

cs304

Software Engineering

TAN, Shin Hwei

陈馨慧

Southern University of Science and Technology

Slides adapted from cs427 (UIUC) and cs304(SUSTech)

Administrative Info

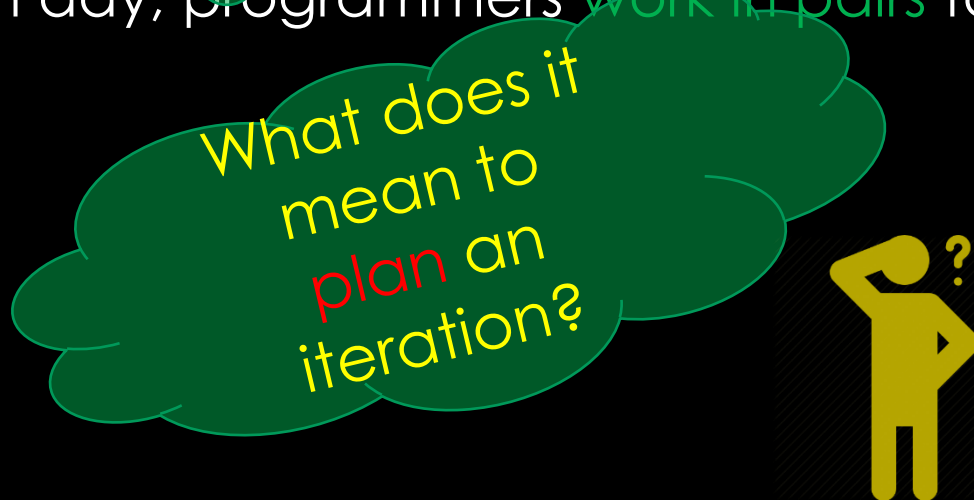
- **MP0 uploaded**, due 7 March 2022 at 11.59pm
 - Simple assignment
 - Should be able to finish in 1-2 hours within lab today
- **Project Proposal uploaded**, due 11 March 2022 at 11.59pm
 - Included frequently asked questions and corresponding answers

Question: Can we have a repository for the class project where we can use for the entire semester?

Answer: Yes, we provide a main repository for you to commit your code for the project for the entire semester. There is no deadline for this repository so you can use it for the entire semester even after finishing the class

RECALL: XP IS AN ITERATIVE PROCESS

- Iteration = two week cycle (1-3 weeks)
- Plan each iteration in an iteration meeting that is held at the start of the iteration
- Iteration is going to implement set of user stories
- Divide work into tasks small enough to finish in a day
- Each day, programmers work in pairs to finish tasks



WHAT ARE THIS SEQUENCE CALLED?
1, 1, 2, 3, 5, 8, 13 AND 21.

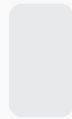
1. Odd numbers
2. $X_{i+1}=X_i+2$
3. Fibonacci Sequence

PLANNING POKER

Feeling lonely? 🤖

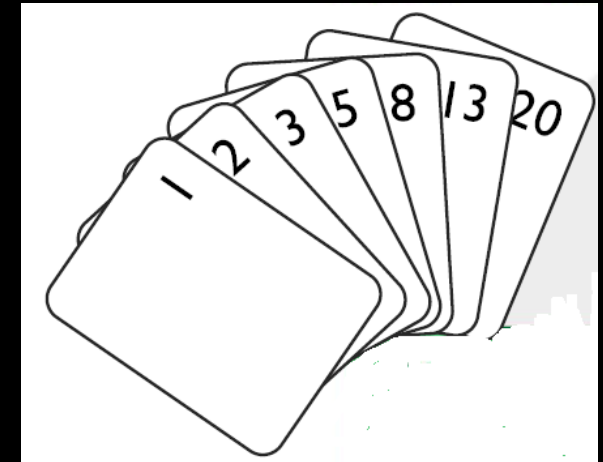
[Invite players](#)

Pick your cards!



**Shin Hwei
Tan**

Choose your card 🖱️



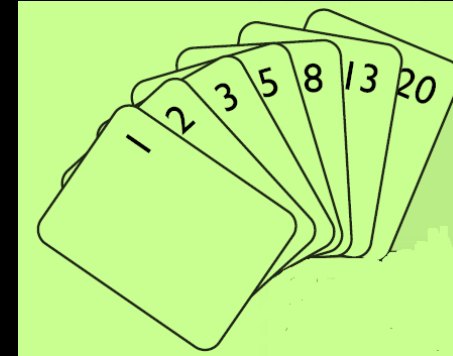
RULES OF THE GAME

- Let's watch this video to find out what planning poker is all about:
- http://youtu.be/0FbnCWWg_NY



PLAYING PLANNING POKER

- Include all players on the development team (but less than 10 people overall):
 - Programmers
 - Testers
 - Database engineers
 - Requirements analysts
 - User interaction designers . . .
- Moderator (usually the product owner or analyst) reads the description and answers any questions
- Each estimator privately selects a card with their estimate
- All cards simultaneously turned over
- Re-estimate
- Repeat until converge



GROUP DISCUSSION: PLAY PLANNING POKER

1-8

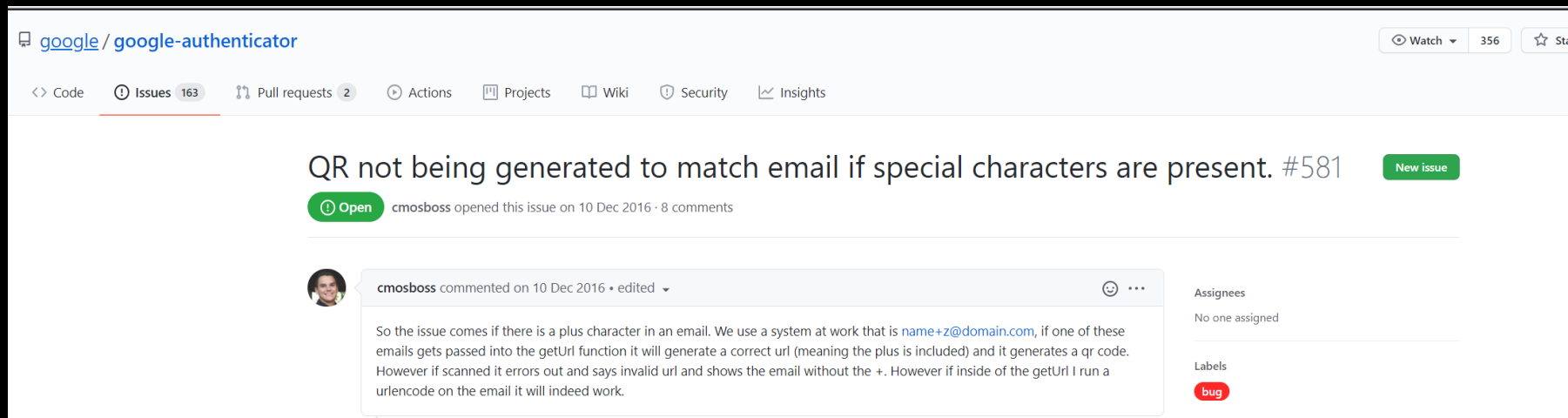
- One student in the group serves as the moderator by providing a set of tasks for an upcoming period (e.g., upcoming week, month, semester) for the rest of the group to estimate cost.
- Each student uses small pieces of papers to write 1, 2, 3, **5**, 8, 13, 20, 40, 100 on each of them.
- Carry out to play planning poker
- <https://planningpokeronline.com/9zdZSs2q4k1whlknoY3R>

GROUP DISCUSSION: PLAY PLANNING POKER

1-9

User stories:

- Add an Espresso test to check the QR code
- Handle special character

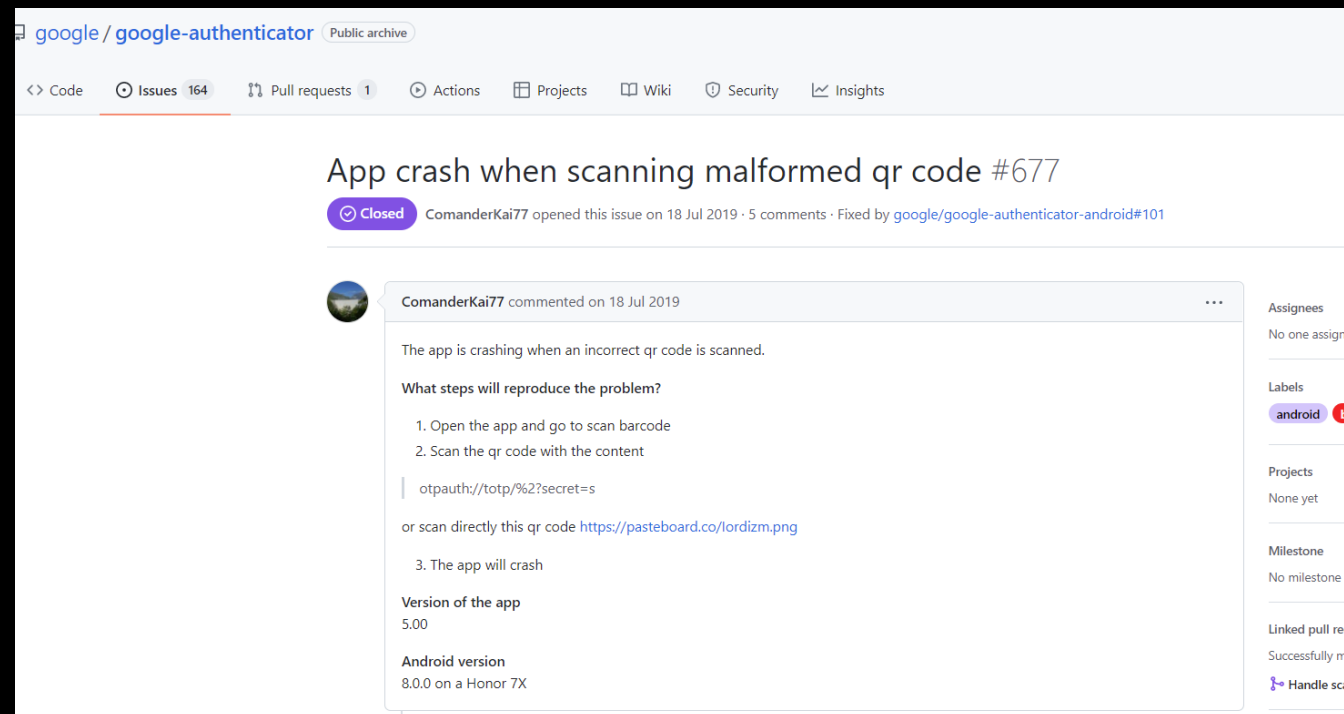


<https://github.com/google/google-authenticator/issues/581>

GROUP DISCUSSION: PLAY PLANNING POKER

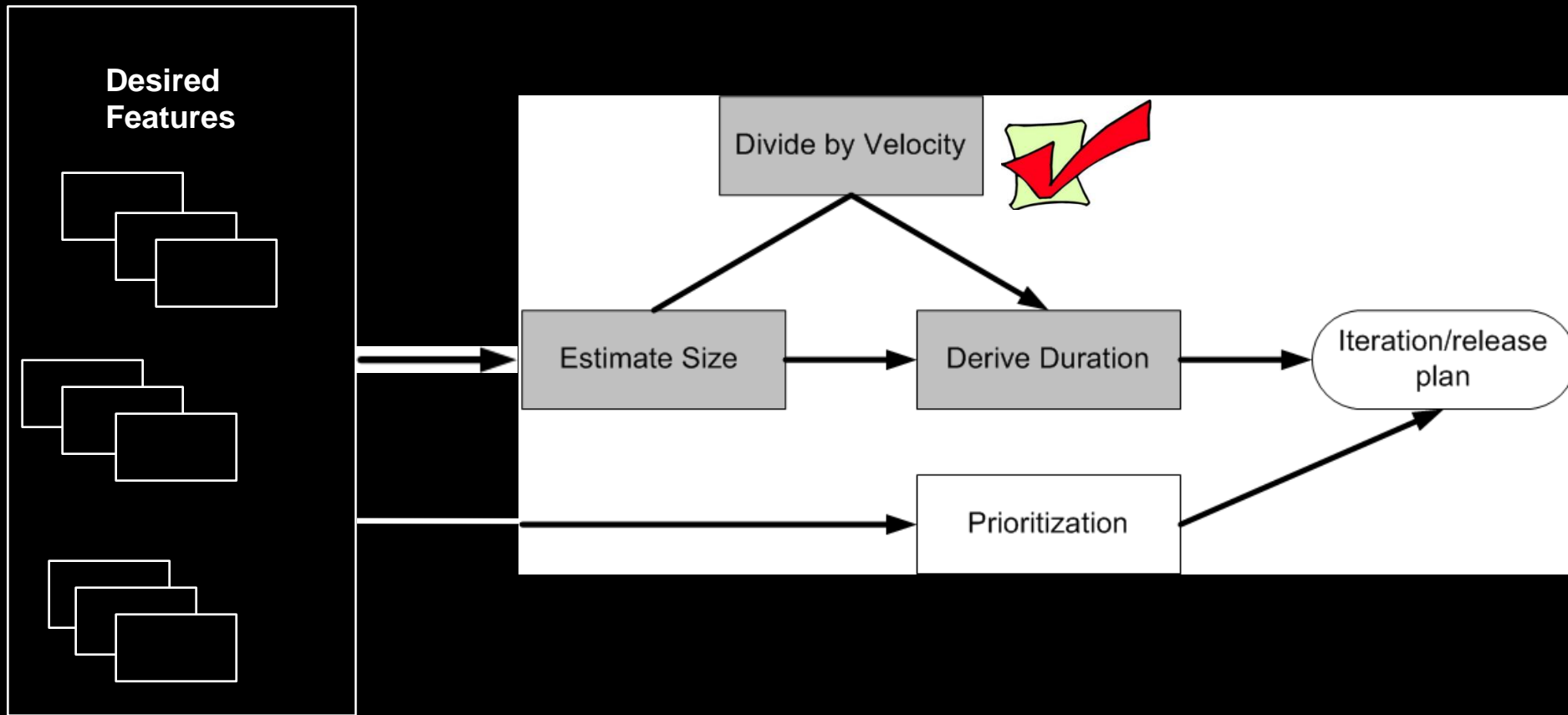
1-10

User story:
- Handle the exception



<https://github.com/google/google-authenticator/issues/677>

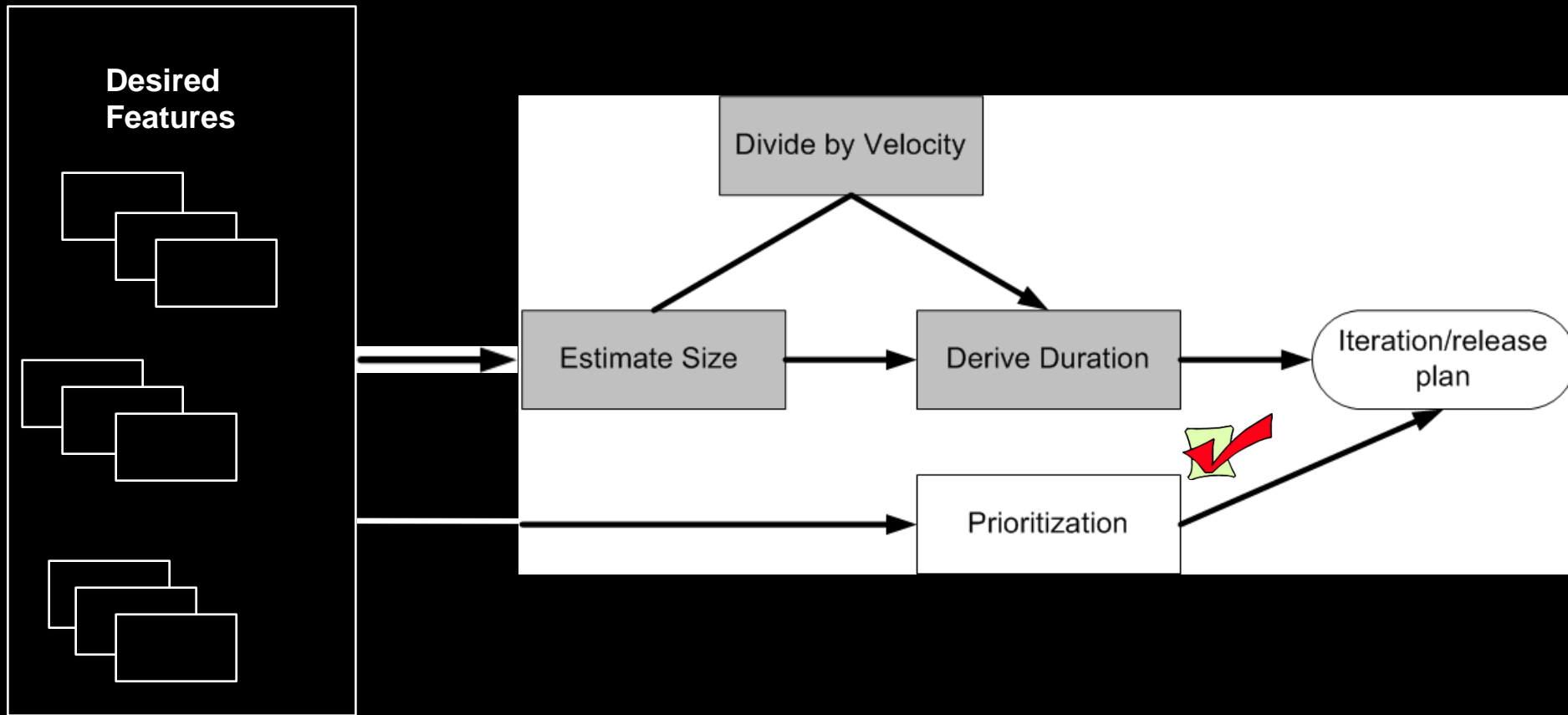
COMING UP WITH THE PLAN



VELOCITY

- Velocity is a measure of a team's rate of progress.
- Velocity is calculated by summing the number of story points assigned to each user story that the team completed during the operation.
- We assume that the team will produce in future iterations at the rate of their past average velocity.
 - “Yesterday's weather”

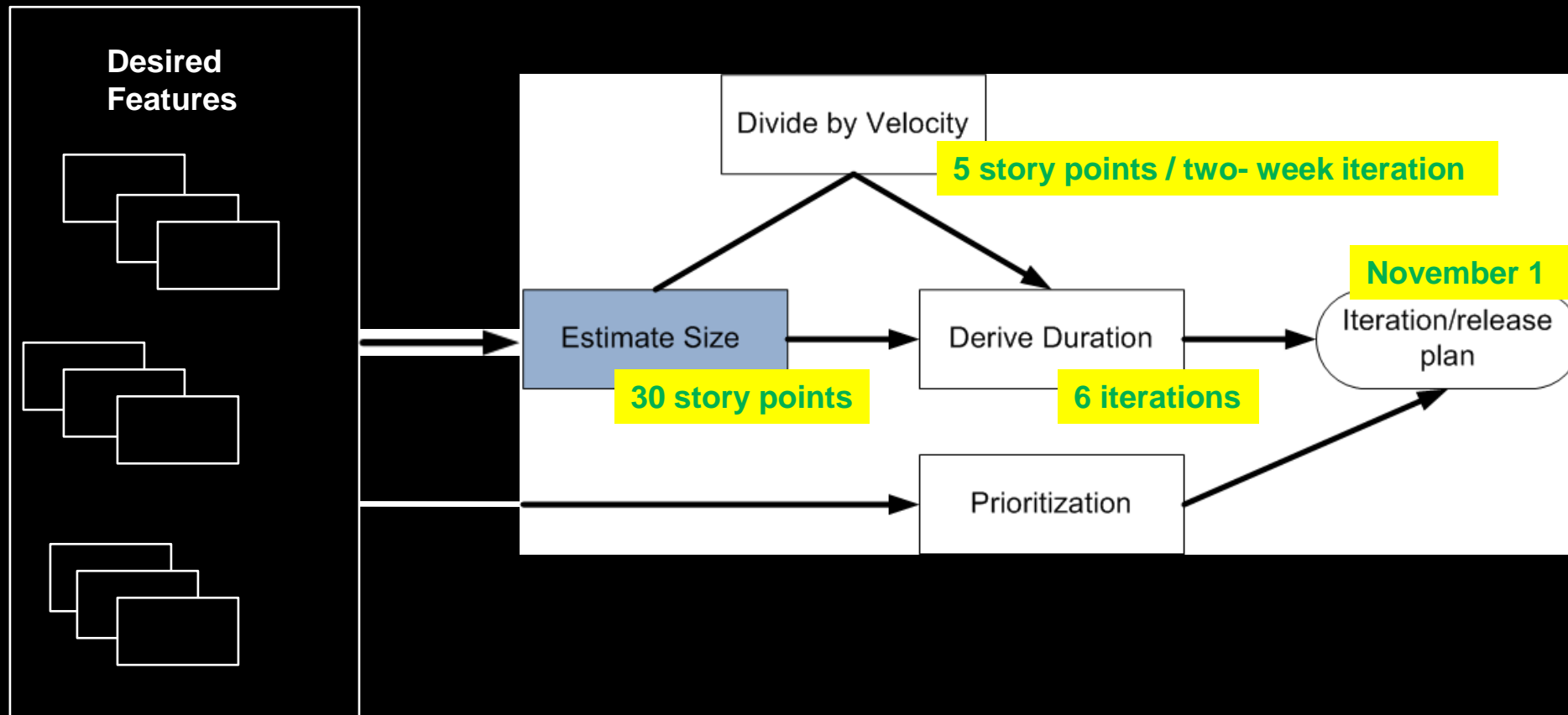
COMING UP WITH THE PLAN



PRIORITIZATION

- Driven by customer, in conjunction with developer
 - Choose features to fill up velocity of iteration, based on:
 - Desirability of feature to a broad base of customers/users
 - Desirability of feature to a small number of important customers/users
 - The cohesiveness of the story in relation to other stories
- Example:
- “Zoom in” a high priority feature
 - “Zoom out” not a high priority feature
 - But it becomes one relative to “Zoom in”

COMING UP WITH THE PLAN



PLANNING GAME

- Customer writes **user stories**
- Programmers estimate time to do each story
- If story is too big, customer splits it
- Customer chooses stories to match **project velocity**
- Project velocity is amount of work done in the previous iteration(s)



PLANNING

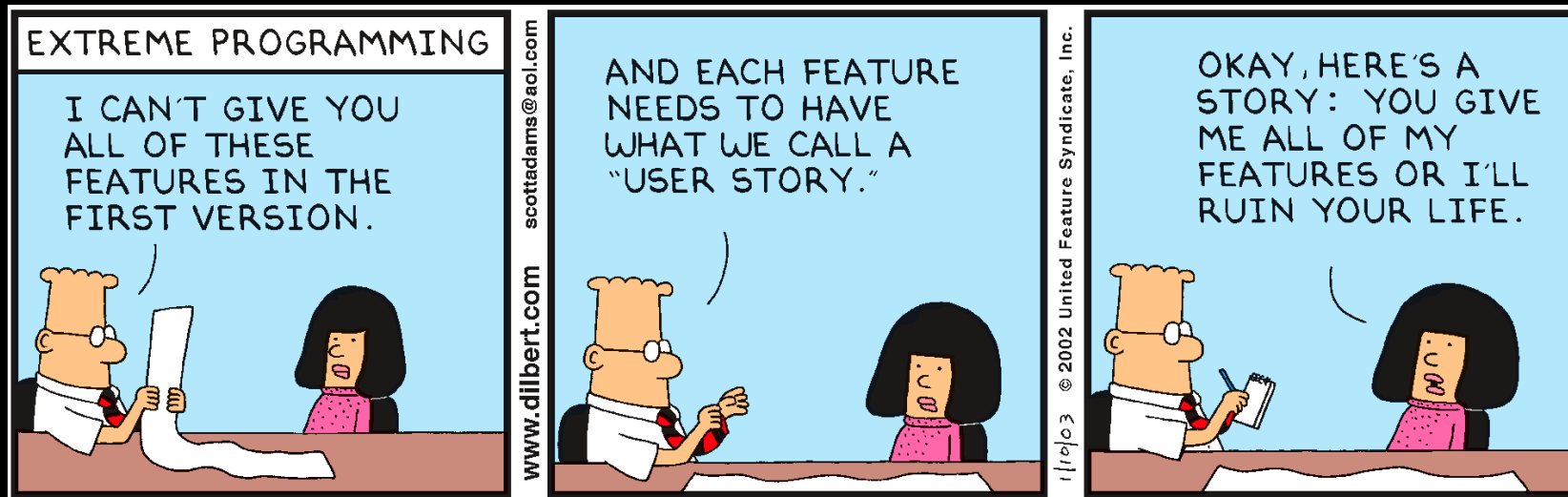
- Programmers only worry about **one iteration at a time**
- Customer can plan as many iterations as desired, but can change future iterations



SIMPLICITY

- Add one feature (user story) at a time
- Don't worry about future stories
- Make program as simple as possible
- The simplest thing that could possibly work

NEED EDUCATED CUSTOMER



XP WORKS BEST WHEN

- Educated customer on site
- Small team
- People who like to talk
- All in one room (including customer)
 - These days the room can also be virtual (slack channel?)
- Changing requirements

UNIT TESTS AND REFACTORING

- Because code is as simple as it can be, adding a new feature tends to make it less simple
- To recover simplicity, you must **refactor** the code
- To refactor safely, you should have a rigorous set of **unit tests**

WORKING SOFTWARE

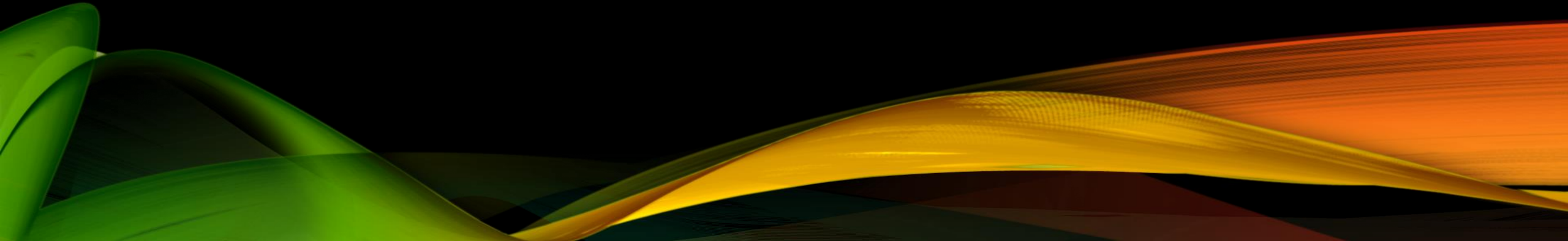
- All software has automated (unit) tests
- All tests pass, all the time
 - Never check in broken code
- How to work on a task
 - Get latest version of the code. All tests pass.
 - Write test first. It fails.
 - Write code to make test pass. Now all tests pass.
 - Refactor (clean up)
 - Check in your code

ONE KEY PRACTICE

- Write **tests first**, then write code
- Various names
 - Test-first programming
 - Test-driven development
- Is it testing or designing?
- Degree to which you stick to it for MP?

WHY TEST?

CONSEQUENCES OF BUGS



WHY TEST?

- Improve quality - find bugs
- Measure quality
 - Prove there are no bugs? (Is it possible?)
 - Determine if software is ready to be released
 - Determine what to work on
 - See if you made a mistake
- Learn the software
- Grade MPs 😊

SOME COSTLY “BUGS”

- NASA Mars space missions
 - Priority inversion (2004)
 - Different metric systems (1999)
- BMW airbag problems (1999)
 - Recall of 15,000+ cars
- Ariane 5 crash (1996)
- Your own favorite examples?

ARIANE 5 CRASH (1996)

- Uncaught exception of numerical overflow



FATAL UBER CRASH (2018)

- Software error in autonomous vehicles



ECONOMIC IMPACT

- “The Economic Impact of Inadequate Infrastructure for Software Testing”
NIST Report, May 2002
- \$59.5B annual cost of inadequate software testing infrastructure
- \$22.2B annual potential cost reduction from feasible infrastructure improvements

ECONOMIC IMPACT – CONT.



- Is Knight's \$440 million glitch the costliest computer bug ever?
 - <http://money.cnn.com/2012/08/09/technology/knight-expensive-computer-bug/index.html>

WHAT IS A TEST?

- Run program with known inputs (test inputs/data), check results (with test oracles)
 - Tests pass (green) or fail (red)
- Tests can document faults
- Tests can document code
- Important terminology to remember:
 - Mistake, fault (or defect, or bug), failure, error
 - Oracle

TERMINOLOGY: MISTAKE, FAULT/BUG, FAILURE, ERROR

Programmer makes a **mistake**

Running the
test inputs ...

Fault (**defect**, **bug**) appears in the **program**

Fault remains undetected during testing

Program **failure** occurs during execution
(program behaves unexpectedly)

Error: difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition

Fault and Failure Example

- A patient gives a doctor a list of **symptoms**
 - **Failures**
- The doctor tries to diagnose the root cause, the **ailment**
 - **Fault**
- The doctor may look for **anomalous internal conditions** (high blood pressure, irregular heartbeat, bacteria in the blood stream)
 - **Errors**

DISCUSSION

WHAT IS THE FAULT, FAILURE, ERROR IN THIS EXAMPLE?



A Concrete Example

Fault: Should start searching at 0, not 1

```
public static int numZero (int [ ] arr)
{ // Effects: If arr is null throw NullPointerException
  // else return the number of occurrences of 0 in arr
  int count = 0;
  for (int i = 1; i < arr.length; i++)
  {
    if (arr [ i ] == 0)
    {
      count++;
    }
  }
  return count;
}
```

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Error: i is 1, not 0, on the first iteration
Failure: none

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

TERMINOLOGY: MISTAKE, FAULT/BUG, FAILURE, ERROR

Programmer makes a **mistake**

Based on
test oracles ...

Fault (**defect**, **bug**) appears in the program

Fault remains undetected during testing

Program **failure** occurs during execution
(program behaves unexpectedly)

Error: difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition

TEST INPUT VS. TEST ORACLE

Objective: double the balance and then add 10

```
int calAmount () {  
    int ret = balance * 3;  
    ret = ret + 10;  
    return ret;  
}
```

test input

test oracle

```
void testCalAmount() {  
    Account account = new Account();  
    account.setBalance(1);  
    int amount = account.calAmount();  
    assertTrue(amount == 12);  
}
```

JUNIT BASICS

- Open source (junit.org) Java testing framework used to write and run repeatable **automated tests**
- A structure for writing **test drivers**
- JUnit features include:
 - **Assertions** for testing expected results
 - **Sharing common test data** among tests
 - **Test suites** for easily organizing and running tests
 - **Test runners**, both graphical and textual
- JUnit is widely used in industry
- Can be used as **stand alone** Java programs (from command line) or **from an IDE** such as IntelliJ or Eclipse

JUNIT TESTS

- JUnit can be used to test ...
 - ... an entire object
 - ... part of an object – method or interacting methods
 - ... interaction between several objects
- Primarily unit & integration testing, not system testing
- Each test is embedded into one test method
- A test class contains one or more test methods
- Test classes include:
 - A test runner to run the tests - main()
 - A collection of test methods
 - Methods to set up the state before and update the state after each test and before and after all tests

WRITING TESTS FOR JUNIT

- Need to use methods of `junit.framework.assert` class
 - `javadoc` gives a complete description of its capabilities
- Each test method checks a condition (`assertion`) and reports to the test runner whether the test `succeeded` or `failed`
- The test runner uses the result to report to the user (in command line mode) or update the display (in an IDE)
- All of the methods `return void`
- A few representative methods (of `junit.framework.assert`):
 - `assertTrue(boolean)`
 - `assertTrue(String, boolean)`
 - `assertEquals(Object, Object)`
 - `assertNull(Object)`
 - `Fail(String)`



EXAMPLE JUNIT TEST

```
public class Calc
{
    public long add (int a, int b)
    {
        return a + b;
    }
}
```

```
import org.junit.Test;
import static org.junit.Assert.*;
public class calcTest
{
    private Calc calc;
    @Test public void testAdd()
    {
        calc = new Calc ();
        assertEquals((long) 5, calc.add(2,3));
    }
}
```

Expected
result

The test

No assertion =
no junit test ...



JUNIT TEST FIXTURES

- A **test fixture** is the state of the test
 - Objects and variables used by more than one test
 - Initializations (*prefix* values)
 - Reset values (*postfix* values)
- Different tests can **use** objects without sharing state
- Objects in fixtures declared as **instance variables**
- They should be initialized in a **@Before** method
 - JUnit runs them **before** every @Test method
- Can be deallocated or reset in an **@After** method
 - JUnit runs them **after** every @Test method

TESTING THE IMMUTABLE STACK CLASS

```
public class Stack
{
...
    public String toString()
    { // EFFECTS: Returns String representation
      // of this Stack from top to bottom.
      StringBuffer buf = new StringBuffer("{}");
      for (int i = size-1; i >= 0; i--)
      {
          if (i < (size-1))
              buf.append(", ");
          buf.append(elements[ i ].toString());
      }
      buf.append("{}");
      return buf.toString();
    }
...
}
```

```
...
    public boolean repOk() {
        if (elements == null) return false;
        if (size != elements.length) return false;
        for (int i = 0; i < size; i++) {
            if (elements[i] == null) return false;
        }
        return true;
    }
}
```

STACK TEST CLASS

- Classes to import:
- Pre-test setup method (prefix) :
- Post-test teardown method (postfix) :

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.assertEquals;
import junit.framework.JUnit4TestAdapter;
```

```
private Stack stack;
// setUp method using @Before syntax
// @Before methods are run before each test
@Before public void runBeforeEachTest()
{
    stack = new Stack();
}
```

```
// tear-down method using @After
// @After methods are run after each test
@After public void runAfterEachTest()
{
    stack = null;
}
```

STACK TEST CASES

```
@Test public void testToString()
{
    stack = stack.push (new Integer (1));
    stack = stack.push (new Integer (2));
    assertEquals ("{2, 1}", stack.toString());
}
```

A problem with this “test” is that it actually combines **four separate tests** in one method

Without automation, large tests have the advantage of **reducing costs** of running many tests

With automation, small tests allow us to more easily identify failures ...

```
@Test public void testRepOk()
{
    1 boolean result = stack.repOk();
    assertEquals (true, result);
    stack = stack.push (new Integer (1));
    result = stack.repOk();
    2 assertEquals (true, result);
    stack = stack.pop();
    result = stack.repOk();
    3 assertEquals (true, result);
    stack = stack.push (new Integer (1));
    stack.top();
    result = stack.repOk();
    4 assertEquals (true, result);
}
```

STACK TEST CASES (2)

```
@Test public void testRepOkA()
{
    boolean result = stack.repOk();
    assertEquals (true, result);
}
```

```
@Test public void testRepOkB()
{
    stack = stack.push (new Integer(1));
    boolean result = stack.repOk();
    assertEquals (true, result);
}
```

```
@Test public void testRepOkC()
{
    stack = stack.push (new Integer (1));
    stack = stack.pop();
    boolean result = stack.repOk();
    assertEquals (true, result);
}
```

```
@Test public void testRepOkD()
{
    stack = stack.push (new Integer (1));
    stack.top();
    boolean result = stack.repOk();
    assertEquals (true, result);
}
```

RUNNING FROM COMMAND LINE

- This is all we need to run JUnit in an IDE
- We could use a `main()` for command line execution
... but even better is to run from the Runner

RUNNING ALL TESTS

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import junit.framework.JUnit4TestAdapter;
```

The name of **your**
test class

```
// This section declares all of the test classes in the program.
@RunWith (Suite.class)
@Suite.SuiteClasses ({ StackTest.class }) // Add test classes here.
```

```
public class AllTests
{
```

```
    // Execution begins at main(). In this test class, we will execute
    // a text test runner that will tell you if any of your tests fail.
```

```
    public static void main (String[] args)
```

```
    {
```

```
        junit.textui.TestRunner.run (suite());
```

```
    }
```

```
    // The suite() method is helpful when using JUnit 3 Test Runners or Ant.
```

```
    public static junit.framework.Test suite()
```

```
    {
```

```
        return new JUnit4TestAdapter (AllTests.class);
```

```
    }
```

```
}
```


HOW TO RUN TESTS

- JUnit provides **test drivers**
 - **Character-based** test driver runs from the command line
 - GUI-based test driver: ***junit.swingui.TestRunner***
 - Allows programmer to specify the test class to run
 - Creates a “**Run**” button
- If a test fails, JUnit gives the location of the failure and any exceptions that were thrown

ADVANCED TOPICS IN JUNIT

- **Assertion patterns**
 - How to decide if your test passes
 - State testing vs. interaction testing patterns
- **Parameterized JUnit tests**
 - How to describe and run very similar tests
- **JUnit theories**
 - Applying the contract model to testing
 - AAA model: Assume, Act, Assert
 - Very powerful approach
 - But also still a work in progress

ASSERTION PATTERNS

- State Testing Patterns
 - Final State Assertion
 - Most Common Pattern: *Arrange. Act. Assert.*
 - Guard Assertion
 - Assert Both Before and After The Action (Precondition Testing)
 - Delta Assertion
 - Verify a Relative Change to the State
 - Custom Assertion
 - Encodes Complex Verification Rules
- Interaction Assertions
 - Verify Expected Interactions
 - Heavily used in *Mocking* tools
 - Very Different Analysis Compared to State Testing
 - Resource: <http://martinfowler.com/articles/mocksArentStubs.html>

PARAMETERIZED TESTS

- Problem: Testing a function with similar values
 - How to avoid test code bloat?
- Simple example: Adding two numbers
 - Adding given pair of numbers is just like adding any other pair
 - You really only want to write one test
- Parameterized unit tests call constructor for each logical set of data values
 - Same tests are then run on each set of data values
 - List of data values identified with `@Parameters` annotation

PARAMETERIZED UNIT TESTS

```
import org.junit.*;
import org.junit.runner.RunWith;
import org.junit.runners.Parameterized;
import org.junit.runners.Parameterized.Parameters;
import static org.junit.Assert.*;
import java.util.*;

@RunWith(Parameterized.class)
public class ParamTest {
    public int sum, a, b;
    public ParamTest (int sum, int a, int b) {
        this.sum = sum; this.a = a; this.b = b;
    }

    @Parameters public static Collection<Object[]> parameters() {
        return Arrays.asList (new Object [][] {{0, 0, 0}, {2, 1, 1}});
    }

    @Test public void additionTest() { assertEquals(sum, a+b); }
}
```

JUNIT THEORIES

- These Are Unit Tests With Actual Parameters
 - So Far, We've Only Seen Parameterless Test Methods
- Contract Model: Assume, Act, Assert
 - Assumptions (Preconditions) Limit Values Appropriately
 - Action Performs Activity Under Scrutiny
 - Assertions (Postconditions) Check Result

```
@Theory public void removeThenAddDoesNotChangeSet(  
    Set<String> set, String string) {           // Parameters!  
    assertTrue(set.contains(string)) ;          // Assume  
    Set<String> copy = new HashSet<String>(set); // Act  
    copy.remove(string);  
    copy.add(string);  
    assertTrue (set.equals(copy));              // Assert  
    // System.out.println("Instantiated test: " + set + ", " + string);  
}
```

QUESTION: WHERE DOES DATA COME FROM?

- Answer:
 - All combinations of values from `@DataPoint` annotations where assume clause is true
 - Four (of nine) combinations in this particular case
 - Note: `@DataPoint` format is an array.

```
@DataPoints
public static String[] string = {"ant", "bat", "cat"};

@DataPoints
public static Set[] sets = {
    new HashSet(Arrays.asList("ant", "bat")),
    new HashSet(Arrays.asList("bat", "cat", "dog", "elk")),
    new HashSet(Arrays.asList("Snap", "Crackle", "Pop"))
};
```

JUNIT THEORIES NEED BOILERPLATE

```
import org.junit.*;
import org.junit.runner.RunWith;
import static org.junit.Assert.*;
import static org.junit.Assume.*;

import org.junit.experimental.theories.DataPoint;
import org.junit.experimental.theories.DataPoints;
import org.junit.experimental.theories.Theories;
import org.junit.experimental.theories.Theory;

import java.util.*;

@RunWith(Theories.class)
public class SetTheoryTest {
    ... // See Earlier Slides
}
```


JUNIT RESOURCES

- Some JUnit tutorials
 - <http://open.ncsu.edu/se/tutorials/junit/>
(Laurie Williams, Dright Ho, and Sarah Smith)
 - <http://www.laliluna.de/eclipse-junit-testing-tutorial.html>
(Sascha Wolski and Sebastian Hennebrueder)
 - <http://www.diasparsoftware.com/template.php?content=jUnitStarterGuide>
(Diaspar software)
 - <http://www.clarkware.com/articles/JUnitPrimer.html>
(Clarkware consulting)
- JUnit: Download, Documentation
 - <http://www.junit.org/>

SUMMARY

- The only way to make testing **efficient** as well as **effective** is to **automate** as much as possible
- JUnit provides a very simple way to **automate** our unit tests
- It is no “**silver bullet**” however ... it does not solve the hard problem of testing :

What test values to use ?

- **This is test design ... the purpose of test criteria**

Test Driven Development (TDD)

One of the practices in XP

Kent Beck's rules

- Beck's concept of test-driven development centers on two basic rules:
 - Never write a single line of code unless you have a failing automated test.
 - Eliminate duplication.



Informal Requirements

Maintenance: The Maintenance function records the history of items undergoing maintenance.

If the product is covered by warranty or maintenance contract, maintenance can be requested either by calling the maintenance toll free number, or through the web site, or by bringing the item to a designated maintenance station.

If the maintenance is requested by phone or web site and the customer is a US or EU resident, the item is picked up at the customer site, otherwise, the customer shall ship the item with an express courier.

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

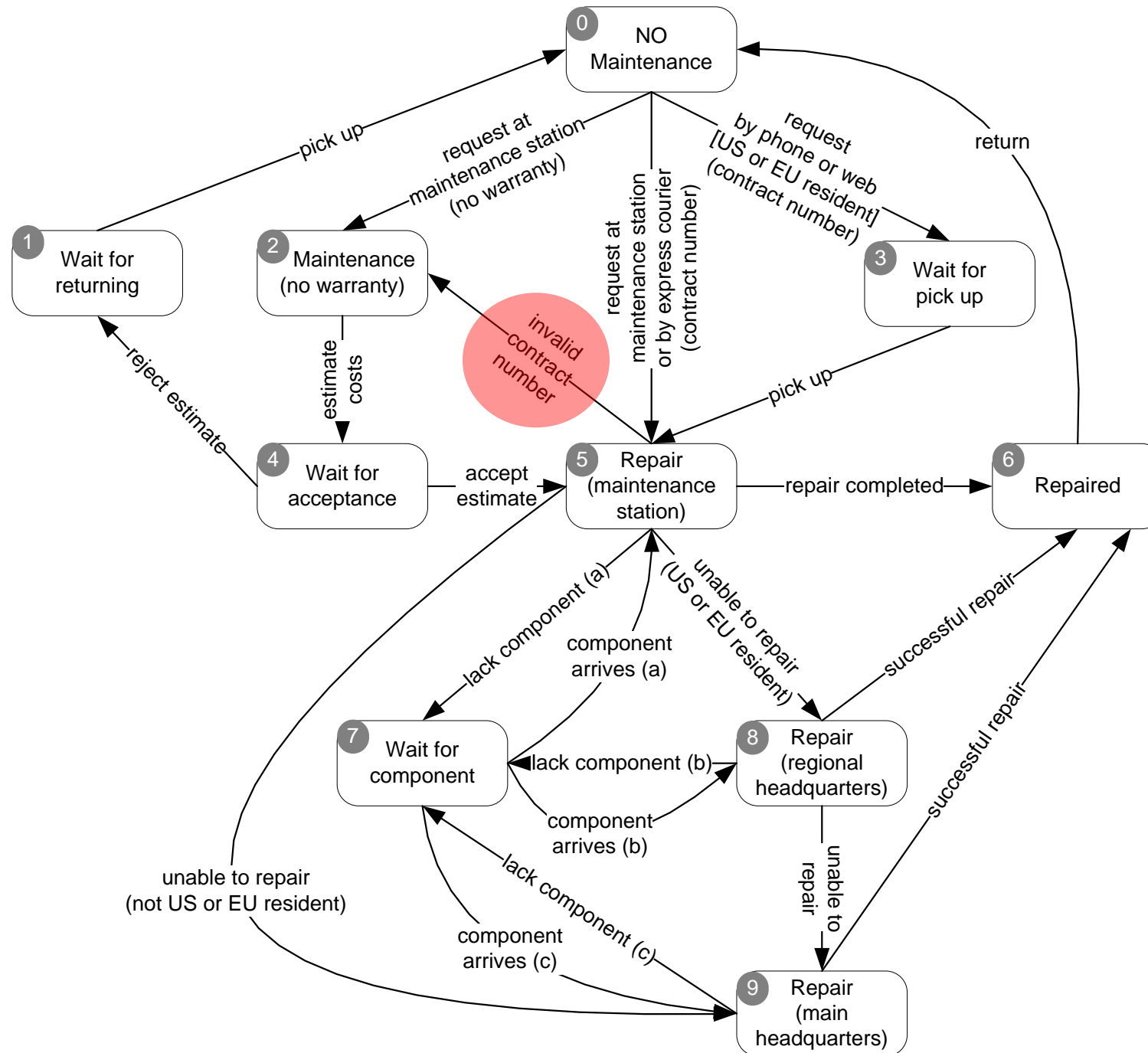
If the product is not covered by warranty or maintenance contract, maintenance can be requested

If the maintenance contract number provided by the customer is not valid, the item follows the procedure for items not covered by warranty.

maintenance main headquarters.

Maintenance is suspended if some components are not available.

Once repaired, the product is returned to the customer.



Ambiguity in Informal Requirements

If the maintenance contract number provided by the customer is not valid

- Contract number cannot contain alphabets or special characters?
- Contract number must be 5 digits?
- Contract number cannot start with 0?

Requirements based on Test Cases

```
@Test
public void testContractNumberCorrectLength() {
    assertTrue(contract.isValidContractNumber("12345"));
}
```

```
@Test
public void testContractNumberTooLong() {
    assertFalse(contract.isValidContractNumber("53434434343"));
}
```

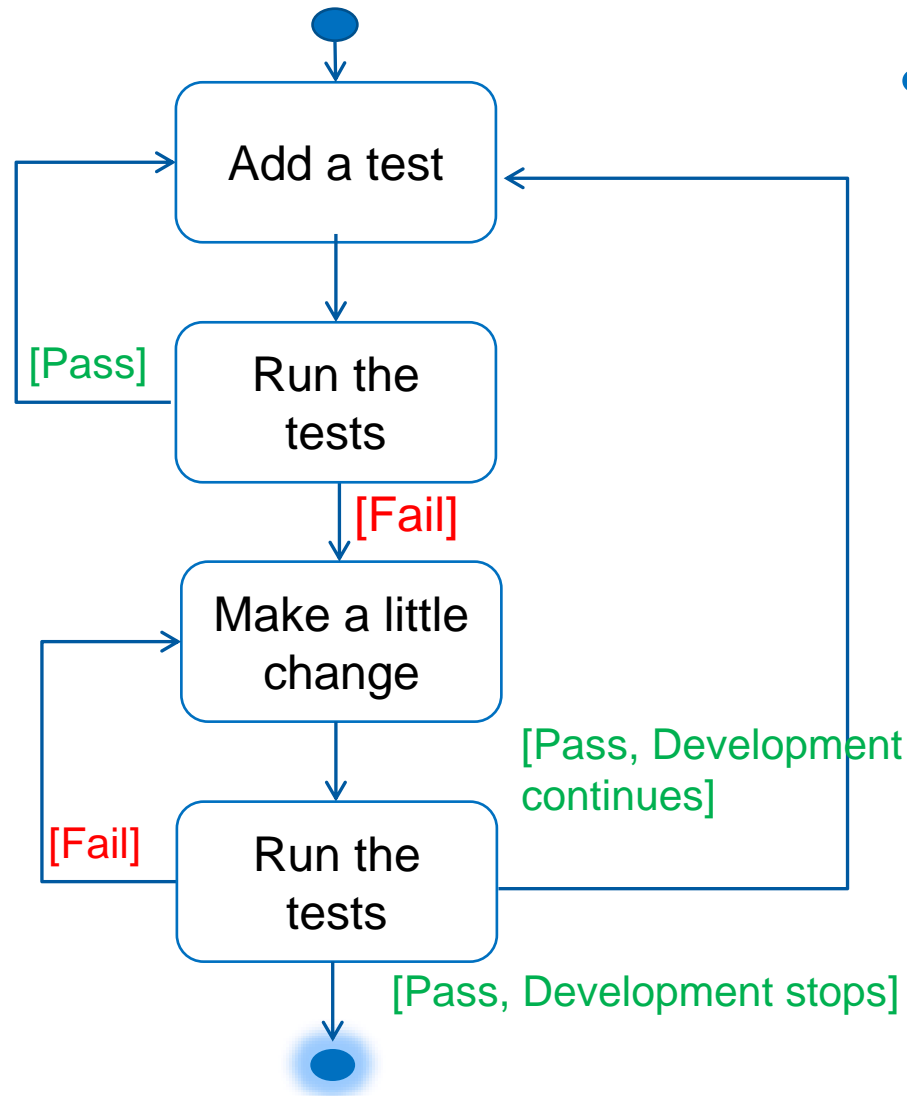
```
@Test
public void testContractNumberNoSpecialCharacter() {
    assertTrue(contract.isValidContractNumber("08067"));
}
```

```
@Test
public void testContractNumberWithSpecialCharacter() {
    assertFalse(contract.isValidContractNumber("98&67"));
}
```


Informal Requirements versus Test cases

- Test cases are more specific than requirements.
- But: How to develop code based on test cases?
 - Follow the steps in Test Driven Development(TDD)

Steps in Test Driven Development (TDD)



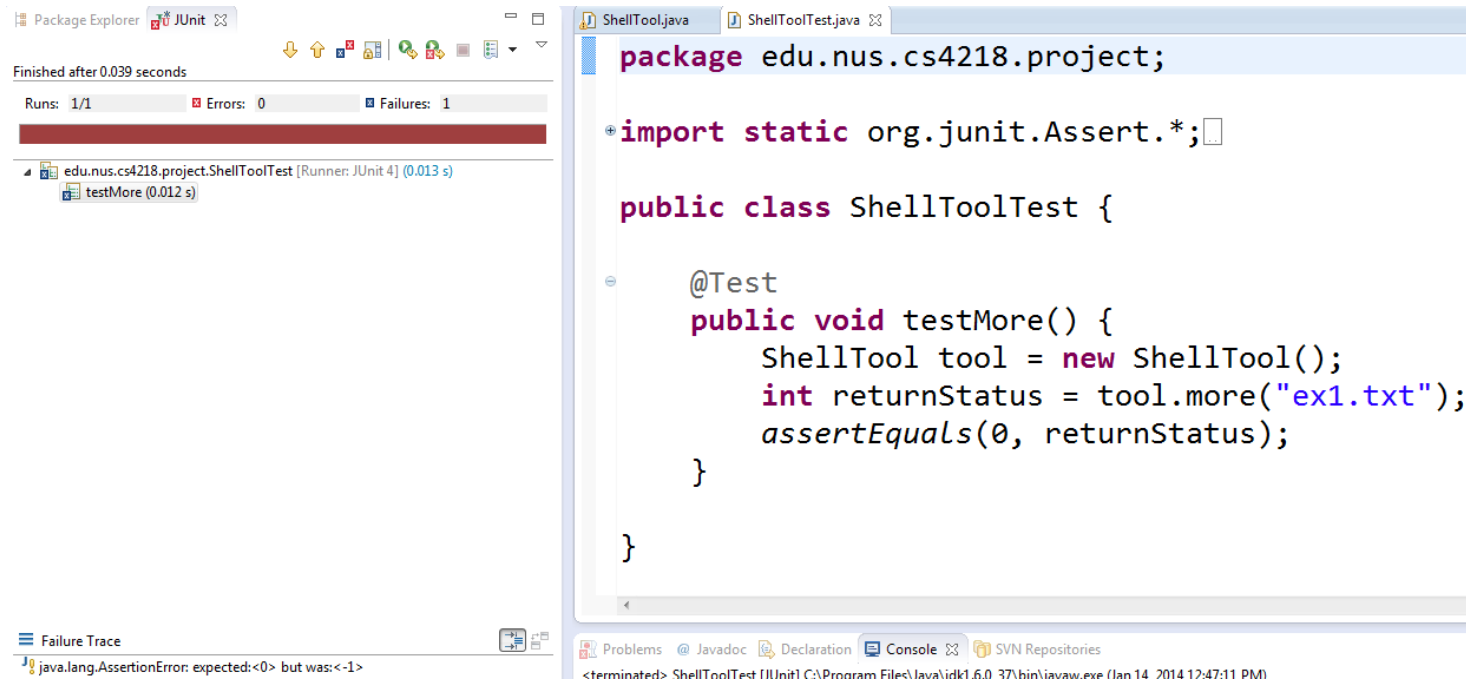
- The iterative process
 - Quickly add a test.
 - Run all tests and see the new one fail.
 - Make a little change to code.
 - Run all tests and see them all succeed.
 - Refactor to remove duplication.

Test First Scenario

- Write test for the newly added functionality
 - These test cases will serve as a specification for your implementation
 - These test cases should fail now because the corresponding methods are not implemented
 - Write minimal code to make the test pass
 - Add more tests

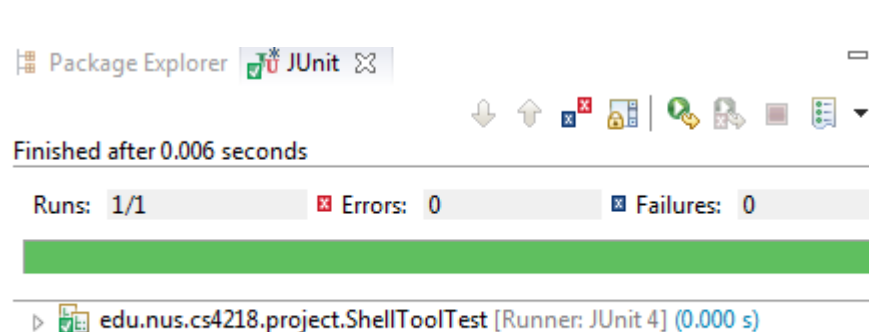
Run the tests

- Run the tests that your team gets to see the **failing** ones
 - Failing test cases indicates missing functionality



Make them Pass

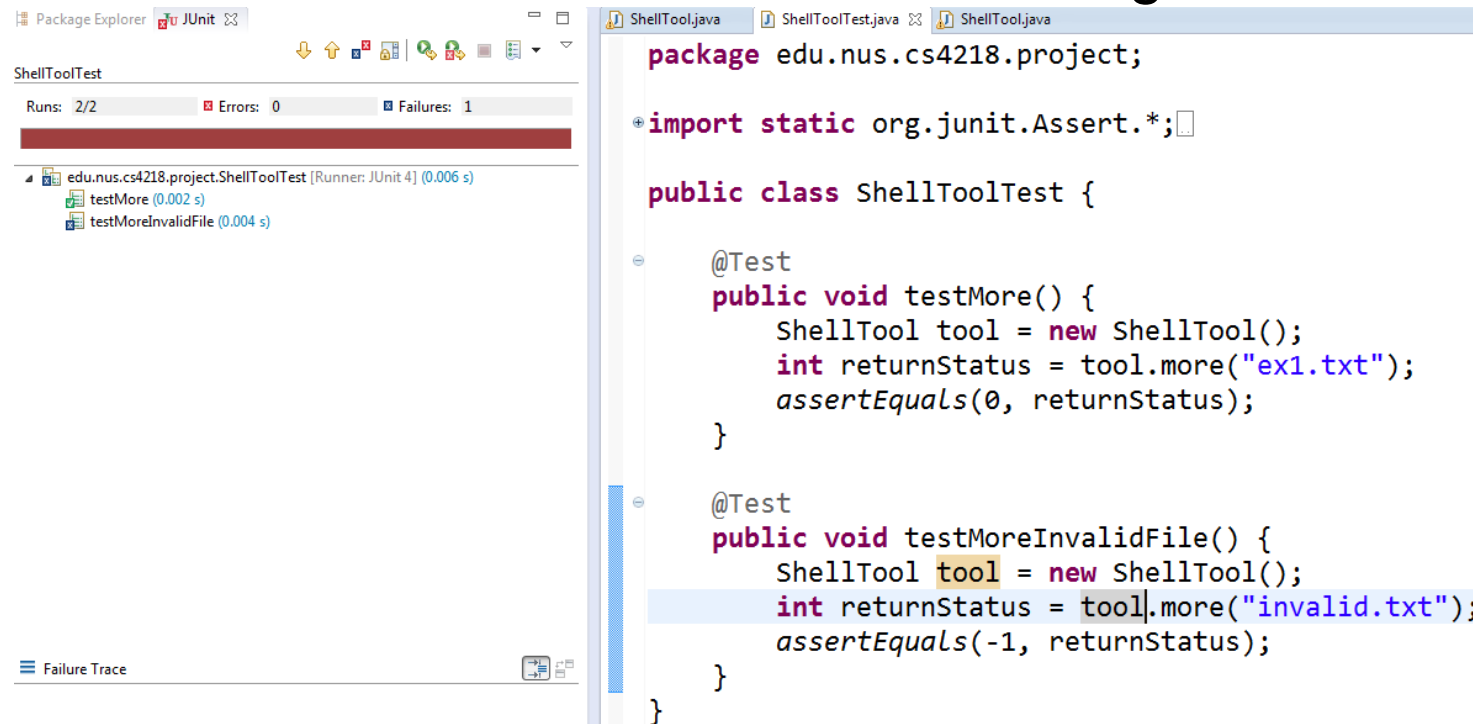
- Add code to make the failing tests **pass**
 - After implementing all the missing functionalities, all failing test cases should now pass



```
/**
 * Shows the first part of a file
 *
 * @param path
 * @return
 */
public int more(String path) {
    BufferedReader in = null;
    try {
        in = new BufferedReader(new FileReader(path));
        String line;
        try {
            while ((line = in.readLine()) != null) {
                System.out.println(line);
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    } catch (FileNotFoundException e) {
        e.printStackTrace();
    }
    return 0;
}
```

Add more tests

- Add more tests for
 - Newly added components or helper methods
 - Checking for corner cases
- Run the new set of tests to see the failing ones



The screenshot shows an IDE with two main panels. The left panel displays the 'JUnit' test runner results for 'ShellToolTest'. It shows 'Runs: 2/2', 'Errors: 0', and 'Failures: 1'. A red progress bar indicates a failure. Below this, a tree view shows the test suite 'edu.nus.cs4218.project.ShellToolTest' with two sub-items: 'testMore (0.002 s)' and 'testMoreInvalidFile (0.004 s)'. The right panel shows the source code for 'ShellToolTest.java'. The code defines a package 'edu.nus.cs4218.project', imports 'org.junit.Assert.*', and defines a class 'ShellToolTest' with two test methods: 'testMore()' and 'testMoreInvalidFile()'. The 'testMore()' method calls 'tool.more("ex1.txt")' and asserts it equals 0. The 'testMoreInvalidFile()' method calls 'tool.more("invalid.txt")' and asserts it equals -1. The 'testMoreInvalidFile()' method is highlighted with a blue selection bar.

```
package edu.nus.cs4218.project;

import static org.junit.Assert.*;

public class ShellToolTest {

    @Test
    public void testMore() {
        ShellTool tool = new ShellTool();
        int returnStatus = tool.more("ex1.txt");
        assertEquals(0, returnStatus);
    }

    @Test
    public void testMoreInvalidFile() {
        ShellTool tool = new ShellTool();
        int returnStatus = tool.more("invalid.txt");
        assertEquals(-1, returnStatus);
    }
}
```

Make them Pass

- Add more code to make the added tests pass
 - After implementing all the helper methods and checking for corner cases, all new failing test cases should now pass

