Testing Fundamentals

CSCE 247 - Lecture 6 - 02/06/2019

When is software ready for release?

Basic Answer...

Software is ready for release when you can argue that it is ***dependable*.**

- Correct, reliable, safe, and robust.
- The primary process of making software dependable (and providing evidence of dependability) is **Verification and Validation**.
  - **Testing** is our primary form of verification.

We Will Cover

- Revisiting Verification & Validation
- 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

Verification and Validation

Activities that must be performed to consider the software “done.”

- **Verification:** The process of proving that the software conforms to its specified functional and non-functional requirements.
- **Validation:** The process of proving that the software meets the customer’s true requirements, needs, and expectations.

Verification and Validation

Barry Boehm, inventor of “software engineering” describes them as:

- **Verification:** “Are we building the product right?”
- **Validation:** “Are we building the right product?”

Verification

- Is the implementation consistent with its specification?
  - “Specification” and “implementation” are roles.
    - Source code and requirement specification.
    - Detailed design and high-level architecture.
    - Test oracle and requirement specification.
- Verification is an experiment.
  - Does the software work under the conditions we set?
  - We can perform trials, evaluate the software, and provide evidence for verification.

Validation

- Does the product work in the real world?
  - Does the software fulfill the users' actual requirements?
- Not the same as conforming to a specification.
  - If we specify and implement all behaviors related to two buttons, we can achieve verification.
  - If the user expected a third button, we have not achieved validation.

Verification and Validation

- Verification
  - Does the software work as intended?
- Validation
  - Does the software meet the needs of your users?
  - **This is much harder.**

Validation shows that software is useful. Verification shows that it is dependable. Both are needed to be ready for release.

Verification and Validation: Motivation

Which is more important?

- Both are important.
  - A well-verified system might not meet the user's needs.
  - A system can't meet the user's needs unless it is well-constructed.

When do you perform V&V?

- Constantly, throughout development.
  - Verification requires specifications, but can begin then and be executed throughout development.
  - Validation can start at any time by seeking feedback.

Required Level of V&V

The goal of V&V is to establish confidence that the system is “fit for purpose.”

How confident do you need to be? Depends on:

- **Software Purpose:** The more critical the software, the more important that it is reliable.
- **User Expectations:** When a new system is installed, how willing are users to tolerate bugs because benefits outweigh cost of failure recovery.
- **Marketing Environment:** Must take into account competing products - features and cost - and speed to market.

Basic Questions

1. When do verification and validation start? When are they complete?
2. What techniques should be applied to obtain acceptable quality at an acceptable cost?
3. How can we assess readiness for release?
4. How can we control the quality of successive releases?
5. How can the development process be improved to make verification more effective (in cost and impact)?

When Does V&V Start?

- V&V starts as soon as the project starts.
- Feasibility studies must consider quality assessment.
- Requirement specifications can be used to derive test cases.
- Design can be verified against requirements.
- Code can be verified against design and requirements.
- Feedback can be sought from stakeholders at any time.

Perfect Verification

- For physical domains, verification consists of calculating proofs of correctness.
- Given a precise specification and a program, we should be able to do the same… Right?
  - Verification is an instance of the halting problem.
  - For each verification technique, there is at least one program for which the technique cannot obtain an answer in finite time.
    - Testing - cannot exhaustively try all inputs.
  - We must accept some degree of inaccuracy.

How Can We Assess the Readiness of a Product?

- Identifying faults is useful, but finding all faults is nearly impossible.
- Instead, need to decide when to stop verification and validation.
- Need to establish criteria for acceptance.
  - How good is “good enough”?
- One option is to measure dependability (availability, mean time between failures, etc) and set a “acceptability threshold”.

Product Readiness

- Another option is to put it in the hands of human users.
- Alpha/Beta Testing - invite a small group of users to start using the product, have them report feedback and faults. Use this to judge product readiness.
  - Can make use of dependability metrics for a quantitative judgement (metric > threshold).
  - Can make use of surveys as a qualitative judgement (are the users happy with the current product?)

Ensuring the Quality of Successive Releases

- Verification and validation do not end with the release of the software.
  - Software evolves - new features, environmental adaptations, bug fixes.
  - Need to test code, retest old code, track changes.
- Faults have not always been fixed before release. Do not forget those.
- Regression Testing - when code changes, rerun tests to ensure that it still works.
  - As faults are repaired, add tests that exposed them to the suite.

Improving the Development Process

- Try to learn from your mistakes in the next project.
- Collect data during development.
  - Fault information, bug reports, project metrics (complexity, # classes, # lines of code, coverage of tests, etc.).
- Classify faults into categories.
- Look for common mistakes.
- Learn how to avoid such mistakes.
- Share information within your organization.

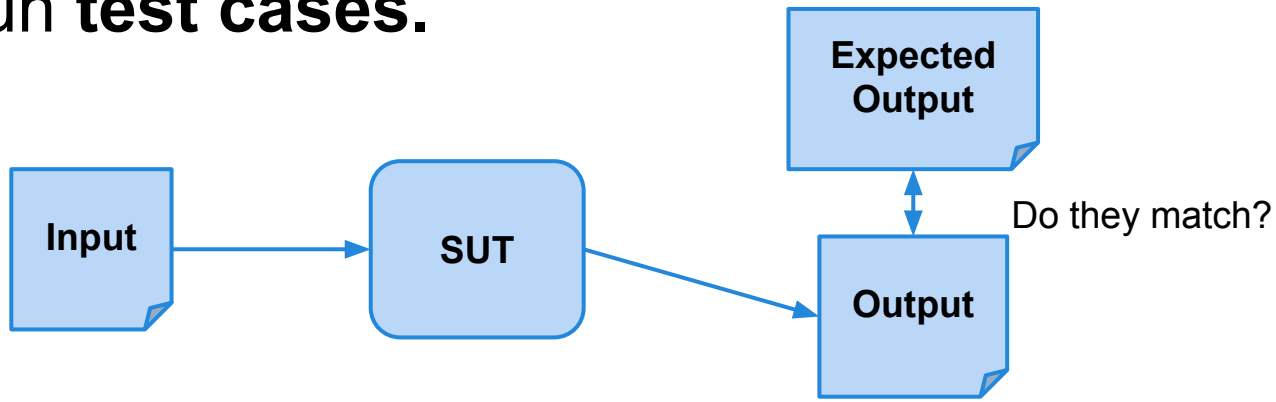
Software Testing:
The Primary Verification Activity

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.

Test Input

- Interactions with a software feature.
- Many means of interacting with software through testing:
  - Most common: a method call + pre-chosen parameter values
    - `trySomething(2,3);`
  - User interface interactions
  - Environment manipulation
- Can be inputted manually by a person or (preferably) through writing executable test code in a testing framework.

Test Oracles

- How we determine software correctness.
- Two components:
  - **Oracle Information:** Knowledge of what the “right” answer is.
    - Generally directly embedded in the test code for the chosen input:
      - `int actual = trySomething(2,3); int expected = 5;`
    - Can also correspond to general properties:
      - `assert(actual > 0);`
  - **Oracle Procedure:** Code to determine whether the actual output met expectations.
    - `assertEquals(expected, actual);`

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 both 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)

Test Plans

- Plan for how we will test the system.
  - **What** is being tested (units of code, features).
  - **When** it will be tested (required stage of completion).
  - **How** it will be tested (what scenarios do we run?).
  - **Where** we are testing it (types of environments).
  - **Why** we are testing it (what purpose does this test serve?).
  - **Who** will be responsible for writing test cases (assign responsibility).

Testing Stages

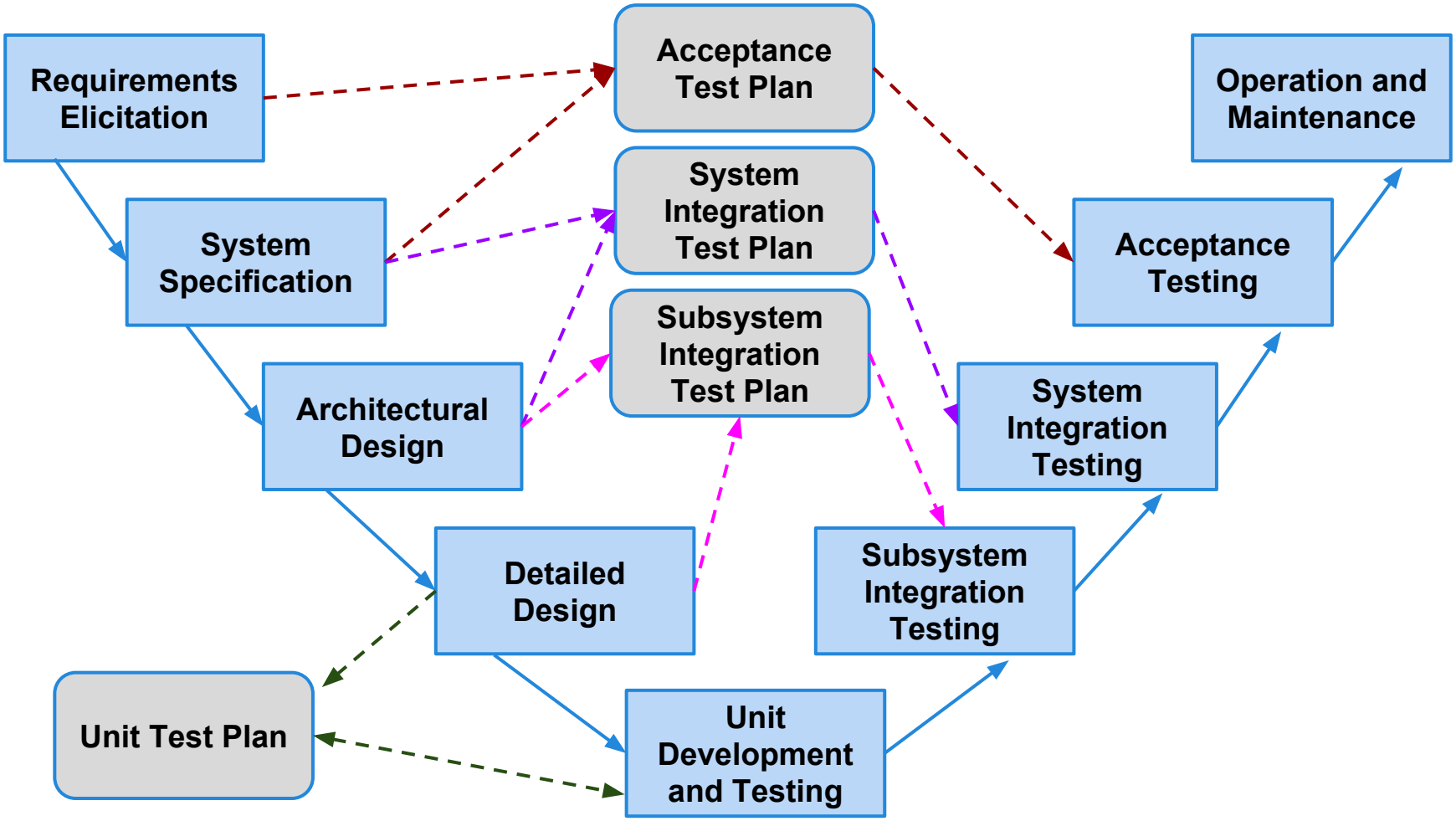
Testing Stages

- 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.

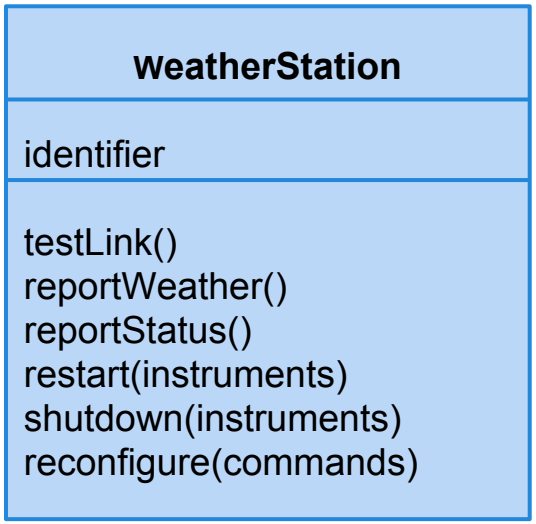
The V-Model of Development



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



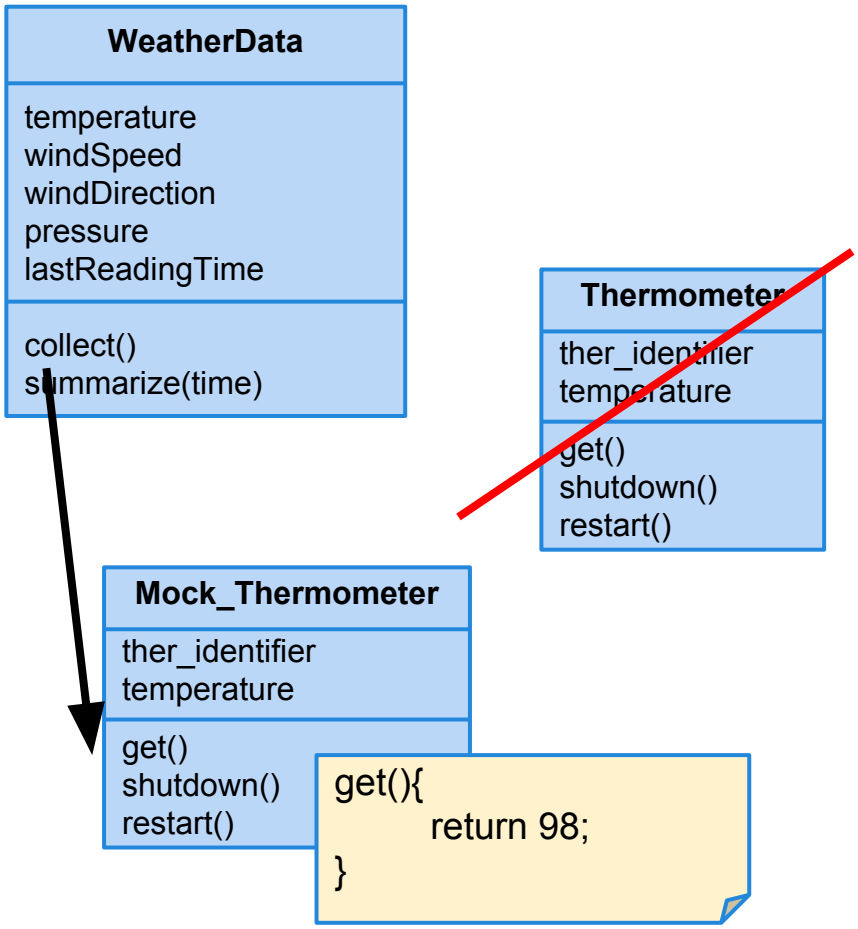
When writing unit tests for WeatherStation, we need:

- 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.



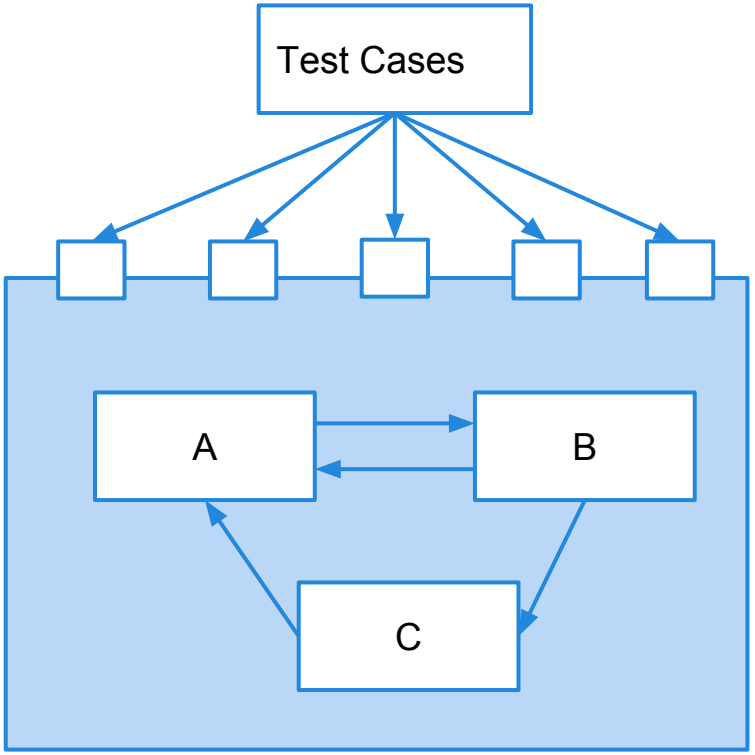
Subsystem Testing

- 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.

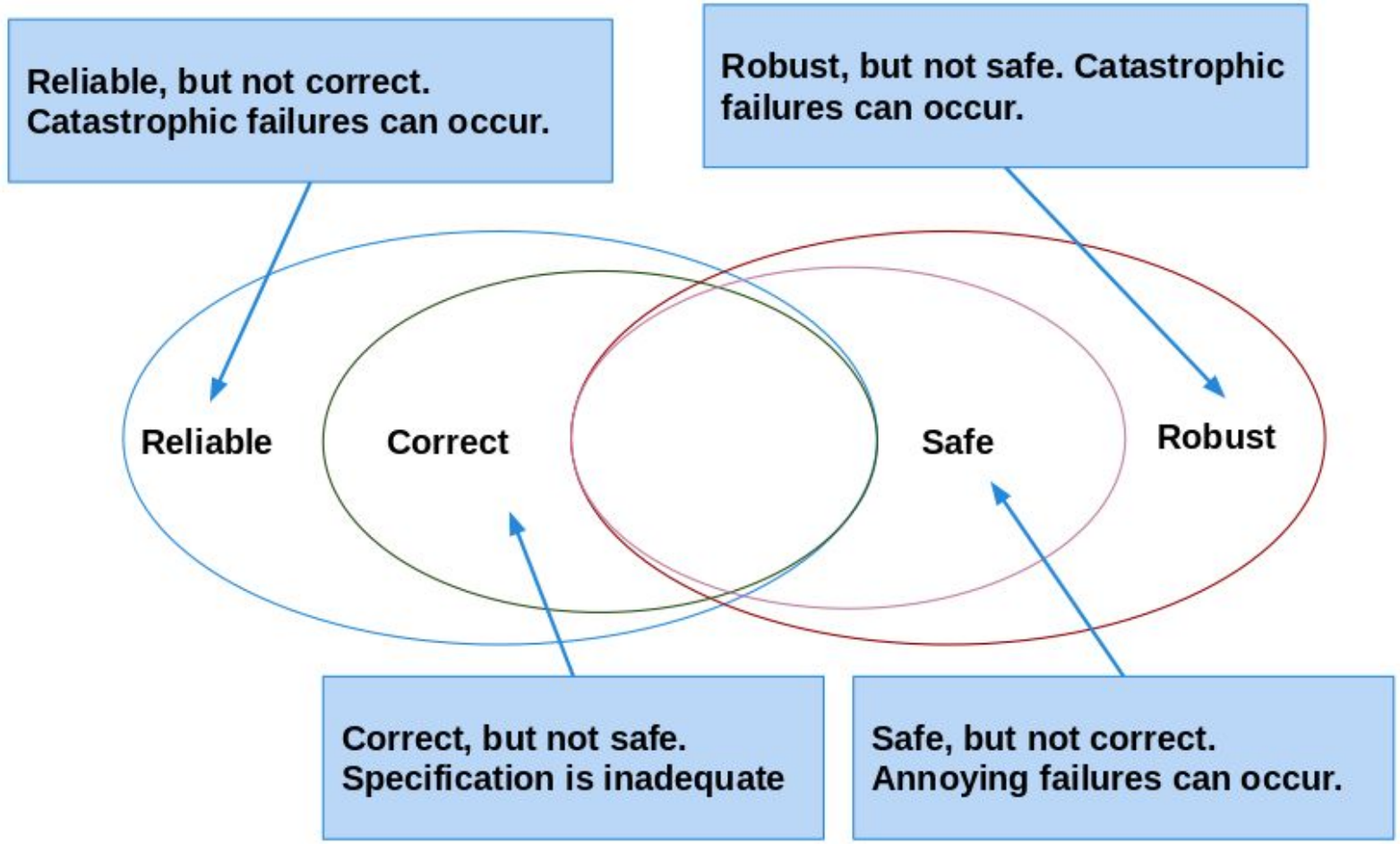
Safety

- 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



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.

Next Time

- Requirements Testability and Refinement

- Homework 1
  - Due February 10th
  - **Keep asking questions!**