University of
South Australia

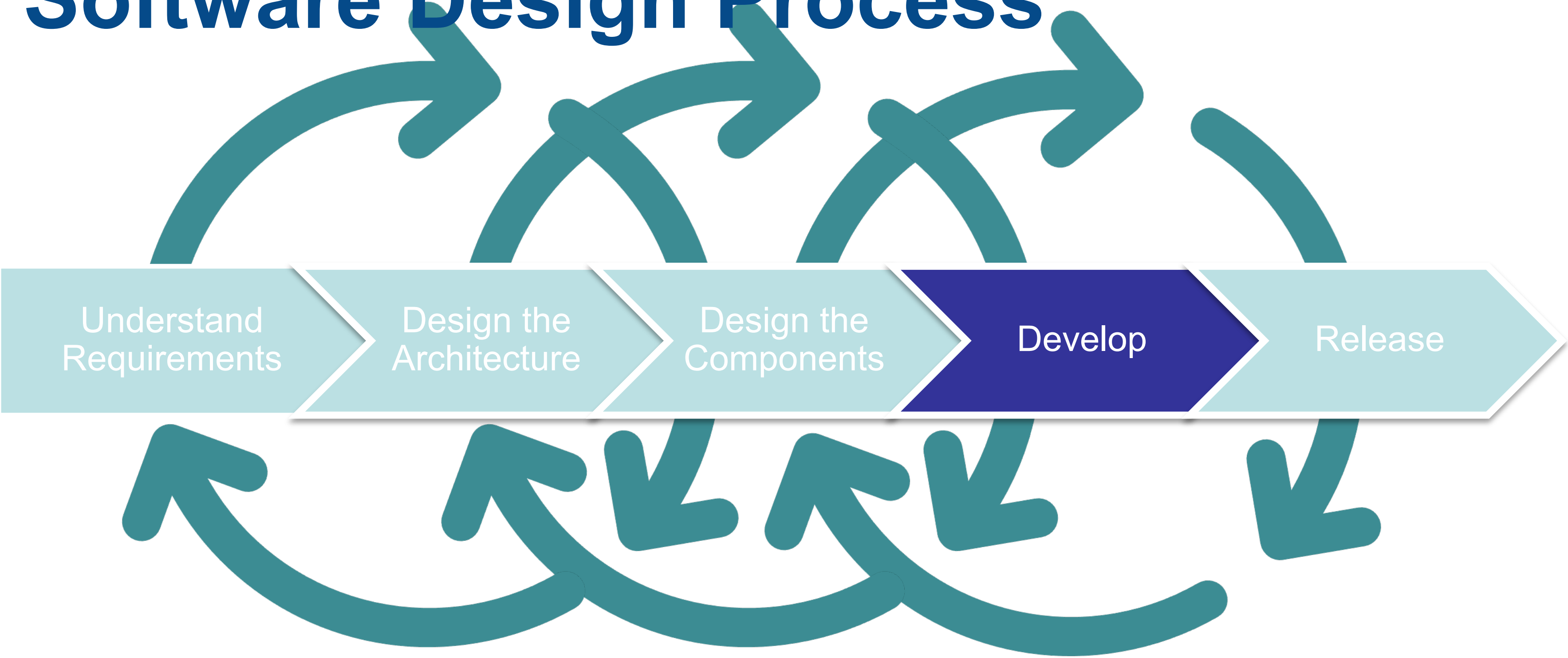
INFS 2044

Week 11
Testing

Learning Objectives

- Understand the purpose of software testing (CO5)
- Explain how software is tested (CO6)
- Understand how good design enables testing (CO6)

Software Design Process



Pillars of Clean Code

- Software design
  - Patterns, reuse, embrace design principles (S.O.L.I.D.), **test**
- Coding conventions
  - Naming things consistently
- Good habits
  - Continually improve code, don’t be clever

Validation vs Verification

- Verification
  - Does the software conform to the specification?
    (**Are we building the product right?**)
  - Code reviews, checking tools, formal methods, **unit testing**
- Validation
  - Does the software do what the user really requires?
    (**Are we building the right product?**)
  - Acceptance testing, a/b testing

Tests

- Tests make changes cheap
- Test code is just as important as production code
- Code coverage is not everything

Software Testing

- Aims to identify the correctness, completeness, and quality of software
- Process of executing a program under positive and negative conditions.
- Checks
  - Specification
  - Functionality
  - Performance

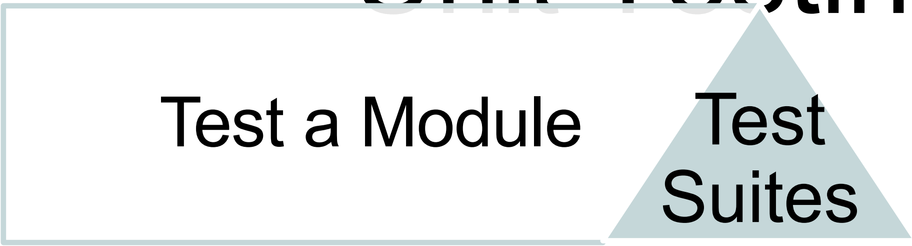
What to Test

- Usability: UX, look & feel, speed, user manual
- Functional: correctness of behavior & output, data validation
- Performance: (peak) load, data volume
- Security: access control, data protection

Scope of Tests

- Unit: test each module individually
- Integration: confirm that modules work together
- System testing: confirm that the system as a whole works as intended

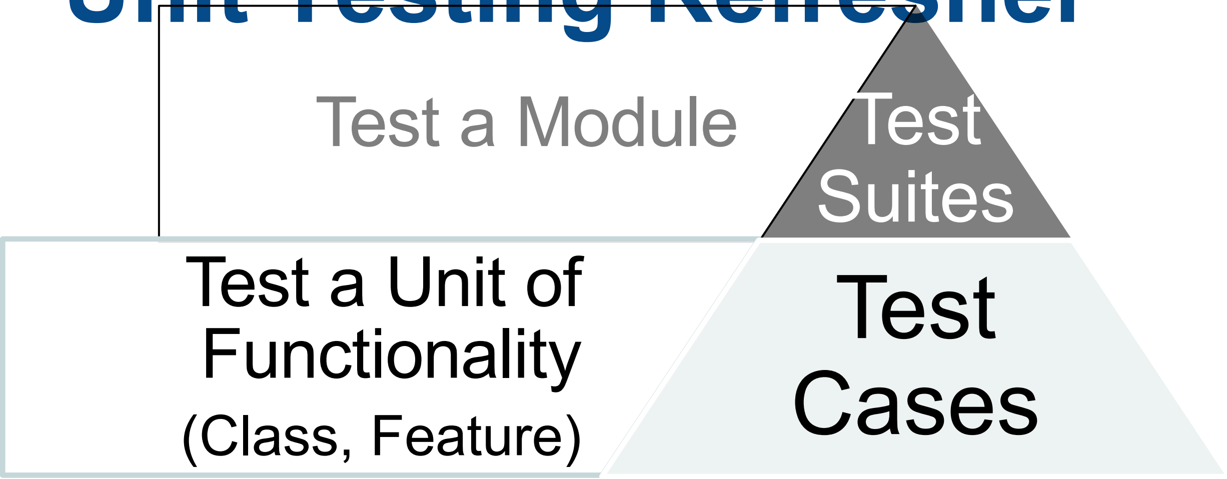
Unit Testing



Test a Module

Test Suites

Unit Testing Refresher



Test a Module

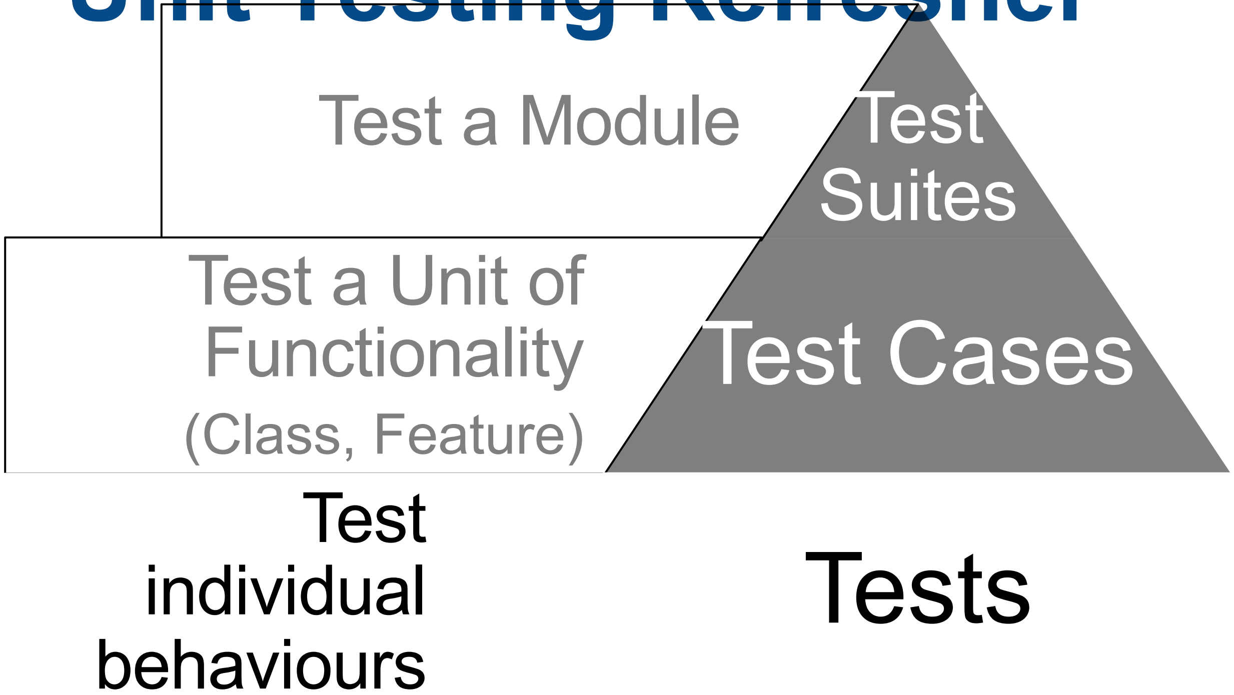
Test Suites

Test a Unit of Functionality

(Class, Feature)

Test Cases

Unit Testing Refresher



Test a Module

Test Suites

Test a Unit of Functionality

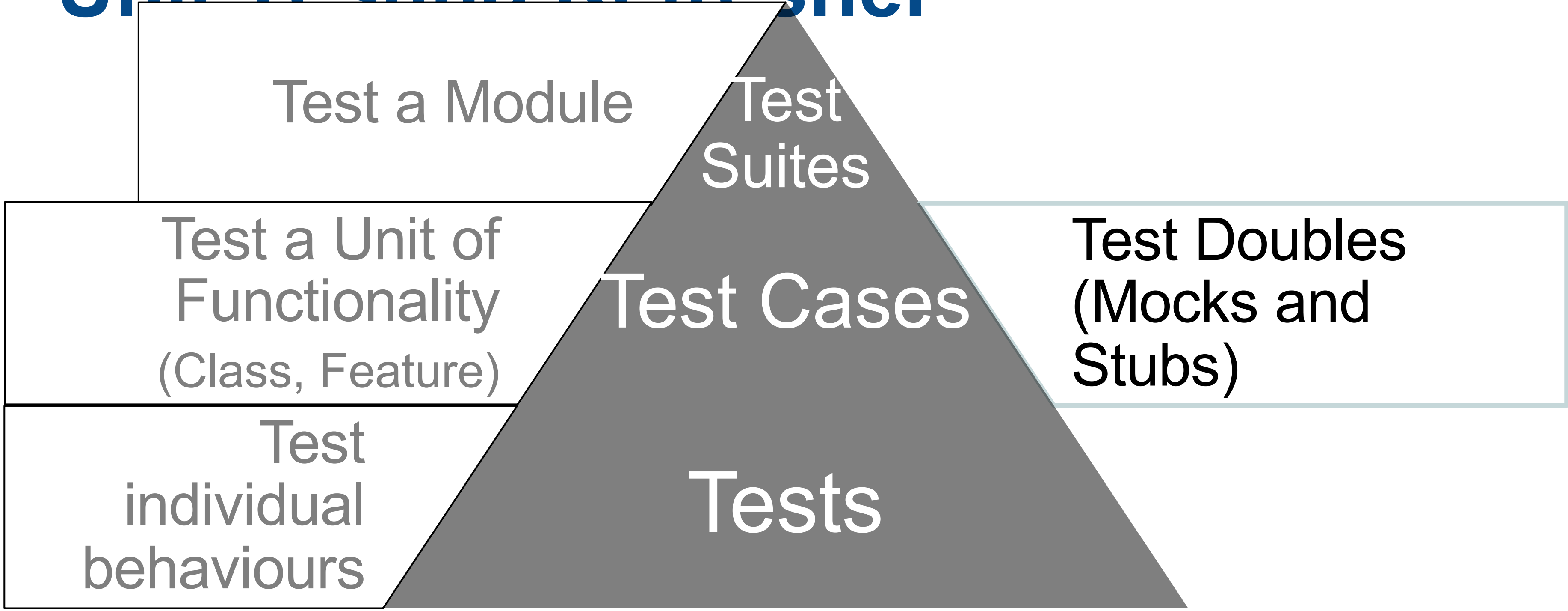
(Class, Feature)

Test Cases

Test individual behaviours

Tests

Unit Testing Refresher



Test a Module

Test Suites

Test a Unit of Functionality (Class, Feature)

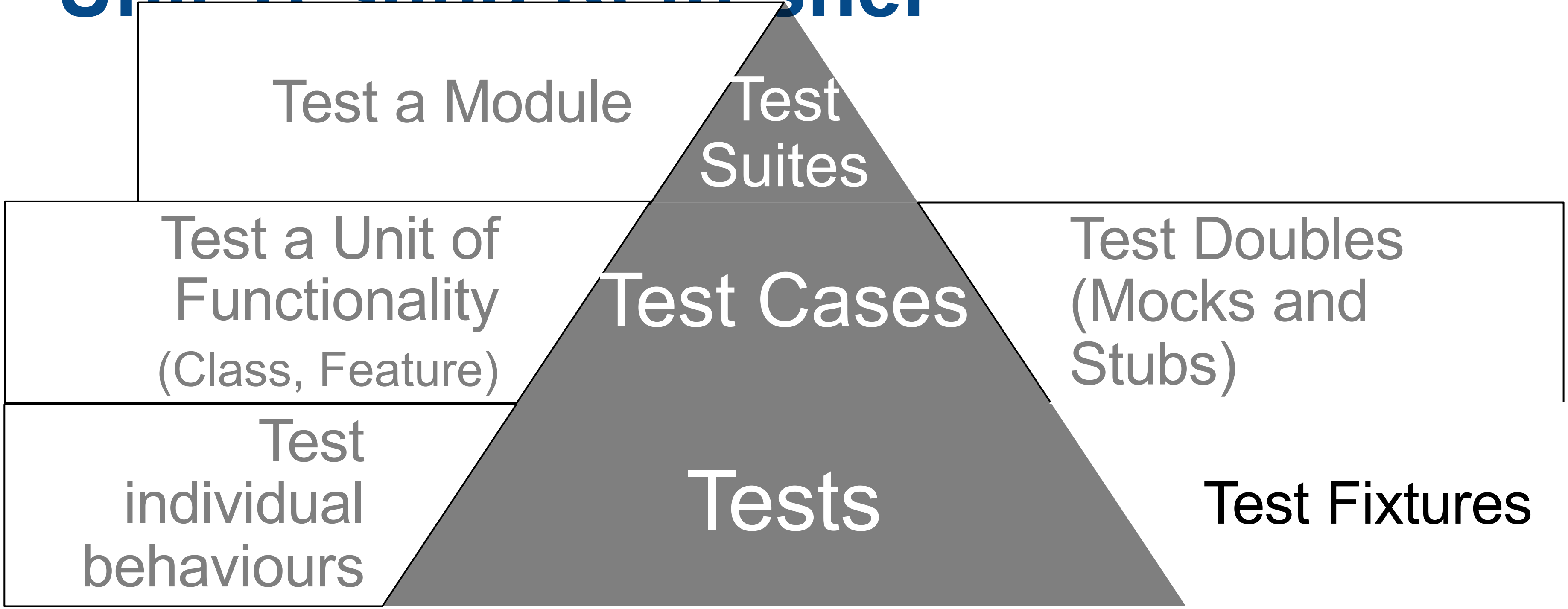
Test Cases

Test Doubles (Mocks and Stubs)

Test individual behaviours

Tests

Unit Testing Refresher



Test a Module

Test Suites

Test a Unit of Functionality (Class, Feature)

Test Cases

Test Doubles (Mocks and Stubs)

Test individual behaviours

Tests

Test Fixtures

Independence of Tests

Each Test must be independent of every other Test!

Guidelines for Good Unit Tests

- **One Test Case per Class** (or small number of strongly related classes)—easier to find/manage tests, execute specific subsets of tests, and prevent dependencies
- **Small Tests**—reduces dependencies between parts of the test, helps make the test more understandable and maintainable
- **Write tests for failure**—tests should initially fail, protects against accidental success

Guidelines for Good Unit Tests

- **Test Only One Thing**—separate assertions into different tests
  - Helps identify bugs when tests fail as you get more information about the error (normally after the first assertion fails a test ends)
  - The ideal is 1 assertion per test, but that is often not possible
- **Good names**—the name of a test should indicate what it tests, naming tests is easier if you stick to the Test Only One Thing guideline

What should/should not be tested?

- Should a struct with public data members be Unit Tested?
- Should constructors be Unit Tested?
- Should getters and setters be Unit Tested?
- Should a (relatively independent) class with core business logic be Unit Tested?
- Should a class with several dependencies and complex configuration be Unit Tested?

What Should/Must be Tested

- Anything with logic: business logic, validation logic, etc.
- Anything with defined or assumed pre-/post-conditions
- Anything that is critical to the success of the application or where failure would have undesirable consequences

Unit Tests are Good For

- Clarify Understanding of the Developer
  - What the code shall do and the expected outcome
  - The unexpected outcomes (negative testing)
  - Provide examples
- Regression testing
  - Allows you to safely refactor code confident that any errors caused by the changes will be detected
- Green ticks make you feel good about your code ☺

When to Write Tests?

- Before coding (Test/Behaviour Driven Development)
  - Focus on requirements
  - Think about how code will be used
  - Stop coding when requirements are met
  - Things are more likely to get tested this way
- After/During coding
  - Focus on code
  - Think about algorithm

Anatomy of a Unit Test

- Arrange (Setup, Fixture)
  - Setup code to establish the right context
- Act
  - Exercise the method under test
  - One line
- Assert
  - Verify the result

What We Can Test

- State
  - Assert that the correct result was achieved
- Behaviour
  - Assert that the correct sequence of methods was called

Test Stub

- A dummy piece of code that enables the test to run
  - Takes arguments and provides result to code under test

- Replaces code that has not been written yet
- Replaces code with side-effects
  - Networking, file system, external APIs, etc
  - Tested code calls on stub instead of the actual implementation

Stub Example: How to Test?

```
class Cart:
    def __init__(self,items):
        self.items = items

    def getTotalPrice(self):
        item_prices = [item.getPrice() for item in self.items]
        return sum(item_prices)
```

Stub Example

```
class ItemStub:
    def __init__(self, price):
        self.price = price

    def getPrice(self):
        return self.price
```

```
def test_total_price():
    stubs = [ItemStub(price)
                      for price in [5,11,2]]
    cart = Cart(stubs)
    total = cart.getTotalPrice()
    assert total == 18
```

Mock Objects

- Test *behaviour* of the code under test
  - Not just the final state/result

- Check that the code under tests calls the right method(s) of other objects
  - If not, the test fails

Mock Example: How to Tests?

```
def test_authentication():
    auth_handler = ...
    servie = MyService(auth_handler)
    service.doSomething(username='john', password='@#$%’)
```

Verify that MyService calls auth_handler.checkCredentials()

Mock Example

```
def test_authentication():
    auth_handler_mock = Mock(return_value=None)
    servie = MyService(auth_handler_mock)
    service.doSomething(username='john', password='@#$%’)
    auth_handler_mock.checkCredentials \
            .assert_called_once_with(username= 'john’, password='@#$%’
```

Design for Testing

- There is an interrelationship between design and testing
  - Good design is easier to test
  - Thinking about how a design will be tested can improve the design
- Design Principles supporting testing:
  - Dependency Inversion (Depend on Interfaces)
  - Dependency Injection (Pass-in dependencies)

Difficult to Test

```
class TodoListManager:
    def __init(self):
        self.accessor = TodoPostgresqlAccessor("db.todo-co.com")
    def addEntry(self, description):
        entry = self.accessor.createEntry(description)
        return entry
```

DI Example

```
class TodoListManager:
    def __init(self, accessor):
        self.accessor = accessor
    def addEntry(self, description):
        entry = self.accessor.createEntry(description)
        return entry
```

```
def test_add_entry():
    accessor_stub = Mock(return_value=1234)
    mgr = TodoListManager(accessor_stub)
    entry = mgr.addEntry("test your code")
    assert entry == 1234
```

Unit Tests Detect Functional Issues

- Unit Tests can find functional defects in the unit
- Other defects cannot be found easily with Unit Testing
  - Bugs arising from multiple units working together
  - Bugs with multi-threading (e.g., race conditions)—special case of the above
  - Issues with the application behaving/performing as expected by end users
- There are other forms of testing to address those
  - Integration testing, Performance testing, Acceptance testing

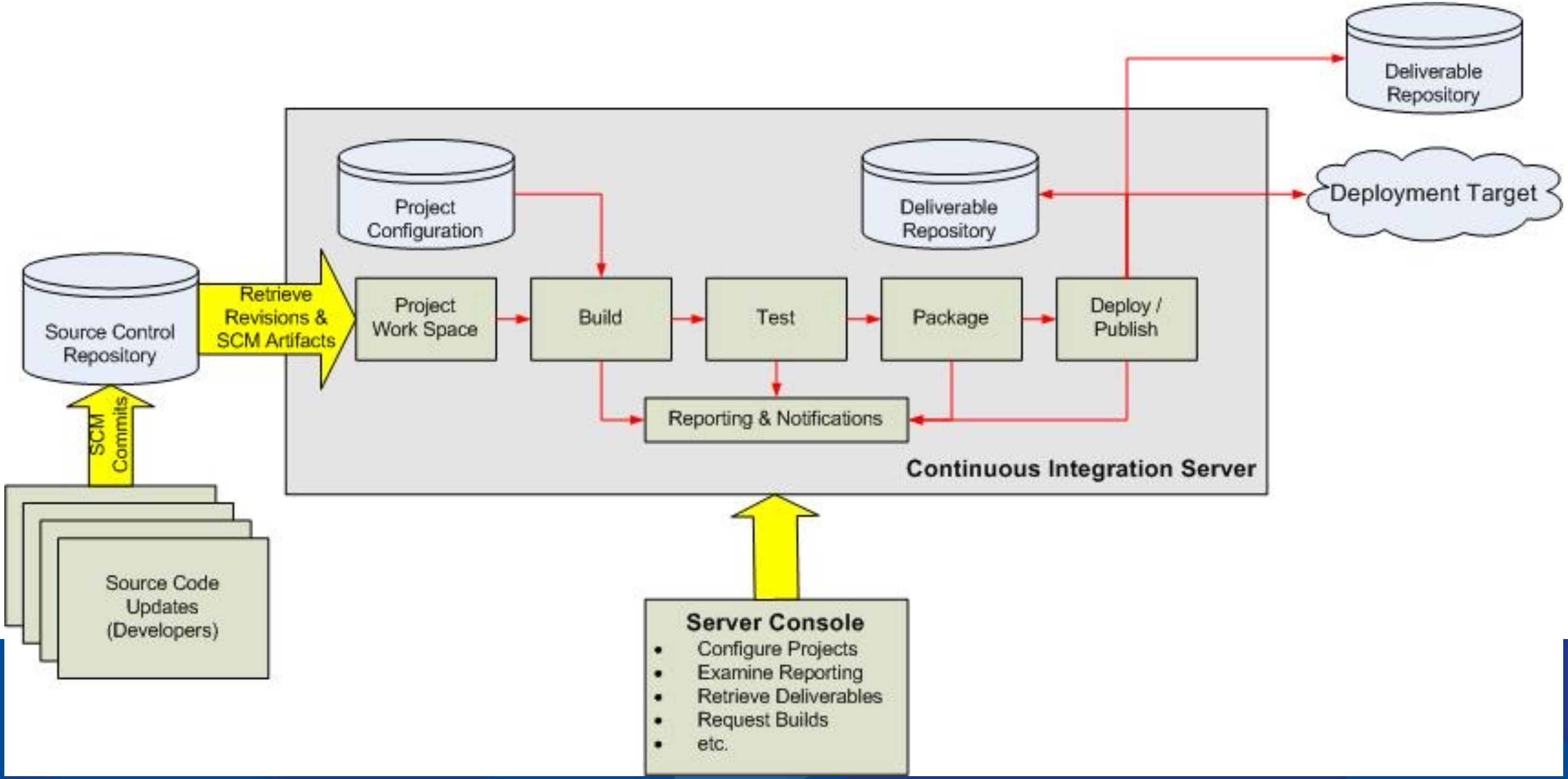
When to Test

- Unit Testing:
  - Before writing code
  - Before using third party modules (Learning Tests)
  - During development
- Integration Testing:
  - During development (commits)
- Acceptance Testing:
  - Prior to releasing the code

Continuous Integration

- Merge all developer working copies to a shared mainline several times a day
- Work is verified using automated build and testing processes to detect integration errors as quickly as possible
  - “Keep the build green”, always ready to release
- Reduces integration effort
  - Increases exponentially with number of components
  - Detect development problems early, reduce risks
  - Visible & measurable code quality

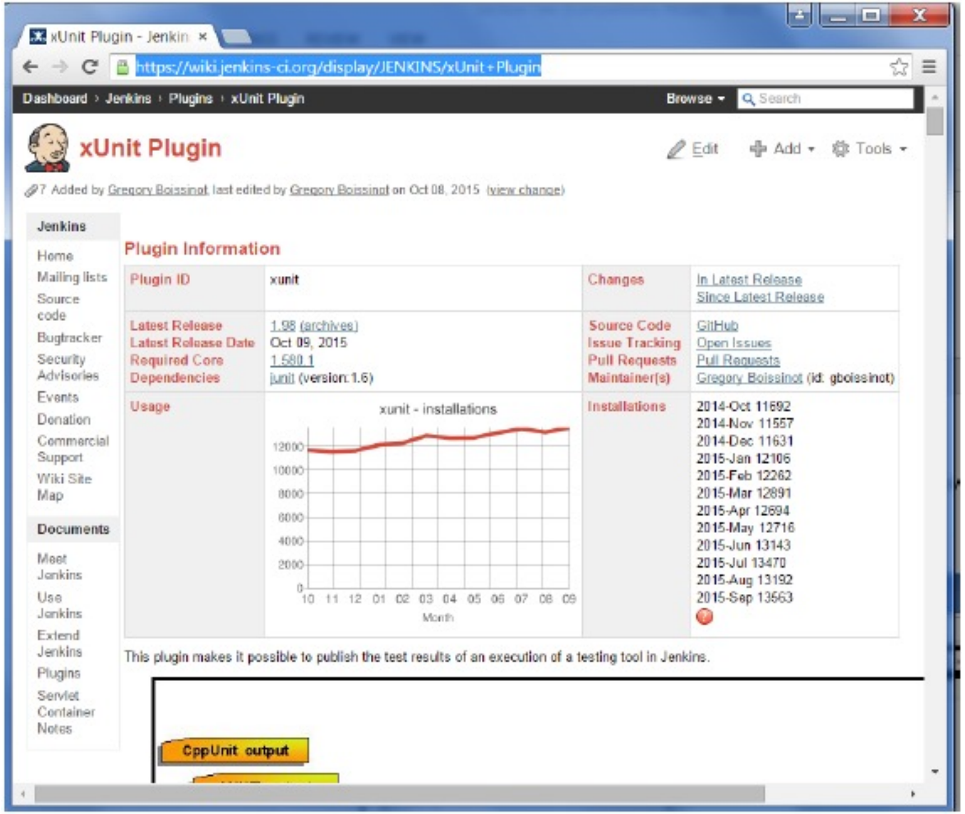
CI Process



http://www.javaworld.com/javaworld/jw-12-2008/images/CIOverview.jpg

CI Tools

- Fetch code
- Build
- Check
- Test
- Package
- Publish
- Notifications
- Report



xUnit Plugin

7 Added by Gregory Boissinot, last edited by Gregory Boissinot on Oct 08, 2015 (view change)

Plugin Information

Plugin ID	xunit	Changes	In Latest Release Since Latest Release
Latest Release Latest Release Date Required Core Dependencies	1.98 (archives) Oct 09, 2015 1.580.1 junit (version:1.6)	Source Code Issue Tracking Pull Requests Maintainer(s)	GitHub Open Issues Pull Requests Gregory Boissinot (id: gboissinot)
Usage	xunit - installations	Installations	2014-Oct 11692 2014-Nov 11557 2014-Dec 11631 2015-Jan 12106 2015-Feb 12262 2015-Mar 12891 2015-Apr 12694 2015-May 12716 2015-Jun 13143 2015-Jul 13470 2015-Aug 13192 2015-Sep 13563

This plugin makes it possible to publish the test results of an execution of a testing tool in Jenkins.

http://www.tutorialspoint.com/jenkins/

Summary

- Testing enables making changes confidently
- Testing early and often helps maintain code quality
- Writing tests early can help us understand requirements better
- Focusing on testability can help improve the software design
- Continuous integration and testing are at the core of mature software development practices

Activities this Week

- Read the required readings
- Participate in Practical 2
- Complete Quiz 7

University of
South Australia