



Agent and Object Technology Lab
Dipartimento di Ingegneria dell'Informazione
Università degli Studi di Parma



Software Engineering

Implementation and Maintenance

Prof. Agostino Poggi

- ◆ Implementation involves taking all of the detailed design documents from the design phase and transforming them into the actual system
- ◆ Primary Implementation activities include:
 - Coding and debugging
 - Testing
 - Documentation completion
 - Installation

- ◆ Good coding style is important because programs are:
 - Written once
 - Read many times
- ◆ Each software company enforces its own coding standards (house style)
- ◆ Important aspects of coding style:
 - Layout
 - Naming
 - Commenting

- ◆ Comments are as important as the code itself
 - Determine successful use of code
 - Determine whether code can be maintained
 - Creation/maintenance = 1/10
- ◆ Documentation belongs in code or as close as possible
 - Code evolves, documentation drifts away
 - Put specifications in comments next to code when possible
 - Write a short description for each method
 - Link the code to the external documentation
- ◆ Avoid useless comments

- ◆ Component Diagrams

- Used to document dependencies between components, typically files, either compilation dependencies or run-time dependencies

- ◆ Deployment Diagrams

- Used to show the configuration of run-time processing elements and the software components and processes that are located on them

- ◆ CASE tools
- ◆ Compilers, interpreters and run-times
- ◆ Visual editors
- ◆ IDEs
- ◆ Version control systems
- ◆ Database management systems
- ◆ Testing tools
- ◆ Installation tools
- ◆ Conversion tools

- ◆ Testing involves bringing all the project pieces together into a special testing environment to test for errors, bugs, and interoperability, in order to verify that the system meets all the business requirements defined in the analysis phase
- ◆ Primary testing activities include:
 - Write the test conditions
 - Perform the system testing
 - Provide feedback to improve software

- ◆ Error prevention (before the system is released):
 - Use good programming methodology to reduce complexity
 - Use version control to prevent inconsistent systems
 - Apply verification to prevent algorithmic bugs
- ◆ Error detection (while system is running):
 - Testing: create failures in a planned way
 - Debugging: start with an unplanned failures
 - Monitoring: deliver information about system state
- ◆ Error recovery (after the system is released):
 - Data base systems (atomic transactions)
 - Modular redundancy
 - Recovery blocks

- ◆ Error
 - Any discrepancy between an actual, measured value and a theoretical, predicted value
- ◆ Fault
 - A condition that causes the software to malfunction or fail (cause)
- ◆ Failure
 - The inability of a piece of software to perform according to its specifications (effect)
 - Failures are caused by faults, but not all faults cause failures
 - A piece of software has failed if its actual behavior differs in any way from its expected behavior

- ♦ A test plan specifies how to demonstrate that the software is free of faults and behaves according to the requirements specification
- ♦ A test plan breaks the testing process into specific tests, addressing specific data items and values

- ◆ Each test has a test specification that documents the purpose of the test
- ◆ If a test is to be accomplished by a series of smaller tests, the test specification describes the relationship between the smaller and the larger tests
- ◆ The test specification must describe the conditions that indicate when the test is complete and a means for evaluating the results

- ♦ A test oracle is the set of predicted results for a set of tests, and is used to determine the success of testing
- ♦ Test oracles are extremely difficult to create and are ideally created from the requirements specification

- ♦ A test case is a set of inputs to the system
- ♦ Successfully testing a system hinges on selecting representative test cases
- ♦ Poorly chosen test cases may fail to illuminate the faults in a system
- ♦ In most systems exhaustive testing is impossible, so a white (glass) box or black box testing strategy is typically selected

- ◆ Glass box testing
 - Allowed to examine code
 - Attempt to improve thoroughness of tests
 - Based on the knowledge of the code

- ◆ Black box testing
 - Treat program as “black box”
 - Test behavior in response to inputs
 - Based on the knowledge of the expected result

- ♦ Logic of a piece of code defines a set of possible execution paths, flow-paths through the function
 - Think of a flow-chart
 - The inputs control which path is taken
- ♦ A good set of test data might make sure every possible path is followed
 - This tests every possible behavior
- ♦ The problem is that for even small programs with loops and conditionals, the total number of paths becomes too large too quickly

- ◆ Boundary values, boundary conditions
 - Extremes are incorrectly identified
 - E.g., the last item or first item in a collection
 - E.g., the start or end of a range of values
- ◆ Off-by-one errors
 - Related to boundary conditions
 - E.g., using the n th value instead of $n-1$
 - E.g., using 0 instead of 1
- ◆ Incorrect inputs
 - Rely on pre-conditions defined for your functions
 - Throws a related exception if programmed
 - Causes an unexpected behavior otherwise

- ◆ Alpha test
 - Test components during development
 - Usually glass box test
 - Divided in unit and integration test
- ◆ Beta test
 - Test in real user environment
 - Always black box test
- ◆ Acceptance

- ♦ The units comprising a system are individually tested
- ♦ The code is examined for faults in algorithms, data and syntax
- ♦ A set of test cases is formulated and input and the results are evaluated
- ♦ The module being tested should be reviewed in context of the requirements specification

- ◆ Not possible to test all flow paths
 - Many paths by combining conditionals, switches, etc.
 - Infinite number of paths for loops
 - New paths caused by exceptions
- ◆ Test coverage
 - Alternative to flow path
 - Ensure each line of code tested
 - Does not capture all possible combinations

- ◆ The goal is to ensure that groups of components work together as specified in the requirements document
- ◆ Four kinds of integration tests exist
 - Structure tests
 - Functional tests
 - Stress tests
 - Performance tests

- ♦ Big bang integration
- ♦ Bottom up integration
- ♦ Top down integration
- ♦ Sandwich testing
- ♦ Variations of the above

- ♦ The subsystem in the lowest layer of the call hierarchy are tested individually
- ♦ Then the next subsystems are tested that call the previously tested subsystems
- ♦ This is done repeatedly until all subsystems are included in the testing
- ♦ Tests are usually performed through a test driver
 - It calls a subsystem and passes a test case to it

- ◆ Test the top layer or the controlling subsystem first
- ◆ Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- ◆ Do this until all subsystems are incorporated into the test
- ◆ Tests are usually performed through a test stub
 - It simulates the activity of a missing subsystem by answering to the calling sequence of the calling subsystem and returning back fake data

- ♦ Bad for functionally decomposed systems because it tests the most important subsystem (UI) last
- ♦ Useful for integrating object-oriented systems, real-time systems and systems with strict performance requirements
- ♦ Test cases can be defined in terms of the functionality of the system
- ♦ Writing stubs can be difficult because they must allow all possible conditions to be tested
- ♦ Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods

- ◆ Test entire software placing it in user environment
- ◆ Test software with
 - Real-world data
 - Real users
 - Typical operating conditions
 - Test cases selected by users
- ◆ Ensure software meets specifications

- ◆ Obviously you have to test your code to get it working in the first place
 - You can do *ad hoc* testing (running whatever tests occur to you at the moment), or
 - You can build a test suite (a thorough set of tests that can be run at any time)

- ♦ Advantages of a test suite
 - Reduces total number of bugs in delivered code
 - Makes code much more maintainable and refactorable
 - This is a *huge* win for programs that get actual use!
- ♦ Disadvantages of a test suite
 - It's a lot of extra programming
 - This is true, but the use of a good test framework can help quite a bit
 - You don't have time to do all that extra work
 - *But* experiments repeatedly show that test suites reduce debugging time more than the amount spent building the test suite

Test Condition Number	Date Tested	Tester	Test Condition	Expected Result	Actual Result	Pass/Fail
1	1/1/05	Emily Hickman	Click on System Start Button	Main Menu appears	Same as expected result	Pass
2	1/1/05	Emily Hickman	Click on Logon Button in Main Menu	Logon Screen appears asking for Username and Password	Same as expected result	Pass
3	1/1/05	Emily Hickman	Type Emily Hickman in the User Name Field	Emily Hickman appears in the User Name Field	Same as expected result	Pass
4	1/1/05	Emily Hickman	Type Zahara 123 in the password field	XXXXXXXXXX appears in the password field	Same as expected result	Pass
5	1/1/05	Emily Hickman	Click on O.K. button	User logon request is sent to database and user name and password are verified	Same as expected result	Pass
6	1/1/05	Emily Hickman	Click on Start	User name and password are accepted and the system main menu appears	Screen appeared stating logon failed and username and password were incorrect	Fail

- ◆ The components to be tested are object classes that are instantiated as objects
- ◆ Larger grain than individual functions so approaches to white-box testing have to be extended
- ◆ No obvious 'top' to the system for top-down integration and testing

- ♦ Testing operations associated with objects
- ♦ Testing object classes
- ♦ Testing clusters of cooperating objects
- ♦ Testing the complete system

- ◆ Complete test coverage of a class involves
- ◆ Testing all operations associated with an object
- ◆ Setting and interrogating all object attributes
- ◆ Exercising the object in all possible states
- ◆ Inheritance makes it more difficult to design object class tests as the information to be tested is not localized

- ◆ Levels of integration are less distinct in object-oriented systems
- ◆ Cluster testing is concerned with integrating and testing clusters of cooperating objects
- ◆ Identify clusters using knowledge of the operation of objects and the system features that are implemented by these clusters

- ◆ Use-case or scenario testing
 - Testing is based on a user interactions with the system
 - Has the advantage that it tests system features as experienced by users
- ◆ Thread testing
 - Tests the systems response to events as processing threads through the system
- ◆ Object interaction testing
 - Tests sequences of object interactions that stop when an object operation does not call on services from another object

◆ Inheritance

- Methods inherited from a superclass must be retested in the context of the subclasses
- Testing using the context of the superclass may not include all the cases that may occur in context of the subclasses

◆ Polymorphism

- Parameters have more than one set of values and an operation may be implemented by more than one method

◆ Dynamic binding

- Methods that implement an operation are unknown until run time

- ◆ Tests are written before the code itself
- ◆ If code has no automated test case, it is assumed not to work
- ◆ A test framework is used so that automated testing can be done after every small change to the code
 - This may be as often as every 5 or 10 minutes
- ◆ If a bug is found after development, a test is created to keep the bug from coming back

- ◆ First, you create one test to define some small aspect of the problem at hand
- ◆ Then you create the simplest code that will make that test pass
- ◆ Then you create a second test
- ◆ Then you add some new code to make this new test pass
 - But no more until you have yet a third test
- ◆ You continue until there is nothing left to test

- ♦ Fewer bugs
- ♦ More maintainable code
- ♦ Continuous integration
- ♦ During development, the program always works
 - It may not do everything required, but what it does, it does right

- ♦ JUnit is a framework for writing tests
- ♦ JUnit was written by Erich Gamma (of design patterns fame) and Kent Beck (creator of XP methodology)
- ♦ JUnit uses Java's reflection capabilities (Java programs can examine their own code)

- ♦ JUnit helps the programmer:
 - Define and execute tests and test suites
 - Formalize requirements and clarify architecture
 - Write and debug code
 - Integrate code and always be ready to release a working version
- ♦ JUnit is not yet included in Sun's SDK, but an increasing number of IDEs include it
 - E.g., Eclipse

Calculator Class Example

```
package system.calc;

public class Calculator {
    private static int result; // Static variable where the result is stored
}
```

```
public void switchOn() { // Switch on the screen, display "hello", beep
    result = 0; // and do other things that calculator do nowadays
}

public void switchOff() { } // Display "bye bye", beep, switch off the screen

public int getResult() {
    return result;
}
```

```
public void add(int n) {
    result = result + n;
}

public void subtract(int n) {
    result = result - 1;
}

public void multiply(int n) {} //Not implemented yet

public void divide(int n) {
    result = result / n;
}

public void square(int n) {
    result = n * n;
}

public void squareRoot(int n) {
    for (; ; ) ; //Bug : loops indefinitely
}

public void clear() { // Cleans the result
    result = 0;
}
```


Calculator Class Example

```
package system.junit.calc;
import calc.Calculator;
import org.junit.Before;
import org.junit.Ignore;
import org.junit.Test;
import static org.junit.Assert.*;
public class CalculatorTest {
    private static Calculator calculator = new Calculator();
}
```

```
@Test(expected = ArithmeticException.class)
public void divideByZero() {
    calculator.divide(0);
}

@Ignore("not ready yet")
@Test
public void multiply() {
    calculator.add(10);
    calculator.multiply(10);
    assertEquals(calculator.getResult(), 100);
}
```

```
@Before
public void clearCalculator() {
    calculator.clear();
}

@Test
public void add() {
    calculator.add(1);
    calculator.add(1);
    assertEquals(calculator.getResult(), 2);
}

@Test
public void subtract() {
    calculator.add(10);
    calculator.subtract(2);
    assertEquals(calculator.getResult(), 8);
}

@Test
public void divide() {
    calculator.add(8);
    calculator.divide(2);
    assertEquals(calculator.getResult(), 5);
}
```

Calculator Class Example

```
java -ea org.junit.runner.JUnitCore system.test.calc.CalculatorTest
```

There were 2 failures:

1) subtract(system.test.calc.CalculatorTest)

java.lang.AssertionError: expected:<9> but was:<8>

at org.junit.Assert.fail(Assert.java:69)


2) divide(system.test.calc.CalculatorTest)

java.lang.AssertionError

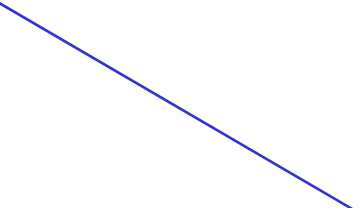
at system.test.calc.CalculatorTest.divide(CalculatorTest.java:40)

FAILURES!!!

Tests run: 4, Failures: 2



```
public void subtract(int n) {  
    result = result - 1;  
}
```



```
@Test  
public void divide() {  
    calculator.add(8);  
    calculator.divide(2);  
    assert calculator.getResult() == 5;  
}
```

- ♦ Is JUnit testing overkill for this little class?
- ♦ XP view is: if it isn't tested, assume it doesn't work
- ♦ You are not likely to have many classes this trivial in a real program, so writing JUnit tests for those few trivial classes is no big deal
- ♦ Often even XP programmers don't bother writing tests for *simple* getter methods such as getResult()

◆ @Test

- Annotates the test methods
- May have parameters declaring:
 - The type of exception that should be thrown
 - E.g., `@Test(expected = ArithmeticException.class)`
 - Test fails either if no exception is thrown or if a different exception is thrown
 - A time-out period in milliseconds
 - E.g., `@Test(time-out=10)`
 - Test fails if it takes more time than the one defined by the time-out

◆ @Ignore

- Informs the test runner to ignore the test, but reporting that it was not run

- ◆ @Before and @After
 - Methods annotated with @Before execute before every test
 - Methods annotated with @After execute after every test
 - There may be any number of @Before and @After methods
 - It is possible to inherit the @Before and @After methods
 - @Before: execution is down the inheritance chain (superclass first)
 - @After : execution is up the inheritance chain (subclass first)
- ◆ @BeforeClass and @AfterClass
 - Only a @BeforeClass method and a @AfterClass method are allowed
 - Provide one-time set up and tear down, that is they are respectively executes once before and after all the tests

- ◆ Use it to document a condition that you “know” to be true
- ◆ Use `assert false` in code that you “know” cannot be reached (such as a default case in a switch statement)
- ◆ Do not use `assert` to check whether parameters have legal values, or other places where throwing an Exception is more appropriate

- ◆ **assertEquals**
 - Asserts that either two objects or two primitive values are equal
- ◆ **assertTrue, assertFalse**
 - Assert that two Boolean values are either equal or are different
- ◆ **assertNull, assertNotNull**
 - Assert that an object either is null or is not null
- ◆ **assertSame, assertNotSame**
 - Assert that two objects either refer to the same object or do not refer to the same object
- ◆ **fail, failNotEquals, failSame, failNotSame**
 - Cause the unconditional / conditional failure of a test

- ♦ JUnit is designed to call methods and compare the results they return against expected results
- ♦ This works great for methods that just return results, but many methods have side effects
 - To test methods that do output, you have to capture the output
 - It's possible to capture output, but it's an unpleasant coding chore
 - To test methods that change the state of the object, you have to write the code that checks the state

- ◆ Heavy use of JUnit encourages a “functional” style, where most methods are called to compute a value, rather than to have side effects
 - This can actually be a good thing
 - Methods that just return results, without side effects (such as printing), are simpler, more general, and easier to reuse

- ◆ Sometimes it is just plain hard to test the system under test (SUT) because it depends on other components that cannot be used in the test environment
- ◆ In these cases each of such components can be replaced with a test double
 - A test double is any object or component used in place of the real component to execute the test
 - A test double doesn't have to behave exactly like the real component
 - A test double merely has to provide the same API as the real one so that the SUT thinks it is the real one!

- ◆ Dummy object
 - Is a placeholder object that is passed to the code under test as a parameter but never used
- ◆ Test stub
 - Is an object that is used by a test to replace a real component to force the system down the path we want for the test
- ◆ Mock object
 - Is an object that is used by a test to replace a real component and that returns hard coded values or values preloaded
- ◆ Fake object
 - Is an object that replace the functionality of the of the real object with an alternate implementation
 - I.e., returning a canned list of values instead of hitting a database

Test Stub and Mock Object

```
public class WarehouseStub implements
    Warehouse
{
    public void add(String product, int i) { }
    public int getInventory(String product) {
        return 0;
    }
    public boolean hasInventory(String product) {
        return false;
    }
    public void remove(String product, int i) { }
}
```

```
public class WarehouseMock implements Warehouse
{
    int inventoryResult;
    boolean hasInventoryResult;
    int expectedCalls, actualCalls;
    ...
    public int getInventory(String product) {
        actualCalls++;
        return inventoryResult;
    }
    public void setGetInventoryResult(int result) {
        this.inventoryResult = result;
        expectedCalls++;
    }
    ...
    public boolean verify(){
        return expectedCalls == actualCalls;
    }
}
```

- ◆ User documentation

- Written or visual information about an application system, how it works, and how to use it
- Help users understand how to use software

- ◆ System documentation

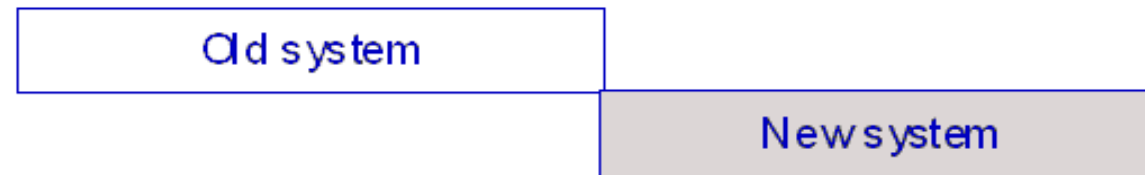
- Detailed information about a system's design specs, its internal workings, and its functionality
- Help coders understand how to modify, maintain software

- ♦ Training manuals organized around the tasks the users carry out
- ♦ On-line computer-based training that can be delivered when the users need it
- ♦ Reference manuals to provide complete description of the system in terms the users can understand
- ♦ On-line help replicating the manuals

- ◆ Use of overview, index, getting started instructions sections, i.e., all that make structured the documentation
- ◆ Based on the description of functionalities
- ◆ Oriented to help in the execution of the tasks of the systems and to recognize the state of the system
 - “How to ...”
 - Frequently Asked Questions
 - Messages & their meanings

- ◆ Set clear learning objectives for trainees
- ◆ Training should be practical and geared to the tasks the users will carry out
- ◆ Training should be delivered 'just in time' not weeks before the users need it
- ◆ Computer-based training can deliver 'just in time' training
- ◆ Follow up after the introduction of the system to make sure users haven't got into bad habits through lack of training or having forgotten what they had been told

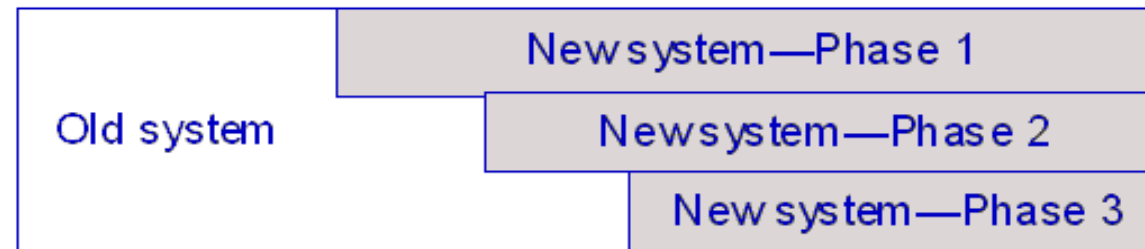
Direct change over



Parallel running



Phased changeover



Time

- ◆ On a date the old system stops and the new system starts
 - + Brings immediate benefits
 - + Forces users to use the new system
 - + Simple to plan
 - No fallback if problems occur
 - Contingency plans required for the unexpected
 - The plan must work without difficulties
- ◆ Suitable for small-scale, low-risk systems

- ◆ Old system runs alongside the new system for a period of time
 - + Provides fallback if there are problems
 - + Outputs of the two systems can be compared, so testing continues into the live environment
 - High running cost including staff for dual data entry
 - Cost associated with comparing outputs of two systems
 - Users may not be committed to the new system
- ◆ Suitable for business-critical, high-risk systems

- ◆ The new system is introduced in stages, department by department or geographically
 - + Attention can be paid to each sub-system in turn
 - + Sub-systems with a high return on investment can be introduced first
 - + Thorough testing of each stage as it is introduced
 - If there are problems rumors can spread ahead of the implementation
 - There can be a long wait for benefits from later stages
- ◆ Suitable for large systems with independent sub-systems

- ◆ Review the system
 - Whether it is delivering the benefits expected
 - Whether it meets the requirements
- ◆ Review the development project
 - Record lessons learned
 - Use actual time spent on project to improve estimating process
- ◆ Plan actions for any maintenance or enhancements

- ◆ Cost benefit analysis
 - Has it delivered?
 - Compare actual ones with projections
- ◆ Functional requirements
 - Have they been met?
 - Any further work needed?
- ◆ Non-functional requirements
 - Assess whether measurable objectives have been met
- ◆ User satisfaction
 - Quantitative and qualitative assessments of satisfaction with the product

- ◆ Problems and issues
 - Problems during the project and solutions so lessons can be learned
- ◆ Positive experiences
 - What went well?
 - Who deserves credit?
- ◆ Quantitative data for planning
 - How close were time estimates to actual ones?
 - How can we use this data?

- ◆ Candidate components for reuse
 - Are there components that could be reused in other projects in the future?
- ◆ Future developments
 - Were requirements left out of the project due to time pressure?
 - When should they be developed?
- ◆ Actions
 - Summary list of actions, responsibilities and deadlines

- ◆ Software maintenance is the modification of a software product after delivery
 - To correct faults
 - To improve performance or other attributes
 - To adapt the product to a modified environment

- ◆ Systems need maintaining after they have gone live
- ◆ Bugs will appear and need fixing
- ◆ Enhancements to the system may be requested
- ◆ Maintenance needs to be controlled so that bugs are not introduced and unnecessary changes are not made

- ♦ Helpdesk, operations and support staff need training to take on these tasks
- ♦ A change control system is required to manage requests for bug fixes and enhancements
- ♦ Changes need to be evaluated for their cost and their impact on other parts of the system, and then planned

- ◆ Bug reporting database
- ◆ Requests for enhancements
- ◆ Feedback to users
- ◆ Implementation plans for changes
- ◆ Updated technical and user documentation
- ◆ Records of changes made