Tools for Integration Testing
What is integration testing?

- **Unit testing** is testing modules individually
  - A software module is a self-contained element of a system
- Then modules need to be put together to construct the complete system
  - Construction of a working system from the pieces is not a straightforward task because of numerous *interface errors*
- Objective of *system integration testing* (SIT)
  - to build a “working” version of the system
  - putting modules together
  - ensuring that the additional modules work as expected without disturbing the functionalities of the modules already put together
What is integration testing?

- *Integration testing* is said to be complete when
  - The system is fully integrated together
  - All the test cases have been executed
  - All the severe and moderated defects found have been fixed
Types of module interfaces

- Common paradigms for interfacing modules:
  - Procedure call interface
  - Shared memory interface
  - Message passing interface

The problem arises when we “put modules together” because of interface errors

**Interface errors**

Interface errors are those that are associated with structures existing outside the local environment of a module, but which the module uses
Types of Interface Errors

- Construction
- Inadequate functionality
- Location of functionality
- Changes in functionality
- Added functionality
- Misuse of interface
- Misunderstanding of interface
- Data structure alteration
- Inadequate error processing
- Additions to error processing
- Inadequate post-processing
- Inadequate interface support
- Initialization/value errors
- Validation of data constraints
- Timing/performance problems
- Coordination changes
- Hardware/software interfaces
Different scopes of Integration

- **Intra Class**
  - Integration concerns different methods within a class
  - Typical strategy: constructors - accessors - predicates - modifiers - iterators – destructors

- **Package Integration**
  - Integration concerns classes within a package

- **Subsystem/System Integration**
  - Integration of packages
Common approaches for system integration testing

- Top-down
- Bottom-up
- Sandwich
- Big-bang
Components for Integration Testing

- When testing a module in isolation
  - Called modules need to be replaced
  - Tested module needs to be called
Top-down Integration

- Incremental strategy
  1. Test high-level modules, then
  2. Modules called until lowest level modules
Top-down Integration

• Possible to alter order such to test as early as possible
  • critical units
  • input/output units
Top-down Integration

- Advantages
  - Fault localization easier
  - Few or no *drivers* needed
  - Possibility to obtain an early prototype
  - Different order of testing/implementation possible
  - Major design flaws found first
    - in logic modules on top of the hierarchy

- Dis-advantages
  - Need lot of stubs
  - Potentially reusable modules (in bottom of the hierarchy) can be inadequately tested
Bottom-up Integration

- Incremental strategy
  1. Test low-level modules, then
  2. Modules calling them until highest level module
Bottom-up Integration

- **Advantages**
  - Fault localization easier (than big-bang)
  - No need for stubs
  - Reusable modules tested thoroughly
  - Testing can be in parallel with implementation

- **Dis-advantages**
  - Need of *drivers*
  - High-level modules (that relate to the solution logic) tested in the last (and least)
  - No concept of early skeletal system
Sandwich Integration

- Combines top-down and bottom-up approaches
- Distinguish 3 layers
  - logic (top) - tested top-down
  - middle
  - operational (bottom) – tested bottom-up
Modified Sandwich Integration

- Combines top-down and bottom-up approaches
- Modules tested first in isolation before integration
Big Bang Integration

- Non-incremental strategy

1. Unit test each module in isolation
2. Integrate as a whole
Big Bang Integration

- Advantages
  - convenient for small systems

- Dis-advantages
  - need driver and stubs for each module
  - do not allow parallel development
  - fault localization difficult
  - easy to miss interface faults
Drivers

- A driver is a method used to call a module that is being tested.
  - Sends values to module under test.
  - Collects return values
- JUnit test cases essentially are test drivers.
- The following could also be a test driver:

```java
public class TestSomething
{
    public static void main ( String[] args )
    {
        int result = 0;
        double send = 98.6;
        Whatever what = new Whatever;
        System.out.println( "Sending someFunc 98.6. " );
        result = what.someFunc( send );
        System.out.println( "SomeFunc returned: " + result );
    }
}
```
Stubs

• A stub is a module used specifically for testing that replaces a called module.

• Why?
  • To gain access to an interface that is otherwise not visible.
  • Actual module is not ready or reliable.
  • Actual module would be a component such as a database, network requiring a large amount of setup.
Creating a Stub

• The stub must have the same signature as the real module.
  • Same name, parameter list, return value type, modifiers (*static*, *public*, etc.)

• Function:
  • If the module is called, check that the incoming parameters are as expected.
  • Provide suitable return values.
    – The stub should *not* attempt to replicate the functionality of the actual module. Instead, preset return values should be used.
Example of a Stub

• Method signature:

```java
public static int someMethod(int aParameter)
```

• A possible stub for this method

```java
public static int someMethod(int aParameter)
{
    int result = 42;
    System.out.println("In someMethod(): "+
                    "Input int = " + aParameter);
    System.out.println("Returning 42.");
    return result;
}
```
Common Stub functions

- Display/log trace message
- Display/log passed parameters
- Return a value according to test objective. This could be determined from:
  - a table
  - an external file
  - based on a search according to parameter value
Stubs – multiple calls

```java
void main()
{
    int x, y; // 1
    x = TestMe1.a();
    if (x > 0)
    {
        // 2
        y = TestMe2.b(x);
        TestMe3.c(y);
    }
    else
    {
        // 3
        TestMe3.c(x);
    }
    System.exit(0); // 4
}
```

- Test for path 1 – 2 - 4
  - Stub for `TestMe1.a()` needs to return `x > 0`

- Test for path 1 – 3 – 4 – 6 – 7
  - Stub for `TestMe1.a()` needs to return `x <= 0`

- Stub will have to be able to return the appropriate values depending on the sequence of calls.

- Problem: stubs can become too complex
Mock Objects as Stubs

• A mock object is used as a substitute object in a situation where a stub is called for.

• Set up variables within objects to have interface types instead of specific class types.

• Multiple classes can implement the interface, if necessary.
  • In particular, a mock object class can also implement the interface, effectively making object instances as stubs.
  • Can be refined incrementally by replacing with actual code
Dependent class associations

- Many classes are set up so that associated classes are referred to specifically by their name, as a type for the association reference variable.
  - This makes it difficult to replace the dependent class.
Dependent Classes

- **Client** is the class which we would like to test.
- **Helper** is a class used by **Client**.

  - Reasons for replacing **Helper**:
    - If it was not ready
    - We want direct control over what is sent to **Client**.
Example

• We are going to test a class called **TestMe**.
  • The class is a wrapper around a class called **Converter** that will convert Celsius to Fahrenheit.

• Class **Converter** has two methods:
  • Method to convert Celsius temperatures to Fahrenheit temperatures.
  • Method to convert Fahrenheit temperatures to Celsius temperatures.

• Class **TestMe** is a wrapper to this class that has one method `convert()`, and uses constants from an enumerated type **ConversionType** to choose the conversion direction.
Example

- Class TestMe creates the Converter object and keeps a reference to it.

- The TestMe object chooses which method to call in Converter.

- The Converter object does the appropriate conversion, and returns its result to the TestMe object.
Creation of associated object

- The Converter object is created by the TestMe constructor:

```java
private Converter converter;

public TestMe( )
{
    converter = new Converter( );
}
```
public Double convert( Double temperature, ConversionType direction )
{
    Double result = null;
    if ( converter != null )
        switch ( direction )
        {
            case C_TO_F:
            {
                result = converter.convertCtoF( temperature );
                break;
            }
            case F_TO_C:
            {
                result = converter.convertFtoC( temperature );
                break;
            }
        }
    else
        throw new IllegalStateException( "No converter specified" );

    return result;
}
Class ConversionDirection

// Constants to specify conversion direction

public enum ConversionType
{
    F_TO_C,
    C_TO_F;
}
public class Converter
{
    public Double convertCtoF( Double temperature )
    {
        return new Double( temperature.doubleValue( ) * 9.0 / 5.0 + 32.0 );
    }

    public Double convertFtoC( Double temperature )
    {
        return new Double( ( temperature.doubleValue( ) - 32.0 ) * 5.0 / 9.0 );
    }
}
Portion of a Test Case for TestMe

```java
private TestMe testMe;
private static double tolerance = 0.0001;

@Before
public void setUp()
{
   testMe = new TestMe();
}

@Test
public void testCtoF_Freezing( )
{
   double input = 0.0;
   double expected = 32.0;
   double actual = testMe.convert( input, ConversionType.C_TO_F);
   assertEquals( expected, actual, tolerance );
}
```
Goals:

- At present, class `TestMe` is tied to a specific type of object.
  - Modifications need to be made to `TestMe` if other types are used.
- We could use subclasses of `Converter`, but these subclasses still inherit behaviour from `Converter`.
  - This functionality may not be complete.
  - Does not provide for replacing `Converter` objects quickly.
- Alternative: use the interface mechanism.
Dependent class associations with interfaces

- Have classes store references using interface types instead of classes.
  - Mock object can then implement the interface with minimal functionality
Class dependencies – object creation

- Another factor that can make using mock objects more difficult is when an object creates its dependent objects.
  - Using `new ObjectClassName()` to create an object still ties the associated object to a particular class, even if the reference is then assigned to an interface type.
Use of interfaces

• Create an interface IHelper.
  • Class Client now has an association with IHelper instead of Helper. Any class that implements the IHelper interface can replace Helper.
  • Class MockHelper will be a mock object that can replace Helper.
Strategies

● Design for testability
  ● (but you tell the designers that they are designing for extensibility / maintenance 😊)

● Option 1: Use the “inversion of control” pattern.
● Option 2: Use the (Abstract) Factory pattern
Inversion of Control Pattern

• The “rule” for this pattern is that objects should not directly create any objects for which they might have an association.
  • If object A is supposed to work with object B, then A should not create B.
  • Instead, A and B should be created separately, and then they are passed references to each other.
Inversion of Control Pattern

• Rationale:
  • For future extensibility, the objective is to increase the independence of classes.
  • If object B is replaced by object B, version 2.0, class A doesn’t have to be changed.

• Rationale for testing:
  • We can replace “real” object B by a mock object for the purpose of testing object A.
Example: Interfaces and inversion of control
Example: Interfaces and inversion of control

- Class **TestMe** now has an association with **IClone** instead of **Converter**.
  - Instead of having **TestMe** create a Converter object, a reference to an **IClone** is passed in via an accessor method. A reference to **IClone** is then stored within **TestMe**.

- Either a **Converter** or **MockConverter** can be created, and then passed to **TestMe** via the accessor.
  - A test case can create either the real or the mock object as appropriate.
// Interface for class to be mocked.

// Implementation will convert Fahrenheit to Celsius, or vice versa

public interface Converter
{
    public double convertFtoC(double temp);
    public double convertCtoF(double temp);
}
public class Converter implements IConverter
{
    public Double convertCtoF( Double temperature )
    {
        return new Double( temperature.doubleValue() * 9.0 / 5.0 + 32.0 );
    }

    public Double convertFtoC( Double temperature )
    {
        return new Double( ( temperature.doubleValue() - 32.0 ) * 5.0 / 9.0 );
    }
}
public class MockConverter implements IConverter {
    // Objective: for a few cases we know without using the formulas, return the appropriate results.

    public Double convertCtoF( Double temperature ) {
        Double result = null;
        if ( temperature.equals( 0.0 ) ) // Freezing
            result = new Double( 32.0 );
        else if ( temperature.equals( 100.0 ) ) // Boiling
            result = new Double( 212.0 );
        else if ( temperature.equals( 37.0 ) ) // Body temp.
            result = new Double( 98.6 );
        else if ( temperature.equals( -40.0 ) ) // Same!
            result = new Double( -40.0 );
        return result;
    }

    // Similar implementation of convertFtoC
}

private TestMe testMe;
private IConverter converter;
private static double tolerance = 0.0001;

@Before
public void setUp()
{
    testMe = new TestMe();
    converter = new Converter();
    testMe.setConverter( converter );
}

@Test
public void testCtoF_Freezing()
{
    double input = 0.0;
    double expected = 32.0;
    double actual = testMe.convert( input, ConversionType.C_TO_F );
    assertEquals( expected, actual, tolerance );
}
Testing using the mock Converter

private TestMe testMe;
private IConverter mock;
private static double tolerance = 0.0001;

@Before
public void setUp()
{
    testMe = new TestMe();
    mock = new MockConverter();
    testMe.setConverter( mock );
}

@Test
public void testCtoF_Freezing()
{
    double input = 0.0;
    double expected = 32.0;
    double actual = testMe.convert( input, ConversionType.C_TO_F );
    assertEquals( expected, actual, tolerance );
}
Factory Pattern

- Objective is to isolate the creation of any objects used in an application.
- Rationale: As class names change through software evolution, only the Factory needs to be updated.
- A class known as a “factory” will be responsible for any object creation,
  - Whenever the application needs a new object, ask the factory class to create the object.
  - Alternative: create a single Factory object from the Factory class.
Abstract Factory Pattern

• With this approach, the Factory is itself an interface, and there could be several implementations of the Factory.

• Rationale (by example):
  • An application may want to create a user interface button, and so it asks the abstract factory to create an object.
  • There could be three implementations of the Factory, one of which is currently active:
    – Windows (as in Microsoft) factory
    – X-Windows (for Linux, etc.) factory
    – Macintosh button factory

• Rationale for testing:
  • Substitute a Factory that creates mock objects.
UML diagram of Abstract Factory pattern
Example, using Abstract Factory Pattern
Example: Factory interface and classes

```java
public interface IFactory {
    IConverter createProduct();
}

public class Factory implements IFactory {
    public IConverter createConverter() {
        return new Converter(); // Creates the real object
    }
}

public class MockFactory implements IFactory {
    public IConverter createConverter() {
        return new MockConverter(); // Creates a mock object
    }
}
```
private static IFactory factory = null;
private IConverter converter;

public TestMe()
{
    converter = null;
    if ( factory != null )
    {
        converter = factory.createConverter();
    }
}

public static void setFactory( IFactory theFactory )
{
    TestMe.factory = theFactory;
}
Testing using the production factory

```java
private TestMe testMe;
private static IFactory factory;
private static double tolerance = 0.0001;

@BeforeClass
public void setUpBeforeClass()
{
    factory = new Factory();
    TestMe.setFactory( factory );
}

@Before
public void setUp()
{
    testMe = new TestMe();
}

@Test
public void testCtoF_Freezing()
{
    double input = 0.0;
    double expected = 32.0;
    double actual = testMe.convert( input, ConversionType.C_TO_F );
    assertEquals( expected, actual, tolerance );
}
```
Testing using the mock factory

private TestMe testMe;
private static IFactory factory;
private static double tolerance = 0.0001;

@BeforeClass
public void setUpBeforeClass()
{
    factory = new MockFactory();
    TestMe.setFactory( factory );
}

@Before
public void setUp()
{
    testMe = new TestMe();
}

@Test
public void testCtoF_Freezing( )
{
    double input = 0.0;
    double expected = 32.0;
    double actual = testMe.convert( input, ConversionType.C_TO_F );
    assertEquals( expected, actual, tolerance );
}
Automating Mock Object Creation

• What did we have to do to create the mock object?
  • Identify methods that can be called.
  • Create a few cases where we know the output for given input.
    – Identify the input as a known case.
    – Return the associated output.

• Objective: build a mock object automatically.
  • Structure can be determined using Java’s reflection capability.
  • Capability to allow user to specify the mock object’s behaviour easily needs to be provided.
Frameworks for Mock Objects

- EasyMock and jMock are two frameworks for creating mock objects at run time.
- Use Java reflection to create a mock object class that matches an interface.
- EasyMock
  - Open source project on SourceForge.
  - web site: www.easymock.org
  - current version: 2.5.2 (Sept. 2009)
- jMock
  - Open source project on Codehaus
  - web site: www.jmock.org
  - current version 2.5.1 (Aug. 2008)
EasyMock Usage

• General steps:

1. Create the mock object
2. Tell the mock object how to behave when called.
3. Activate the mock object (this is called “replay” in the EasyMock documentation)
4. Use the mock object during a test.
5. After the test has run, check with the mock object to see if the expected calls were made.
Mock Object Creation

• A call to the EasyMock framework will generate a mock object.
  • Specify a Java object of type **Class** as the parameter to the method call.
    – Normally, this would be an interface.
  • At this point, you can specify whether the mock object is to have **strict** behaviour.
    – **Normal**: mock object will not enforce the ordering of calls to its methods.
    – **Strict**: calls to mock object methods must be in the sequence that is specified.
Example: Creation of Mock Object

```java
import org.easymock.EasyMock;

private TestMe testMe;
private IConverter mock;

@Before
public void setUp()
{
    testMe = new TestMe();
    mock = EasyMock.createMock( IConverter.class );
    testMe.setConverter( mock );
}
```
Specify the Mock Object behaviour

- Items to do:
  - Select a method in the mock object that could be called.
  - For that method:
    - Provide a set of input parameter values to be matched.
    - Provide a value to be returned when the method is called with those parameter values.
Specify the Mock Object behaviour

- Mechanism:
  - Option 1: If no value needs to be returned, just make the method call on the mock object with a parameter value that should be provided.
  - Option 2: If a value needs to be returned, use the EasyMock static method `expect()` with the method call (a `Method` object) as a parameter.
    - The return value is of a type that permits calling a method `andReturn()`. The parameter to that method will be the return value when the input matches the values that appear in the call to `expect()`.
Set up mock object behaviour

```java
@Test
public void testCtoF_Freezing()
{
    Reference to mock object
    EasyMock.expect( mock.convertCtoF( 0.0 ) ).andReturn( 32.0 );

    // Activate mock object
    // Rest of test case
}
```

Called method

Case is when parameter matches this value.

When method is called with input parameter 0.0, return the value 32.0
Activating the Mock Object

• After specifying the mock object’s future behaviour, the next step is to activate it.

• With the active mock object, we can then run our tests that call the object under test that collaborates with the mock object.
Set up mock object behaviour, and run test

```java
@Test
public void testCtoF_Freezing() {
    EasyMock.expect(mock.convertCtoF(0.0)).andReturn(32.0);
    EasyMock.replay(mock);
    double input = 0.0;
    double expected = 32.0;
    double actual = testMe.convert(input, ConversionType.C_TO_F);
    assertEquals(expected, actual, tolerance);
}
```
After the test

- We want to be sure that the mock object actually received the expected calls.
- The EasyMock framework provides a `verify()` method that takes the mock object as a parameter.
  - If the expected calls occurred, the method will return.
  - If an expected call was missed, an exception will be thrown.
    - This exception will cause a JUnit failure.
Verification of calls to mock object

private IConverter mock;

@After
public void tearDown( )
{
    EasyMock.verify( mock );
}
Additional EasyMock Features

- Specifying a number of repeated calls with the same input parameters.
- Specifying that the mock object is to throw an exception when a method is called with a specified value.
- Specifying that a method should be called with specified values:
  - At least once
  - Between \texttt{min} and \texttt{max} number of times
  - Any number of times
- Pattern matching on input values, instead of equality.
jMock Usage

General steps:
1. Create the mock object
2. Define the “expectations” for the mock object.
3. Use the mock object during a test.

The test runner will ensure that the expectations are met, or the test case will fail.
- **Caveat:** jMock’s test runner looks for internal classes within junit.jar. The classes are available in junit.jar if you download the library from junit.org, but are **not** in the version of the JUnit library installed within Eclipse.
jMock JUnit test structure

```java
import org.jmock.Expectations;
import org.jmock.Mockery;
import org.jmock.integration.junit4.JMock;
import org.jmock.integration.junit4.JUnit4Mockery;

@RunWith( JMock.class )
public class JMockTest
{
    private Mockery context = new JUnit4Mockery();

    // tests go here
}
```
Mock Object Creation

- jMock uses an interface called **Mockery** to represent the context in which a mock object is used.

- Create an instance of a Mockery specific to the version of JUnit:
  ```java
  Mockery context = new JUnit4Mockery();
  ```

- To create a mock object:
  ```java
  IConverter mock = context.mock( IConverter.class );
  ```
  - Specify a Java object of type **Class** as the parameter to the method call.
    - Normally, this would be an interface.
private TestMe testMe;
private Mockery context = new JUnit4Mockery();
IConverter mock;

@Before
public void setUp( )
{
    testMe = new TestMe( );
    mock = context.mock( IConverter.class );
    testMe.setConverter( mock );
}
Specify the Mock Object behaviour

• jMock defines behaviour using “expectations”.

• Each expectation consists of:
  • An invocation count
  • A mock object method call reference, with specified parameter values
  • [optional] An action to perform.
    – example: return a value.
Invocation counts

`exactly(n).of(m)`
- Method `m` must be called exactly `n` times.

`atLeast(n).of(m)`
- Method `m` must be called `n` or more times.

`atMost(n).of(m)`
- Method `m` must be called `n` or fewer times.

`between(min,max).of(m)`
- Method `m` must be called at least `min` times and at most `max` times.

`allowing(m)`
- Method `m` can be called, but does not have to be called.

`never(m)`
- Method `m` cannot be called.

`one(m)`
- A shorthand for `exactly(1).of(m)`
@Test
public void testCtoF_Freezing( )
{
    context.checking( new Expectations( )
    {
        one( mock ).convertFtoC( 32.0 ); will( returnValue( 0.0 ) );
    });
    // Rest of test case
}