Testing

Department of Computing Science
University of Alberta

CMPUT 301 – Introduction to Software Engineering
Slides adapted from Dr. Hazel Campbell, Dr. Ken Wong, and Dr. Abram Hindle

Table of Contents

- Why do we test?
- Testing Methodologies
    - Test-driven Development
    - State-based Testing
    - Black Box Testing
- Testing Strategies
- Design for Testing
    - Dependency Injection
    - Mock Object
    - Fixing Bad Example
        - ImageProcessor Example
- The Way of Testivus
- More Information

Why do we test?

Software Defects

- Some terms:
  - Human *errors* can lead to *faults* in work products, which may cause *failures* when running the software
  - Can try to find faults through *testing*, reviews, proof, model checking, code analysis, etc.
  - Some avoid the term *bug*, since it implies something wandered into the code

Examples of Defects

- Actual behavior differing from expected:
  - Algorithmic
    - Code logic does not produce the proper output
  - Overload
    - Data structure unexpectedly filled completely
  - Performance
    - Violates service level agreement
  - Accuracy
    - Calculated result not to the desired level of accuracy
  - Timing
    - Race condition in coordinating concurrent processes

Failure

- AT&T failure (1990):
  - 114 switching nodes of their long-distance system crashed
  - The outage lasted for 9 h, 70 million calls went uncompleted
- Reason:
  - If a node crashes, it tells neighboring nodes to reroute traffic around it
  - A bug in handling this message caused the receiving node to also crash, etc.

Fault in Code

- Root cause:

```
do {
    switch (…) {
    case …:
        if (…) {
            …
            break;
        } else {
            …
        }
    …
    }
} while (…);
```

After expensive testing phase, a small change was made without again retesting

Goal

- Does program P obey specification S?
    - What is P?
    - What is S?

Why Test?

- Goals:
    - Verification
        - Check that requirements are satisfied
    - Not only to *confirm* normal behavior
        - Find problems to *refute* that the program is correct
    - Establish due diligence
        - Evidence in case of product liability litigation
    - Avoid regression
        - Prevent previous problems from reoccurring

Regression Testing

- Goal:
    - To avoid breaking things that should work
        - Collect, reuse, and re-run automated test cases
    - Do regression test after a change or fix
        - Re-run tests to check whether previously passing tests of the system now fail
        - E.g., old defect somehow became unfixed

Approaches

- Reasoning about the state model for P:
    - Typically, a huge number of states
    - Every practical technique must be inaccurate
    - Could *abstract* states
    - Could *sample* states
    - Or both

Approaches

- Abstraction:
    - Often used in static software analysis techniques
        - E.g., model checking P for some specific S
    - Techniques often pessimistically inaccurate
        - May report P is faulty when P is correct

Approaches

- Sampling:
  - Often used in dynamic analysis techniques
    - E.g., testing, profiling
  - Techniques often optimistically inaccurate
    - May report P is correct when P is faulty
    - Testing drives P through a sampling of states, but the samples may not generalize to actual situations

Testing Techniques

- Creating good tests:
    - Test every error message
        - Error-handling code tends to be weaker
    - Test under other configurations
        - Programmers are biased to their own setup

Limits of Testing

- Issues:
    - A program cannot be tested completely
        - Too many inputs and path combinations to cover
    - Testing cannot find all defects
        - Cannot show their absence, just their presence
    - Challenging
        - Testing may be expensive and frustrating
        - Test code itself could add its own defects

Test-driven Development

Automated Testing

- Purpose:
    - Write software to help test software
        - Automation essential to test-driven development and refactoring
- Limitations:
    - Manual testing still need to observe certain problems
        - E.g., strange noises from the speaker, flickering graphics

Automated Testing

- A good automated unit test:
    - Is simple to write and understand
        - Reduces the chance of defects in the test code
    - Runs quickly
        - Allows it to be re-run frequently while developing
    - Is isolated
        - Could run multiple unit tests in parallel
    - Shows exactly what went wrong if it fails
        - Reduce time spent in a debugger

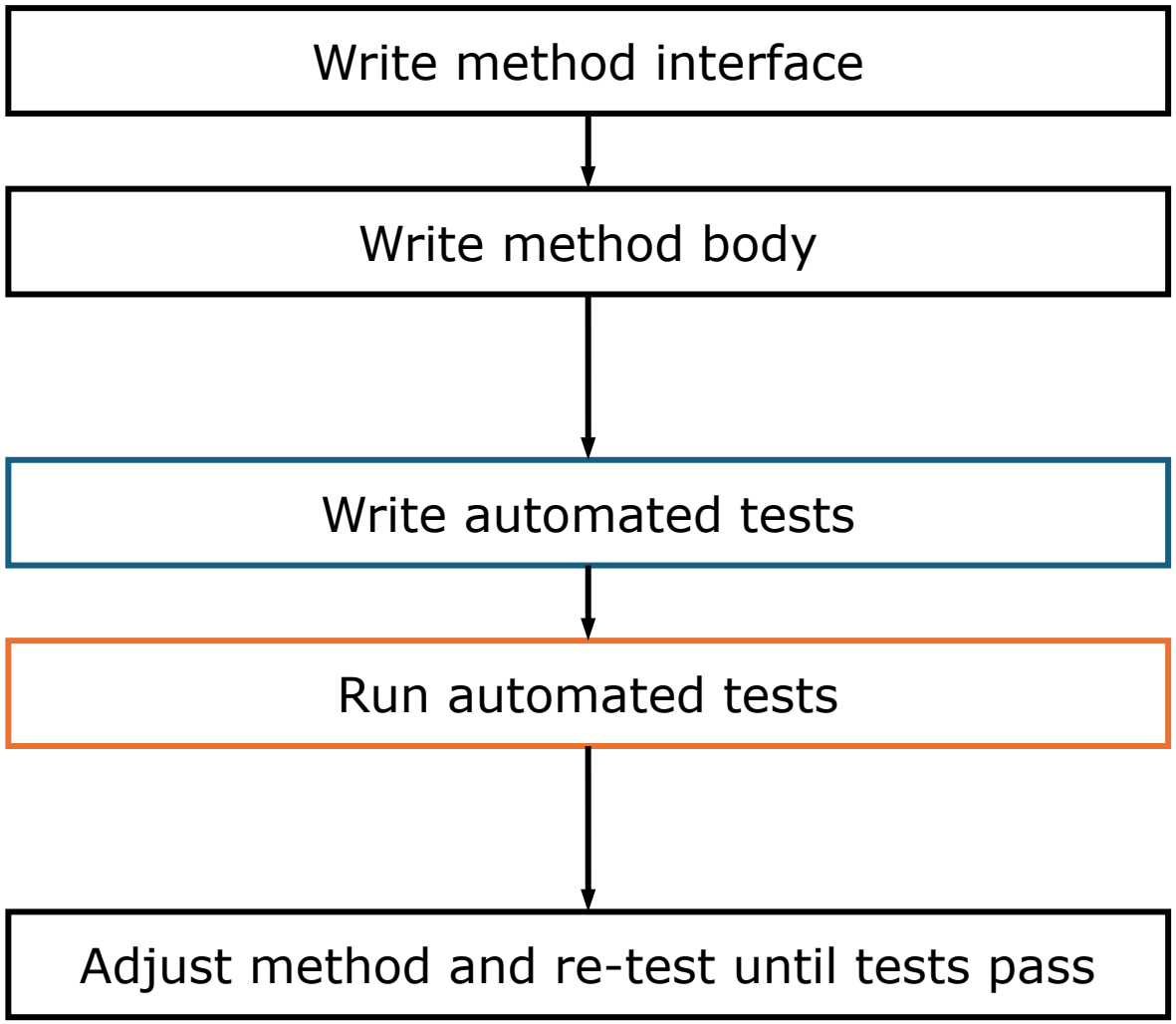
Automated Testing

- “Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.” — Martin Fowler

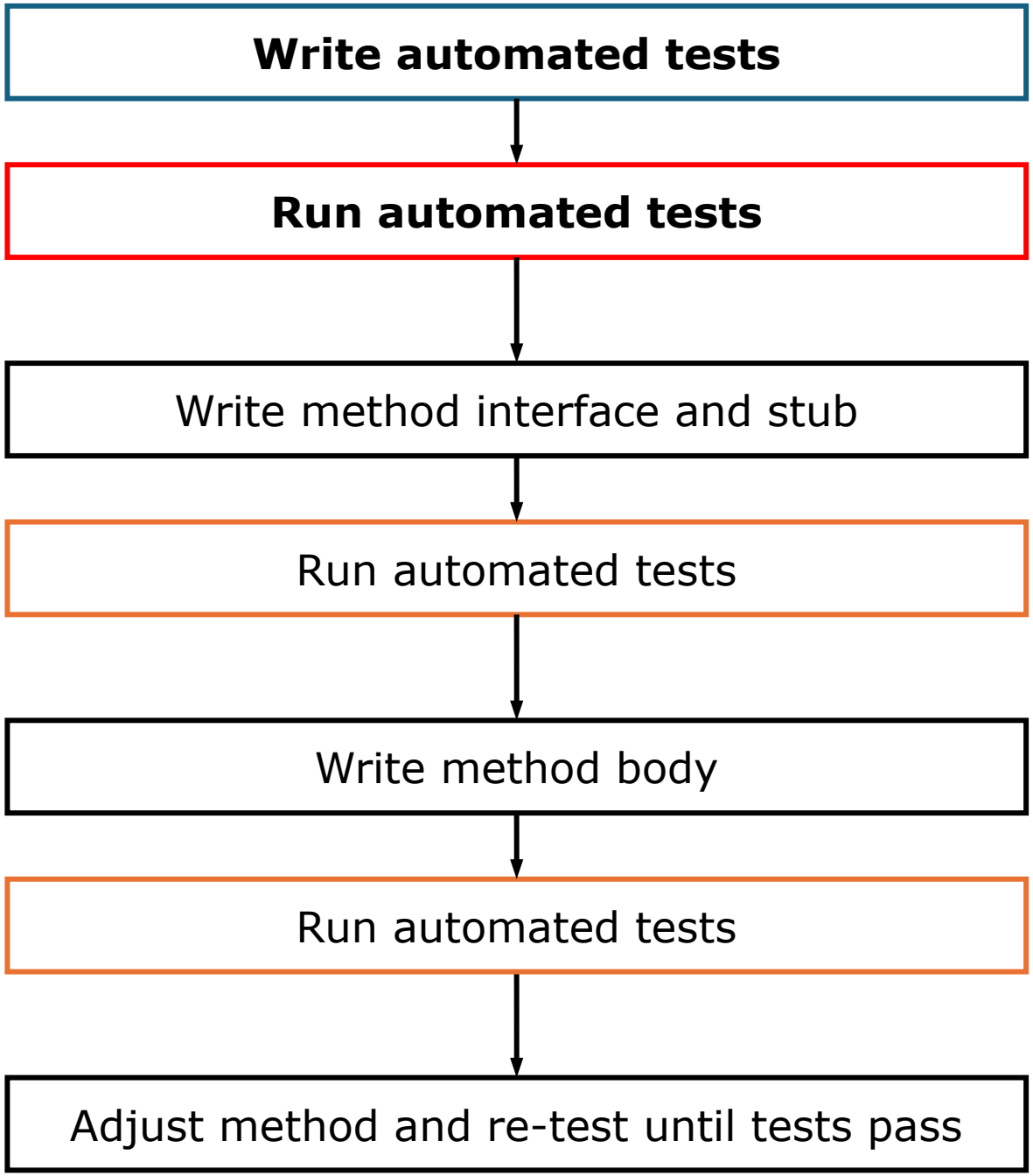
Test-driven Development

- Idea:
  - If testing is so useful, let's write the tests first
  - These automated tests capture *code-level requirements* to be satisfied
  - Once code is written so that these tests pass, then these requirements are met

Traditional development



Test-driven development



← Should fail!

Test-driven Development Example: isEven

- Write automated tests
- Run automated tests: should fail

```
@Test
fun isEven_two_returnTrue()
{
    assertTrue(isEven(2))
}

@Test
fun
isEven_one_returnFalse() {
    assertFalse(isEven(1))
}
```

Test-driven Development Example: isEven

- Write method interface and stub
- Run automated tests

```
fun isEven(n: Int): Boolean {
    return true
}
```

```
@Test
fun isEven_two_returnTrue()
{
    assertTrue(isEven(2))
}
```

```
@Test
fun
isEven_one_returnFalse() {
    assertFalse(isEven(1))
}
```

Test-driven Development Example: isEven

- Write method body
- Run automated tests

```
fun isEven(n: Int): Boolean {
    if (n % 2 == 0) {
        return true
    } else {
        return false
    }
}
```

```
@Test
fun isEven_two_returnTrue()
{
    assertTrue(isEven(2))
}

@Test
fun
isEven_one_returnFalse() {
    assertFalse(isEven(1))
}
```

Test-driven Development Example: isEven

- Adjust or Refactor as needed
- Run automated tests

```
fun isEven(n: Int): Boolean {
    return n % 2 == 0
}
```

```
@Test
fun isEven_two_returnTrue()
{
    assertTrue(isEven(2))
}

@Test
fun
isEven_one_returnFalse() {
    assertFalse(isEven(1))
}
```

Test-driven Development with GitHub Actions

State-Based Testing

State-Based Testing

- Steps:
    - Set up software into a known state
        - E.g., initialize variables
    - Trigger transitions to cause state changes
        - E.g., call methods to change variables
        - E.g., interact with the user interface
    - Verify the actual arrived state is expected
        - E.g., see if actual values in variables meet expectations

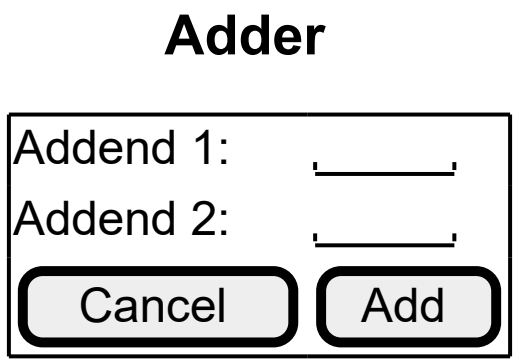
State-Based Testing

- Relating it to the JUnit testing framework:
    - Set up software into a known state
        - E.g., use @Before annotation
    - Trigger transitions to cause state changes
        - E.g., use @Test annotation
    - Verify the actual arrived state is expected
        - E.g., use assertions (assertTrue(…), assertFalse(…), assertEquals(…), etc.)
    - Clean up resources to restart in a clean state
        - E.g., use @After annotation

Black Box Testing

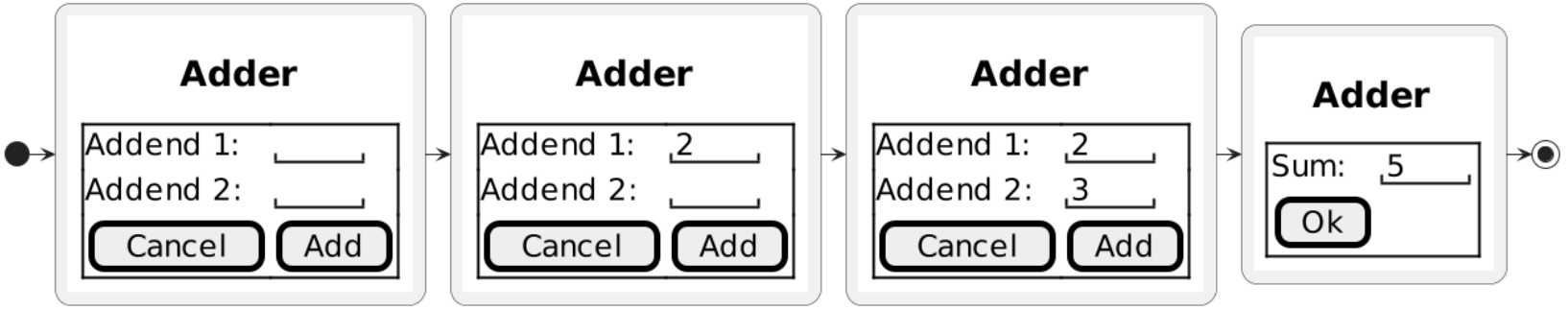
Black Box Testing

- Testing, without seeing the code:



Black Box Testing

- Expected behavior:



  - Deviations from the expected interaction?

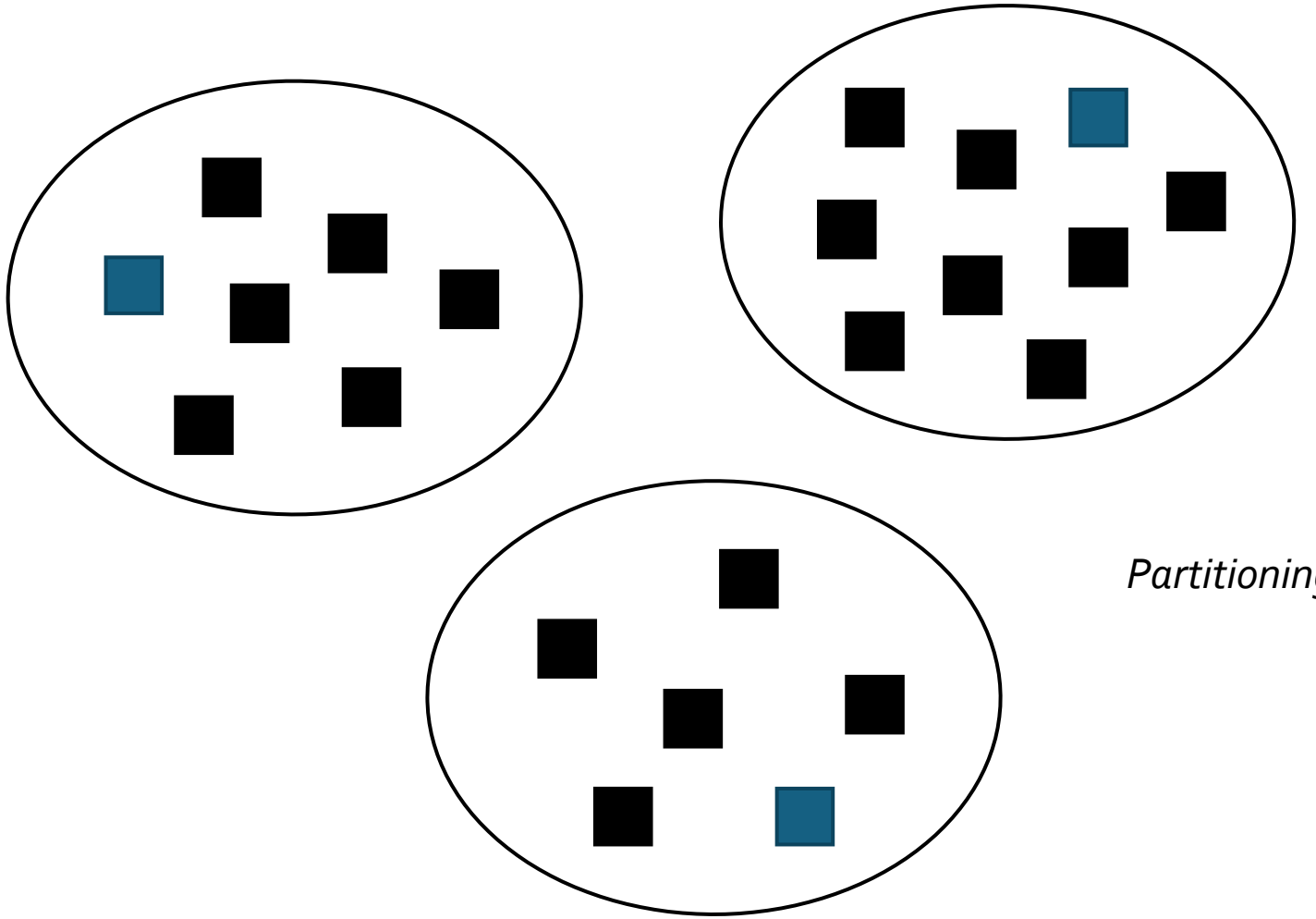
Black Box Testing

- Tips:
    - Want be systematic about what to test
        - E.g., focus on the adder functionality for now
    - Avoid redundant tests
        - Too easy to keep adding meaningless extra tests
    - Determine *equivalence classes* of tests

Black Box Testing

- **Equivalence classes**:
  - Each test inside an equivalence class checks the “same thing”
  - If a test inside the class will catch a defect, the other tests probably also will
  - If a test inside the class will not catch a defect, the other tests probably also will not
  - Keep only a few tests in each class, as representatives

Equivalence Classes



Partitioning of test cases

Black Box Testing

- Example test cases:
  - Be systematic about what to test, not knowing the internal code

	Addends	Sum	Description (also check commutative)
2	3	5	Something simple
99	99	198	Large positive pair
99	-14	85	Large positive plus negative
99	16	115	Large positive plus positive
-99	-99	-198	Large negative pair
-99	-14	-113	Large negative plus negative
-99	16	-83	Large negative plus positive
-99	99	0	Large positive plus large negative
9	9	18	Largest single digit positive pair

Black Box Testing

- Example test cases:
    - Guessing at internal algorithm or representation

	Addends	Sum	Description (also check commutative)
0	0	0	All zero special case
0	23	23	Zero plus positive
-78	0	-78	Negative plus zero
127	127	254	Max signed bytes
-128	127	-1	Min and max signed bytes
-128	-128	-256	Min signed bytes
2147483647	2147483647		Max signed integers
-2147483648	2147483647	-1	Min and max signed integers
-2147483648	-2147483648		Min signed integers

Black Box Testing

- Example test cases:
    - Data input from fields in user interface

	Addends	Sum	Category (also check commutative)
4/3	2		Expression
$2	$2		Currency symbols
+5	3		Addition sign
(9)	9		Parentheses around negatives
l	1		Lower case letter l
O	0		Upper case letter O
<tab>	<tab>		No input
1.2	5		Decimal
A	b		Invalid characters

Black Box Testing

- Example test cases:
    - And even more user interface explorations
        - Editing with delete, backspace, cursor keys, etc.
        - Using F1, escape, and control characters
        - Vary timing of data entry

Testing Strategies

Testing Strategies

- Big-bang strategy:
  - Test thoroughly only after the whole system is put together
  - Pro(?)
    - "Project almost finished, only testing left"
  - Cons
    - Hard to pinpoint the cause of a failure

Testing Strategies

- Top-down incremental strategy:
    - Implement/test the highest-level modules first
        - Provide stubs for lower-level functionality not yet implemented
        - Higher-level modules are the test drivers
- Bottom-up incremental strategy:
    - Implement/test the lowest-level modules first
        - Need to write test drivers

Design for Testing

Good Software Design

- Software should be flexible:
  - Easy to change to respond to new needs
  - Easy to understand
  - Easy to extend, without exploding complexity
- Software should be testable:
  - Easy to construct the units
  - Easy to set up units into desired state
  - Easy to drive code and witness effects

Software design techniques: Dependency Injection, Mock Objects

Dependency Injection

Dependency Injection

- Used to reduce high coupling (lots of dependencies between classes)
- Done by providing required objects from an external source instead of creating them inside the class
- Allows classes to depend on abstractions or interfaces rather than specific implementations

Dependency Injection Example

```
class Wand {
    fun shoot() {
        // do something
    }
}

class Wizard {
    val wand = Wand()
    fun castSpell() {
        wand.shoot()
    }
}
```

NO Dependency Injection

- Suppose the wands instantiated from the Wand class can only shoot fire
- What if we wanted to shoot ice?
- We can't since the Wizard class **creates a specific implementation** of the Wand Class

Dependency Injection Example

```
class Wand {
    fun shoot() {
        // do something
    }
}

class Wizard {
    val wand = Wand()
    fun castSpell() {
        wand.shoot()
    }
}
```

NO Dependency Injection

Problems:

- Difficult to test
- Difficult to change implementation
- Reduced Reusability

Dependency Injection Example

WITH Dependency Injection

```
interface Wand {
    fun shoot()
}

class FireWand : Wand {
    override fun shoot() {
        // do something
    }
}
```

```
class IceWand : Wand {
    override fun shoot() {
        // do something else
    }
}

class Wizard(private val wand: Wand) {
    fun castSpell() {
        wand.shoot()
    }
}
```

Dependency Injection Example

```
interface Wand {
    fun shoot()
}
class FireWand : Wand {    // concrete implementation
    override fun shoot() {
        // do something
    }
}
class IceWand : Wand {
    override fun shoot() {   // another concrete implementation
        // do something else
    }
}
class Wizard(private val wand: Wand) {   // inject Wand
    fun castSpell() {
        wand.shoot()
    }
}
```

WITH Dependency Injection

- We inject wand as a parameter for the Wizard class to allow for more flexibility and testability

Dependency Injection Example

```
fun main() {
    val fireWand = FireWand()
    val fireWizard = Wizard(fireWand)
    fireWizard.castSpell()

    val iceWand = IceWand()
    val iceWizard = Wizard(iceWand)
    iceWizard.castSpell()
}
```

WITH Dependency Injection

- Now different wizards can use different wands!
- Notice how the creation of Wand instances happen **outside** the Wizard class

Mock Object

Mock Object

- We define and then instantiate a hard coded version of an object, which then becomes a mock object
- Mock objects are used in Unit Testing to test behaviors in isolation

Mock Object Example

```
interface Inventory {
    fun getItems(id: Int): String
}

class MockInventory : Inventory {
    override fun getItems(id: Int): String {
        return “Fire Wand”
    }
}
```

```
class MockInventoryTest {
    @Test
    fun testGetItems() {
        val inventory : Inventory = MockInventory()
        val result = inventory.getItems(1)
        assertEquals("Fire Wand", result)
}
```

Mock Object Example – Breaking it Down

```
interface Inventory {
    fun getItems(id: Int): String
}

class MockInventory : Inventory {
    override fun getItems(id: Int): String {
        return "Fire Wand"
    }
}
```

← We want to test Inventory

← So, we create a fake version of Inventory

← We return a *hardcoded* value

Mock Object Example – Breaking it Down

```
1 class MockInventoryTest {
2       @Test
3       fun testGetItems() {
4             val inventory : Inventory = MockInventory()
5             val result = inventory.getItems(1)
6             assertEquals("Fire Wand", result)
7 }
```

- **Line 4**: We create an instance of MockInventory that implements Inventory
- **Line 5**: We pass in the argument 1, but it doesn't matter what integer is passed in as MockInventory's getItems method *should* return "Fire Wand"
- **Line 6**: the assert function is expected to return true

Discussion: Fixing Bad Example

How can we fix this bad example?

```
/*
 * Process photo album requests,
 * parse user preferences,
 * apply image transformations,
 * assemble images into albums,
 * and deliver results to users
*/
class PhotoAlbumServer {
      // lots of code
}
```

How can we fix this bad example?

What's wrong with it?

→ Lots of responsibilities for one class!

- Poor flexibility:
  - Difficult to extract and reuse parts
  - Complex to add new features
  - Instance variables are "global"
- Poor testability:
  - Only end-to-end testing possible
  - Need golden results files for every combination of preference settings and image transformations

How can we fix this bad example?

Improvement Suggestions:

- Use separation of concerns; one class for each responsibility:
    - RequestHandler class
    - UserPreferencesReader class
    - UserPreferencesParser class
    - ImageProcessor Class
    - ImageEffect class
    - ImageTransformer class
    - …
- For each class, use dependency injection and mock objects

How can we fix this bad example?

By implementing the improvement suggestions, we can get:

- Better flexibility:
    - Uses object-oriented design
    - Easier to understand smaller, separate units
- Better testability:
    - More focused tests of each unit
    - Test fixtures easier to provide for each unit
    - Easier to check results

ImageProcessor Example

- Let’s make a small refactor the PhotoAlbumServer “god class”

    - Extract a responsibility from PhotoAlbumServer, by making a ImageProcessor class from it

ImageProcessor Example

```
interface ImageEffect {
    fun getEffect(image: Image): Image
}

class ImageProcessor(private val effect: ImageEffect) {
    fun process(image: Image): Image {
        return effect.getEffect(image)
    }
}
```

Assume that Image is a List<Int>

ImageProcessor Example

```
class FakeBlackoutEffect : ImageEffect {
    override fun getEffect(image: Image): Image {
        return image.map { 0 }
    }
}
```

ImageProcessor Example

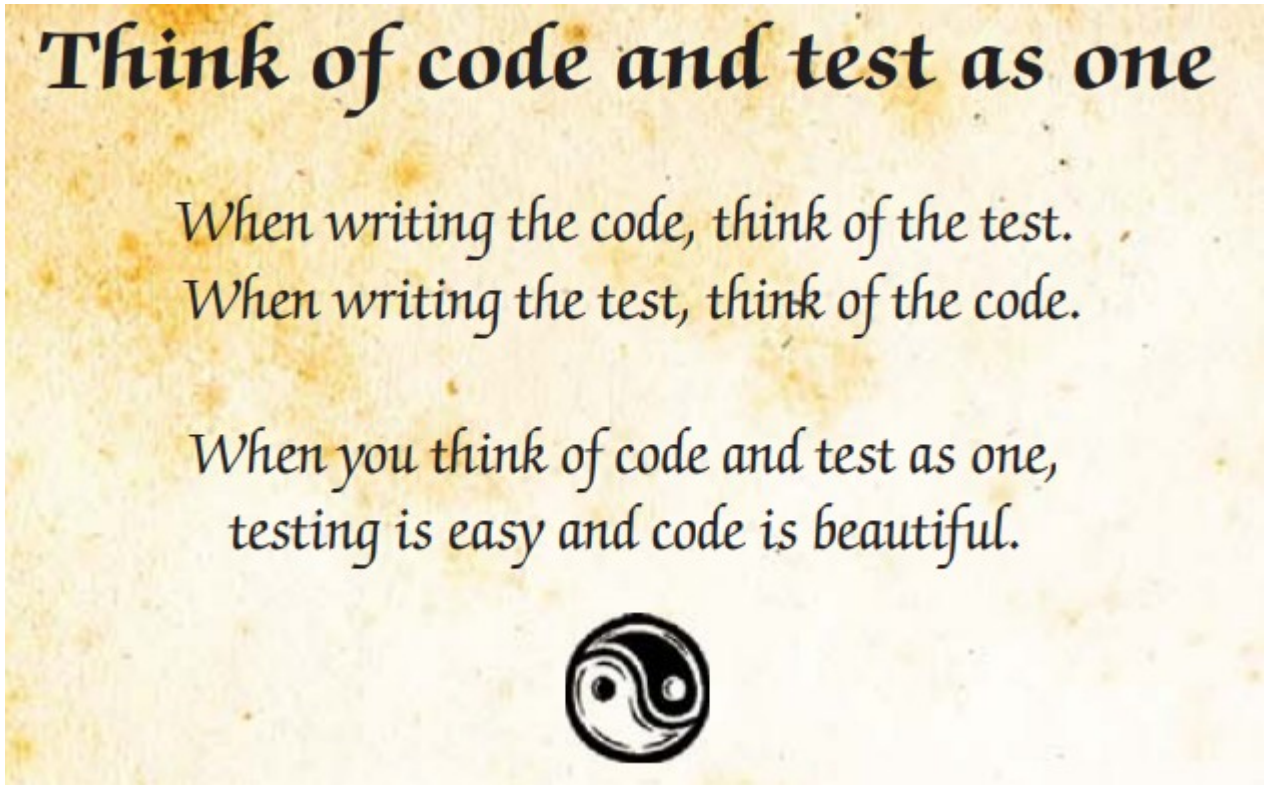
```
class ImageProcessorTest {
        @Test
        fun process_turnWhiteImageIntoBlack() {
            val whiteImage = listOf(255, 255, 255) // 3 pixel image
            val processor = ImageProcessor(FakeBlackoutEffect())
            val result = processor.process(whiteImage)

            assertEquals(listOf(0, 0, 0), result)
        }
}
```

ImageProcessor Example

- ImageProcessor uses dependency injection
  - ImageProcessor depends on the ImageEffect interface, not a specific instance
- In the ImageProcessorTest class, we replaced the real dependency with a mock object

The Way of Testivus



https://www.albertosavoia.com/uploads/1/4/0/9/14099067/thewayoftestivus.pdf

The Way of Testivus

The best time to test
is when the code is fresh

Your code is like clay.
When it's fresh, it's soft and malleable.
As it ages, it becomes hard and brittle.

If you write tests when the code is fresh
and easy to change, testing will be easy,
and both the code and the tests will be strong.

https://www.albertosavoia.com/uploads/1/4/0/9/14099067/thewayoftestivus.pdf

More Information

- Links:
  - Cause of AT&T Network Failure
    - http://catless.ncl.ac.uk/Risks/9.62.html#subj2
  - The Way of Testivus
    - https://www.albertosavoia.com/uploads/1/4/0/9/14099067/thewayoftestivus.pdf
  - JUnit Resources for Test-Driven Development
    - https://junit.org/

More Information

- Books:
  - Test-Driven Development
    - K. Beck
    - Addison-Wesley, 2003
  - Testing Computer Software
    - C. Kaner, J. Falk, H. Q. Nguyen
    - Wiley, 1999
  - Lessons Learned in Software Testing
    - C. Kaner, J. Bach, B. Pettichord
    - Wiley, 2002
  - Flexible Design? Testable Design? You Don't Have to Choose!
    - R. Rufer and T. Bialik