CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Lecture 4: Testing Fundamentals

Gregory Gay
DIT635 - January 31, 2020

Verification and Validation

- Verification - the process of ensuring that an implementation conforms to its specification.
  - AKA: Under these conditions, does the software work?
- Validation - the process of ensuring that an implementation meets the users' goals.
  - AKA: Does the software work in the real world?
- Proper V&V produces *dependable* software.
  - **Testing is the primary verification activity.**

We Will Cover

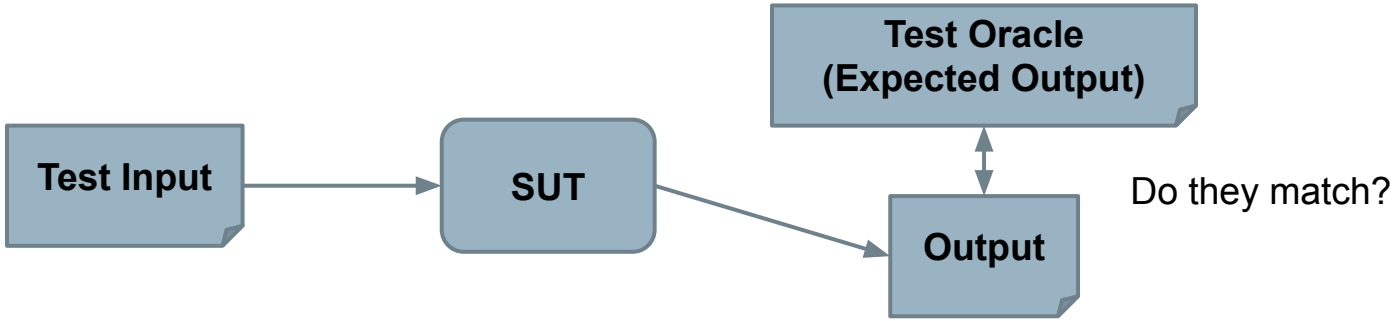
- What is testing?
- Testing definition:
  - Let's get the language right.
  - What are the components of a test?
- Principles of analysis and testing.
- Testing stages:
  - Unit, Subsystem, System, and Acceptance Testing

Software Testing

- An investigation conducted to provide information about system quality.
- Analysis of *sequences* of **stimuli** and **observations**.
  - We create **stimuli** that the system must react to.
  - We record **observations**, noting *how* the system reacted to the stimuli.
  - We issue **judgements** on the *correctness* of of the sequences observed.

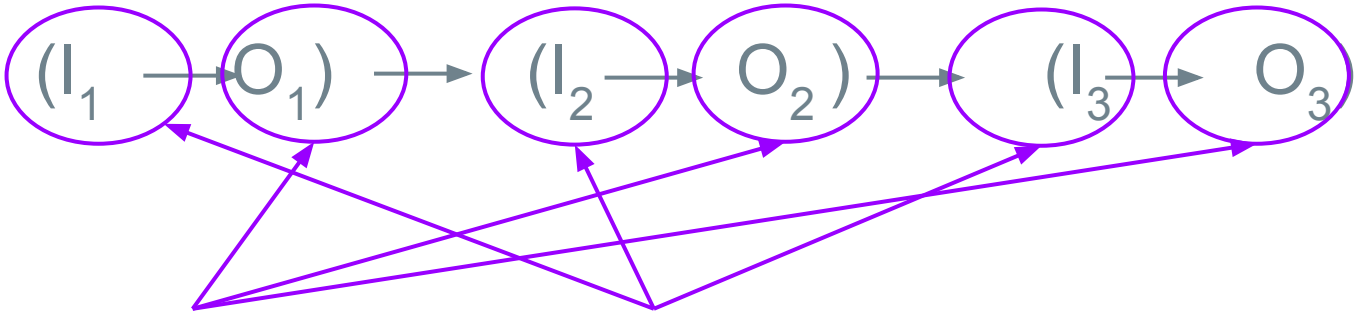
The Basic Process

During testing, we instrument the **system under test** and run **test cases.**



- Instrumentation allows us to gather information *about* what happens during test execution.
  - Variable values, timestamps, and more.
- Test cases consist of sequences of **stimuli** and **observations**, and conclude by issuing a **verdict**.

Anatomy of a Test Case



if O_n = Expected(O_n)
then… Pass
else… Fail

Test Inputs

How we "stimulate" the system.

Test Oracle

How we check the correctness of the resulting observation.

Anatomy of a Test Case

- Initialization
  - Any steps that must be taken before test execution.
- Test Steps
  - Interactions with the system, and comparisons between expected and actual values.
- Tear Down
  - Any steps that must be taken after test execution.

Test Input

- Interactions with a software feature.
- Many means of interacting with software:
  - Most common: a method call + pre-chosen values
    - `trySomething(2,3);`
    - API call also common
  - User interface interactions
  - Environment manipulation
    - Set up a database with particular records
    - Set up simulated network environment
    - Configure timing for real-time systems

Test Input

- Can be inputted manually by a person or (preferably) by running executable tests.
  - Most languages have unit testing frameworks (JUnit)
  - Frameworks for manipulating web browsers (Selenium)
  - Capture/replay tools can re-execute UI-based tests (SWTBot for Java)
  - Fuzzing tools can generate input automatically (AFL, EvoSuite)

Test Oracle - Definition

If a software test is a sequence of activities, an **oracle** is a predicate that determines whether a given sequence is acceptable or not.

An oracle will respond with a *pass* or a *fail* verdict on the acceptability of any test sequence for which it is defined.

Test Oracles and Specifications

- An oracle is an *implementation* of a *specification*.
  - Requirements or design state expectations on how the system should work.
    - … but usually in writing, not in a form that we can run or check
  - Test oracles are code that can be executed that can check whether the software meets the specification.

Oracles are Code

- Oracles must be developed.
  - Like the project, an oracle is built from the requirements.
    - … and is subject to interpretation by the developer
    - … and may contain faults
- A faulty oracle can be trouble.
  - May result in false positives - “pass” when there was a fault in the system.
  - May result in false negatives - “fail” when there was not a fault in the system.

Test Oracle Components

- **Oracle Information**
  - The information used by the oracle to judge the correctness of the implementation, given the inputs.
  - A specification, stored in a form that can be used directly by the testing code.
- **Oracle Procedure**
  - Code that uses that information to arrive at a verdict.
  - A form of *automated verification*.
  - Commonly as simple as...

```
if (actual value == expected value)
```

Oracle Trade-Offs

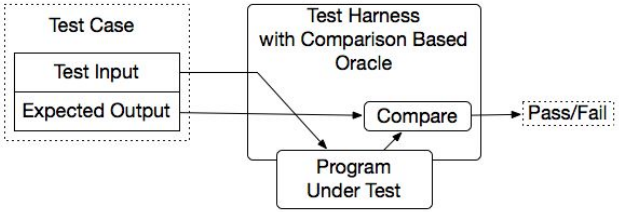
- We can specify the exact output behavior expected from the system.
  - This usually means we must write a unique oracle for **every test**.
- Or, trade *precision* for *generality*.
  - Specify properties that should be obeyed by a function.
  - Build a model of a function.
  - Check for anomalies that all programs can suffer from.

Types of Oracles

- **Specified Oracles**
  - Developers, using the requirements, formally specify properties that correct behavior should follow.
- **Derived Oracles**
  - An oracle is derived from development artifacts or system executions.
- **Implicit Oracles**
  - An oracle judges correctness using properties expected of many programs.
- **Human Oracles**
  - Judge tests manually.

Specified Oracles

Specified Oracles judge behavior using a human-created specification of correctness.



- Most oracles are specified oracles.
  - Any manually-written test case has a specified oracle.

Expected-Value Oracles

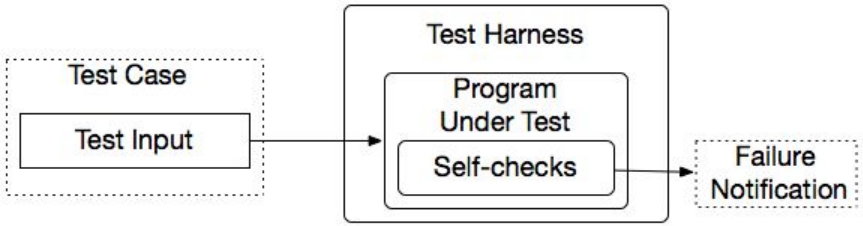
- Simplest oracle - what exactly should happen?

```
int expected = 7;
int actual = max(3, 7);
assertEquals(expected, actual);
```

- Oracle written for a single test case, not reusable.

Self-Checks as Oracles

Rather than comparing actual values, use properties about results to judge sequences.



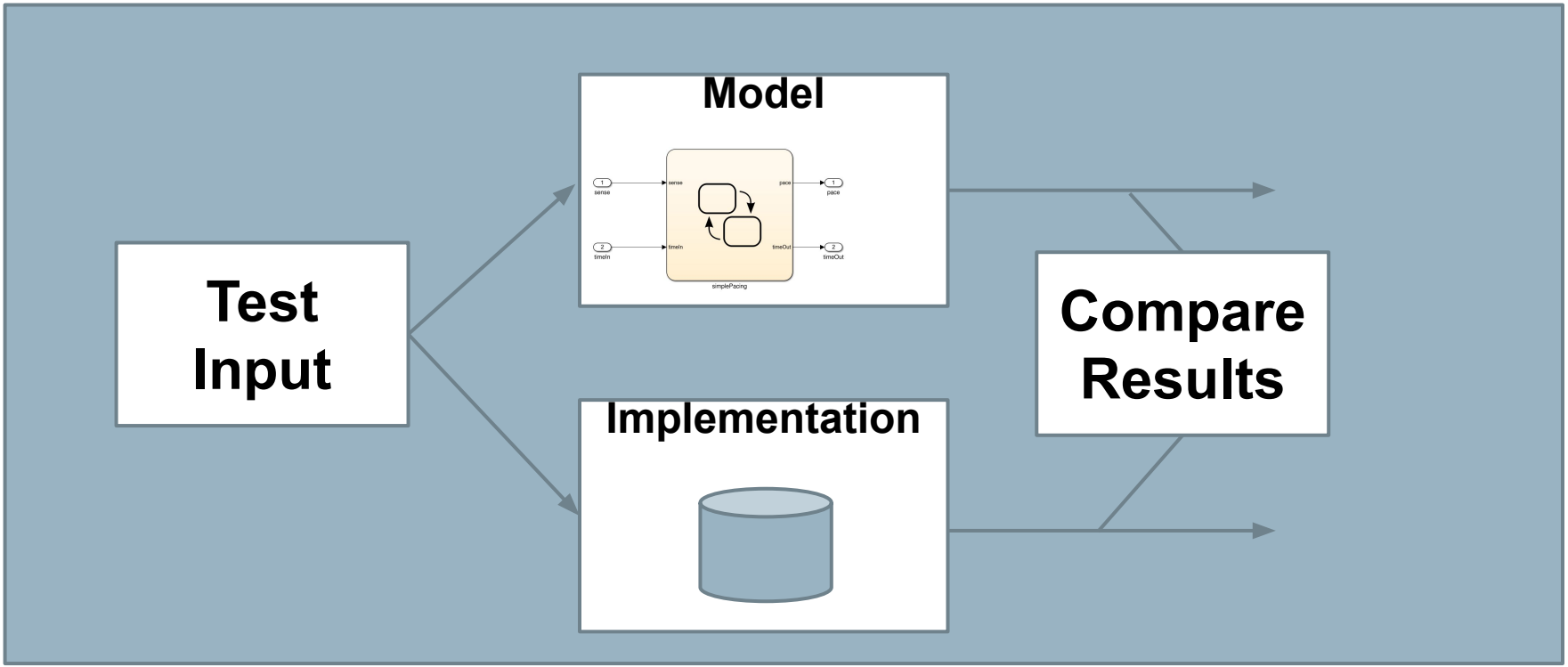

```
@Test
public void propertiesOfSort (String[] input) {
  // Tests
      String[] sorted = quickSort(input);
      assert(sorted.size >= 1, "This array can't be empty.")
      for (int item = 1; item < sorted.length; item++)
        assert(sorted[item] > sorted[item - 1], "Items
                  should be sorted in ascending order");
}
```

Uses assertions, contracts, and other logical properties.

Self-Checks

- Usually written at the function level.
  - For one method or one high-level “feature”.
  - Properties based on behavior of that function.
- Work for any input to that function.
- Only accurate for those properties.
  - Faults may be missed even if the specified properties are obeyed.
  - More properties = more expensive to write.

System Models as Oracles



Problem: Abstraction

- A model is not a real system.
  - Models are useful for requirements analysis, but may not reflect operating conditions.
  - May get “fail” verdict because the system’s behavior does not match, but the system acted correctly.
- Models are highly reusable, but less accurate than other oracles.

Derived Oracles

Oracles can *sometimes* be automatically derived from existing sources of information:

- Project Artifacts
  - Documentation
  - Existing tests
  - Other versions of the system
- Program Executions
  - Invariant detection
  - Specification mining

Implicit Oracles

- Implicit oracles check properties that are expected of any runnable program.
  - Network irregularities
  - Deadlock.
  - Memory leaks.
  - Excessive energy usage or downloads.
- These are faults that do not require expected output to detect.

Bugs? What are Those?

- Bug is an overloaded term.
- Does it refer to the bad behavior observed?
- The source code problem that led to that behavior?
- Both?

Faults and Failures

- **Failure**
  - An execution that yields an incorrect result.
- **Fault**
  - The problem that is the source of that failure.
  - For instance, a typo in a line of the source code.
- When we observe a failure, we try to find the fault that caused it.

Software Testing

- The main purpose of testing is to find faults:

  “Testing is the process of trying to discover every conceivable fault or weakness in a work product”
  - Glenford Myers

- Tests must reflect normal system usage and extreme boundary events.

Testing Scenarios

- **Verification:** Demonstrate to the customer that the software meets the specifications.
  - Tests tend to reflect “normal” usage.
  - If the software doesn’t conform to the specifications, there is a fault.
- **Fault Detection:** Discover situations where the behavior of the software is incorrect.
  - Tests tend to reflect extreme usage.

Axiom of Testing

"Program testing can be used to show the presence of bugs, but **never their absence**."

- Dijkstra

Black and White Box Testing

- Black Box (Functional) Testing
  - Designed without knowledge of the program's internal structure and design.
  - Based on functional and non-functional requirement specifications.
- White Box (Structural) Testing
  - Examines the internal design of the program.
  - Requires detailed knowledge of its structure.
  - Tests typically based on coverage of the source code (all statements/conditions/branches have been executed)

Let’s take a break.

Testing Stages

Testing Stages

- Unit Testing
  - Testing of individual methods of a class.
  - Requires design to be final, so usually written and executed simultaneously with coding of the units.
- Module Testing
  - Testing of collections of dependent units.
  - Takes place at same time as unit testing, as soon as all dependent units complete.

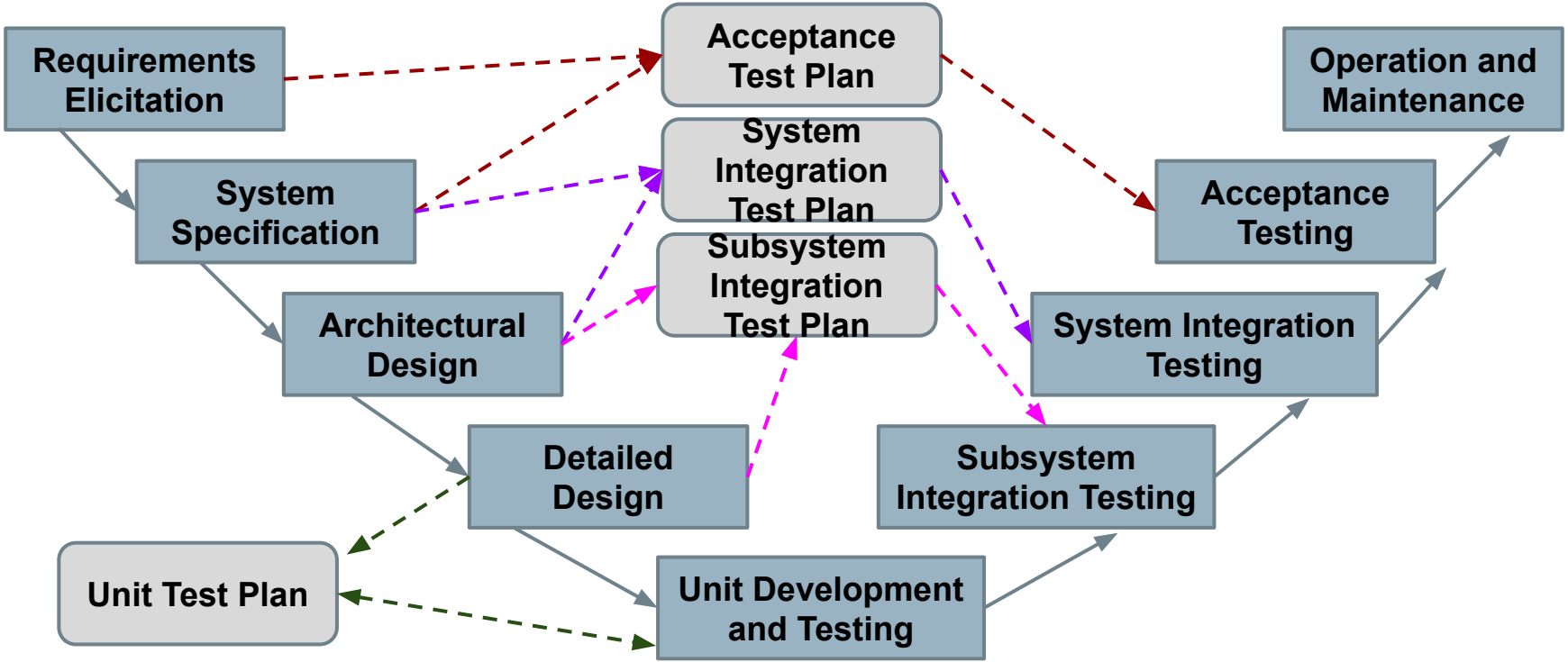
Testing Stages

- Subsystem Integration Testing
  - Testing modules integrated into subsystems.
  - Tests can be written once design is finalized, using SRS document.
- System Integration Testing
  - Integrate subsystems into a complete system, then test the entire product.
  - Tests can be written as soon as specification is finalized, executed after subsystem testing.

Testing Stages

- Acceptance Testing
  - Give product to a set of users to check whether it meets their needs. Can also expose more faults.
    - Alpha/beta Testing - controlled pools of users, generally on their own machine.
    - Acceptance Testing - controlled pool of customers, in a controlled environment, formal acceptance criteria
  - Acceptance planning can take place during requirements elicitation.

The V-Model of Development



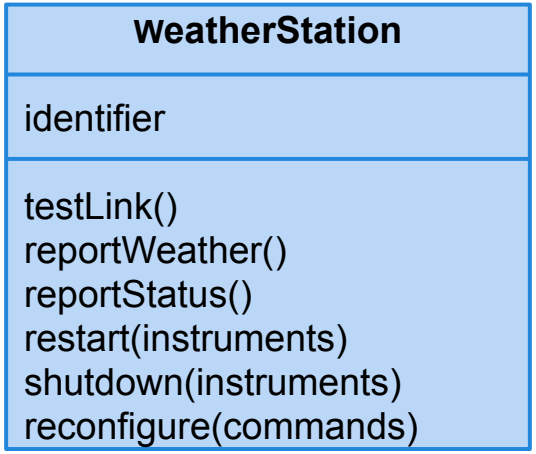
Unit Testing

- Unit testing is the process of testing the smallest isolated “unit” that can be tested.
  - Often, a class and its methods.
  - A small set of dependent classes.
- Test input should be calls to methods with different input parameters.

Unit Testing

- For a unit, tests should:
  - Test all “jobs” associated with the unit.
    - Methods belonging to a class.
    - If methods should be used in different orderings, try each.
  - Set and check value of all class variables.
    - Look at different ways those variables change in response to method calls. Put the variables into all possible states (types of values).

Unit Testing - WeatherStation



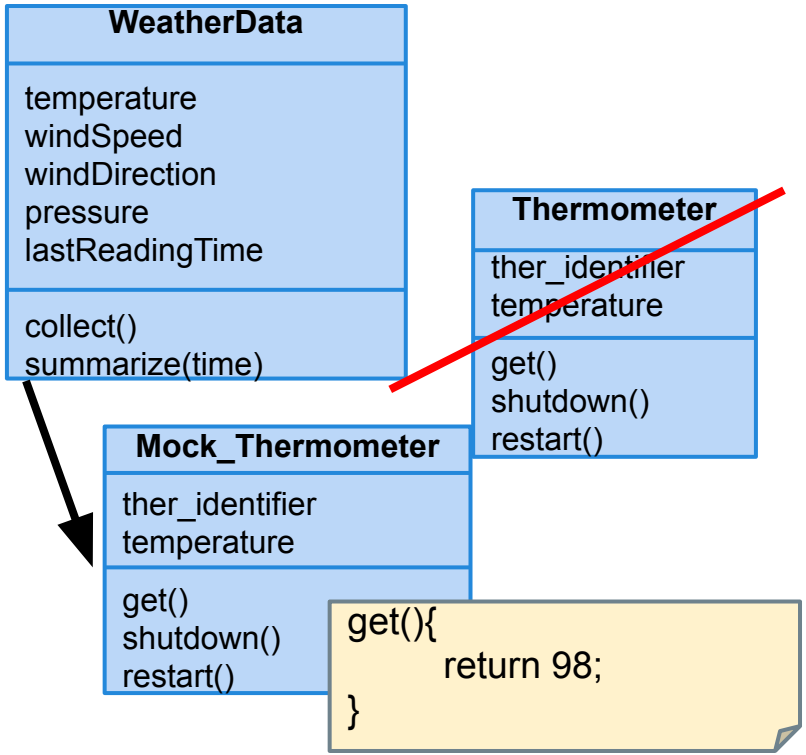
When writing unit tests for WeatherStation, we need:

- Set and check identifier.
  - Can any methods change the identifier?
- Tests for each “job” performed by the class.
  - Methods that work together to perform that class’ responsibilities.
- Tests that hit each outcome of each “job” (error handling, return conditions).

Unit Testing - Object Mocking

Components may depend on other, unfinished (or untested) components. You can **mock** those components.

- Mock objects have the same interface as the real component, but are hand-created to simulate the real component.
- Can also be used to simulate abnormal operation or rare events.



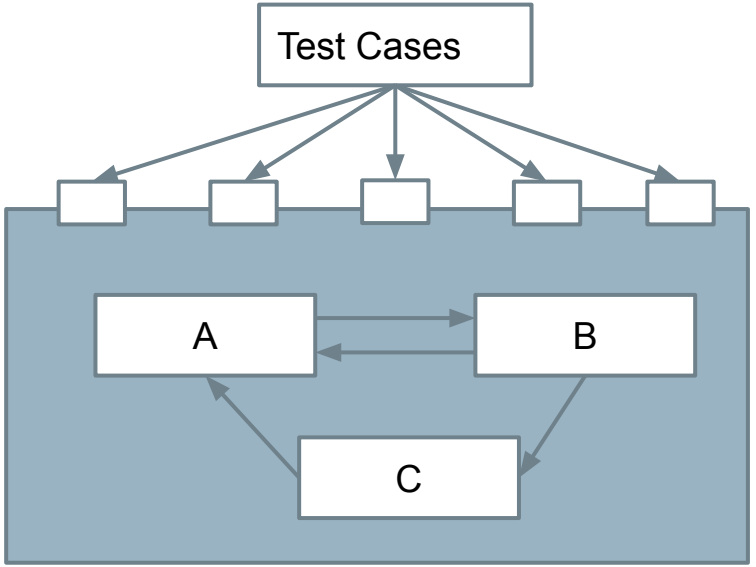
Subsystem Testing

- Most software works by combining multiple, interacting components.
  - In addition to testing components independently, we must test their *integration*.
- Functionality performed across components is accessed through a defined interface.
  - Therefore, integration testing focuses on showing that functionality accessed through this interface behaves according to the specifications.

Subsystem Testing

We have a subsystem made up of A, B, and C. We have performed unit testing...

- However, they work together to perform functions.
- Therefore, we apply test cases not to the classes, but to the interface of the subsystem they form.
- Errors in their combined behavior result are not caught by unit testing.



Interface Types

- Parameter Interfaces
  - Data is passed from one component to another.
  - All methods that accept arguments have a parameter interface.
  - If functionality is triggered by a method call, test different parameter combinations to that call.
- Procedural Interfaces
  - When one component encapsulates a set of functions that can be called by other components (offers an API).
  - Controls access to subsystem functionality. Thus, is important to test rigorously.

Interface Types

- Shared Memory Interfaces
  - A block of memory is shared between components.
  - Data is placed in this memory by one subsystem and retrieved by another.
  - Common if system is architected around a central data repository.
- Message-Passing Interfaces
  - Interfaces where one component requests a service by passing a message to another component. A return message indicates the results of executing the service.
  - Common in parallel systems, client-server systems.

Interface Errors

- Interface Misuse
  - A calling component calls another component and makes an error in the use of its interface.
  - Wrong type or malformed data passed to a parameter, parameters passed in the wrong order, wrong number of parameters.
- Interface Misunderstanding
  - Incorrect assumptions made about the called component.
  - A binary search called with an unordered array.
- Timing Errors
  - In shared memory or message passing - producer of data and consumer of data may operate at different speeds, and may access out of data information as a result.

System Testing

- Systems are developed as interacting subsystems.
- Once units and subsystems are tested, the combined system must be tested.
  - Advice about interface testing still important here
  - Two important differences:
    - Reusable components (off-the-shelf systems) need to be integrated with the newly-developed components.
    - Components developed by different team members or groups need to be integrated.

Acceptance Testing

Once the system is internally tested, it should be placed in the hands of users for feedback.

- Users must ultimately approve the system.
- Many faults do not emerge until the system is used in the wild.
  - Alternative operating environments.
  - More eyes on the system.
  - Wide variety of usage types.

Acceptance Testing Types

- Alpha Testing
  - A small group of users work closely with development team to test the software.
- Beta Testing
  - A release of the software is made available to a larger group of interested users.
- Formal Acceptance Testing
  - Customers decide whether or not the system is ready to be released.

Acceptance Testing Stages

- Define acceptance criteria
  - Work with customers to define how validation will be conducted, and the conditions that will determine acceptance.
- Plan acceptance testing
  - Decide resources, time, and budget for acceptance testing. Establish a schedule. Define order that features should be tested. Define risks to testing process.

Acceptance Testing Stages

- Derive acceptance tests.
  - Design tests to check whether or not the system is acceptable. Test both functional and non-functional characteristics of the system.
- Run acceptance tests
  - Users complete the set of tests. Should take place in the same environment that they will use the software. Some training may be required.

Acceptance Testing Stages

- Negotiate test results
  - It is unlikely that all of the tests will pass the first time. Developer and customer negotiate to decide if the system is good enough or if it needs more work.
- Reject or accept the system
  - Developers and customer must meet to decide whether the system is ready to be released.

Let’s take a break.

Principles of Analysis and Testing

Basic Principles

- Engineering disciplines are guided by core principles.
  - Provide rationale for defining, selecting, and applying techniques and methods.
- Testing and analysis are guided by six principles:
  - Sensitivity, redundancy, restriction, partition, visibility, and feedback.

Sensitivity

- Faults may lead to failures, but faulty software might not always fail.
- **Sensitivity Principle:** It is better to fail every time rather than only on some executions.
  - Earlier a fault is detected, the lower the cost to fix.
  - A fault that triggers a failure every execution is unlikely to survive testing.
  - The goal of sensitivity - try to make faults easier to detect by making them cause failure more often.

Sensitivity

- Principle can be applied at design & code, testing, and environmental levels.
  - Design & Code: Change *how* the program reacts to faults.
  - Testing: Choose a technique more likely to force a failure when a fault exists.
  - Environmental: Reduce the impact of environmental factors on the results.

Sensitivity - Design

- Take operations known to potentially cause failures and ensure that they will fail when used improperly.
- Ex: C string manipulation.

```
strcpy(target,source);
  // May cause failure if source
  string too long.

void stringCopy(char *target, const
char *source, int howBig){
  assert(strlen(source) < howBig);
  // Check whether source string is
  too long.
  strcpy(target,source);
  // If length ok, copy the string.
}
```

Sensitivity - Test and Analysis

- Choose fault classes and favor techniques that cause faults to manifest in failures.
- Deadlocks/race conditions:
  - Testing cannot try enough combinations.
  - Model checking/reachability analysis will generally work.
- Test adequacy criteria specify rules on how certain types of statements are executed.
  - May help expose types of faults - i.e., condition coverage is likely to uncover problems with boolean expressions.

Redundancy

- If one part of a software artifact constrains the content of another, it is possible to check them for consistency.
- In testing, we want to detect differences between intended and actual behavior. We can better do this by adding **redundant** *statements of intent*.
  - Make clear how code should be executed, then ensure that your intentions are not violated.

Redundancy

- Ex: Type Checking
  - Type declaration is a statement of intent (this variable is an integer).
    - Redundant with how it is used in the code.
  - Type declaration constrains the code, so a consistency check can be applied.
- Java requires that methods explicitly declare exceptions that can be thrown.
- Many analysis tools check consistency between code and other project artifacts.

Restriction

- When there is no effective or cheap way to check a property, sometimes one can solve a different, more **restrictive** property.
  - Or limit the check to a more **restrictive** set of programs.
- If the restrictive property encompasses the complex property, then we know that the complex property will hold.
  - That is, being overprotective avoids bad situations.

Restriction

```
static void questionable{
    int k;
    for (int i=0; i < 10; ++i){
        if(condition(i)){
            k=0;
        }else{
            k += i;
        }
    }
}
```

- Can `k` ever be uninitialized the first time `i` is added to it?
- This is an undecidable question.
- However, Java avoids this situation by enforcing a simpler, **stricter** property.
  - No program can compile with potentially uninitialized references.

Partition

- AKA: Divide and conquer.
- The best way to solve a problem is to **partition** it into smaller problems to be solved independently.
  - Divide testing into stages (unit, subsystem, system).
  - Many analysis tools built around construction and analysis of a model.
    - First, simplify the system to make proof feasible.
    - Then, prove the property on the model.

Visibility and Observability

- **Visibility** is the ability to measure progress or status against goals.
  - Clear knowledge about current state of development or testing.
  - Ability to measure dependability against targets.
- **Observability** is the ability to extract useful information from a software artifact.
  - Be able to understand an artifact, to make changes to it, and to observe and understand its execution.
  - Equality checks, ability to convert data structures to text encodings.

Feedback

- Be able to apply lessons from experience in process and techniques.
  - In systematic inspection and code walkthroughs, use past experience to write and refine checklists.
  - In testing, prioritize test efforts based on likelihood of exposing certain fault classes.
  - Use experience in acceptance testing in creating user surveys.

We Have Learned

- What is testing?
- Testing terminology and definitions.
  - Oracles, faults, failures
- Testing stages include unit testing, subsystem testing, system testing, and acceptance testing.

We Have Learned

- Six principles guide analysis and testing:
  - **Sensitivity**: better to fail every time than sometimes.
  - **Redundancy**: make intentions explicit.
  - **Restriction**: make the problem easier.
  - **Partition**: divide and conquer.
  - **Visibility**: make information accessible.
  - **Feedback**: apply lessons from experience to refine techniques and approaches.

Next Time

- Exercise session today: quality scenarios
- Next lecture: Unit Testing
  - Optional reading: Pezze and Young, Ch 17
  - Before next Friday (February 7), make sure you have one laptop per group with an IDE installed with JUnit support.
    - Make sure JUnit tests can be run
      - IntelliJ: https://www.jetbrains.com/help/idea/configuring-testing-libraries.html
      - Eclipse: https://help.eclipse.org/2019-12/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2FgettingStarted%2Fqs-junit.htm

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY