

Testing of Scout Application

Ludwigsburg, 27.10.2014



The Tools approach...

JUnit



Squish®

The Testing Theory approach...

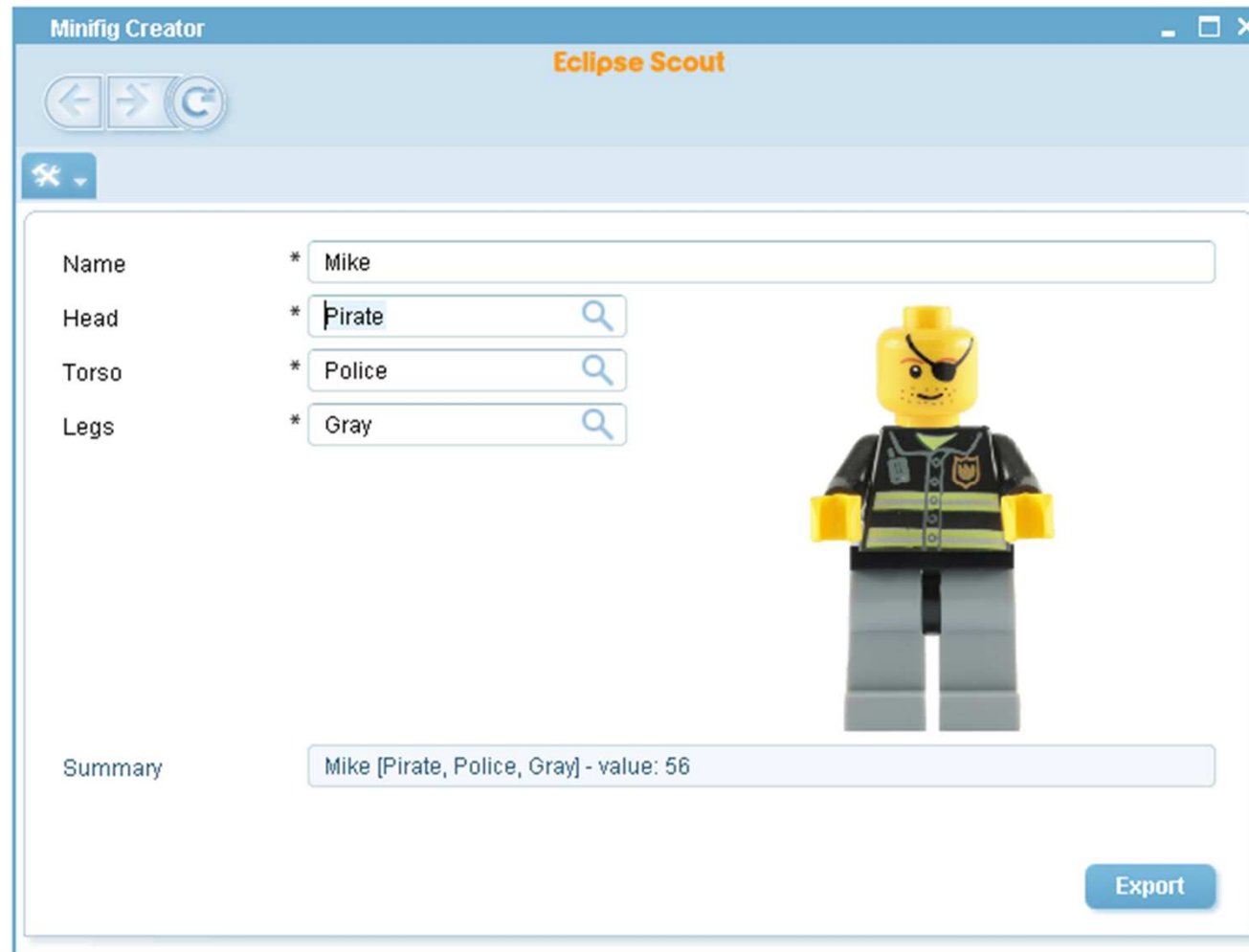
- Unit testing
- White box testing
- Black box testing

- Integration testing
- Functional testing
- System testing
- End-to-end testing

«What is your goal?»

Application under test

The application under test



Requirements (1)

Modification
of the input
fields...

Name: Mike

Head: Bart

Torso: Police

Legs: Odd

Summary: Mike [Bart, Police, Odd] - value: 68

Export

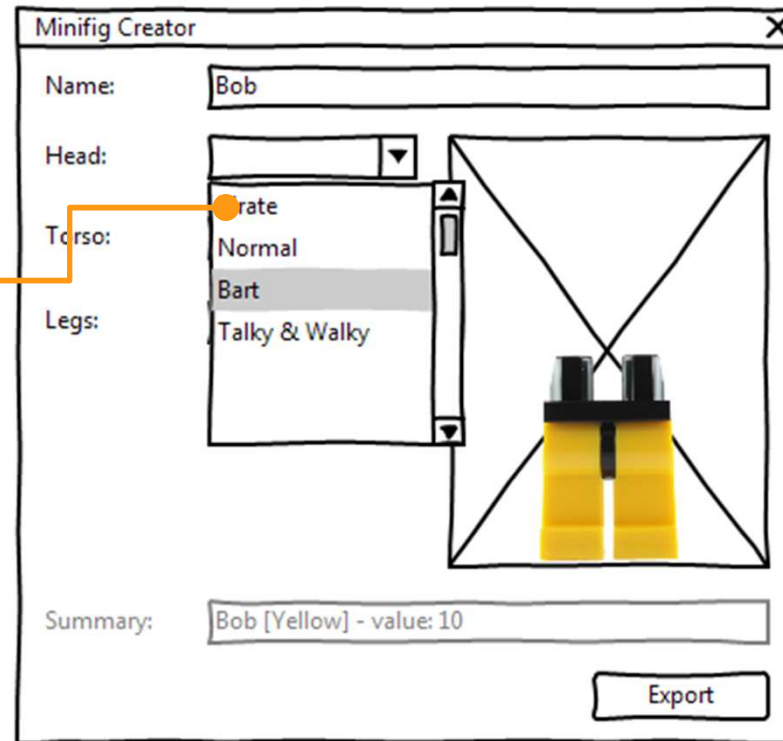
... will update
the image and
summary fields

Specific format: <name> [<parts>] value: <value>

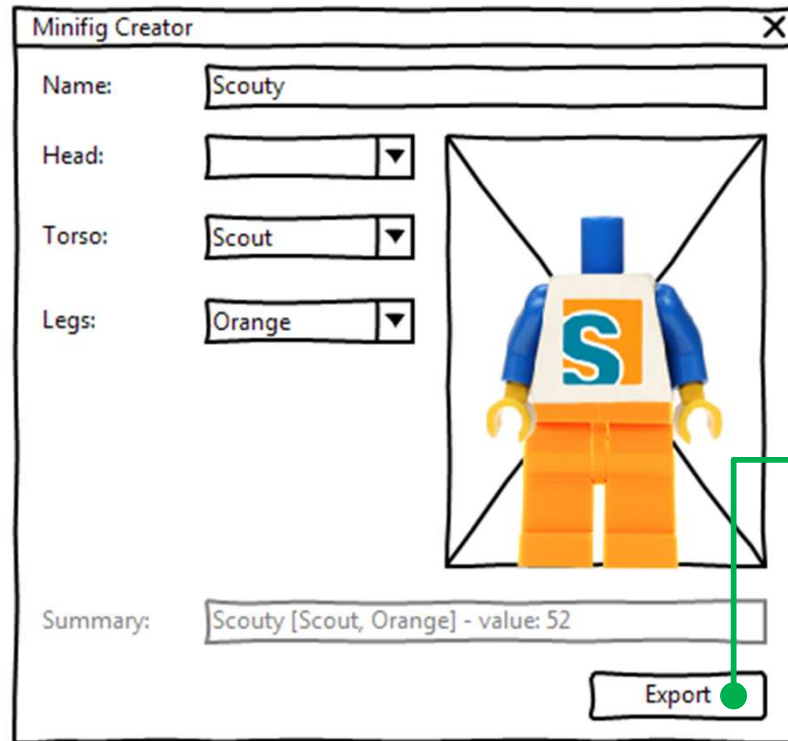
Requirements (2)

Only the available parts are listed in the field

Is only a part available, the field is disabled



Requirements (3)



The screenshot shows a window titled "Minifig Creator" with a close button (X) in the top right corner. The window contains the following elements:

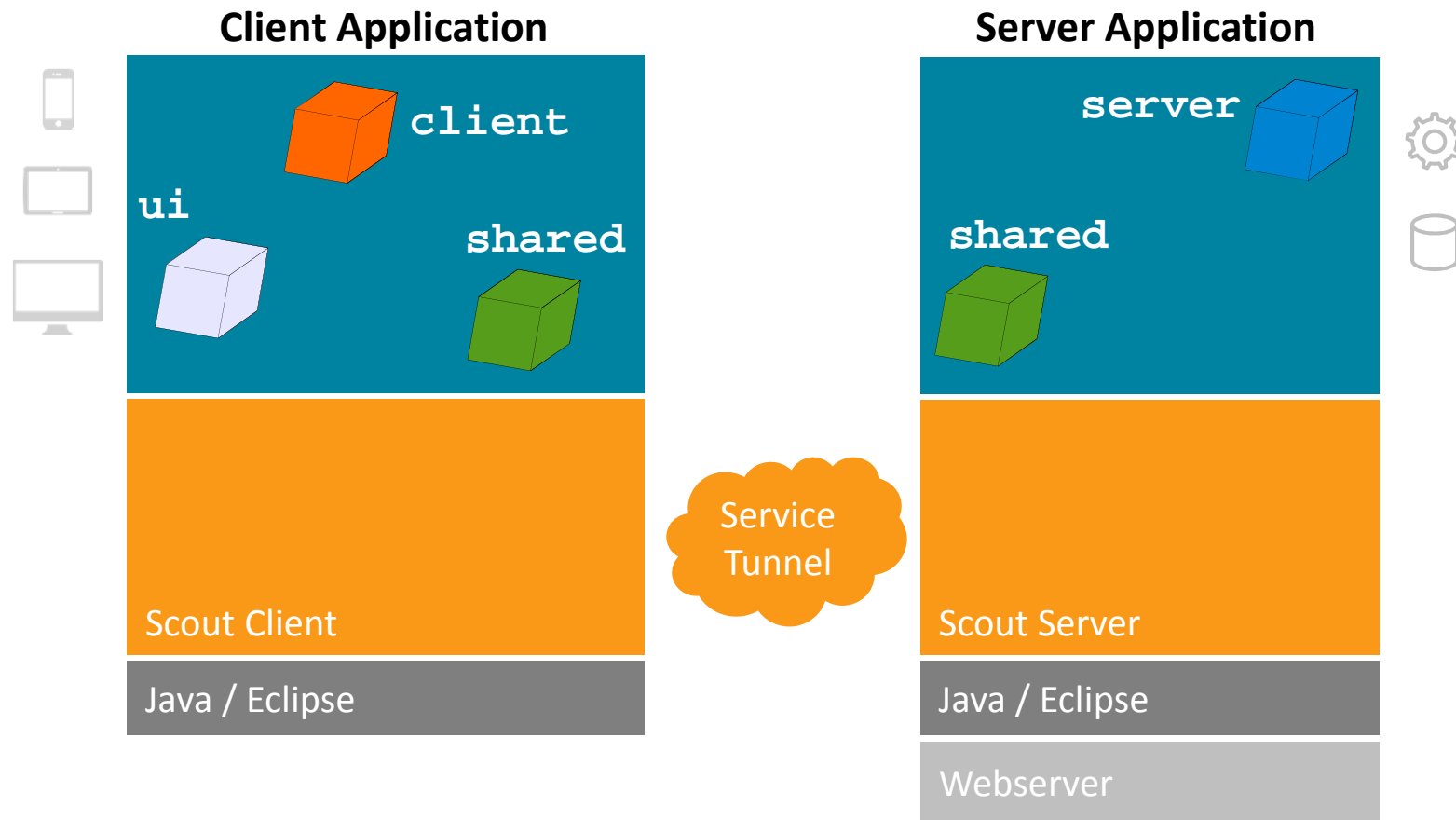
- Name:** A text input field containing "Scouty".
- Head:** A dropdown menu with a downward arrow.
- Torso:** A dropdown menu containing "Scout".
- Legs:** A dropdown menu containing "Orange".
- Preview:** A central area showing a 3D model of a LEGO minifig with a blue head, blue arms, a white torso with a blue and orange "S" logo, and orange legs.
- Summary:** A text input field containing "Scouty [Scout, Orange] - value: 52".
- Export:** A button with a green dot on its right side.

Export button:

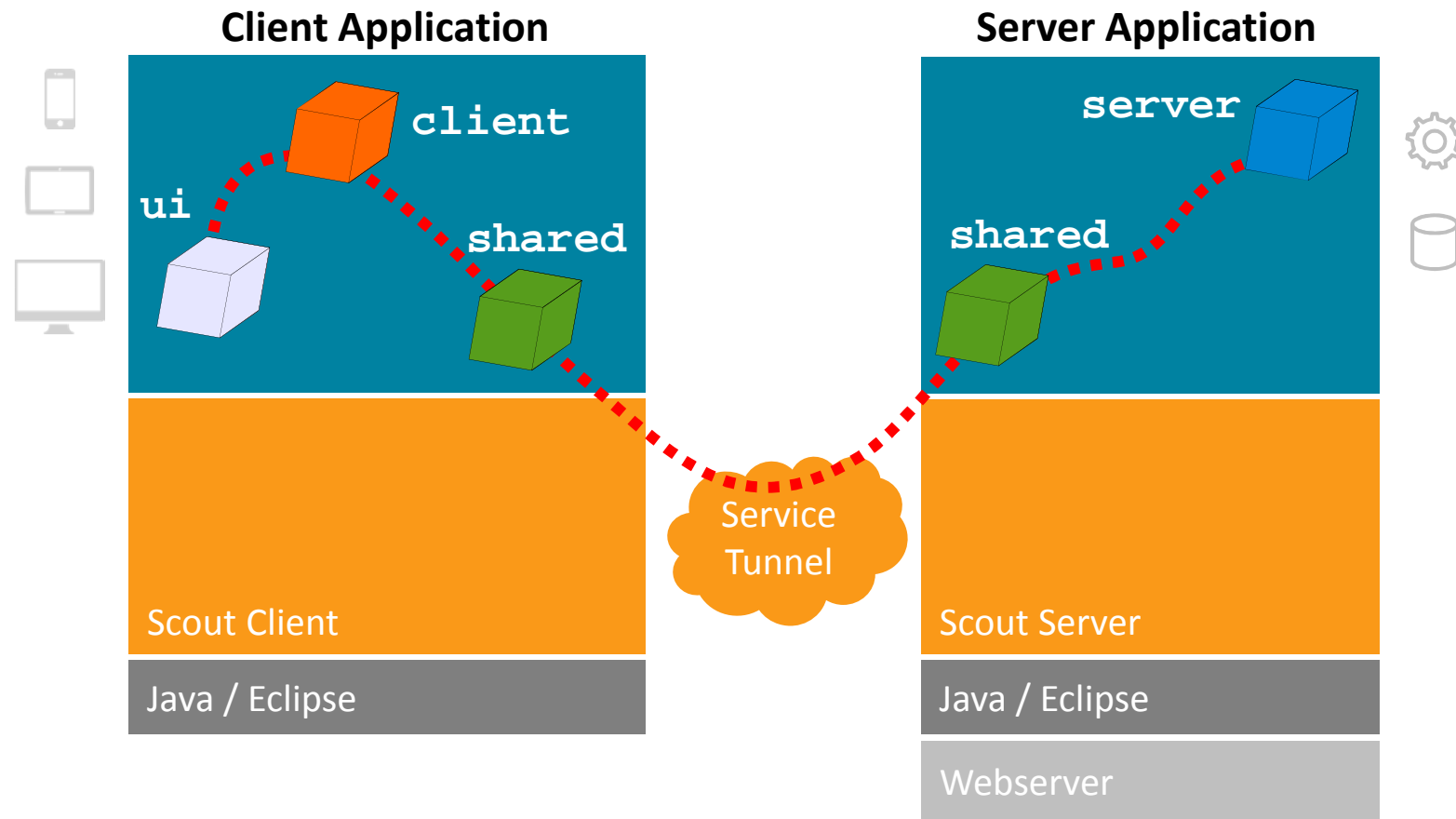
- validate the form
- register the minifig in the server
- reset the form

Scout architecture

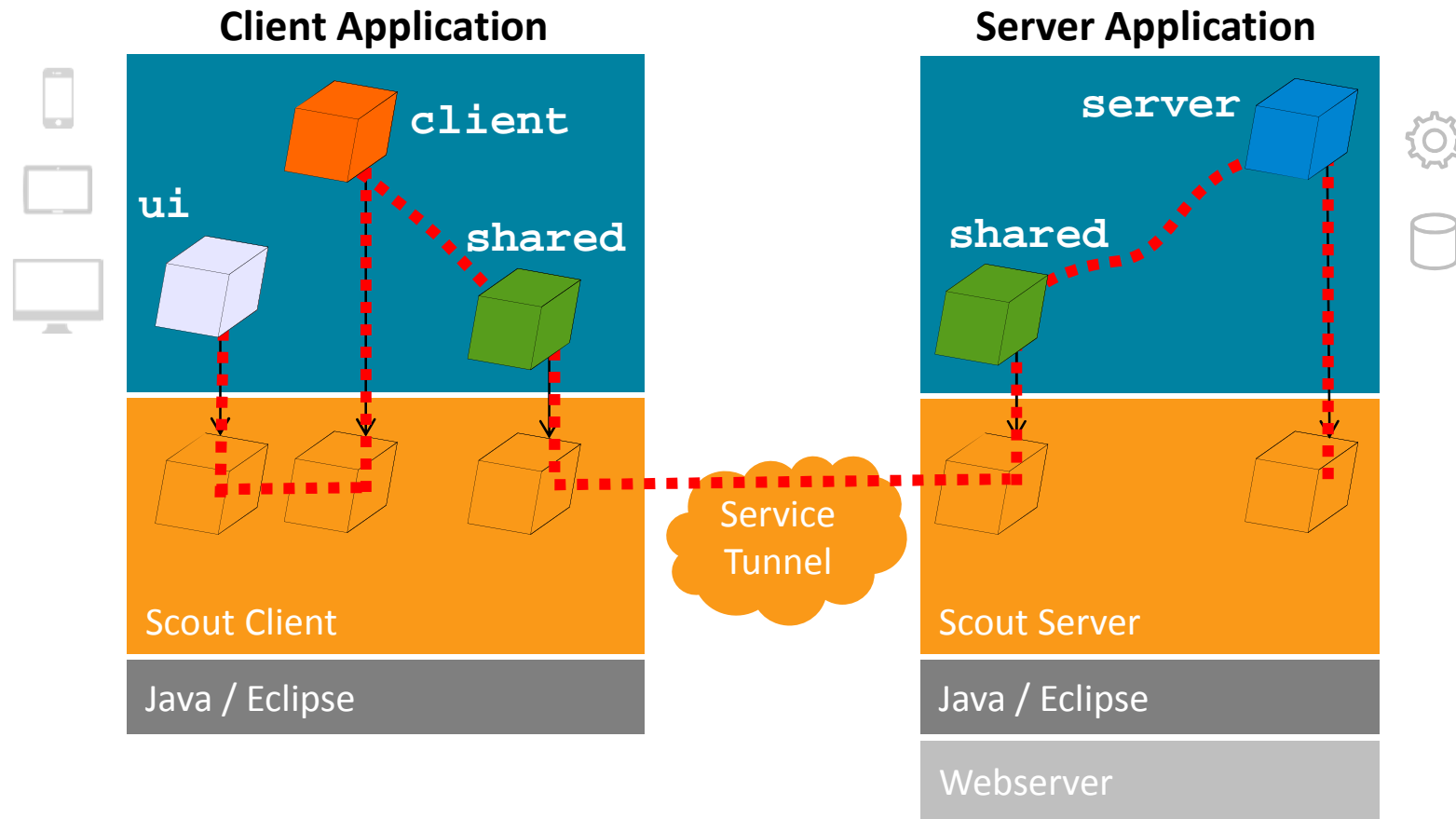
A Scout application



A Scout application

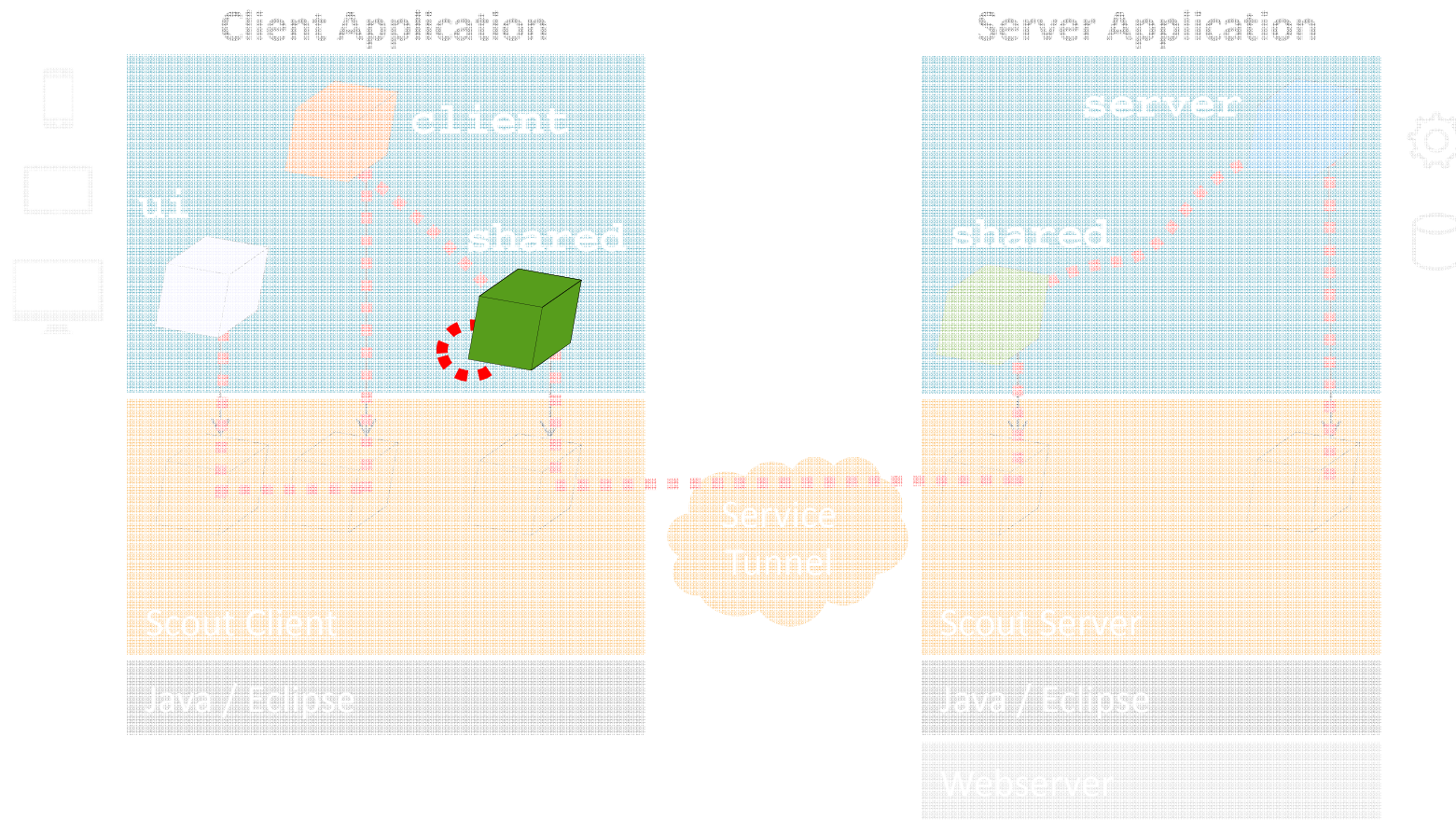


A Scout application

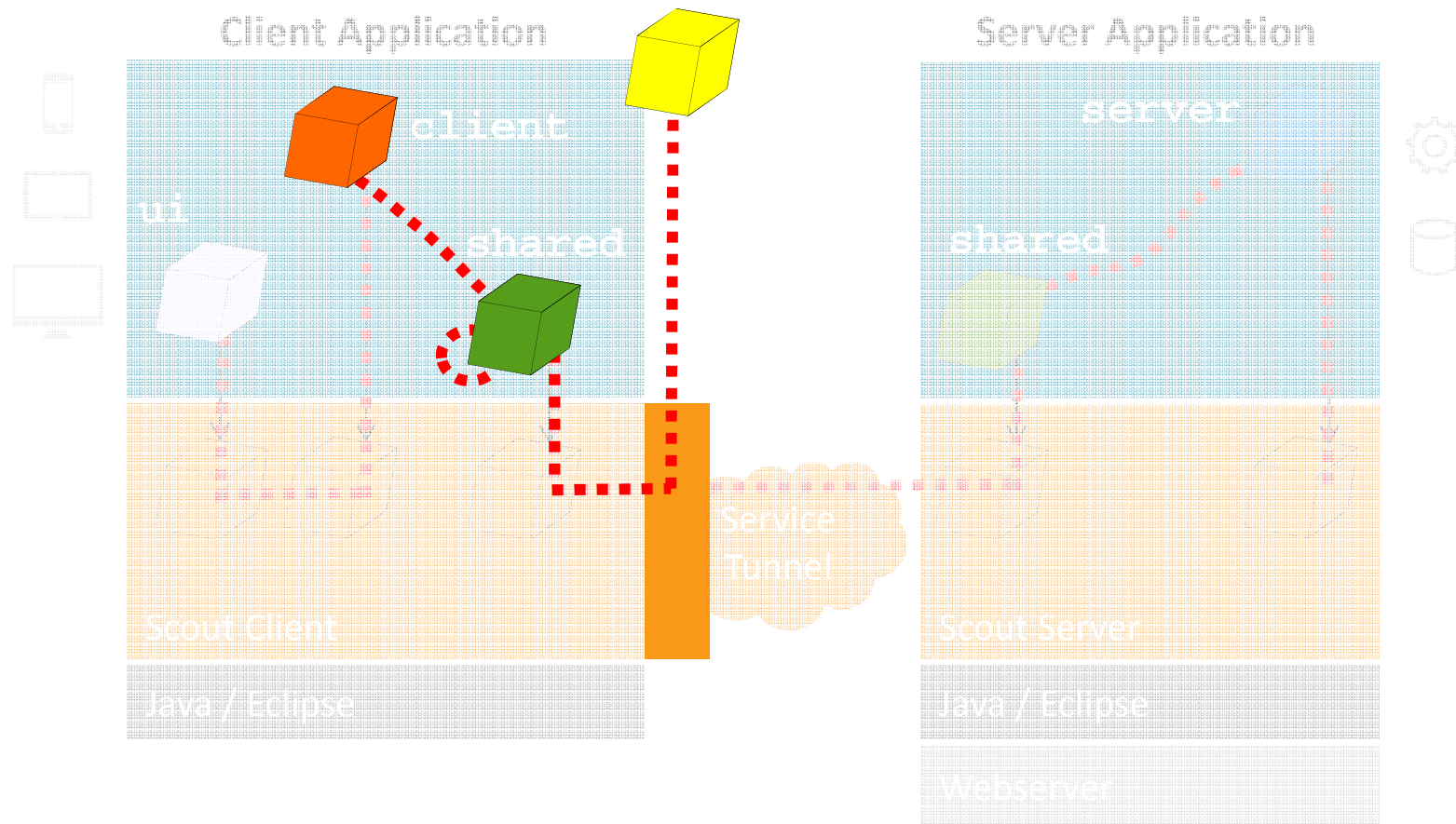


Unit testing

Test for logic in the shared plugin



Test with the Scout services



@RunWith annotation

Annotate the test class with the Annotation:

```
@RunWith(ScoutClientTestRunner.class)
public class DesktopFormTest {

    // ...

}
```

It adds:

- Equinox OSGi Runtime
- Scout Context, Services, ...

Mock remote Services



→ Create the mock

```
private IDesktopProcessService m_mockService =  
    Mockito.mock(IDesktopProcessService.class);
```

→ Define the behavior for your tests:

```
Mockito  
    .when(m_mockService.load(  
        Mockito.any(DesktopFormData.class)))  
    .thenReturn(someFormData);
```

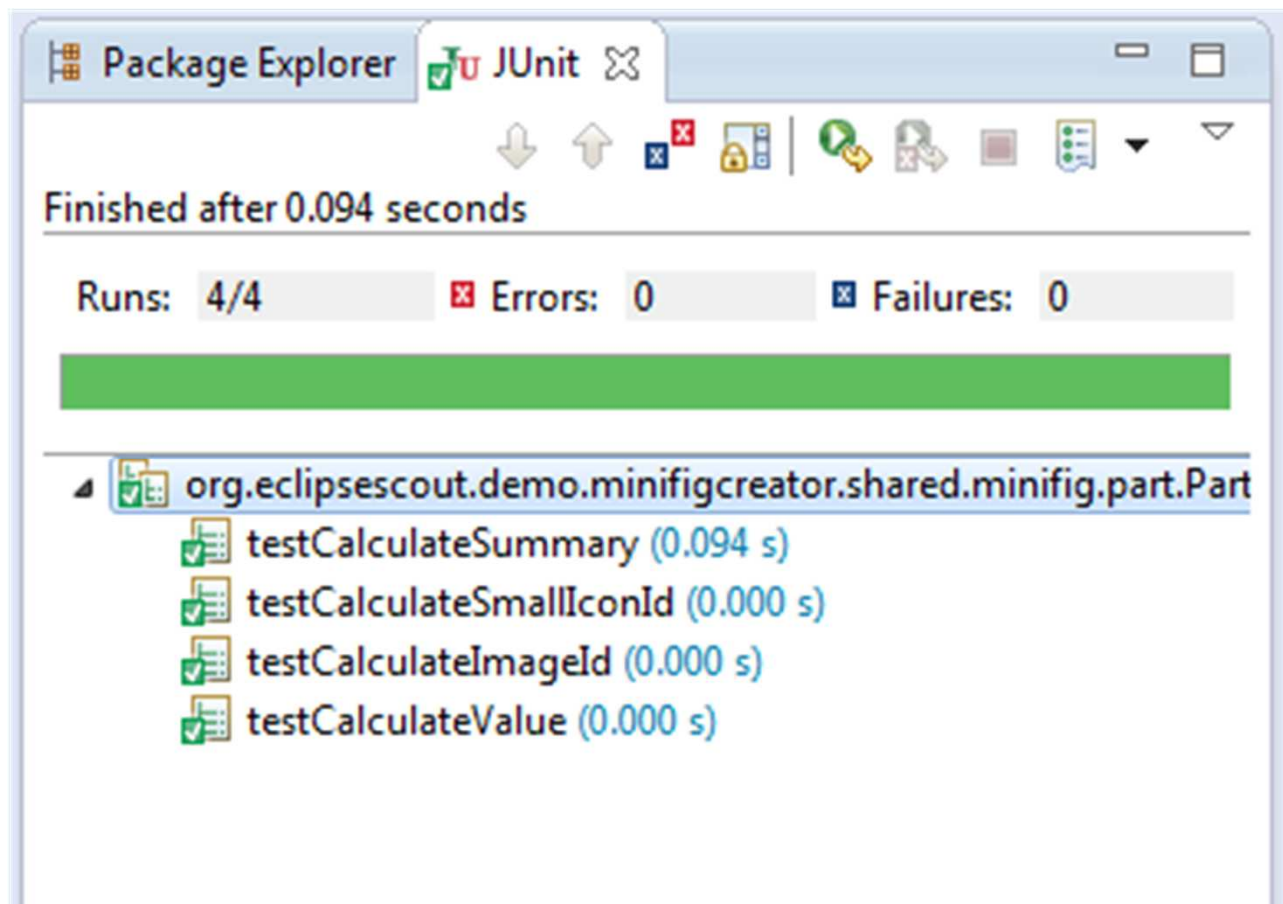
TestingUtility.registerServices(..)

→ Dynamically register your mocked service:

```
@Before
public void setUp() {
    m_registeredServices = TestingUtility.registerServices(
        Activator.getDefault().getBundle(), 1000, m_mockService);
}
```

```
@After
public void tearDown() {
    TestingUtility.unregisterServices(m_registeredServices);
}
```

Demo

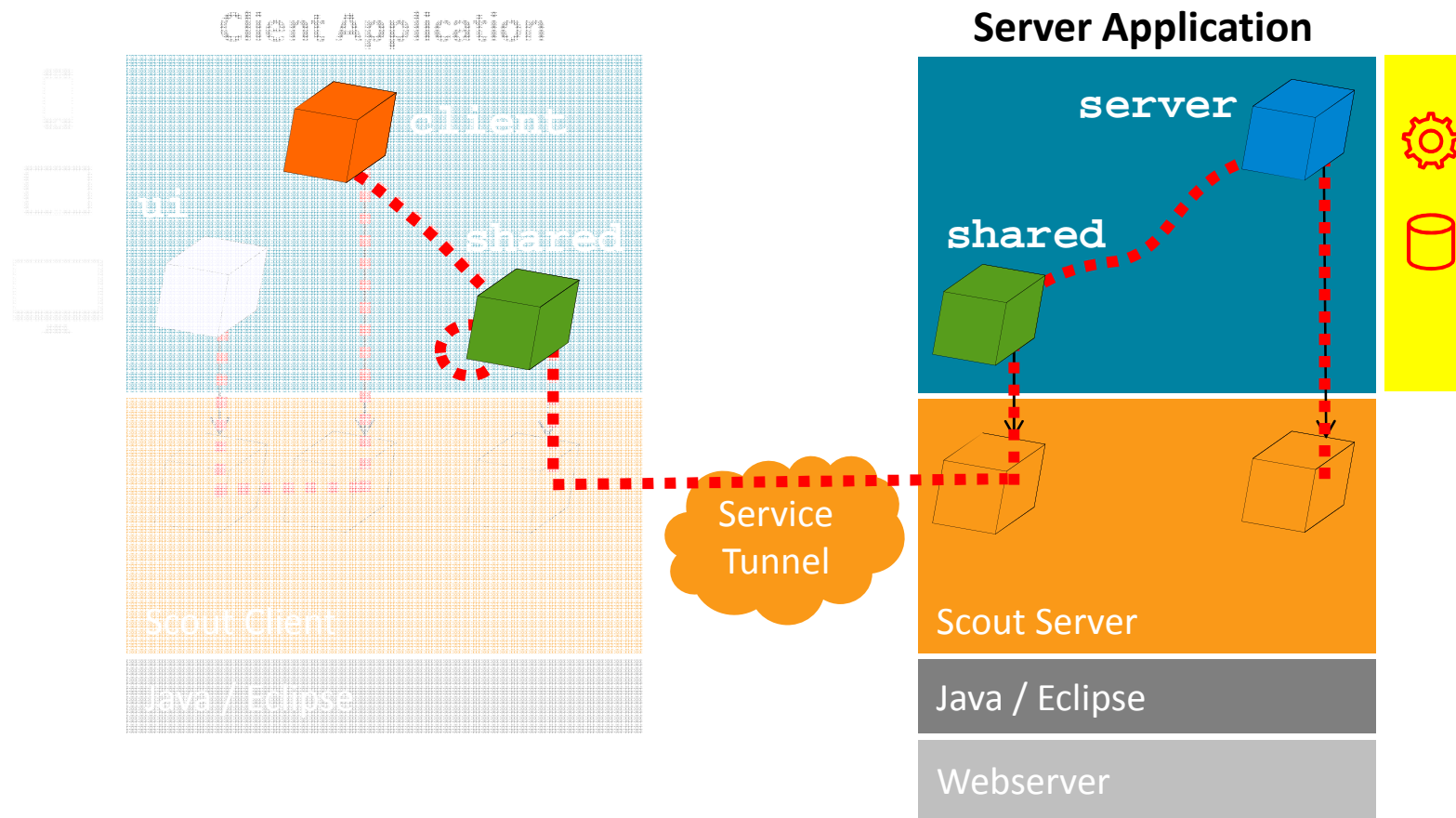


The screenshot shows the Eclipse IDE's JUnit test runner interface. At the top, the window title is "JUnit" with a close button. Below the title bar is a toolbar with various icons for test execution and management. The main area displays the test results for the package `org.eclipse.scout.demo.minifigcreator.shared.minifig.part.Part`. The test run is complete, having finished after 0.094 seconds. The summary shows 4/4 runs, 0 errors, and 0 failures. A green progress bar indicates that all tests passed. The list of tests includes:

- `testCalculateSummary` (0.094 s)
- `testCalculateSmallIconId` (0.000 s)
- `testCalculateImageId` (0.000 s)
- `testCalculateValue` (0.000 s)

Integration tests

Integration tests

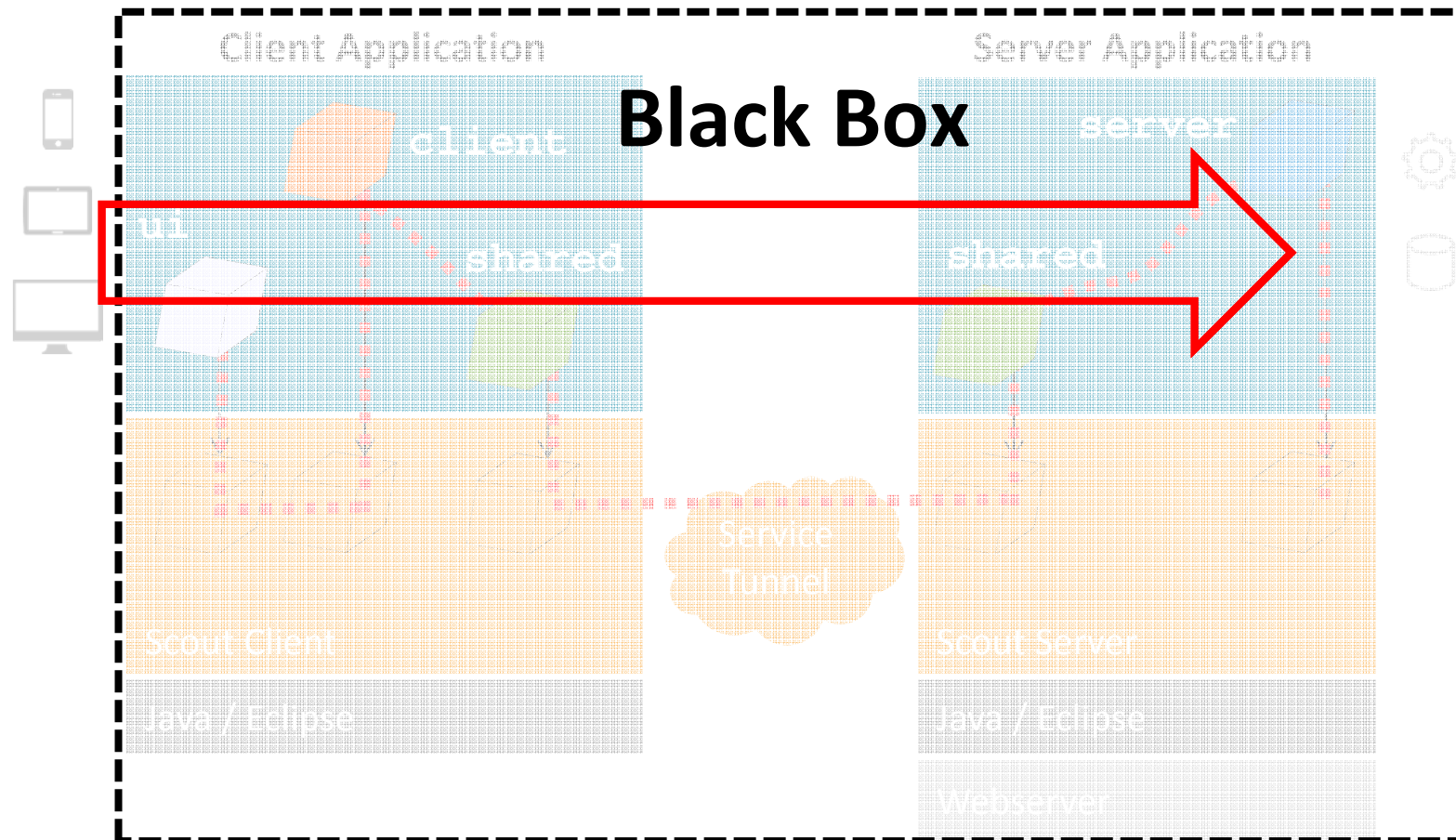


Integration tests: Example setup

- Deploy the server:
 - In a managed environment (database, external services...)
 - As near as possible from the productive environment
- Start an head-less client:
 - Browse through the data (outline, pages)
 - Open some forms
- Depending on how-much effort you want to put in the client, it is possible to write one generic test for all pages and forms

Automated user tests

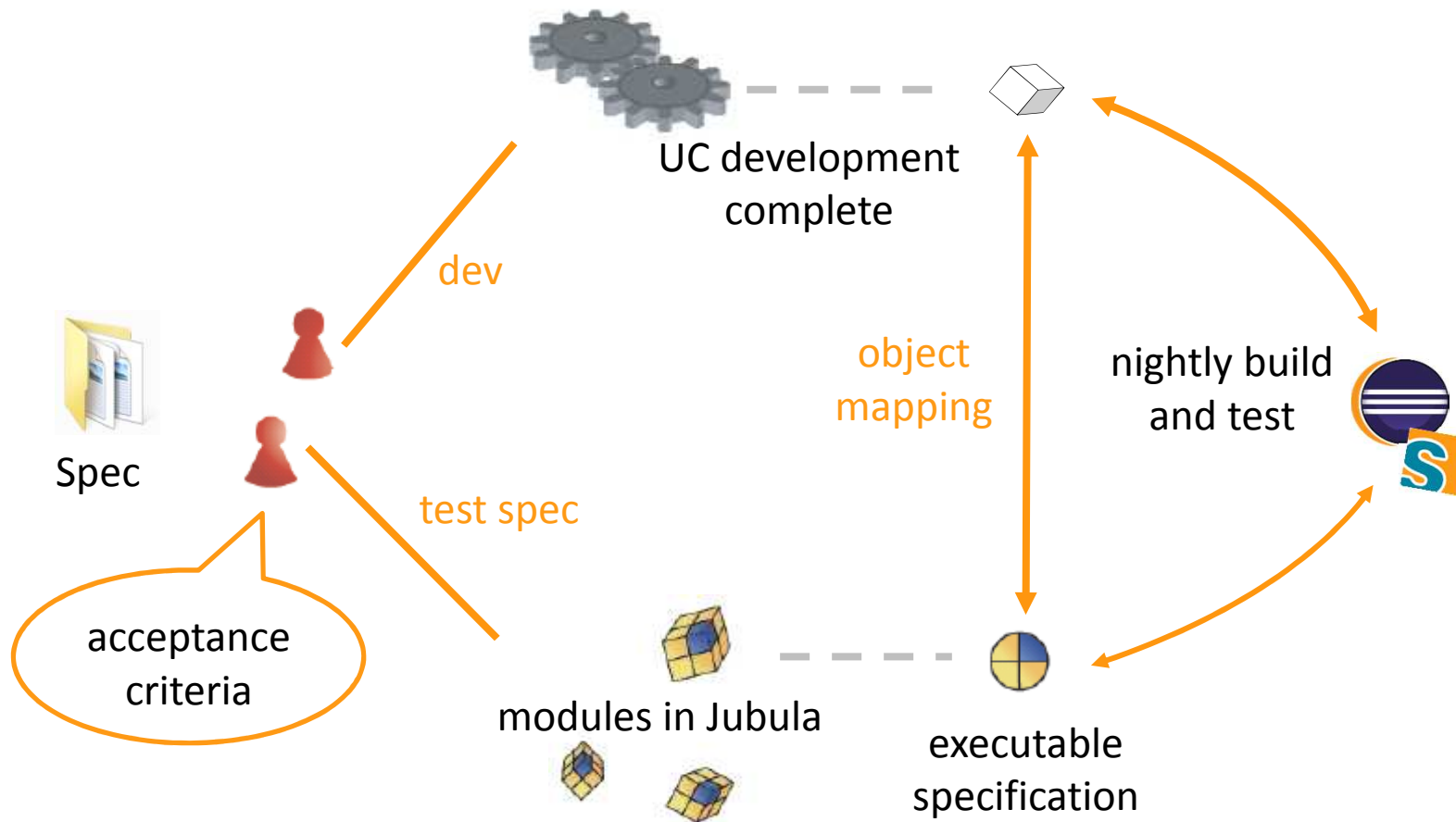
Test with Jubula



Jubula

- As a user would work – passing through all layers
- Test creation, execution, analysis
- Drag and drop test creation:
 - No recording
 - No programming
 - Very similar to development code
- Constant feedback about quality
 - Acceptance testing
 - Regression testing

Workflow



Using the specification to automate tests

module

Replace text

Select from smart field

Select from smart field

Select from smart field

Check text

Minifig Creator

Name: Mike

Head: Bart

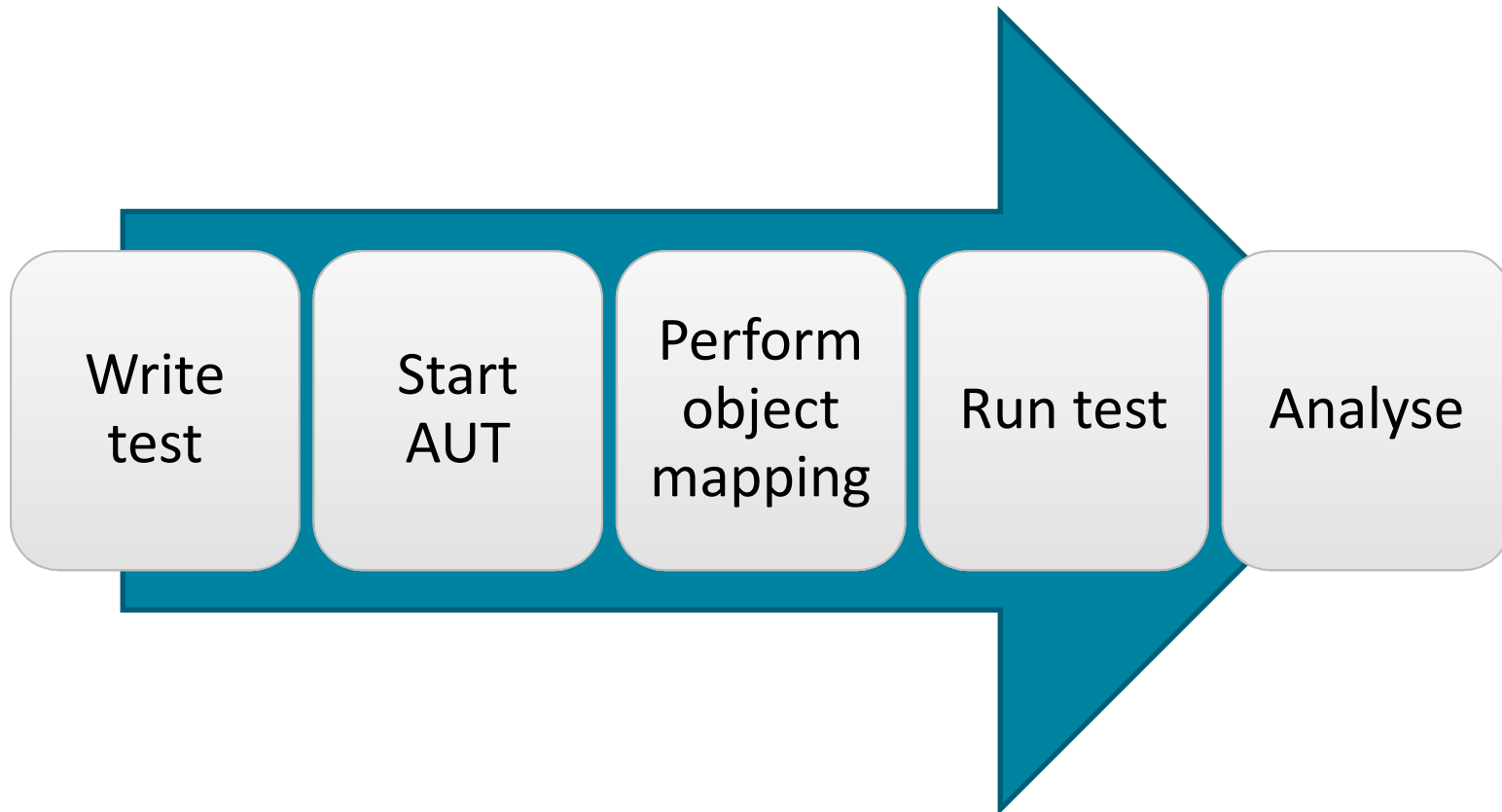
Torso: Police

Legs: Odd

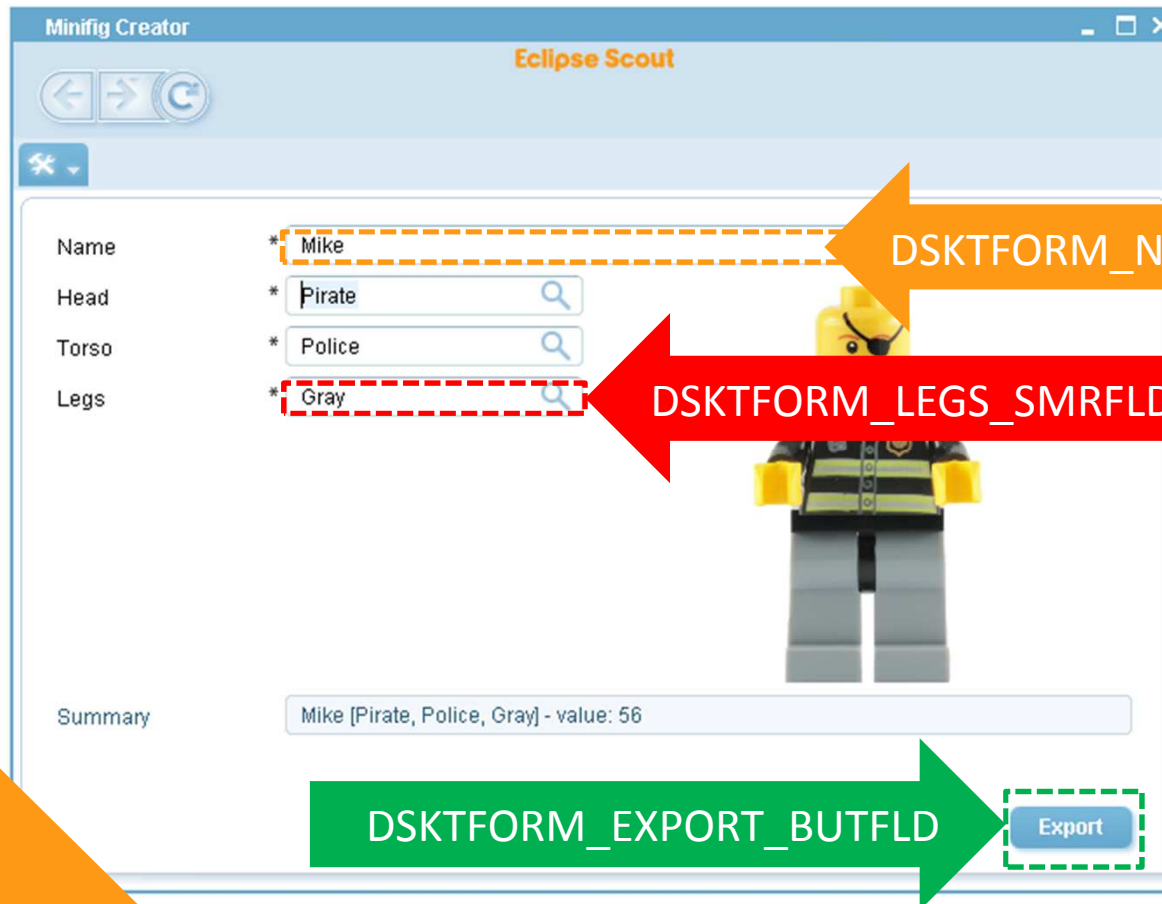
Summary: Mike [Bart, Police, Odd] - value: 68

Export

Testing an application with Jubula



Assign ids to the scout fields



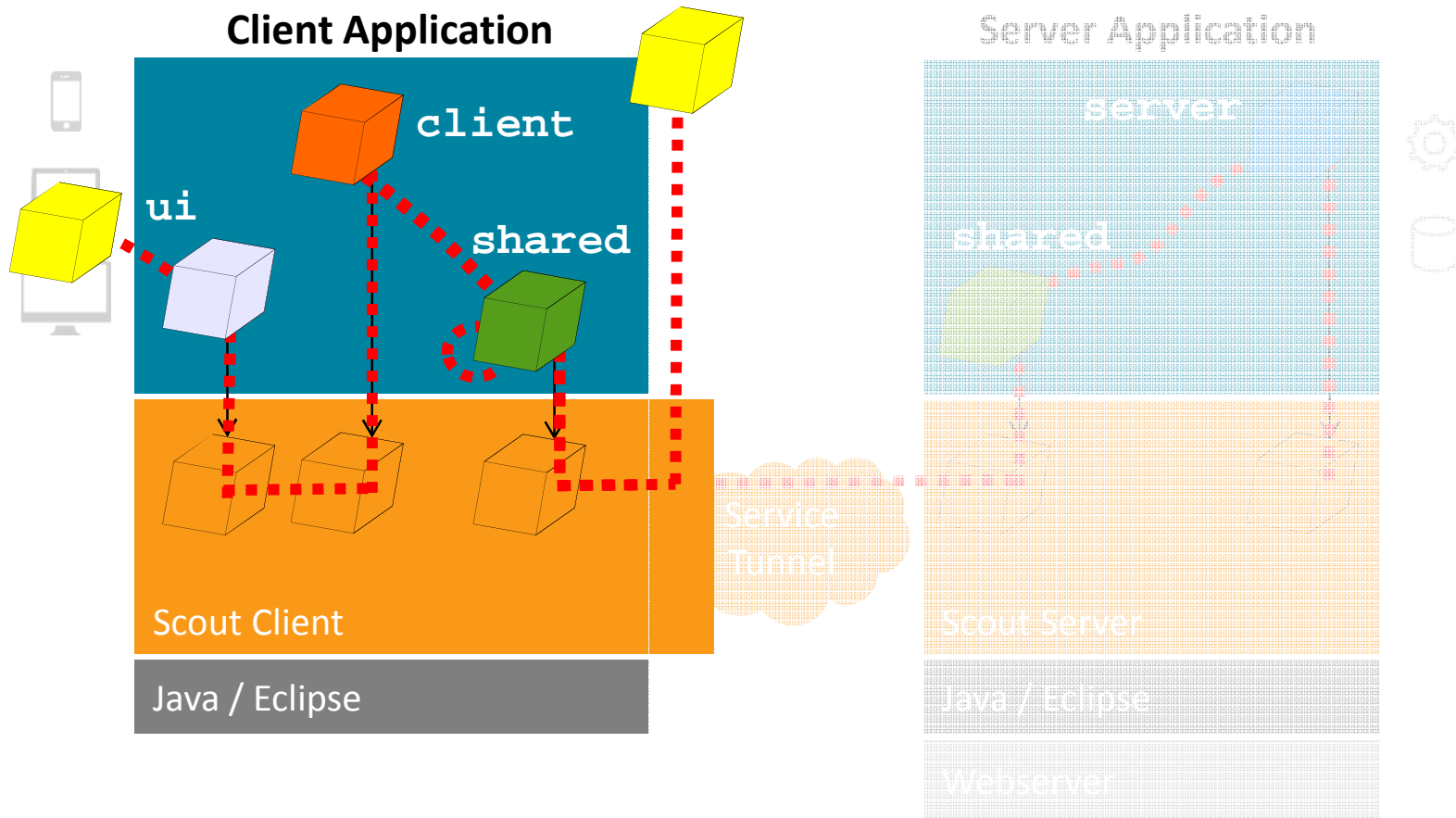
New
with Luna

Demo



Scout UI Tests

Unit tests with UI




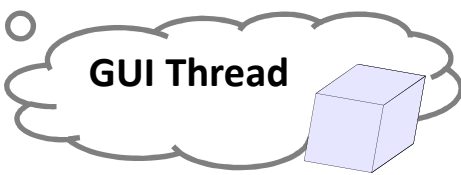
AbstractTestWithGuiScript

```
public class DesktopFormUiTest extends
    AbstractTestWithGuiScript {

    @Override
    protected void runModel() throws Throwable {
    }

    @Override
    protected void runGui(IGuiMock gui) throws Throwable {
    }
}
```

Client Thread 

GUI Thread 

IGuiMock

- Abstraction for the UI layer
- Definition of UI interaction:
 - `gui.pressKey(Key)`
 - `gui.typeText(FieldType, int)`
 - `gui.gotoField(type, index)`
 - ...
- Interface with implementations:
 - For Swing
 - For Swt

**«How are you testing your
Scout application?»**

Summary

- As with any other application, writing automated tests for your eclipse scout application is possible
- Everything is possible
 - Unit tests
 - UI tests
 - Integration tests
 - Performance tests
- There is a cost, so:
test only what makes sense for your application.

Thank You