Integration testing

unibz

Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Integration testing

- When to use it: when individual units are **combined and tested** as a group
- **Goal:** to expose faults in the **interaction** between integrated units
  - It verifies **interface specifications and model breakdown**
  - It verifies the **software architecture**

Integration testing

- Scaffolding is extensively required
  - *Test drivers* and *test stubs* are used to assist in integration testing

- **Faults** are related to interactions and compatibility

Integration Faults

- Inconsistent interpretation of **parameters or values**
  - Example: Mixed units (meters/yards) in Martian Lander
- Violations of **value domains, capacity, or size limits**
  - Example: Buffer overflow

Integration Faults

- **Side effects** on parameters or resources
  - Example: Conflict on (unspecified) temporary file
- Omitted or misunderstood **functionality**
  - Example: Inconsistent interpretation of web hits
- **Non-functional** properties
  - Example: **Unanticipated** performance issues

Integration Faults

- Dynamic **mismatches**
  - Example: Incompatible polymorphic method calls

Integration testing - approaches

- **Big Bang:** test almost all units in one shot
- **Top Down:** top-level units are tested first and lower level units are tested step by step after that. *Test Stubs* are needed to simulate lower level units
- **Bottom Up:** bottom level units are tested first and upper-level units step by step after that. *Test Drivers* are needed to simulate higher level units
- **Sandwich/Hybrid:** *combination* of Top Down and Bottom Up approaches

JUnit 5

- Use

```
@RunWith(Suite.class)
@SuiteClasses({MyFirstClassUnitTest.class,MySecondClassUnitTest.class})
```

- to select the units you want to test

Scaffolding

Why Scaffolding?

- Test case design includes input and expected output behaviour
- It may be simply a matter to fill a template (Acceptance test), but …

Why Scaffolding?

- **Testing can be complex:** a test specification is connected to several test cases:
    - e.g., a sorted sequence, length greater than 2, with items in ascending order with no duplicates
- We need a structure to support test execution

Why Scaffolding?

- To **test in small:** independent drivers just test some functionalities of a large user interface
- To **drive coding** (e.g., TDD)
- To perform **integration testing**

Elements

- Test driver
- Test stub
- Test harness
- Oracle

Test driver

- To drive program under test through test cases
- *Test driver* is a software module used to **invoke a module under test** and,
- It often provides *test inputs, control and monitor execution, and report test results*
- Drivers can become automated test cases

Example - driver

```
movePlayer(Player1, 2);
```

- Call to movePlayer with Player1 and two-spaces movement
- A unit test would execute this driver and test through *myPlayer.getPosition()* to make sure the player is now on the expected cell on the board

Example

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import it.unibz.Player;
class PlayerTest {
static Player thePlayer;

     @BeforeAll
     public static void setUp() {
          thePlayer = new Player("Babsi");
     }
     public Player movePlayer(Player thePlayer, int increment) {
          this.thePlayer = thePlayer;
          thePlayer.position += increment;
          return thePlayer;
     }
     @Test
     void testPosition() {
          movePlayer(thePlayer, 2);
          assertEquals(1,thePlayer.getPosition(thePlayer));
     }
}
```

Driver to test the position; I am not testing the existence of a current Player! For this I need another test

This test fails!

Test stub

- A stub is a program statement **substituting for the body of a software module** that is or will be defined elsewhere or
- A dummy component or object used to **simulate the behaviour of a real component** until that component has been developed

Test stub

- Developing stubs allows programmers to call a method in the code being developed, even if the method does not yet have the desired behaviour
- Stubs can be “filled in” to form the actual method

Example - stubs

- If the movePlayer() has not been written yet

```
public Player movePlayer(Player thePlayer, int
increment) {
        return thePlayer;
    }
```

- Or if it has been written

```
public Player movePlayer(Player thePlayer, int increment)
{
        this.thePlayer = thePlayer;
        thePlayer.position += increment;
        return thePlayer;
    }
```

Example

Stub

```
import static org.junit.jupiter.api.Assertions.*;
import org.junit.jupiter.api.BeforeAll;
import org.junit.jupiter.api.Test;
import it.unibz.Player;
class PlayerTest {
static Player thePlayer;

    @BeforeAll
    public static void setUp() {
        thePlayer = new Player("Babsi");
    }
    public Player movePlayer(Player thePlayer, int increment) {
        return thePlayer;
    }
    @Test
    void testPosition() {
        movePlayer(thePlayer, 2);
        assertEquals(1,thePlayer.getPosition(thePlayer));
    }
}
```

Test harness

- Environment in which to **execute the tests**
- Substitutes for other parts of the deployed environment
  - Ex: Software simulation of a hardware device, Unit frameworks

Oracle

- A test oracle is a piece of software that **provides a pass/fail service** to the program execution (e.g., *assert*)
- In principle, an oracle **classifies every execution and detects every failure**, but it can even **give false alarms**
  - False alarms increase cost of maintenance and reduce resources to dedicate to real failures
  - Thus, *there is no perfect oracle*

Examples - false positive

- As the exception expectation is placed around the whole test method, this might not actually test what is intended to be tested

```
@Test(expected = FooException.class)
public void testWithExceptions() {
   foo.prepareToDoStuff();
   foo.doStuff();
}
```

Expected exceptions

- It lacks the ability of asserting both the message and the cause of the exception that has been thrown.
- As good exception messages are valuable, assertions on messages should be taken into account

- JUnit 5

```
Throwable thrown = assertThrows(FooException.class,
() -> foo.doStuff());
```

Exercise

Auction

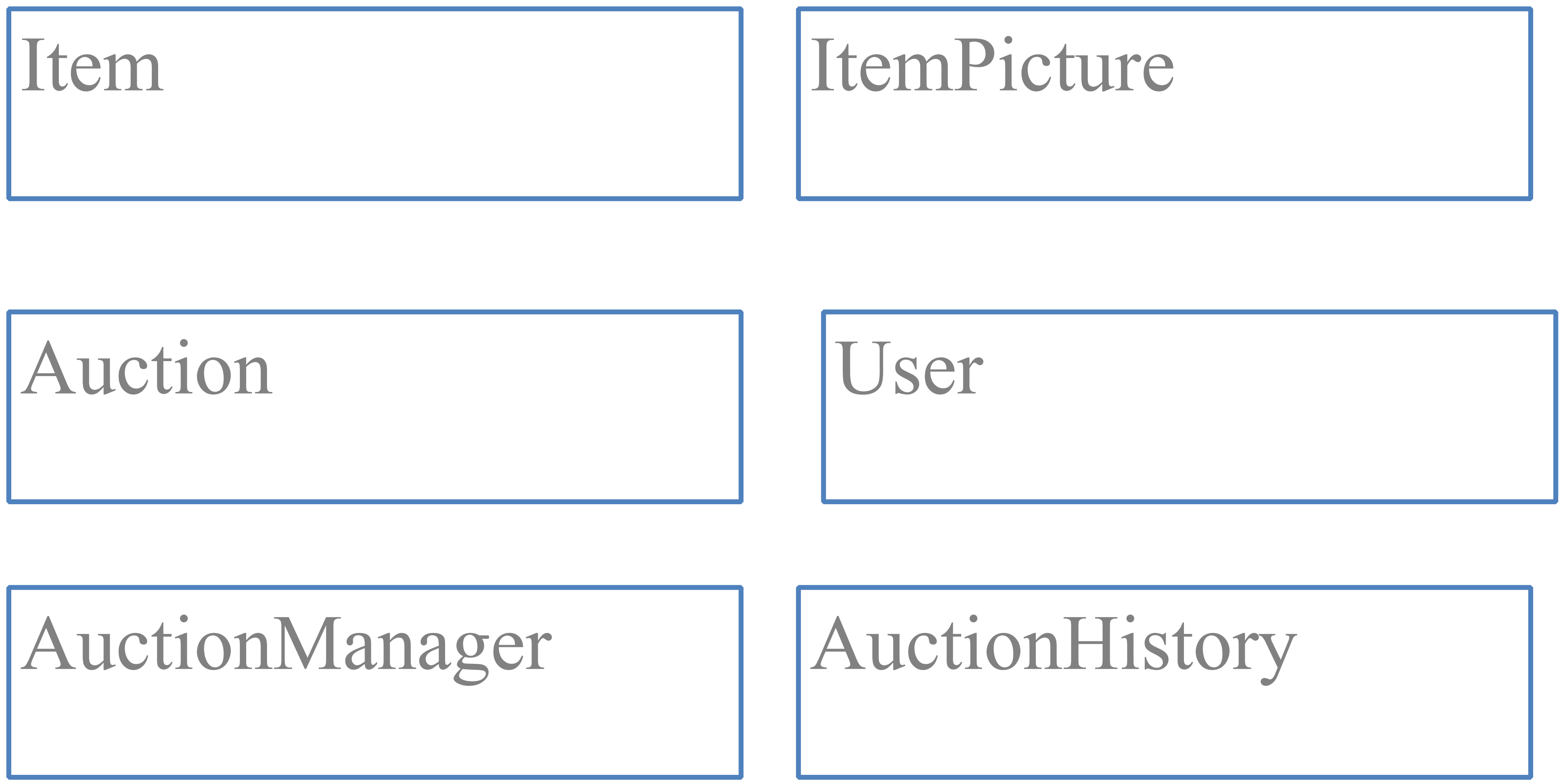
- -startAuction()
- -endAuction()
- +getItem()
- -computeMax()
- +displayMax()

User

- +placeBid()
- +setMickName()
- -setCardNo()
- +registerToAuction()
- +viewCurrentAuctions()

what are the services we need to test for integration?

Bottom Up or Top Down?



Test Driven Development

- Just an example to give you the flavor of it
- It is a development practice
- Tests are created before regular code
- Use of compiler and execution environment (e.g., JVM) to drive the development
- Use stubs and incrementally implement them
- Use drivers in assertions to test the development

- Practice for writing unit tests and production code concurrently and at a very fine level of granularity
- Programmers
  - first write a small portion of a unit test, and
  - then they write just enough production code to make that unit test compile and execute

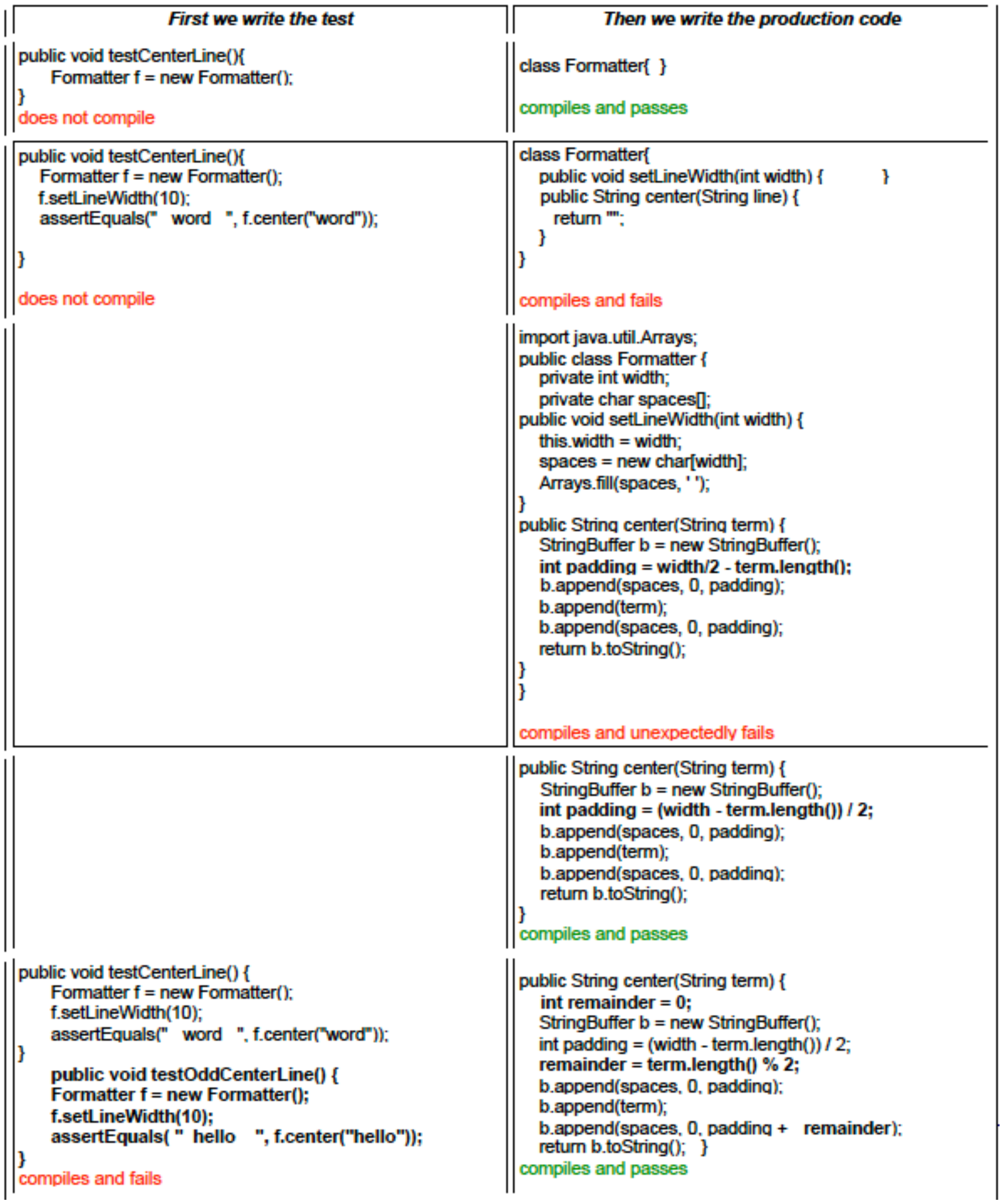
- This cycle lasts somewhere between 30 seconds and five minutes. Rarely does it grow to ten minutes.
- In each cycle, the tests come first.
- Once a unit test is done, the developer goes on to the next test until they run out of tests for the task they are currently working on

Example

- **TextFormatter:** A text formatter that take arbitrary strings and horizontally center them in a page
-

- Few issues:
  - What are the methods:
    - setLineWidth()
    - centerLine()
  - What is a Line?
  - Can I use String?

- First understand the entities to test



First we write the test

```
public void testCenterLine(){
      Formatter f = new Formatter();
}
```
does not compile

Then we write the production code

```
class Formatter{  }
```
compiles and passes

First we write the test

```
public void testCenterLine(){
    Formatter f = new Formatter();
    f.setLineWidth(10);
    assertEquals("   word   ", f.center("word"));

}
```
does not compile

Then we write the production code

```
class Formatter{
    public void setLineWidth(int width) {            }
    public String center(String line) {
      return "";
    }
}
```
compiles and fails

Then we write the production code

```
import java.util.Arrays;
public class Formatter {
    private int width;
    private char spaces[];
public void setLineWidth(int width) {
    this.width = width;
    spaces = new char[width];
    Arrays.fill(spaces, ' ');
}
public String center(String term) {
    StringBuffer b = new StringBuffer();
    int padding = width/2 - term.length();
    b.append(spaces, 0, padding);
    b.append(term);
    b.append(spaces, 0, padding);
    return b.toString();
}
}
```
compiles and unexpectedly fails

```
public String center(String term) {
    StringBuffer b = new StringBuffer();
    int padding = (width - term.length()) / 2;
    b.append(spaces, 0, padding);
    b.append(term);
    b.append(spaces, 0, padding);
    return b.toString();
}
```
compiles and passes

First we write the test

```
public void testCenterLine() {
      Formatter f = new Formatter();
      f.setLineWidth(10);
      assertEquals("   word   ", f.center("word"));
}

      public void testOddCenterLine() {
      Formatter f = new Formatter();
      f.setLineWidth(10);
      assertEquals( "  hello   ", f.center("hello"));
}
```
compiles and fails

Then we write the production code

```
public String center(String term) {
    int remainder = 0;
    StringBuffer b = new StringBuffer();
    int padding = (width - term.length()) / 2;
    remainder = term.length() % 2;
    b.append(spaces, 0, padding);
    b.append(term);
    b.append(spaces, 0, padding +   remainder);
    return b.toString();   }
```
compiles and passes

Exercise

- Extend the previous example by allowing any line length
- Extend the example above by allowing terms that are concatenation of word

What are the benefits of TDD?

- Line Test Coverage: If you follow the rules of TDD, then virtually 100% of the lines of code in your production program will be covered by unit tests
- This does not cover 100% of the paths through the code, but it does make sure that virtually every line is executed and tested

What are the benefits of TDD?

- Test Repeatability. The tests can be run any time you like
- Documentation. The tests describe your understanding of how the code should behave.
- They also describe the API. Therefore, the tests are a form of documentation.

What are the benefits of TDD?

- API Design. When you write tests first, you put yourself in the position of a user of your program’s API. This can only help you design that API better
- Reduced Debugging. When you move in the tiny little steps recommended by TDD, it is hardly ever necessary to use the debugger. Debugging time is reduced enormously

Regression Testing

Regression testing

- When to use it:
- Adding **new/changing module** impacts a system:
  - *New data flow paths established*
  - *New I/O may occur*
  - *New control logic invoked*

Regression testing

- It is *re-execution of a subset of tests* that have already been run after changes are made
- Ensures changes have not propagated **unintended side effects**

Major types

- **Procedural**: *Run the same tests again*
- **Risk-oriented**: Expose errors caused by change. *Test after changes*
- **Refactoring support**: Help the programmer discover implications of her code changes. *Change detectors*

Procedural - types of testing

- *Manual*, scripted regression testing
- *Automated GUI* regression testing
- *Smoke* testing (manual or automated)

Example - Smoke testing

- Retest with a relatively small test suite to decide *when a new build is stable* to proceed with further testing
- It aims at ensuring that *the most important functions work*
- Also known as Build Verification Testing

Example - not stable build

- When to use it:
- Some components are broken in obvious ways that suggest a *corrupt build* or
- Some *critical fixes* that are the primary intent of the new build *didn't work*

Risk-oriented - types of testing

- *Bug regression:*
- Retest **a specific bug** that has been supposed to be fixed: show that a bug fix didn't fix the bug
- *Old fix regression testing:*
- Retest **several old bugs** that were fixed, to see if they are back: show that old fixes were broken or had a side effect

Risk-oriented - types of testing

- *General functional regression*:
- Retest the product broadly:
  - To show that a change caused a working area to break

Refactoring support - types of testing

- Exercise every function in interesting ways, so that *when the code has been refactored*, the developer can quickly see:
  - What would break if developer made *a change to a given variable, data item or function* or
  - What the *developer did break* by making the change

connectors

Refactoring support - types of testing

- Test-driven development using white-box testing tools like:
  - JUnit, httpUnit, and fitnesse

- The developer creates tests and runs them every time the code is compiled (continuous testing)

Other types of testing

- *Conversion or port testing:*
- Retest the program when it has been **ported to a new platform** to determine whether the port was successful

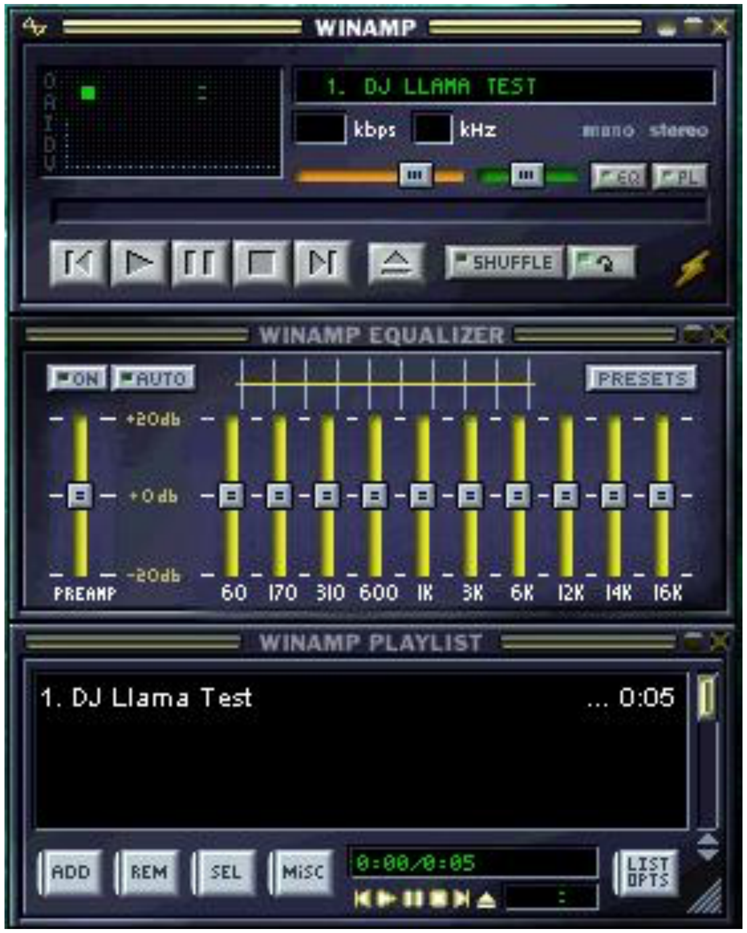
Other types of testing

- *Configuration testing:*
- Retest when the program is run with a new device or on a new version of the operating system or in conjunction with a new application

Other types of testing

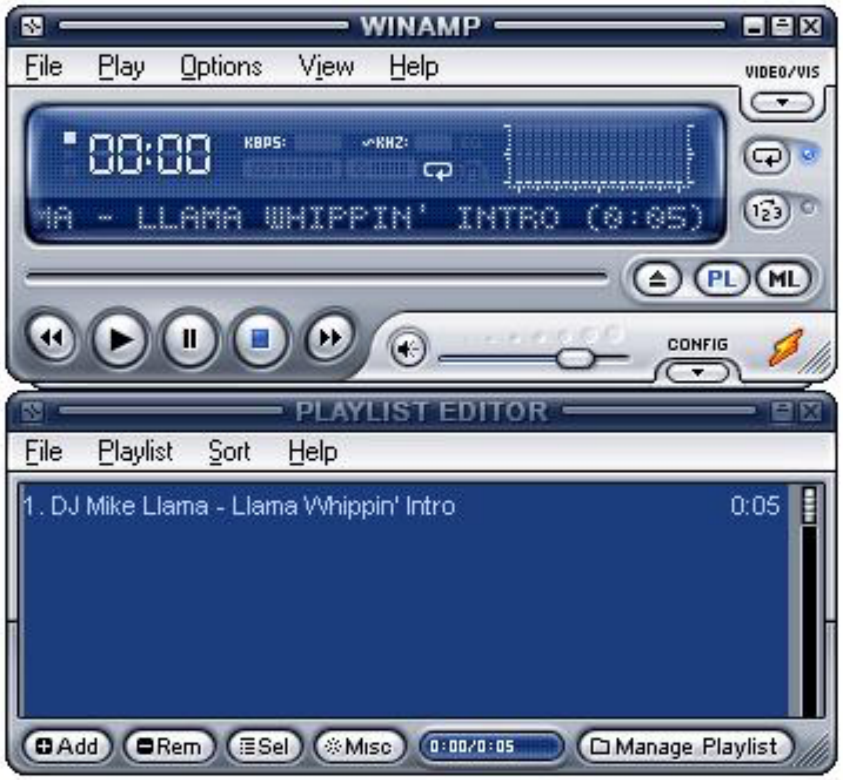
- *Localization testing*:
- Retest when the program has a new different language and/or following a different set of cultural rules

Example - Risk-oriented



1. DJ Llama Test ... 0:05

0:00/0:05



MA - LLAMA WHIPPIN' INTRO (0:05)

1. DJ Mike Llama - Llama Whippin' Intro 0:05

0:00/0:05

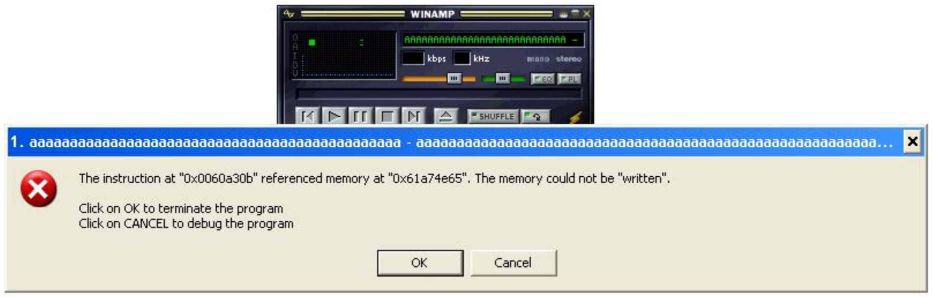
WinAmp v2.79

WinAmp v5.03

After replaying the old tests

- The player makes use of meta-data to store information about an MP3 file within the audio file itself like artist, title, track number
- Old bug:
  - Input: string with more than 30,000 characters
  - Output: (Warning window with) crash
- New Bug found with regression tests
  - Input: String with more than 30,000 characters
  - Output: The file plays, but one cannot hit any other buttons and pressing play multiple times, more and more duplicate menu options are added into a menu

Example - Risk-oriented



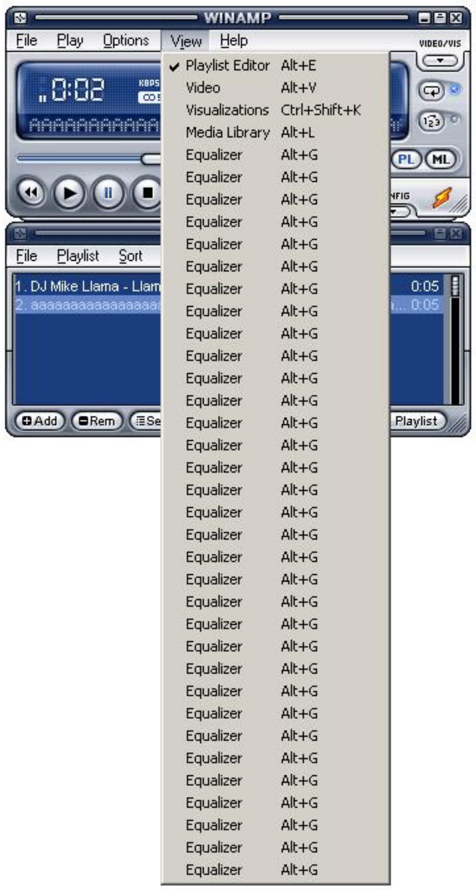
1. aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa - aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa...

The instruction at "0x0060a30b" referenced memory at "0x61a74e65". The memory could not be "written".

Click on OK to terminate the program
Click on CANCEL to debug the program

OK Cancel

WinAmp v2.79



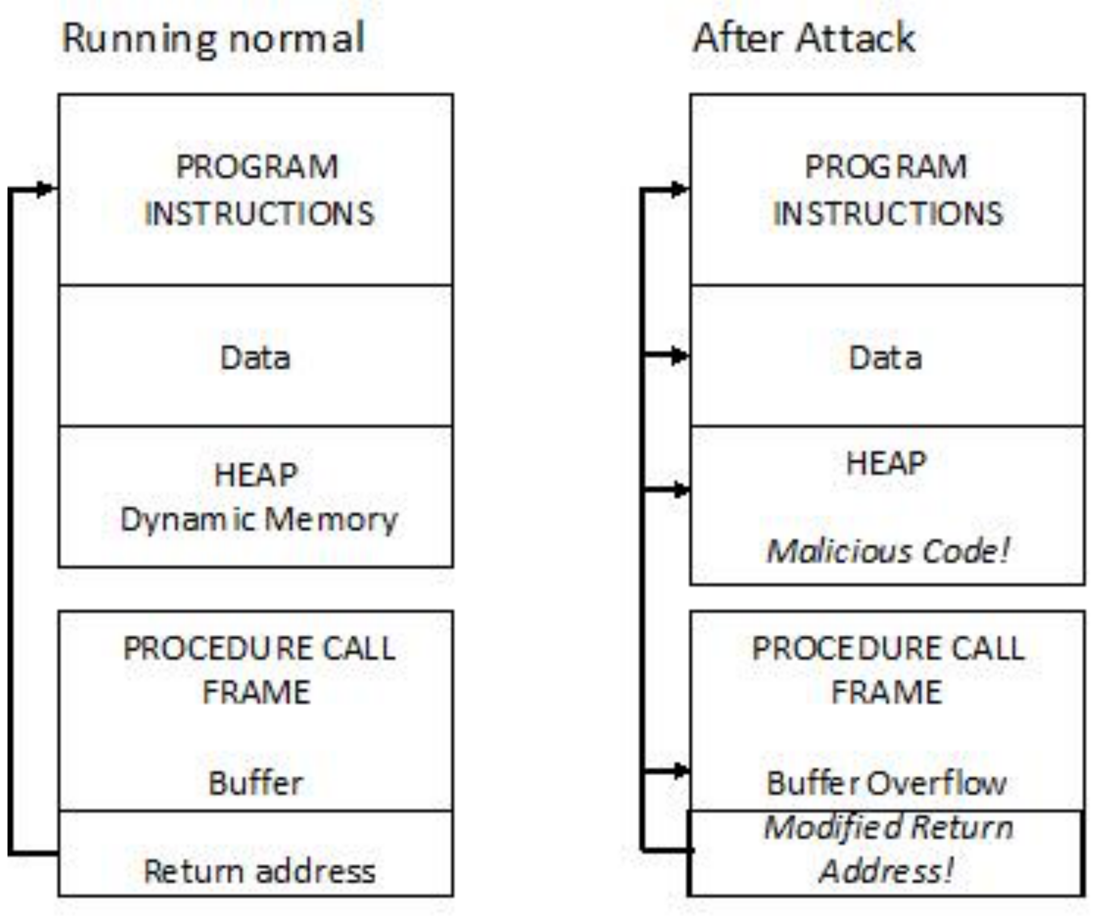
WinAmp v5.03

unibz Freie Universität Bozen
Libera Università di Bolzano
Università Liedia de Bulsan

Buffer overflow

- Buffer overflows occur when too much memory is used and not enough memory was allocated.
- Return calls for functions can be lost or overwritten, allowing malicious users to exploit them to access and modify other parts of a system

Buffer stack overflow



In Java this can happen with C++ libraries.
For instance, the parameters and local variables within JNI methods include several C++ object pointers

Attackers inject code that over flows buffer and corrupts the return address.

Instead returning to the appropriate call procedure the modified address points to malicious code located somewhere in the process memory

How would you change the code and re-test?

```
int foo (int a, int b, int c, int d, float e) {
    if (a == 0) {
        return 0;
    }
    int x = 0;
    if ( (a==b) || ( (c == d) && bug(a) ) ) {
        x=1;
    }
    e = 1/x;
  return e;
}
```

bug(a) = TRUE if !a==0 else FALSE

How would you change the code and re-test?

- Original test suite T(0,0,0,0,0) and T(1,1,0,0,0)

- Does the old suite capture the bug?

GUI Regression testing

- Test suite contains:
  - Representative sample of tests that exercises all GUI features
- Focus on GUI features that have been changed
- Focus on GUI features likely affected by change

Capture/Playback

- Capture/Playback tools:
  - 1. Capture test cases and results (manual)
  - 2. Playback and
  - 3. Compare
- A capture/replay tool is test execution tool in which the entries are recording during manual testing with the goal of generating automated test scripts that can be replayed afterwards
- Especially used for **GUI testing**

Regression testing strategy

- Create a test case
- Run it and inspect the output
- If the program fails, report a bug and try again later
- If the program passes the test, save the resulting outputs
- In future tests, run the program and compare the output to the saved results
- Report an exception whenever the current output and the saved output don’t match