FINDING THE SCOPE

Patterns and best practices for testing Eclipse RCP Applications

Matthias Kempka (mkempka@eclipsesource.com)

EclipseSource

Why is testing RCP Applications a topic?

EclipseSource

WHAT IS A SCOPE?

EclipseSource

WHAT IS A SCOPE?

AND WHY IS IT IMPORTANT?

EclipseSource

SCOPE: UNIT-TEST

EclipseSource

Advantage:

- ✓ Testing on method/class scope
- ✓ Easy to execute in IDE
- ✓ Fast

EclipseSource

SCOPE: UNIT-TEST

Common problems:

- Code in legacy systems
- Class dependencies
- Misused as Mini-integration tests

EclipseSource

UNIT-TESTS

Finding the scope of a unit test

EclipseSource

Demo: A plain JUnit Test grows into a PDE Test

EclipseSource

A COMMON MISUNDERSTANDING

I want to write a Unit Test

↓

PDE Tests are the Eclipse way to write unit tests

↓

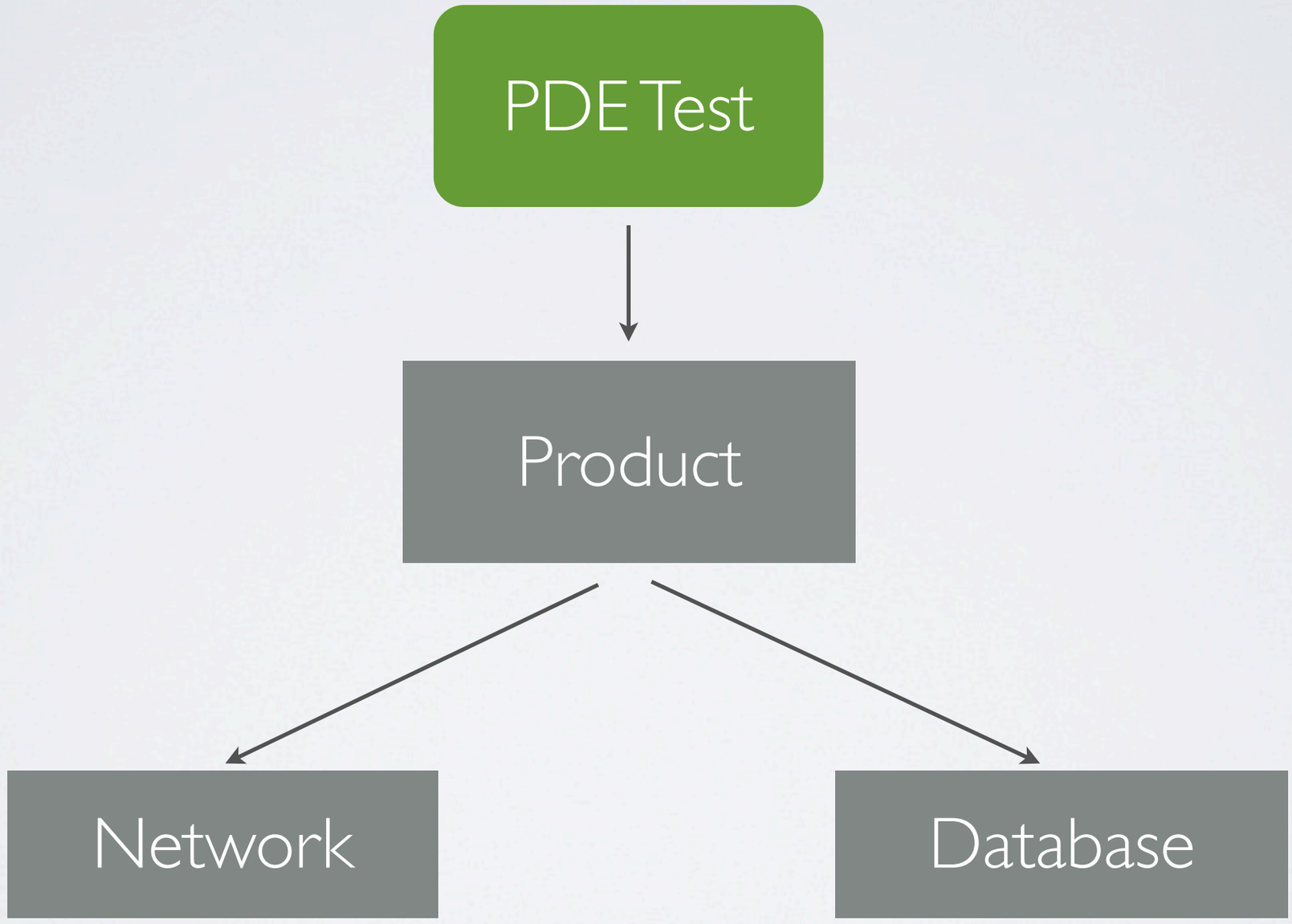
PDE Tests allow execution of code in the product scope

↓

I'll *just* write an integration test

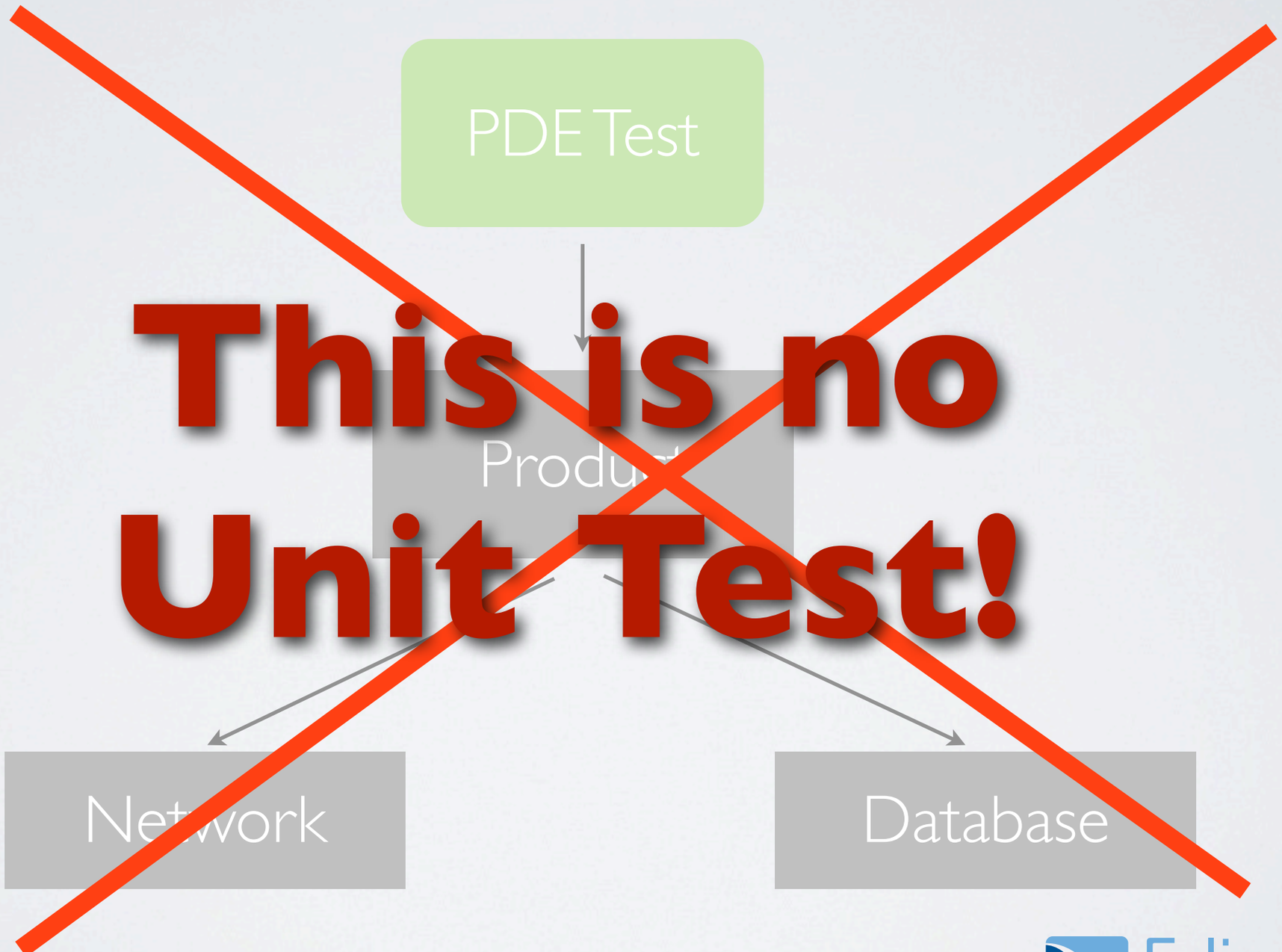
EclipseSource

THE MISUNDERSTOOD PDE TEST



EclipseSource

THE MISUNDERSTOOD PDE TEST



EclipseSource

UNIT TEST

- A unit is a small chunk of code
- Should be tested independent from other code
- That's not always possible, especially in systems that where not created test driven

EclipseSource

A SOFTWARE UNIT

```
public static boolean needsLogin(IConfiguration config,
                                 ILocalProfile profile) {
    if (config.getForceLogin()) {
      return true;
    }
    if (profile == null) {
      return false;
    }
    if (!profile.getShared()) {
      return false;
    }
    if (profile.getType().equals(Type.OPEN)) {
      return false;
    }
    return true;
}
```

EclipseSource

INTERCEPTION POINTS

An **Interception Point** is a point in your program where you can detect the effects of a particular change. (Michael Feathers, *Working Effectively with Legacy Code*)

EclipseSource

INTERCEPTION POINT

```
public class Invoice {

    public Money getValue() {
        Money total = itemsSum();
        if( billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                    originator.getState().equals("NY")) {
                total.add(getLocalShipping());
            } else {
                total.add(getDefaultShipping());
            }
        } else {
            total.add(getSpanningShipping());
        }
        total.add(getTax());
        return total;
    }
```

EclipseSource

INTERCEPTION POINT

- The method on the previous slide needs some changes.
- On the way it shall be refactored into this method:

```
public class Invoice {

    public Money getValue() {
        Money total = itemsSum();
        total.add(shippingPricer.getPrice());
        total.add(getTax());
        return total;
    }
```

EclipseSource

EFFECTS OF GETVALUE

- `getValue()` will change.
- The only use is in another class `BillingStatement`

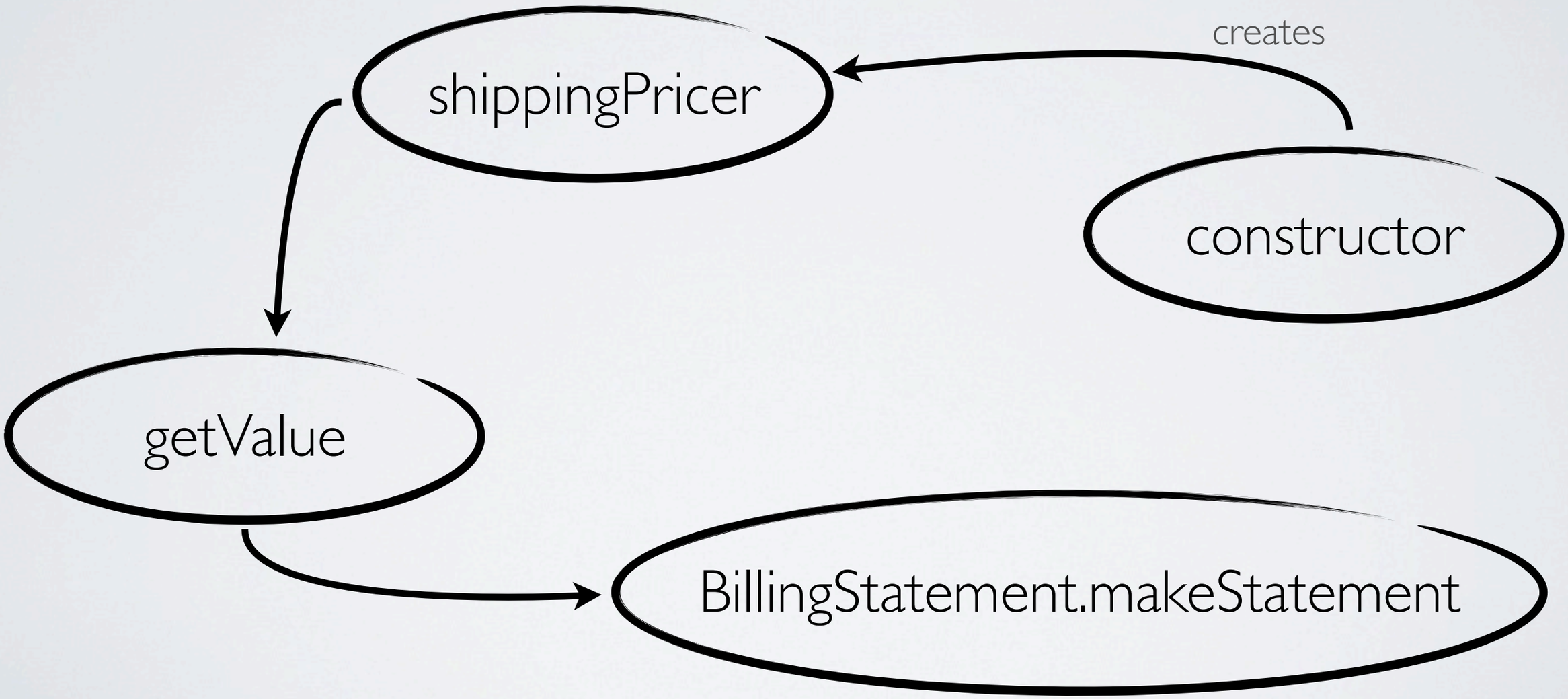
EclipseSource

EFFECTS ON GETVALUE

- Several other things will change that affect `getValue`.

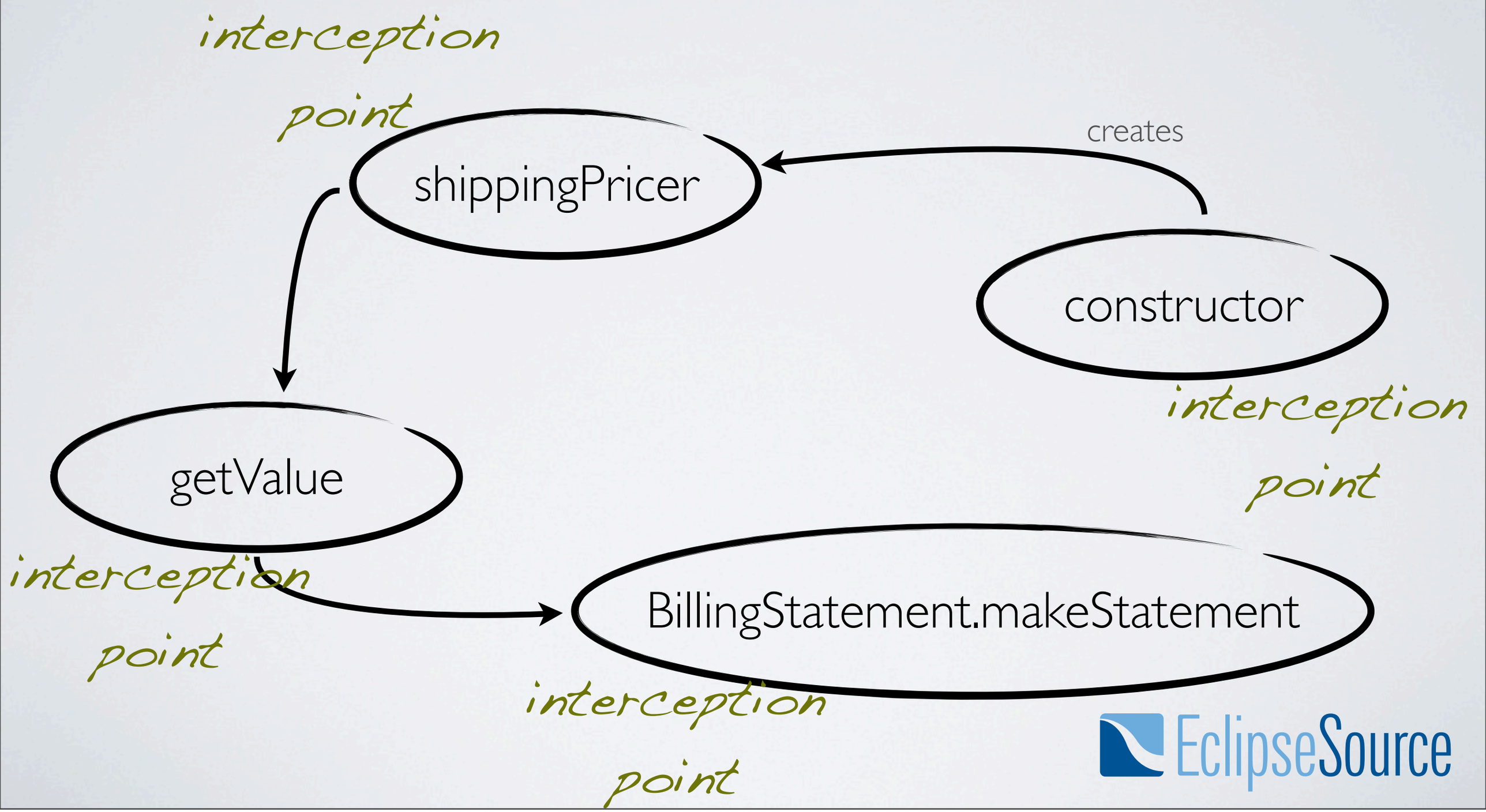
EclipseSource

A CHAIN OF EFFECTS



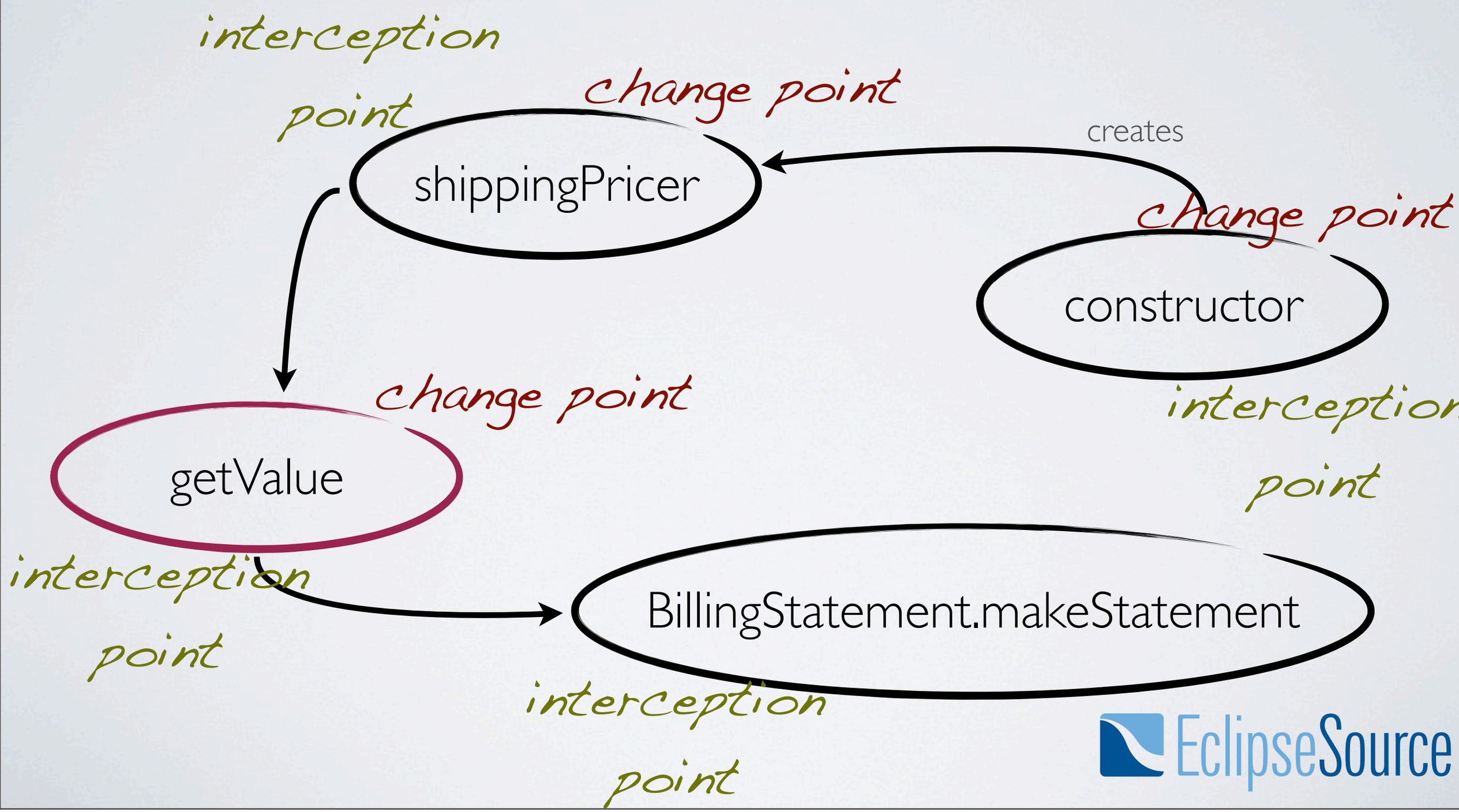
EclipseSource

INTERCEPTION POINTS



EclipseSource

CHANGE POINTS



EclipseSource

INTERCEPTION POINTS

- Pick your interception point close to the change points
  - **Safety**: Every step between a change point and an interception point is like a logical argument
  - **Practicability**: In general (not always) it's harder to set up interception points that are far away
  - **Maintainability**: Your tests serve as regression tests. You don't want to observe more effects than necessary.

EclipseSource

A JUNIT FEATURE

JUnit Rules

EclipseSource

JUNIT RULES

- Rules have been (unnoticed) in JUnit for a while (since 4.7)
- A simple way to get code run before and after the test
  - In the past, test runners where used for that

EclipseSource

TEMPORARYFOLDER

- A rule is a field, annotated with @Rule
  - must be public

```
@Rule
public TemporaryFolder tmpfolder = new TemporaryFolder();

@Test
public void useFiles() throws Exception {
    File propsfile = tmpfolder.newFile("myfile.properties");
    // do something with the file
}
```

EclipseSource

PREDEFINED RULES

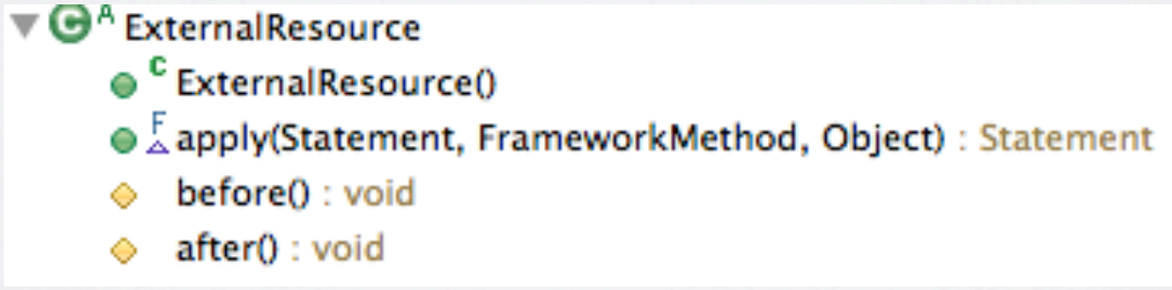
- JUnit comes with a few rules predefined
  - TemporaryFolder - Provides files that live as long as the test
  - ExpectedException - A replacement for @Test(expected=...)
  - TestName - Provides access to the test name
  - Timeout - A replacement for @Test(timeout=...)
  - ErrorCollector - Collect test failures instead of failing at the first error

EclipseSource

CREATING A RULE

- Creating a rule is done by implementing `org.junit.rules.MethodRule`

- If you simply want to execute something before and after a test method, extend `org.junit.rules.ExternalResource`



EclipseSource

A RULE FOR TESTS WITH SWT

```
public class SWTShell extends ExternalResource {
    private Shell parent;

    @Override protected void before() throws Throwable {
        Shell shell = new Shell(Display.getCurrent());
        shell.pack();
        shell.setVisible(true);
        setParent(shell);
    }
    @Override protected void after() {
        getParent().dispose();
    }

    public Shell getParent() {
        return parent;
    }
```

EclipseSource

INTERCEPTION POINTS

Demo: Testing View code without workbench

EclipseSource

UNIT TESTS IN THE CONTINUOUS INTEGRATION

- If set up correctly, Unit tests generally can run as PDE tests
- Set up a test suite that runs all Unit and PDE tests as PDE tests
- Alternative: Put all the plug-ins on the java classpath and run the JUnit Tests in a normal Java environment (common solution in OSGi and RAP applications)

EclipseSource

SCOPE: INTEGRATION TEST

EclipseSource

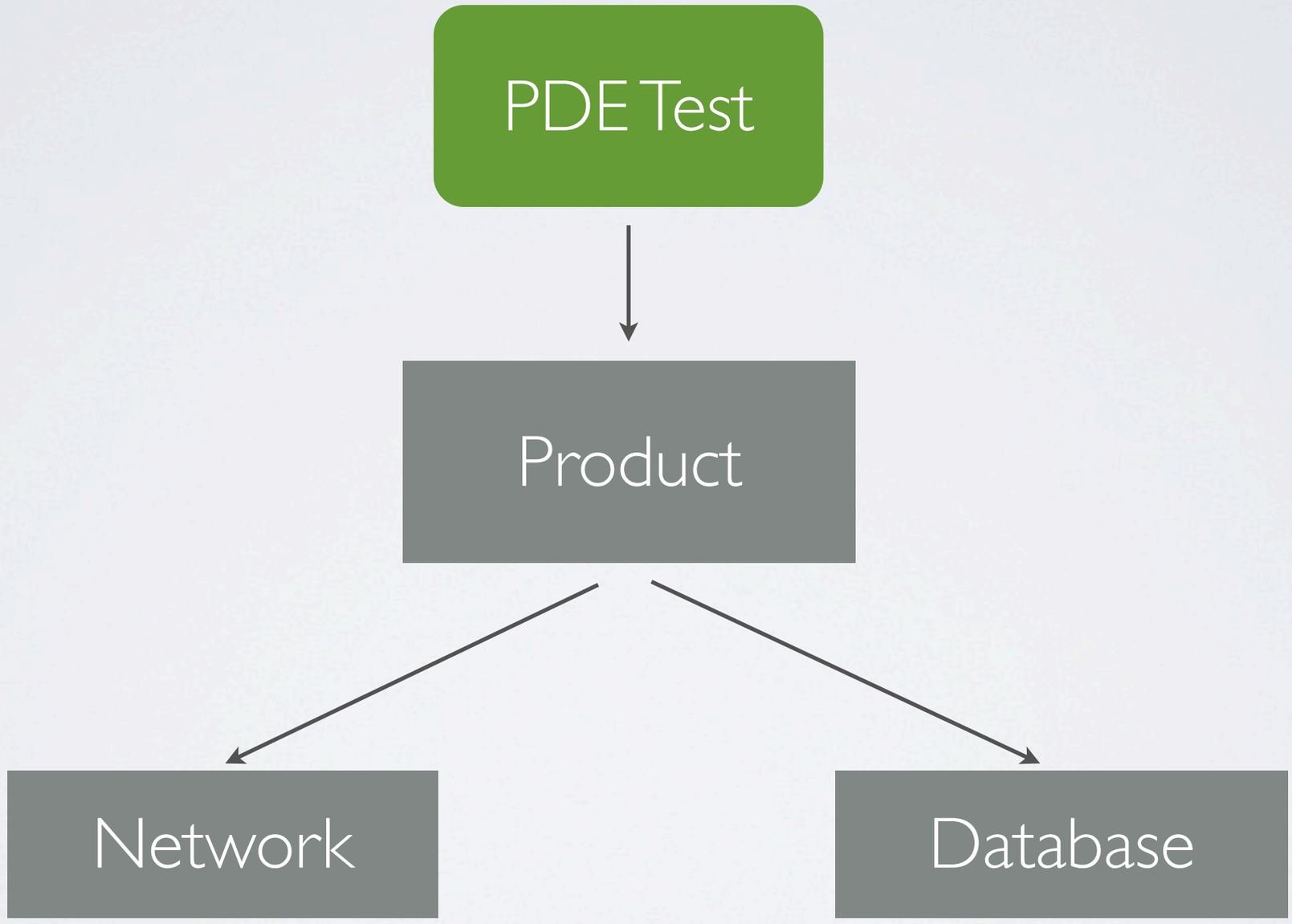
Advantage:

✓ Regression tests on a scope where you don't execute on a per-day basis

✓ High trust factor

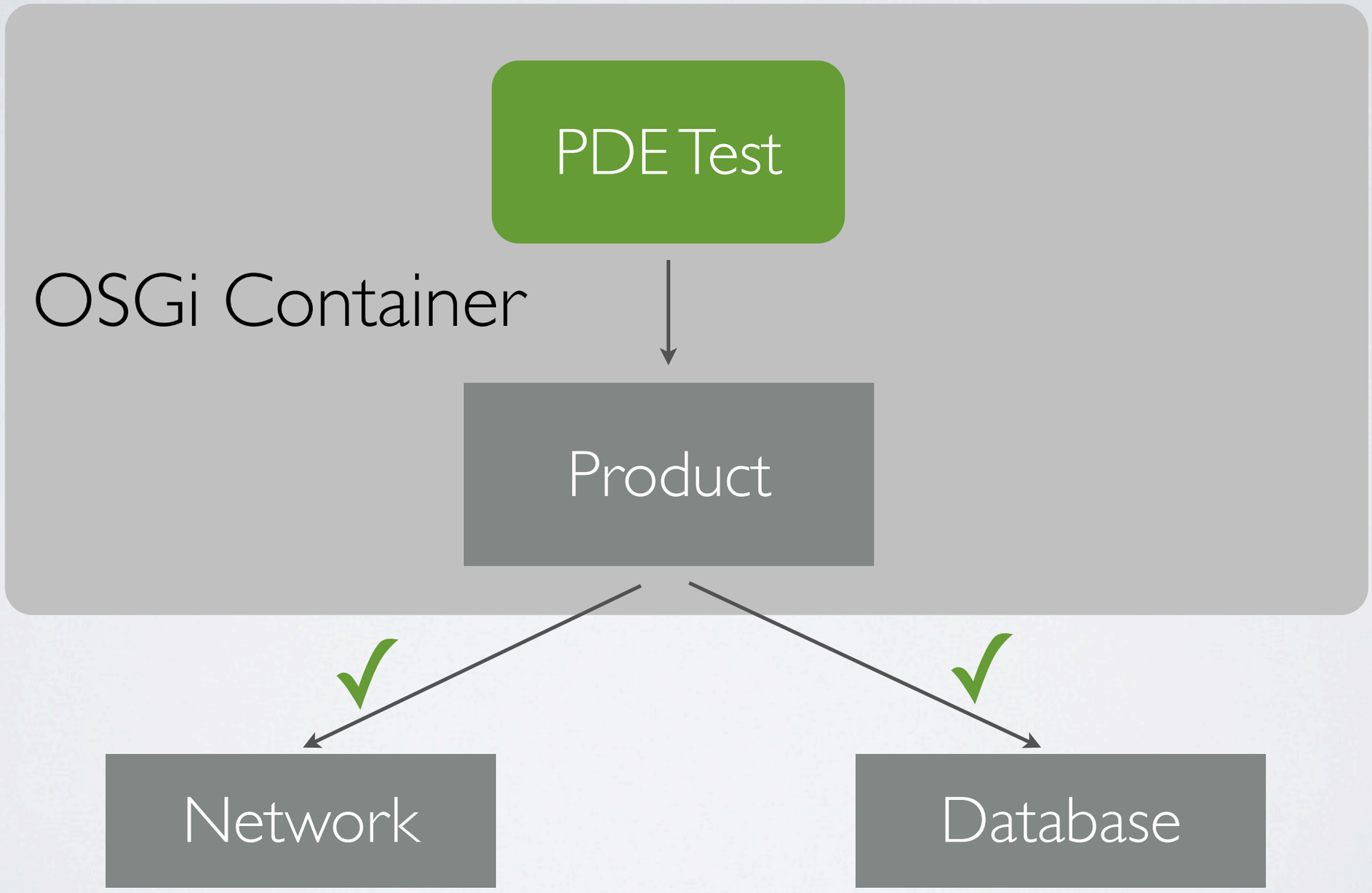
EclipseSource

PDE TEST AS INTEGRATION TEST



EclipseSource

PDE TEST AS INTEGRATION TEST



EclipseSource

PDE TEST AS INTEGRATION TEST

- Common problems:

- A PDE Test can't do much more than verify that a configuration exists

- Setting external resources up (and cleaning them up) often must be done external

EclipseSource

COMMON SOLUTIONS

- Common solutions for integration tests include

✓ Having a fixed test user/test data in the development database

✓ Scripts that can set up and tear down the environment

- Hard to set up and maintain

EclipseSource

SCOPE: FUNCTIONAL TEST

EclipseSource

Advantage:

- ✓ Test on the same abstraction level as the user sees it
- ✓ High trust factor

EclipseSource

SCOPE: FUNCTIONAL TESTS

Common problems:

- Which tool is the right one?
- Executing in IDE vs automated environment

EclipseSource

FUNCTIONAL TEST TYPES

- Functional tests can be created through
  - Programming
  - Capture/Refactor/Replay
- In any case you need another tool than plain JUnit/PDE Test

EclipseSource

SWTBOT

- SWTBot finds SWT Widgets
- It provides an API for using widgets as if you where a user
- http://www.eclipse.org/swtbot/

EclipseSource

Demo: SWTBot Test Case

EclipseSource

ACCESS TO CODE

- The code for the Rule CaptureScreenshotOnFailure can be found at http://eclipsesource.com/blogs/2010/09/09/capture-screenshot-on-failing-swtbot-tests/

EclipseSource

WHY NOT FUNCTIONAL TESTS?

- Good Unit-Tests can achieve >80% code coverage
- With a careful design, all controller and model logic can be tested
- You won't be able to test Layouting with functional tests

EclipseSource

WHY FUNCTIONAL TESTS

- Is the controller logic attached to the UI?
- Are various code units connected?
- Regression tests for tricky passages

EclipseSource

PROPERTIES OF FUNCTIONAL TESTS

- Functional tests are several orders of magnitude slower than unit tests
- Immediate feedback almost impossible
- Interception points for functional tests are far away from the actual code
- Nothing (automated) is closer to the user experience

EclipseSource

TEST PROJECT STRUCTURE

Test Setup in RCP Application

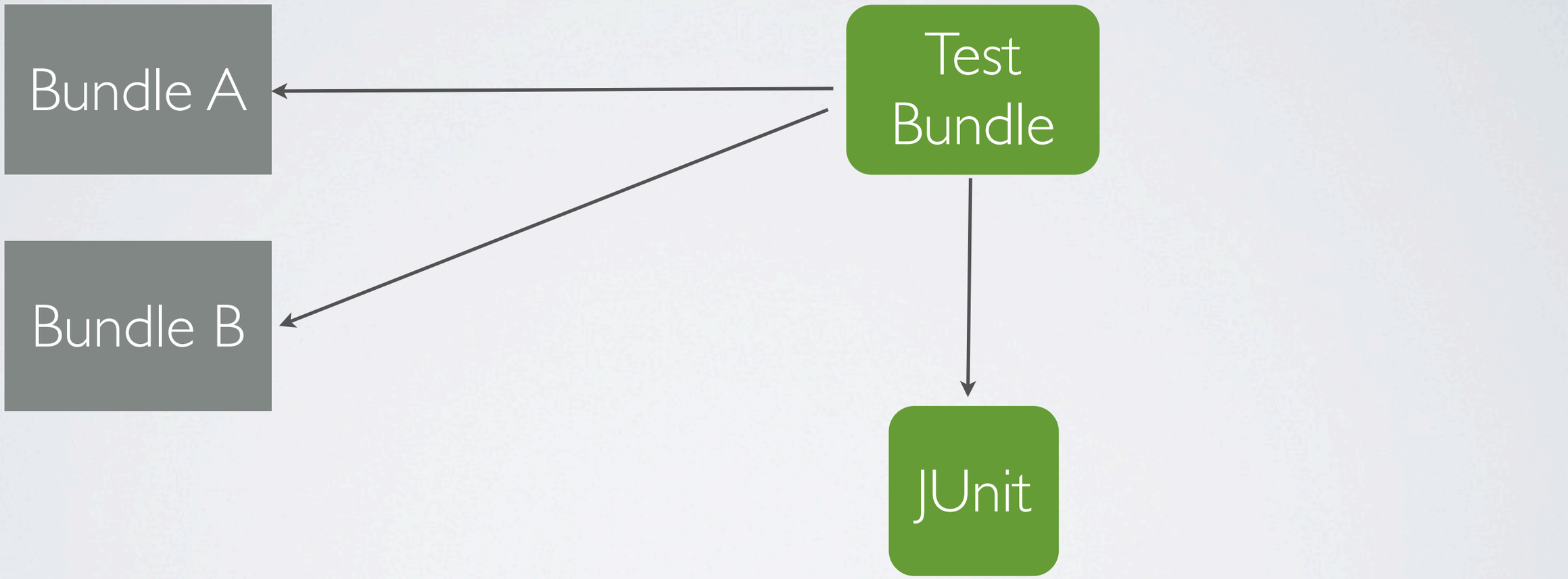
EclipseSource

SETUP IN RCP APPLICATIONS

- We don't want to ship JUnit with the application
- We want to use PDE tests and the Eclipse Testing Framework

EclipseSource

TESTS IN SEPARATE BUNDLES



EclipseSource

TESTS IN SEPARATE BUNDLES

- Advantages:

✓ A Test plug-in for a bunch of bundles

✓ Separate plug-in for Unit Tests

✓ Rather easy to set up and maintain plug-in structure

EclipseSource

TESTS IN SEPARATE BUNDLES

- Consequences:
  - Hard to access internal classes (need to be exported)
  - Every method under test must be public

EclipseSource

USES IN RCP APPLICATIONS

- Works well to some extend
- You'll find a lot of code like this:

```
/* public visibility for testing reasons */
public boolean isSaved() {
    return saved;
}
```

EclipseSource

USES IN RCP APPLICATIONS

- Works well to some extend
- You'll find a lot of code like this:

```
/* public visibility for testing reasons */
public boolean isSaved() {
    return saved;
}
```

- Or worse, like this:

```
/* public visibility for testing reasons */
public boolean saved;
```

EclipseSource

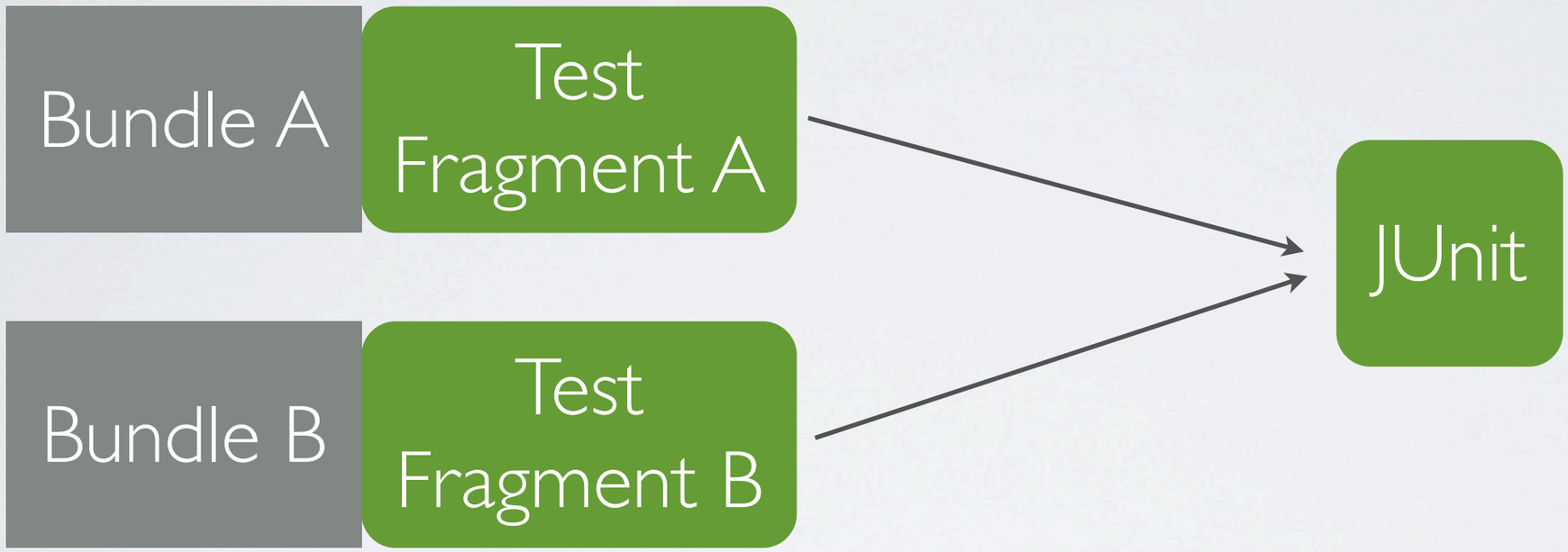
USES IN RCP APPLICATIONS

- Your Manifest.mf contains a lot entries like this:

```
Export-Package: org.eclipse.mail.client;
  x-friends:="org.eclipse.mail.client.testplugin"
```

EclipseSource

TESTS IN FRAGMENTS



EclipseSource

TESTS IN FRAGMENTS

- Advantages:

  ✓ No classloader between test and class

  ➡ We can narrow down the visibility to default

```
/* default visibility for testing reasons */
boolean isSaved() {
    return saved;
}
```

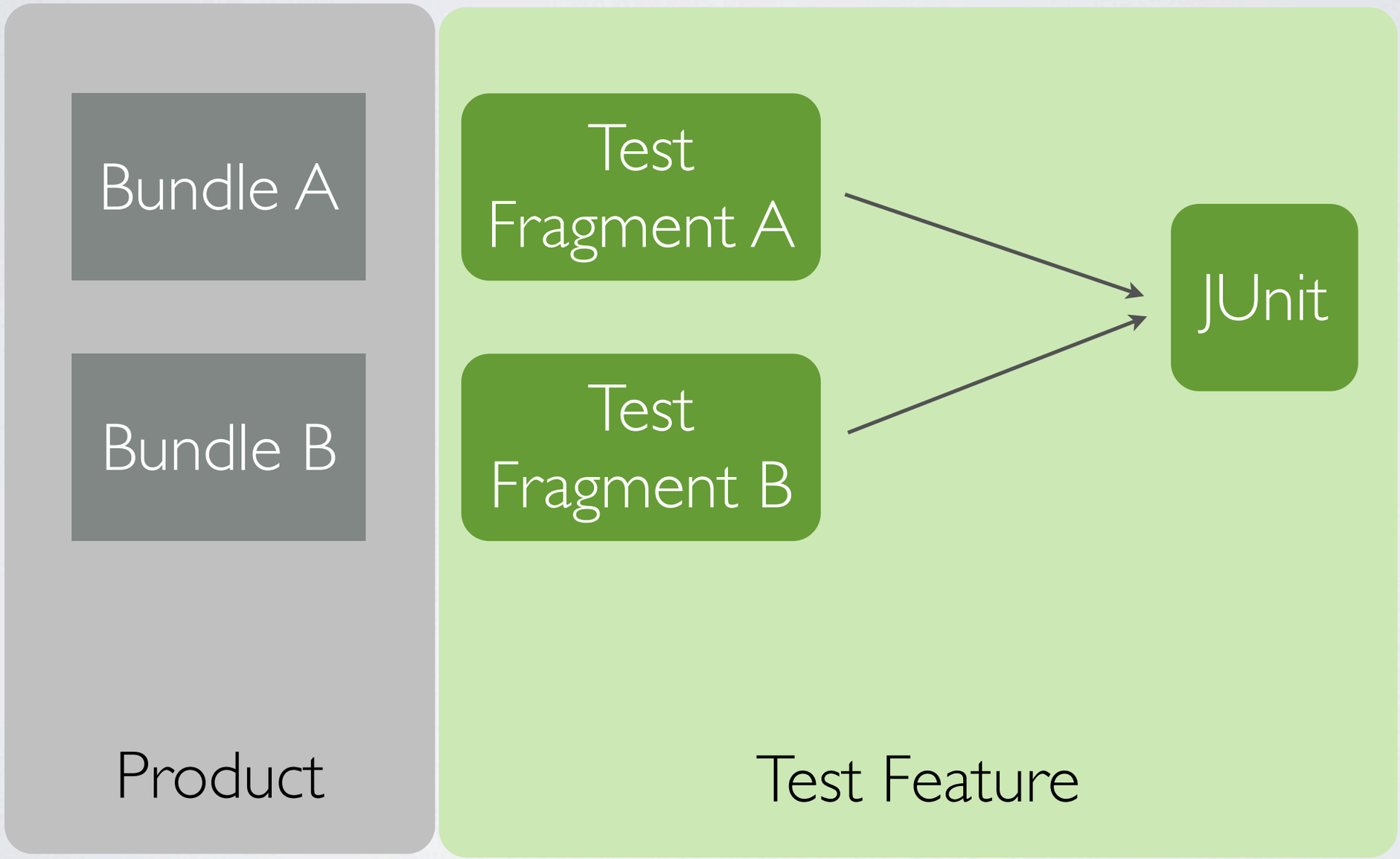
EclipseSource

TESTS IN FRAGMENTS

- Consequences:
  - Every bundle needs a separate test fragment
  - Creating and integrating bundles in the application becomes a heavy-weight task
  - Especially the initial setup frightens off developers

EclipseSource

TESTS IN SEPARATE ARTIFACTS



CONTINUOUS INTEGRATION CONCERNS

- Executing Tests requires:
  - The RCP Application
  - The Test Feature
  - Eclipse Testing Framework
  - JDT + Requirements (This will likely vanish in 3.7)

EclipseSource

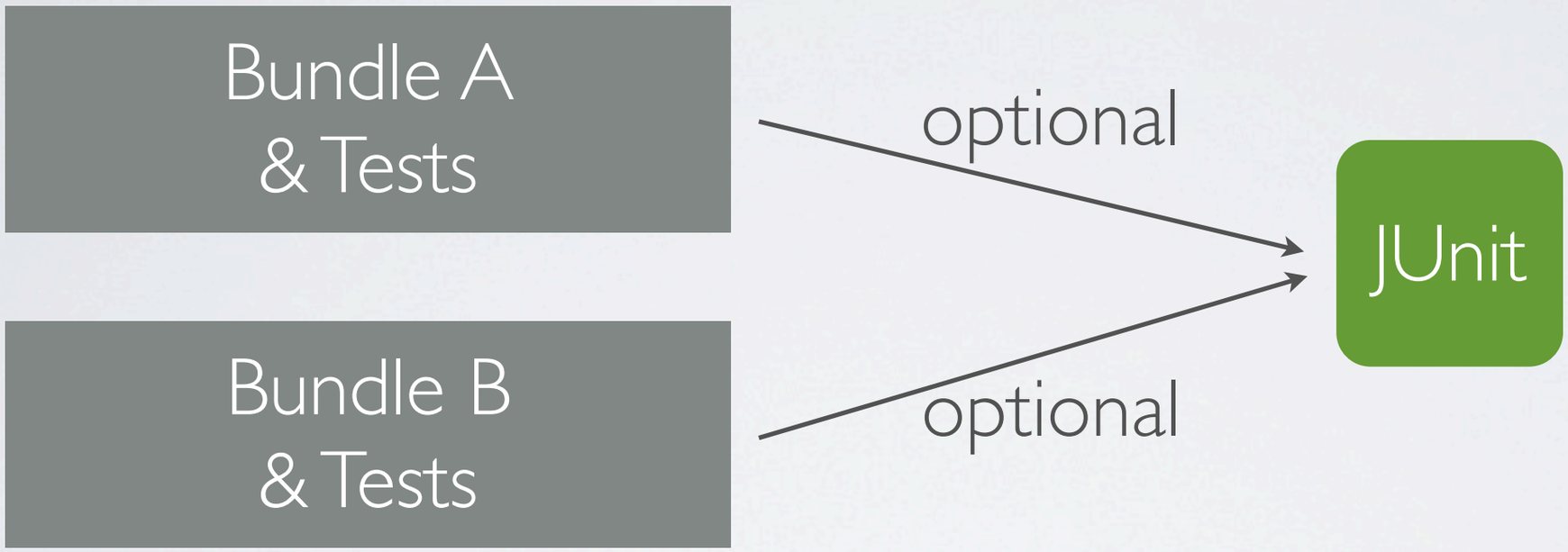
CONTINUOUS INTEGRATION CONCERNS

- In practice, sometimes tests don't get executed because of changed dependencies
- Hard to find out why

➡ Use p2 to install the tests into the product

EclipseSource

TESTS IN THE PLUG-IN



EclipseSource

TESTS IN THE PLUG-IN

- Advantages:

  ✓ No test dependency management overhead

EclipseSource

TESTS IN THE PLUG-IN

- Consequences:
  - Tests ship with the product
  - Hazzles with test/productive code interdependencies

EclipseSource

USES IN OSGI APPLICATIONS

- This is a common structure for OSGi projects
- Not so common in RCP applications

EclipseSource

TEST SUITE

- Don't try to set up a Test Suite across different bundles yourself.
- There'll be another talk about test suites later today
- Bundle Testcollector from Patrick Paulin makes it easy to set up Test Suites in an OSGi container

EclipseSource

BUNDLE TESTCOLLECTOR

- Bundle Testcollector Input
  - pattern for bundle id
  - pattern for class name
- Goes through the specified bundles, pulls together the classes and puts them on a PDE test suite
- http://www.modumind.com/2008/06/12/running-unit-tests-for-rcp-and-osgi-applications/

EclipseSource

BUNDLE TEST COLLECTOR

- Small problem:

- The Bundle Testcollector is only able to construct JUnit3 Test Suites

- A small change is necessary to make it compatible with JUnit4 tests (wrap the found class in a JUnit4TestAdapter).

EclipseSource

TEST SUITE

Demo: Setting up a Test Suite with BundleTestCollector

EclipseSource

TEST SUITES

- Structure your test suites by execution speed and Test Runners
  - SWTBot tests need a separate test runner
  - Unit tests are meant to be fast, developers will execute them regularly
  - Integration tests may take a while, they will mainly be executed in the continuous integration

EclipseSource

ACCESS TO CODE

The Bundle Testcollector that was demonstrated can be accessed at http://eclipsesource.com/blogs/2010/09/09/an-almost-perfect-test-suite/

EclipseSource

CONCLUSION

- Use the tools at hand
- Efficient testing comes with differentiation and structure

EclipseSource

REFERENCES

- Michael C. Feathers, *Working Effectively with Legacy code*
- http://eclipse.org/swtbot

EclipseSource