REASONS TO STOP TESTING

THERE ARE LOTS OF REASONS WHY
YOU MAY WANT TO STOP TESTING.
HERE ARE A FEW...

THERE ARE BUGS EVERYWHERE

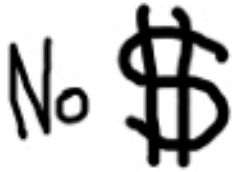
YOU NEED A BREATHER. TAKE A COFFEE BREAK

TIMES UP! RELEASE IT!

ONE BIG MAMA OF A BUG

IT'S HOME TIME

IT'S MILLER TIME. TIME TO PARTY!



NO ONE IS PAYING YOU TO TEST

EVERYTHING YOU PLANNED IS COMPLETE

YOU CAN'T FIND ANY MORE BUGS

THERE'S A NEW FAMILY MEMBER

Of course, your plan might be rubbish, but that's not my problem.

AG

CPEN 321

Testing

Last Class (Recap)

- Black-box vs. white-box testing
- Manual vs. automated testing
- Test-last vs. test-first
- Regression testing
- Systematic testing
  - Cover all user stories
  - Code coverage: statement, branch, path
  - Symbolic execution

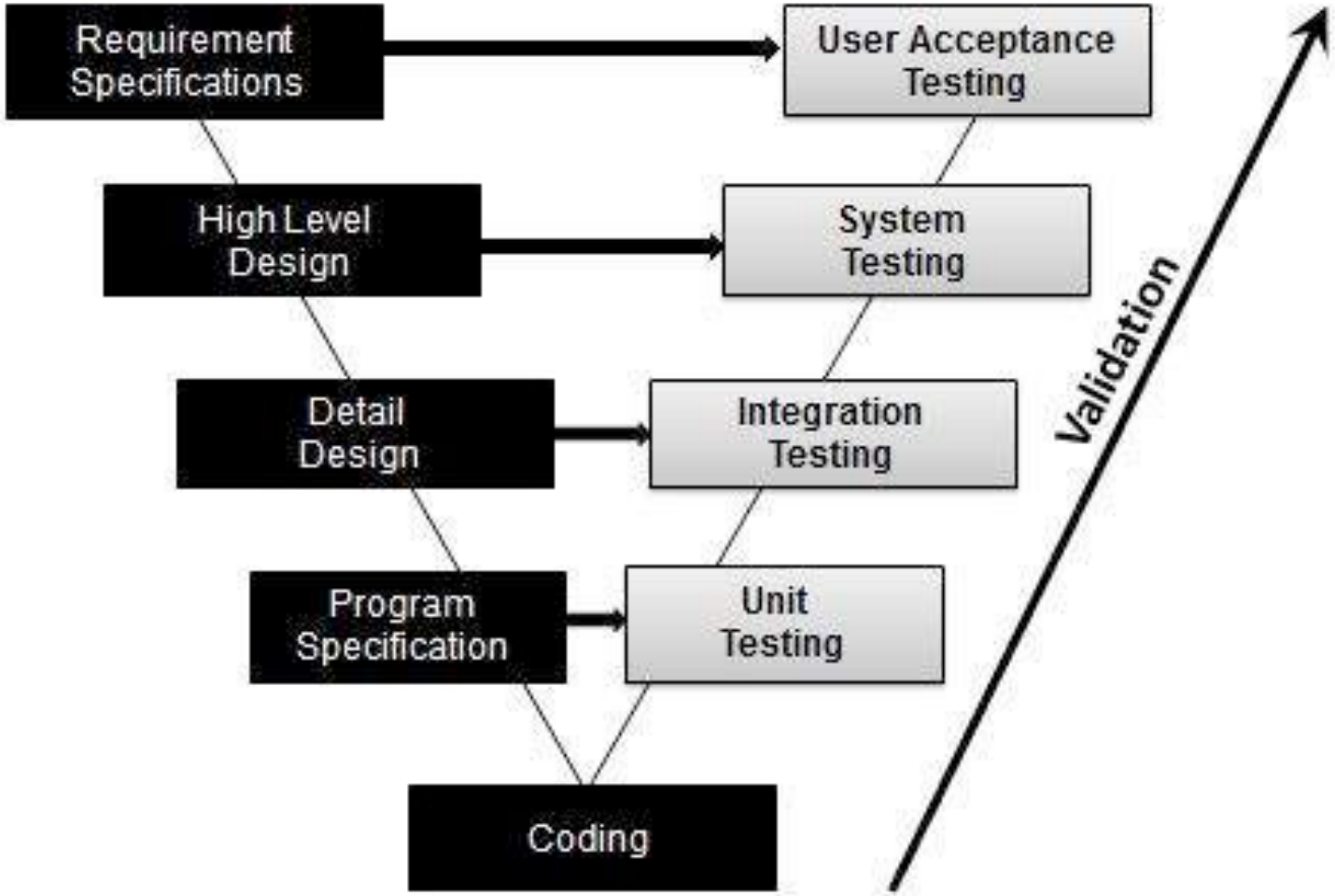
Summary: Testing Best Practices

- When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs (TDD).
- When someone discovers a bug in your code, first write a test case that fails (finds the failure). Then debug and repair the code until the test succeeds.
- Automated regression
- (Usually) statement-level coverage

Agenda

- Main types of testing activities
  - Unit testing
  - Integration testing
  - System testing
  - User acceptance testing
- GUI testing
- Announcements
- Expectations for the Testing Milestone (M5, November 5)

Major Types of Testing Activities



Unit testing

- Tests the behavior of an individual module (method, class, interface) in isolation
- Typically written by developers
- Typically automated

JUnit

- JUnit is a **unit test environment** for Java programs
  - Specifying test cases
  - Executing test cases
  - Pass/fail? (expected result = obtained result?)

- Consists of a **framework** providing all the tools for testing.
  - framework: set of classes and conventions to use them.

JUnit

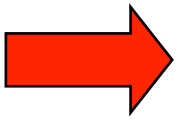
- Test framework
  - test cases are Java code
  - test case = “sequence of operations +inputs + expected results”
- What to do?
  - write a sub-class of **TestCase**
  - add one or more **test methods**
    - Method names starting with “test”: testMean()
  - run

Program code

public int min(…)*{*

//return the minimum

}



Test code

*void **testMin(…)** {*

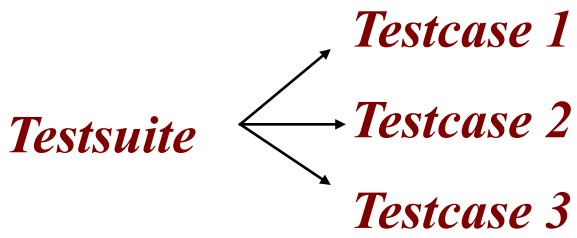
int result= obj.mean(2, 7);

assertEquals(2, result);

}

Framework elements

- **TestCase**
  - Base class for classes that contain tests
- **assert*()**
  - Method family to ***check conditions***
- **TestSuite**
  - Enables grouping several test cases



Assert()*

- Public methods defined in the base class TestCase and used in test methods
- Their names begin with “assert”
  - **assertTrue(“stack should be empty”, aStack.empty());**

- If the condition is true:
  - execution continues normally

- If the condition is false:
  - test fails
  - execution skips the rest of the test method
  - the message (if any) is printed

An example

```
import junit.framework.TestCase;
  public class StackTester extends TestCase {

  public void testIsEmpty() {
    Stack aStack = new Stack();
    assertTrue("stack should be empty",  aStack.empty());
  }

}
```

Assert()*

- for a boolean condition
  - **assertTrue(“message”, condition);**
  - **assertFalse(“message”, condition);**
- for object, int, long, and byte values
  - **assertEquals(expected_value, obtained_value);**
- for objects references
  - **assertNull(reference)**
  - **assertNotNull(reference)**
- **…**

http://junit.org/apidocs/org/junit/Assert.html

Assert: example

```
public void testStack() {
  Stack aStack = new Stack();
  assertTrue("Stack should be empty!", aStack.isEmpty());
  aStack.push(10);
  assertFalse("Stack should not be empty!", aStack.isEmpty());
  aStack.push(4);
  assertEquals(4, aStack.pop());
  assertEquals(10, aStack.pop());
}
```

One concept at a time …

```
public class StackTester extends TestCase {
 public void testStackEmpty() {
    Stack aStack = new Stack();
    assertTrue("Stack should be empty!", aStack.isEmpty());
    aStack.push(10);
    assertFalse("Stack should not be empty!", aStack.isEmpty());
 }
 public void testPushPop() {
    Stack aStack = new Stack();
    aStack.push(10);
    aStack.push(-4);
    assertEquals(-4, aStack.pop());
    assertEquals(10, aStack.pop());
 }
}
```

TestSuite

- **Groups several test cases:**

```
public class AllTests extends TestSuite {

  public static TestSuite suite() {
       TestSuite suite = new TestSuite();
       suite.addTestSuite(StackTester.class);
       suite.addTestSuite(AnotherTester.class);
     return suite;
  }
}
```

Other Unit Test Frameworks

- Java: NUnit, TestNG, …
- C++: Embunit
- …

What about other part of the system?
What about TDD?

Answer: Test Double Objects (Mocks and Stubs)

Purpose:

1. Test partially implemented systems
2. Eliminate dependencies of your system so your tests are more focused on your functionality

A controllable replacement for an existing software unit to which your code under test has a dependency.

Test Double Objects

- **Stubs** provide canned answers to calls made during the test (return constant values).
- **Mocks** are objects pre-programmed with expectations, checking the specification of the calls they are expected to receive (e.g., the right parameters).
- **Fake objects** actually have working implementations, but usually take some shortcut which makes them not suitable for production (e.g., an in-memory database is a good example).
- **Dummy objects** are passed around but never actually used. Usually, they are just used to fill parameter lists.

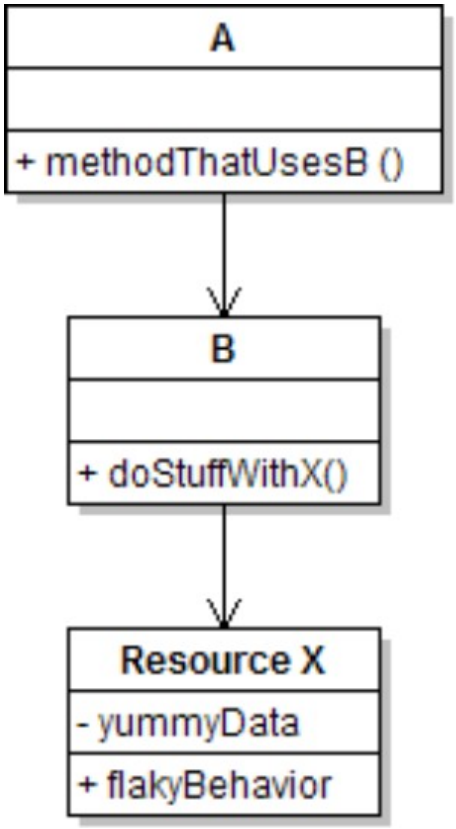
When used?

- Unimplemented code
- Difficult-to-control elements:
  - network / internet
  - time/date-sensitive code
  - database, files, io, threads, memory
  - brittle legacy code /systems

Testing with Stubs: Core Idea

Identify the external dependency.

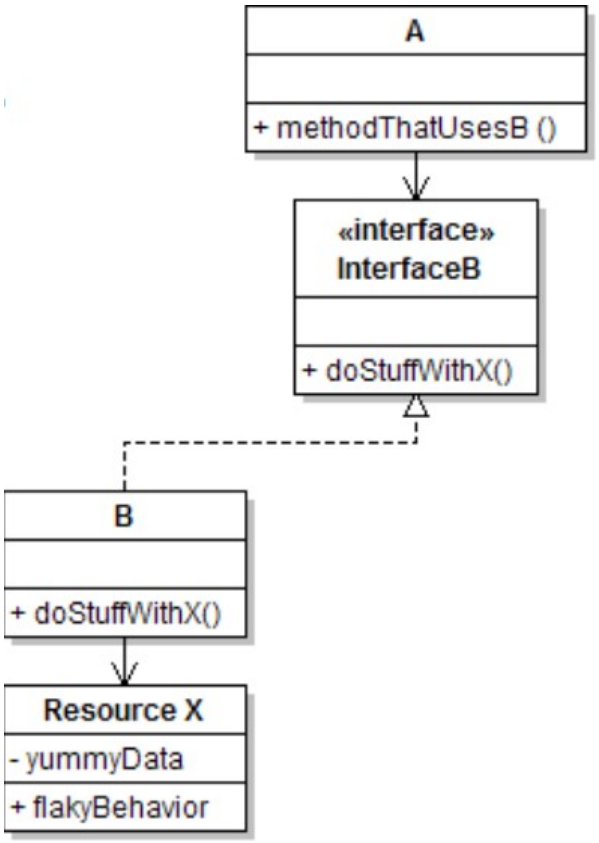
- Say, Class A depends on Class B.



Testing with Stubs: Core Idea

Extract the core functionality of the object into an interface.

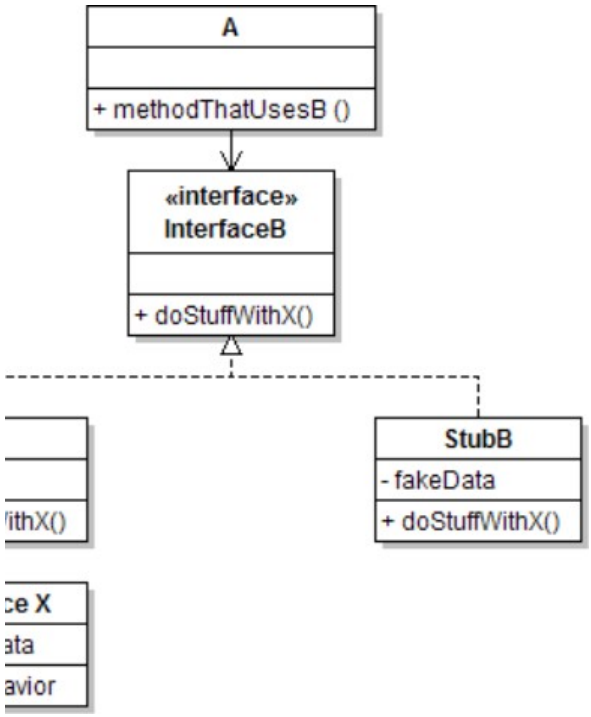
- Create an InterfaceB based on B
- Change all of A's code to work with type InterfaceB, not B



Testing with Stubs: Core Idea

Write a "stub" class that also implements the interface, but returns pre-determined fake data.

- Now A's dependency on B is abstracted away and can be tested easily.
- Can focus on how well A integrates with B's external behavior.



Frameworks for Using Test Double Objects

- Mockito
- Cmocka
- Cgreen
- TestApe
- …

Agenda

- Main types of testing activities
  - Unit testing
  - **Integration testing**
  - System testing
  - User acceptance testing
- GUI Testing
- Announcements
- Expectations for the Testing Milestone (M5, November 5)

Integration testing

- Phase in which individual software modules are combined and tested as a group.
- Approaches:
  - big-bang
  - bottom-up
  - top-down
  - mixed (sandwich)
  - risky-hardest

Big-Bang

Most of the developed modules are coupled together to form a complete software system

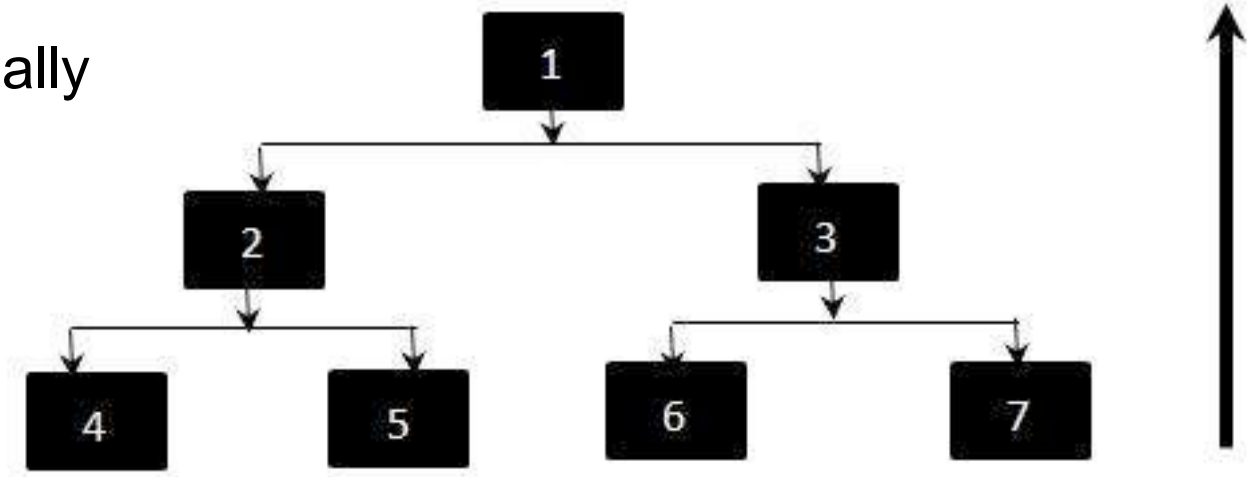
+ effective for saving time in the integration testing process

- failures are hard to pinpoint

Bottom-up

- The lowest level components are tested first
- They are used to facilitate the testing of higher level components.
- Repeat until the component at the top of the hierarchy is tested.
- Helpful only when all or most of the modules of the same development level are ready.

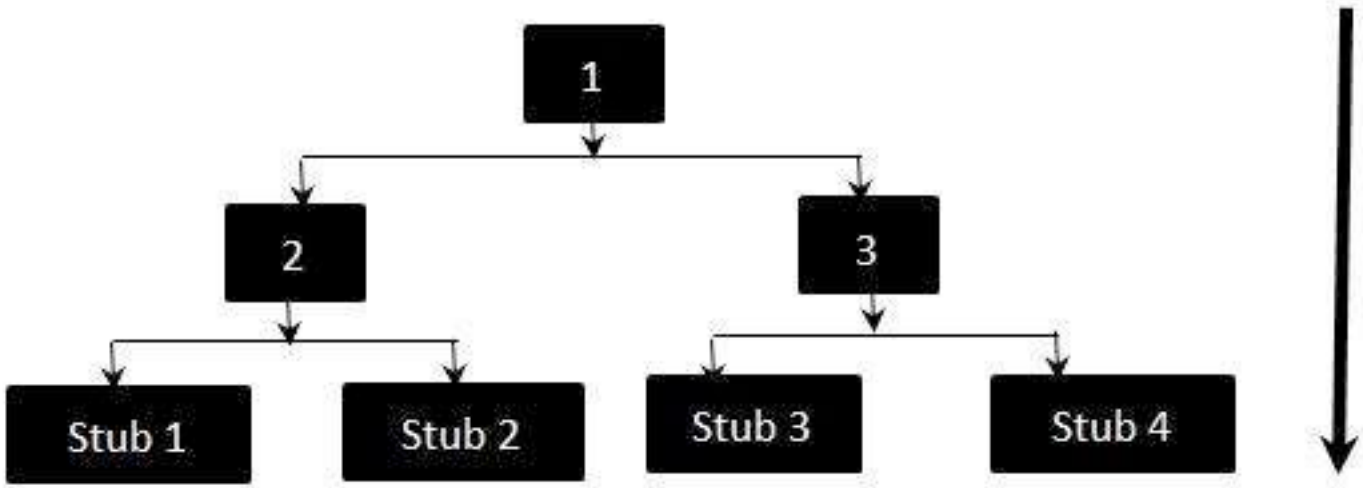
Test 4,5,6,7 individually
Test 2+4
Test 2+5
Test 2+4+5
…



Top-down

- Simulate the behaviour of the lower-level modules that are not yet integrated.

Test 1+2
Test 1+3
Test 1+2+3
…



Mixed and Risky-Hardest

- **Mixed**: combines top-down testing with bottom-up testing.

- **Risky – hardest**: the integration testing is performed starting with the risky and hardest software module first.

Agenda

- Main types of testing activities
  - Unit testing
  - Integration testing
  - **System testing**
  - User acceptance testing
- GUI Testing
- Announcements
- Expectations for the Testing Milestone (M5, November 5)

System Testing

- Tests the behavior of the system as a whole
  - Functional testing (all requirements are met)
  - Usability
  - Installation
  - Performance, load, stress testing
  - Graphical user interface testing
  - …

System Testing

- Tests the behavior of the system as a whole
  - Functional testing (all requirements are met)
  - Usability
  - Installation
  - **Performance**, load, stress testing
  - **Graphical user interface testing**
  - …

Performance

- Performance is a major aspect of program acceptance by users
- Your intuition about what's slow is often wrong
- Measure before optimizing:
  - Runtime / CPU usage
  - Memory usage
  - Web page load times, requests/minute, latency …

Optimization Hints: Think High Level

- Focus on high-level optimizations (algorithms, data structures)
  - Leave the low-level ones to the compiler
- Lazy evaluation saves you from computing/loading
  - don't read / compute things until you need them
- Caching save you from reloading resources
- Combine multiple database queries into one query
- …

Profiling

- Log and monitor!!!
  - Especially for cloud-based systems
- But profiling is expensive and it slows down your code
  - Design profiling to be very short
- The app is only too slow if it doesn't meet your project's stated performance requirements.
  - If it meets them, don't optimize it!

Agenda

- Main types of testing activities
  - Unit testing
  - Integration testing
  - System testing
  - **User acceptance testing**
- GUI Testing
- Announcements
- Expectations for the Testing Milestone (M5, November 5)

Acceptance Testing

- System is shown to the user / client / customer to make sure that it meets their needs.
  - A form of black-box system testing

Beta Testing (by Customers)

Advantages

- Customers test for free!
- Gives you test cases representative of customer use.
- Helps to determine what is most important to the customers.
- Test in real settings other than in lab.

Disadvantages:

- You do not want to exhaust your beta-testers
- Beta testers might have a particular perspective to the system: may result in not catching diverse system bugs.

More about beta testing, dark launching, etc. in the next class

Graphical User Interfaces (GUI) Testing

GUI Testing

- GUI responds to user **events** (e.g., clicks)
  - GUIs are event-driven systems
- GUI interacts with the underlying code by method **calls** or **messages**
- Testing GUI correctness is critical for system usability, robustness, and safety

How is GUI testing different from non-GUI testing?

Non-GUI testing: test cases invoke methods of the system and catch the return value(s)

GUI testing: test case are

- able to **identify** the components of a GUI;
- able to **exercise** GUI events (e.g., clicks);
- able to provide **inputs** to the GUI components (e.g., filling text fields);
- able to test the functionality underlying a GUI set of components (indirectly);
- able to assert the GUI properties to see if they are consistent with the expectations;

Types of GUI-based Testing

- **During acceptance testing**
  - Accept the system
- **As regression testing**
  - Test the system w.r.t. changes

Approaches for GUI-based testing

- **Manual**
  - Based on the domain and application knowledge of the tester
- **Capture and Replay**
  - Based on capture and replay of user sessions
- **Test Generation**
  - Random Event Generator (Android Monkey)
  - Model-based
  - Search-based

GUI Errors: Examples

- Incorrect action flow
  - e.g., e2 should be enabled after e1
- Missing commands
  - e.g., send email is missing
- Incorrect GUI screens/states
  - The absence of mandatory UI components (e.g., text fields and buttons)
  - Incorrect **default values** for fields or UI objects
  - Data validation errors
  - Incorrect messages to the user, after errors
  - Wrong layout (UI construction)
- ….

GUI Testing Challenges

- GUI test **maintenance** is hard and costly
  - Non-deterministic behavior
  - GUIs are dynamic and change
  - Small structural changes can break the test case (**fragility**)
- Measuring **adequacy** of tests is problematic
  - Have we covered all states? How many states are there?
  - Have we covered all events and their combinations?
- Often GUI test automation is **technology-dependent**
  - GUI tests for Android don't work on iOS

Coverage criteria for GUI-based testing

- Conventional code-based coverage does not work well:
  - GUI-based systems rely on third-party libraries. Hard to calculate for pre-compiled libraries as no source code is available.
  - Event-based systems. The number of possible permutations of even sequences is not adequately mapped to code coverage

- **Possible coverage criteria?**
  - Event coverage: all events of the GUI need to be executed at least once
  - Screen coverage: all screens of a particular app are covered at least once
  - Widget coverage: all widgets of a particular app are covered at least once
  - Event-sequence coverage: all sequences of events (bounded loops) are executed at least once
  - Scenario coverage: each user story is executed at least once

Mobile Testing – Automation

- Android:
  - Monkey (crash test)
  - UIAutomator
  - Expresso
  - Appium
  - Calabash
  - ...
- IOS
  - XCTest
  - KIF
  - Appium
  - Calabash
  - ...

Monkey Tester

- Fires random events
- Reports crashes or application errors.
- Struggles to provide appropriate text inputs to text boxes, which restricts it from exploring deeper states of the application
- Low code coverage

Capture and Replay

1. The tester interacts with the system GUI, generating sessions of sequence of mouse clicks, UI and keyboard events;
2. The tool captures and stores the events and the GUI screenshots for each session;
3. The tester can automatically replay the execution by running the script
4. In case of GUI changes, the script must be updated

Suitable to replay the user session: sophisticated scenarios

Capture and Replay: Oracle?

- Difficult to detect faults looking at the GUI (except crash testing)
- An incorrect GUI state can take the user to an unexpected/wrong interface screen or it can make the user unable to do a specific action
- Usually relies on screen diffing (i.e., a regression)
- Some tools produce scripts that can be updated by the tester to include conditions and acceptance criteria

Appium

- Appium calls platform-specific testing API (XCTest for IOS and UIAutomator for Android)

- For example, an Appium Test looks like

```
driver.findElement(userId).sendKeys("xyz");
driver.findElement(password).sendKeys("abc");
driver.findElement(showPassword).click();
driver.findElement(login_Button).click();
```

- findElement is the API to find an element on a screen

- Actions like tap, entering text, etc. are supported

Expresso

- Espresso (by Google) is an instrumentation-based framework
  - uses Android Instrumentation to inspect and interact with Activities under test
- Base Expresso Code to write test steps

```
@Test
public void greeterSaysHello() {
    onView(withId(R.id.name_field)).perform(typeText("Steve"));
    onView(withId(R.id.greet_button)).perform(click());
    onView(withText("Hello Steve!")).check(matches(isDisplayed()));
}
```

- It Finds the view, performs an action on the view, and validates the result
- Provides a record-replay tool that lets you create test cases without writing any code.

Summary

- Testing is one of the most important SE activities!
  - Test, test, test!
  - Write a test for each failure you detect
  - Automate and run regression
- Be systematic
  - Cover all user stories
  - Code coverage: statement, branch, path
- Different type of testing:
  - Unit testing + mocks, integration testing, system testing, acceptance testing
- Augment testing with static analysis
  - E.g., code smell detection

Summary: Tools that help you automate

- Version Control Systems
  - Git, svn
- Build Process
  - Ant, Maven, Gradle
- Testing Frameworks
  - Unit testing: JUnit, Mockito, …
  - GUI testing: Monkey, Expresso, Appium
- Code Coverage
  - Java: Emma, EclEmma
- Continuous Integration (CI) / running regression
  - Travis CI, Jenkins

Agenda

- Main types of testing activities
  - Unit testing
  - Integration testing
  - System testing
  - User acceptance testing
- GUI testing
- **Announcements**
- **Expectations for the Testing Milestone (M5, November 5)**

Quest Lecture

- Monday, October 29
- **Anthony Chu**, Microsoft

- What's a container? Why containers? How do they work?
- Basic Docker demos on local machine
- Container registries
- Deploying a container to the cloud
- Container orchestrators and the problems they solve (Kubernetes)
- Kubernetes demos
- Summary + Q&A

Future Milestones (Updated)

- W4: Development team and the customer discuss the requirements.
- W5: M1 – Requirements (both customer and development teams).
- W6: M2 – Design (development team).
- W8: M3 – MVP (development team).
- W9: M4 – Code review (development teams).
- W10: M5 – Test plan (development team).
- W11: M6 – Refined specifications (development team).
- W12: M7 – Test results + customer acceptance testing (both customer and development teams).

Deliverables for the Test Plan Milestone W10, M5 (Nov 5)

- You will need to perform the following tasks
  1. Backend:
     - Set up a regression execution for your test cases
     - Produce unit and system-level tests (without mocks)
     - Explicitly test each API exposed to the client
     - Measure statement-level coverage
  2. Mobile / GUI
     - Produce an automated test for one main use case (likely from the MVP)

- Deliverables
  - Weekly progress report
  - A description of tools you selected for running regression, backed testing, GUI testing, and for measuring code coverage
  - A description of how you will encode success / failure criteria for backend and GUI tests (test oracle)

Deliverables for the Refined Spec Milestone W11, M6 (Nov 12)

- Your chance to reflect on the process and progress, report on modifications in scope, changes in spec., etc.
  - The new scope should be equivalent in size and complexity to the old one
  - Changes such as “we extended the functionality of feature X and thus dropped feature Y” are acceptable
- Submit:
  - Weekly progress report
  - Updated specifications from M1
  - Updated system design from M2
  - Clearly mark and explain differences from M1 and M2 deliverables
- **Your final project will be evaluate according to this new spec!**
- No report needed if no changes were introduced
  (just state that in the weekly report)

Deliverables for the Test Results + Customer Acceptance Testing Milestone W12, M7 (Nov 19)

- Development team
  - Statistics about the number of test you created, per category (unit tests, system-level test, GUI tests)
  - A log of your automated test execution + status of the tests
  - A report on code coverage
- Customer team
  - A report on 1 buggy execution scenario that you found via an ad-hoc GUI testing
  - The report should focus on a major fault
  - It should contain the execution scenario (sequence of events, with screenshots) and the description of the fault