
Lesson 15

Implementation and Test

*Thought leads to action; action leads to achievement; achievement leads to **fulfillment** .*

MUMSched Workflow

- The first week of class we focused on WHAT to build:
 1. Use case level -- Use Case Descriptions and Use Case Realizations
 2. System level – Analysis Mechanisms, Key Abstractions, and High level system architecture
- The second week of class we focused on HOW to build our project:
 1. System level – Design Mechanisms, Design Classes, Subsystems Context, and Proof-of-Concept
 2. Use case level – Use Case Design, Subsystem design, and class design
- The third and fourth week are focused on building our project, testing it, and demonstration of working code.
- What is the best approach to build your code as a team?


Best approach to team coding

- Use Scrum to keep track of what each person is doing and delivering for their use cases
- Each person in the group must have the whole project running on their development machine.
- Have one person be the owner of integration of code
- Use Github for source versioning. We have a Github and source versioning guide on sakai in: [Resources/github guide](#)
- Add code in small increments, build and test on your machine before pushing to the group Github project.
- Do daily integration of your group code and each person pull to their development machine and build on that code.
- Watch out for rebase! You don't want to lose your latest changes!

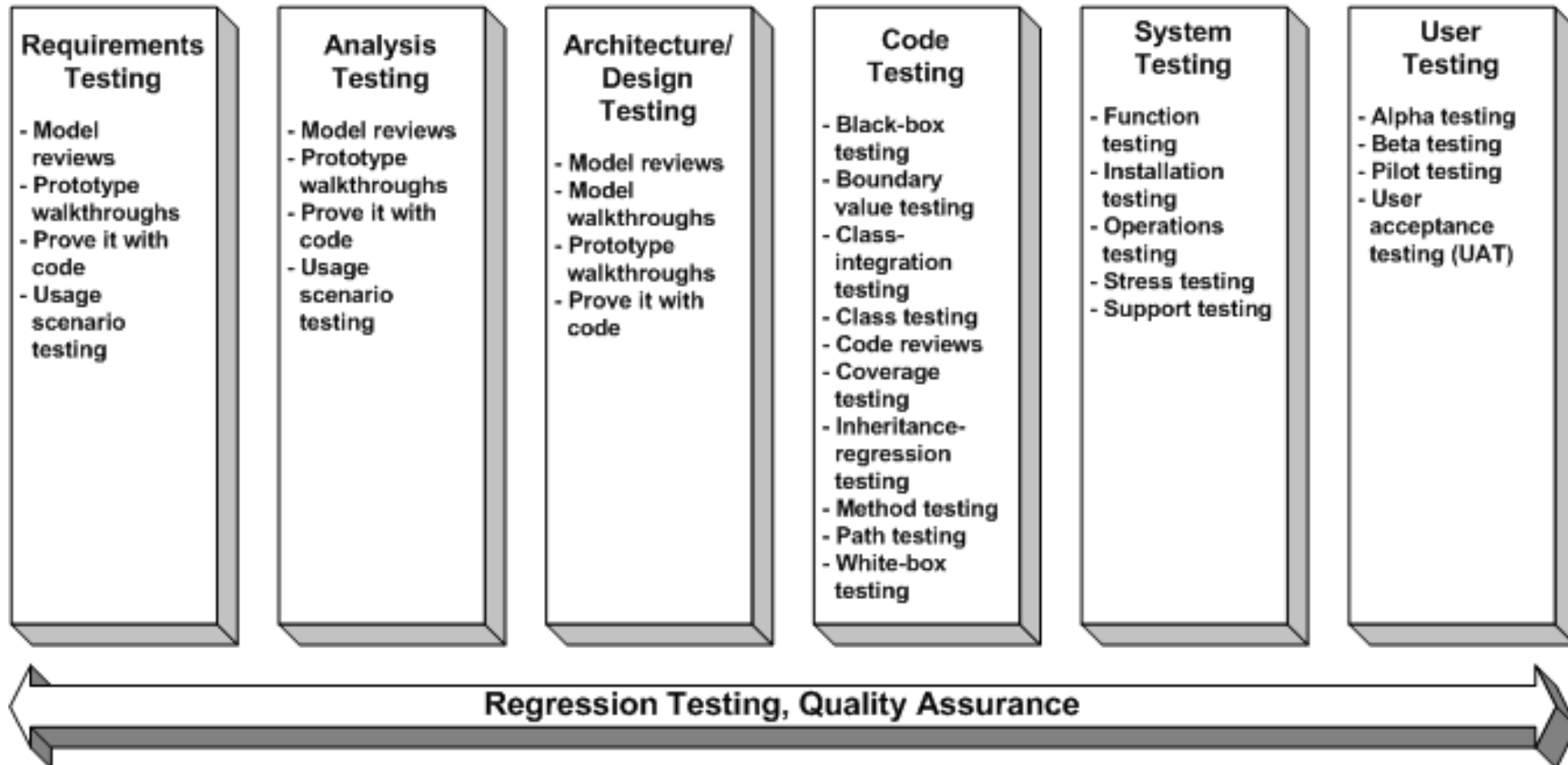
What is SCRUM?

- Scrum video: <http://www.youtube.com/user/axosoft>
- Simplest SCRUM approach for MUMSched development:
 1. Each person estimate in hours coding and unit test of their use cases
 2. Sum up the hours for your group for project start
 3. Each day, everyone estimates the remaining hours for their use cases
 4. Sum up the remaining hours for the group each day and update your progress
 5. Calculate your velocity (average number of hours of development the group completes each day)
 6. Predict and track your projected code completion
 7. Daily Scrum meetings to check on progress and obstacles

Testing Methodologies

- Like a development process, testing has its own methodology.
 - RUP provides many tools and approaches to testing
 - FLOOT (Full Lifecycle Object Oriented Testing) is part of the Agile Modeling paradigm (and due to Scott Ambler)
<http://www.ambyssoft.com/essays/floot.html>
 - Developer responsibility for testing varies with different companies, but more developer testing is the trend.
-  If possible - find out the testing culture at companies before interviewing. Knowing their testing approach can mean getting the job.

Full Lifecycle Object Oriented Testing



Copyright 2004 Scott W. Ambler

Testing Methodologies

Agile Modeling Testing Philosophies

1. The goal is to find defects
2. You can validate all artifacts
3. Test often and early
4. Testing builds confidence
5. Test to the amount of risk of the artifact
6. One test is worth a thousand opinions
7. Testing is *not* about fixing things
8. Perform regression testing wherever testing is being done

Testing Methodologies

The Test-Driven Development Paradigm

- The TDD paradigm says that the best testing strategy is to develop tests as part of the implementation process. In some companies the policy is that test code for a method or a class should be written before the actual code for the method or class is written.
- Robert Martin ("Professionalism and Test-Driven Development", IEEE Software, May/June 2007) describes a three-pronged TDD discipline:
 - a) You may not write production code unless you've first written a failing unit test
 - b) You may not write more of a unit test than is sufficient to fail
 - c) You may not write more production code than is sufficient to make the failing unit test pass
- Benefits of TDD
 - a) Safe environment for code cleanup.
 - b) Documentation.
 - c) Minimal debugging.

Testing - Requirements, Analysis, and Design

1. Model reviews.

An inspection, ranging from a formal technical review to an informal walkthrough, by others who are not directly involved with the development of the model.

2. Prototype walkthroughs

A process by which users work through a collection of use cases, using a prototype as if it were the real system

3. User Interface testing

As the user interface design unfolds, GUI shells can be implemented and tested for usability and for conformance with standards

4. Prove it with code

Settle questions about analysis and design models by writing code (a quick prototype) to demonstrate how things will work.

analysis testing => "can this be done?"

design testing => "what is the best way to do this?"

Testing - Code

1. *Test case*. A single test that needs to be performed. When fully documented, it includes information about how to set up an environment for performing the test, the actual steps of the test, and the expected results
2. *Test script*. Details the actual steps involved in executing a test case
3. *Test suite*. A collection of test scripts
4. *Test harness*. The context in which a suite of test scripts is run

Testing - Code

Standard Levels of Testing

- Unit Testing. Developer tests his/her own code in isolation, in the development environment. Tester is the developer
- Integration Testing. The team puts their pieces together to see if the code works together. Tester is still the development team.
- System Testing. The integrated release is placed in an environment matching the intended production environment, and tested. Tester is typically an outside team – such as a QA team.
- Customer/User Acceptance Testing (CAT/UAT). The release is tested in a realistic environment, this time by a representative team of users.

Sometimes user tests are divided into stages, depending on readiness of the project:

alpha testing – expect bugs but gives a rough idea

beta testing – most bugs are known, but seeking further refinement

pilot project – a test on a select group of users

Testing - Code

Best Practices For Unit Testing

1. Unit tests should be designed for regression testing
 - a) the same unit test should run (and pass) throughout the subsequent development of the system
 - b) no dependency on values (such as database values or customer-entered values) that could change over time.
 - c) If part of a test requires reading database values, those values should be inserted into a (non-production) table, then tested, then deleted from the table.
2. Unit tests should always test boundary cases for methods, use cases, or whatever else is being tested.
3. Unit tests --- even unit test methods --- must be independent of one another. The output of one unit test should not be used in another.

Testing - Code

Best Practices For Unit Testing (cont)

4. Each unit test should have a single purpose.
Rather than validate many features of a method or class in a single test, it is better to craft many tests that test each of the features separately. This makes it much easier to identify where there are problems in the code that is being tested.
5. Unit tests should be readable.
The code should make clear what the expected result is, what the actual result is, and perform an assertion that compares these.

Developer Testing Example

Mockito – <http://site.mockito.org/>

See BasicMockitoTesting.doc in sakai/resources/class demos

Optimized for Enterprise Architectures and dependency injection

- Replace **@Autowired** components in the class you want to test with mock objects.
- Unit test controllers by injecting mock services.
- Unit test the service layer by using mock DAOs
- Unit test the DAO layer by mocking DB APIs.

Developer Testing Example

- Mockito example code – adding dependency in Maven

```
<dependency>
```

```
  <groupId>org.mockito</groupId>
```

```
  <artifactId>mockito-all</artifactId>
```

```
  <version>1.9.5</version>
```

```
</dependency>
```

Mockito Example

```
import static org.mockito.Mockito.*;

public class MockCreationTest {
    private ProductDao productDao;
    private Product product;

    @Before
    public void setupMock() {
        product = mock(Product.class);
        productDao = mock(ProductDao.class);
    }

    @Test
    public void testMockCreation(){
        assertNotNull(product);
        assertNotNull(productDao);
    }
}
```


Mockito Example (cont.)

```
public class CustomerServiceTest {
```

```
    @Mock
```

```
    private CustomerDao daoMock;
```

```
    @InjectMocks
```

```
    private CustomerService service;
```

```
    @Before
```

```
    public void setUp() throws Exception {
```

```
        MockitoAnnotations.initMocks(this);
```

```
    }
```

```
    @Test public void test() {
```

```
        //assertion here
```

```
    }
```

```
//see https://javacodehouse.com/mockito-tutorial
```

Mockito Example (cont.)

Methods provided for Mock objects:

The when - then pattern examples:

`when(daoMock.save(customer)).thenReturn(true);` OR

- `thenThrow(exception)`
- `thenCallRealMethod()`
- `thenAnswer()`

VerificationMode examples:

- `times(int wantedNumberOfInvocations)`
- `atLeast(int wantedNumberOfInvocations)`
- `atMost(int wantedNumberOfInvocations)`
- `calls(int wantedNumberOfInvocations)`
- `only(int wantedNumberOfInvocations)`
- `atLeastOnce()`
- `never()`

Mockito Example (cont.)

Verify examples:

//verify that the save method has been invoked

```
verify(daoMock).save(any(Customer.class));
```

//the above is similar to : verify(daoMock, times(1)).save(any(Customer.class));

//verify that the exists method is invoked one time

```
verify(daoMock, times(1)).exists(anyString());
```

//verify that the delete method has never been invoked

```
verify(daoMock, never()).delete(any(Customer.class));
```

Testing - Code

Standard Testing Terms

- *Black box testing.*

Test cases are built around testing interfaces. Internals of the code are not viewed or accessed. This is a good way to test requirements ("see if the code does what it claims to do"). Example: Testing that each user defined input returns the expected value.

- *White box testing.*

A developer designs tests of his/her code based on a knowledge of the code. For example, where there are if..else statements, each branch is given one or more tests. This is done during Unit Testing primarily.

Testing - Code

Standard Testing Terms

- *Boundary value testing.*

Part of white box testing.

Test methods using values at the boundary of allowed values.

- If a method accepts an object argument, what happens when you pass in "null"?
- If computations are done assuming input lies in a certain range, what happens when you pass in value that are out of range, or on the boundary of the range?
- If a method depends on date values, what happens if you give unusual dates, like February 29?

Testing - Code

Standard Testing Terms

- *Coverage testing.*

Typically, test cases are designed to "test everything," but this means different things in different projects:

Could mean:

- Test all paths through the code (in effect, test every line of code)
- Test all significant flows in the use cases

For example, a department might declare testing "complete" when 95% of one of these coverage testing strategies is complete

Testing - Nonfunctional Requirements

Standard Testing Terms

1. Performance testing.

Testing to see that response time meets requirements

2. Stress testing.

Testing to see if the application can handle large volumes and concurrent access.

3. Installation testing.

Can the application be installed in the intended environment successfully? Verify config info, online documentation, supporting software is available, impact on the target environment.

4. Support testing.

Test that support personnel have been identified, adequate training processes worked out, realistic timeline for training, personnel are competent to provide support, etc.

5. Other nonfunctional requirements.

Any of these specified in nonfunctional requirements documentation should be subject to some form of testing.

Summary

- For both RUP and Agile - testing can be applied to every step of development from Analysis to final delivery.
- SCRUM is standard practice for most SW development in the U.S.
- Try SCRUM and see how you like it for tracking your group development

SCI Question

- What is a good SCI point that is analogous to and explains one of the benefits of using SCRUM.