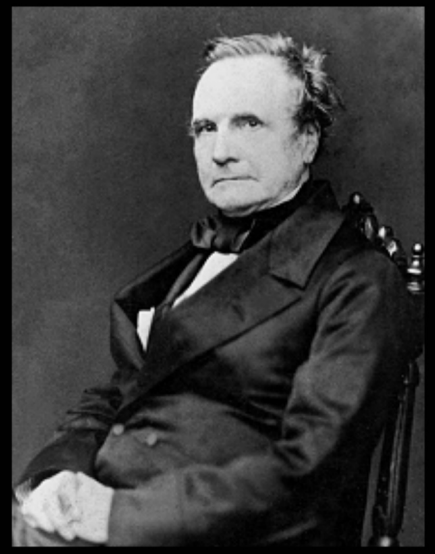
4. Testing Principles for TDD

Java Test Driven Development with JUnit

Module Topics

1. Good Enough Quality
2. Efficient and Effective
3. Failures, Faults and Errors
4. Unit and Acceptance Tests
5. Test First – Robust Test Cases and Code
6. Creating Unit Tests for TDD

On two occasions I have been asked, ‘If you put into the machine wrong figures, will the right answers come out?’ I am not able rightly to apprehend the kind of confusion of ideas that could provoke such a question.



Charles Babbage

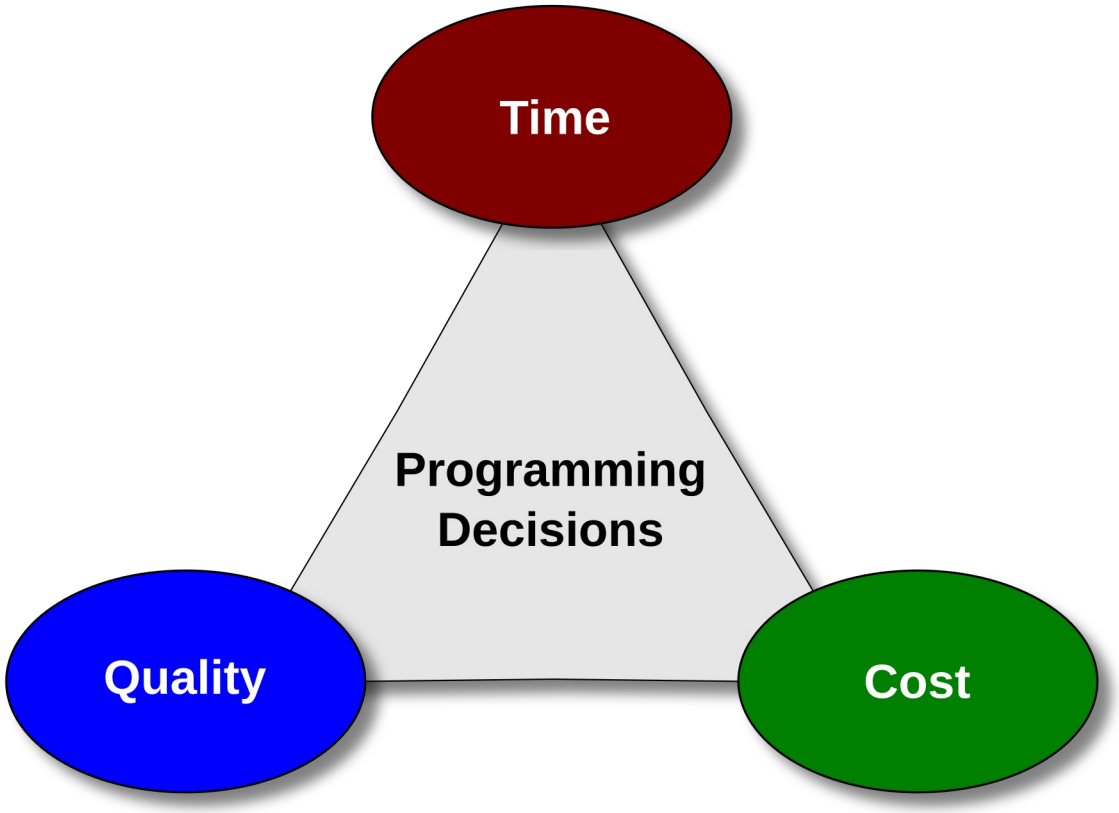
Correctness is clearly the prime quality.

If a system does not do what it supposed to do, then everything else about it matters little.

Bertrand Meyer

Good Enough Quality

The Iron Triangle of Software Development



The Iron Triangle

Often used to illustrate the difference between programming as an activity and software engineering – sometimes described as “software engineering is programming under the constraints of time, resources and meeting customer expectations.” This is often also expressed as “you can have the result good, fast and cheap – pick any two.”

GEQ - Good Enough Quality

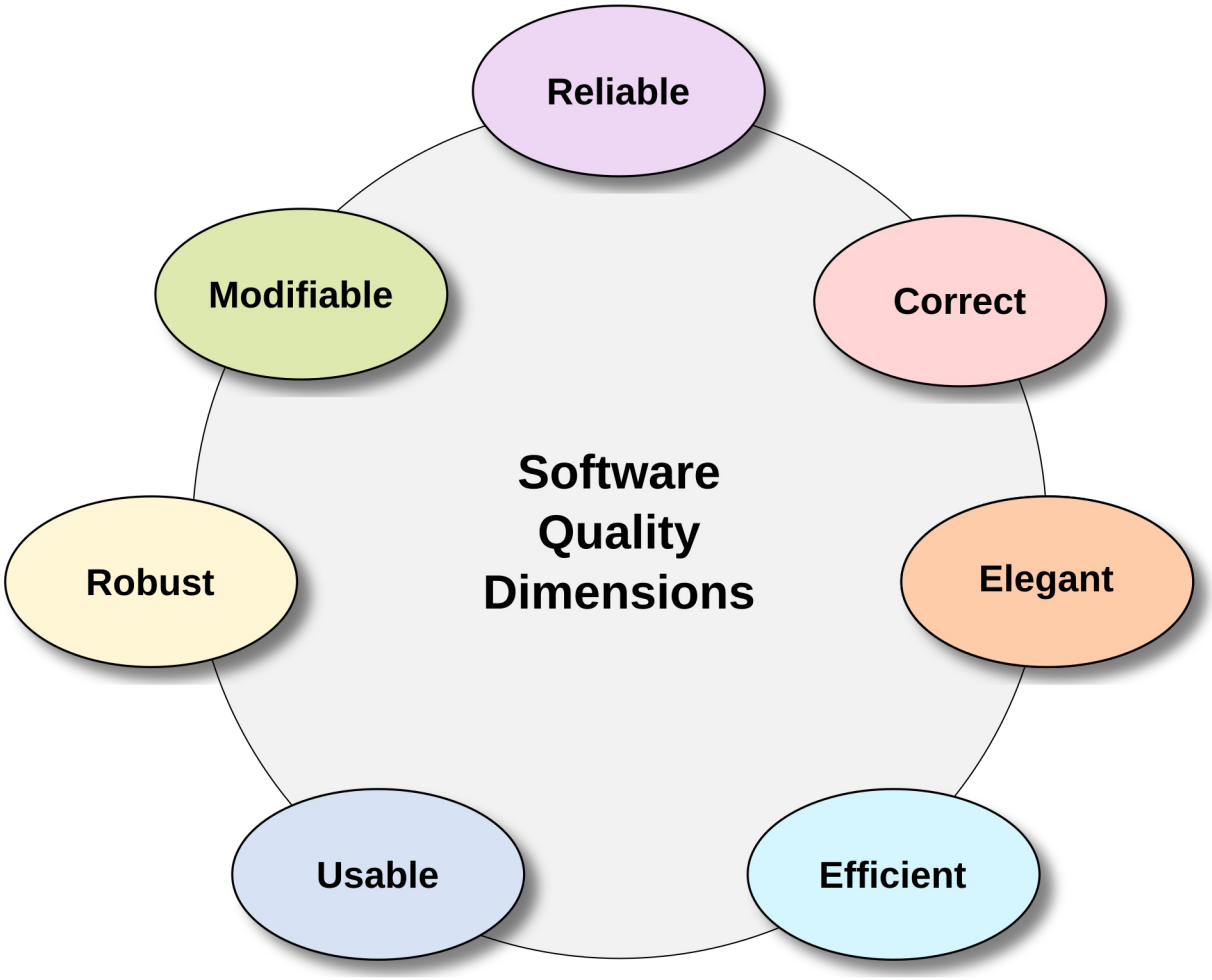
- Programming decisions have to be made within a“Good Enough Quality” context
  - *Agile view of quality: how much quality to we really need?*
  - *Decided at the inception of the project by the team*
  - *Allows us to estimate how much work we need to do to meet GEQ*
- Knowing GEQ means we can figure out:
  - *How robust the code needs to be*
  - *How complete the code needs to be*
  - *How much testing needs to be done*
- GEQ allows development goals to be set and prioritized

GEQ Decisions

Making GEQ Decisions

Deciding on how to meet the goals of the GEQ requires balancing three dimensions: the resources available, the required quality levels and the risk analysis of the software not meeting the quality requirements. Making it more challenging is that these are not nice orthogonal dimensions but they are all inexorably interdependent.

Software Quality Dimensions



Dimensions of Software Quality

These various concepts of software quality are discussed in the student manual. While there is no definitive list of dimensions of software quality, these are the dimensions that are often mentioned in the literature. A more detailed discussion is provided in the student manual.

Risk Analysis

Severity / Frequency	Catastrophic	Critical	Marginal	Negligible
Frequent	Extreme	High	Serious	Moderate
Probable	High	High	Serious	Moderate
Occasional	High	Serious	Moderate	Low
Remote	Serious	Moderate	Moderate	Low
Improbable	Moderate	Moderate	Moderate	Low
Prevented	None	None	None	None

Risk Analysis

The risk level is a combination how bad the outcome of an event would be and how often it would occur. What we generally do in what is called risk mitigation is to reduce the likelihood of the even occurring. This is reflected in the amount of testing we do to ensure that the event is prevented from happening. We usually have no control over the severity.

Beizer's Phases

- Boris Beizer is famous for his “Five Phases of a Tester’s Mental Life”

  0 *There’s no difference between testing and debugging. Other than in support of debugging, testing has no purpose.*

  1 *The purpose of testing is to show that the software works.*

  2 *The purpose of testing is to show that the software doesn’t work.*

  3 *The purpose of testing is not to prove anything, but to reduce the perceived risk of a software not working to an acceptable value.*

  4 *Testing is not an act. It is a mental discipline that results in low-risk software without much testing effort.*

- This describes how testers change in their view of what testing does as they become more experienced and skilled.

Beizer's Phases Adapted

- Adapting Beizer to TDD:
  - *0 Code is written and then tested only when it doesn't work.*
  - *1 Code is written to positive tests to ensure correctness – the right result is produced for valid inputs.*
  - *2 Code is written to negative tests as well to ensure robustness – bad or unexpected conditions and inputs are handled gracefully by the code.*
  - *3 Code is written to tests that satisfy the specified levels of GEQ*
  - *4 While developing the tests, possible sources of error are identified and eliminated, code is tested to GEQ levels as it is written*

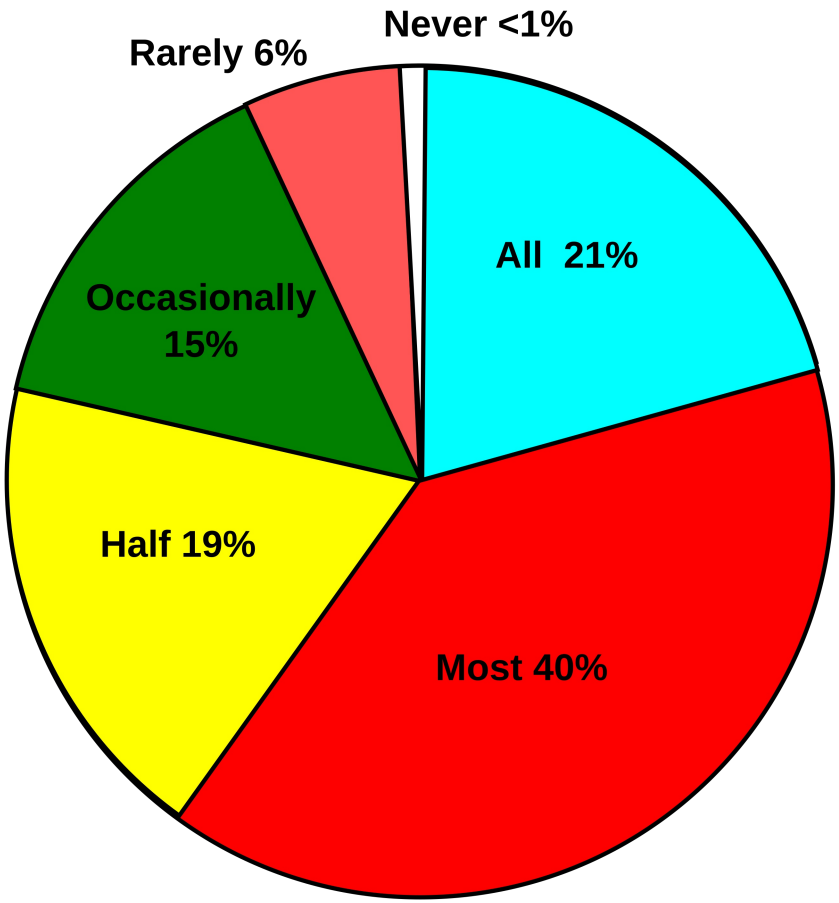
- Each level includes the levels below it, except for level 0 which becomes a separate debugging activity

Efficient and Effective

Effective Process

- According to the IEEE, an effective software development process
  - *Produces the correct result as determined by the client*
  - *Meets all of the stakeholder requirements and specifications*
  - *The final product passes all the acceptance tests*
- If we have an effective process we do not waste our time and effort building the wrong thing
- Validation is the term used when we test to see if the correct software is being built

The Problem of Rework



Geneca Study on Rework

Geneca surveyed a broad cross section of IT professionals and asked them “What percentage of your time do you spend doing rework that could have been prevented?”

80% of those asked spent at least half their time doing rework that could have been prevented – in other words, work they should never have had to do.

Efficient Process

- According to the IEEE, an efficient software development process
  - *Produces software with optimal use of resources*
  - *Examines the output of each task to ensure the result is correct*
  - *Correct errors as they occur*
- If we have an efficient process, errors do not persist
  - *Error that persist cause further errors later in development*
  - *Internal measures are used as opposed to customer acceptance*
  - *Rework occurs as we correct old errors and their effects*
- Verification is the process of evaluating and correcting the outputs at each step

TDD: Effective and Efficient

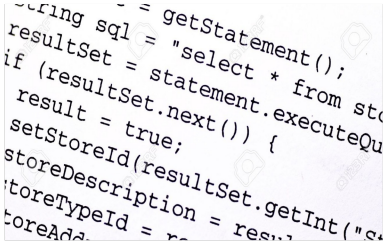
- TDD improves both effectiveness and efficiency in coding
- Defining the tests correctly ensures the right code is written
  - *Only code that delivers the required functionality is written*
  - *No code that does not contribute to the functionality is written*
- Running the tests continuously ensures errors are corrected as they occur.
  - *The unit tests are the local verification criteria*
  - *Refactoring allows improving code quality with continuous verification*

Failures, Faults and Errors

Root Causes of Defects

Defects

causes



causes

Error **Fault** **Failure**

Error: a human action that eventually leads to a fault

Fault: an incorrect step in building the system at any point that results in failure

Failure: any place the software does not perform as required

Defect: a generic term for any of the above

TDD: Fixing the Defects

- TDD focuses on fixing both the issues of faults and errors
  - *Continuous automated unit testing eliminates faults*
  - *Correct preparation of tests eliminates errors*
- Basic TDD axiom:
  - *“If you can’t write a test to describe what your code should do, then you don’t know enough to write code that functions correctly.”*
- The rest of the module will focus on what “good tests” are

Examples of Errors

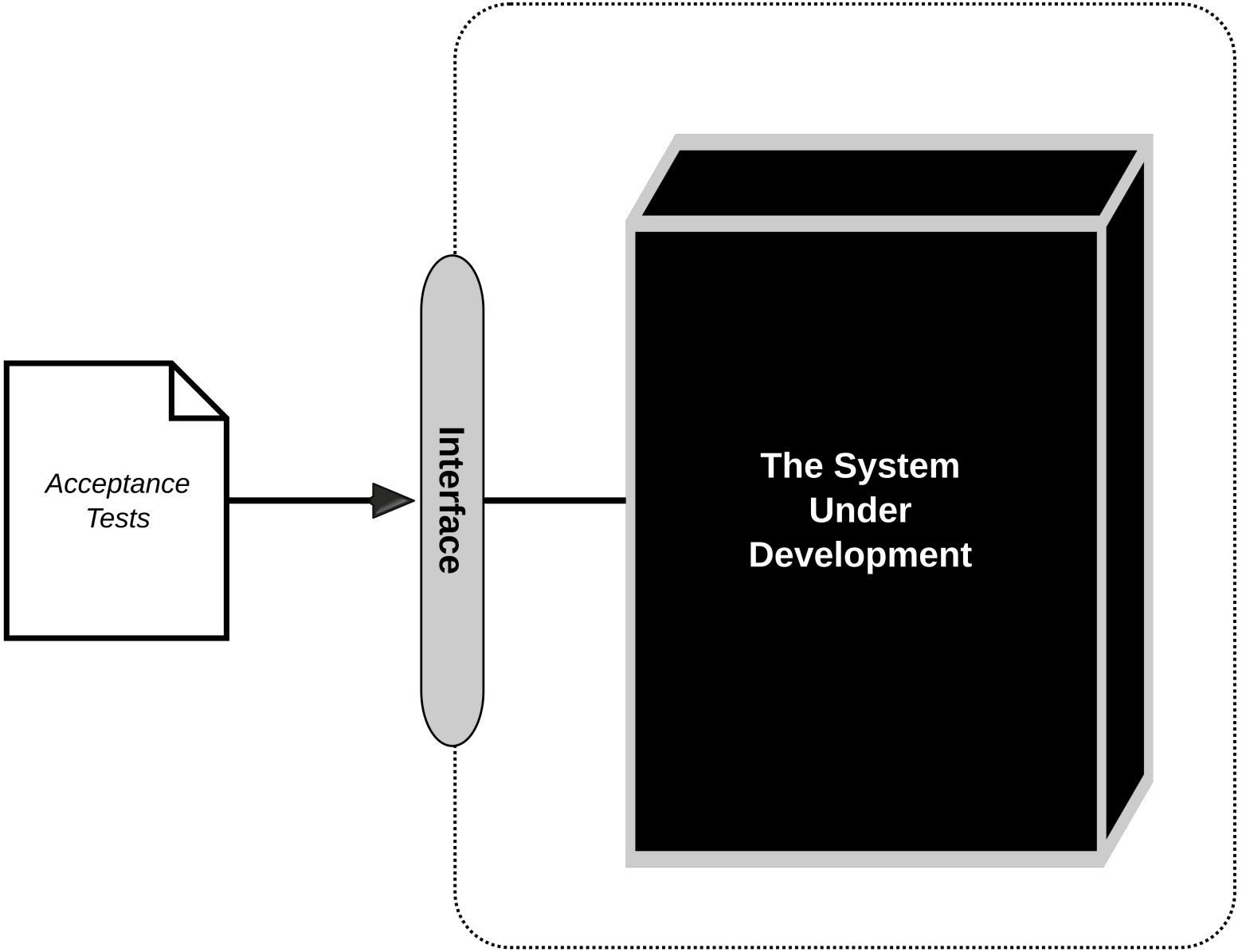
- Some examples of the kinds of errors we want to correct:
  - *Communication errors – ambiguity, misunderstanding etc.*
  - *Misunderstood requirements – solving the wrong problem*
  - *Missing cases – things that "fell through the cracks"*
  - *Ambiguous or vague specs – not clear what should happen in some cases*
  - *Impossible specs – system has to do contradictory things*
  - *Meandering design because the end result is not clearly defined*
  - *Over-engineering – development of features no one wants or will use*
  - *Outliers – failure to account for valid but unusual cases*
  - *Scope creep – trying to unrealistically deal with all possible cases*
- These are just examples of the issues we try to prevent from happening

Unit and Acceptance Tests

Acceptance Tests

- A software specification describes the functionality of software
- In the Agile world, a main component of a spec is a set of acceptance tests
  - *The tests are run on the whole system frequently*
  - *Each run validates that system development is correct so far*
  - *Each acceptance test is a regular test case*
  - *The acceptance tests describe all possible types of inputs and the system response*
- Acceptance Test Driven Development (ATDD)
  - *ATDD is a discipline used to develop acceptance tests*
  - *We will just assume that a set of acceptance tests exists.*

Acceptance Tests



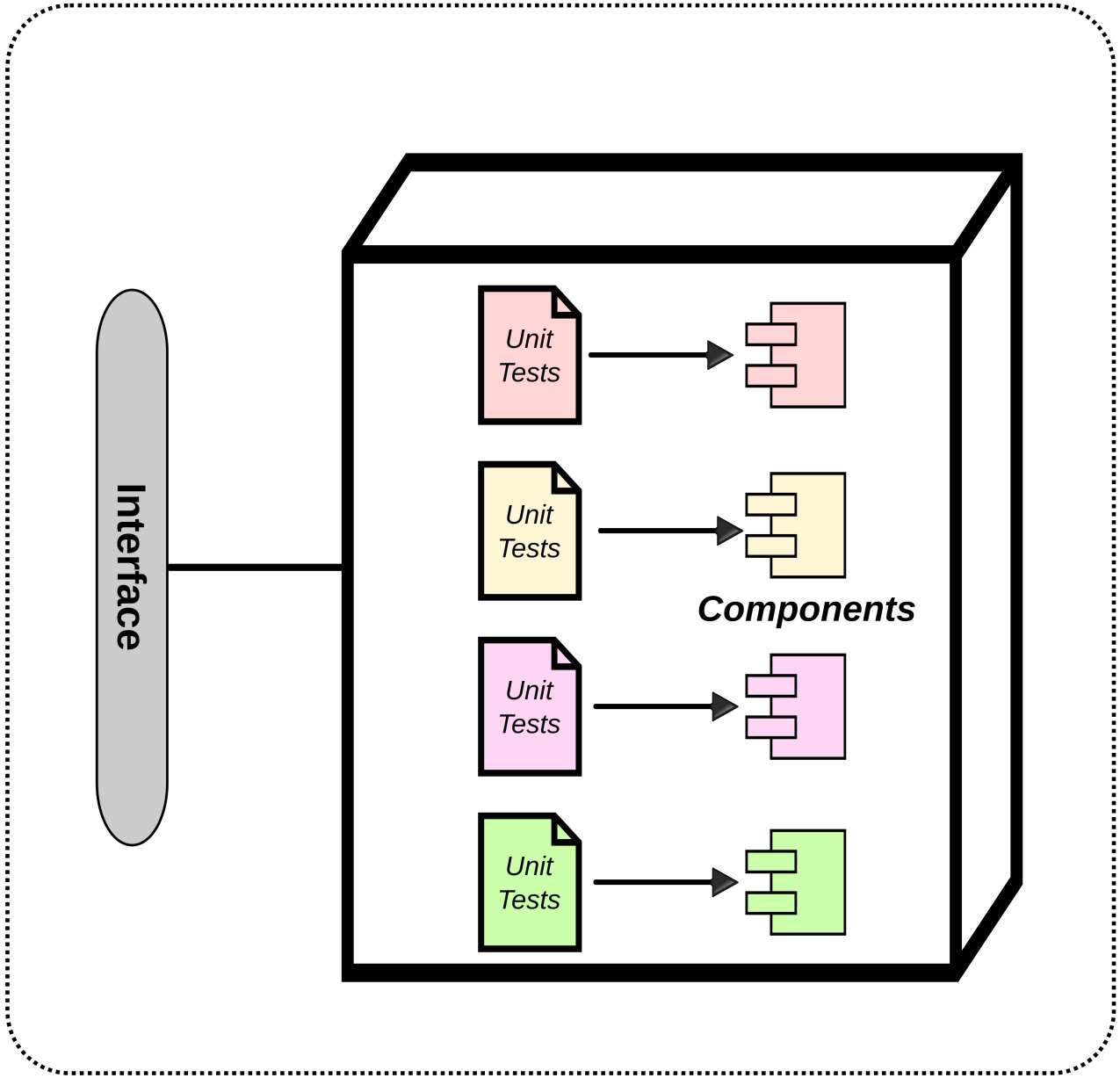
Unit Tests

- Unit testing is often taken to mean testing the smallest component – not true

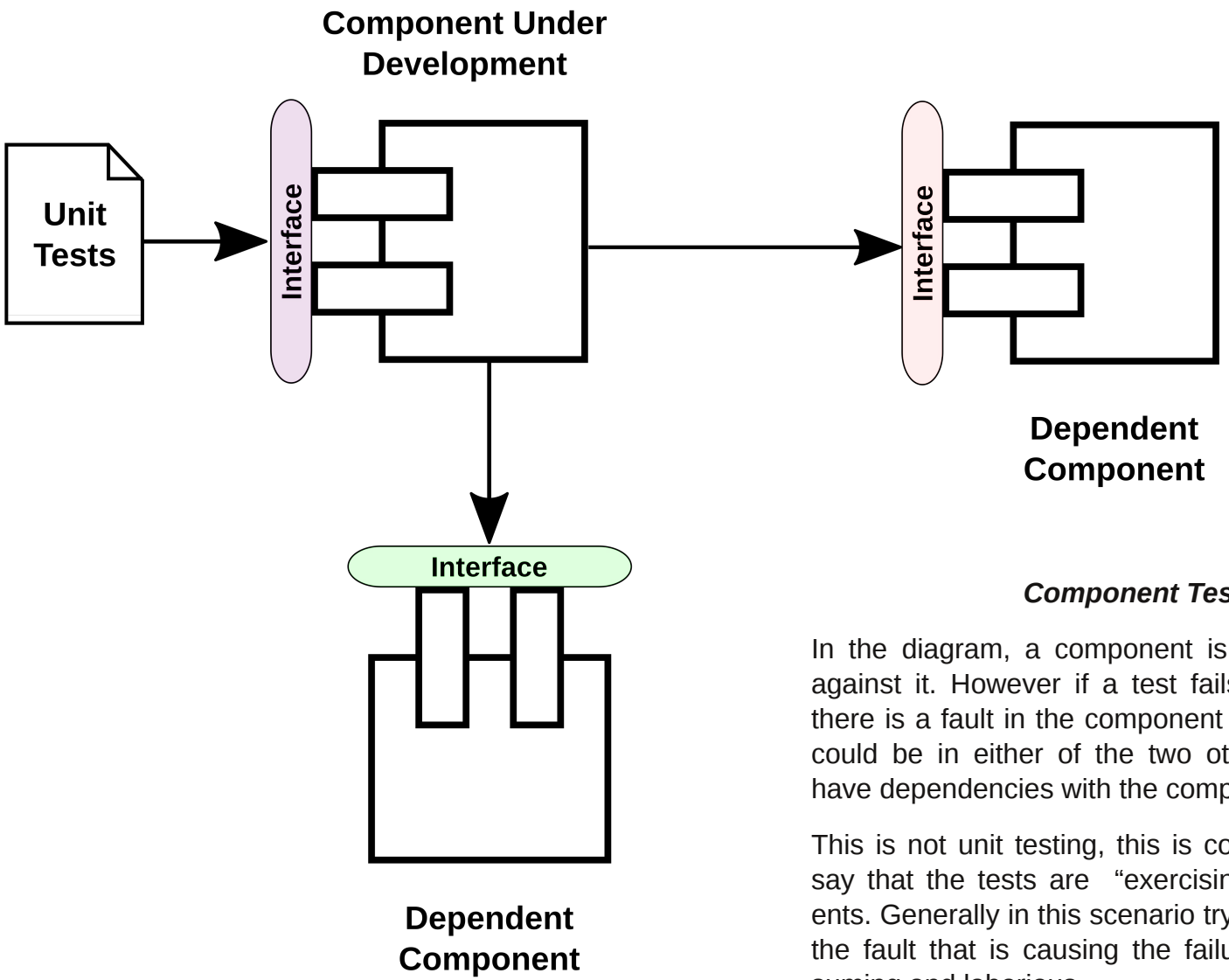
- Unit testing means testing a single component in isolation
  - *That means that results of the test depend only on that component*
  - *Components of any size can be unit tested*
  - *However in practice, the tend to be the smaller components*

- Unit testing requires a controlled test environment
  - *We replace other components with mock ups*
  - *The mock ups all perform perfectly*

Unit Tests


Interface
Unit Tests
Unit Tests
Components
Unit Tests
Unit Tests

Component Tests with Dependencies

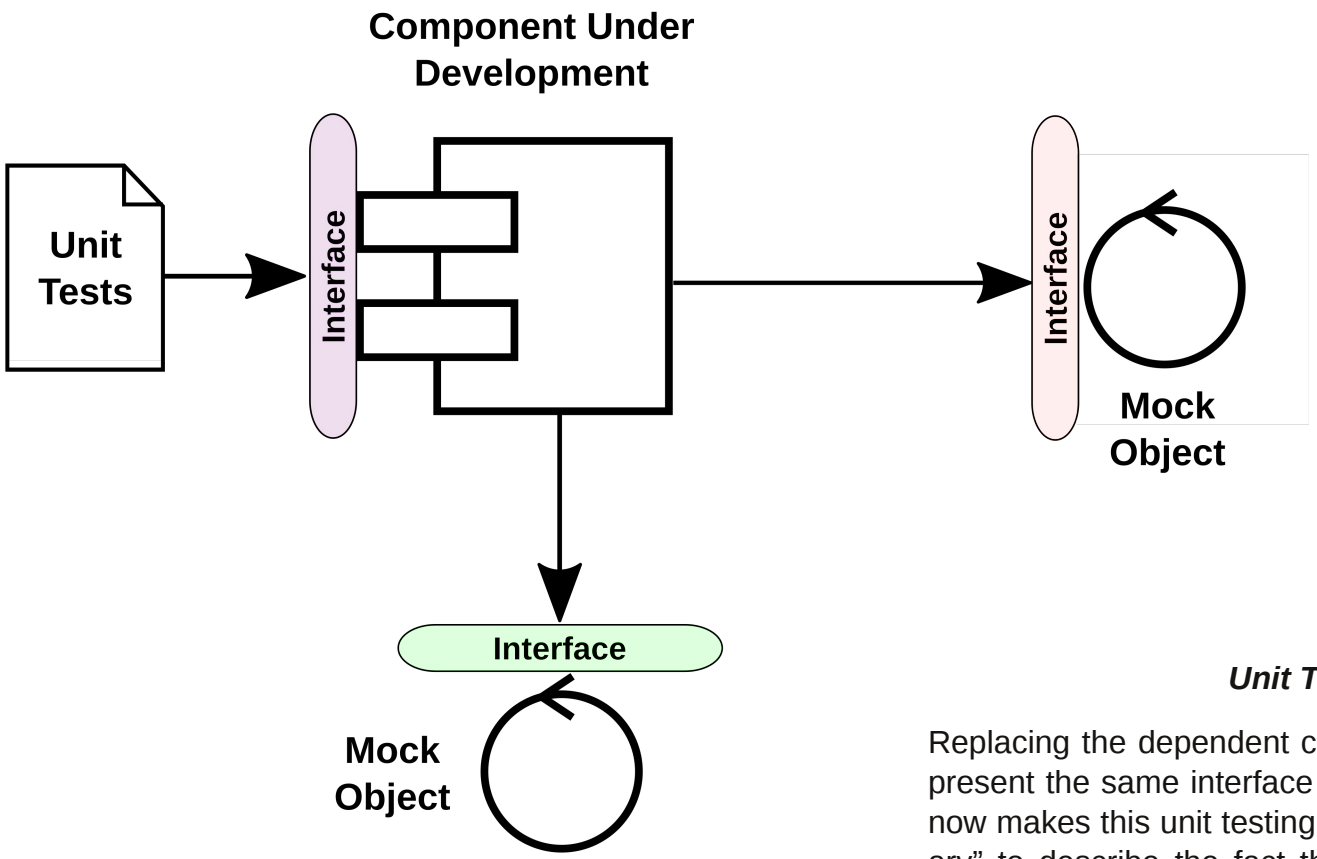


Component Testing

In the diagram, a component is having unit test run against it. However if a test fails, it is not clear that there is a fault in the component under test – the fault could be in either of the two other components that have dependencies with the component under test.

This is not unit testing, this is component testing. We say that the tests are "exercising" all three components. Generally in this scenario trying to debug and find the fault that is causing the failure can be time consuming and laborious.

Unit Tests with Mocks

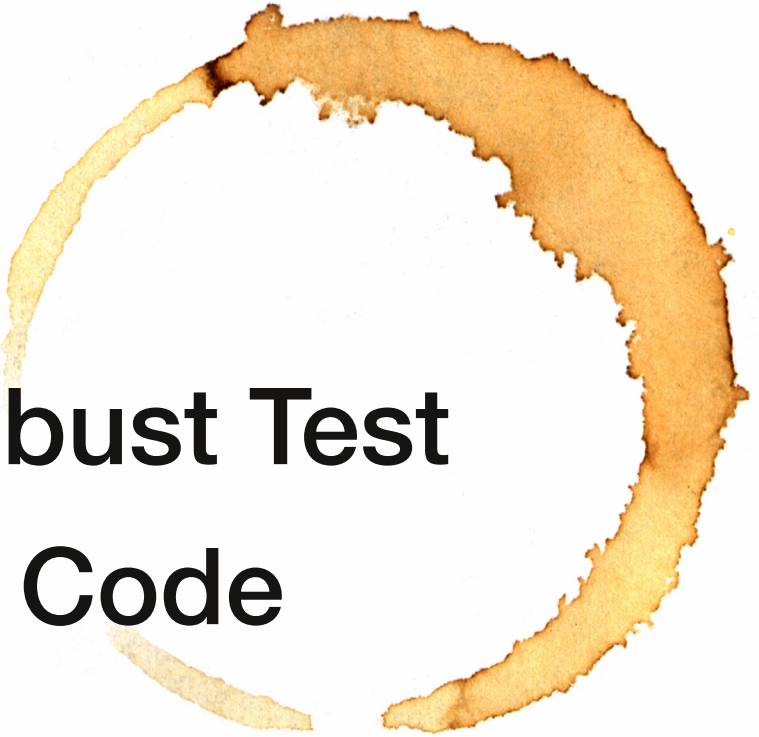


Unit Testing

Replacing the dependent components with mocks that present the same interface as the original components now makes this unit testing. The term unit means “unit-ary” to describe the fact that only one thing is under test.

Because each mock will work perfectly for the test cases, any failures must be due to faults in the component under development.

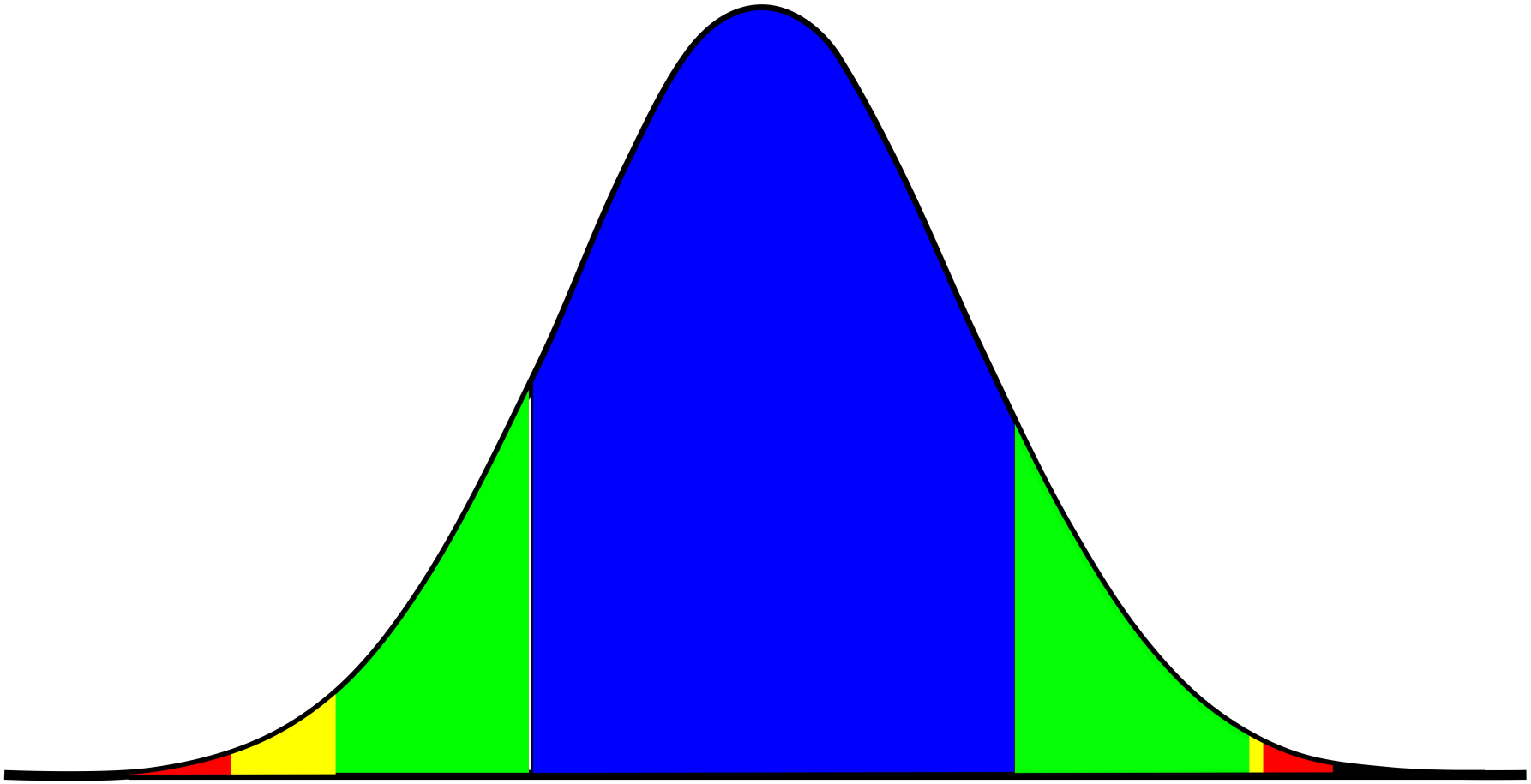
Test First – Robust Test Cases and Code



Developing Code

- When developing software programmers tend to:
  - *Write code to process the most common or frequent or “normal” cases.*
  - *Code is then added to handle some of the less common cases*
  - *Code may be added to add some of the cases that should not occur*
  - *Code is rarely added to handle outliers – valid cases that are rare*
- The expansion of the code to cover these cases is haphazard
  - *When programmers write tests after writing code the follow the same logic the used to write the code*
  - *This usually leaves gaps in what we call coverage*
  - *This is also called following the garden path*

Distribution of Cases



Invalid - should not occur

Outliers - valid but very rare

Uncommon - valid but not frequent

Garden Path - valid and most commonly occuring

Test First

- By writing the tests first:
  - *The full range of cases have to be considered and how each case will be handled by the system*
  - *We set a scope as to what cases we need to handle and which ones we should reject*
  - *We identify cases that are not covered in the requirements*
  - *We develop an understanding of what should be done in each case*
- A couple of benefits:
  - *We have a general sense of the overall design of the code to be written*
  - *We have an estimate of the amount of work required*
  - *We have corrected a number of error before we begin work*

Developing Code

- When developing software programmers tend to:
  - *Write code to process the most common or frequent or “normal” cases.*
  - *Code is then added to handle some of the less common cases*
  - *Code may be added to add some of the cases that should not occur*
  - *Code is rarely added to handle outliers – valid cases that are rare*
- The expansion of the code to cover these cases is haphazard
  - *When programmers write tests after writing code the follow the same logic the used to write the code*
  - *This usually leaves gaps in what we call coverage*
  - *This is also called following the garden path*

Creating Unit Tests for TDD

Deriving Unit Tests

- Unit tests guide the development of the code
- They must be a full set of acceptance tests for the component
  - *The must be complete, correct, etc.*
  - *They must be consistent with the system level acceptance tests*
- They must also test architectural constraints
  - *These are the factors that affect the component based on how it is designed to interact with other components*
  - *For example, is it possible for a parameter to receive a null pointer rather than a valid input object*
  - *This can not be determined by looking at the system level acceptance tests*

Acceptance Tests Derivations

- Unit tests guide the development of the code
- They must be a full set of acceptance tests for the component
  - *The must be complete, correct, etc.*
  - *They must be consistent with the system level acceptance tests*
- They must also test architectural constraints
  - *These are the factors that affect the component based on how it is designed to interact with other components*
  - *For example, is it possible for a parameter to receive a null pointer rather than a valid input object*
  - *This can not be determined by looking at the system level acceptance tests*

Acceptance Tests Derivations

- If a development process is being followed, XP for example
  - *There are system acceptance tests*
  - *These define all possible inputs to the system and the correct responses that should be made*
  - *Each of those acceptance tests may require a contribution from the component under development to produce a result*
  - *Based on the expected value of the acceptance test, we can compute what the expected correct contribution of the component should be*

- This is generally not a 1-1 relationship
  - *Multiple acceptance tests may require the same contribution from the component*
  - *Some acceptance tests may not require any contribution from the component*
  - *These derived unit tests are a minimal set of unit tests needed for development*

Example

User Name	Password	Result	AC tested
Mango	Passw0rd!	Pass	Valid
2Mango	Passw0rd!	Fail	Bad User Name
Man go	Passw0rd!	Fail	Bad User Name
M@ngo	Passw0rd!	Fail	Bad User Name
Mango	Password!	Fail	Bad Password
Mango	Passw0rd	Fail	Bad Password
Mango	P0!	Fail	Bad Password

Login System Example

The application being developed is a log in utility which has the set of acceptance tests shown above. The column labeled "AC" is the acceptance criteria that explains the result of the test. If we assume that these are a complete set of acceptance tests, then if we are writing the component that validates the password, these tests describe all of the input types that our component will have to deal with.

However, this reasoning only applies for this specific system and how our component will behave within this design. If we want to build some sort of re-usable component to be used in other designs, we will need to use a much more comprehensive and robust approach to developing test cases.

Example

Password	Result
Passw0rd!	True
Password!	False
Passw0rd	False
P0!	False

Login System Example

The example now shows the minimal required set of unit tests derived from the application acceptance tests. The set is minimal because if we drop any of the tests, then our unit tests will not cover an input type and they will not be complete.

However, we may also note that architecturally, the data will be passed as a String object so we may want to add another unit test to cover the possibility that we might get a null pointer. The null pointer unit test case is not based on the acceptance criteria but an understanding of the application design.

End of Module 4