Testing

Black box testing, white box testing, testing types/strategies, test plans, test data, execution paths

Testing introduction

- **Testing code** means executing code with the intent of detecting errors.

  *Testing: "the **dynamic** verification of the behavior of a program on a finite set of test cases, **suitably selected** from the usually infinite executions domain, against the **expected behavior.**"*

- **Validation** means checking that the software meets the user requirement.

  Are we building the right product?

- **Verification** means checking that the software does what it does correctly.

  Are we building the product right?

Types of errors

Three different categories of errors to test for:

1. Exceptions. Software runtime errors.
2. Faulty results.
3. Failure to follow agreed upon specifications.

Validation vs Verification

Validation

Are you building the product right?

I asked for a screwdriver with a Phillips head tip. The one you built has only a flat head tip.

Verification

Are you building the right product?

I wanted something to loosen a screw, not pound in a nail.

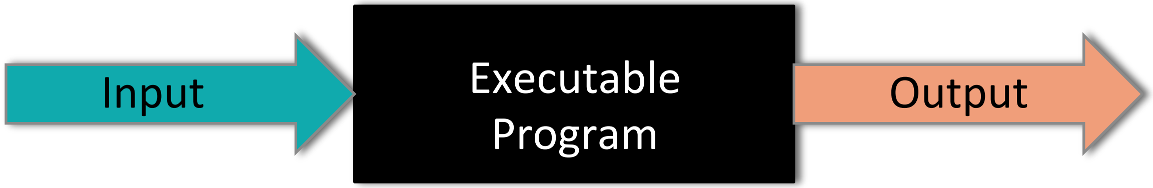
Testing strategies

There are two essential approaches to testing software:

- **Black box.** Testing by specification without knowledge of the underlying code.
- **White box.** Testing according to the underlying code.

Black box testing

- The internal code of the program is ignored. (The program is a "black box" that takes input and produces output.) The tester concentrates only what the program is *supposed* to do.



- Black box testing comprises most high-level testing.
- Black box testing can detect:
  - Missing or incorrect functionality
  - Improper permissioning
  - Bad results
  - Performance errors under load
  - Failure to connect with required data sources (database, web service etc)
  - Poor user interface design
  - Integration errors
  - Initialization or termination errors
  - ...

White box testing

- Test cases are driven by the structure of the code.
- The code should guide the testing. (Creating white box tests causes close code examination.)
- Pay special attention to individual part of the code that could fail (which is often almost everything).
- Tests should cover:
  - Loops at their boundaries
  - Logical decisions as true and as false
  - Independent paths within the source code
  - Internal data structures

Types of testing

- **Unit testing** - testing code that has limited, indivisible functionality.
- **Integration testing** – do the different pieces of software function together properly?
- **Functional testing** – does the software satisfies its functional specifications?
- **Acceptance testing** – does the software meets the agreed criteria agreed upon by the customer and the software team/ company?
- **Performance testing** (stress or load testing) – how does the system behave under stress (many users, large calculations, …)?
- **Regression testing** - retesting previously tested software that has been modified to make sure the changes did not introduce new faults.

Testing process

The testing process follows these general outline:

- Identify what is to be tested. Is it a method? A particular piece of functionality? User interface design?
- Create test cases.
- Create input data for the test cases.
- Determine expected output/behavior from the input data.
- Execute the test cases using the created input data.
- Compare actual and expected outputs.

Unit testing

- Unit testing means testing individual units of functionality of the code. Individual units in Java typically correspond to methods.
- Unit tests can be written *before* the methods they are supposed to test are written, and they can be run *as soon as* the methods are written.
- Unit testing is almost always white box.
- With JUnit, unit testing is automated into the build procedure.
- What to test? Everything that could possibly contain an error.

Integration testing

- Integration testing combines individual software modules tests them together as a group. The focus is on the interaction between classes, typically from composition or inheritance.
- Done by programmers as they integrate their code into code base.
- Generally white box, but may have some black box (such as with closed source libraries)
- Integration testing approaches:
  - **Big bang.** Wait until everything is completed and then start running tests.
  - **Bottom up.** Good for finding low level (and thus easy to elude) bugs. Lowest level modules are tested in conjunction first. The testing proceeds "upward" as higher level modules are completed. This approach is convenient for measuring how far along the testing process has completed.
  - **Top down.** Good for finding missing branches. The most important modules are tested first. As more modules are completed, they are added to the entire system to be tested.

Functional testing

- Functional testing is done on a complete, integrated system.
- Functional testing bases test cases on the design and specifications of the software component under test. Testing can be done for:
  - Completeness (meets all functional requirements)
  - Stress/load testing
  - Usability testing
- Typically done by QA testers who are independent of the rest of the development team.
- Functional testing is always black box.
- We test your assignments by functional testing.

Acceptance testing

- Does the software meet *all* the specifications in the contract between developer and client?
- Acceptance testing is done on a complete, integrated system.
- Typically done by users/customers or customer representatives.
- Also called user acceptance testing (UAT).
- Always black box.
- We test your assignments by acceptance testing (too).

Regression testing

- Done whenever a change has been made to software that has already been tested.
- Regression testing is re-testing.
- Concentration on the part of the software that changed.
- Special attention to verify that the changes did not cause unintended side effects and that the component still satisfies its requirements.
- Must be done whenever there is a change – *even a change in comments.*
- Can be black box or white box.

Performance testing

- Also called load testing or stress testing.
- Runs the software under high loads (many users, much data, etc).
- Usually tool driven and automated.

Beta testing

- End user testing before official release of the software.
- Not scripted. Typically no specially constructed set of test cases.
- Error reporting relies on users performing the tests ("trying out the software").
- Errors can be of any type:
    - Missing functionality
    - Bad functionality/results
    - Poor user interface design
    - Exceptions/data corruption
- Always black box.
- Software marked as "release" has undergone all of its planned beta testing.

Creating test cases

- A **test case** is a specific test for some of the software. Test cases can be specified in documents or in code (as JUnit).
- Each test case requires the following information:
  - *A unique identifier*
    - Black box testing: unique, descriptive identifier in the document.
    - White box testing: the name of the method/assert being tested.
  - *Input data*
    - Exact data to test on. Can be multiple input data for different scenarios.
    - Black box: includes how the user runs the program and views results.
    - White box: inputs to methods that are being tested
  - *Expected output*
    - Exact results expected based on input requirements.
  - *Actual results of running the test case*
    - Black box: user's observations of results.
    - White box: return values from methods or changes of state (detected by other methods)

Black box test plans

- BBTP: a formal document describing black box test cases for a project.
- Each description follows the test case requirements of the previous page, plus the preconditions that describe the state of the program immediately prior to execution.
- Description must be repeatable and with specific values.
- Expected results must have specific values.
- Write black box tests *before* writing the code.

Test ID	Description	Expected Results	Actual Results
Test Name (Test Type)	Preconditions Test Inputs	Expected test outputs	Actual outputs

Example problem

A library classifies its books as those for children and those for adults. It has a strict lending policy as follows:

- Books can be checked out a maximum of two weeks.
- The overdue fine for a children’s book is 2cents/day.
- For an non-children’s book (adult book), the overdue fine is 2 cents/day up to a maximum of 21 days. The fine becomes 5 cents/day for each day after the 21st. After 42 days, the customer must pay the cost of replacing the book.

You are writing a program for customers to estimate their overdue fine based on the type of book and the number of days the book has been checked out.

Testing requirements

Fill in plan for *each* customer requirement.

Test ID	Description	Expected Results	Actual Results
Overdue42	Precondition: The program is running and waiting for customer input. Select book type: Adult Select check out date: 9/9/17 Select check in date: 10/21/17	Amount due: $1.19	
ChildrenOverdue	Precondition: The program is running and waiting for customer input. Select book type: Children Select check out date: 9/9/17 Select check in date: 12/12/17	Amount due: $1.70	

Equivalence classes of test input

- Divide the possible input values into equivalence classes.
- Test *each* equivalence class.
- Test at least one value in the middle of each equivalence class.
- One test can cover multiple equivalence classes:
  - One for each type of input/output
  - The test can focus on one equivalence class, but other values may be needed for a full test.

Equivalence classes by days since check out:

- Invalid: check in date earlier than check out date
- Values 0 - 14
- Values 15 – 21.
- Values 22 – 42
- Values >= 43

Equivalence classes by book type

- Children's
- Adult

Boundary values for test input

- Test at the boundaries of equivalence classes
  - 14
  - 15
  - 21
  - 22
  - 42
  - 43
- "Bugs live at the edges." [Andrew Hunt and David Thomas]

Example boundary values

- String input
  - Null
  - Null string
  - Single character string
  - Exceptionally long string
- Numeric
  - Invalid numbers (according to problem domain)
- Collections: adding or removing
  - Empty collections
  - Add/remove at the beginning
  - Add/remove at the end
  - Add/remove when the collection is at capacity

Test for things that "shouldn't happen"

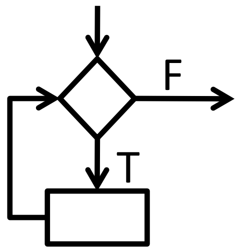
When do things go wrong?

- User errors: Negative numbers when only positive ones make sense.
- User errors: Wrong input type (string rather than number)
- User errors: Failure to enter required fields.
- User malice: Hacking through a required password.
- System failures: Lack of access to a necessary database.

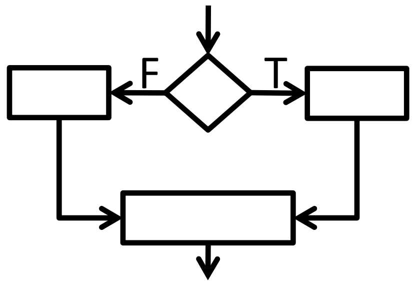
Bottom line: You want to run through the diabolical tests before your customers/graders do.

White box testing

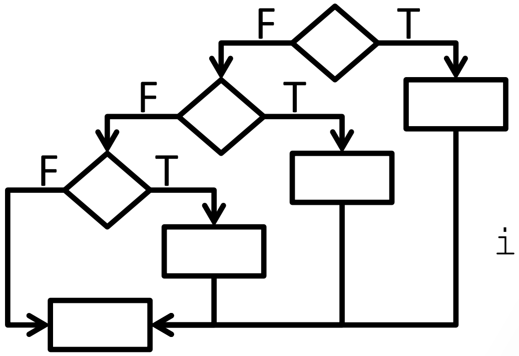
- Test cases depend on the structure of the code.
- Structure? *Test all paths through a method:*
  - Paths branch via if statements and loops.
  - For if statements, test true and false conditions.
- Create control flow diagrams to determine the number of paths:
  - Rectangles: statements
  - Diamonds: decisions
  - Arrows: to indicate flow of control of execution
- Code is easier to test if the conditions are not compound (including loop conditions).
- The more complicated the paths, the more complicated the testing.



`for` and `while` Loops



`if/else` Statement



Nested `if/else/if` Statements

Software Metrics

Software metrics became popular in the 1970s as a way to measure how complicated code is. The more complicated the code, the more difficult to test and the more likely an error.

Cyclomatic Complexity is a software metric that measures of the number of linearly independent paths through code.

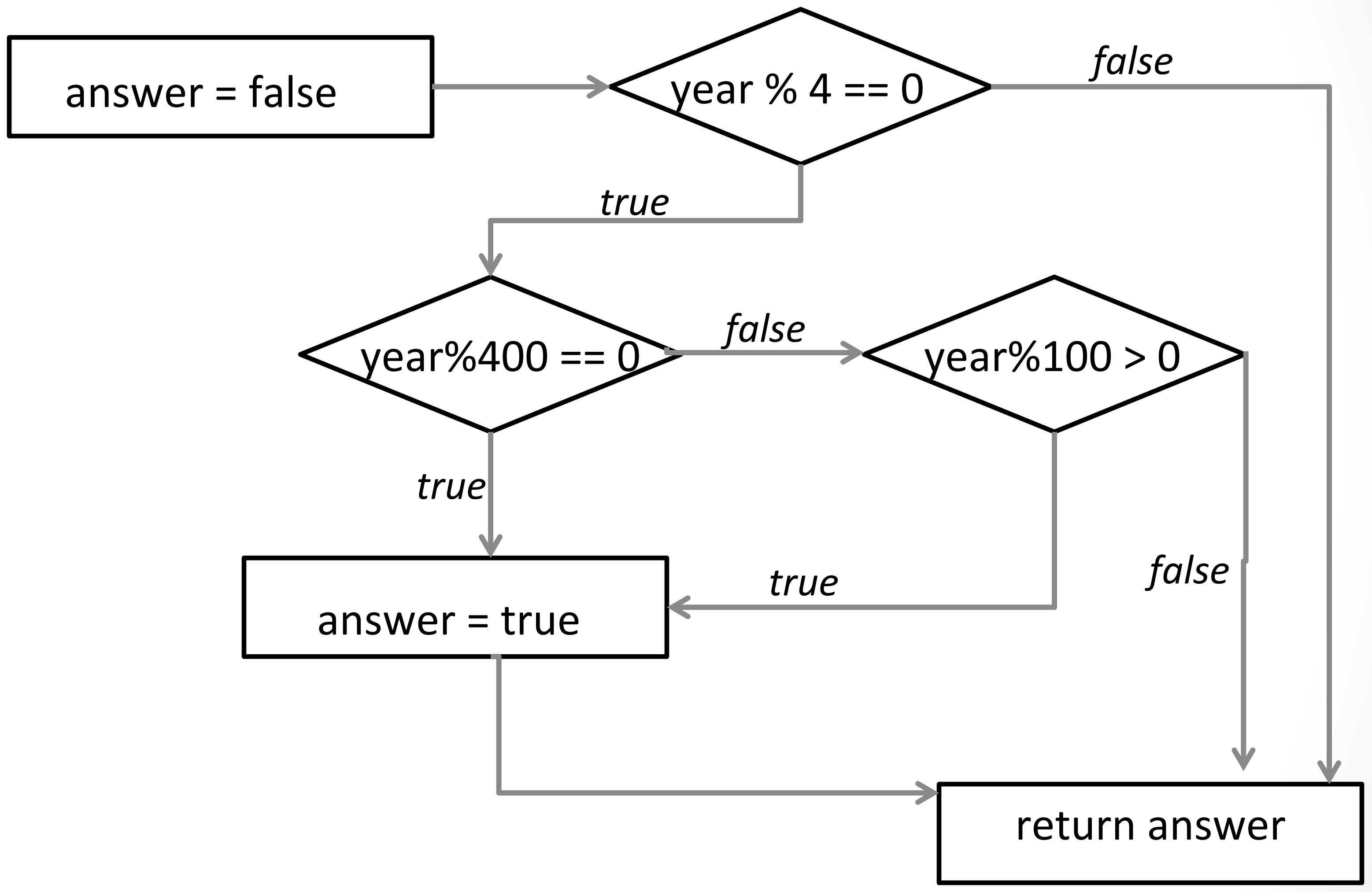
- guide to the number of test cases required for the code.
- To compute, first construct a control flow diagram. Make sure there is exactly one entry point and one exit point.
- Formula: cc = # conditions plus 1

Example: cyclomatic complexity

Compute the cyclomatic complexity of the following method.

```
public boolean isLeapYear(int year) {
   boolean answer = false;
   if (year % 4 == 0) {
       if (year % 400 == 0)
           answer = true;
       else if (year % 100 > 0)
           answer = true;
   }
   return answer;
}
```

Control flow diagram



And what are good test case inputs?

Writing white box test cases

- Use JUnit for automating testing.
- Create a test class for each class that you need to test.
- Create a test method for each public method in the class that could fail, including constructors.
- For each method under test, consider each path through the method. Create multiple test methods for complicated paths.
- For each path through the method, create test data. Consider:
  - equivalence classes
  - boundary values
  - diabolic input
- Always know the intended result for each test. (JUnit enforces this.)

Code coverage

Code coverage is a measure of the degree to which code has been tested. It includes:

- **Method coverage** – Have all methods been called.
- **Statement coverage** – Have all statements been executed?
- **Condition/Decision coverage** – Have all conditions/decisions been executed with both true and false values?

Code coverage tool. **EclEmma** eclipse plugin.

Static analysis

Static analysis means testing software without actually running it. Static analysis includes:

- Code inspections and walkthroughs.
- Identification of code that is likely to be problematic.
- Typically done through software tools.

Static analysis tools

- **PMD**. Looks for potential problems such as:
  - Possible bugs - empty try/catch/finally/switch statements.
  - Dead code - unused local variables, parameters, private methods.
  - Suboptimal code - wasteful String/StringBuffer usage.
  - Overcomplicated expressions.
  - Duplicate code.
- **FindBugs**. Looks for bug patterns.
- **CheckStyle**. Checks code against standards. Highly configurable.
- **Jenkins**. It uses PMD, FindBugs, CheckStyle for:
  - Style: indentation, naming conventions, javadoc, etc.
  - Potential bugs.

References

- Sarah Heckman. CSC 216 slides.