# UECS2363 SOFTWARE CONSTRUCTION AND CONFIGURATION

# CHAPTER 5 : Software Testing, Unit Testing and Test-Driven Development

DR FARIZUWANA AKMA
farizuwana@utar.edu.my

# CHAPTER 5
# Software Testing, Unit Testing and Test-Driven Development (PART I)

# Introduction

- "If you look at how most programmers spend their time, you'll find that writing code is actually a small fraction.
- Some time is spent figuring out what ought to be going on, some time is spent designing, but most time is spent debugging.
- I am sure every reader can remember long hours of debugging, often long into the night. Every programmer can tell a story of a bug that took a whole day (or more) to find.
- Fixing the bug is pretty quick, but finding it is a nightmare.
- And then when you do fix a bug, there's always a chance that another one will appear and that you might not even notice it until much later. Then you spent ages finding that bug."
- -- Refactoring, Martin Fowler

# Feedback

- Very important element in many activities.
- It tells us whether our actions are having the right effect.
- The soon we get feedback, the more quickly we can react.
- **Testing** is all about getting feedback on software.

# Feedback

- Having feedback in development process give us confidence in the software that we write.
- It lets us work more quickly and with less paranoia.
- It lets us focus on the new functionality we are adding by having the tests tell us whenever we break old functionality.

# Testing

- It comes from the world of quality assurance.
- Traditionally, testing occurs after the software is complete.
- Hence, it is a way of measuring quality — not a way of building quality into the product.

# Testing

- In many organizations, testing is done by someone other than the software developers.
- The feedback provided by this kind of testing is very valuable, but it comes so late in the development cycle that its value is greatly diminished.
- So what kind of testing should software developers do to get feedback earlier?

# Developer Testing

- Software developer rarely believes he can write code that works “first time, every time.”
- Developers do testing, too.
- Some do their testing the same way as testers do it: by testing the whole system as a single entity.
- Most, prefer to test their software unit by unit.

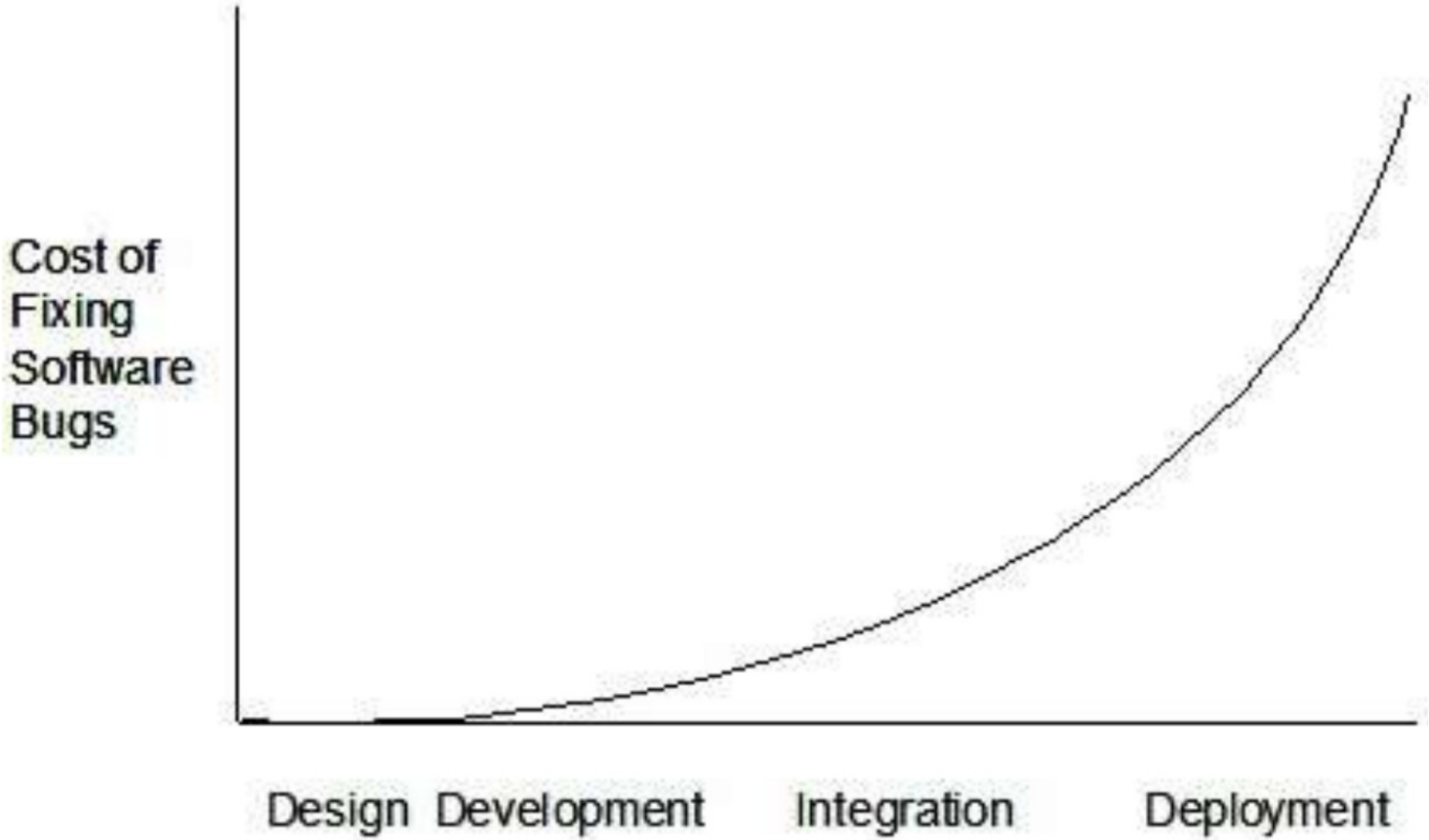
# Developer Testing

- The “units” may be larger-grained components or they may be individual classes, methods, or functions.
- The key thing that distinguishes these tests from the ones that the testers write is that the units being tested are a consequence of the design of the software, rather than being a direct translation of the requirements.

# Automated Test

- A process which covers:
  - the use of software to control the execution of tests
  - the comparison of actual outcomes to predicted outcomes
  - the setting up of test preconditions
  - other test control and test reporting functions.

# Why testing is critical?



# Why testing is critical?

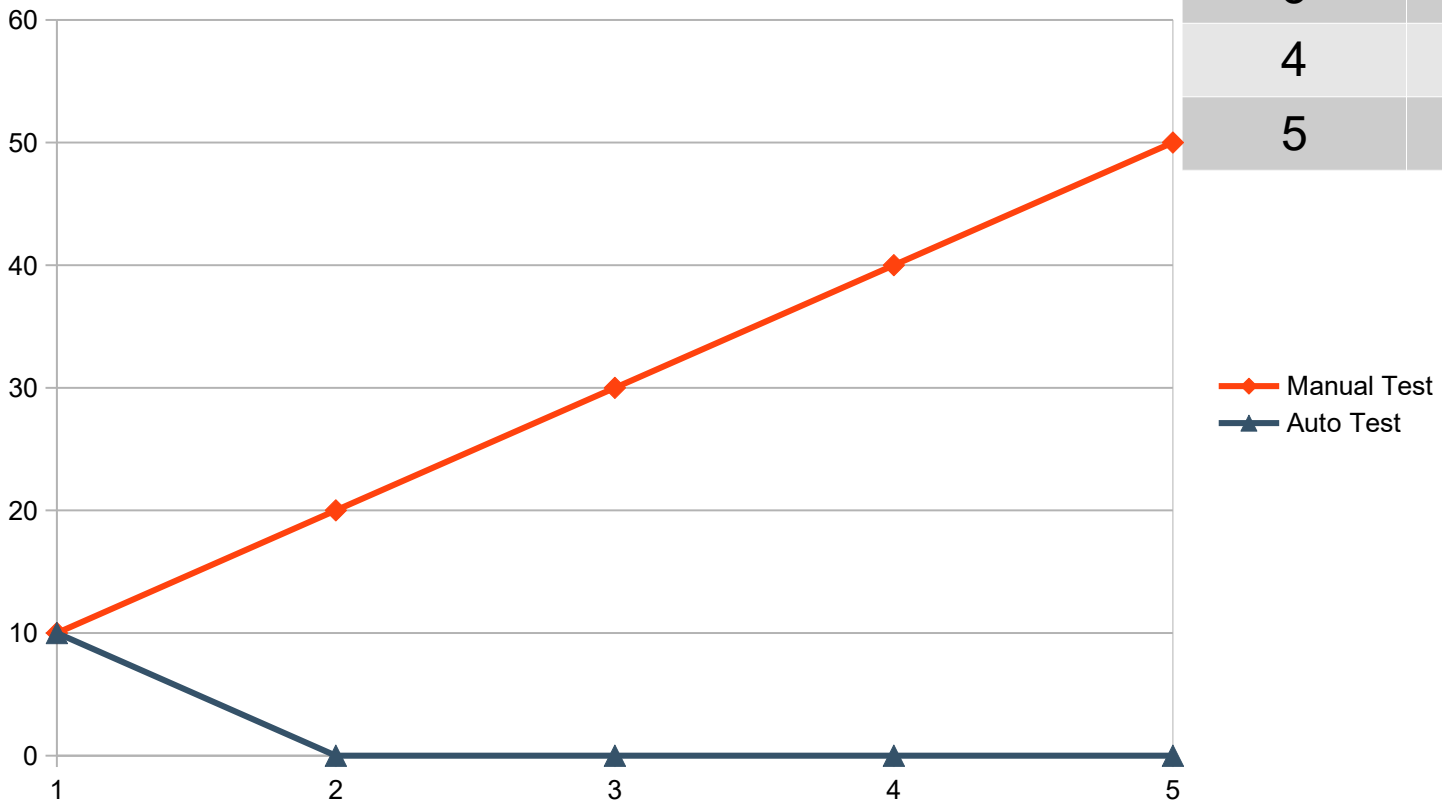
- A time savings translates directly into cost savings
- Improves testing productivity
- Improves accuracy
- Increases test coverage
- Does what manual testing cannot do.

# Effort and Cost

- Let’s assume 6 test cases
- Effort required to run all 6 manually => 10 min
- Effort required to write unit tests for all 6 cases => 10 min
- Effort required to run unit tests for all 6 cases => < 1 min
- Number of testing iterations => 5
- Total manual testing time => 50 min
- Total unit testing time => 10 min

# Effort and Cost

| Release | Manual Test | Auto Test | Manual Test Cumulative |
|---|---|---|---|
| 1 | 10 | 10 | 10 |
| 2 | 10 | 0 | 20 |
| 3 | 10 | 0 | 30 |
| 4 | 10 | 0 | 40 |
| 5 | 10 | 0 | 50 |



# Automated vs. Manual testing

| Automatic Testing | Manual Testing |
|---|---|
| Need to run a set of tests repeatedly | Test cases have to be run a small number of times |
| Helps performing "compatibility testing" (on different configurations and platforms) | Allows the tester to perform more specific tests |
| Long term costs are reduced | Short term testing costs are reduced |
| Possible to run regressions on a code that is continuously changing and in shorter time | The more time tester spends testing a module the grater chance to find real bugs |
| It's more expensive to automate (bigger initial investments) | Manual tests can be very time consuming |
| You cannot automate everything, some tests still have to be done manually | For every release you must rerun the same set of tests which can be tiresome |

# Common types of automated testing methods

- Monkey testing
- Capture / playback
- Code-based (Unit) testing
- Intelligent test automation

# Monkey testing

- Randomly selecting inputs from a large range of values and monitoring if exceptions are thrown.
- For example, a monkey test can enter random strings into text boxes to ensure handling of all possible user input.
- It applies not only for GUI or WEB testing, but also for Unit testing.

# Tools example

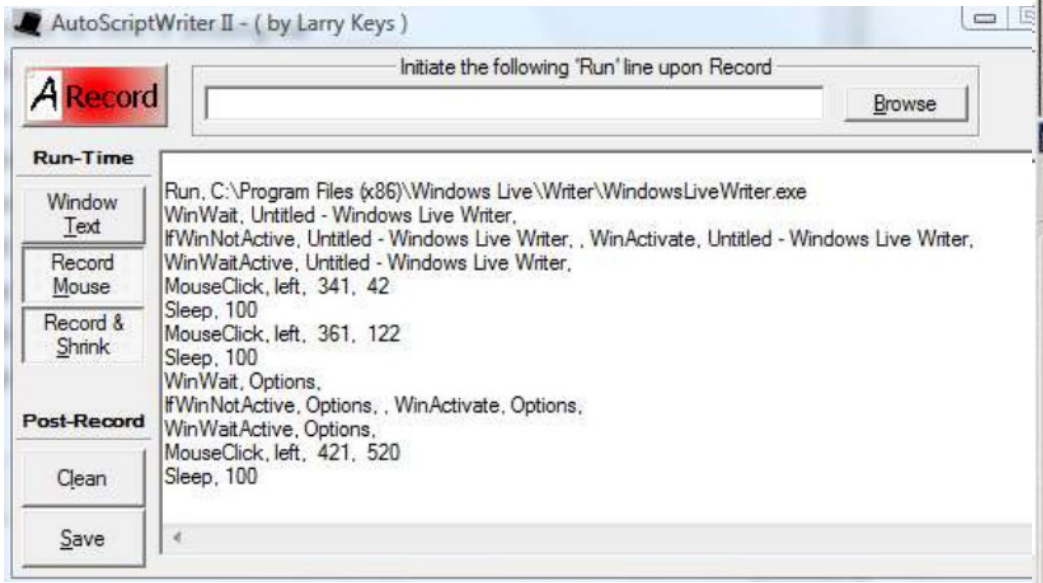
- Monkey – Android testing tools.
- The simplest way to use the monkey is with the following command, which will launch the application and send 500 pseudo-random events to it.
- `$ adb shell monkey -v -p your.package.name 500`

# Capture / playback

- It’s a set of software programs that capture user inputs and stores it into a script for later replay.
  - + Repeated testing can be performed quickly.
  - + Does not require programming skills.
  - - When the GUI changes, input sequences previously recorded may no longer be valid.
  - - Difficult to determine location of bugs.

# Tools example

- **AutoIt, Autohotkey - Free keyboard macro program. Supports hotkeys for keyboard, mouse.**




# Code-based (Unit) testing

- Individual units of source code are tested to determine if they fit for use.
- Ideally, each test case is independent from the others

✓ + Bugs can be found in early development stage

✓ + Easy to test boundary cases

× - Not effective for the integrated system testing

# How do you ensure your code runs correctly?

- `printf() or writeline()`
- `assert()`
- `void analyze(char *string, int length)`
- `{`
- `assert(string != NULL); /* cannot be NULL */`
- `assert(*string != '\0'); /* cannot be empty */`
- `assert(length > 0); /* must be positive */`
- `}`

# Test Code

```
[TestMethod]
public void Withdraw_ValidAmount_ChangesBalance()
{
    // arrange
    double currentBalance = 10.0;
    double withdrawal = 1.0;
    double expected = 9.0;
    var account = new CheckingAccount("JohnDoe", currentBalance);
    // act
    account.Withdraw(withdrawal);
    double actual = account.Balance;
    // assert
    Assert.AreEqual(expected, actual);
}

[TestMethod]
[ExpectedException(typeof(ArgumentException))]
public void Withdraw_AmountMoreThanBalance_Throws()
{
    // arrange
    var account = new CheckingAccount("John Doe", 10.0);
    // act
    account.Withdraw(1.0);
    // assert is handled by the ExcpectedException
}
```

# Unit Testing

- Unit Testing is the idea of writing addition code, called test code, for the purpose of testing the smallest blocks of product code to the full extent possible to make sure they are error free.

# Characteristics of a Good Unit Test

- Runs **fast**, runs **fast**, runs **fast**. If the tests are slow, they will not be run often.
- **Separates** or simulates environmental dependencies such as databases, file systems, networks, queues, and so on. Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.

# Characteristics of a Good Unit Test

- Is very **limited in scope**. If the test fails, it's obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It's important to only test one thing in a single test.
- **Runs and passes in isolation**. If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.

# Characteristics of a Good Unit Test

- Often uses stubs and mock objects. If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
- Clearly reveals its intention. Another developer can look at the test and understand what is expected of the production code.

# CHAPTER 5
# Software Testing, Unit Testing and Test-Driven Development (PART II)

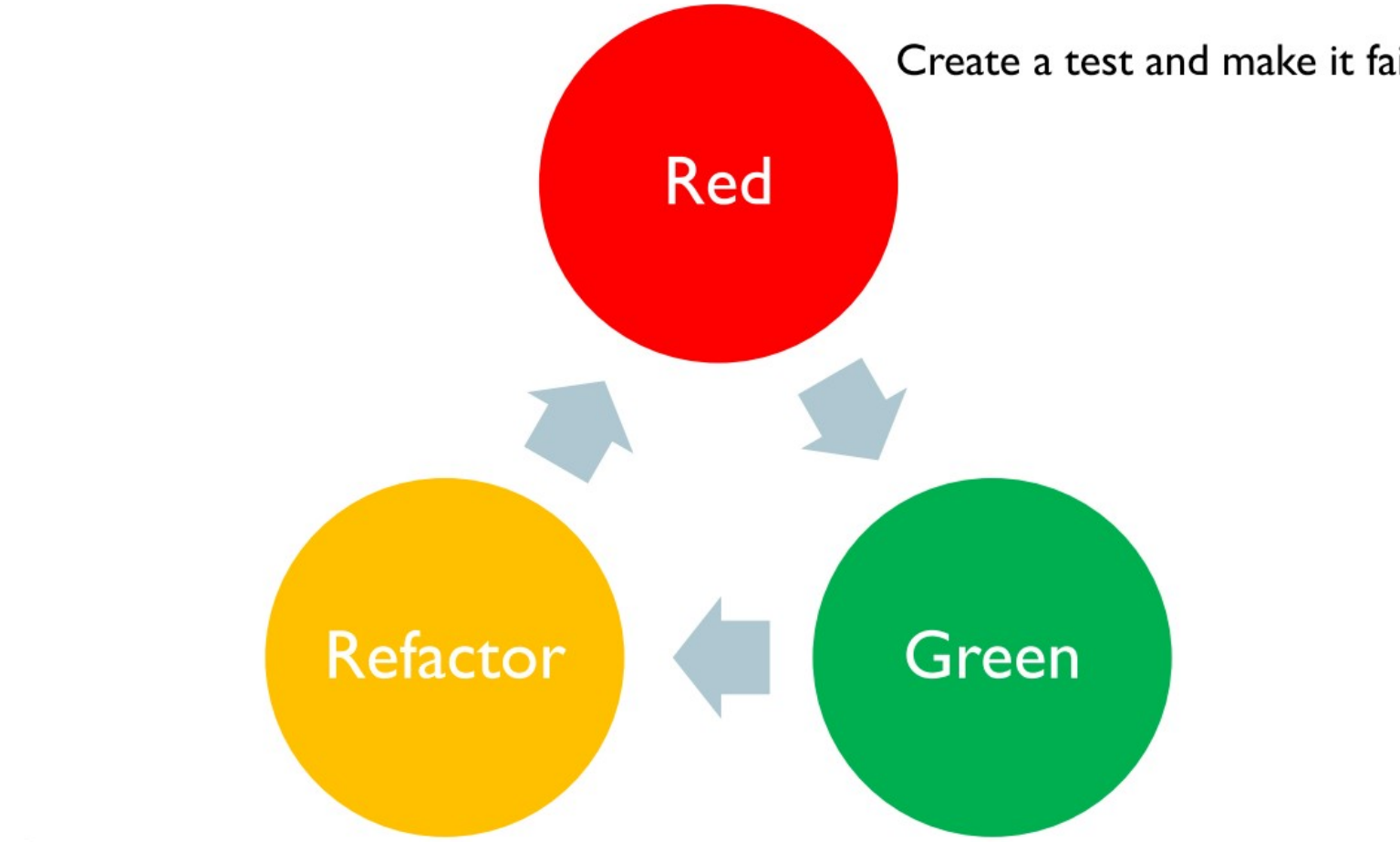
# Test Driven Development (TDD)

- TDD is an advanced technique of using automated unit tests to drive the design of software and force decoupling of dependencies.
- The result of using this practice is a comprehensive suite of unit tests that can be run at any time to provide feedback that the software is still working.

# The Three laws of TDD

- 1st Law:
- You may not write production code until you have written a failing unit test
- 2nd Law:
- You may not write more of a unit test than is sufficient to fail, and not compiling is failing.
- 3rd Law:
- You may not write more production code than is sufficient to pass the currently failing test.

# TDD Process Cycle



# Benefits of TDD

- The suite of unit tests provides constant feedback that each component is still working.
- The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
- When the test passes and the production code is refactored to remove duplication, it is clear that the code is finished, and the developer can move on to a new test.

# Benefits of TDD

- **Test-driven development forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.**
- **The software tends to be better designed, that is, loosely coupled and easily maintainable, because the developer is free to make design decisions and refactor at any time with confidence that the software is still working. This confidence is gained by running the tests. The need for a design pattern may emerge, and the code can be changed at that time.**

# Benefits of TDD

- The test suite acts as a regression safety net on bugs: If a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.
- Reduced debugging time!

# END OF LECTURE 08