Agenda

- Errors in programming
- Testing and debugging:
  How to detect and locate errors.
- Test cases: Online shop project
- Test automation: How to automate test with JUnit

We have to deal with errors

Syntax errors

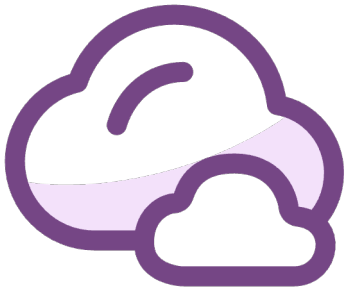
- The compiler will spot these.

Logic errors - runtime errors

- The compiler cannot help with these
- Also known as bugs.

Some logical errors have no immediately obvious manifestation.

- Commercial software is rarely error free.



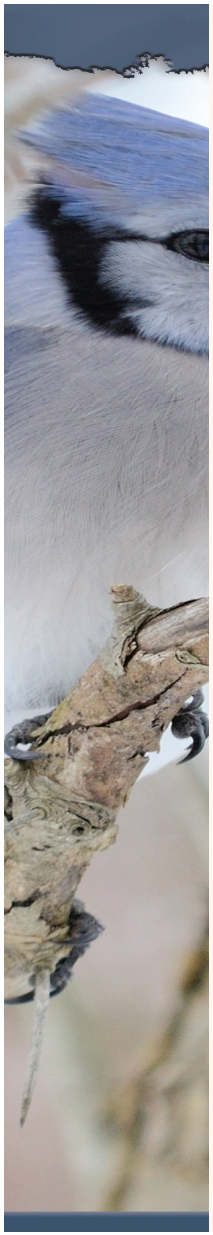
Was haben Sie bisher schon gemacht um Fehler zu entdecken / vermeiden?

Testing, debugging, writing for maintainability

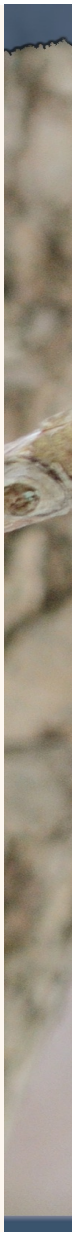
Testing: searches for the presence of errors.

Debugging: searches for the source of errors. Localizing the error. The manifestation of an error may well occur some ‘distance’ from its source.

Writing for maintainability: preventing errors



Demo of *online-shop*



Test Case: input & output

Test	Situation	behaviour	OK
T1	Can comments be added?	T1.1 Get number of comments (empty, 0) T1.2 Get number of comments (1, 1) T1.3 Get number of comments (1+1+1+1, 4)?	
T2	Can comments be removed?	T2.1 removeComment()	
T3	Does show info show **all** information?	T3.1 ShowInfo	
T4	One comment per user enforced?	T4.1 Positive test: add (“Karl”,succeed) T4.2 Negative test: add(“Karl”, false)	
T5	Only rating between 1-5 entered?	T5.1 enter comment (1, suceed) T5.2 enter comment (5, suceed) T5.3 enter comment (6, false) T5.4 enter comment (0, false) T5.5 enterComment (-3, false)	
T6	Returns most helpful comment	T6.1 find MostHelpfulComment(mostHelpful, mostRated) T6.2 findMostHelpfulComment(0, 0)	

Testing boundary conditions

How do things behave at the beginning?

How do things behave when there is none?

How do things behave at the end?

Testing fundamentals

- Understand what the unit should do – its contract.
- You will be looking for violations.
- Use positive tests: what is expected to succeed
- and negative tests: what is expected to fail.
- Test boundaries.
- Zero, One, Full.
- Search an empty collection.
- Add to a full collection.

Testing and debugging techniques

- Manual walkthroughs
- Print statements
- Debuggers
- Using JUnit, Unit testing (within BlueJ) and test automation

Unit testing

Each unit of an application may be tested.

Method, class, module (package in Java).

Should be done during development.

Finding and fixing early lowers development costs (e.g. programmer time).

A test suite is built up.

Unit Testing vs. Application Testing

- Unit testing: focuses on individual parts (such as a class or a group of classes), Application testing: considers the system as a whole
- A Unit is **not** a Class, can be group of classes focus on small pieces of the system.
- In BlueJ, there is direct interaction with classes, which may not be the case in other environments.
- Early Testing: The note suggests that it's never too early to start testing.
- Cost-Effectiveness: It's cheaper to fix problems early during unit testing.
- Test Cases and Changes: Test cases can be reused after changes are made to ensure that no additional errors have been introduced.

Junit for regression Testing

“Unter einem **Regressionstest** (von lateinisch *regredior*, *regressus sum* ‚zurückschreiten‘) versteht man in der Softwaretechnik die Wiederholung von Testfällen, um sicherzustellen, dass *Modifikationen* in bereits getesteten Teilen der Software keine neuen Fehler („Regressionen“) verursachen. Solche Modifikationen entstehen regelmäßig z. B. aufgrund der Pflege, Änderung und Korrektur von Software.”

Fokus auf Programmfehler, die durch dynamische Laufzeitparameter (z.B. Nutzer-Interaktion) auftreten!

https://de.wikipedia.org/wiki/Regressionstest

Test automation

Good testing is a creative process, but ...

... thorough testing is time consuming and repetitive.

Regression testing involves re-running tests.

Use of a *test rig* or *test harness (Geschirr)* can relieve some of the burden.

Test harness (Pferdegeschirr)

Additional test classes are written to automate the testing.

Objects of the harness classes replace human interactivity.

Creativity and imagination required to create these test classes.

Test classes must be kept up to date as functionality is added.

Test automation

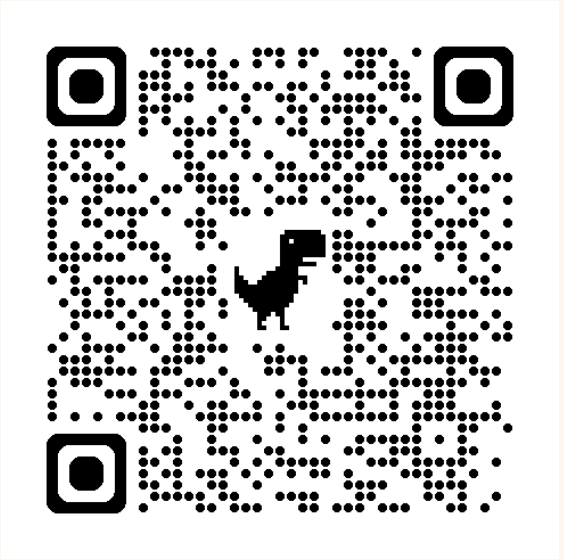
Test frameworks exist to support automation.

Explore fuller automation through the *online-shop-junit* project.

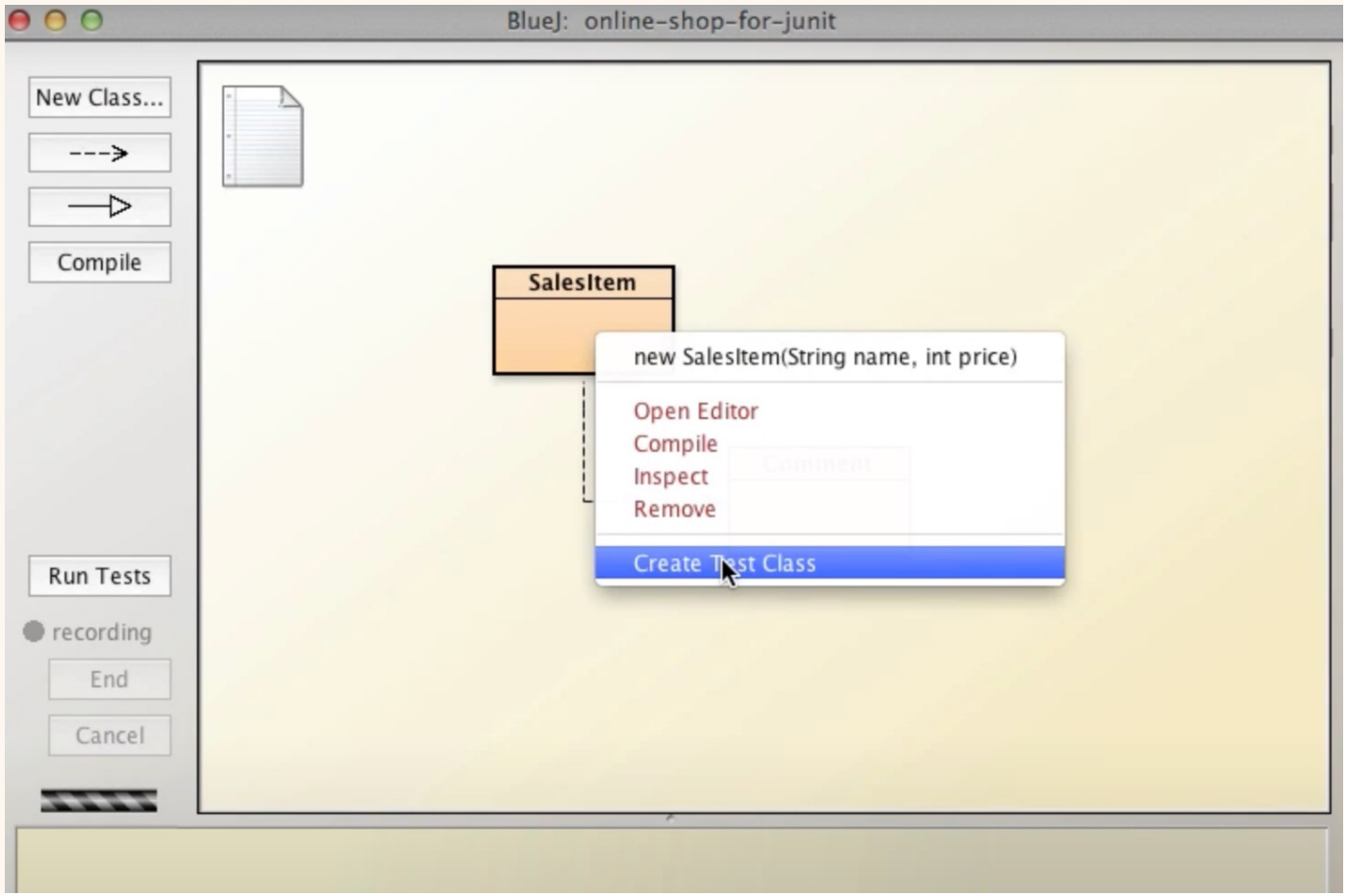
- Intervention only required if a failure is reported.

Demo of *online-shop-junit*

https://www.youtube.com/watch?v=iENnVe4aKuo



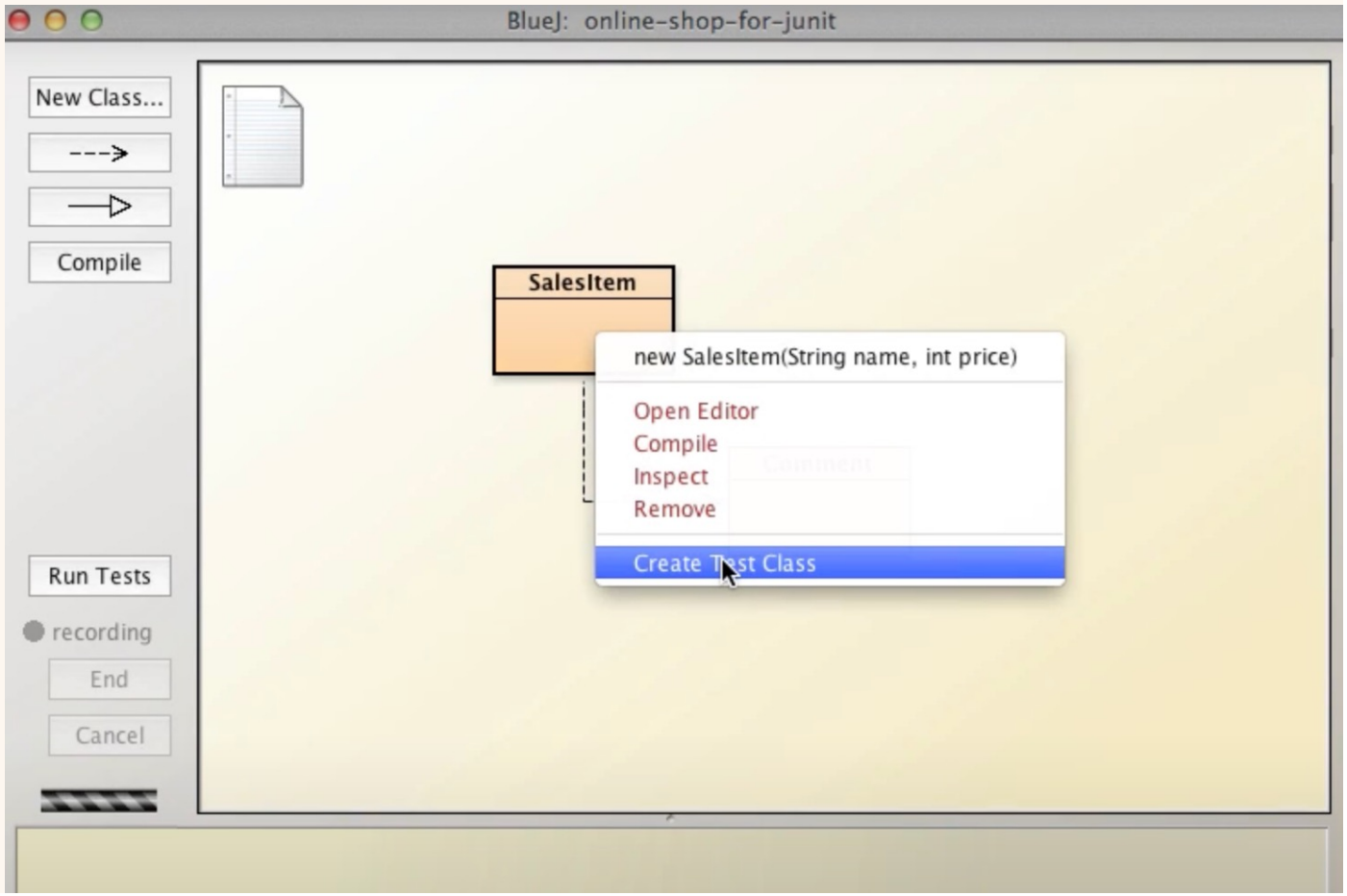
Create Test Class



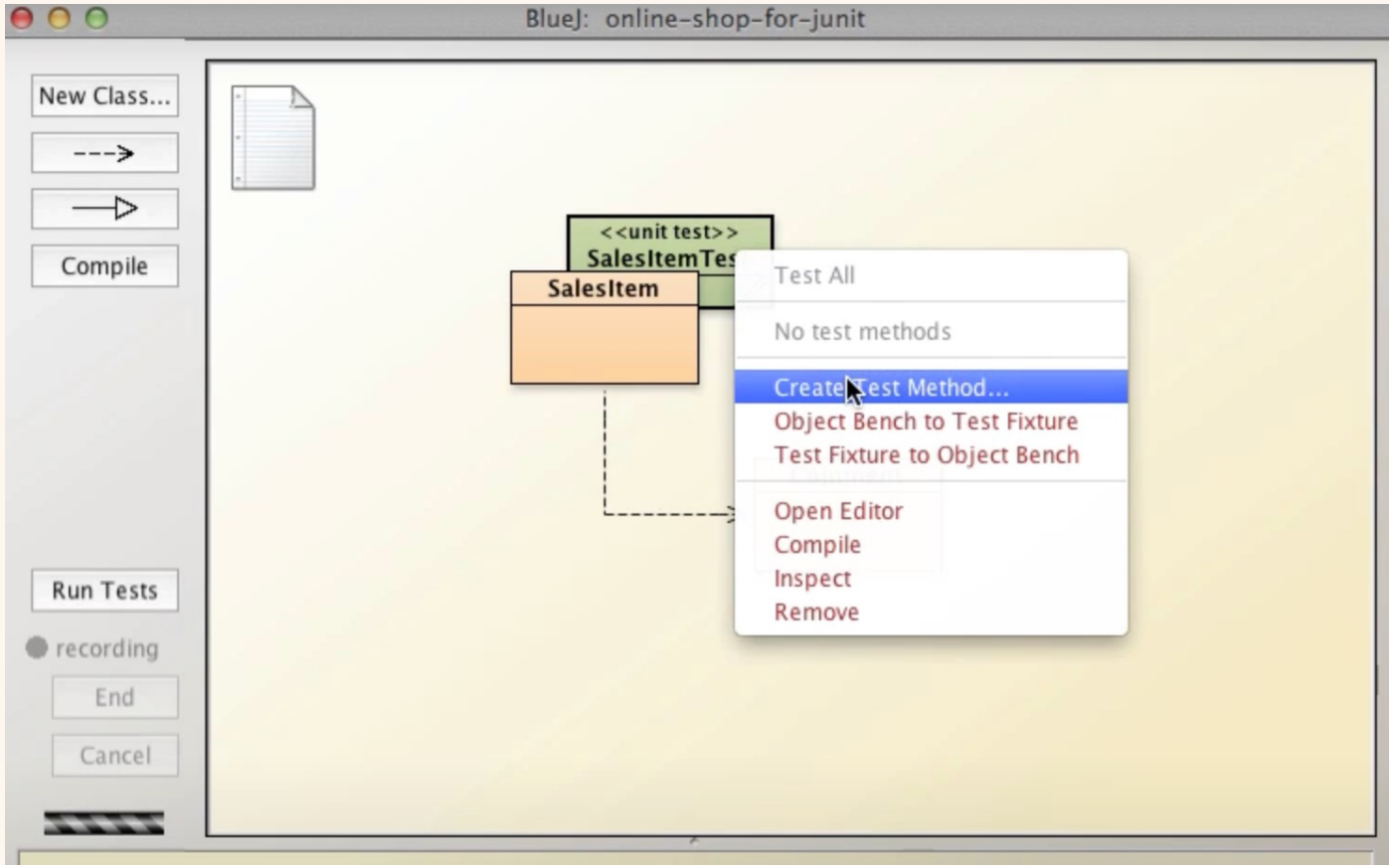
Test Case: input & output

Test	Situation	behaviour	OK
T1	Can comments be added?	T1.1 Get number of comments (empty, 0) T1.2 Get number of comments (1, 1) T1.3 Get number of comments (1+1+1+1, 4)?	
T2	Can comments be removed?	T2.1 removeComment()	
T3	Does show info show **all** information?	T3.1 ShowInfo	
T4	One comment per user enforced?	T4.1 Positive test: add (“Karl”,succeed) T4.2 Negative test: add(“Karl”, false)	
T5	Only rating between 1-5 entered?	T5.1 enter comment (1, suceed) T5.2 enter comment (5, suceed) T5.3 enter comment (6, false) T5.4 enter comment (0, false) T5.5 enterComment (-3, false)	
T6	Returns most helpful comment	T6.1 find MostHelpfulComment(mostHelpful, mostRated) T6.2 findMostHelpfulComment(0, 0)	

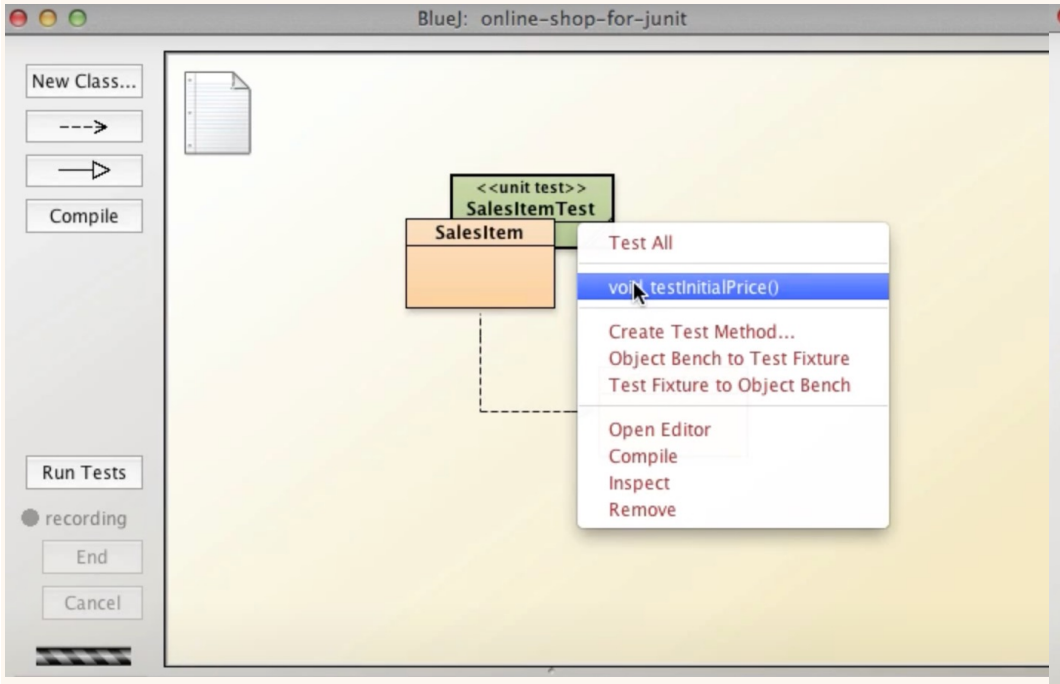
Create Test Class

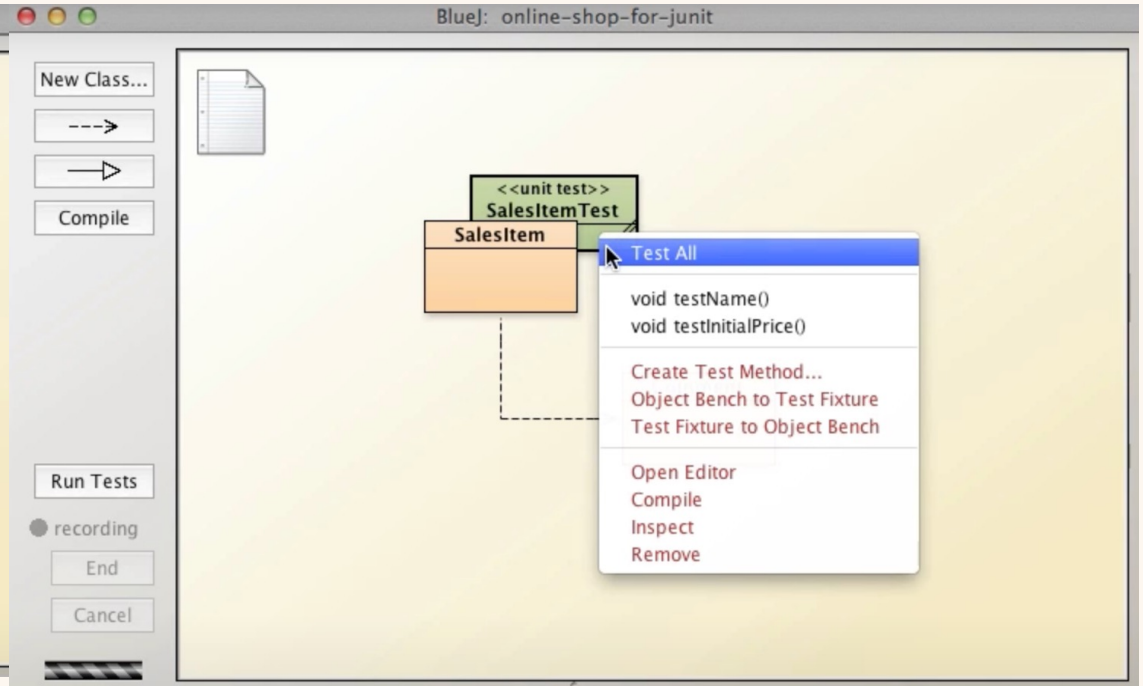


Define a new method for each Test



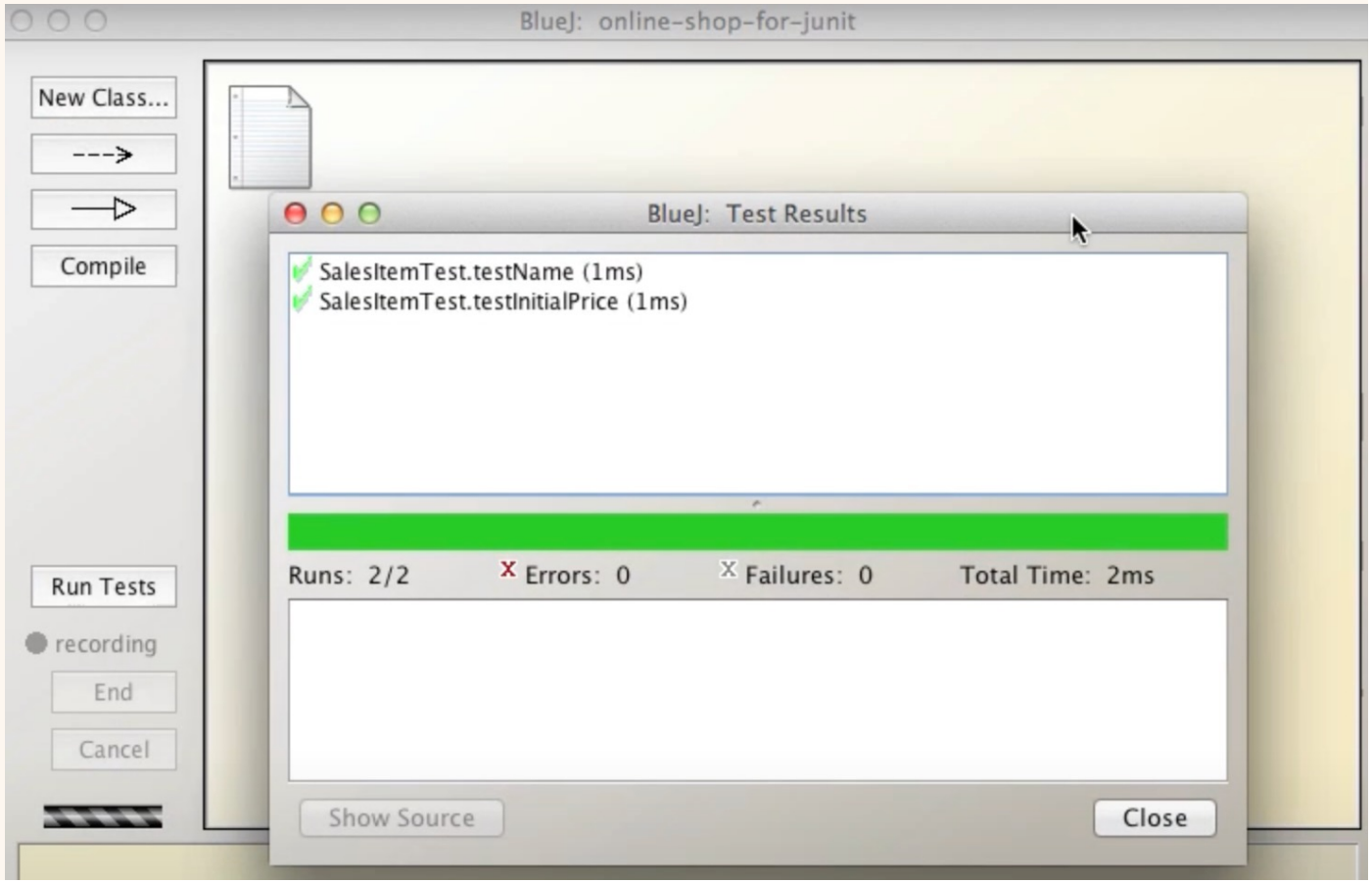
Run Test


BlueJ: online-shop-for-junit
New Class...
Compile
Run Tests
recording
End
Cancel
<<unit test>>
SalesItemTest
SalesItem
Test All
void testInitialPrice()
Create Test Method...
Object Bench to Test Fixture
Test Fixture to Object Bench
Open Editor
Compile
Inspect
Remove

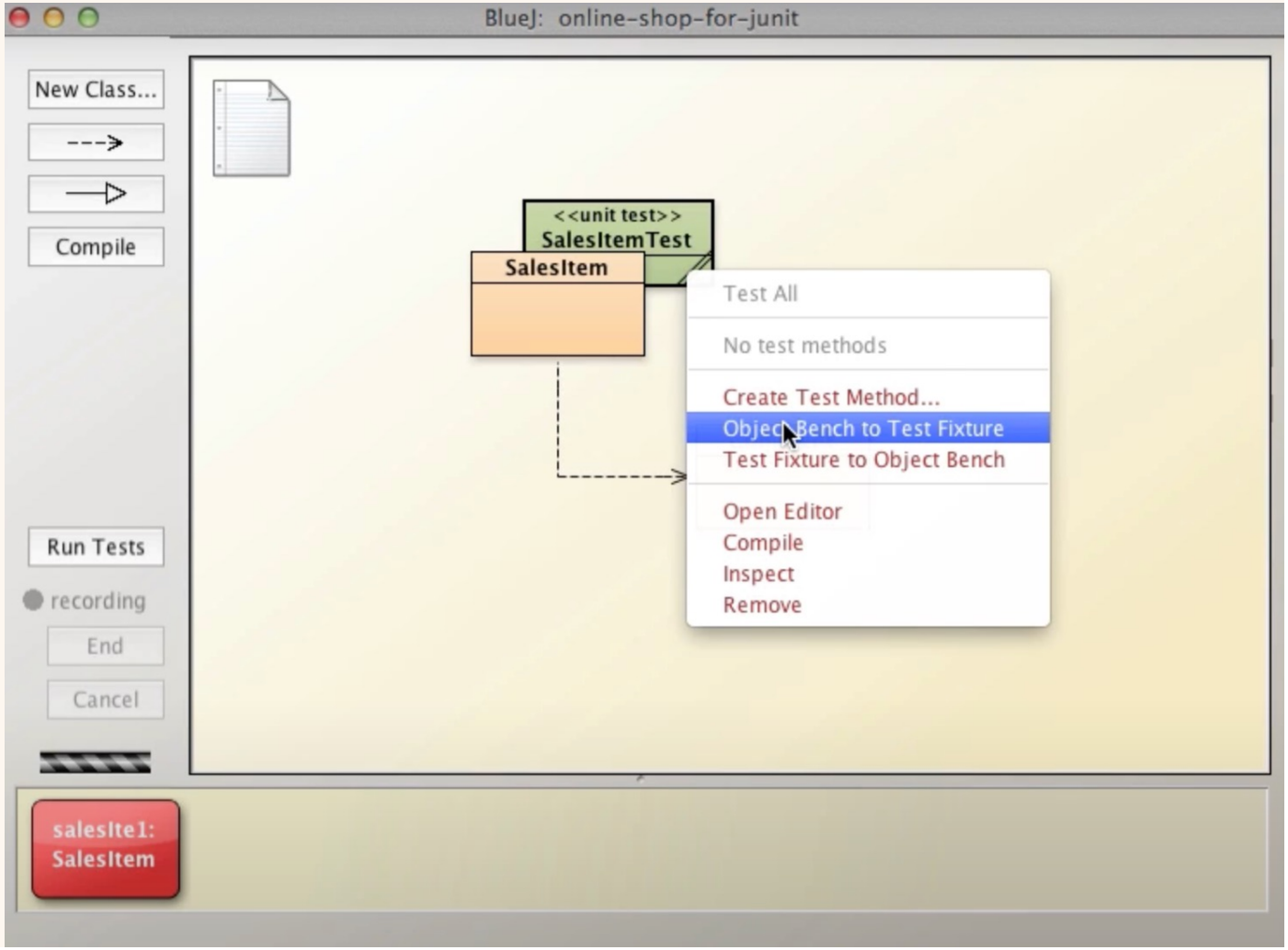

BlueJ: online-shop-for-junit
New Class...
Compile
Run Tests
recording
End
Cancel
<<unit test>>
SalesItemTest
SalesItem
Test All
void testName()
void testInitialPrice()
Create Test Method...
Object Bench to Test Fixture
Test Fixture to Object Bench
Open Editor
Compile
Inspect
Remove

See Test results

Test all



Create a setup



Summary

- JUnit test framework has been integrated to BlueJ.
- JUnit allows us to record tests interactively and run those tests repeatedly.
- This test automation feature makes it more likely that we will run regression tests whenever we make modifications to our program.
- It ensures that we don’t just test the most recent additions.
- Changes to one part of a program often have unexpected and undesired effects on other parts that we were not aware of.
-

JUnit

JUnit is a Java test framework

Test cases are methods that contain tests

Test classes contain test methods

Assertions are used to assert expected method results

Fixtures are used to support multiple tests

```java
/**
 * Sets up the test fixture.
 *
 * Called before every test case method.
 */
@BeforeEach
public void setUp()
{
}
```

```java
/**
 * Tears down the test fixture.
 *
 * Called after every test case method.
 */
@AfterEach
public void tearDown()
{
}
```

Debugging techniques

- Print statements
- Debuggers
- Manual walkthroughs

Print statements

The most popular technique.

No special tools required.

All programming languages support them.

Only effective if the right methods are documented.

Output may be voluminous!

Turning off and on requires forethought.

Detection vs Prevention

We can improve the chances of detection:

- Use software engineering practices, like modularization and good documentation.

We can develop detection skills.

We can lessen the likelihood of errors (upcoming):

- Use software engineering techniques, like encapsulation.
- Pay attention to cohesion and coupling.

Debuggers

Debuggers are both language- and environment-specific.

- BlueJ has an integrated debugger.

Support breakpoints.

Step and *Step-into* controlled execution.

Call sequence (stack).

Object state.

Modularization and interfaces

Applications often consist of different modules:

- E.g. so that different teams can work on them.

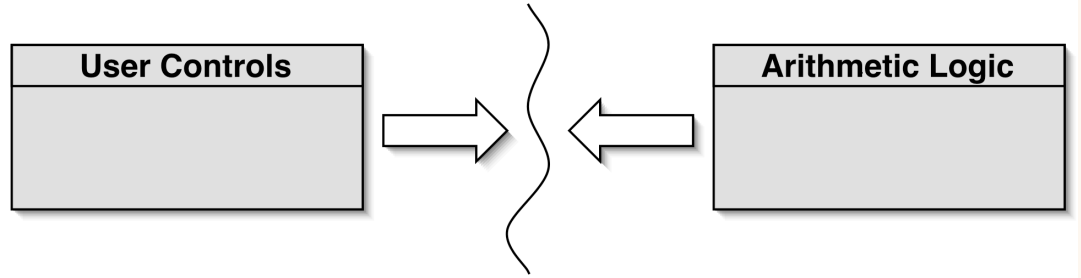
The *interface* between modules must be clearly specified.

- Supports independent concurrent development.
- Increases the likelihood of successful integration.

Modularization in a calculator

Each module does not need to know implementation details of the other.

- User controls could be a GUI or a hardware device.
- Logic could be hardware or software.



Debugging

It is important to develop code-reading skills.

- Debugging will often be performed on others' code.

Techniques and tools exist to support the debugging process.

Explore through the *calculator-engine* project.

Manual walkthroughs

Relatively underused.

- A low-tech approach.
- More powerful than appreciated.

Get away from the computer!

‘Run’ a program by hand.

High-level (Step) or low-level (Step into) views.

Verbal walkthroughs

Explain to someone else what the code is doing:

- *They* might spot the error.
- *You* might spot the error, through the process of explaining.

Group-based processes exist for conducting formal walkthroughs or *inspections*.

Tabulating object state

An object's behavior is largely determined by its state …

… so incorrect behavior is often the result of incorrect state.

Tabulate the values of key fields.

Document state changes after each method call.

Choosing a test strategy

Be aware of the available strategies.

Choose strategies appropriate to the point of development.

Automate whenever possible.

- Reduces tedium.
- Reduces human error.
- Makes (re)testing more likely.

Review

Errors are a fact of life in programs.

Good software development techniques can reduce their occurrence.

Testing and debugging skills are essential.

Make testing a habit.

Automate testing where possible.

Continually repeat tests.

Practice a range of debugging skills.