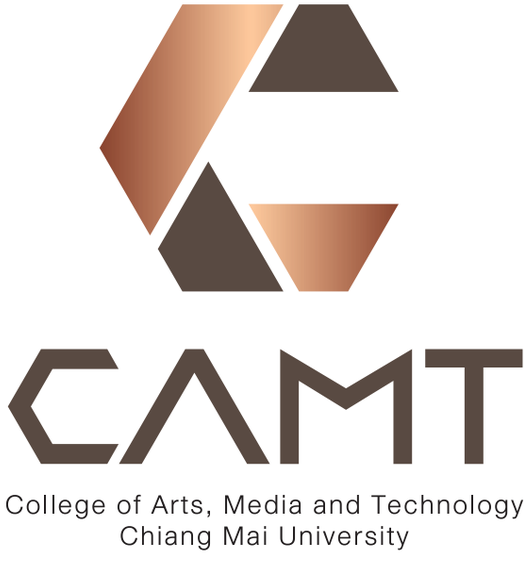
SE 233 Advanced Programming

Chapter V Unit test



Lect Passakorn Phannachitta, D.Eng.

passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

Agenda

- Basic software testing
- Unit test
- Test-driven development

Why are bugs inevitable

- As software gets more features and supports more platforms it becomes increasingly difficult to make it created bug-free.
- However, we can reduce the chance — Effective testing.

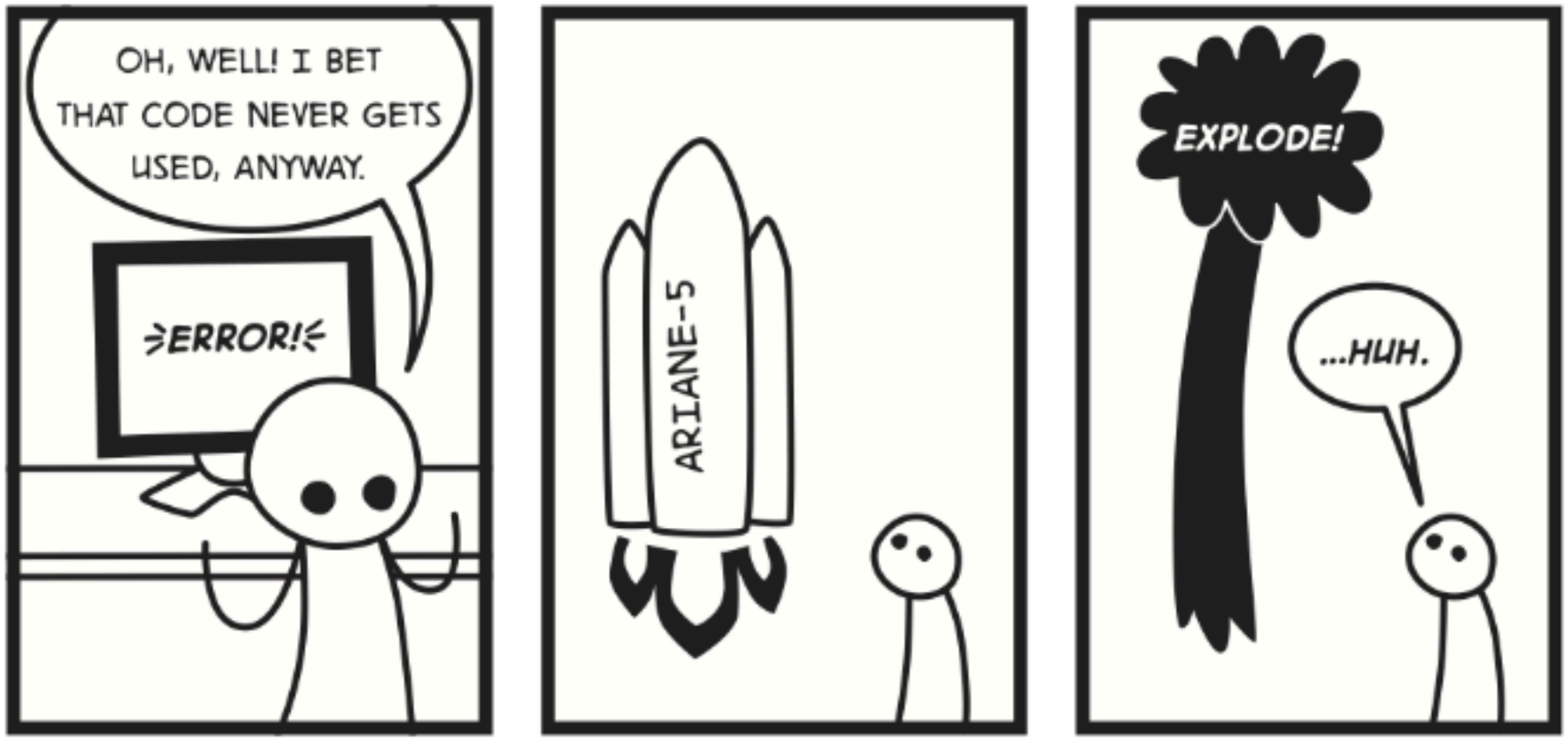
A software bug occurs when

- The software **does not do** something that the specification says **it should do**.
- The software **does something** that the specification says **it should not do**.
- The software **does something** that the specification **does not mention**.
- The software **does not do something** that the product specification **does not mention but <u>should</u>**.

Basic software testing

- Software testing is defined as an **empirical technical investigation** conducted to provide stakeholders with information about the product's quality or service under test.
  - **Empirical** - derived from experiment, experience, and observation.
  - **Technical** - having special skill or practical knowledge.
  - **Investigation** - a detailed inquiry or systematic examination.

Why test?


OH, WELL! I BET
THAT CODE NEVER GETS
USED, ANYWAY.
ERROR!
ARIANE-5
EXPLODE!
...HUH.

Example cases

- A spacecraft named Ariane 5 was exploded at its launch due to an integer overflow problem

Example cases

- An F-18 crashed because of a missing exception condition: if … then … without the else clause that was thought could not possibly arise.

Example cases

- Five nuclear power plants in the united states were temporary shut down in because of a fault in the simulation program used to design a nuclear reactor to withstand earthquakes
  - The cause of the problem was the use of an arithmetic **sum** of a set of numbers instead of their **absolute** values.

Other example cases

- A Norwegian bank ATM consistently dispersed 10 times the amount required.
- A software flaw caused another UK bank to duplicate every transfer payment request for half an hour.

Other example cases

- The unintended and uncontrolled acceleration cases of Toyota

Other example cases

- The unintended and uncontrolled acceleration cases of Toyota

More on: https://catless.ncl.ac.uk/risks/

Common software problems

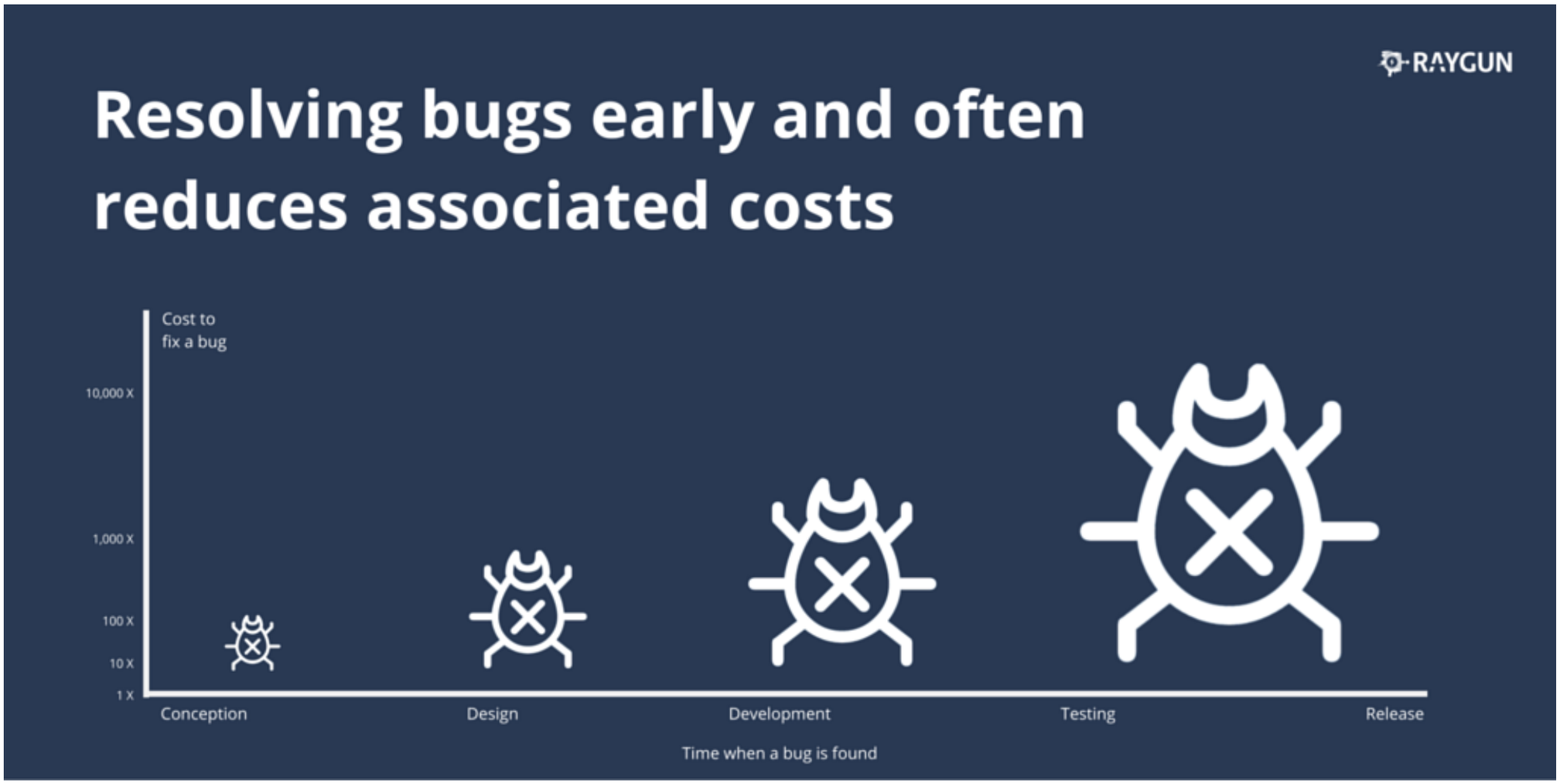
- Incorrect calculation
- Incorrect data edits & ineffective data edits
- Incorrect matching and merging of data
- Data searches that yields incorrect results
- Incorrect processing of data relationship
- Incorrect coding / implementation of business rules
- Inadequate software performance

Common software problems (cont)

- Confusing or misleading data
- Software usability by end users & Obsolete Software
- Inconsistent processing
- Unreliable results or performance
- Inadequate support of business needs
- Incorrect or inadequate interfaces with other systems
- Inadequate performance and security controls
- Incorrect file handling

Bugs found later cost more to fix

- In other words:



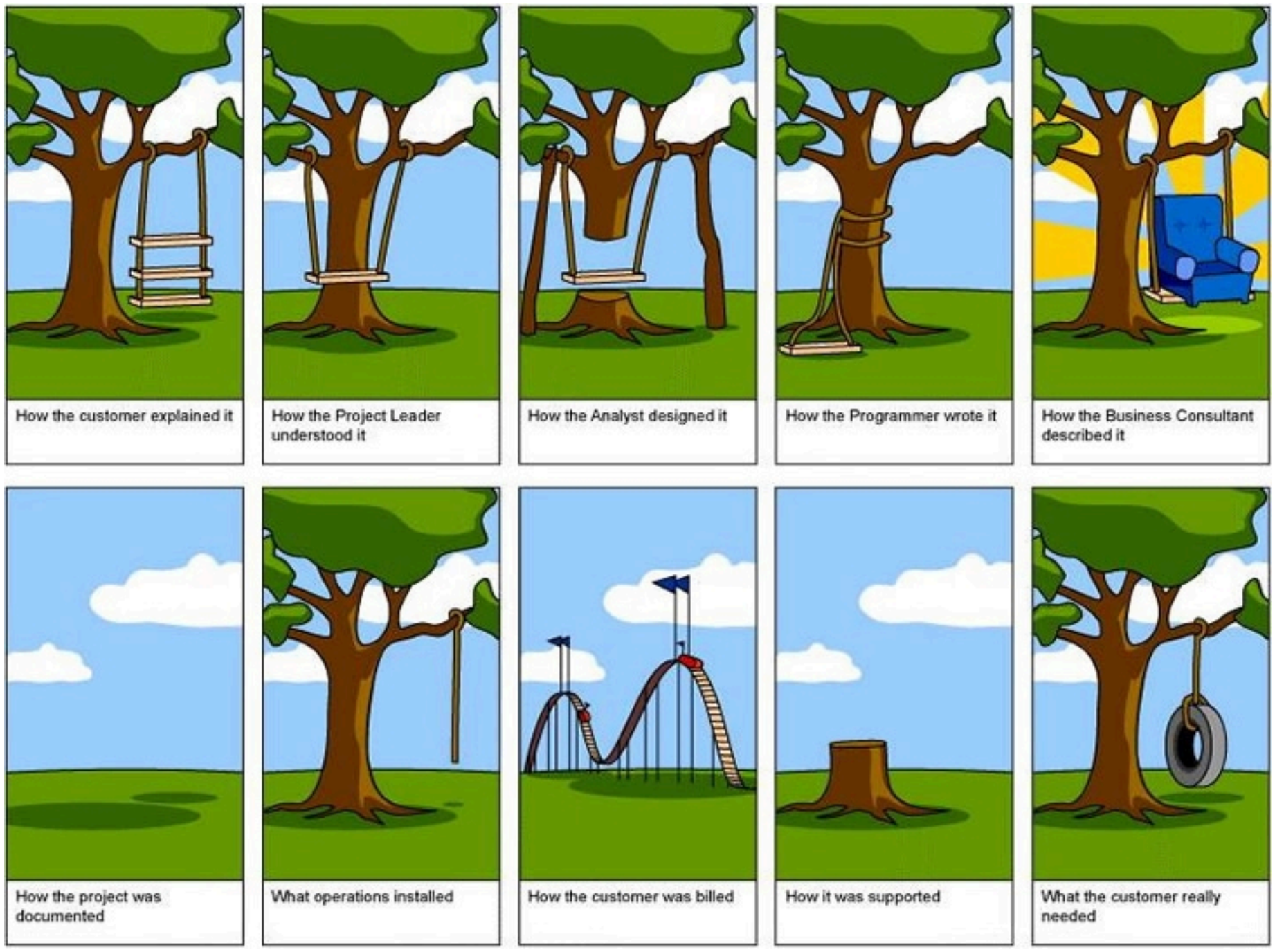
Bugs found later cost more to fix

- **Observation**: Cost to fix a bug increases exponentially.
  - E.g., **a bug** found during specification costs $1 to fix during specification;
  - … if found later in design, the cost may be $10;
  - … if found much later in coding, the cost may be $100;
  - … if found ways so much later after its release, the cost may be $1000,

What can be the sources of problems

- **Requirements Definition:** Erroneous, incomplete, inconsistent requirements.
- **Design:** Fundamental design flaws in the software.
- **Implementation:** Mistakes in chip fabrication, wiring, programming faults, malicious code.
- **Support Systems:** Poor programming languages, faulty compilers and debuggers, misleading development tools.
- **Inadequate Testing:** Incomplete testing, poor verification, mistakes in debugging.

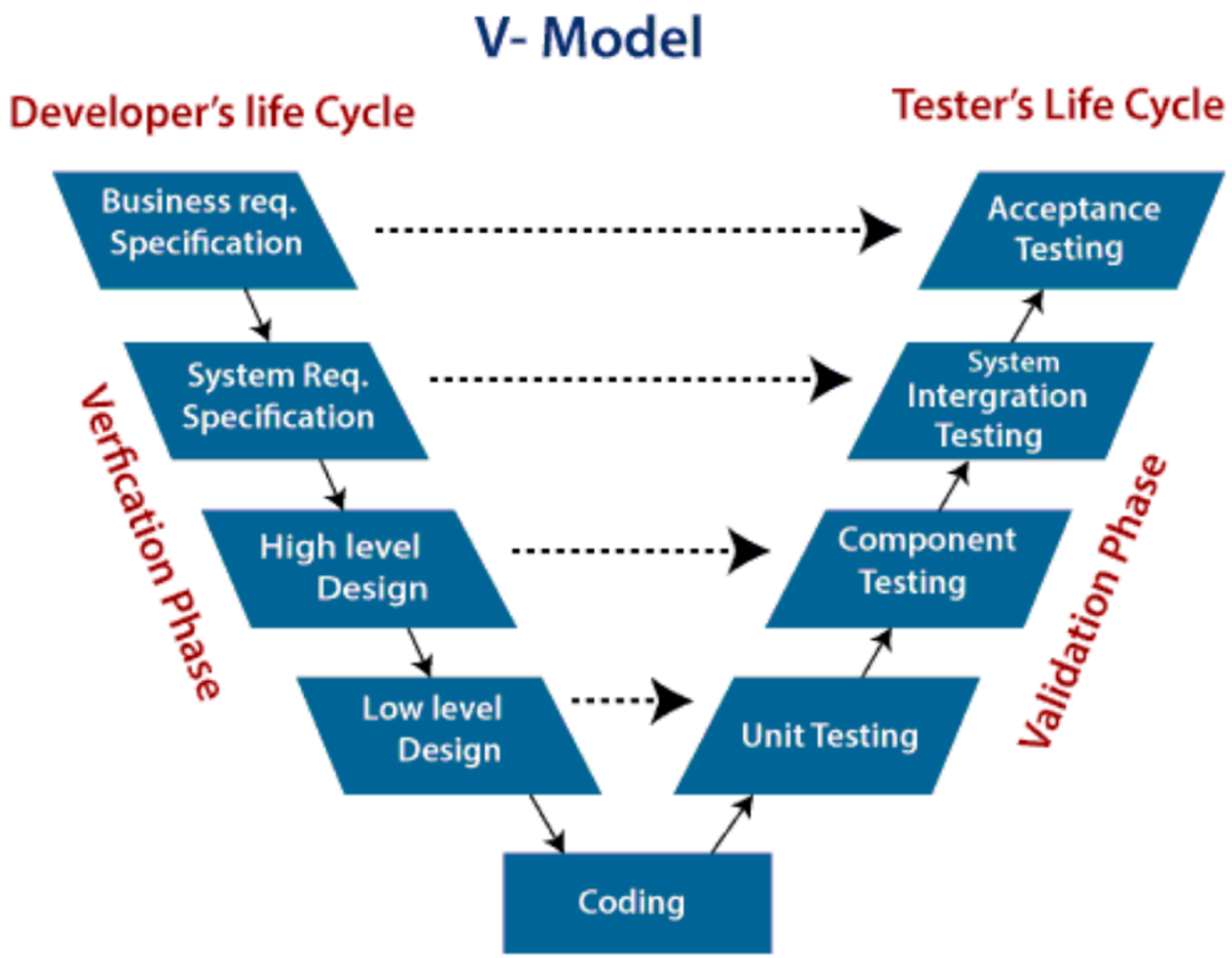
Requirement comic



Bugs are not just because of coding mistakes

- Coding and testing phases are roughly accounted for only 50 percent of the total population.
- Earlier phase bugs are not less significant in practice.

The V-model


V- Model
Developer's life Cycle
Tester's Life Cycle
Business req. Specification
Acceptance Testing
System Req. Specification
System Intergration Testing
High level Design
Component Testing
Low level Design
Unit Testing
Coding
Verification Phase
Validation Phase

Different levels of testing

- Unit testing
- Integration testing
- System testing
- Acceptance testing

Unit testing

- The most ‘micro’ scale of testing.
- Tests done on particular functions or code modules.
- Requires knowledge of the internal program design and code.
- Done by Programmers (not by testers).

Integration testing

- Continuous testing of an application as and when a new functionality is added.
- Application’s functionality aspects are required to be independent enough to work separately before completion of development.
- Done by programmers or testers.

System testing

- To verify that the system components perform control functions
- To demonstrate that the system performs both functionally and operationally as specified
- To perform appropriate types of tests relating to Transaction Flow, Installation, Reliability, Regression etc.

Acceptance testing

- To verify that the system meets the user requirements

Software testing goals

- ... to find bugs
- ... as early in the software development processes as possible
- ... and make sure they get fixed.

What is a good test

- A good test case is one that has a probability of finding an as yet undiscovered error.
- A successful test is one that uncovers a yet undiscovered error.
- A good test is not redundant.
- A good test should be “best of breed”.
- A good test should neither be too simple nor too complex.

Test plan

- A tool to help plan the testing activity product to inform others of test process
  - To create a set of testing tasks.
  - Assign resources to each testing task.
  - Estimate completion time for each testing task.

Test cases

- A test case is defined as
  - A set of test inputs, execution conditions and expected results, developed for a particular objective.
  - Documentation specifying inputs, predicted results and a set of execution conditions for a test item.
  - Specific inputs that will be tried and the procedures that will be followed when the software tested.
  - Sequence of one or more subtests executed as a sequence as the outcome and/or final state of one subtests is the input and/or initial state of the next.

Good test cases

- Have high probability of finding a new defect
- Unambiguous tangible result that can be inspected
- Repeatable and predictable
- Traceable to requirements or design documents
- Push systems to its limits
- Execution and tracking can be automated
- Do not mislead
- Feasible

Test cases example

- System test of input of numeric month into data field

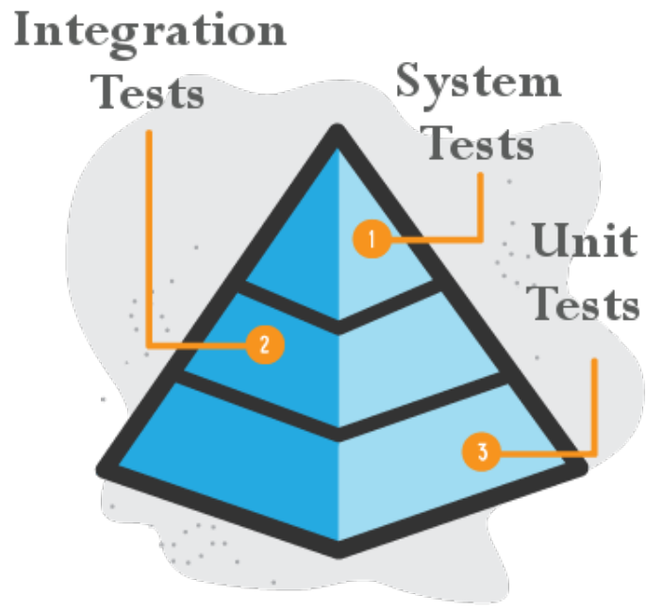
Ref.	Field	Action	Input	Expected Result	Pass/Fail
1	Month	Enter Data	0	Data rejected. Error Message 'Invalid Month'	Fail
2	Month	Enter Data	1	Data Accepted, January Displayed	Pass
3	Month	Enter Data	6	Data Accepted, June Displayed	Pass
4	Month	Enter Data	12	Data Accepted, December Displayed	Pass
5	Month	Enter Data	13	Data rejected. Error Message 'Invalid Month'	Fail

Testing mantra

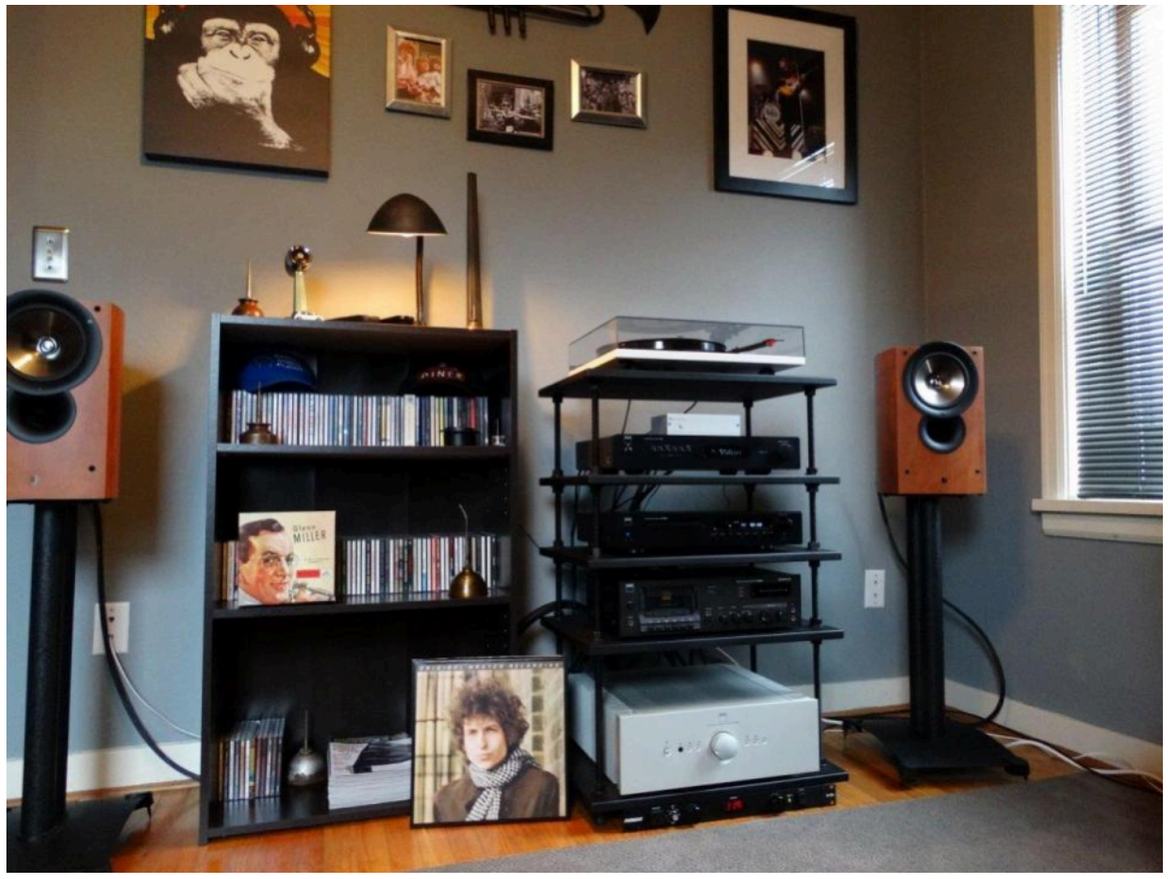
- Test as early as possible
- Gather as much knowledge of the application under test as possible
- Build ‘Bug Taxonomies’ (Classification)
- Record all tests/defects/issues/recommendations
- Never, ever, ever make it personal

Unit testing

- A unit test is a piece of code that exercises a small, specific area of functionality, e.g., a particular method in a particular context.
- If testing indicates that code works as expected, we proceed to assemble and test the whole system.

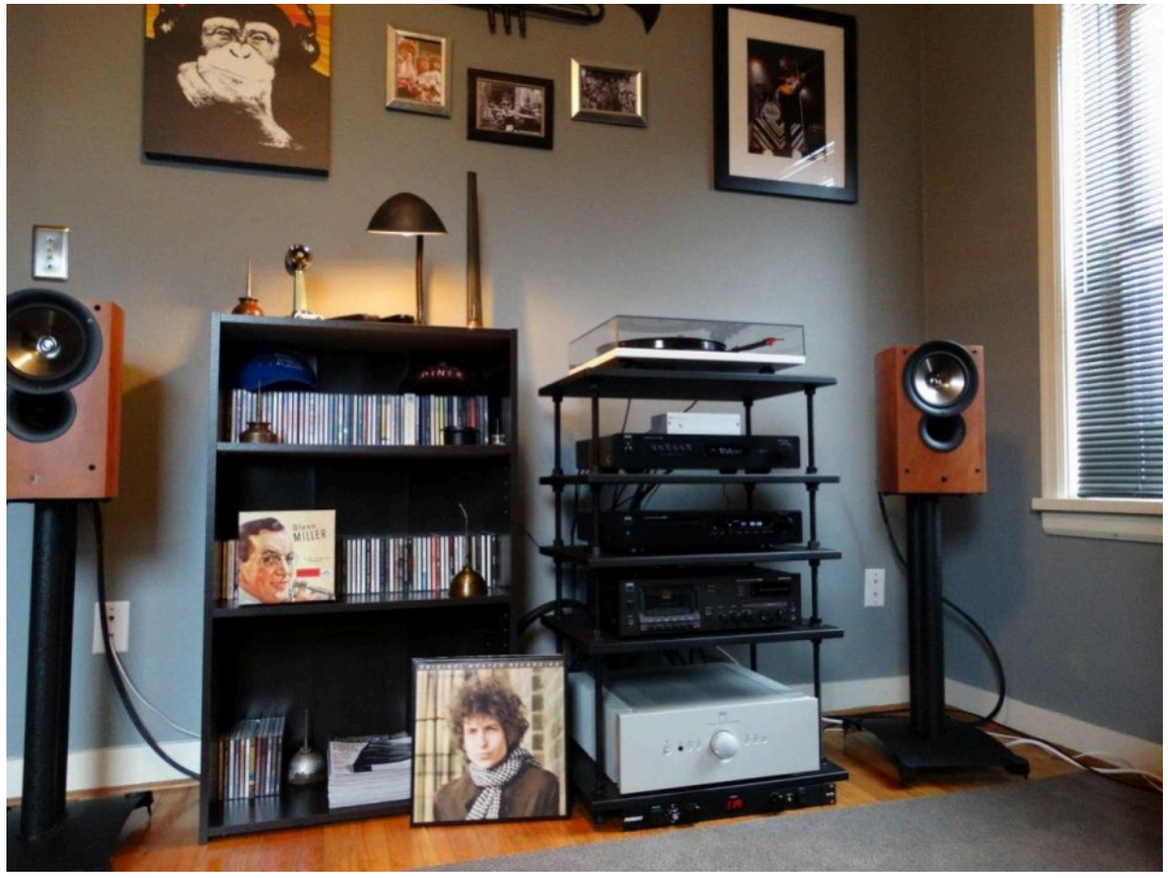


Unit testing



Debugging

Unit testing



Testing

Unit testing objective

- Coding with more confidence
- Finding bugs as early as possible
- Locating bugs as soon as possible
- Lessening the time spent on debugging
- Checking the consistency among requirements, design patterns and implementation code

JUnit

- The most commonly used framework to undertake unit testing in Java
  - For a given class Foo, we create another class named FooTest containing various test cases to test its methods at the unit level.
  - Each method looks for particular results and passes / fails.

JUnit

```
+ 1 | import org.junit.*;
+ 2 | import static org.junit.Assert.*;
+ 3 | public class nameTest {
+ 4 |    @Test
+ 5 |    public void name() {
+ 6 |       // a test case method ...
+ 7 |    }
+ 8 | }
```

- A method with @Test is flagged as a JUnit test case.
- All @Test methods run when JUnit runs your test class.

JUnit

- JUnit provides **assert** commands to help us write tests.
  - The idea: Put **assertion calls** in your test methods to check things you expect to be **true**.
  - If the assertion result turns to be **false**, the test will be failed.

Assert methods

assertTrue(**test**)	fails if the boolean test is false
assertFalse(**test**)	fails if the boolean test is true
assertEquals(**expected, actual**)	fails if the values are not equal
assertSame(**expected, actual**)	fails if the values are not the same (by ==)
assertNotSame(**expected, actual**)	fails if the values *are* the same (by ==)
assertNull(**value**)	fails if the given value is *not* null
assertNotNull(**value**)	fails if the given value is null
fail()	causes current test to immediately fail

- Each method can also be passed a string to display if it fails, e.g. assertEquals("message", expected, actual)
- Why is only **fail()** existed? … and when to use it?

JUnit

```
+ 1  import org.junit.*;
+ 2  import static org.junit.Assert.*;
+ 3  public class ArrayListMethodTest {
+ 4     @Test
+ 5     public void memberShouldBeRetrievedInOrder() {
+ 6        ArrayIntList list = new ArrayIntList();
+ 7        list.add(42);
+ 8        list.add(-3);
+ 9        list.add(15);
+ 10       assertEquals(42, list.get(0));
+ 11       assertEquals(-3, list.get(1));
+ 12       assertEquals(15, list.get(2));
+ 13    }
+ 14    @Test
+ 15    public void listShouldBeEmpty() {
+ 16       ArrayIntList list = new ArrayIntList();
+ 17       list.add(123);
+ 18       list.remove(0);
+ 19       assertTrue(list.isEmpty());
+ 20    }
+ 21 }
```

JUnit naming conventions

- Widely-used naming method for **test classes** is to let the keyword Test be the class's postfix to be tested.
  - DateObjectTest
  - FooTest

JUnit naming conventions

- Widely-used naming method for **test cases** is to give the very long descriptive names to test methods.
  - to give a simple but really long method name, such as newArrayListsHaveNo Element;
  - to use the word **should** in the test method name, such as **thememberShouldBeRetrievedInOrder** method
  - to name all the test methods in the form **Given[ExplainTheInput]When[WhatIsDone]Then[WhatIsTheExpectedResult]**, such as **GivenDateObjectWhenDayIsAddedThenDateShouldBeAdvanced**

JUnit structure conventions

- Put the expected values on the left side of the parameter list of any assertion methods, e.g.,
  assertEquals(2020, d.getYear());
- Assertion methods are widely recommended to have messages explaining what is being checked;
- When both expected values and the actual values are not single values, it is suggested to let the two be objects.

Example

```
+ 1  import org.junit.*;
+ 2  import static org.junit.Assert.*;
+ 3  public class DateObjectTest {
+ 4     @Test
+ 5     public void GivenDateObjectWhenDayIsAddedThenDateShouldBeAdvanced() {
+ 6        Date d = new Date(2020, 1, 15);
+ 7        d.addDays(14);
+ 8        Date expected = new Date(2020, 1, 29);
+ 9        assertEquals("date after +14 days", expected, d);
+10     }
+11  }
```

JUnit testing tips

- We cannot test every possible input, parameter value, etc.
  - So we must think of a limited set of tests likely to expose bugs.

- We have to think about boundary cases
  - Positive; zero; negative numbers
  - Right at the edge of an array or collection's size

JUnit testing tips

- We also have to think about empty cases and error cases
  - 0, -1, null; an empty list or array

- Finally we should think about testing behaviors in combination
  - Maybe add usually works, but fails after you call remove
  - If we make multiple calls; maybe size fails the second time only

JUnit testing tips — No torture test

```
+ 1  import org.junit.*;
+ 2  import static org.junit.Assert.*;
+ 3  public class DateTest {
+ 4     // test every day of the year
+ 5     @Test(timeout = 10000)
+ 6     public void tortureTest() {
+ 7        Date date = new Date(2050, 1, 1);
+ 8        int month = 1;
+ 9        int day = 1;
+10        for (int i = 1; i < 365; i++) {
+11           date.addDays(1);
+12           if (day < DAYS_PER_MONTH[month]) {
+13              day++;
+14           } else {
+15              month++;
+16              day=1;
+17           }
+18           assertEquals(new Date(2050, month, day), date);
+19        }
+20    }
+21    private static final int[] DAYS_PER_MONTH = {
+22       0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31
+23    };
+24 }
```

JUnit testing tips — Trustworthiness

- Test one thing at a time per test method.
  - 10 small tests are much better than 1 test 10x as large.
- Each test method should have few (likely 1) assert statements.
- Tests should avoid logic.
  - minimize if/else, loops, switch, etc.
  - avoid try/catch -> if it's supposed to throw, use expected= ...

Test Suites

- Generally, we have many test classes in one single application
- Unit provides a test suites tool to ease the test on many classes by let us combine all the test classes into a test suite.
- Running a test suite will execute all the test classes we declared in that test suite.

Test Suites

```
+ 1  import org.junit.runner.*;
+ 2  import org.junit.runners.*;
+ 3  @RunWith(Suite.class)
+ 4  @Suite.SuiteClasses({
+ 5     WeekdayTest.class,
+ 6     TimeTest.class,
+ 7     CourseTest.class
+ 8  })
+ 9  public class AllTests {
+ 10
+ 11 }
```

Reflection

- Reflection is an instrumental technique allowing us to examine or modify the run-time behavior of applications.
- We can inspect the types or properties of an object at runtime.
  - If needed, we can modify the behavior of the object as well.
- It is a very useful technique when first learning an OO language.
  - e.g., we can use reflection to gain a deeper understanding of polymorphism and the java event model.
  - Mostly used for inspecting private classes.

Reflection

- With reflection, the followings are what we can do at runtime:
  - Examine an object's class structure and behavior;
  - Construct an object for a particular class;
  - Examine a class's field and method;
  - Invoke any method of an object;
  - Change the accessibility flag of Constructor, Method, and Field.

Reflection

```
+ 1  import java.lang.reflect.Method;
+ 2  public class ReflectionExample {
+ 3     public static void main(String[] args){
+ 4        Foo f = new Foo();
+ 5        System.out.println(f.getClass().getName());
+ 6     }
+ 7  }
+ 8  class Foo {
+ 9     private void print() {
+ 10       System.out.println("abc");
+ 11    }
+ 12 }
```

Reflection

```
+ 1  import java.lang.reflect.Method;
+ 2  public class ReflectionExample {
+ 3     public static void main(String[] args){
+ 4        Foo f = new Foo();
- 5        System.out.println(f.getClass().getName());
+ 6        Method method;
+ 7        try {
+ 8           method = f.getClass().getMethod("print", new Class<?>[0]);
+ 9           method.setAccessible(true);
+ 10          method.invoke(f);
+ 11       } catch (Exception e) {
+ 12          e.printStackTrace();
+ 13       }
+ 14    }
+ 15 }
+ 16 class Foo {
+ 17    private void print() {
+ 18       System.out.println("abc");
+ 19    }
+ 20 }
```

Test-Driven development (TDD)

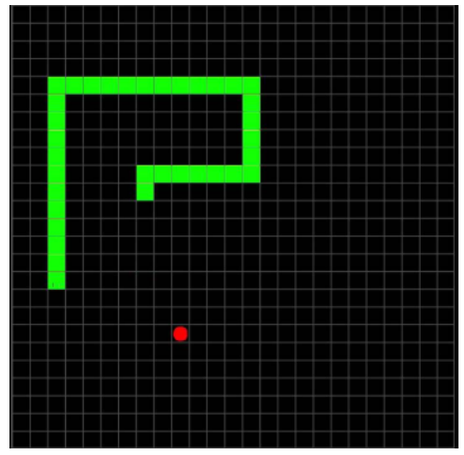
- Unit tests can be written **after**, **during**, or even **before** coding.
  - **TDD**: Write tests, then write code to pass them.
- Imagine that we'd like to add a method subtractWeeks to our Date class, that shifts this Date backward in time by the given number of weeks.
  - If we write code to test this method before it has been written.
  - Then once we do implement the method, we'll know if it works.

In steps,

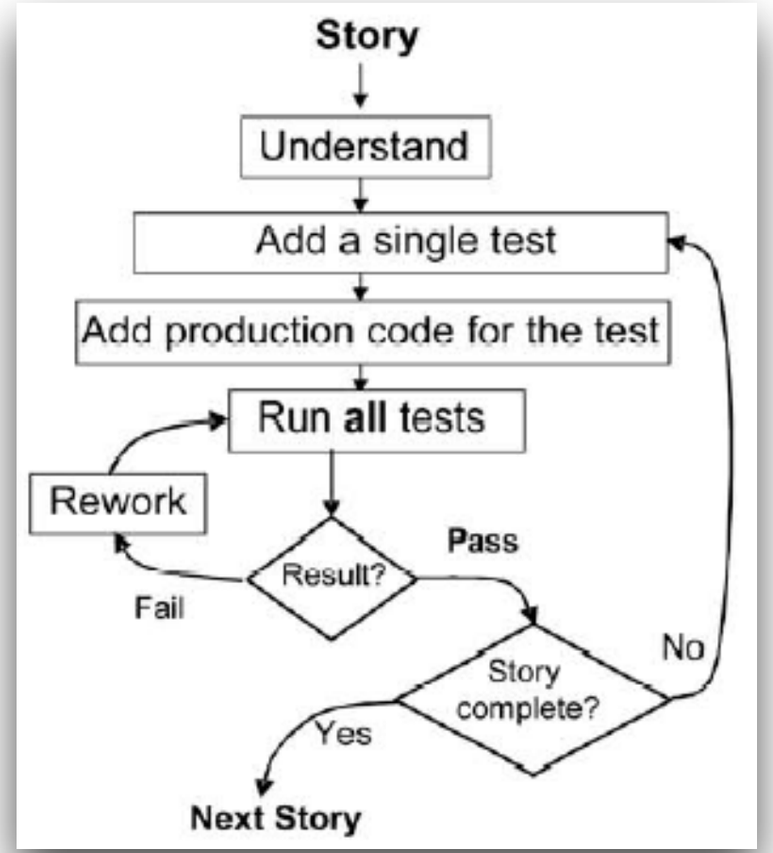
- Broken down the tasks into small tasks and prioritize them;
- Select a task to be completed, and write a test for the task;
- Run all the tests to verify that the new test fails, at least the new test case;
- Write minimal production code to complete the task;
- Run all the tests again to verify that all the tests app this time;
- Repeat from the first step until all the tasks are completed.

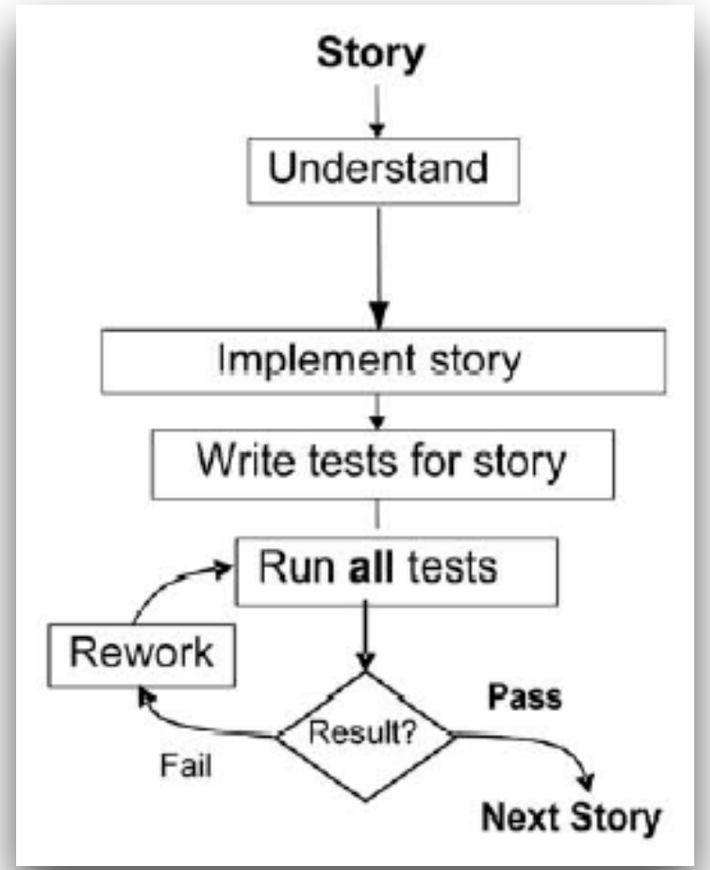
Test-Driven development (TDD)

- An example checklist
  - Realtime
  - Snake keep moving forward if the user do nothing
  - Snake can move in 4 direction
  - Input forcing snake to move to the opposite side to the current forward direction is not accepted.
  - Snake can grow if …
  - Snake can die if …
  - Snake cannot move through the screen



Test first (TDD) vs Test last





TDD benefits

- Instant Feedback
  - Developer knows instantly if new code works and if it interferes with existing code
- Better Development Practices
  - Encourages the programmers to decompose the problem into manageable, formalized programming tasks.
  - Provides context in which low-level design decisions are made By focusing on writing only the code necessary to pass tests, designs can be cleaner and clearer than is often achieved by other methods

TDD benefits

- Quality Assurance
  - Having up-to-date tests in place ensures a certain level of quality
  - Enables continuous regression testing
  - TDD practices drive programmers to write code that is automatically testable
  - Whenever a software defect is found, unit test cases are added to the test suite prior to fixing the code

TDD benefits

- Lower Rework Effort
  - Since the scope of a single test is limited, when the test fails, rework is easier
  - Eliminating defects early in the process usually avoids lengthy and tedious debugging later in the project
  - “Cost of Change” is that the longer a defect remains the more difficult and costly to remove

Summary

- Software testing is defined as an empirical technical investigation conducted to provide stakeholders with information about the product's quality or service under test.
- Tests need failure atomicity (ability to know exactly what failed).
  - Each test should have a clear, long, descriptive name.
  - Assertions should always have clear messages to know what failed.
  - Write many small tests, not one big test.
  - No torture test
- **TDD** is to write tests first, then write minimum production code to pass them.

Questions