

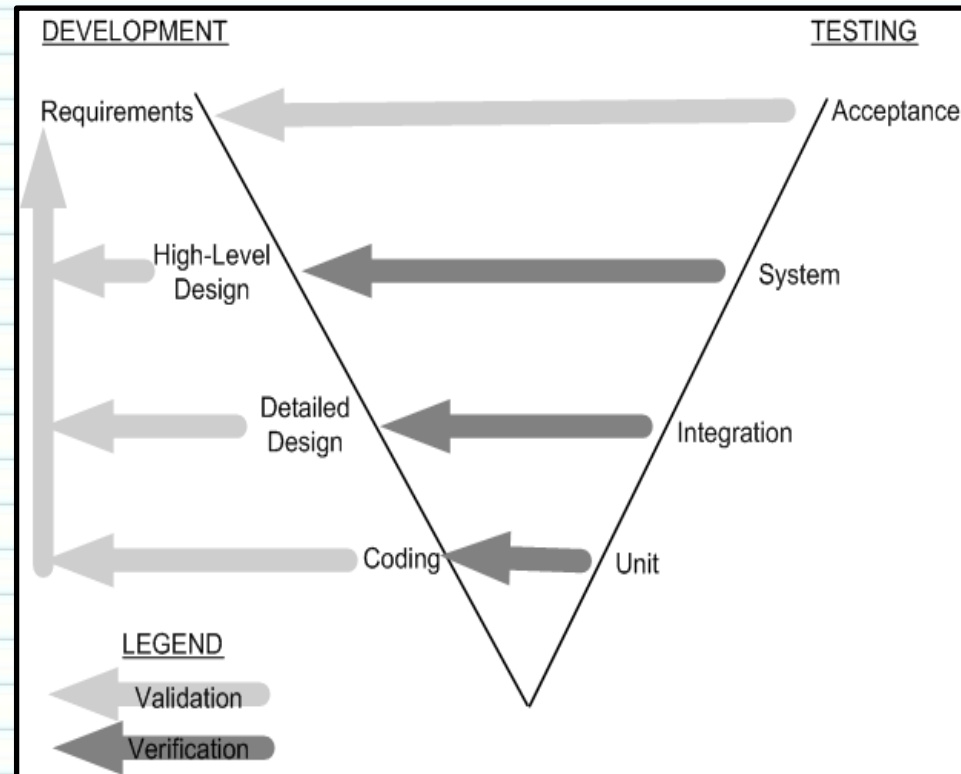
Chapter 20 – Software Testing (cont.)

Dr. Michael F. Siok, PE, ESEP
UT Arlington
Computer Science and Engineering

Throughout this course slides will be heavily re-used from:
Dr. David Kung, UTA CSE
Dr. Christoph Csallner, UTA CSE

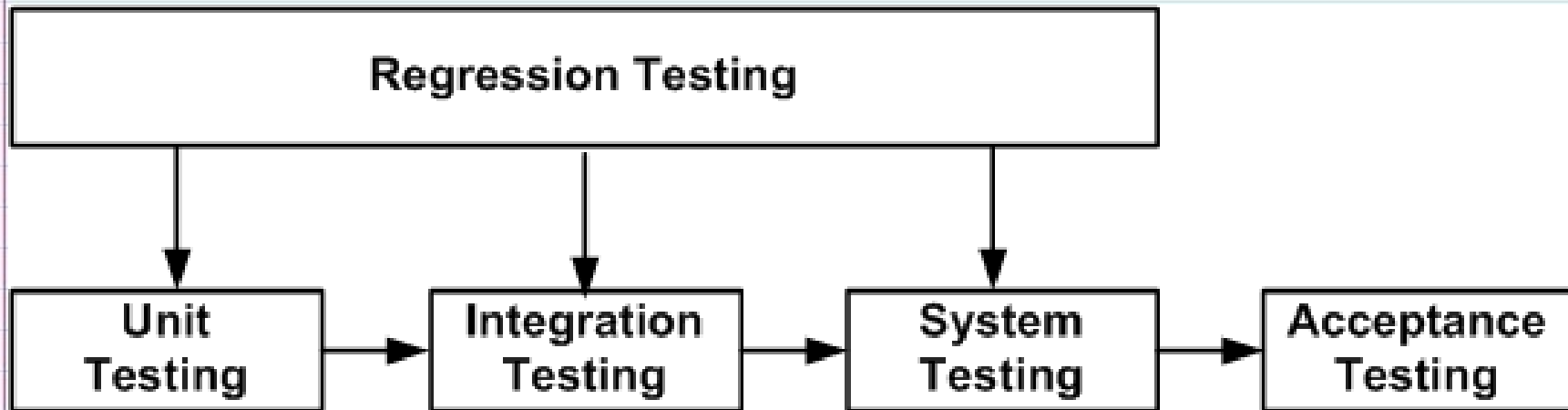
Testing Level

- Unit testing
 - Individual program units are tested in isolation. Unit testing is the lowest level of testing performed.
- Integration testing
 - Units are assembled to construct larger groups and tested
- System testing
 - Includes wide spectrum of testing such as functionality, and stress
- Acceptance testing
 - Customer's expectations from the system



Development and Testing Phases in the V model

Re-Testing Level

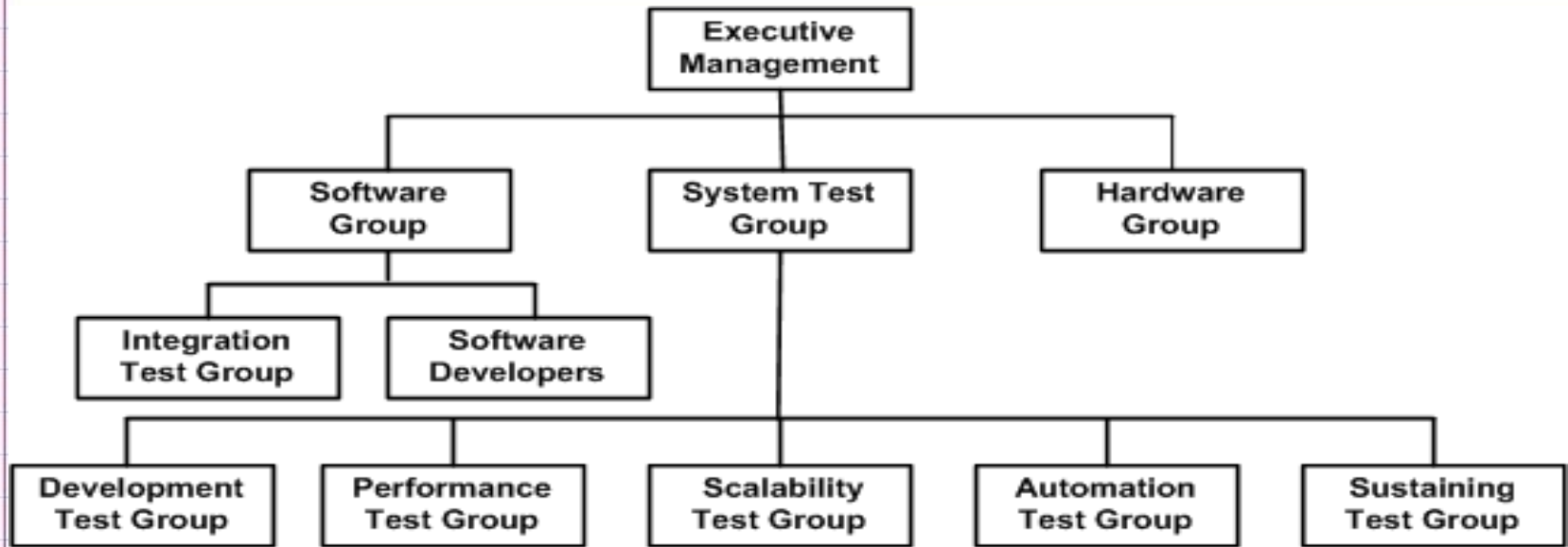


- During product evolution two kinds of testing occur:
 - Testing of the new changes being made
 - Testing to ensure that the new changes did not impact some existing functionality
- Regression testing is performed at software testing levels - many times in parallel
- Regression testing is performed to ensure that the software function wasn't impacted because of:
 - Unforeseen linkages/impacts between software components
 - Subtle timing or tasking issues that cannot be verified other than test
 - Configuration management problems

Monitoring and Measuring Test Execution

- Each level of test has specific measures used to monitor and measure test executions and progress
- Metrics for monitoring test execution
 - Test cases reviewed
 - Test cases developed, debugged, and executing
 - Test cases captured in the developmental CM system
- Metrics for monitoring defects
 - Test cases passing/failing
 - Test cases blocked (awaiting a software fix)
 - Software fixes targeted for a specific release
- Test case effectiveness metrics
 - Measure the “defect revealing ability” of the test suite
 - Use the metric to improve the test design process
- Test-effort effectiveness metrics
 - Number of defects found by the customers that were not found by the test engineers

Test Team Organization and Management



A Notional Organization Structure of Test Groups

- Unit and Integration testing is many times performed by the developers.
- System level test and Acceptance testing is typically performed by independent test organizations
 - “Independent” means not the software group
- Technology obsolescence in software development and test is very high
 - New training must be continually provided to maintain an efficient work force
- To retain test engineers, management must recognize the importance of testing efforts at par with development efforts

Test Documentation

- Testing is a complicated task
 - Mature organizations have a test process, test standard, and test plan
 - These address all test activities: Unit, Integration, and System testing and include Regression test
- Test Process
 - Test methodologies (e.g., specification based, code based, etc.)
 - Test organization (how tests are organized, where kept, how changed, etc.)
 - Who approves various test plans, documents, and artifacts
 - Organizational roles and responsibilities
 - Test Tools and automation
 - Training of personnel (new and existing)
- Test Standard
 - Describes when testing is complete and the various activities used to support test completion

Test Plan

- The purpose is to get ready and organized for test execution
- A test plan provides a:
 - Framework
 - A set of ideas, facts or circumstances within which the tests will be conducted
 - Scope
 - The domain or extent of the test activities
 - Details of resource needed
 - Effort required
 - Schedule of activities
 - Test objectives
 - Identified from different sources

Test Procedures

- Documented set of test cases needed to complete a test per the test plan
- Typically,
 - Each test case is designed as a combination of modular test components called test steps
 - Test steps are combined together to create more complex tests
- Test procedures may be manual procedures, automated procedures, or a combination of manual and automated procedures
 - In accordance with the Test Plan
- Test procedures provide test steps, pass/failure criteria, and a record of the performance of tests

Test Results

- These are the results of the test execution across all test activities: Unit, Integration, and System testing including regression tests
 - May be a single report or provided in multiple reports
- Test results are typically tied to a specific product configuration
 - Such as a specific test or delivery milestone
- Test results are used as evidence to demonstrate the software function(s) correctly works as specified by the requirements
- Test results may indicate or uncover one or more deficiencies as described earlier in the course

Unit - What is it?

- This is the age old question
- The industry has many different definitions for what constitutes a unit - the differing terminology is because the term unit has evolved over time as code went from
 - Large files of assembly code
 - Smaller files of high-order programming language
 - More organized methods of code organization - such as structured coding
 - Object-oriented software
- A unit can be a method, a class, or a file
 - it is supposed to represent the smallest "unit" or piece of code

Unit - What is it? (cont.)

- Rather than argue about what a unit is, we can understand that code has to be tested in this fashion (multiples of these can be grouped into a single test activity)
 - Methods
 - Within a single class
 - Across Classes
 - As a whole
- When we test from smallest to largest we have the most efficient test approach because the test being developed and conducted has the most narrow scope - at the unit level.
- It is the most efficient because
 - It has the least interference from outside forces (other changing or not yet developed units)
 - It is the easiest to develop test cases for - the number of inputs and possible combinations is drastically reduced

Object Oriented Code Truths

- About 85 percent of the object-oriented code base will have a cyclomatic complexity of 3 or less
 - Much of the code is getters/setters
 - The highest complexity code will be the controllers (the class causing things to happen)
 - Typically, errors are related to the cyclomatic complexity so concentrating on testing the controllers is one method to reduce errors
- The object-oriented approach of developing software that has high-cohesion, low-coupling, information hiding, and encapsulation meaning that it is most suited to unit level testing
- It also means that much of the complexity from traditional structured approaches is now moved to the interface between classes
 - Integration testing between classes is essential

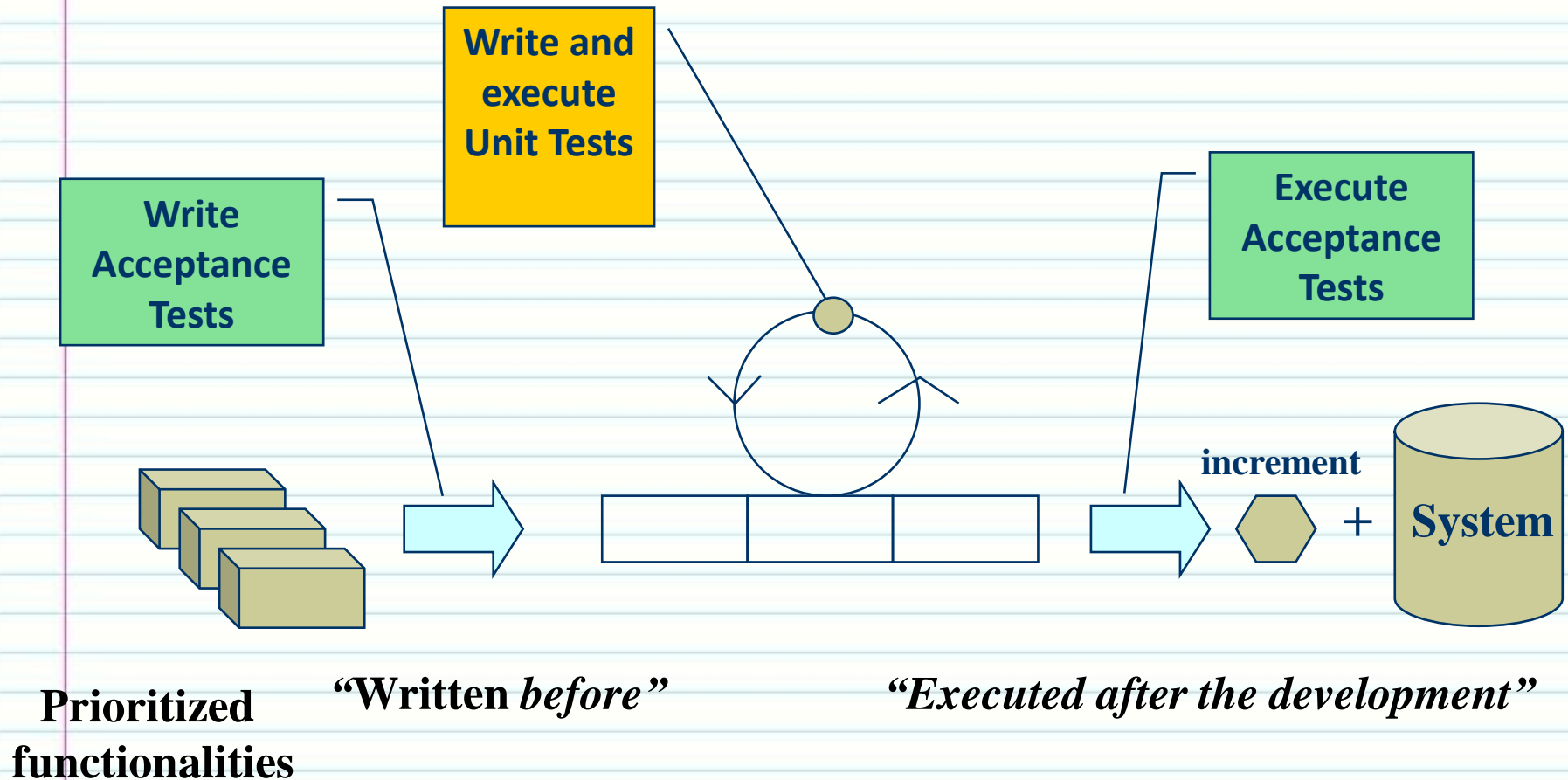
Unit Testing

- Involves testing a single isolated unit
- Note that unit testing allows us to isolate the errors to a single unit
 - We know that if we find an error during unit testing it is in the unit we are testing
- Units in a program are not isolated, they interact with each other. Possible interactions:
 - Calling procedures in other units
 - Receiving procedure calls from other units
 - Sharing variables
- For unit testing, we need to isolate the unit we want to test; we do this using:
 - Drivers
 - stubs
 - mock objects

Advantages of Unit Testing

- Advantages of Unit Testing:
 - Faster Debugging
 - Unit tests smaller amounts of code, easily isolating points of error and narrowing the scope for errors.
 - Faster Overall Development
 - Less time debugging with unit tests
 - Encourages aggressive refactoring resulting in better design and easier maintenance
 - Adding enhancements can be performed with confidence that existing behavior of the system continues to operate in the same manner
 - *Modify current unit test in add tests for enhancements*
 - Better Overall Design
 - As mentioned before, unit testing encouraging more refactoring.
 - Unit tests make developers focus more on the contracts of a class and those classes that might use it
 - Excellent Regression Tool
 - Major refactoring and restructuring of the code can be performed
 - Reduce Future Cost
 - Unit testing is an investment in the project. It takes time to ramp up, but in the long-term provides a useful basis for the project

Iterative Software Development

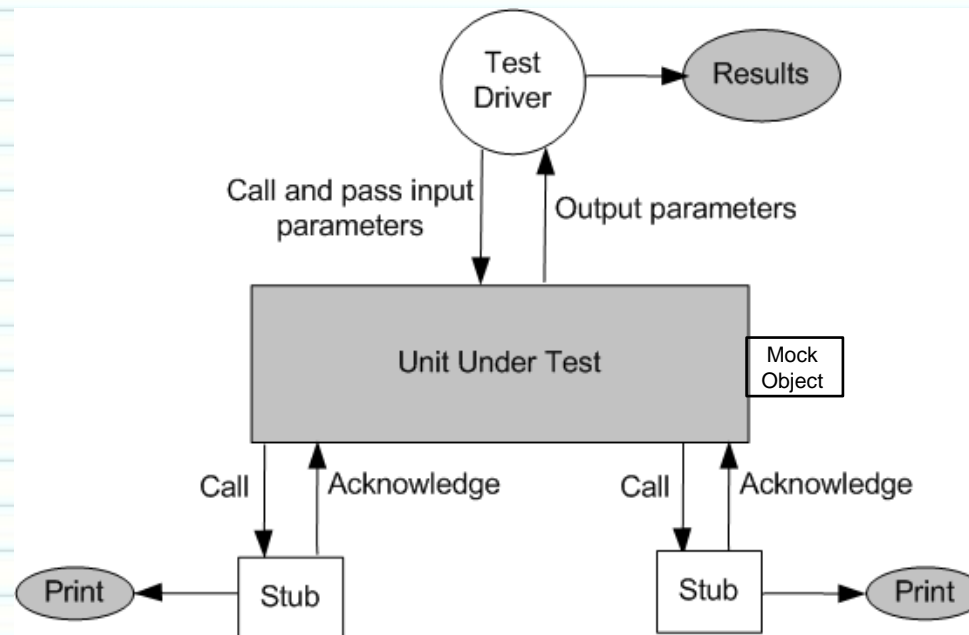


Unit Testing is the key to most testing efforts

Unit Testing Framework

- The environment of a unit is emulated and tested in isolation
- The caller unit is known as *test driver*
 - A *test driver* is a program that invokes the Unit Under Test (UUT)
 - The *Test driver* provides input data to the UUT and reports the test result
- The emulation of the units called by the UUT are called *stubs*
 - It is a dummy program
- A *mock object* is a test program that substitutes for the actual object method
 - The mock simulates the interface and behavior of the real method
- The *test driver*, the *stubs*, and *mock objects* are together called *scaffolding*
- The low-level design documents provide guidance for selection of input test data

Unit test
environment



Unit Testing Framework (cont.)

- A driver is the thing that calls the UUT
- Typically the driver is a stand-alone class/file that is not part of the delivered software
 - It is stand-alone to make it easily separable from the deliverable software
- When the UUT calls another method outside its class it is typically stubbed out
 - This means that the test case supplies the correct return value from that method (instead of actually calling it and getting the return)
 - This helps to isolate one unit from another so that they can independently change
 - As the units mature they may either be un-stubbed or stay stubbed

Why use a Testing Framework?

- Each class must be tested when it is developed
- Each class needs a regression test
- Regression tests need to have standard interfaces
- Thus, we can build the regression test when building the class and reuse it again later when the unit needs to be retested or modified and retested
 - Provides for a better, more stable product for less work

Drivers, Stubs, and Mock Objects for Unit Testing

- Driver
 - A method (or main) that passes test case data to the component being tested, verifies the returned results, and summarizes pass/fail outcomes
 - JUnit is such a driver, some drivers are developed by hand without tools
- Test double
 - Substitution for production code during test - typically either a mock or stub (fakes and spies are also used)
 - Both stubs and mocks are used to force the method being called to return the needed value to the UUT
- Stubs
 - Hand-written classes that mimic the methods behavior, but do that with significant simplification.
- Mocks
 - Dynamic "wrapper" used to simulate a method call/return using a tool
 - They use the method's exact interface and are replaced by production code for later integration tests
- Drivers, stubs, and mock objects represent overhead test cost scaffolding)
 - This in addition to production code

Drivers, Stubs, and Mock Objects for Unit Testing (cont.)

- Fake objects actually have working implementations, but usually take some shortcut which makes them not suitable for production
- Stubs provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.
 - Stubs may also record information about calls, such as an email gateway stub that remembers the messages it 'sent', or maybe only how many messages it 'sent'.
- Mocks are objects pre-programmed with expectations which form a specification of the calls they are expected to receive (more sophisticated state behavior interaction).
- Of these, only mocks objects simulate the behavior of the actual object.

Advantage of Using a Test Automation Tool

- Driver
 - The driver is limited to a tool such as JUnit.
 - Other methods are even more automated providing sophisticated test case design and test reporting functions
 - The driver code required is anywhere from somewhat less to considerable less
- Stubs
 - Some tools provide automated methods to stub out method calls and returns so that this is totally invisible to the test case
- Mock Objects
 - Some tools provide sophisticated means to develop mock objects and return appropriate test inputs and outputs
- Drivers, stubs, and mock objects both represent overhead
 - The overhead is limited because the tool is generating this behind the scenes

Advantage of Using a Test Automation Tool

- Expense
 - Some of these tools have a significant upfront and yearly cost
- Learning curve
 - Many of these tools can take months to become proficient to use the basic features
 - Some of the tools are quite feature rich
- Driver
 - The driver code can inject a considerable amount of hidden instrumentation in the code
 - This can make the code performance under test very sluggish - making performance assessments difficult
- The tradeoff between a test tool and use of driver code is one that needs to be made from a cost, schedule, and quality standpoint

Regression Test

- ISTQB defines Regression Testing as:
 - *Testing of a previously tested program following modification to ensure that defects have not been introduced or uncovered in unchanged areas of the software as a result of the changes made. Regression testing is typically performed when the software or its environment is changed.*
- Note that the industry and academic use of this term is pretty loose
 - The common fault is including testing of new changes - this does not expose a regression error
 - During product evolution, two kinds of testing occurs
 - Testing of the new changes being made
 - Testing to ensure that the change did not impact some existing function

Regression Test (Cont.)

- Regression testing is performed at software testing levels - many times in parallel
- Regression testing is performed to ensure that the software function was not impacted because of:
 - Unforeseen linkages/impacts between software components
 - Subtle timing or tasking issues that cannot be verified other than test
 - Configuration management problems
- Some approaches (Agile) perform daily unit testing
 - There are pros and cons associated with this approach
 - I prefer to test a unit when the code for that unit is changed only (this includes changes to the interface or inheritance structure).

Integration Testing

- There are several methods of performing integration testing
 - Testing across classes
- Horizontal Integration Testing
 - “Big bang” integration
 - Bottom-up integration
 - Top-down integration
 - Middle-out integration
- Vertical Integration Testing
- As mentioned previously we need to test:
 - Methods
 - *Within a single class*
 - Across Classes
 - *As a whole*
- This is what integration seeks to achieve

Code for JUnit Example

- The following is the code for the MealNames enumeration in Java
- This code is in a class MealNames.java as follows

MealNames.java

```
public enum MealNames {breakfast, lunch, supper}
```

The code will be attached to slide M12 in blackboard

Code for JUnit Example (cont.)

The following code is in Meals.java

```
public class Meals {
```

```
MealNames meal = MealNames.breakfast;
```

```
public void nextMeal() {
```

```
switch (meal) {
```

```
case breakfast:
```

```
meal = MealNames.lunch;
```

```
break;
```

```
case lunch:
```

```
meal = MealNames.supper;
```

```
break;
```

```
case supper:
```

```
meal = MealNames.breakfast;
```

```
break;
```

```
default:
```

```
meal = MealNames.breakfast;
```

```
}}
```

```
public void skipmultiplemeals(int numberOfMeals) {
```

```
for (int i = 0; i < numberOfMeals; i++)
```

```
nextMeal();}}
```

The JUnit Created Test Code

```
import static org.junit.Assert.*;
```

```
import org.junit.After;  
import org.junit.AfterClass;  
import org.junit.Before;  
import org.junit.BeforeClass;  
import org.junit.Test;
```

```
public class MealsTest {
```

```
    @BeforeClass  
    public static void setUpBeforeClass() throws Exception {  
    }
```

```
    @AfterClass  
    public static void tearDownAfterClass() throws Exception {  
    }
```

```
    @Before  
    public void setUp() throws Exception {  
    }
```

```
    @After  
    public void tearDown() throws Exception {  
    }
```

```
    @Test  
    public void testNextMeal() {  
        fail("Not yet implemented");  
    }
```

```
    @Test  
    public void testSkipmultiplemeals() {  
        fail("Not yet implemented");  
    }
```

```
}
```

This code is also attached to M12 in Canvas



JUnit test generated code.txt

Replace these two test methods with the code on the next slide

The Correct Test Code for Our Example

@Test

public void testMeals() {

```
    Meals mymeal = new Meals();  
    assertEquals(MealNames.breakfast, mymeal.meal);  
    mymeal.nextMeal();  
    assertEquals(MealNames.lunch, mymeal.meal);
```

This code is also attached to M12 in Canvas



Our modified JUnit test code.txt

```
// Remove the comment delimiters below to raise a wrong assertion to show the exception raised  
// assertEquals(MealNames.breakfast, mymeal.meal);  
// mymeal.nextMeal();  
// assertEquals(MealNames.supper, mymeal.meal);  
  
// Remove the comment delimiters below to raise a wrong assertion to show the exception raised  
// assertEquals(MealNames.lunch, mymeal.meal);  
// mymeal.nextMeal();  
// assertEquals(MealNames.breakfast, mymeal.meal);  
  
// Remove the comment delimiters below to raise a wrong assertion to show the exception raised  
// assertEquals(MealNames.lunch, mymeal.meal);  
}
```

@Test

public void testSkipmultiplemeals () {

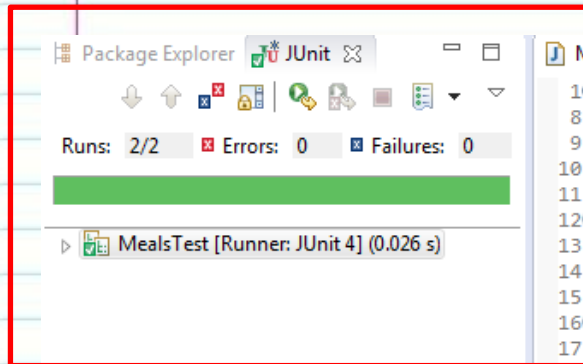
```
    Meals mymeal=new Meals();  
    mymeal.skipmultiplemeals(4);  
    assertEquals(MealNames.lunch, mymeal.meal);
```

```
// Remove the comment delimiters below to raise a wrong assertion to show the exception raised  
// assertEquals(MealNames.breakfast, mymeal.meal);
```

What is the Test Code Doing?

- JUnit has several methods to assist with unit testing.
- Our test code is using the assertEquals method. This works by having JUnit automatically check that the two values being compared are equal.
- If they are equal the test passes. If not, an exception is raised.
- Here is an example of the test code that we're using to perform these checks.
`assertEquals(MealNames.lunch, mymeal.meal);`
- It's checking to see if the lunch attribute in the mymeal object is equal to the enumeration value of lunch.

A Successful JUnit Test Run



Package Explorer JUnit

Runs: 2/2 Errors: 0 Failures: 0

MealsTest [Runner: JUnit 4] (0.026 s)

```
MealsTest.java
1 import static org.junit.Assert.*;
2
3
4
5
6
7
8
9
10 public class MealsTest {
11
12     @BeforeClass
13     public static void setUpBeforeClass() throws Exception {
14     }
15
16     @AfterClass
17     public static void tearDownAfterClass() throws Exception {
18     }
19
20     @Before
21     public void setUp() throws Exception {
22     }
23
24     @After
25     public void tearDown() throws Exception {
26     }
27
28     @Test
29     public void testNextMeal() {
30
31         Meals mymeal = new Meals();
32         assertEquals(MealNames.breakfast, mymeal.meal);
33         mymeal.nextMeal();
34         assertEquals(MealNames.lunch, mymeal.meal);
35
36         // Remove the comment delimiters below to raise a wrong assertion to show the exception raised
37         // assertEquals(MealNames.breakfast, mymeal.meal);
38         mymeal.nextMeal();
39         assertEquals(MealNames.supper, mymeal.meal);
40
41         // Remove the comment delimiters below to raise a wrong assertion to show the exception raised
42         // assertEquals(MealNames.lunch, mymeal.meal);
43         mymeal.nextMeal();
44     }
45 }
```

Failure Trace

An Unsuccessful JUnit Test Run

- I have added some commented lines in these tests so you can remove them and inspect what happens when an assertion is not satisfied.
- The following slide shows the result of removing one of these comment delimiters.
- The test fails and the line number where the exception is raised is identified. The test results from the comparison are also identified - the expected and actual results.

An Unsuccessful JUnit Test Run (cont.)

Package Explorer JUnit

Finished after 0.063 seconds

Runs: 2/2 Errors: 0 Failures: 1

MealsTest [Runner: JUnit 4] (0.034 s)

- testSkipmultiplemeals (0.017 s)
- testNextMeal (0.017 s)

1 Failure

Expected vs actual output

Failure Trace

java.lang.AssertionError: expected:<breakfast> but was:<lunch>
at MealsTest.testNextMeal(MealsTest.java:37)

Line number of exception

```
MealsTest.java
1 import static org.junit.Assert.*;
2
3
4
5
6
7
8
9
10 public class MealsTest {
11
12     @BeforeClass
13     public static void setUpBeforeClass() throws Exception {
14     }
15
16     @AfterClass
17     public static void tearDownAfterClass() throws Exception {
18     }
19
20     @Before
21     public void setUp() throws Exception {
22     }
23
24     @After
25     public void tearDown() throws Exception {
26     }
27
28     @Test
29     public void testNextMeal() {
30
31         Meals mymeal = new Meals();
32         assertEquals(MealNames.breakfast, mymeal.meal);
33         mymeal.nextMeal();
34         assertEquals(MealNames.Lunch, mymeal.meal);
35
36         // Remove the comment delimiters below to raise a wrong assertion to show the exception
37         assertEquals(MealNames.breakfast, mymeal.meal);
38         mymeal.nextMeal();
39         assertEquals(MealNames.supper, mymeal.meal);
40     }
41 }
```

JaCoCo

- Java code coverage (JaCoCo) is a free ECL Emma based tool for Eclipse or NetBeans
 - It provides direct code coverage analysis while in the workbench:
 - **Fast develop/test cycle:** Launches from within the workbench like JUnit -- test runs can directly be analyzed for code coverage.
 - **Rich coverage analysis:** Coverage results are immediately summarized and highlighted in the Java source code editors.
 - **Non-invasive:** EclEmma does not require modifying your projects or performing any other setup.
- The tool provides support of the individual developer in an interactive way.
- Eclipse installation: <http://www.eclemma.org/installation.html>
- NetBeans installation: <http://plugins.netbeans.org/plugin/48570/tikione-jacocoverage>

JaCoCo Source Code Annotations

- Branch coverage of the active coverage session is directly displayed in the Java source editors. This works for Java source files contained in the project as well as source code attached to binary libraries.
- Source lines containing executable code get the following color code:
 - green for fully covered lines,
 - yellow for partly covered lines (branches missed)
 - red for lines that have not been executed at all
- In addition colored diamonds are shown at the left for lines containing decision branches. The colors for the diamonds have a similar approach:
 - green for fully covered branches,
 - yellow for partly covered branches and
 - red when no branches in the particular line have been executed.
- The default colors can be modified in the Preferences dialog - do not do this please
- The source annotations automatically disappear when you start editing a source file or delete the coverage session.

JaCoCo Coverage Example

```
MealNames.java  Meals.java  MealsTest.java

1 public class Meals {
2
3     MealNames meal = MealNames.breakfast;
4
5     public void nextMeal() {
6         switch (meal) {
7             case breakfast:
8                 meal = MealNames.lunch;
9                 break;
10            case lunch:
11                meal = MealNames.supper;
12                break;
13            case supper:
14                meal = MealNames.breakfast;
15                break;
16            default:
17                meal = MealNames.breakfast;
18        }
19    }
20
21    public void skipmultiplemeals(int numberOfMeals) {
22        for (int i = 0; i < numberOfMeals; i++)
23            nextMeal();
24    }
25 }
26
27
28
29
```

This is the result of running the MealsTest.java - the JUnit test for Meals.java


It is showing partial coverage of the switch, and

Non-coverage of the default (as we would expect)

Hovering over the diamond will show the coverage metrics

Problems @ Javadoc Declaration Console Coverage

MealsTest (Apr 10, 2015 9:49:33 AM)

Element	Coverage	Covered Branches	Missed Branches	Total Branches
▶ M12	 83.3 %	5	1	6

JUnit and JaCoCo Work Together

- From the Package Explorer view select the xxxTest.java (JUnit test case)
- Launch the xxxTest.java coverage session using the Run JaCoCo on the Run toolbar:



- Click on the file you wish to see the coverage analysis
- As expected you will get the coverage assessments with the colors and colored diamonds indicating coverage achieved
- If you select the xxxTest.java JUnit test case you will also see the coverage of that as well
- For your homework and project all I need is the coverage report of the source, not the JUnit test case

```
MealsTest.java  Meals.java x
1  public class Meals {
2
3      MealNames meal = MealNames.breakfast;
4
5      public void nextMeal() {
6
7          switch (meal) {
8              case breakfast:
9                  meal = MealNames.lunch;
10                 break;
11              case lunch:
12                  meal = MealNames.supper;
13                 break;
14              case supper:
15                  meal = MealNames.breakfast;
16                 break;
17              default:
18                  meal = MealNames.breakfast;
19          }
20      }
21
22      public void skipmultiplemeals(int numberOfMeals) {
23
24          for (int i = 0; i < numberOfMeals; i++)
25              nextMeal();
26      }
27  }
28
29  }
```


Testing Web Applications

- Web applications exploit
 - Navigation and interaction facilities of web pages
 - Requesting information from the user
 - Providing information to the user
- Navigation and interaction patterns are important aspects of Web applications testing.

Web Page

- A Web page contains
 - the information to be displayed to the user
 - the navigation links to other pages
 - organization and interaction facilities (e.g., frames and forms).
- Navigation from page to page is modeled by *links*.
- Contents of static pages are fixed.
- Contents of dynamic pages are generated according to the input provided by the user (represented by the use attribute).

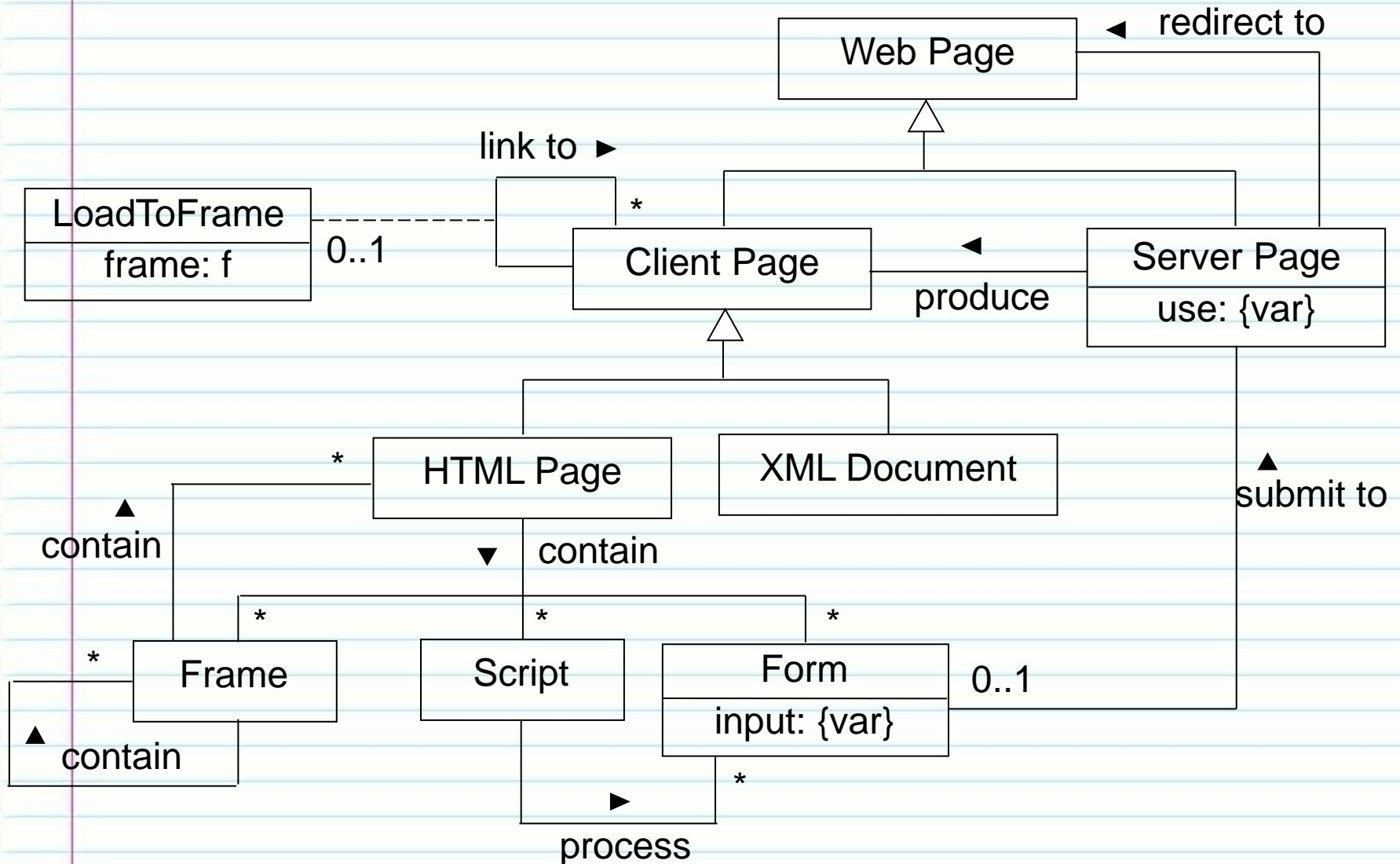
Relationships

- A page can contain any number of forms.
- A page may be split into a number of frames.
- A page may include other pages.
- Forms have input variables modeled by the input attribute of form.
- Forms are submitted to dynamic pages.

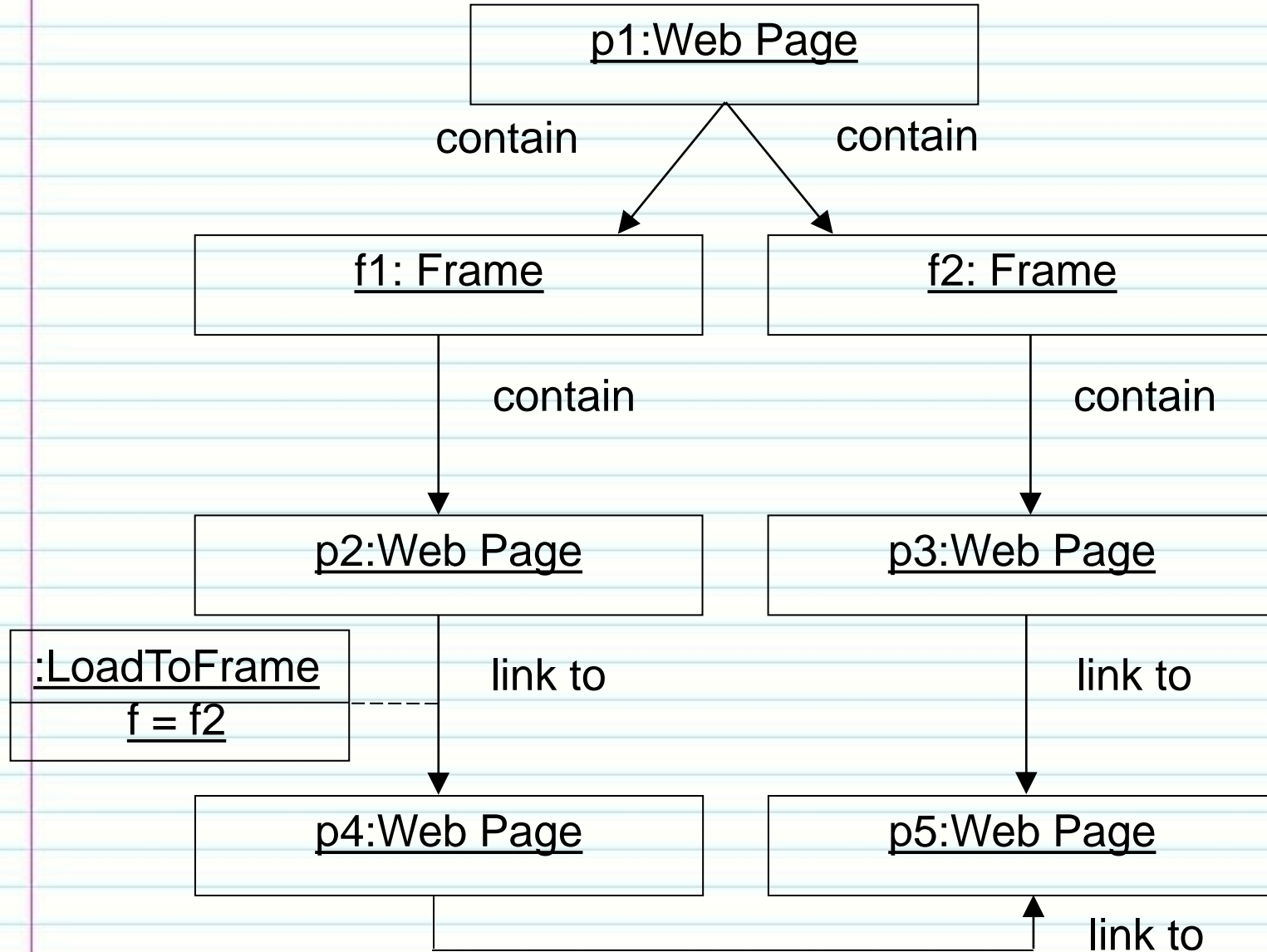
Frames

- A frame is a rectangular area in the page where navigation can take place independently.
- Frames on a page can interact with each other.
- A link in a page of a frame can force the loading of a page into another frame.

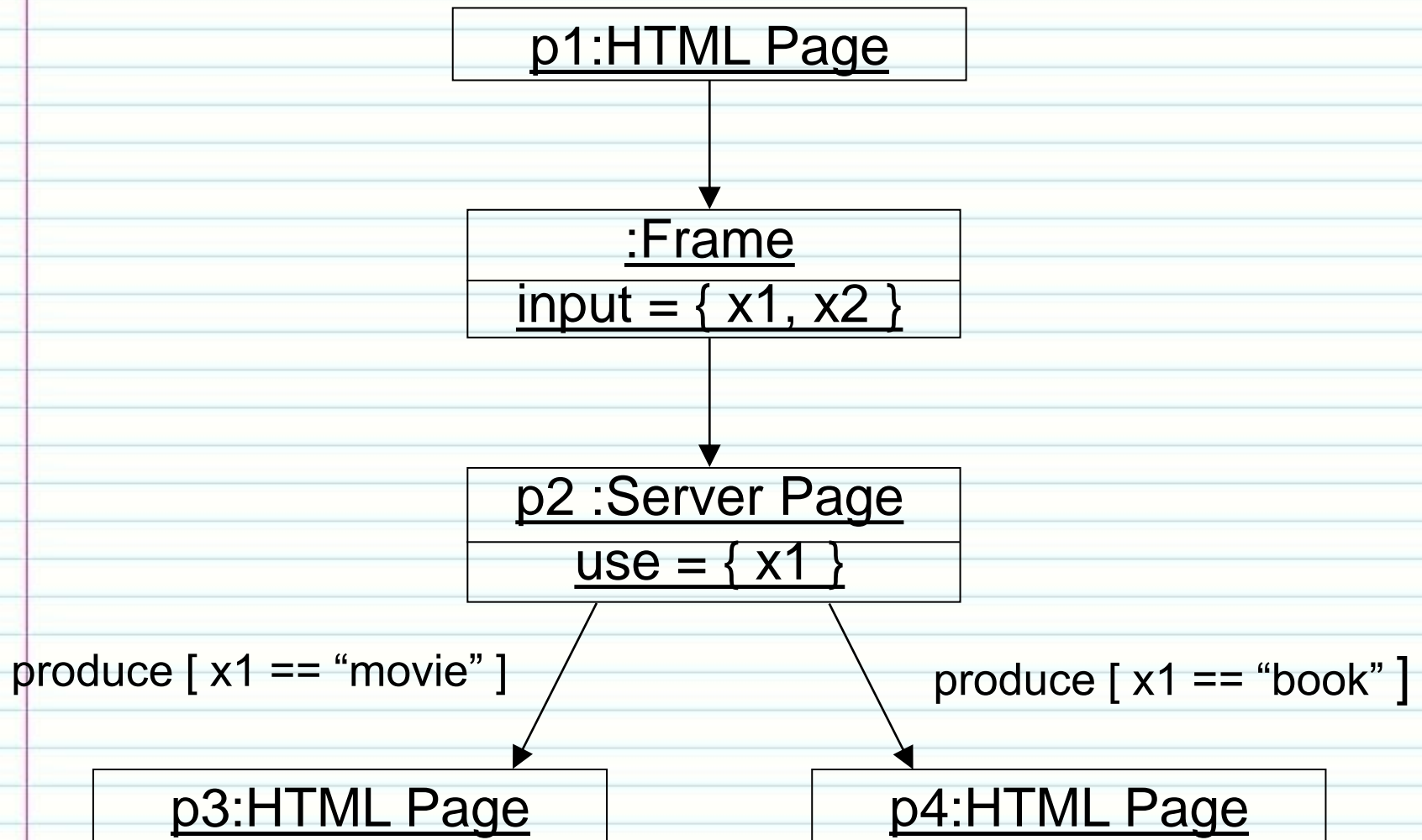
A Meta-Model for a Web Application



A Dummy Example



Another Dummy Example



Usefulness of a Meta Model

- Defining test criteria
- Guiding the test process
- Detecting
 - Non-compliance with user requirements
 - abnormal situations (e.g., ghost pages, unreachable pages)
- Techniques used to accomplish above
 - static analysis --- no code execution is required
 - dynamic validation --- exercise the code/system and compare the outcomes with expected results

Static Analysis

- Unreachable pages are pages that are available at the server site but cannot be reached by any path from an initial page.
- Unreachable pages can be detected by using a graph traversal algorithm.
- Ghost pages are pages that are referenced by a link but do not exist. Check should include external pages referenced by a page of the web application.
- Ghost pages can be checked by visiting the URL.

Static Analysis

- In web applications, pages should be loaded into appropriate frames.
- The set of reachable frames for a given page
 - frames that have the page as the initial page
 - frames that is loaded with the page due to a link
- Reachable frames show the assignment of pages to frames.
- Can be used to detect inappropriate assignment of pages to frames.

Static Analysis: Data Flow Testing

- Select test paths of a program according to the locations of definitions and uses of variables in the program, then test these define-use chains.
- Steps
 - Determine define-use chains for each variables
 - Determine testing criteria
 - Design test cases to satisfy the test criteria

Data Flow Testing for Web Applications

- The input attributes of a Form define the input variables.
- The definitions propagate along the edges of the site graph.
- If the definition reaches a dynamic page which uses one of the variables, a define-use dependency is identified.
- Traditional data flow analysis techniques are then used to analyze the define-use path.

Static Analysis

- Shortest path
 - the minimal number of pages that must be visited before reaching a target page.
 - this information is an indication of potential problems for searching a page if the shortest path is long.
- Comparing the above metrics for different versions of a web site may be used to assess the quality improvement (or not).

Test Cases/Dynamic Analysis

- A test case is defined as:
 - A sequence of URLs to be visited, along with
 - The input variable values that must be supplied.
- Execution of a test case is:
 - Requesting the web server for the URLs along with the input values, and
 - Storing the output pages
- Unlike traditional program testing, URLs can be directly visited without having to satisfy the path conditions.

Coverage Criteria

- Page coverage: every page in the site is visited at least once in some test case.
- Link coverage: every hyperlink from every page in the site is traversed at least once.
- Define-use coverage: at least one navigation path from every definition of a variable to every use of it, forming a data dependence, is exercised.
- All-uses coverage: all navigation paths from every definition of a variable to every use of it, forming a data dependence, is exercised.
- All-paths coverage: every path in the site is traversed in some test case at least once.

Removing Static Pages

- Static pages without a form displays only static information.
- Such static pages can be removed from the site graph during dynamic validation.
- The predecessors of a static page is directly linked to all of its successors.
- In the resulting graph, an entry node and an exit node are added.

Entry Node and Exit Node

- An entry node is added and is connected to every node that does not have a predecessor.
- An exit node is added and is reachable from all dynamic pages with non-empty user input variables, because:
 - There is no intrinsic notion of termination of a computation in web browsing
 - Whenever information is displayed to the user, the computation is completed

When do we Stop Testing?

- Testing costs money
 - *“If the design and code job is done correctly, we don’t need to test.”*
-- anonymous
 - **The above quote does not ever work in practice!**
 - People make mistakes
 - *At every step and sub-step of the development process*
 - Tools can insert faults into the work products
 - Test tools (for verification and/or validation) can have faults in them too
 - Test activities must be planned, reviewed, executed, and reviewed
 - Maintaining a Test organization is like maintaining any other organization in engineering
 - *Requires budget, investment, costing, and accountability*
 - The more we test, the more it costs
 - Alternatively, the more we test, the better the product likely becomes
- So, when do we stop testing?

So When do we Stop Testing?

- When we run out of money . . .
- When we run out of schedule . . .
- ???
- How about we stop testing . . .
 - When test coverage requirements are met or exceeded?
 - When test objectives have been met?
 - When the product meets or exceeds its quality objectives?
 - Or . . .

All of these could be correct

Stop testing when all severity 1 and 2 defects have been fixed and no more have been discovered since the last full test

Stop testing when the calculated defect density is below xx defects/KSLOC

Stop testing when no defects have been found in the last 150 hours of continuous testing to our operational profile

$$\mu(t) = \alpha[1 - (1 + \beta t)e^{-\beta t}]$$

Stop testing when the Mean Time Between Failures (MTBF) has reached 1,000 hours

$$\lambda(t) = \alpha\beta^2 t e^{-\beta t}$$

Stop testing when the estimated failure intensity reaches 0.3795

.

Summary

- Software testing is a dynamic validation technique
- Software test techniques include
 - White Box (basis path, condition, Data Flow, Loop, Symbolic execution)
 - Black Box
 - Use-case based
 - State dependent
- Software testing can detect errors in the software but cannot prove that the software is free of errors
- Test coverage provides assurance
 - Logic Coverage
 - Decision Coverage
 - Condition Coverage
 - Modified Condition/Decision Coverage
- Software testing increases the development team's confidence in the software product

Last Slide

