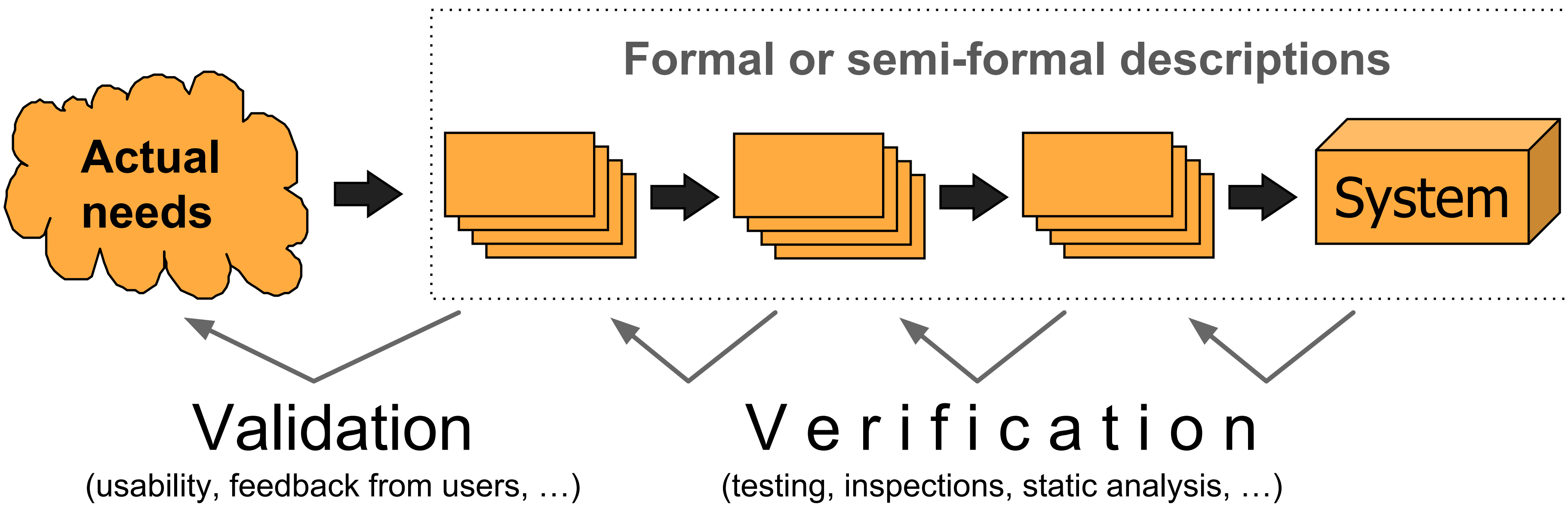
Software Testing

Goal of testing: Verification vs. Validation

- **Validation: Are we building the right product?**
  To what degree the software fulfills its (informal) requirements?
- **Verification: Are we building the product right?**
  To which degree the implementation is consistent with its (formal or semi-formal) specification?



Three stages of testing

- Development testing
  - unit testing
  - component testing
  - system testing
- release testing
  - requirement-based testing
  - Scenario testing
- user testing
  - Alpha testing
  - Beta testing
  - Acceptance testing

Testing techniques

- Testing background information
  - Goals of testing
  - Common Myths
  - Terminology
- Testing Process
- Black-box vs. white-box techniques

Goals of Verification

- Improve software quality by finding errors

  "A Test is successful if the program fails" (Goodeneogh, Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Jan. 85)

- Provide confidence in the dependability of the software product

  (A software product is dependable if it is consistent with its specification.)

Quick Example of Testing

```
function isLeapYear(year)
    if year modulo 400 is 0
        then is_leap_year
    else if year modulo 100 is 0
        then not_leap_year
    else if year modulo 4 is 0
        then is_leap_year
    else
        not_leap_year
```

Testcase 1:

- **Test inputs: 1976**
- **Expected output: true**

Testcase 2:

- **Test input: 1977**
- **Expected output: false**

Did we do a good job testing?

Some Important Facts

Fact 1: It's cheaper to find and fix faults at the unit level -- most faults are incorrect computations within subroutines

Fact 2: It's good to do as much testing early as you can -- you can't test quality in, you have to build it in

Exhaustive Testing

"Many new testers believe that:

- they can fully test each program, and
- with this complete testing, they can ensure that the program works correctly.

On realizing that they cannot achieve this mission, many testers become demoralized. [...] After learning they can't do the job right, it takes some testers a while to learn how to do the job well."

(C. Kaner, J. Falk, and H. Nguyen, "Testing Computer Software", 1999)

Exhaustive Testing - Example

How long would it take (approximately) to test exhaustively the following program?

```
int sum(int a, int b) {return a + b;}
```

1. 2^{32} x 2^{32} = 2^{64} =~ 10^{19} tests
2. Assume 1 test per nanosecond (10^9 tests/second)
3. **W**e get 10^{10} seconds…
4. About 600 years!

Testing Techniques

There exists a number of techniques

- Different processes
- Different approaches

There are no perfect techniques

- Testing is a best-effort activity

There is no best technique

- Different contexts
- Complementary strengths and weaknesses
- Trade-offs

Terminology: Basic

- **Application Under Test (AUT):** software that is being evaluated
- **Test input:** data provided to AUT
- **Expected output:** answer we expect to see
- **Test case:** test inputs + expected output
- **Test suite:** collection of test cases

Terminology: Failure, Fault, Error

Failure

Observable incorrect behavior of a program. Conceptually related to the behavior of the program, rather than its code.

Fault (bug)

Related to the code. Necessary (not sufficient!) condition for the occurrence of a failure.

Error

Cause of a fault. Usually a human error (conceptual, typo, etc.)

Failure, Fault, Error: Example

```
1.   int double(int param) {
2.       int result;
3.       result = param * param;
4.       return(result);
5.   }
```

- A call to double(3) returns 9
- Result 9 represents a failure
- **The** failure is due to the fault at line 3
- The error is a typo

Terminology: Coincidental Correctness

IMPORTANT: Faults don't imply failure - a program can be coincidentally correct if it executes a fault but does not fail

For example, **double**(2) returns 4

Function **double** is coincidentally correct for input 2

Corollary: Just because tests don't reveal a failure doesn't mean there are no faults!

Testing Process

(1) while testing requirements not met

n= select next testing requirement (2)

generate inputs to satisfy n (3)

(4) run program with n, gather output o check that o matches expected output (5)

report thoroughness of testing

(1) When Do We Stop Testing?

- Satisfy the testing criterion or criteria
- Rate of finding faults decreases to an acceptable level
- Exhaust resources---time, money, etc.
- Determine that cost of finding remaining faults is more than fixing faults found after release
- Determine that cost of lost customers greater than cost of dissatisfied customers
- Test Requirements: those aspects of the program that must be covered according to the considered criterion

(2) Test Selection Criteria

- Test Selection Criterion (C): rule for selecting the subset of inputs or requirements to use in a test suite
- We want a C that gives test suites that guarantee correctness…
  - but we settle for a C that gives test suites that improve confidence
- Types of criteria:
  - black-box: based on a specification
  - white-box: based on the code

Complementary

(3) Test Input Generation

Given a testing requirement R, how do we generate test inputs to satisfy R?

Examples:

- Path condition
- Functional requirement
- Missing branch

Figuring out the inputs that correlate with a requirement can be very hard!

(4) Check Output of Test

An **oracle** provides the expected (correct) results of a test and is used to assess whether a test is successful or not.

There are different kinds of oracles:

- Human (tedious, error prone)
- Automated (expensive)

Determining the right answer can be very challenging!

(5) Thoroughness of Testing

Test adequacy criteria: the adequacy score of a testsuite is the percentage of "coverable" items (as defined by the criteria) that are covered by the testsuite.

Examples:

- Execution of statements
- Execution of all paths
- All the customer-specified requirements
- Any test cases we think the professor will check

Some of these are not very good ways to measure thoroughness.

General Approaches to Testing

Black box

- Is based on a functional specification of the software
- Depends on the specific notation used
- Scales because we can use different techniques at different granularity levels (unit, integration, system)
- Cannot reveal errors depending on the specific coding of a given functionality

White box

- Is based on the code; more precisely on coverage of the control or data flow
- Does not scale (mostly used at the unit or small-subsystem level)
- Cannot reveal errors due to missing paths (i.e., unimplemented parts of the specification)

Characteristics of Black-Box Testing

- Black-box criteria do not consider the implemented control structure and focus on the domain of the input data
- In the presence of formal specifications, it can be automated
- In general, it is a human-intensive activity
- Different black-box criteria
  - Category partition method
  - State-based techniques
  - Combinatorial approach
  - Catalog based techniques
  - ...

White-Box Testing

- Selection of test suite is based on some structural elements in the code
- Assumption: *Executing the faulty element is a necessary condition for revealing a fault*
- Example white-box testing criteria:
    - Control flow (statement, branch, basis path, path)
    - Condition (simple, multiple)
    - Loop
    - Dataflow (all-uses, all-du-paths)

Statement Coverage

Test requirements: Statements in program

$$C_{stmts} = \frac{\textbf{(number of executed statements)}}{\textbf{(number of statements)}}$$

Is this a white box or black box test requirement?

Black-box vs. White-box: Example 1

- Specification: function that inputs an integer *param* and returns half the value of *param* if *param* is even, *param* otherwise.
- Function *foo* works correctly only for even integers
- The fault may be missed by white-box testing
- The fault would be easily revealed by black-box testing

```
int foo(int param) {
    int result;
    result=param/2;
    return (result);
}
```

Black-box vs. White-box: Example 2

- Specification: function that inputs an integer and prints it
- Function *foo* contains a typo
- From the black-box perspective, integers < 1024 and integers > 1024 are equivalent, but they are treated differently in the code
- The fault may be missed by black-box testing
- The fault would be easily revealed by white-box testing

```
void foo(int param) {
    if(param < 1024)
printf(“%d”, param);
    else printf(“%d KB”,
param/124);
}
```

Perceiving Software *Quality*

- What does *Quality* mean?
- How can you *see it*?
  - *Poor quality software*
  - *Great quality software*
- Software Quality – two dimensions*:
  - **Functional** → How well the ***behavior*** conforms to design/specifications?
  - **Structural** → How *well* is the functionality ***implemented*** (i.e., architecture/design, code)*?*

****https://en.wikipedia.org/wiki/Software_quality***

Implementing Quality

- How to implement quality at *ground zero?*
  - i.e., the level of an individual class/component
- Say we're implementing a calculator with the four primitive operations:
  - Add
  - Subtract
  - Multiply
  - Divide

Calculator

```
public class Calculator
{
  public int Add(int a, int b)
  {
    return (a + b);
  }
  public int Subtract (int a, int b) { return a – b;}
  public int Multiply (int a, int b) { return a * b;}
  public int Divide (int a, int b) { return a/b;}
}
```

Quality of *Add(…)*

- How can we ensure *Add* works correctly?
- What does "*correctly*" mean?
  - Where does the definition of correctness come from?
- *Examples* of *correct behavior*:
  - 3 + 3 = 6 (and other such cases)
  - 2 + (-2) = 0
  - 5 + (-6) = -1
  - …

How to Test Add?

```
public class CalculatorTest
{
    public void TestAddPositiveNumbers()
    {
        int a = 1, b = 2;
        Calculator calc = new Calculator();
        int sum = calc.Add(a, b);
        Assert.True(sum == 3)
    }
    …TestAddNegativeNumber
    …
}
```

Good, but not great ☹

How to **Write Better** Tests for Add?

public class CalculatorTest

{ //Assume *public void* ...

-WhenAddingTwoPositiveNumbersThenSum ShouldBePositive()

-WhenAddingTheSameNumberAsPositiveAnd NegativeThenSumShouldBeZero()

-WhenAddingPositiveNumberToHigherNegative NumberThenSumShouldBeLessThanZero()

}

What's the Difference?

- Tests read more like specifications
  - Can have traceability to requirements
- They focus on *behavior* that needs to be verified
- Easier maintenance owing to clearer intent
  - Numerous tests over time are difficult to manage
  - Minimize surprises when reading a test
    - Just read method name and you'll know exactly what it does
    - No need to look at the code implementation!

Is Add of High Quality?

- Can you break Add?
  - 1 + 0.5 = 1.5 (*Our calculator will return 1*)
  - 1 + (-0.5) = 0.5 (*Our calculator will return 0*)
  - 999999999999 + 1 (May throw an exception!)
  - (1 + 2i) + (3 + 4i) (*Should* it work?)
  - …
- Similarly, can you break Subtract, Multiply, Divide?
  - Divide by zero?
  - Multiplication overflow?

What Did We Learn?

- It's possible to *test for implementation quality* of the expected behavior
- *Naming* the tests is as important as *writing* the test
- Only testing what is asked for may not be sufficient:
  - Possible that functionality may be missed
  - Possible that functionality not deemed appropriate
  - Must think of ways of breaking the *expected* behavior (to identify the above)

Reminder

- Two general types of testing
    - Black-box
        - Test requirements are the requirements in the software requirements specification*
    - White-box
        - Test requirements are based on the structure of the code
- Techniques are complementary

Tool Overview

- White-box testing
  - Commonly done as “unit testing”
  - *You will explore some WBT tools in your Assignment 4*

- Black-box testing
  - Encode requirements as test cases
  - *You will explore some BBT tools in your Assignment 4*

Basic Parts of a Test

- Setup: things to do before the test starts
  - Also called “given” or “arrange”
- Actions: things that will be done by test
  - Also called “when” or “act”
- Check results: is response correct?
  - Also called “assert,” “assertions,” or “then”
  - Requires oracle

Unit Testing

- The act of testing the smallest possible *unit* of code
  - Usually a single class in OOP (Java, C#, PHP etc.,)
  - Mostly *white-box testing*
- The test tests that unit ***in complete isolation***
  - If you have to fire the browser to test your class you’re doing it wrong
  - Still wrong if you need *a network connection*
  - It’s wrong even if you need a database
  - It’s still wrong if you need a bunch of other classes

Why do Unit Testing

- It may seem time consuming first, but pays for itself many times over:
  - How to know if your *small change* didn't break anything?
  - How to know if your *big change* didn't break anything?
  - How to know if you *preserved existing behavior*?
  - What if you ran all unit tests daily (1x) or on every check in?

Unit Testing Tips

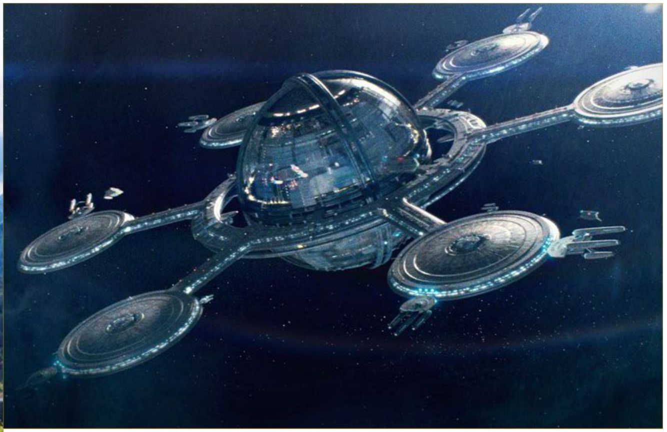
- Write a test whenever you want to use print statement
- Keep unit tests small and fast
  - Ideally run test suite before every code check-in
- Unit tests should be fully automated
- Cover boundary cases
  - E.g.: for numbers test –ve, 0, +ve, NaN, infinity
- Cover as many paths in the program as possible
- Remember: arrange/given, act/then, assert/then

Testing Requirements

- Can we take the learning from unit-testing and apply it to requirements?
    - YES!! ☺
- How come?
    - By focusing on the ***behavior!***
- If the requirements can be specified unambiguously their intended behavior can be correctly validated!
    - **Big "IF"**

Two-Minute Exercise

Create a means for protecting a small group of human beings from the hostile elements of their environment



Ambiguity in Requirements

- A major source of headache and cost overruns
- Diverse interpretations of the same requirement
- Cost of ambiguity:

Phase in which found	Cost Ratio
Requirements	1
Design	3-6
Coding	10
Development/Testing	15-40
Acceptance Testing	30-70
Operation	40-1000

Sources of Ambiguity

- Observational & recall errors:
  - (Not) seeing the same things or retaining what you saw
- Interpretation errors
  - What did “points” refer too?
- Mixtures of above
- Effects of human interaction
  - Things lost in conversation

Stifling Ambiguity by Examples

- What if requirements were *detailed* using examples?
- What if those examples could auto compile into test cases that can be executed?
- Specification by examples forces thinking about *(un)expected behavior*
- Helps resolve ambiguity early on → Higher quality requirements!

Given-When-Then Specifications

Calculator (shall) Requirement:
The system shall add two numbers and provide their sum as result

<u>Scenario: Add two positive numbers</u>

Given two numbers “3” and “2”

When I add them

Then I get “5” as the resultant sum

<u>Scenario: Add two negative numbers</u>

GWT Scenario definition

…

Gherkin

- The GWT syntactical specification is known as the *Gherkin Language*
- *Given, When, Then, And, But etc.,* are keywords
- Adds simple structure to English specifications that can be auto converted to test cases!
- Used as part of the Behavior Driven Development (BDD) Paradigm
    - BDD = Specification by example using GWT

Example Requirement

Feature: **AddTrueFalseQuestion**

As a survey admin I can add a true/false question to the prescreening question bank

@some-tag

Scenario: Add true/false question

Given I am on the Prescreening question bank page

When I create a true/false question titled “Do you speak English?"

And it has the following details:

response	weight	iscorrect
true	10	yes
false	0	no

And belongs to any category

And I save it

Then the question titled "Do you speak English?" should be shown in the list of questions on the Prescreening question bank page

Automated Testing of Requirements

- The method stubs can be filled with code that “tests” the requirement by driving the system
- They typically test the entire “flow” of the application and not a single unit
- They help confirm that the behavior conforms to the requirement specifications – a.k.a., black box testing
- Typically slower than unit tests and may be run less frequently
- Insanely valuable!

Quiz

- How do we generate test input to satisfy a requirement?
- When do we stop testing?
- What is Unit testing and why do we do it?
- What are the sources of ambiguity in requirements?