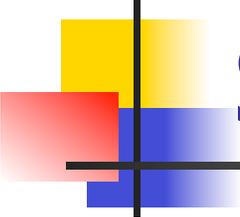


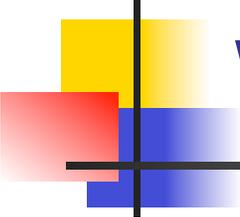
Automated GUI testing

How to test an interactive application automatically?



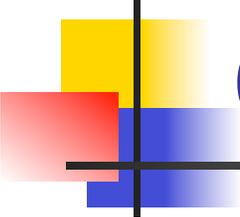
Some GUI facts

- Software testing accounts for 50-60% of total software development costs
- GUIs can constitute as much as 60% of the code of an application
- GUI development frameworks such as Swing make GUI development easier
- Unfortunately, they make GUI testing much harder



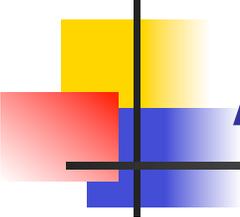
Why is GUI testing difficult?

- Event-driven architecture
 - User actions create events
 - An automatic test suite has to simulate these events somehow
- Large space of possibilities
 - The user may click on any pixel on the screen
 - Even the simplest components have a large number of attributes and methods
 - JButton has more than 50 attributes and 200 methods
 - The state of the GUI is a combination of the states of all of its components

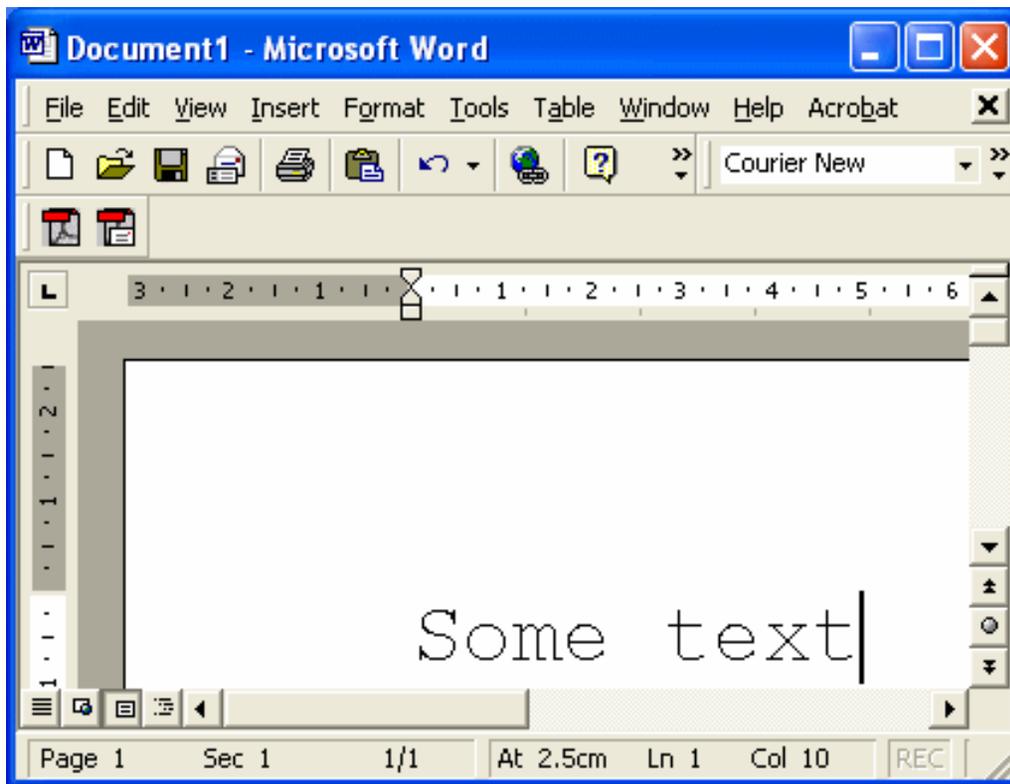


Challenges of GUI testing

- **Test case generation:** What combinations of user actions to try?
- **Oracles:** What is the expected GUI behaviour?
- **Coverage:** How much testing is enough?
- **Regression testing:** Can test cases from an earlier version be re-used?
- **Representation:** How to represent the GUI to handle all the above?



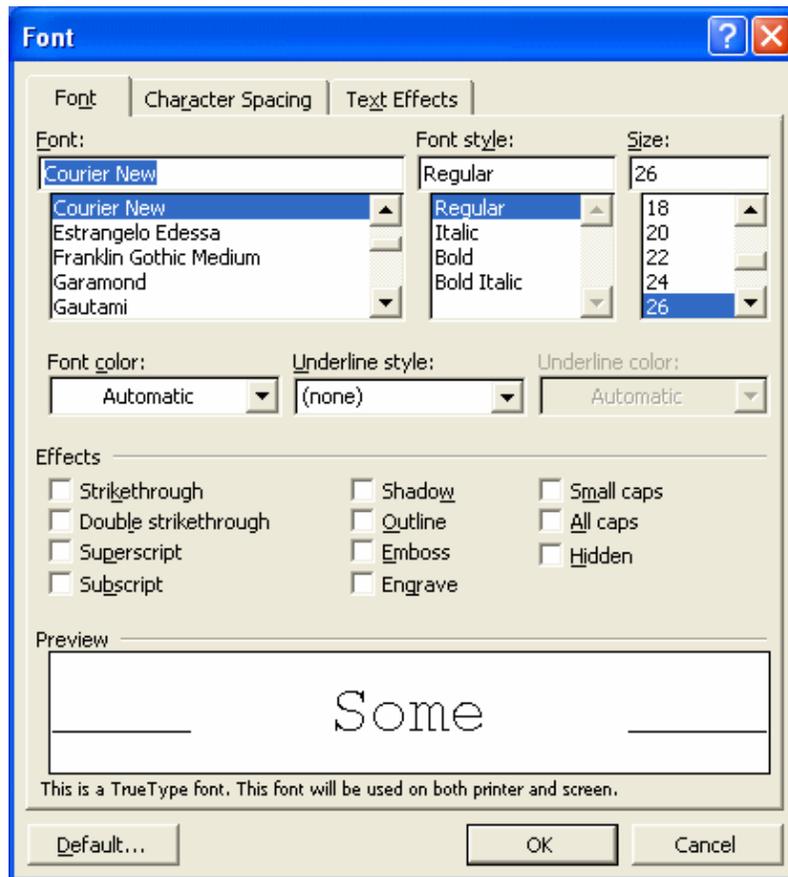
A GUI test case



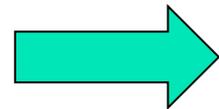
1. Select text "Some"
2. Menu "Format"
3. Option "Font"



A GUI Test Case

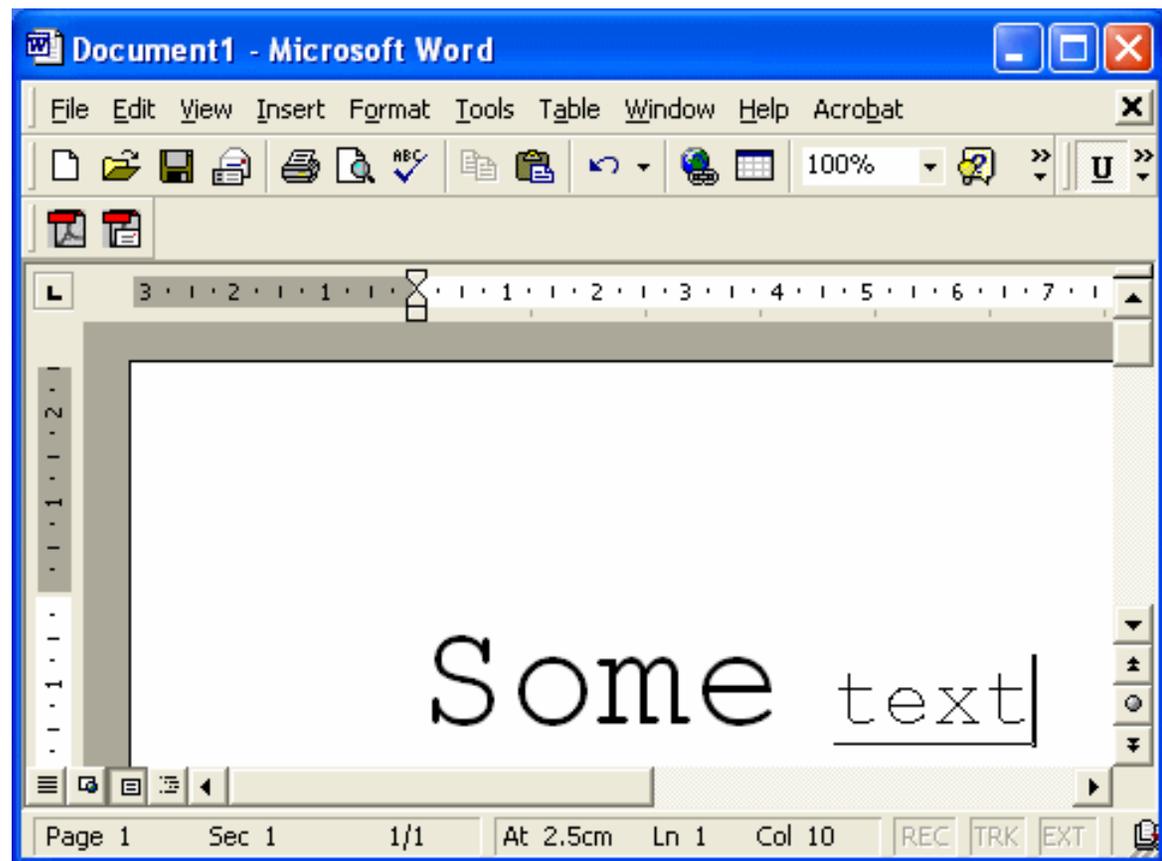


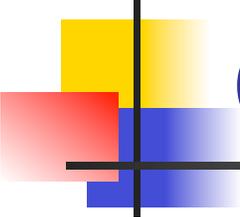
4. Combobox “Size”
5. Click on 26
6. Click OK



A GUI Test Case

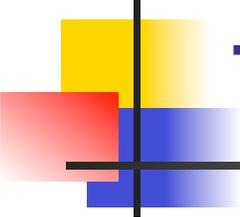
7. Select “text”
8. Click **U**
9. Verify that the output looks like this





GUI vs. business model testing

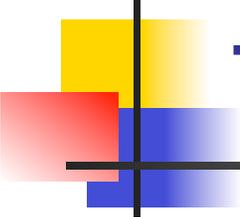
- GUI testing
 - The look of the text in the editor window corresponds to the operations performed
 - The U button is selected
 - All appropriate actions are still enabled, i.e. we can italicize the underlined text
- Business model testing
 - Word's internal model reflects the text formatting we performed



Two approaches to GUI testing

1. Black box

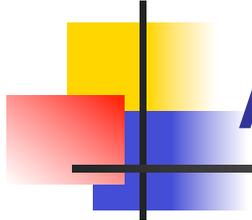
- Launch application
- Simulate mouse and keyboard events
- Compare final look to an existing screen dump
 - Very brittle test cases
 - Cannot test business model
 - Framework independent



Two approaches to GUI testing

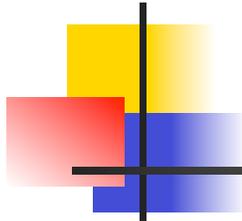
2. Glass box

- Launch application in the testing code
- Obtain references to the various components and send events to them
- Assert the state of components directly
 - Test cases harder to break
 - Business model can be tested
 - Framework dependent

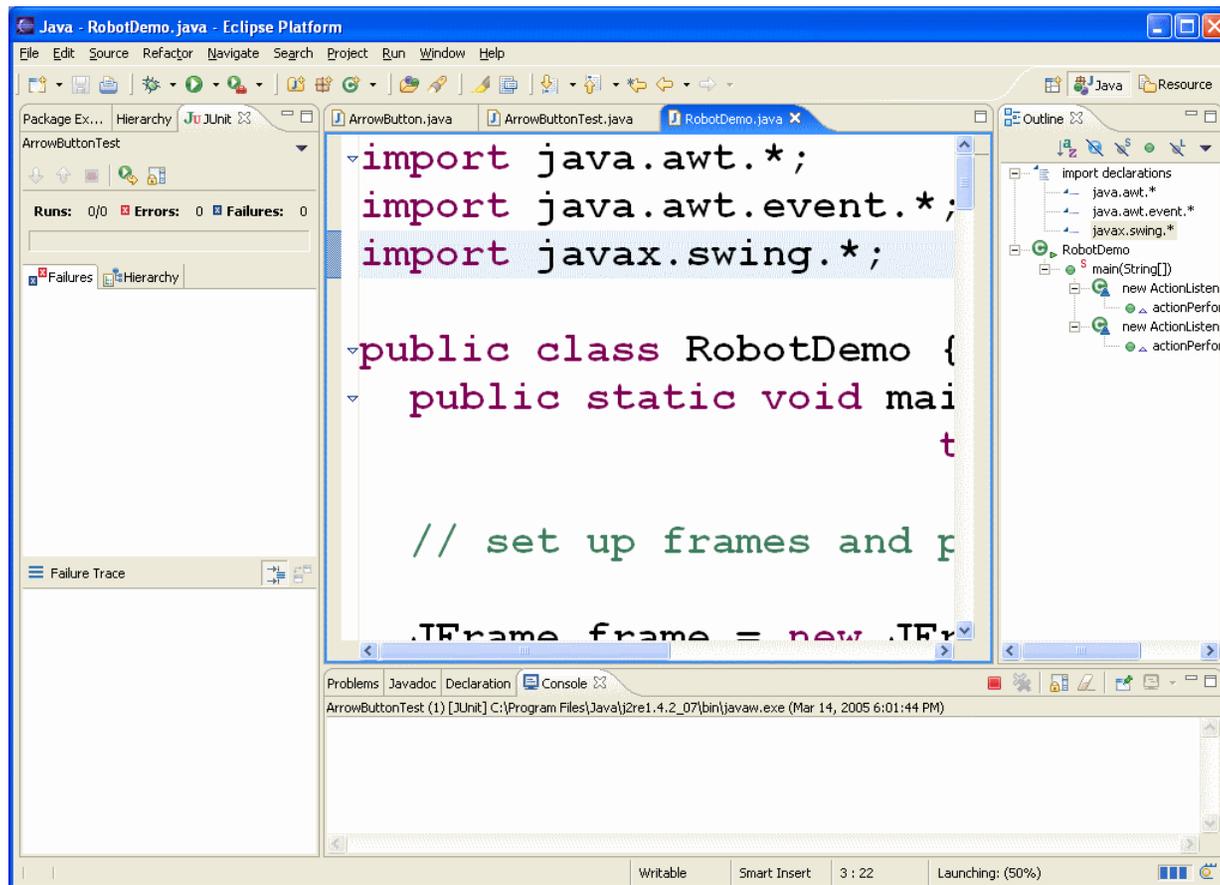


A first approach

- The Java API provides a class called `java.awt.Robot`
- It can be used to generate native system input events
 - Different than creating Event objects and adding them to the AWT event queue
 - These events will indeed move the mouse, click, etc.



RobotDemo

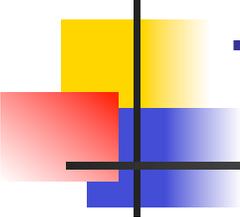


```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

public class RobotDemo {
    public static void main(
        t

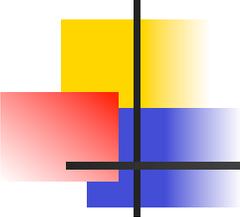
    // set up frames and p

    JFrame frame = new JFrame(...
```



Testing with Robot

- User input can be simulated by the robot
- How to evaluate that the correct GUI behaviour has taken place?
 - Robot includes method
public BufferedImage
createScreenCapture(Rectangle screenRe
ct)
 - Creates an image containing pixels read from the screen



Problems with this approach

- Low-level

- Would rather say “Select "blue" from the colour list” than

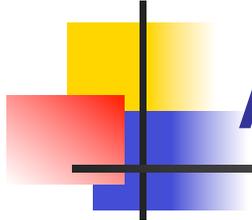
```
Move to the colour list  
co-ordinates
```

```
Click
```

```
Press ↓ 5 times
```

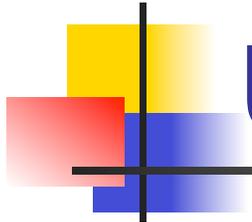
```
Click
```

- Brittle test cases (regression impossible)



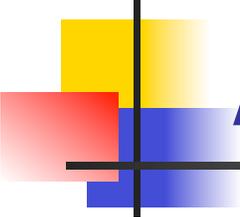
A better approach

- Every GUI component should provide a public API which can be invoked in the same manner via a system user event or programmatically
- Component behaviour should be separated from event handling code
- For example, class JButton contains the `doClick()` method



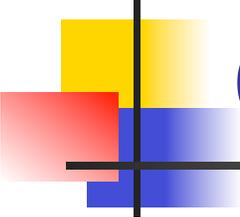
Unfortunately...

- Most GUI development frameworks are not designed in this fashion
- In Swing, event handling is mixed with complex component behaviour in the Look and Feel code
- Few components offer methods such as `doClick()`



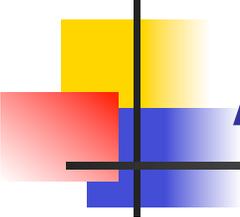
Abbot – A Better ' Bot

- A GUI testing framework for Swing
- Works seamlessly with Junit
 - Uses some Junit 3 features
- Can be used to create
 - Unit tests for GUI components
 - Functional tests for existing GUI apps
- Open source
 - <http://abbot.sourceforge.net/>



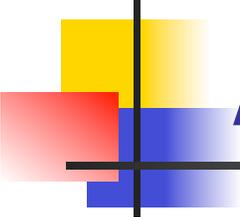
Goals of the Abbot framework

- Reliable reproduction of user input
- High-level semantic actions
- Scripted control of actions
- Loose component bindings



Abbot overview

- A better Robot class is provided
 - `abbot.testers.Robot` includes events to click, drag, type on any component
- For each Swing widget a corresponding Tester class is provided
 - E.g. `JPopupMenuTester` provides a method called `getMenuLabels()`
- Components can be retrieved from the component hierarchy
 - No direct reference to any widget is necessary



A typical test case

```
 JButton button = (JButton) getFinder().find(
    new Matcher() {
        public boolean matches(Component c) {
            return c instanceof JButton &&
                ((JButton) c).getText().equals("OK");
        }
    });
AbstractButtonTester tester =
    new AbstractButtonTester();
Tester.actionClick(button);
assertEquals("Wrong button tooltip",
    "Click to accept", button.getToolTipText());
```

Testing with Abbot demo

```
package example;

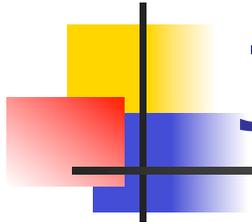
import java.awt.event.*;

public class ArrowButton
    extends ComponentTest
    // ComponentTestFixture

    private ComponentTest
    protected void setUp()
        tester = ComponentTest
```

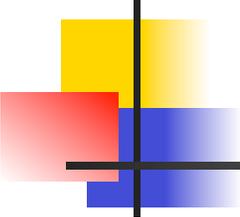
The screenshot shows the Eclipse IDE with the following details:

- Title Bar:** Java - ArrowButtonTest.java - Eclipse Platform
- Menu Bar:** File, Edit, Source, Refactor, Navigate, Search, Project, Run, Window, Help
- Toolbar:** Standard Eclipse development icons.
- Package Explorer:** Shows the package structure for ArrowButtonTest.
- Outline View:** Displays the class structure of ArrowButtonTest, including methods like setUp(), gotClick(), testClick(), and testRepeatedFire().
- Code Editor:** Contains the Java source code for ArrowButtonTest.java.
- Console:** Shows the output of the JUnit test run: ArrowButtonTest (1) [JUnit] C:\Program Files\Java\j2re1.4.2_07\bin\javaw.exe (Mar 14, 2005 7:55:06 PM)
- Status Bar:** Writable, Smart Insert, 86 : 1



JUnit 3 features

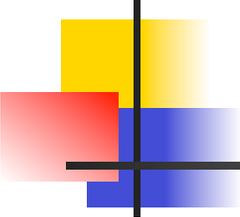
- Abbot requires JUnit 3
- Only the differences between JUnit 3 and JUnit 4 are presented in the next slides
- The JUnit 3 jar file is included in the abbot distribution



Extending TestCase

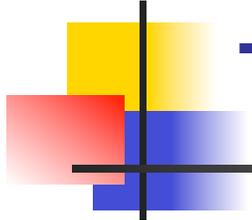
- Each test class needs to extend class `junit.framework.TestCase`

```
public class SomeClassTest
    extends junit.framework.TestCase {
        ...
    }
```



Naming vs. Annotations

- `protected void setUp()`
 - The `@Before` method must have this signature
- `protected void tearDown()`
 - The `@After` method must have this signature
- `public void testAdd()`
`public void testToString()`
 - All `@Test` methods must have names that start with `test`
- Do not include any annotations



Test suite creation

- Creating a test suite with JUnit 3 is also different
- Use the code in the next slide as a template

```
import junit.framework.*;

public class AllTests {

    public static void main(String[] args) {
        junit.swingui.TestRunner.run(AllTests.class);
    }

    public static Test suite() {
        TestSuite suite = new TestSuite("Name");
        suite.addTestSuite(TestClass1.class);
        suite.addTestSuite(TestClass2.class);
        return suite;
    }
}
```