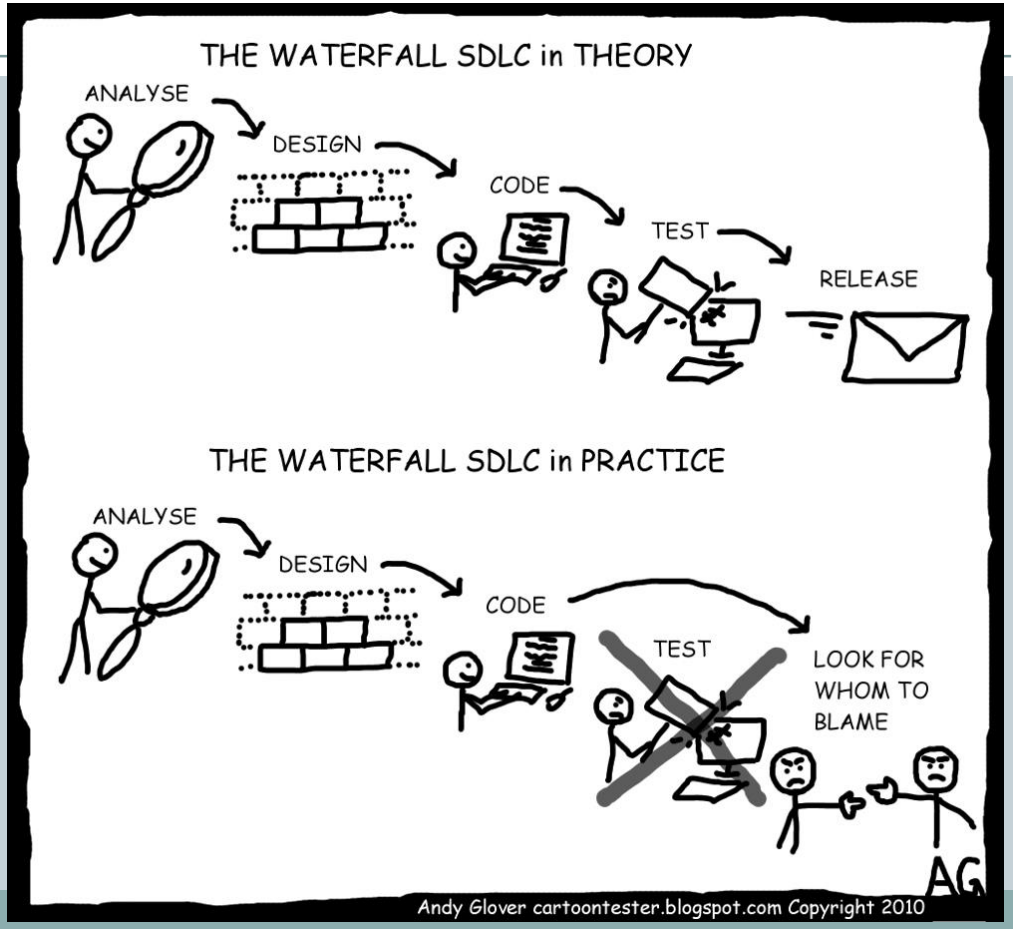
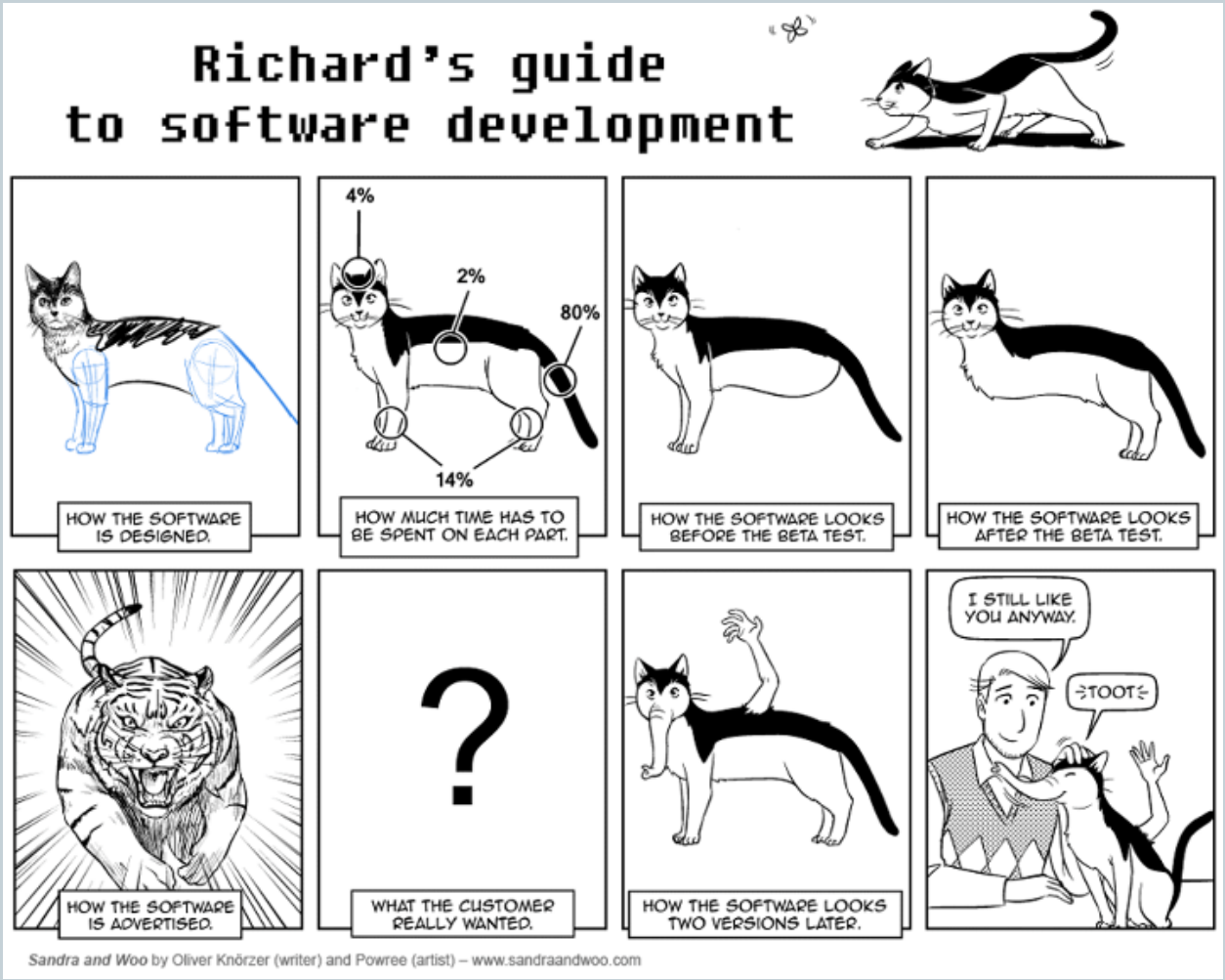
Software Testing



Outline

- Software Quality
- Unit Testing
- Integration Testing
- Acceptance Testing



“Quality” is Hard to Pin Down

- Concise, clear definition is elusive
- Not easily quantifiable
- Many things to many people
- *“You'll know it when you see it”*

Good Quality Software Has…

- **Understandability**
  - The ability of a reader of the software to understand its function
  - Critical for maintenance
- **Modifiability**
  - The ability of the software to be changed by that reader
  - Almost defines "maintainability"

Good Quality Software Has…

- **Reliability**
  - The ability of the software to perform as intended without failure
  - If it isn't reliable, the maintainer must fix it
- **Efficiency**
  - The ability of the software to operate with minimal use of time and space resources
  - If it isn't efficient, the maintainer must improve it

Good Quality Software Has…

- **Testability**
  - The ability of the software to be tested easily
  - Finding/fixing bugs is part of maintenance
  - Enhancements/additions must also be tested
- **Usability**
  - The ability of the software to be easily used (human factors)
  - Not easily used implies more support calls, enhancements, corrections

Good Quality Software Has…

- **Portability**
  - The ease with which the software can be made useful in another environment
  - Porting is usually done by the maintainer

Notice all related to maintenance but these qualities need to be instilled during development

Why Test?

- No matter how well software has been designed and coded, it will inevitably still contain defects
- Testing is the process of executing a program with the intent of finding faults (bugs)
- A “successful” test is one that finds errors, not one that doesn’t find errors

Why Test?

- Testing can “prove” the presence of faults, but can <u>not</u> “prove” their absence

- But can increase confidence that a program “works”

What to Test?

- ***Unit test*** – test of small code unit: file, class, individual method or subroutine
- ***Integration test*** – test of several units combined to form a (sub)system, preferably adding one unit at a time
- ***System (alpha) test*** – test of a system release by "independent" system testers
- ***Acceptance (beta) test*** – test of a release by end-users or their representatives

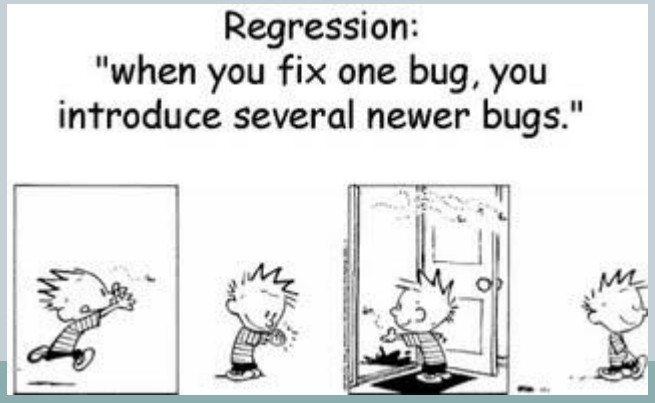
When to Test?

Early

- "Agile programming" developers write unit test cases *before* coding each unit
- Many software processes involve writing (at least) system/acceptance tests in parallel with development

Often

- *Regression testing*: rerun unit, integration and system/acceptance tests
  - After refactoring
  - Throughout integration
  - Before each release



Defining a Test

- Goal – the aspect of the system being tested
- Input – specify the actions and conditions that lead up to the test as well as the input (state of the world, not just parameters) that actually constitutes the test
- Outcome – specify how the system should respond or what it should compute, according to its requirements

Test Harness (Scaffolding)

- ***Driver*** - supporting code and data used to provide an environment for invoking part of a system in isolation

- ***Stub*** - dummy procedure, module or unit that stands in for another portion of a system, intended to be invoked by that isolated part of the system
  - May consist of nothing more than a function header with no body
  - If a stub needs to return values, it may read and return test data from a file, return hard-coded values, or obtain data from a user (the tester) and return it

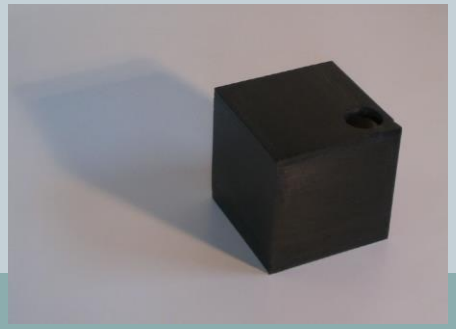
Unit Testing

Unit Testing Overview

- Unit testing is testing some program unit in isolation from the rest of the system
- Usually the programmer is responsible for testing a unit during its implementation
- Easier to debug when a test finds a bug (compared to full-system testing)

Unit Testing Strategies

- Black box (specification-based) testing
- White box (program-based) testing, aka glass-box
- Normally perform both (not alternatives!)



White Box Testing

- Test suite constructed by inspecting the *program* (code)
- Look at specification (requirements, design, etc.) only to determine what is an error
- Attempt to exercise all statements, all branches, or all paths (control flow and/or data flow)
- Intuition: If you never tested that part of the code, how can you have any reason to believe that it works?

Whitebox Approaches to Unit Testing

1. Execute all (reachable) statements
2. Execute all branches of logical decisions, including boundaries of loops
3. Execute all (feasible) control flow paths in combination
4. Execute all data flow paths (from each variable definition to all its uses)

- Usually applied only to individual subroutines rather than larger unit (due to combinatorics)

Example

- Consider a function that takes as input a string assumed to be a URL and checks to see if it contains any characters that are illegal
- Illegal URL characters are control characters (ascii 0-31, 127), space (ascii 32), and delimiter characters (">", "<", "#", "%", and the double quote character)
- The function returns true if the URL is valid (does not contain an illegal character), and false if the URL is invalid (contains an illegal character)

```
def isLegalURL (url):
    valid = True
    i = 0
    while i < len(url) and valid:
        c = url[i]
        if ord(c) >= 0 and ord(c) <= 32:
            valid = False
        else:
            if c == '>' or c == '<' or
               c == '#' or c == '%' or c == '\\':
                valid = False
        i += 1
    return valid
```

Black Box Testing

- Test suite constructed by inspecting the *specification* (requirements, design, etc.), not the source code
- Tests unit against functional and, sometimes, extra-functional specifications (e.g., resource utilization, performance, security)
- Attempts to force behavior (outcome) that doesn't match specification

Blackbox Approaches to Unit Testing

- Functional testing – exercise code with valid or nearly valid input for which the expected outcome is known (outcome includes global state and exceptions as well as output)
- Exhaustive testing usually infeasible, so need way(s) to select test cases and determine when "done" testing
- Choose test cases to attempt to find different faults
  - Equivalence partitioning
  - Boundary value analysis

Equivalence Partitioning

- Assume similar inputs will evoke similar responses
- *Equivalence class* is a related set of valid or invalid values or states
  - Valid inputs
  - Invalid inputs
  - Errors, exceptions, and events
  - Boundary conditions
  - Everything that could possibly break!
- Only one or a few examples are selected to represent an entire equivalence class
- Good for basic functionality testing

Equivalence Partitioning

- Divide input domain into equivalence classes
- Divide outcome domain into equivalence classes
  - Need to determine inputs to cover each output equivalence class
  - Also attempt to cover classes of errors, exceptions and external state changes

Boundary Value Analysis

- Consider input values that are “between” different expectations of functionality
  - Sometimes called “corner cases”
- Programmers tend to make common errors
  - Off-by-one
  - “<” instead of “<=”

Example

- A student must be registered for at least 12 points to be considered full-time
  - Full-time: some number 12 or greater
  - Not full-time: some number less than 12
- The method `isFullTime` takes an int and returns a boolean
- What inputs should we use to test it?

Another Example

- The function `stringSqrRoot` takes a String as input, converts it to a number, and returns that number's square root
- It throws an exception if the String is not numeric

- What inputs should we use to test it?

Automated Testing

- Testing by hand is tedious, slow, error-prone and not fun
- Computers are much less easily bored than people
- So write code to test your code!

Automated Testing

- Write code to set up the unit, call its methods with test inputs, and compare the results to the known correct answers
- Once tests are written, they are easy to run, so you are much more likely to run them
- Python library, unittest is a commonly used tool for testing

Unit Testing Summary

- Unit testing is testing some program unit in isolation from the rest of the system
- Usually the programmer is responsible for testing a unit during its implementation
- Strategies:
  - Black box (specification-based) testing
  - White box (program-based) testing
- Normally perform both (not alternatives!)

unittest

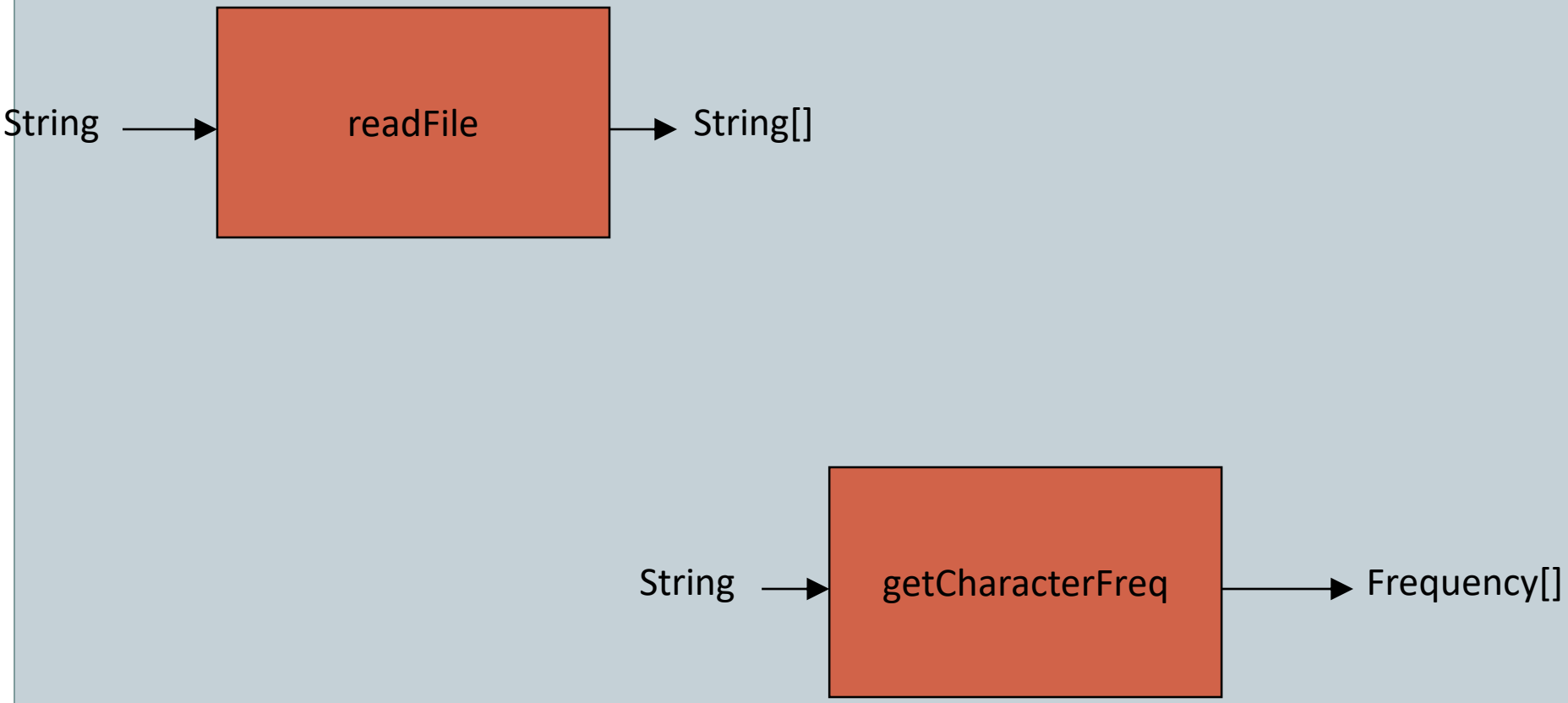
```
import unittest
from TestMe import isLegalURL

class URLTestCase(unittest.TestCase):

    def test_valid(self):
        self.assertTrue(isLegalURL('cs.mtech.edu'))

    def test_space(self):
        self.assertFalse(isLegalURL('cs.m tech.edu'))

# … and a whole bunch of other tests in here…

    def test_quote(self):
        valid = isLegalURL('cs.mtech"edu')
        self.assertEqual(valid, False)

    def test_valid2(self):
        valid = isLegalURL('www.google.com')
        self.assertEqual(valid, True)

if __name__ == '__main__':
    unittest.main()
```

Integration Testing

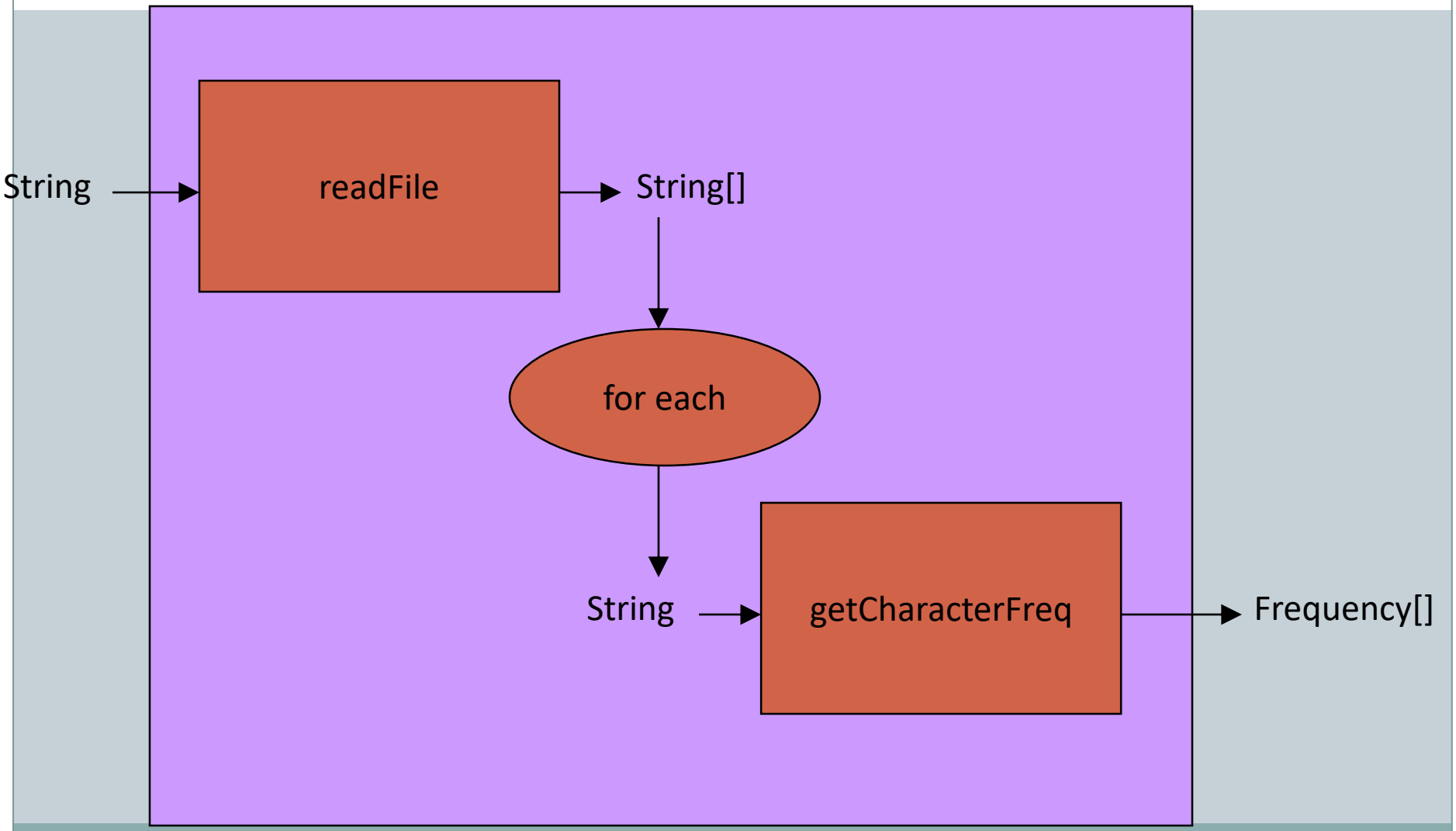
Integration Testing

- Performed after all units to be integrated have passed all black box unit tests
- Reuse unit test cases that cross unit boundaries (that previously required stub(s) and/or driver standing in for another unit)
- White box testing might be combined with integration as well as unit testing (tracking coverage)

Example: Two Units



Example: Integration Testing



System/Acceptance Testing

System/Acceptance Testing

- Also known as user testing
- All units in the system are combined into the final program/application
- Ensure that the system works the way that the user expects, i.e. that it meets the user specifications for functionality

System/Acceptance Testing

- Usually difficult to automatically mimic users' input (keyboard, GUI, etc.)
- Requires human users to try different input:
  - Valid vs. invalid actions
  - Various sequences of actions
  - Unanticipated actions

Summary

- Software Quality
- Unit Testing
- Integration Testing
- Acceptance Testing