float>( [illegible]
unsigned layer = 50; // default
if ( item->Attribute( "layer" ) [illegible]
{
    layer = boost::lexical_cast<unsigned>( [illegible]
}
elDesc.name_ = sp_name;
elDesc.spriteName_ = spritename;
[illegible] = x;
```

JUnit

- JUnit is a test **framework** for **implementing testing** in Java. It provides a simple way to explicitly test **specific areas** of a Java program.
  - Link: http://www.junit.org/index.htm
- It is **extensible** and can be employed to test a **hierarchy of program code**, either singularly or as multiple units.
- It also integrates with **Ant** for automating the test and **others**.
  - http://jakarta.apache.org/ant/.

Why use a test framework?

- Using a testing **framework** is beneficial because it **forces** you to **explicitly declare** the expected **results** of specific **program** execution routes.
- When **debugging** it is possible to write a test which expresses **the result you are trying to achieve** (for the debugging purpose) and then debug until the test comes out positive.
- By having a set of tests that test all the core components of the project it is possible to **modify specific areas** of the project and immediately see the effect the modifications have on the **other areas** by the results of the test, hence, **side-effects** can be quickly realised.

How to Use: Test then code with Junit

- JUnit promotes the idea of first testing then coding, in that it is possible to setup test data for a unit which defines what the expected output is and then code until the tests pass.
- It is believed by some that this practice of "test a little, code a little, test a little, code a little..." increases programmer **productivity** and **stability** of program code whilst **reducing defect density**, **programmer stress** and the **time** spent debugging. Literally, the vicious cycle is gone.

Ant and Test Environment

- Use **Ant** to automate **test** and **others**
- Install **Ant**
  - Download .zip package of Ant from http://ant.apache.org
  - Extract the .zip file to a directory, say, C:\Ant
  - Append "{Ant_install_dir}\bin" to "Path" environment variable. ("C:\Ant\apache-ant-1.6.2\bin" in this example)
  - Add a new environment variable named "ANT_HOME", and set its value to be the directory the Ant was installed. ("C:\Ant" in this example)

Automate All Tasks Using Ant

- Invoke following command from command line: Ant
- In order to use Ant to automate all the tasks, you need to write a XML-format **buildfile** (always named as "build.xml") including what tasks to do, and the order of the task execution.
- An example of buildfile is as follows (for more information about the meaning of each tag. Refer to online manual of Ant http://ant.apache.org/manual/index.html, especially the "Using Ant", "Running Ant", and "Ant Tasks" sections).

Automate All Tasks Using Ant – 1

```
<project name="example" default="run" basedir=".">

  <property name="src" location="."/>
  <property name="build" location="build"/>
  <property name="report" location="report"/>

  <target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${report}"/>
  </target>

  <target name="compile" depends="init">
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="run" depends="compile, test">
    <java classname="example">
       <classpath>
          <pathelement path="${build}"/>
```

Automate All Tasks Using Ant - 2

```
      </classpath>
    </java>
  </target>

  <target name="test" depends="compile">
    <junit printsummary="yes" fork="yes" haltonfailure="yes">
      <formatter type="plain"/>
      <classpath>
        <pathelement path="${build}"/>
      </classpath>

      <test name="testexample" todir="${report}"/>
    </junit>
  </target>

  <target name="clean">
    <delete dir="${build}"/>
    <delete dir="${report}" />
  </target>
</project>
```

Summary

Now, you should be able to:

- Explain the problems faced for testing
- Explain the emergence of test then code process
- Describe the Test Driven Development (TDD) circle of life process
- Apply the technique of test then code with JUnit
- Use Ant to automate all tasks

NANYANG TECHNOLOGICAL UNIVERSITY

Special Thanks to Kydon during the TEL Efforts of the Lecture

End of Test Driven Development, Test then Code, Junit and Ant (Becoming Driven by Tests)

Faculty	:	Dr Althea Liang
School	:	School of Computer Science and Engineering
Email	:	qhliang@ntu.edu.sg
Office	:	N4-02c-107