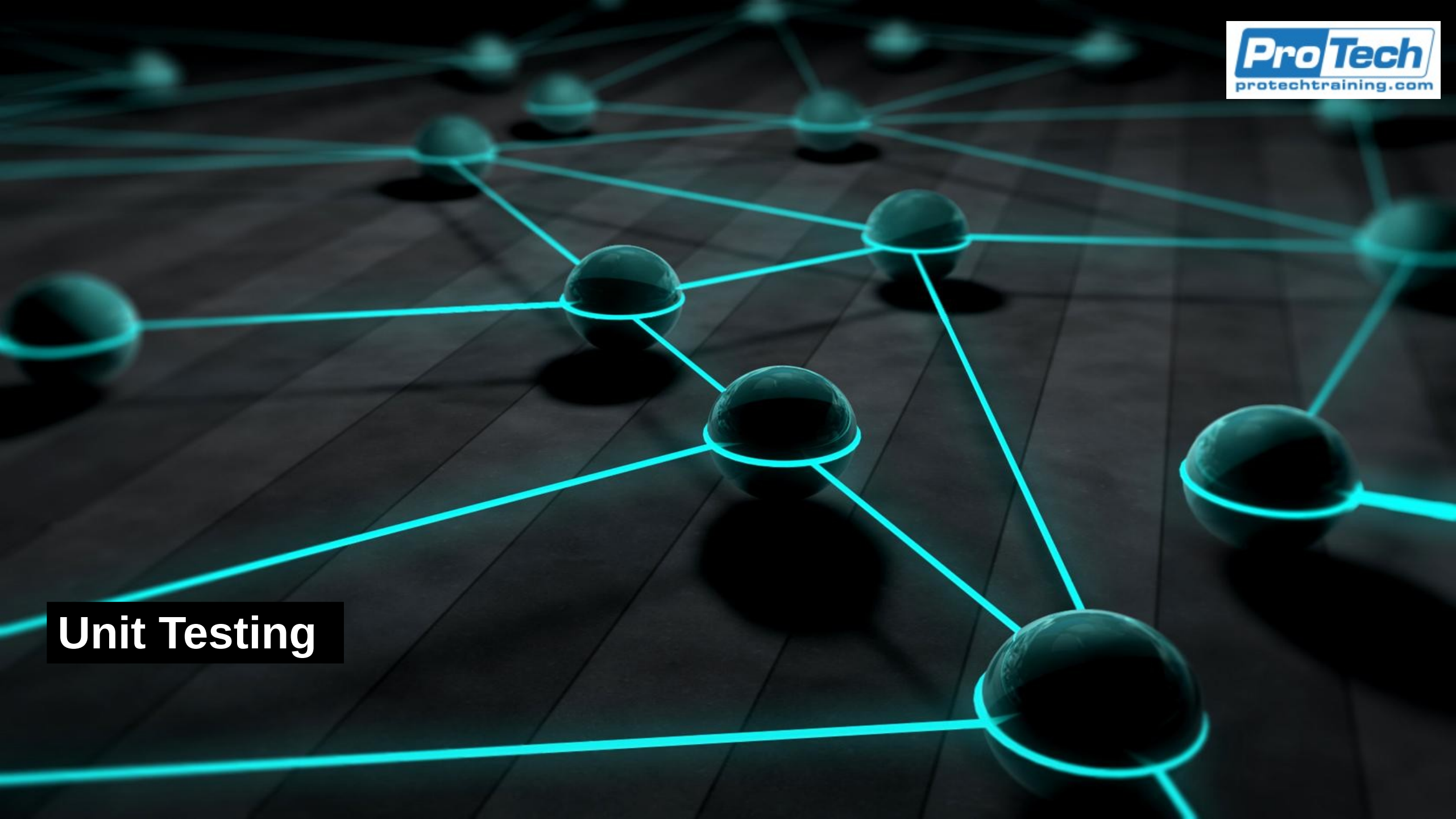
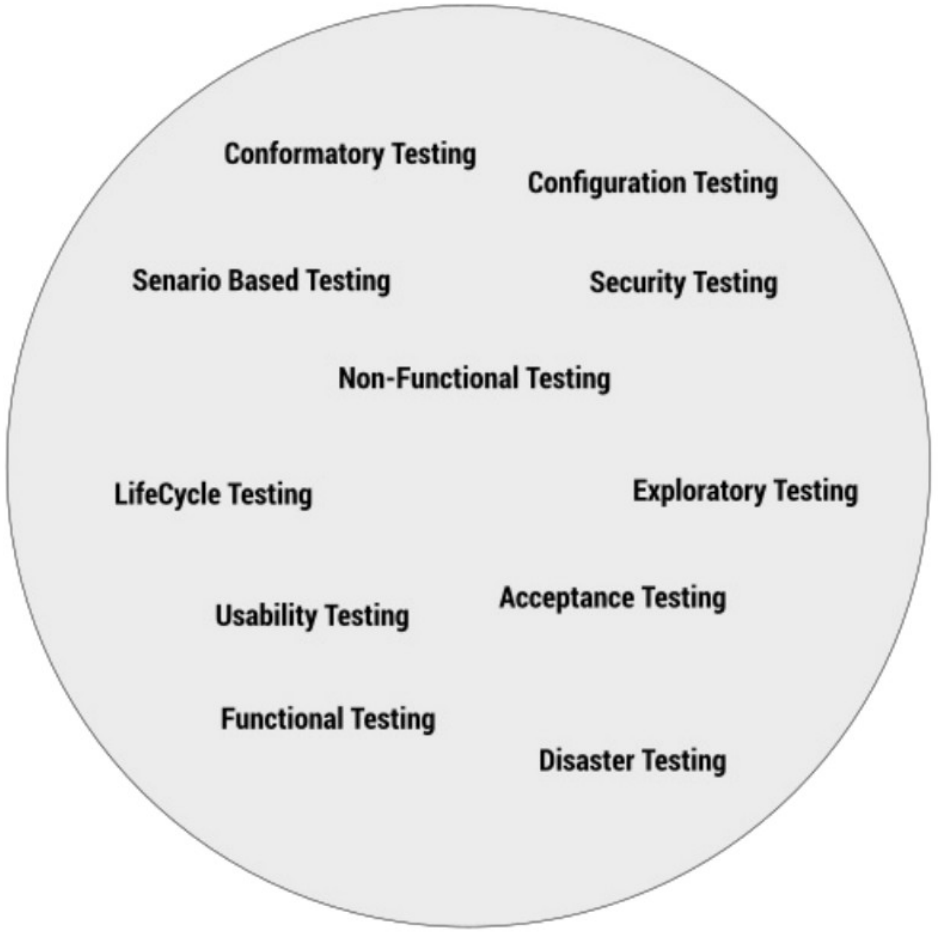



# Unit Testing

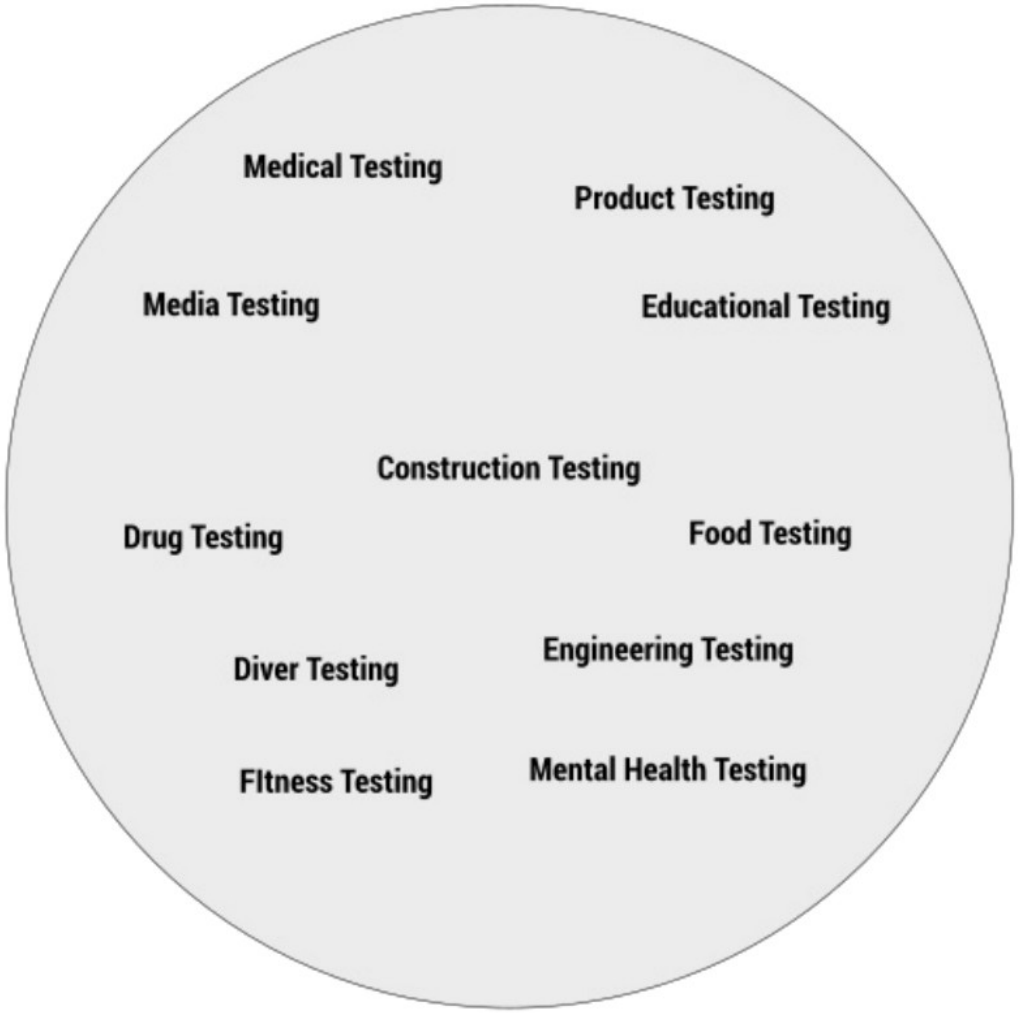
# Defining Software Testing

- There is no standard definition of software testing
  - There are many different kind of software testing
  - Each serves a specific function in the software ecosystem
- Software testing is
  - A branch of general testing
  - Industrial testing, product testing, medical testing, etc.
  - Adopted many of the common testing protocols and methods and adapted them to software



# Defining Software Testing

- These professions have defined best practice that have been adopted in software testing
  - This include testing methodologies
  - Controlling the quality of the testing
  - Developing a testing mindset
- Software testing is characterized by professional practices



# Why We Test

- In the literature, there are always lists of reasons why we test
- The are all just special cases of four different functions of testing
  - These are true across all forms of testing.
- In general terms, these are:
  - *Validation*: Testing to validate that we are doing or building the right thing
  - *Verification*: Testing to verify that we are building things correctly
  - *Quality*: Testing to ensure that we are building things to the right level of quality
  - *Risk*: Testing to identify potential business, project and product risk

# Exploratory Testing

- Asks the "what if?" questions to find the risk of failures that are often overlooked, especially when they are not immediately obvious
- As Douglas Adams notes
  - *The major difference between a thing that might go wrong and a thing that cannot possibly go wrong is that when a thing that cannot possibly go wrong goes wrong it usually turns out to be impossible to get at or repair*

**Brenan Keller**
@brenankeller

A QA engineer walks into a bar. Orders a beer. Orders 0 beers. Orders 99999999999 beers. Orders a lizard. Orders -1 beers. Orders a ueicbksjdhd.

First real customer walks in and asks where the bathroom is. The bar bursts into flames, killing everyone.

# Historical Evolution of Software Testing

- 1950s – mid 1970s:
  - Debugging oriented period
- Mid 1970s – early 1980s:
  - Introduction of engineering style specifications and methods test whether software meets the specification
- Early 1980s – early 1990s:
  - Risk analysis and robustness concerns emphasize testing to ensure that software doesn't break
- Early 1990s to mid 2000s:
  - Introduction of well defined software development processes lead to "testing throughout the development cycle" and applying QA principles to reduce the overall risk in development
- Mid 2000s to present:
  - Agile and DevOps continuous development processes require constant collaboration between testers, analysts and developers

# Beizer's Levels

- Boris Beizer in the 1980s in his book noted that as software tester got better at their profession, the way they approached testing evolved through a series of stages
  - Phase 1: There is no difference between testing and debugging. Other than in support of debugging, testing has no purpose
  - Phase 2: The purpose of testing is to show that software works
  - Phase 3: The purpose of testing is to show where software doesn't work
  - Phase 4: The purpose of testing is not to show anything, but to reduce the risk of not working to an acceptable value
  - Phase 5: Testing is not an activity. It is a mental discipline that results in low-risk software without much testing effort

# Effective or Good Testing

- Standard terminology
  - Effectiveness: A process is effective when it produces the right result
    - Effective testing is correct (to be defined in a sec)
  - Efficiency: A process is efficient when it is optimal in terms of how resources are used.
    - Efficient testing means our testing is economical in terms of time and resources
- Good testing has four characteristics:
  - *Validity*: We are actually testing what we claim we are testing
  - *Accuracy*: Our test are correctly reporting the existence of defects
  - *Reliability*: Our tests results are not influenced by extraneous factors
  - *Comprehensiveness:* Our tests meet the required quality and test objectives

# Test Validity

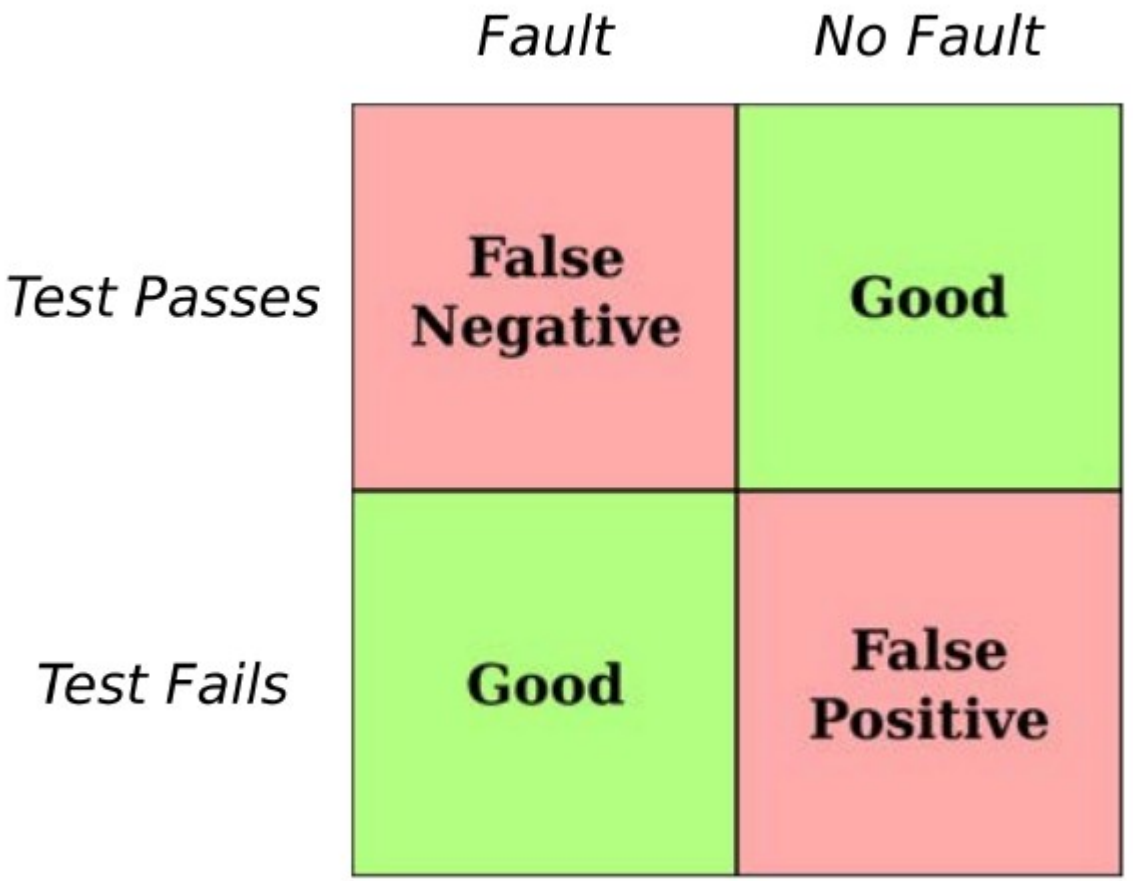
- Content Validity:
  - Our tests do not have content validity when we are claiming to test something that can't actually be measured or tested
    - The thing we want to measure is not defined in a way that we can test it
      - "Determine the mood of users after they sign up"
    - The thing we want to measure is not directly measurable, often because it represents some internal state of our user or other stakeholder
      - "Run some tests to find out much our users like our website"
- Construct Validity:
  - Our tests do not have construct validity when we think we are testing one thing, but we are in fact testing something else
  - For example, we think our tests are testing the database connection but in fact they are testing the UI because the UI is intercepting them before they get to the database

# Test Validity

- Predictive Validity:
  - Our tests do not have predictive validity if predictions made on the basis of the tests are incorrect
  - For example, all our tests passed but we get high amounts of bug reports from production

# Test Accuracy

- Assume that we are working with a single test intended to identify a particular class of faults
  - The test passes when it does not detect a fault fails when it detects the fault
  - These are two outcomes we want
  - A false negative occurs when there is a fault and the test passes or fails to detect the fault.
  - A false positive takes place when our test fails, meaning it indicates a fault exists but in reality there is no underlying fault.
  - Accurate tests have low rates of false positives and false negatives



# Test Reliability

- Testing is said to be reliable when the results of the tests depend only on what is being tested
  - For example, if a set of tests are performed a number of times and the results depend on when the tests are run or who runs them, then the tests are not reliable
- Reliability is ensured when we have standardized methods for:
  - Setting up for testing
  - Executing tests
  - Having clear measurable criteria to determine if the test passed or failed
  - Standardized ways of recording the result

# Test Comprehensiveness

- A very important and central issue in testing and deals with how much testing we do, what sort of testing and to what level of exhaustiveness
  - A very common problem with non-comprehensive testing is that things “slip through the cracks”
  - The problem is knowing how much testing to do so we don’t miss anything, and yet not doing so much testing that we use up massive amounts of resources for rather small decreases in results
  - The amount of possibilities our tests take into account is called “test coverage”

# Sources of Errors

- There are four main sources of error for testing:
  - Poor test planning
  - Poor test design
  - Poor test execution
  - Systemic errors
- We won’t be covering systemic errors in this class in depth
  - These generally refer to cognitive biases, perception errors, etc.
  - These are usually controlled for by using protocols for designing and running tests, and evaluating the results of the test

# Myers' Principles

- Glenford Myers articulated some of these protocols in 1979
  - A necessary part of a test case is a definition of the expected output or result
  - Programmers should avoid attempting to test their own code
  - A programming organization should not test its own code
  - Thoroughly inspect the result of each test
  - Test cases must be written for invalid and unexpected, as well as valid and expected input conditions
  - Examining a program to see if it does not do what it should is only half the battle - the other half is seeing whether the program does what it is not supposed to do
  - Avoid throw away test cases unless the program is truly a throw away program

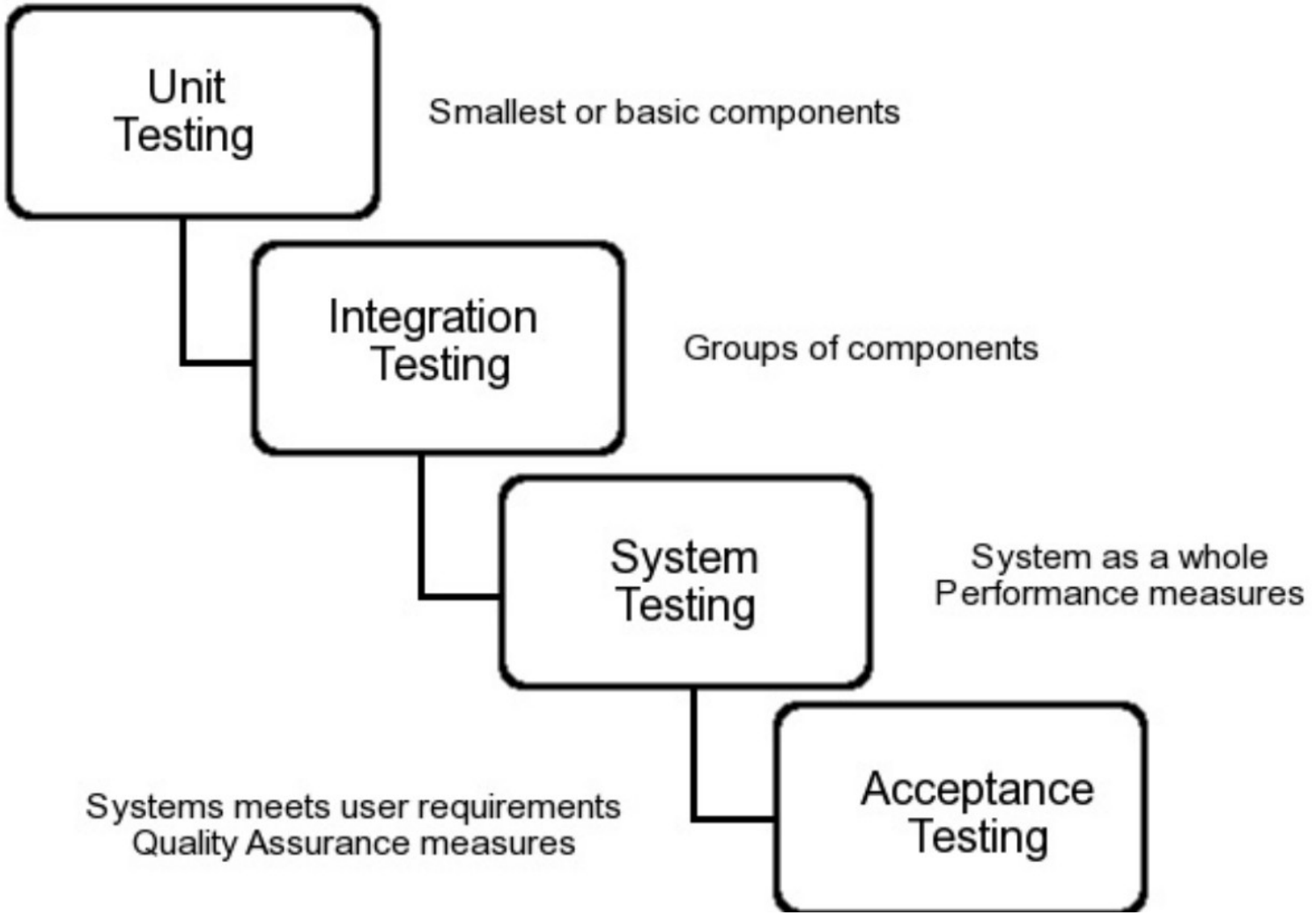
# Myers' Principles

- Glenford Myers (cont)
  - Do not plan a testing effort under the tacit assumption that no error will be found
  - The probability of the existence of more errors in a section of a program is proportional to the number of errors already found in that section
  - Testing is an extremely creative and intellectually challenging task
  - Testing is the process of executing a program with the intent of finding errors

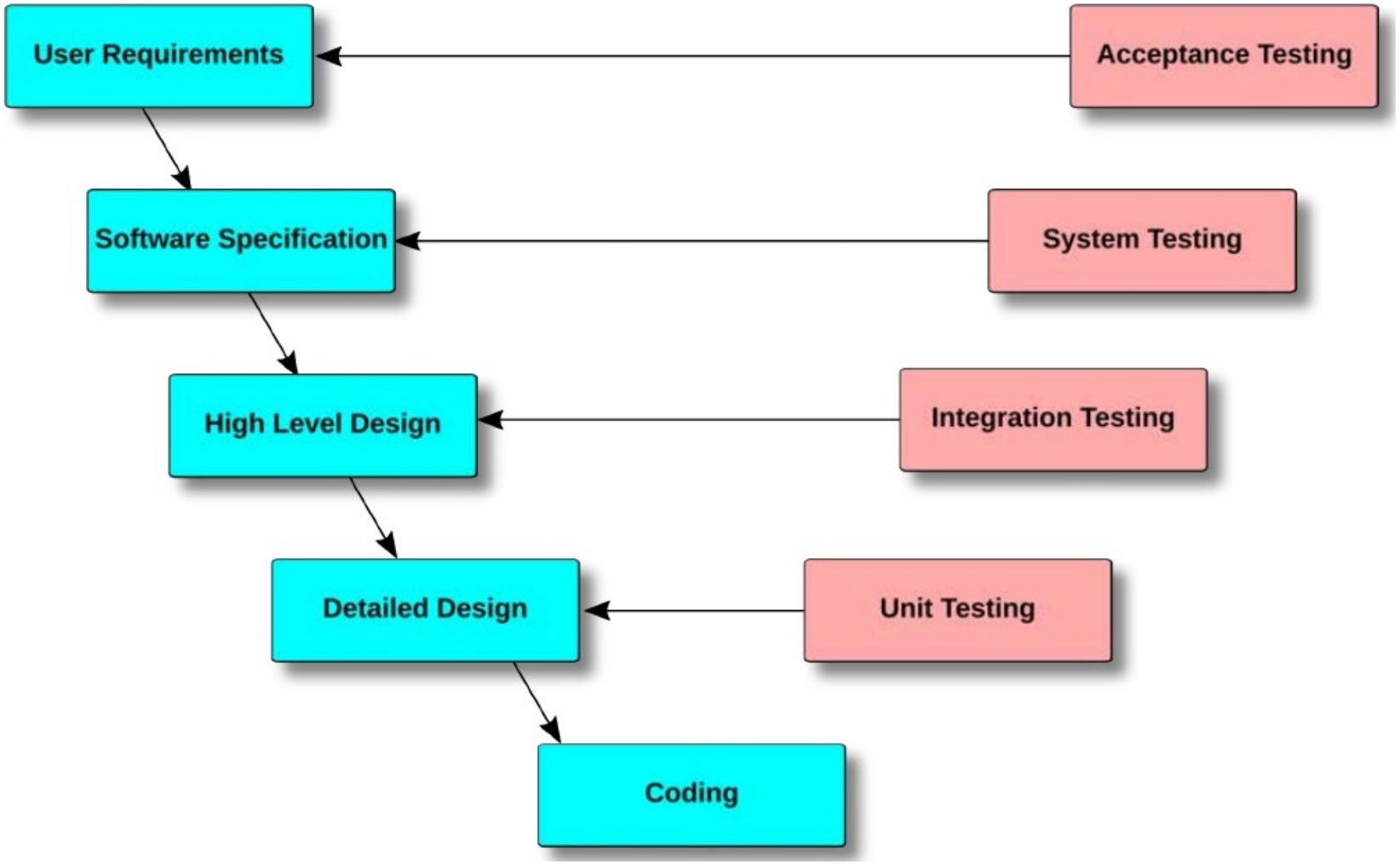
# Formal and Repeatable Processes

- All testing is planned and documented so that it is repeatable
  - In the same way that a science experiment is documented so that it also is reproducible
  - We have standardized ways of working with the testing life cycle and processes in order to have reliable testing
  - When ad hoc or informal testing is being done, we often lose the quality or reliability
- We use formal and standardized processes for the following reasons
  - It allows for repeatable tests and reliable results
  - It allows for formal reviews of the testing process
  - It allows us to use testing resources more effectively including the use of automated tools
  - It is essential for testing process improvement

# Levels of Testing



# The Classic V Model of Testing



# Levels of Testing

- Unit Testing
  - Unit testing depends on the interfaces and functional responsibility of the individual components
  - “Does each component satisfy its designed responsibilities?”
  - However, unit testing is not an actual level of testing but rather a testing strategy that can be used to simplify complex testing problems
- Integration Testing
  - Testing of collections of components to ensure they are “all fitting together” correctly
  - Explores the interaction and consistency of successfully tested components
  - The reason we do integration testing is to identify specifically those bugs that are observable in the interaction between two components but not when each component is tested individually

# Levels of Testing

- System Testing
  - Explores systems behaviors that cannot be tested by unit, component or integration testing.
  - Generally refers to non-functional testing, however, it can refer to any characteristic of the software that is a property of the system a whole and not a specific component
  - For example:
    - *Stress Testing:* The system is subjected to the maximum capacity it is specified to handle for a specified period of time
    - *Performance Testing:* Attempts to locate areas where the system does not meet performance specs such as response time, CPU time, and throughput
    - *Reliability Testing:* Tests for failures to meet specific reliability specifications; a system may have a reliability specification of two hours down time per forty years of operation
- Regression Testing
  - Not actually a level of testing, but is a critical phase of testing and a key concept in testing.
  - Regression testing is the retesting that occurs when changes are made to a system to ensure the that the new version of the software still has the required functionality the previous version and no new bugs have been introduced as a result of the changes

# Levels of Testing

- Acceptance Tests
  - We deploy the system into a production type environment and let the users “test drive” the system.
    - Acceptance testing moves the test criteria from “does it work” to “does it work for you the user”
  - Also include automated scenario testing that ensures the required functionality is correct
    - For example, Selenium test suits that automatically test all the different scenarios in a use case

# The Unit Testing Strategy

- Complexity
  - Consider a machine that has a single steel part
    - This machine can fail in one way – the part can break
  - Add a second part the same as the first part, and bolt the two together
    - Now the machine can fail in 7 ways – if we blindly add a test case for each possible combination, the number of test cases increases exponentially
    - The unit testing strategy is to break the complex machine into three components and then perform three unit tests
    - That provides us with enough information to determine if each unit tests pass, then we do one integration test and we

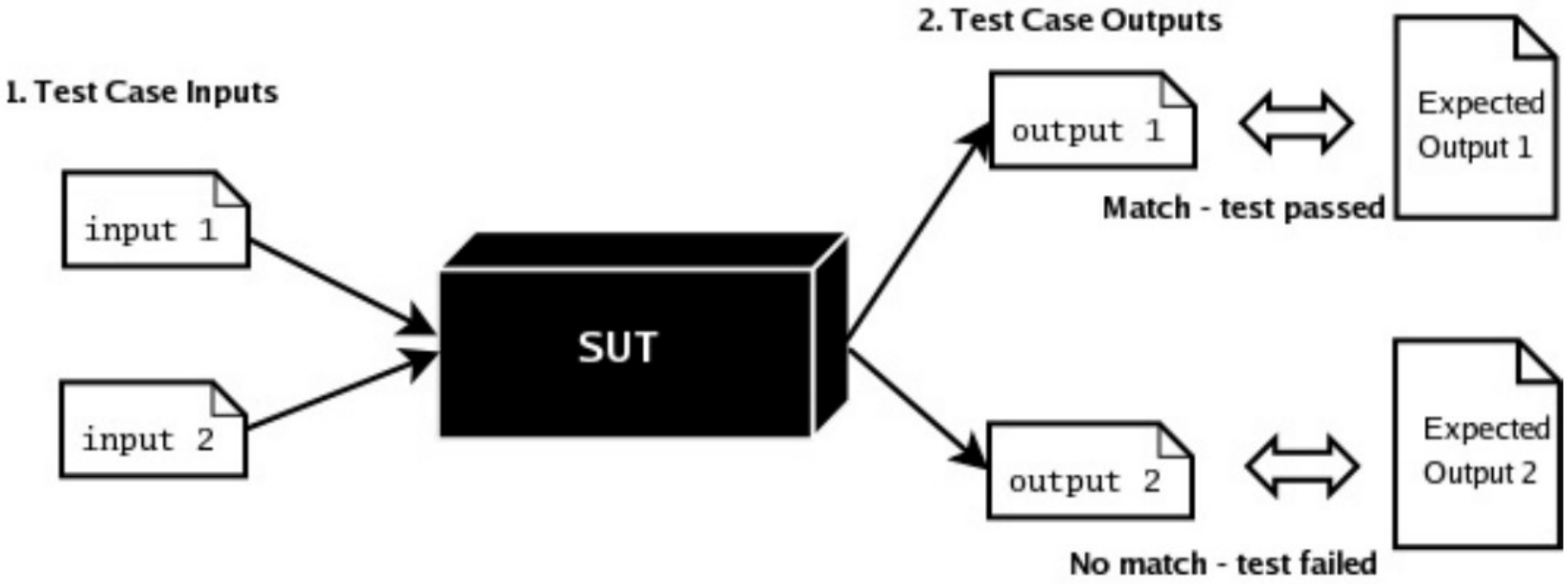
| Part | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Part 1 | P | P | P | P | F | F | F | F |
| Bolt | P | P | F | F | P | P | F | F |
| Part 2 | P | F | P | F | P | F | P | F |

# The Unit Testing Strategy

- Basic approach
  - Select the component to be unit tested, it can be any size from a piece of code to a subsystem
  - Create a driver
    - This is stand-in for the rest of the system
    - It feeds the test cases as input to the component and records the component's response
    - Any test failures are due to the component since we have sanitized the inputs
  - Create stubs (mocks)
    - Anytime the component needs to call another component
    - A stub is designed to return exactly the right response to each request the component makes
    - We know the correct replies in advance because they are determined by the test cases
    - For example, a database stub just has a list of the right responses for the queries that will be sent when the component executes the test cases
  - The test failures now are isolated to the component under test
  - We are testing only one thing (unit = one) which is the component under test

# Functional Testing

- Functional testing
  - Also referred to as behavioral testing and black box testing
  - A test methodology adapted from testing in engineering environments



# Functional Testing

- Process
  - The specification is analyzed to ensure it is "testable"
    - If it is not testable, it is referred back to whoever is responsible for specification for correction or modification to make it testable
  - Once we have a testable specification, we analyze it to identify and classify the functionality of the system under various conditions and inputs, which helps us identify the types of test cases that will be needed
  - A set of valid test cases are designed to test the correctness of the system
    - Each test case includes an expected result based on what the specification says should happen when that test case executes.
    - We can think of these test cases "ensuring the system does what is supposed to do"
  - A set of Invalid test cases are designed to test the robustness of the system
    - As with the valid test cases, an expected result is part of the test case
    - We can think of these test cases as "ensuring the system doesn’t do something it shouldn’t"

# Functional Testing

- Process
  - The system under test (SUT) is run with both the valid and invalid test cases provided as input
  - The actual output produced by the SUT for each test case is recorded and compared to the expected output previously calculated for that test case
  - For each test case where the actual output matches the expected output, the test case is recorded as a pass. When the actual output does not match the expected output, the test case is recorded as a failure
- Advantages
  - Can be applied to any level of testing (unit, integration, etc)
  - Does not require any knowledge about the internals of the system being tested
    - Refactoring of the architecture and code does not require rewriting functional tests
  - Supports regression testing

# Functional Testing

- The coverage dilemma
  - Functional testing is a trade-off between
    - Effectiveness: Using test cases that have the highest probability of identifying defects
    - Efficiency: Testing uses the limited testing resources, including time, in an optimal manner
  - We can get 100% test effectiveness by testing every possible input and condition, but since there are often an infinite number of possible inputs, we wind up with 0% test efficiency
  - If we do no testing at all, we have achieved 100% efficiency but 0% effectiveness
- Surprise functionality
  - We write tests based on what the specification tells us the system is supposed to do
    - But if there is functionality that is not documented in the specification
    - Then we can’t write tests for it because we don’t know it exists
    - Functionality that is not documented in the specification is called "surprise functionality"
    - We need to supplement functional testing with structural testing to deal with surprise functionality

# Functional Testing

- Security implications of surprise functionality
  - Obviously, attempts to subvert a system are not documented in the specification
- Malicious surprise functionality
  - *System Back Doors:* System security bypassed in an undetectable manner to give unauthorized access to the system or its data
  - *Logic Bombs:* Code that has been secretly inserted into the code base that will do some sort of damage to the system,when triggered by either some event
  - *Trojan Horses:* A program that provides some clearly stated and useful technology but also contains code that does something else surreptitious

# Specification Testing

- The specification is a description of the system to be built
  - Describes interfaces to the system, its functionality and any other data required to create a comprehensive design for building the system
  - Functional tests are written by using the behavior described in the spec as a baseline
  - If the specification is not good enough to write a test to, then it is not good enough to use as a guideline for construction
    - E.g. If I can't predict the outcome of an input to write a test, then I don't know what the system should do in that case, which means I have no idea what my code should do in that case
- IEEE Best Practices for Software Requirements Specification
  - Industry gold standard for what a spec needs to write good test cases
  - Developers should push back against poor specs
  - Otherwise they have to guess at what their code should do

# Specification Testing

- The basic properties for a spec to have, according to the IEEE

| | |
|---|---|
| **Complete** | *Specification description covers all possible alternatives that can occur, both valid and expected as well as invalid and unexpected.* |
| **Consistent** | *No two spec items require to system to behave differently when the state of the system and input conditions are the same.* |
| **Correct** | *The result of each spec item meets the acceptance criteria.* |
| **Testable** | *All inputs, states, decisions and outputs are quantified and measurable.* |
| **Verifiable** | *There is a finite cost effective process for executing each test case alternative.* |
| **Unambiguous** | *There is only possible way to interpret or understand each spec item.* |
| **Valid** | *Everyone can read, understand, and analyze the soec well enough to formally approve the described items.p* |
| **Modifiable** | *The spec items are organized in a way so that they are easy to use, modify and update.* |
| **Ranked** | *The spec items are in a priority order that everyone on both the business and technical sides agree on.* |
| **Traceable** | *Every spec item can be traced back to an example and acceptance criteria that motivated it and then back to the original requirement.* |

# Testing Methods

- Returning to the concept of test efficiency
- There are a number of ways to reduce the amount of testing without significantly impacting the effectiveness of the testing
- This is just a high level introduction
- Equivalence classes
  - Method for grouping test cases by identifying various classes of input conditions with the assumption that each member of a class causes the same kind of processing and output to occur as any other member of the class
  - Assumes that inputs that are subject to the same logic in the spec are processed by the same code – assumes that the code is competently written

# Testing Methods

- A group of tests forms an equivalence class if
  - They all test the same thing
  - If one test case in an equivalence class detects a defect, then every other test case in the same equivalence class will detect the same defect
  - If one test case in an equivalence class fails to detects a defect, then every other test case in the same equivalence class will fail to detect the same defect
- Reduces an infinite number of inputs to a finite number of classes
  - The we write a test using a representative data point from that class
  - Turns out to be very effective in real life when coupled with other methods

# Testing Methods

- Boundary Value Analysis
  - The boundaries between equivalence classes is where many defects occur
  - Usually decision logic errors
    - For example, does “up to four” mean up to and including four or up to but not including four?
  - Can be applied to any data that can be ordered
  - Test cases are chosen by taking
    - The largest valid value in the class and the smallest valid value in the class
    - The smallest value that is too large
    - The largest value that is too small
    - Other values around the boundary as necessary


| BOUNDARY VALUE ANALYSIS | | |
|---|---|---|
| **Invalid (min -1)** | **Valid (min, +min, -max, max)** | **Invalid (max +1)** |
| 17 | 18, 19, 55, 56 | 57 |

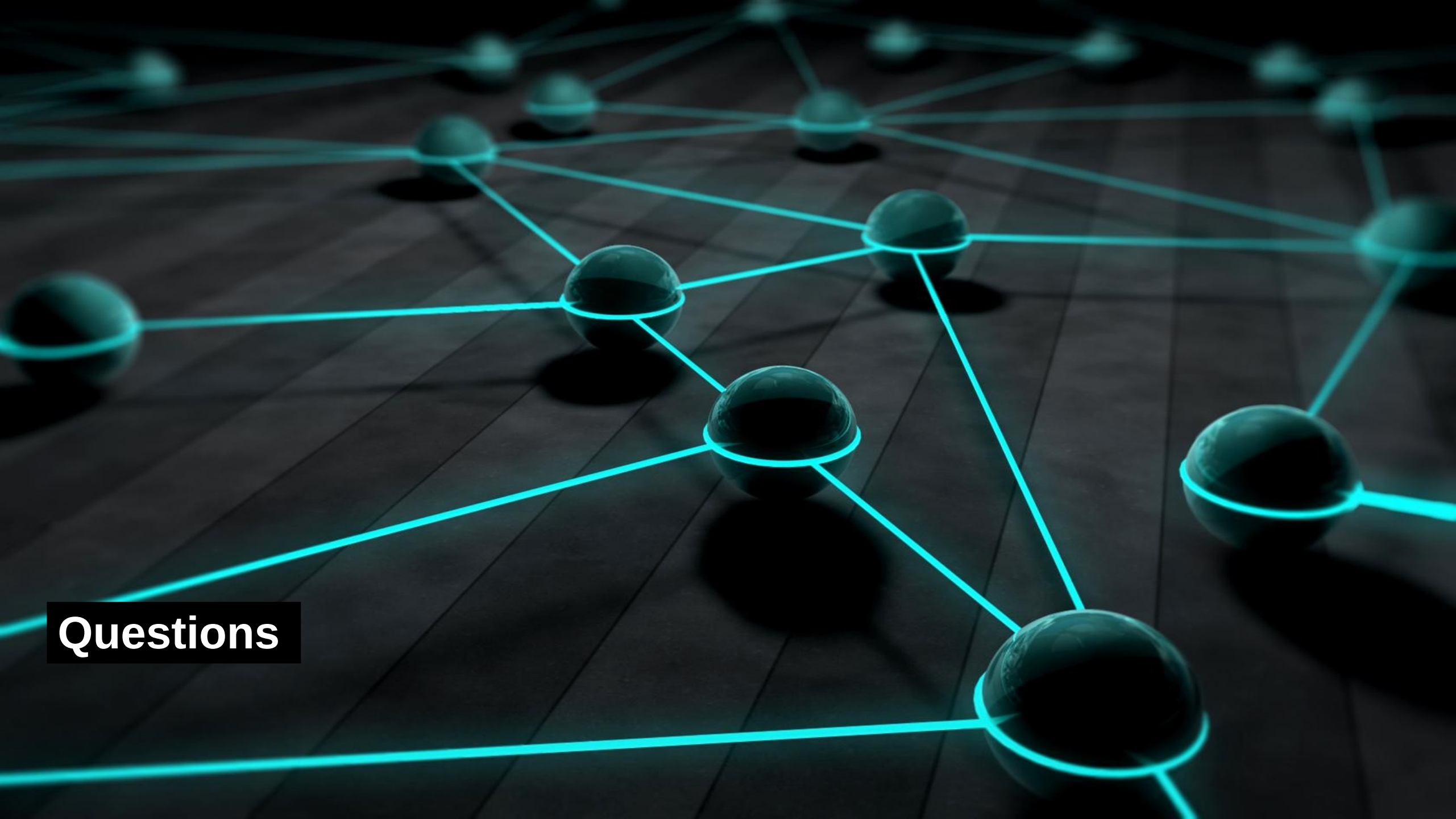
# Generating Unit Tests

- Unit tests are not generated in isolation
- They are derived from the acceptance tests
  - The acceptance tests and system architecture determine the inputs for a component
  - The acceptance tests that would result in an call to the component are isolated
  - The input each test provides is associated with what the correct output should be
  - This usually reduces the number of inputs
- The unit tests remain stable unless there is change in the architecture or the acceptance tests for the system

# Generating Unit Tests

- IEEE Best Practices expressed in terms of acceptance tests

| | |
|---|---|
| **Complete** | *The acceptance tests cover all possible inputs and conditions, both valid and expected as well as invalid and unexpected.* |
| **Consistent** | *No two acceptance tests require to system to behave differently when the state and input of two tests are the same.* |
| **Correct** | *The expected result of each test meets the acceptance criteria.* |
| **Testable** | *All test inputs, states and outputs are quantified and measurable.* |
| **Verifiable** | *There is a finite cost effective process for executing each test.* |
| **Unambiguous** | *There is only possible way to interpret or understand each test and test result.* |
| **Valid** | *Everyone can read, understand, and analyze the tests well enough to formally approve the tests.* |
| **Modifiable** | *The test cases are organized in a way so that they are easy to use, modify and update.* |
| **Ranked** | *The test cases are in a priority order that everyone on both the business and technical sides agree on.* |
| **Traceable** | *Every test case can be traced back to the example and acceptance criteria that motivated it and then back to the original requirement.* |


Questions