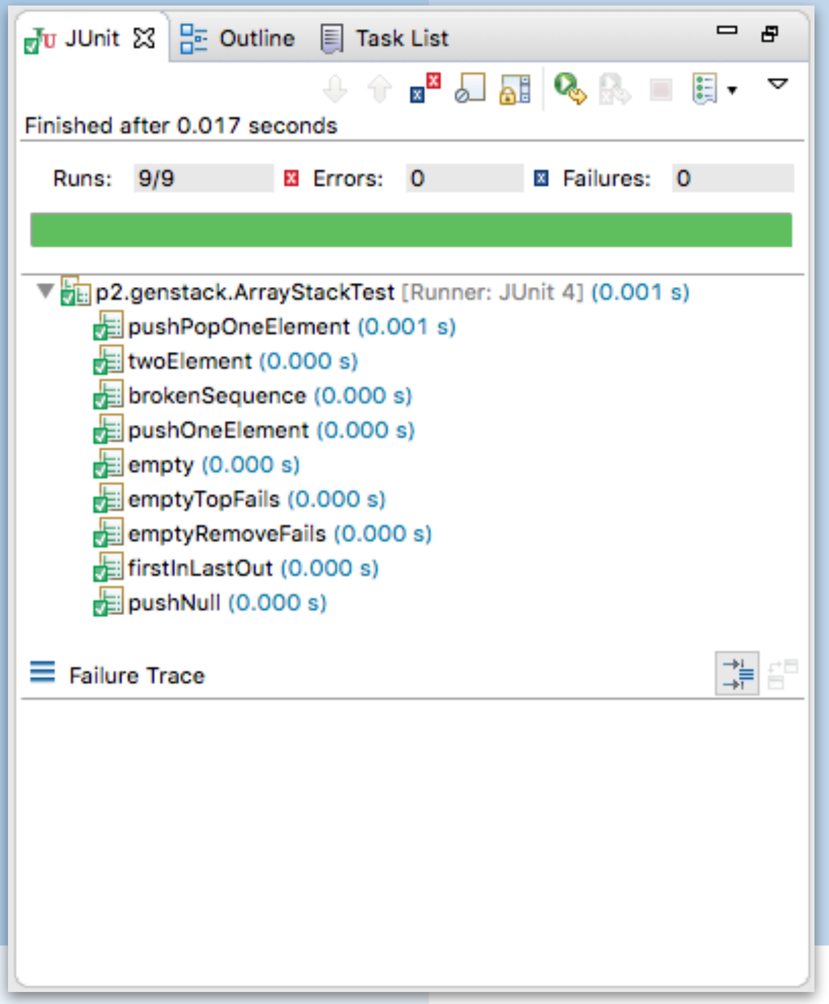
4. A Testing Framework

Oscar Nierstrasz



b
UNIVERSITÄT
BERN

A Testing Framework

Sources

> JUnit documentation (from www.junit.org)

Roadmap

- Junit — a testing framework
- Testing an interface
- Testing an algorithm

Roadmap

> **Junit — a testing framework**
  - Testing practices
  - Frameworks vs. Libraries
  - Junit 3 vs. Junit 4 and 5 (annotations)

> Testing an interface

> Testing an algorithm

The Problem

> *"Testing is not closely integrated with development. This prevents you from measuring the progress of development — you can't tell when something starts working or when something stops working."*
>
> *— "Test Infected", Beck & Gamma, 1998*

Interactive testing is tedious and seldom exhaustive.

Automated tests are better, but,

- —how to introduce tests interactively?
- —how to organize suites of tests?

Note that the “Test Infected” article was written in 1998. Since then, the Unit Testing approach promoted by Beck and Gamma has had a huge influence on software development, and testing is much better integrated into model development processes.

Tests should be repeatable, deterministic and automated.

Testing Practices

During Development

> When you need to add new functionality, *write the tests first.*
  — You will be done when the test runs.

> When you need to redesign your software to add new features, refactor in small steps, and *run the (regression) tests after each step.*
  — Fix what's broken before proceeding.

During Debugging

> When someone discovers a defect in your code, *first write a test* that demonstrates the defect.
  — Then debug until the test succeeds.

"Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead."
Martin Fowler

Test-Driven Development (TDD) is a practice in which tests are written before any functional code is written. One of the advantages of TDD is that writing the tests first influences the design of the code: it makes clear what functional interfaces are needed, and also ensures that the design supports testing.

JUnit - A Testing Framework

> JUnit is a simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks written by Kent Beck and Erich Gamma

The first unit testing framework was SUnit, written for Smalltalk. Beck and Gamma later ported the design to Java, and since then xUnit frameworks have been developed for most mainstream programming languages.

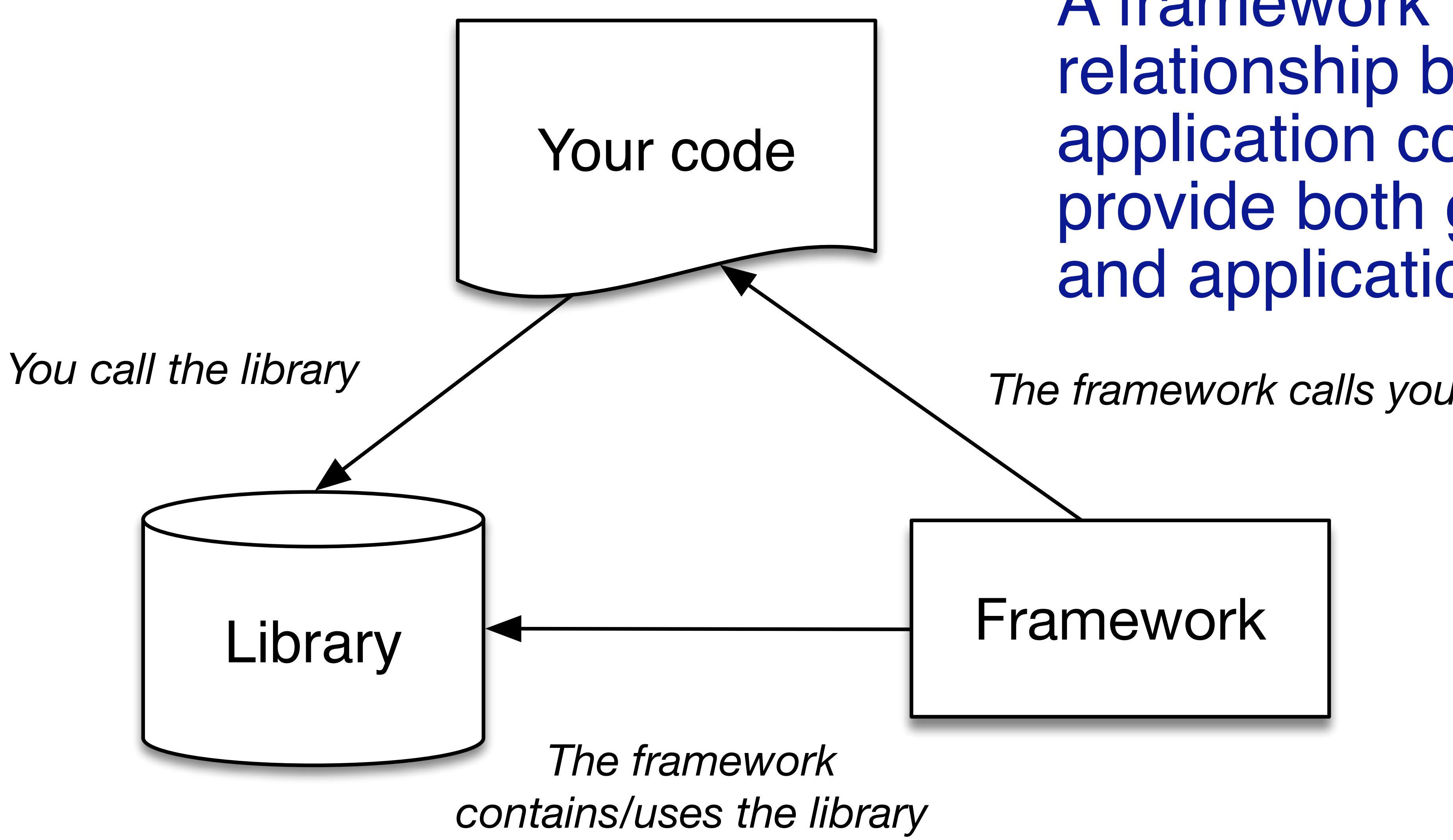
JUnit documentation can be found here:

`junit.sourceforge.net/doc/cookbook/cookbook.htm`

Frameworks vs. Libraries

In traditional application architectures, user code makes use of library functionality in the form of procedures or classes.

A framework *reverses* the usual relationship between generic and application code. Frameworks provide both generic functionality and application architecture.



Essentially, a framework says: "Don't call me — I'll call you."

The word “framework” suggests a skeleton that can be filled in with details.

The difference between a library and a framework lies in the fact that a framework represents a complete application, with only certain functional details missing. As a result, the framework is in control: you don't call the framework, the framework calls you.

JUnit 3.8

JUnit is a simple “testing framework” that provides:

> classes for writing *Test Cases and Test Suites*
> methods for *setting up and cleaning up test data* (“fixtures”)
> methods for *making assertions*
> textual and graphical tools for *running tests*

JUnit is a framework in the sense that it provides all the infrastructure needed to organize and run tests. The “missing details” to be filled in consist of the concrete tests that you must provide.

JUnit 3.8 is a classical *object-oriented framework* that relies on *inheritance*: JUnit provides a set of interacting classes, and you provide *subclasses* of the framework classes that provide the actual tests to be run.

JUnit 4 and later versions are *component-based frameworks* that instead rely on pluggable interfaces and annotations to tailor the framework. Since a lot of Java unit tests are based on the older framework, it is important to understand both approaches.

Failures and Errors

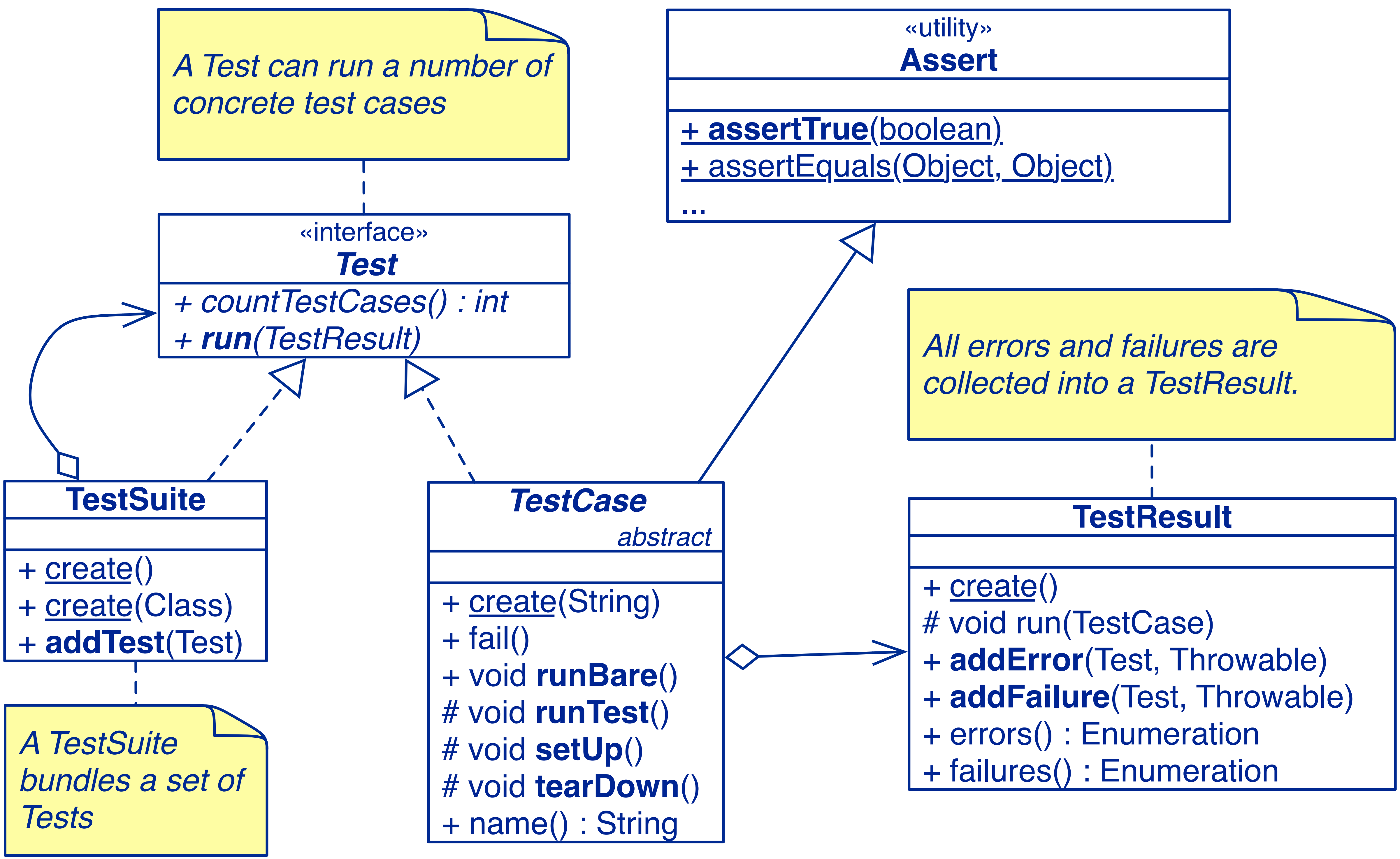
JUnit distinguishes between *failures and errors:*

- A failure is *a failed assertion*, i.e., an anticipated problem that you test.
- An error is *a condition you didn't check for*, i.e., a runtime error.

Note the similarity to the terminology that we saw in the lecture on Design by Contract, but also the differences. Both *failures* and *errors* cause *exceptions* to be raised, but here a failure refers specifically to a *failed test* (assertion), i.e., something you explicitly test for. An error, on the other hand, refers to something you didn't test. JUnit keeps track of both kinds of exceptions.

Assertions, as in DbC, are *predicates that are assumed to hold*. If an assertion fails, this means there is a defect (bug) in the code (or perhaps in the test). JUnit provides a rather richer set of assertion methods than basic Java, so we can produce more informative error messages.

The JUnit 3 Framework



There are many more classes in JUnit than are shown here, but these are the essential ones. Based on this design, it is easy to implement a basic *x*Unit framework for your favourite programming language *x*.

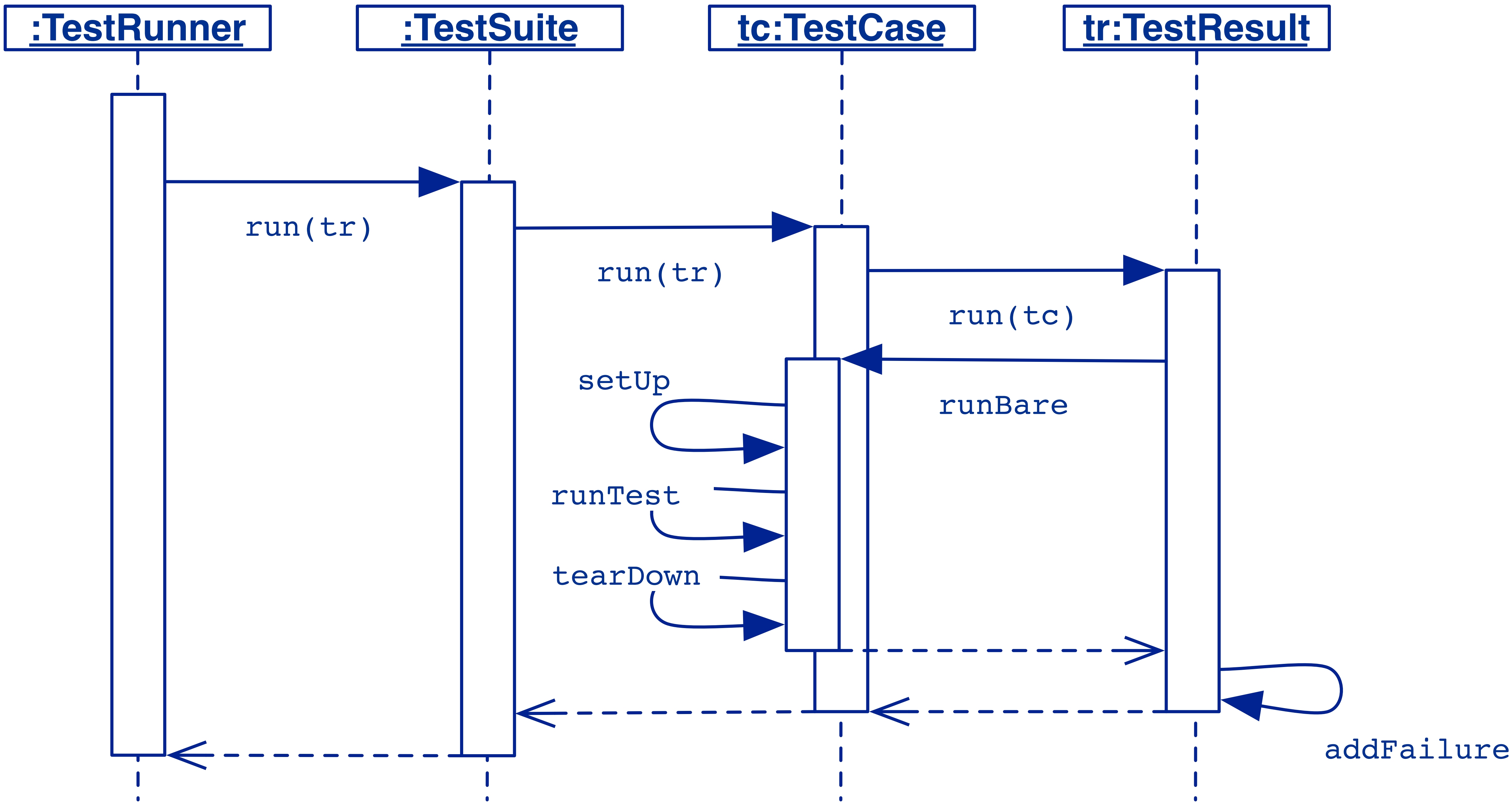
`TestCase` is the most important class: to tailor the framework, simply define a subclass of `TestCase` that defines a number of methods named "test…". The framework will automatically collect these into a test suite that can be run. Both test cases and test suites support the `Test` interface.

NB: This is a classic example of the **Composite** design pattern; `TestSuite`s are composed of nested `TestCase`s, and both can be tested through a common interface.

`Assert` is a utility class (i.e., a library) of useful assert methods.

`TestResult` is the class that actually runs the tests within try-catch statements, so it can catch all the failures and errors into a report.

A Testing Scenario



The framework calls the test methods that you define for your test cases.

Here we see clearly how the various classes interact: a `TestRunner` runs one or more `TestSuites`. A `TestSuite` simply runs each `TestCase` it contains. The `TestRunner` instantiates a `TestResult` object (tr) and passes it along to collect the results. A `TestCase` asks the `TestResult` object to run itself and collect any failures or errors. Note that it is the *same* `TestResult` object that runs all the tests.

Each individual `TestCase` object may additional specify a `setUp` method to prepare the test (e.g., to create needed objects for the test) and a `tearDown` method to clean up afterwards, if necessary.

JUnit 3 Example Code

```
import junit.framework.TestCase;
public class LinkStackTest extends TestCase {
   protected StackInterface<String> stack; // test data
   protected int size;

   protected void setUp() throws Exception {
      super.setUp();
      stack = new LinkStack<String>();
   }

   public void testEmpty() {
      assertTrue(stack.isEmpty());
      assertEquals(0, stack.size());
   }
…
}
```

Here we see the central idea of object-oriented frameworks.

We specialize the JUnit framework by defining `LinkStackTest` as a subclass of `TestCase`. Even though JUnit does not and cannot know our specific test classes, we can *plug* `LinkStackTest` into JUnit because it is a subclass of a framework class. It therefore satisfies the `TestCase` interface and also inherits any useful methods from it.

`LinkStackTest` defines specific test methods, such as `testEmpty`, and also a `setUp` method to prepare the test data. (We don't need a `tearDown` method here.)

Annotations in J2SE 5

- J2SE 5 introduced the Metadata feature (data about data)
- Annotations allow you to add decorations to your code (remember javadoc tags: *@author*)
- Annotations are used for code documentation, compiler processing (@Deprecated), code generation, runtime processing

http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html

JUnit 4 and 5

JUnit is a simple “testing framework” that provides:

> Annotations for marking methods as *tests*

> Annotations for marking methods that *setting up and cleaning up test data* (“fixtures”)

> methods for *making assertions*

> textual and graphical tools for *running tests*

JUnit 4 and later provide the same functionality as the earlier object-oriented framework, but it no longer relies on inheritance to plug concrete tests into the framework. Instead, annotations are used to flag test methods.

The current release (2021) is JUnit 5:

https://junit.org/junit5/

JUnit 5 Example Code

```
import junit.framework.TestCase;
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class LinkStackTest extends TestCase {
    protected StackInterface<String> stack;
    private int size;

    @BeforeEach public void setUp() {
       stack = new LinkStack<String>();
    }

    @Test public void testempty() {
       assertTrue(stack.isEmpty());
       assertEquals(0, stack.size());
    }
…
```

Test classes no longer need to inherit from a specific class. They just need to contain test methods.

Test methods not longer are required to be named “`test`…”. Instead we just need to annotate test methods with `@Test`. Setup methods do not need to be named “`setUp`”, but just need to be annotated as `@BeforeEach` methods.

Testing Style

> *"The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run."*

- write unit tests that *thoroughly test a single class*
- write tests *as you develop* (even before you implement)
- write tests for *every new piece of functionality*

"Developers should spend 25-50% of their time developing tests."

Roadmap

> Junit — a testing framework
  - Testing practices
  - Frameworks vs. Libraries
  - Junit 3.x vs. Junit 4.x (annotations)

> **Testing an interface**

> Testing an algorithm

What to test?

- Test every *public method* (test the interface)
- Test *boundary conditions*
- Test *key scenarios*
- Test *exceptional scenarios*
- Test every *line of code*
- Test every *path* through the code

There are two very general strategies to writing tests. *Black-box testing* focuses on testing the interface to a class (i.e., without looking inside the code). *White-box testing* instead focuses on exercising all the code. Typically a combination of both approaches is needed.

By *testing every public method* in the interface of a class, we ensure that we test *everything of interest to a client*. Helper methods will be tested indirectly by testing the public methods that use them.

Many *bugs occur at the boundaries* of inputs, for example, at minimum or maximum values of ranges or collections. We therefore should construct tests that explicitly test these boundaries.

A scenario will test multiple methods in combination. With limited resources, *at least the most common scenarios* should be explicitly tested. If possible, unusual or *exceptional scenarios should also be tested*, as this is often where bugs arise.

Every line of code should be tested by at least one test. We may need to write *several tests with different data* to ensure that every line of code is reached.

Similarly checking that *every path through the code* is exercised may require multiple tests for a single method. Note that just by testing every line of code is tested we may not necessary pass through every possible path.

There also exist testing practices that are more specific to JUnit. Here is an interesting discussion on the topic:

http://www.kyleblaney.com/junit-best-practices/

Testing the StackInterface

Recall our stack interface from last lecture

```
public interface StackInterface<E> {
   public boolean isEmpty();
   public int size();
   public void push(E item);
   public E top();
   public void pop();
}
```

We will develop some tests to exercise all the public methods.

Testing public methods

`isEmpty()`	True when it’s empty; false otherwise (needs a push)
`size()`	Zero when empty; non-zero otherwise (needs a push)
`push()`	Possible any time; affects size and top
`top()`	Only valid if not empty; needs a push; returns the last element pushed
`pop()`	Only valid if not empty; needs a push first; affects size and top

Without looking at the implementation, we can already see that (i) we may need multiple tests for a given method, and (ii) some methods can only be tested in combination with others.

A LinkStackTest class

Start by setting initializing the fixture (`stack`)

```
import org.junit.jupiter.api.*;
import static org.junit.jupiter.api.Assertions.*;

public class LinkStackTest {
   protected StackInterface<String> stack;
   private int size;

   @BeforeEach public void setUp() {
      stack = new LinkStack<String>();
   }
…
}
```

Testing the empty stack

We can test both `isEmpty()` and `size()` with an initial stack

```
@Test public void empty() {
   assertTrue(stack.isEmpty());
   assertEquals(0, stack.size());
}
```

Alternatively, we could write two separate tests, one for each condition `assertTrue`, etc. are static imported methods of the `Assertions` class of the JUnit 5 Framework and raise an `AssertionError` if they fail.

Junit 3 raises a JUnit `AssertionFailedError` (!)

JUnit offers a wide range of assert methods that provide more informative error messages than the basic `assert(`*`boolean`*`)` method. For example, `assertEquals(x,y)` can report: "I expected `x` but I got `y`", whereas `assert(p)` can only tell us "`p` was not true".

Note that we use `assertTrue()` to test `isEmpty()`, but `assertEquals()` to test `size()`. In the first case, `isEmpty()` already returns a Boolean. We would get no advantage from testing:

```
assertEquals(false, stack.isEmpty()); // useless
```

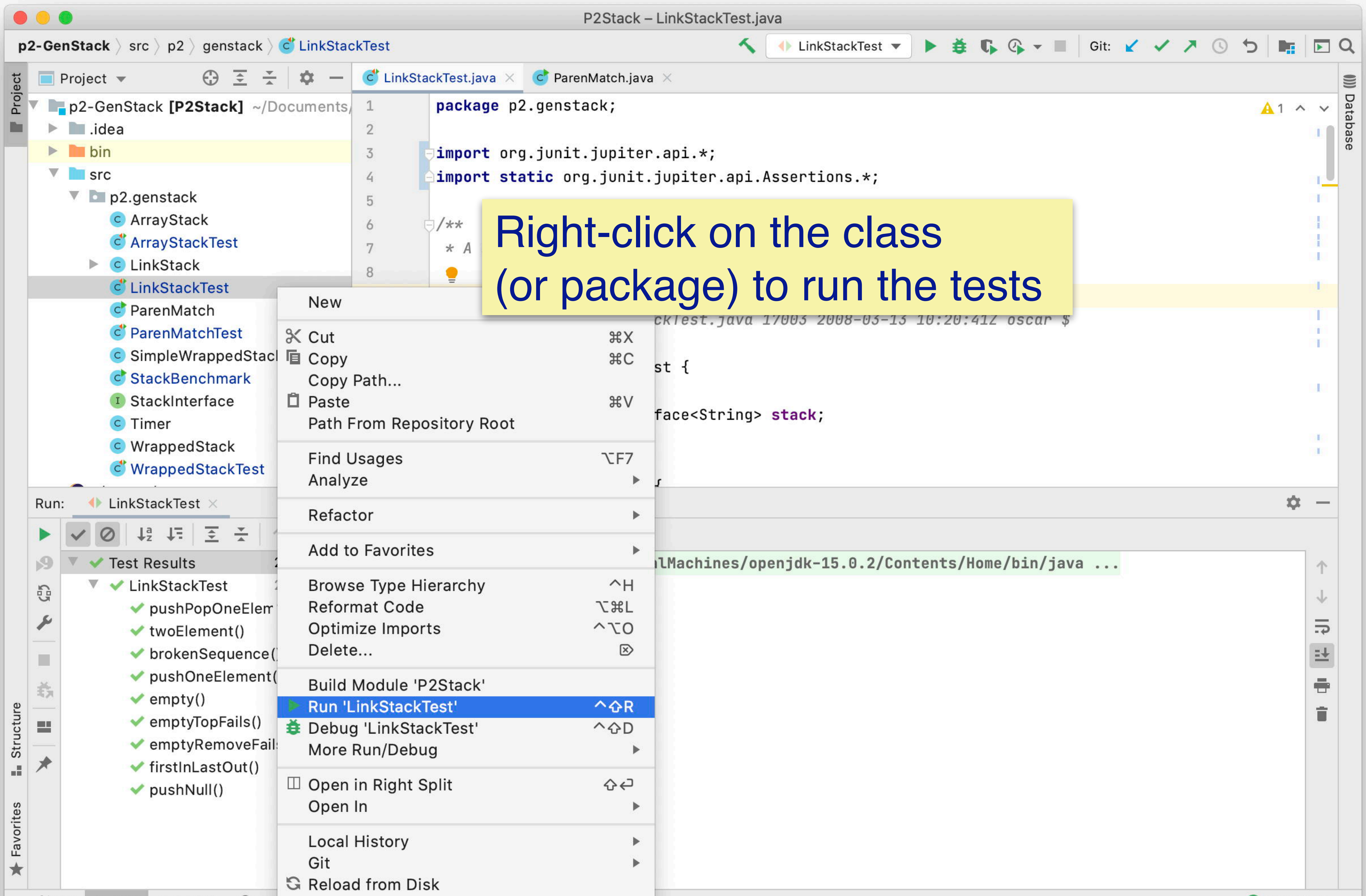
But in the second case `assertEquals()` provides us with useful additional information.

We would *lose information* were we to write:

```
assertTrue(stack.size() == 0); // bad style
```

since a failed assertion would no longer report that we *expected* the value 0.

Running tests from IntelliJ



Testing a non-empty stack

We modify the stack and test the new state:

```
@Test public void pushOneElement() {
      stack.push("a");
      assertFalse(stack.isEmpty());
      assertEquals(1, stack.size());
      assertEquals("a", stack.top());
}
```

Testing pop

We push and pop and test if the stack is empty.

```
@Test public void pushPopOneElement() {
      stack.push("a");
      stack.pop();
      assertTrue(stack.isEmpty());
      assertEquals(0, stack.size());
}
```

Testing top

We still need to test the “stack-like” behavior of top:

```
@Test public void twoElement() {
      stack.push("a");
      assertEquals("a", stack.top());
      stack.push("b");
      assertEquals("b", stack.top());
      stack.pop();
      assertEquals("a", stack.top());
      stack.pop();
      assertTrue(stack.isEmpty());
}
```

At this point we have minimally tested the entire stack interface

Without this test, a queue would also pass all the tests we have defined up to now. Here we are testing that `top` accurately returns the last element pushed, and will return previous elements pushed after a `pop`.

Testing boundary conditions

Bugs frequently occur at boundaries in the input.
These should be carefully tested.

A new report says eight new F-22 fighter jets like these experienced total computer failure when crossing the international date line. Kim Hong-Ji/Reuters

Testing boundary conditions

The only boundary value in the stack interface could be if `null` is pushed:

```
@Test public void pushNull() {
      stack.push(null);
      assertFalse(stack.isEmpty());
      assertEquals(1, stack.size());
      assertEquals(null, stack.top());
}
```

Testing for failure

A special kind of boundary condition is checking whether the class behaves as expected when the preconditions for a method do not hold.

```
@Test
public void emptyTopFails() {
  assertThrows(AssertionError.class, () -> stack.top());
}

@Test
public void emptyRemoveFails() {
  assertThrows(AssertionError.class, () -> stack.pop());
}
```

Note the elegant use of a Java 8 lambda to pass the call to stack.top() to the assertion method.

To accomplish the same in JUnit 3 is far more clumsy:

```
public void testEmptyTopFails() {
  try {
    stack.top();
    fail("Calling top() on an empty stack should fail");
  } catch (AssertionError e) {
    assertEquals(null, e.getMessage());
  }
}
```

Here we must explicitly run the faulty code in a try-catch clause, fail if the code passes, and pass if it fails!

Testing a key scenario

We should also test a more complex scenario that exercises the interaction between methods:

```
@Test
public void firstInLastOut() {
      stack.push("a");
      stack.push("b");
      stack.push("c");
      assertEquals("c", stack.top());
      stack.pop();
      assertEquals("b", stack.top());
      stack.pop();
      assertEquals("a", stack.top());
      stack.pop();
      assertTrue(stack.isEmpty());
}
```

Testing an exceptional scenario

```
@Test
public void brokenSequence() {
        stack.push("a");
        stack.pop();
        assertThrows(AssertionError.class,
           () -> stack.pop());
}
```

Roadmap

> Junit — a testing framework
  - Testing practices
  - Frameworks vs. Libraries
  - Junit 3.x vs. Junit 4.x (annotations)
> Testing an interface
> **Testing an algorithm**

Testing the parenMatch algorithm

To cover every line of code, we must reach all the bold lines:

```
public boolean parenMatch() {
  for (int i=0; i<line.length(); i++) {
    char c = line.charAt(i);
    if (isLeftParen(c)) {
      stack.push(matchingRightParen(c)); // (1)
    } else {
      if (isRightParen(c)) {
        if (stack.isEmpty()) { return false; } // (2)
        if (stack.top().equals(c)) {
          stack.pop(); // (3)
        } else { return false; } // (4)
      } // else not a paren char (5)
    }
  }
  return stack.isEmpty(); // (6)
}
```

Note that covering every line of code is not necessarily the same as covering every possible path through the code. We can have a path that goes through point (5), but there is no line of code there!

Instantiating paren matchers

We need an easy way to create a new paren matcher for a given test case:

```
public class ParenMatchTest {
  protected ParenMatch pm;

  protected ParenMatch makePm(String input) {
    return new ParenMatch(input, new LinkStack<Character>());
  }

  @Test
  public void empty() {
    pm = makePm("");
    assertTrue(pm.parenMatch());
  }
…
}
```

Which path is tested here?

If we had started writing tests earlier, we would perhaps have designed the ParenMatch class differently to better support tests.

It would be convenient to pass the string to check directly as an argument to the parenMatch method, which should also reset the internal stack to be empty.

Testing early can have a positive influence on the design process.

Path testing

```java
@Test public void balancedWithOtherChars() {
  pm = makePm("public void main() { return true; }");
  assertTrue(pm.parenMatch()); // (1) (3) (5) (6)
}
```

```java
public boolean parenMatch() {
  for (int i=0; i<line.length(); i++) {
    char c = line.charAt(i);
    if (isLeftParen(c)) {
      stack.push(matchingRightParen(c)); // (1)
    } else {
      if (isRightParen(c)) {
        if (stack.isEmpty()) { return false; } // (2)
        if (stack.top().equals(c)) {
          stack.pop(); // (3)
        } else { return false; } // (4)
      } // else not a paren char (5)
    }
  }
  return stack.isEmpty(); // (6)
}
```

How would you construct tests to pass through points (2) or (4)?

To reach point (2), we must have just read a right parenthesis, but the stack must be empty.

To reach point (4), we must have read a right parenthesis, but it does not match the top of the stack.

What test inputs could lead to these results?

What you should know!

- *How does a framework differ from a library?*
- *What is a unit test?*
- *What is an annotation?*
- *How does JUnit 3.x differ from JUnit 4.x*
- *What is a test "fixture"?*
- *What should you test in a test case?*
- *How can testing drive design?*

Can you answer these questions?

- How does the TestRunner invoke the right suite() method?
- How do you know when you have written enough tests?
- How many assertions should a test contain?
- Is it better to write long test scenarios or short, independent tests?