

CSE 403

Lecture 13

Black/White-Box Testing

Reading:

Software Testing: Principles and Practices, Ch. 3-4 (Desikan, Ramesh)

slides created by Marty Stepp

<http://www.cs.washington.edu/403/>

Testing questions

- Should I test my own code, or should somebody else?
- Which code of my project should I test the most/least?
- Can I test all possible inputs to see whether something works?
- How do I know if I've tested well/enough?
- What constitutes a good or bad test case method?
- Is it good or bad if a test case fails?
- What if a test case itself has a bug in it?

JUnit exercise

Given a `Date` class with the following methods:

```
- public Date(int year, int month, int day)
- public Date() // today
- public int getDay(), getMonth(), getYear()
- public void addDays(int days) // advances by days
- public int daysInMonth()
- public String dayOfWeek() // e.g. "Sunday"
- public boolean equals(Object o)
- public boolean isLeapYear()
- public void nextDay() // advances by 1 day
- public String toString()
```

- Come up with unit tests to check the following:
 - That no `Date` object can ever get into an invalid state.
 - That the `addDays` method works properly.
 - It should be efficient enough to add 1,000,000 days in a call.

Test-driven development

- 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.
- Write code to test this method *before* it has been written.
 - This way, once we do implement the method, we'll know whether it works.

Black and white box testing

What is the difference between black- and white-box testing?

- **black-box** (procedural) **test**: Written without knowledge of how the class under test is implemented.
 - focuses on input/output of each component or call
- **white-box** (structural) **test**: Written with knowledge of the implementation of the code under test.
 - focuses on internal states of objects and code
 - focuses on trying to cover all code paths/statements
 - requires internal knowledge of the component to craft input
 - example: knowing that the internal data structure for a spreadsheet uses 256 rows/columns, test with 255 or 257

Black-box testing

- black-box is based on requirements and functionality, not code
- tester may have actually seen the code before ("gray box")
 - but doesn't look at it while constructing the tests
- often done from the end user or OO client's perspective
- emphasis on parameters, inputs/outputs (and their validity)

Types of black-box

- requirements based
- positive/negative
 - checks both good/bad results
- boundary value analysis
- decision tables
- equivalence partitioning
 - group related inputs/outputs
 - based on object state diagrams
- state-based
- compatibility testing
- user documentation testing
- domain testing

Boundary testing

- **boundary value analysis:** Testing conditions on bounds between classes of inputs.
- Why is it useful to test near boundaries?
 - likely source of programmer errors (< vs. <=, etc.)
 - language has many ways to implement boundary checking
 - requirement specs may be fuzzy about behavior on boundaries
 - often uncovers internal hidden limits in code
 - example: array list must resize its internal array when it fills capacity

Boundary example

- Imagine we are testing a `Date` class with a `daysInMonth(month, year)` method.
 - What are some conditions and boundary tests for this method?
- Possible answers:
 - check for leap years (every 4th yr, no 100s, yes 400s)
 - try years such as: even 100s, 101s, 4s, 5s
 - try months such as: June, July, Feb, invalid values

Decision tables

Status	Status of spouse	Age > 65? > 65?	Age of spouse	Blind? spouse	Spouse blind?	Standard deduction amount
Single	—	NO	—	NO	—	\$4,750
Single	—	NO	—	YES	—	\$5,750
Single	—	YES	—	NO	—	\$5,750
Single	—	YES	—	YES	—	\$6,750
Married, filing separate return	Claimed standard deduction	NO	—	NO	—	\$7,000
Married, filing separate return	Claimed standard deduction	NO	—	YES	—	\$8,000
Married, filing separate return	Claimed standard deduction	YES	—	NO	—	\$8,000
Married, filing separate return	Claimed standard deduction	YES	—	YES	—	\$9,000
Married, filing separate return	Did not claim standard deduction	—	—	—	—	\$0
Married, filing joint return	—	NO	NO	NO	NO	\$9,500
Married, filing joint return	—	YES	—	NO	NO	\$10,500
Married, filing joint return	—	YES	—	YES	—	\$11,500

Equivalence testing

- **equivalence partitioning:**

- A black-box test technique to reduce # of required test cases.
- What is it?
- steps in equivalence testing:
 - identify classes of inputs with same behavior
 - test on at least one member of each equivalence class
 - assume behavior will be same for all members of class
- criteria for selecting equivalence classes:
 - *coverage* : every input is in one class
 - *disjointedness* : no input in more than one class
 - *representation* : if error with 1 member of class, will occur with all

White-box testing

Some kinds of white box testing don't involve unit tests:

- "static testing"
 - code walkthroughs, inspections, code reviews
 - static analysis tools
 - Lint (and variants) JiveLint, JLint, [PMD](#), CheckR, JSLint, php -l
 - CheckStyle <http://checkstyle.sourceforge.net/>
 - FindBugs <http://findbugs.sourceforge.net/>
 - code complexity analysis tools
 - PMD, CheckStyle, etc.

Static analysis example

The screenshot shows the FindBugs: Critters application interface. On the left, a tree view shows the bug hierarchy: Bugs (56) > Bad practice (17) > Bad use of return value from method (2) > Method ignores exceptional return value (2) > ClassUtils.writeAndLoadClass(String, String, boolean). The main window displays the source code for ClassUtils.java, with line 218 highlighted: `new File(classFileName).renameTo(new File(classFileName + className + CLASS_EXTENSION));`. Below the code, a bug report is shown: `ClassUtils.writeAndLoadClass(String, String, boolean) ignores exceptional return value of java.io.File.renameTo(File)` at line 218. The bug description states: **Method ignores exceptional return value**. This method returns a value that is not checked. The return value should be checked since it can indicate an unusual or unexpected function execution. For example, the `File.delete()` method returns false if the file could not be successfully deleted (rather than throwing an Exception). If you don't check the result, you won't notice if the method invocation signals unexpected behavior by returning an atypical return value.

<http://findbugs.sourceforge.net>

UNIVERSITY OF MARYLAND

Complexity analysis

The screenshot shows the Eclipse IDE with the PMD plugin. The main editor displays the source code for `TTTGame.java`. The code is as follows:

```
1 package ttt.model;
2 import java.util.*;
3 import com.sun.org.apache.regexp.internal.RE;
4
5 /** Models a tic-tac-toe game board. */
6 public class TTTGame extends Observable {
7     // constants
8
9     /** The tic-tac-toe board's width and height
10     public static final int BOARD_SIZE = 3;
11
12     /** Constants that occupy board squares. */
```

The bottom panels show the Violations Outline and Violations Overview.

Violations Outline

Error Message	Line
Overridable method 'reset' called d...	35
Overridable method 'newGame' cal...	36
System.out.print is used	114
Found non-transient, non-static me...	24
Private field 'players' could be mad...	25
Found non-transient, non-static me...	25
Found non-transient, non-static me...	26
Avoid unused private fields such as...	26
Private field 'moves' could be mad...	27

Violations Overview

Element	# Violations	# Violations/...	# Violati.
TTTGame.java	48	355.6 / 1000	4.0
SystemPrintln	1	7.4 / 1000	0.0
ShortVariable	2	14.8 / 1000	0.1
MethodArgumentCouldBeFinal	(max) 5	37.0 / 1000	0.4
DataflowAnomalyAnalysis	2	14.8 / 1000	0.1
ConstructorCallsOverridableMet	2	14.8 / 1000	0.1
LocalVariableCouldBeFinal	(max) 5	37.0 / 1000	0.4

Path testing

- **path testing**: an attempt to use test input that will pass once over each path in the code
 - path testing is *white* box
 - What would be path testing for `daysInMonth(month, year)`?
 - some ideas:
 - error input: `year < 1, month < 1, month > 12`
 - one month from `[1, 3, 5, 7, 10, 12]`
 - one month from `[4, 6, 9, 11]`
 - month 2
 - in a leap year, not in a leap year

Code coverage testing

- **code coverage testing:** Examines what fraction of the code under test is reached by existing unit tests.
 - statement coverage - tries to reach every line (impractical)
 - path coverage - follow every distinct branch through code
 - condition coverage - every condition that leads to a branch
 - function coverage - treat every behavior / end goal separately
- Several nice tools exist for checking code coverage
 - EclEmma, Cobertura, Hansel, NoUnit, CoView ...

Code coverage example

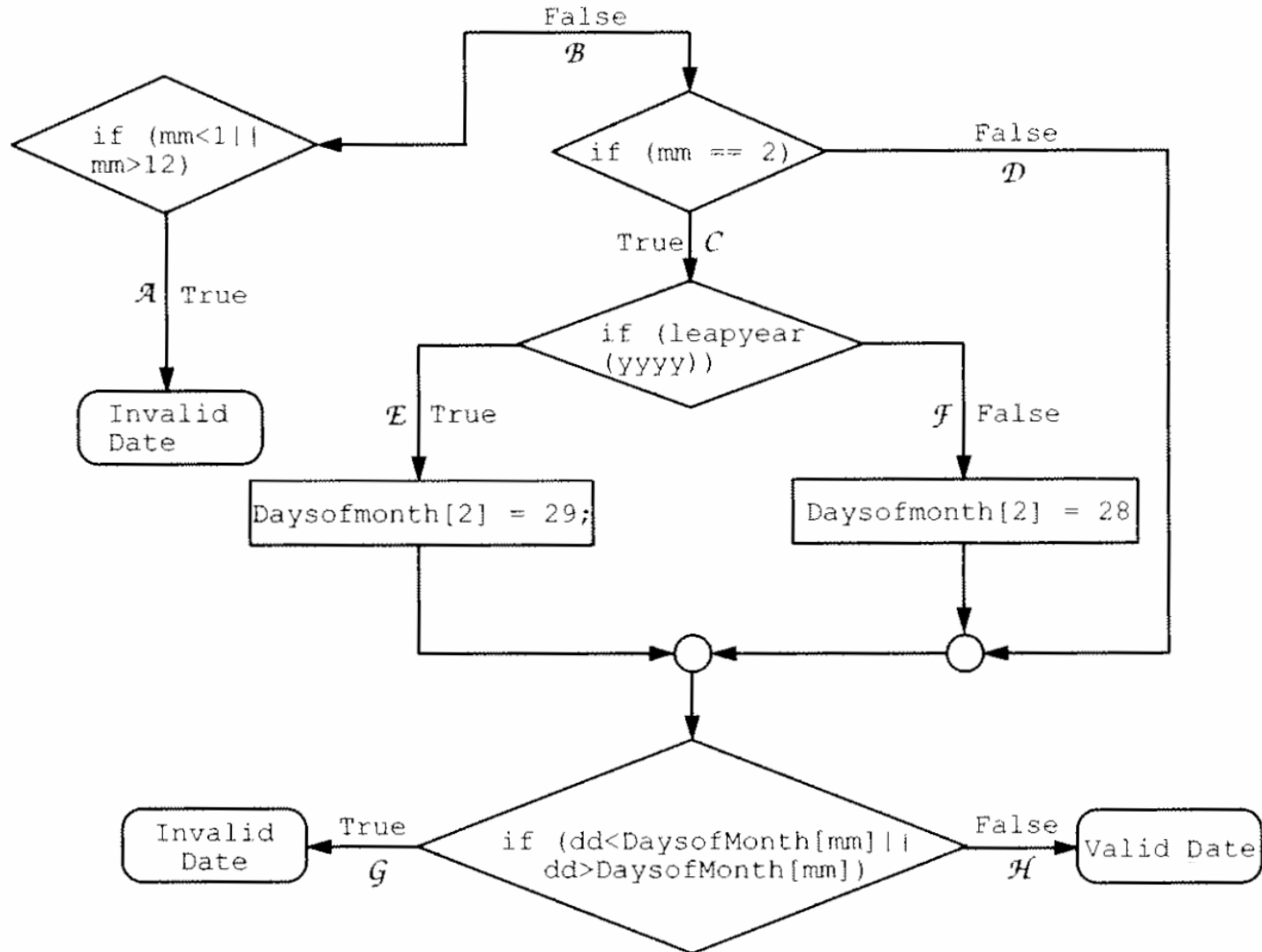
The screenshot shows the Eclipse IDE with the following components:

- Package Explorer:** Shows the test results for DateTest [Runner: JUnit 4] (0.321 s). The tests are: testDateIntIntInt (0.280 s), testAddDays1TrivialSameMonth (0.031 s), testAddDays2WrapMonth (0.031 s), testGetDaysInMonth (0.000 s), testGetDayOfWeek (0.010 s), and testIsLeapYear (0.000 s). The status is "Runs: 6/6", "Errors: 0", and "Failures: 4".
- Code Editor:** Shows the source code for Date.java. The code is highlighted in green, indicating it was executed. The code includes a constructor that throws an IllegalArgumentException for invalid day values and a no-argument constructor that initializes the date to 1970, JANUARY, 1.
- Problems Console:** Shows a warning: "java.lang.AssertionError: expected:<16> to be <17> at DateTest.testAddDays2WrapMonth(DateTest.java:16)".
- Coverage Table:** Shows the coverage for the 'date' package. The table has the following data:

Element	Coverage	Covered Instruc...	Total Instructions
date	52.3 %	203	388

The status bar at the bottom shows "Writable", "Smart Insert", and the time "13 : 27".

Path coverage example



White box testing is hard

- Developers can't easily spot flaws in their own code.
- Test cases that are too focused on code may not be thinking about how the class is actually going to be used.
- Code coverage tools can give a false sense of security.
 - Just because code is "covered" doesn't mean it is free of bugs.
- Code complexity can be misleading.
 - Complex code is not always bad code.
 - Complexity analysis tools can be overly picky or cumbersome.

Testing exercise 1

- Imagine that we have a Date class with working methods called `isLeapYear(year)` and `daysInMonth(month, year)`.
 - Question: What is the pseudo-code for the algorithm for an `addDays(days)` method that moves the current Date object forward in time by the given number of days. A negative value moves the Date backward in time.
 - Question: Come up with a set of test values for your `addDays` method to test its correctness.

Testing exercise 2

- Consider tests to determine whether a Scrabble move is legal:
 - Are all tiles in a straight line?
 - Are the tiles contiguous or separated only by existing old tiles?
 - Are the tiles touching an existing old tile?
 - On each of the words made:
 - What is the score of this word?
 - Is this word in the dictionary?
- Question: What is/are some suitable Scrabble test board configuration(s) and moves that check each of these conditions?
 - Make both passing and failing tests.