Testing Fundamentals

CSCE 747 - Lecture 2 - 01/12/2017

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 is the key to producing *dependable* software.
  - Testing is the primary verification activity.

We Will Cover

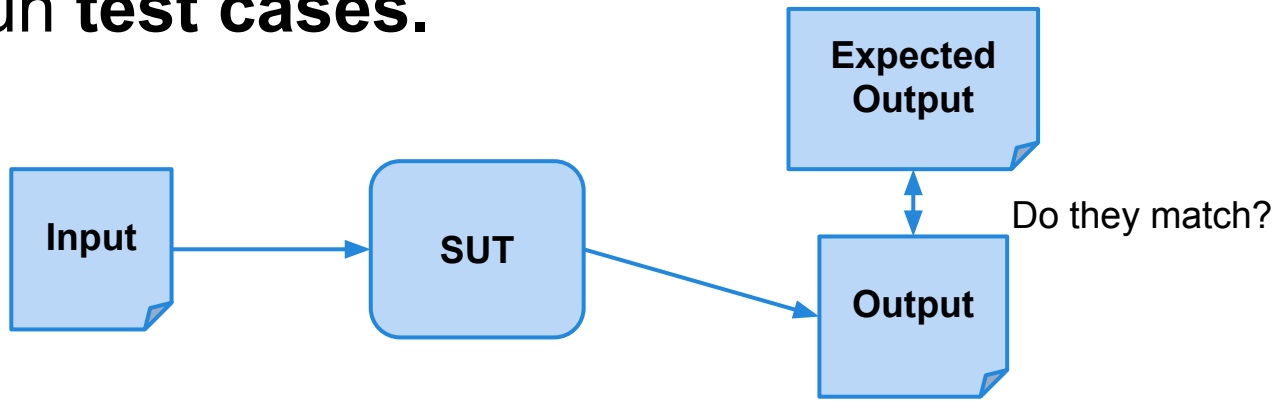
- What is testing?
- Testing definitions:
  - Let's get the language right.
- What is a test?
- Principles of analysis and testing.
- Testing stages:
  - Unit, Subsystem, System, and Acceptance Testing

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.

What is a Test?

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



To test, we need:

- **Test Input** - Stimuli fed to the system.
- **Test Oracle** - The expected output, and a way to check whether the actual output matches the expected output.

Anatomy of a Test Case

- Input
  - Any required input data.
- Expected Output (Oracle)
  - What *should* happen, i.e., values or exceptions.
- 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.

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, or both?
- **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)

Testing Stages

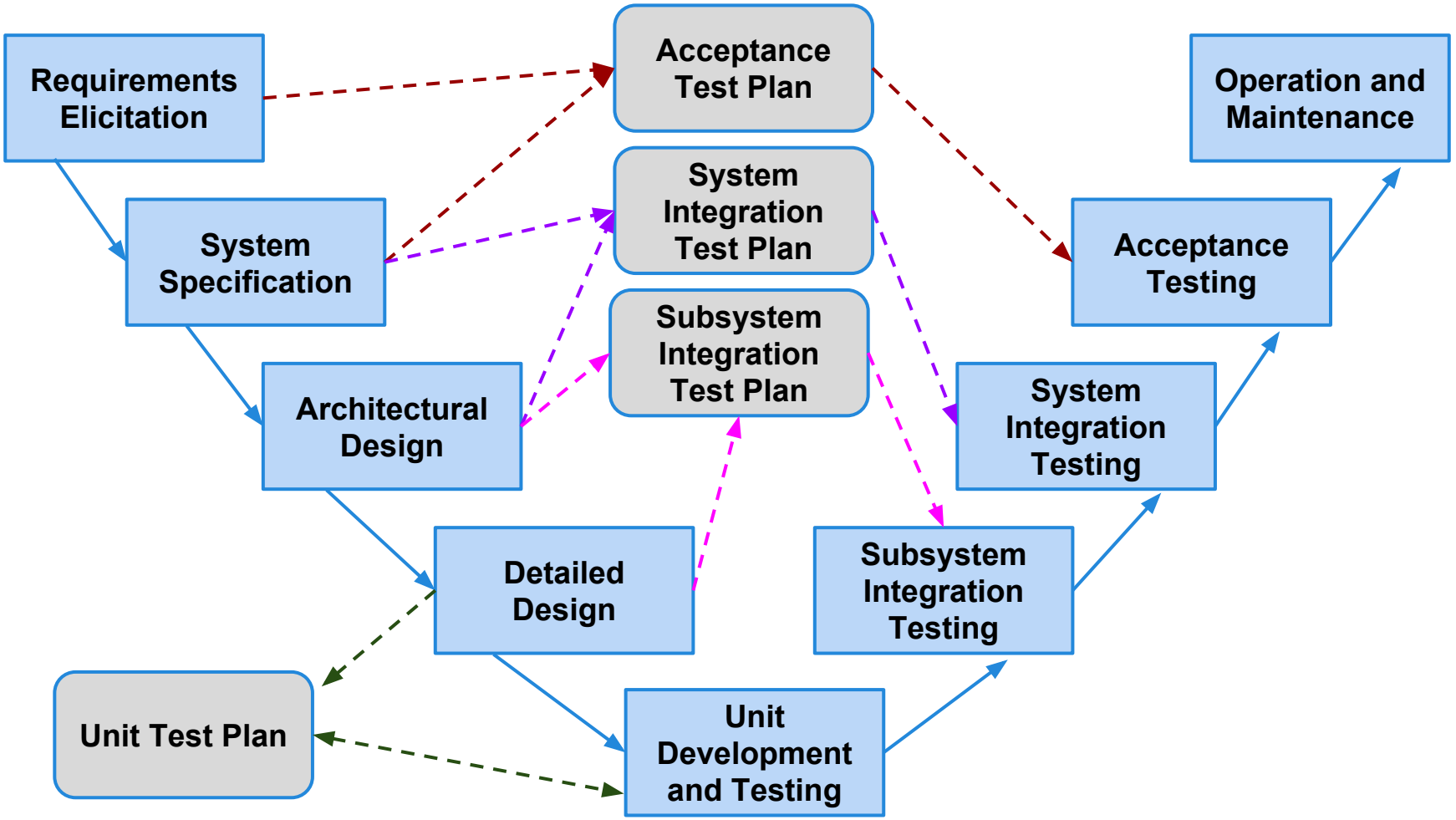
Testing Stages

- 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.
- Subsystem Integration Testing
  - Testing modules integrated into subsystems.
  - Tests can be written once design is finalized, using SRS document.

Testing Stages

- 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.
- Acceptance Testing
  - Give product to a set of users to check whether it meets their needs. Can also expose more faults.
  - Also called alpha/beta testing.
  - Acceptance planning can take place during requirements elicitation.

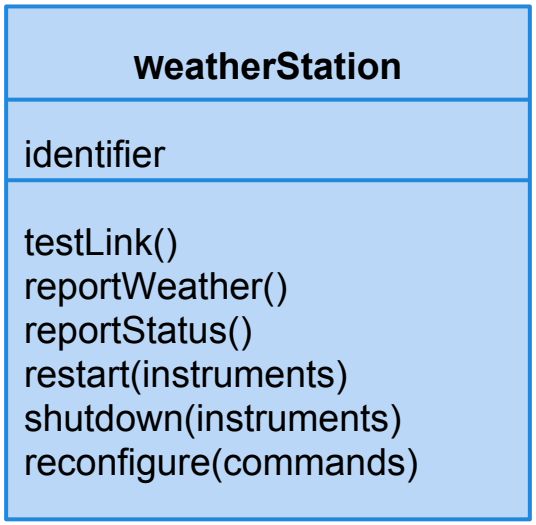
The V-Model of Development



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.
- For a class, tests should:
  - Test all “jobs” associated with the class.
  - Set and check the value of all attributes associated with the class.
  - Put the class into all possible states.

Unit Testing - WeatherStation



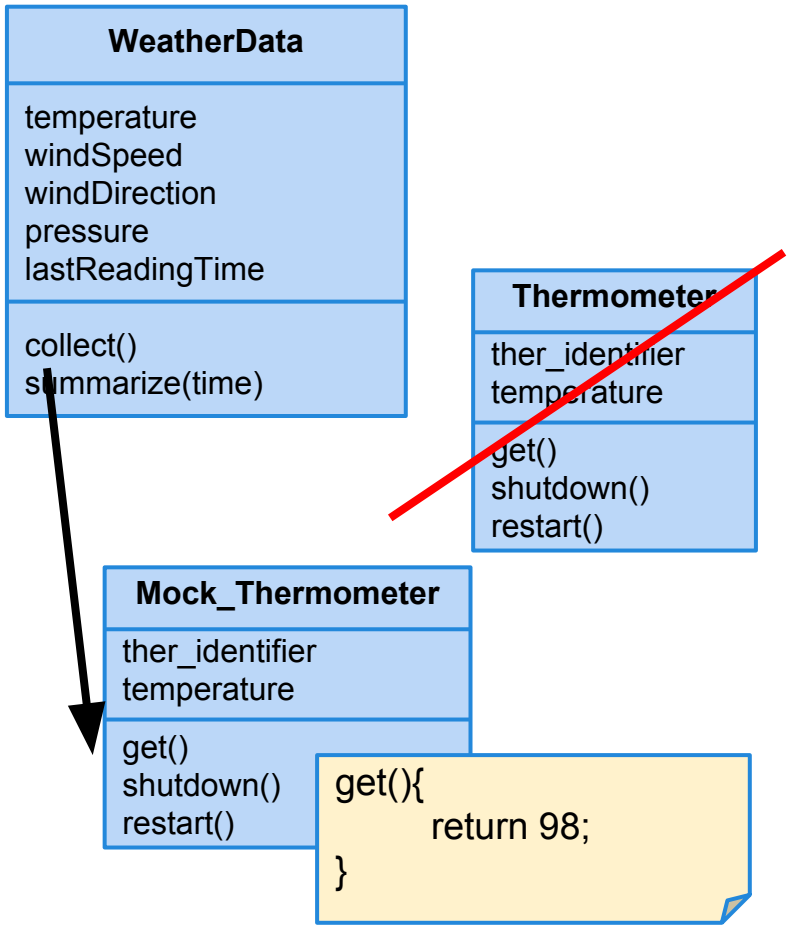
When writing unit tests for WeatherStation, we need:

- Set and check 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.



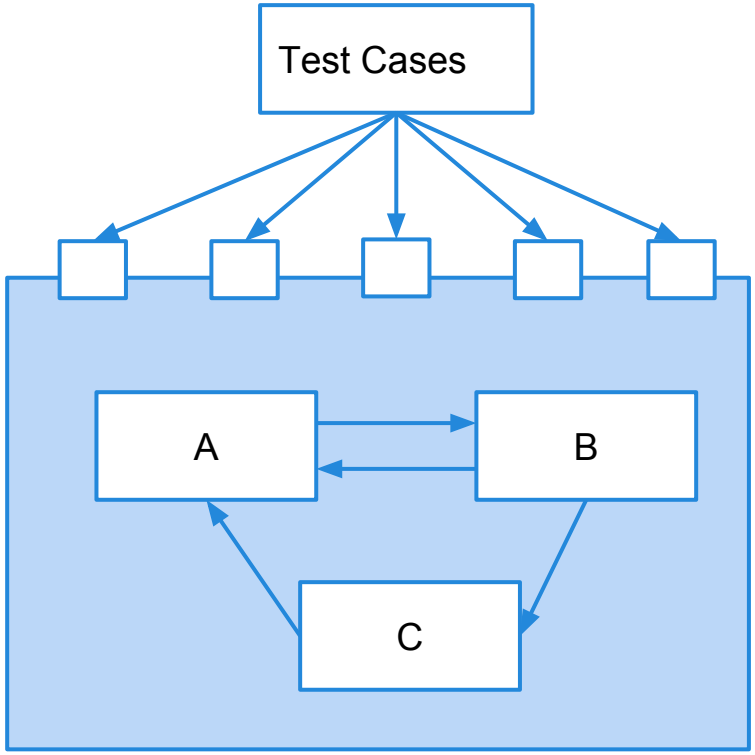
Subsystem Testing

- 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.
  - 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 (you interact with a system through some interface).
- 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 allows users to try the system under controlled conditions.

Acceptance Testing Types

Three types of user-based testing:

- 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.
- 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.
- Derive acceptance tests.
  - Design tests to check whether or not the system is acceptable. Test both functional and non-functional characteristics of the system.

Acceptance Testing Stages

- 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.
- 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.

Software Dependability

Dependability Properties

- When performing verification, we want to prove four things about the system:
  - That it is **correct**.
  - That it is **reliable**.
  - That it is **safe**.
  - That is is **robust**.

Correctness

- A program is **correct** if it is consistent with its specifications.
  - A program cannot be 30% correct. It is either correct or not correct.
  - A program can easily be shown to be correct with respect to a bad specification. However, it is often impossible to prove correctness with a good, detailed specification.
  - Correctness is a goal to aim for, but is rarely provably achieved.

Reliability

- A statistical approximation of correctness.
- Reliability is a measure of the likelihood of correct behavior from some period of observed behavior.
  - Time period, number of system executions
  - Measured relative to a specification and a usage profile (expected pattern of interaction).
    - Reliability is dependent on how the system is interacted with by a user.

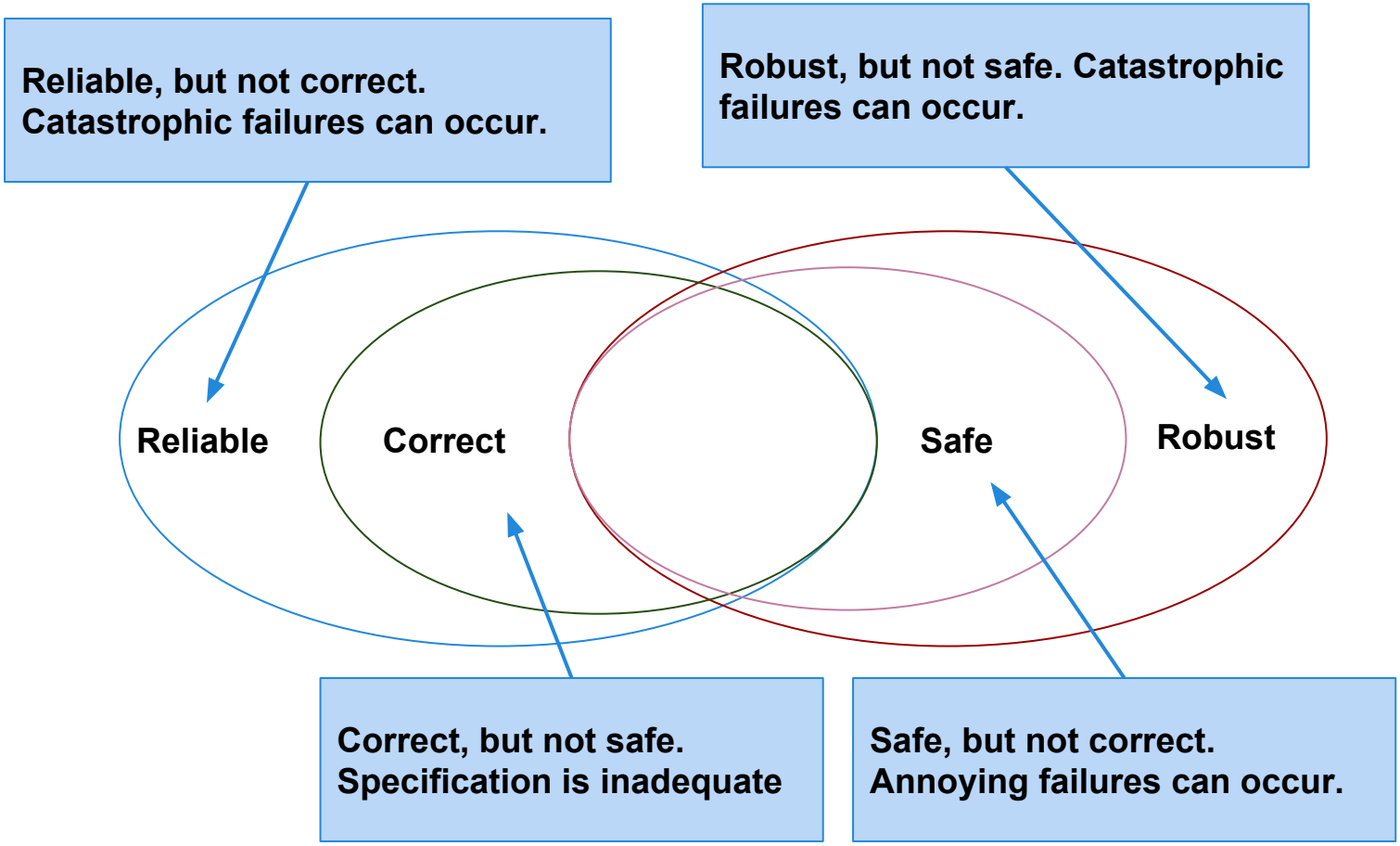
Safety

- Two flaws with correctness/reliability:
  - Success is relative to the strength of the specification.
  - Severity of a failure is not considered. Some failures are worse than others.
- **Safety** is the ability of the software to avoid *hazards*.
  - Hazard = any undesirable situation.
  - Relies on a specification of hazards.
    - But is only concerned with avoiding hazards, not other aspects of correctness.

Robustness

- Correctness and reliability are contingent on normal operating conditions.
- Software that is “correct” may still fail when the assumptions of its design are violated. *How* it fails matters.
- Software that “gracefully” fails is **robust**.
  - Consider events that could cause system failure.
  - Decide on an appropriate counter-measure to ensure graceful degradation of services.

Dependability Property Relations



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.
    - Especially once software has been released.
  - 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 are suited to these problems.
- Test adequacy criteria specify rules on how certain types of statements are executed.
  - Some are correlated to 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 smaller, 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 property.
  - No paths 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 the 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 fault classes.
  - Use experience in acceptance testing in creating user surveys.

We Have Learned

- What is testing?
- Testing terminology and definitions.
- Testing stages include unit testing, subsystem testing, system testing, and acceptance testing.
- We want testing to result in systems that are correct, reliable, safe, and robust.

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

- Finite Models
  - Representations of programs that we can use for analysis.

- Reading:
  - Chapter 5
- Team selection - due January 19th.