Testing with JUnit

This book explains JUnit concepts and best practices applied to the test-first approach.

From the beginning, you'll be guided through a practically relevant example and pick up background knowledge and development techniques every step of the way. Starting with the basics of tests organization, you'll soon comprehend the necessity of well-structured tests. In conjunction with third-party tools, you'll be trained in writing your tests efficiently, adapt your test case environment to particular demands and increase the expressiveness of your verification statements. Finally, you'll experience continuous integration as the perfect complement to support short feedback cycles and quality related reports for your whole team.

The tutorial gives a profound entry point in the essentials of unit testing with JUnit and prepares you for test-related daily work challenges.

Who this book is written for
No matter what your specific background as a Java developer is, unit testing is the way to go. This book provides you with a comprehensive, but concise entrance, advancing your knowledge, step-wise, to a professional level.

What you will learn from this book
- Organize your test infrastructure and resources reasonably
- Understand and write well-structured tests
- Decompose your requirements into small and independently testable units
- Increase your testing efficiency with on-the-fly generated stand-in components and deal with the particularities of exceptional flow
- Employ runners to adjust to specific test demands
- Use rules to increase testing safety and reduce Boilerplate
- Use third-party supplements to improve the expressiveness of your verification statements


Frank Appel

Master high-quality software development driven by unit tests

Prices do not include local sales tax or VAT where applicable
In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 6 'Reducing Boilerplate with JUnit Rules'
- A synopsis of the book’s content
- More information on Testing with JUnit
Frank Appel is a stalwart of agile methods and test-driven development in particular. He has over 2 decades of experience as a freelancer and understands software development as a type of craftsmanship. Having adopted the test first approach over a decade ago, he applies unit testing to all kinds of Java-based systems and arbitrary team constellations. He serves as a mentor, provides training, and blogs about these topics at codeaffine.com.
Preface

Testing with JUnit is a skill that presents much harder challenges than you might expect at first sight. This is because, despite its temptingly simple API, the tool plays ball with profound and well-conceived concepts. Hence, it's important to acquire a deep understanding of the underlying principles and practices. This avoids ending up in gridlocked development due to messed-up production and testing code.

Mastering high-quality software development driven by unit tests is about following well-attuned patterns and methods as a matter of routine rather, than reinventing the wheel on a daily basis. If you have a good perception of the conceptual requirements and a wide-ranging arsenal of solution approaches, they will empower you to continuously deliver code, which achieves excellent ratings with respect to the usual quality metrics out of the box.

To impart these capabilities, this book provides you with a well-thought-out, step-by-step tutorial. Foundations and essential techniques are elaborated, guided by a golden thread along the development demands of a practically relevant sample application. The chapters and sections are built on one another, each starting with in-depth considerations about the current topic’s problem domain and concluding with an introduction to and discussion of the available solution strategies.

At the same time, it’s taken care that all thoughts are backed up by illustrative images and expressive listings supplemented by adequate walkthroughs. For the best possible understanding, the complete source code of the book’s example app is hosted at https://github.com/fappel/Testing-with-JUnit. This allows you to comprehend the various aspects of JUnit testing from within a more complex development context and facilitates an exchange of ideas using the repository’s issue tracker.
**Preface**

**What this book covers**

*Chapter 1, Getting Started,* opens with a motivational section about the benefits of JUnit testing and warms up with a short coverage of the toolchain used throughout the book. After these preliminaries, the example project is kicked off, and writing the first unit test offers the opportunity to introduce the basics of the test-driven development paradigm.

*Chapter 2, Writing Well-structured Tests,* explains why the four fundamental phases' test pattern is perfectly suited to test a unit's behavior. It elaborates on several fixture initialization strategies, shows how to deduce what to test, and concludes by elucidating different test-naming conventions.

*Chapter 3, Developing Independently Testable Units,* shows you how to decompose big requirements into small and separately testable components and illustrates the impact of collaboration dependencies on testing efforts. It explains the importance of test isolation and demonstrates the use of test doubles to achieve it.

*Chapter 4, Testing Exceptional Flow,* discusses the pros and cons of various exception capture and verification techniques. Additionally, it explains the meaning of the *fail fast* strategy and outlines how it intertwines with tests on particular boundary conditions.

*Chapter 5, Using Runners for Particular Testing Purposes,* presents JUnit's pluggable test processor architecture that allows us to adjust test execution to highly diverse demands. It covers how to write custom runners and introduces several useful areas of application.

*Chapter 6, Reducing Boilerplate with JUnit Rules,* unveils the test interception mechanism behind the rule concept, which allows you to provide powerful, test-related helper classes. After deepening the knowledge by writing a sample extension, the chapter continues with the tools' built-in utilities and concludes by inspecting useful third-party vendor offerings.

*Chapter 7, Improving Readability with Custom Assertions,* teaches the writing of concise verifications that reveal the expected outcome of a test clearly. It shows how domain-specific assertions help you to improve readability and discusses the assets and drawbacks of the built-in mechanisms, Hamcrest and AssertJ.

*Chapter 8, Running Tests Automatically within a CI Build,* concludes the example project with important considerations of test-related architectural aspects. Finally, it rounds out the book by giving an introduction to continuous integration, which is an excellent brief of the *test first* approach and establishes short feedback cycles efficiently by automation.

Appendix, References, lists all the bibliographic references used throughout the chapters of this book.
We already explained how to minimize clutter and redundancy in test cases by extracting coherent utility code into separate test helpers, as seen in Chapter 3, Developing Independently Testable Units. In this chapter, you'll learn how to enhance this strategy by means of JUnit rules. You'll be given an introduction to the approach's mechanics and potential. To deepen this knowledge, we'll continue by writing and varying a sample extension. Then, we'll examine advanced features of rules-related functionality and conclude with the inspection of useful third-party vendor offerings. In short, this will make us capable of the following:

- Understanding rules
- Working with advanced concepts
- Employing custom solutions

Understanding rules
The benefits of JUnit rules, especially when dealing with integration tests, can hardly be overrated. That's why, we start with a thorough explanation of how they work and how you can implement your own extensions. This will put you in a position to increase your daily work efficiency with the aid of custom rules.
What are JUnit rules?

"Pedantry and mastery are opposite attitudes toward rules."

– George Pólya

JUnit offers a particular test helper support that we’ve already encountered in the course of this book. Recollect the introduction of the ExpectedException rule in Chapter 4, Testing Exceptional Flow, where you got a first impression of the possible applications. But what are rules exactly?

To answer this question, let’s take a look at an example. Until now, we’ve concentrated our efforts on unit tests, which use DOCs as stand-in components to isolate SUTs from their dependencies. But at some point in time, you’ll have to turn towards these dependencies, provide real-world implementations, and verify their correct behavior. This is when you write tests that incorporate costly system calls and third-party libraries and potentially span over several application layers—in short, integration tests.

As you’ve already learned, these are important flanking activities to separate our application code—the code we are in control of—from extrinsic influences. Unfortunately, this test type is considerably more expensive than a unit test because it has to deal with things such as environmental settings, framework startup, and last but not least, housekeeping.

Assume that we’ve decided to store our timeline state on the local filesystem. Hence, we’d need an appropriate SessionStorage realization. It seems natural to develop such FileSessionStorage against a proper integration test, but this involves dealing with tedious filesystem demands. First, there is the determination of a platform-independent storage location, and second, we have to clean up the remains after a test run.

Obviously, it's a pretty common challenge that cries out for a test helper. For this reason, JUnit provides the TemporaryFolder class. To avoid running into file access privilege trouble, it makes sense to choose storage locations for test output below the system's temporary directory. As the name implies, the helper offers several API methods for the creation of files or directories (below the temporary directory root) and a cleanup functionality, delete. The latter removes all content from the disk that has been originated by a temporary folder instance.
From what we've learned until now, we'd probably expect to use this helper as follows:

1. Create the storage location during test setup.
2. Call the delete method in an @After annotated method to do the housekeeping.

But since TemporaryFolder is a rule, the second step isn't necessary. The only thing we need to do is to register the rule correctly. Regard the SessionStorage interface and a simplistic Memento type sufficient for our considerations here. The storage reads and writes state mementos. The following code shows this:

```java
public interface SessionStorage {
    void store( Memento memento );
    Memento read();
}
```

The state holder itself oversimplifies serialization a bit by employing a parameterized constructor and the toString method for this purpose. But this shouldn’t bother us in the context of the current topic. We’ll refine this approach in Chapter 7, Improving Readability with Custom Assertions. The following code shows this:

```java
public class Memento {
    private String content;

    public Memento( String content ) {
        this.content = content;
    }

    @Override
    public String toString() {
        return content;
    }

    [...]
}
```

Let’s apply our main attention to the following FileSessionStorageITest listing instead. As you can see, the temporary folder instance is created implicitly during field initialization. The temporaryFolder public field is annotated with @Rule. This is how rules have to be registered. But before we explain the reason for this, we’ll complete the test case’s examination first:

```java
public class FileSessionStorageITest {

    private static final String CONTENT = "content";
```
@Rule
public TemporaryFolder temporaryFolder = new TemporaryFolder();

private FileSessionStorage storage;
private File storageLocation;

@Before
public void setUp() throws IOException {
    storageLocation = temporaryFolder.newFile();
    storage = new FileSessionStorage( storageLocation );
}

@Test
public void store() throws IOException {
    Memento memento = new Memento( CONTENT );
    storage.store( memento );
    assertEquals( CONTENT, readStoredContent() );
}

@Test
public void read() throws IOException {
    writeContentToStore( CONTENT );
    Memento memento = storage.read();
    assertEquals( CONTENT, memento.toString() );
}

[...]

private String readStoredContent() throws IOException {
    byte[] bytes = Files.readAllBytes( storageLocation.toPath() );
    return new String( bytes, StandardCharsets.UTF_8 );
}

private Path writeContentToStore( String content ) throws IOException {
    byte[] bytes = content.getBytes( StandardCharsets.UTF_8 );
    return Files.write( storageLocation.toPath(), bytes );
}
The basic idea is to use the file session storage to serialize a memento to a content string and store this on the local filesystem. Afterwards, we load the actual content from the file and check that it matches the expected one. For the verification of the read functionality, we proceed conversely. See how we rely always on the storage location provided by our test helper.

It is noteworthy that the store and read utility methods may indicate a code redundancy with respect to the component's implementation. We could get rid of this duplication using the storage itself to complete the particular turnarounds. Nevertheless, it seemed more plausible to have them here for initial understanding. Furthermore, we assume that the storage file already exists. Among other things, it would be worthwhile to verify how the storage deals with a situation where the file doesn't exist. So, feel free to refactor and complement the preceding listing appropriately as an exercise.

Now, it's about time to focus our attention on the rule mechanics. Rules provide a possibility of intercepting test method calls similar as an AOP framework would do. Comparable to an around advice in AspectJ, the test method gets embedded dynamically into a code block. This allows the inserting of useful functionality before and/or after the actual test execution. It even enables you to skip the original call completely. Thus, the temporary folder instance is able to delete its files after a test run automatically.

But bear in mind that a test run in that regard includes the @Before and @After annotated callbacks because they constitute implicitly executed test phases. The rule clamp has to embrace these invocations, which in turn, explains why the rule's field initialization takes place at construction time. The rule simply has to exist before any testing-related action happens, to avoid interleaving. The tool creators probably found this manner sufficient and refrained from introducing, for example, another callback type.

While these abstract explanations surely sound reasonable, the best way to obtain a profound understanding is to write a rule by yourself. Although you might anticipate this to be complicated, it's actually a pretty simple thing to do.
Writing your own rule

The API part of a rule definition has to implement `TestRule`. The only method of this interface is `apply`, which returns an instance of `Statement`. `Statement` represents—simplistically spoken—your tests within the JUnit runtime and `Statement#evaluate()` executes them. Inspecting the argument list of `apply`, we recognize that a statement is also given as an input. The basic thought is to provide a wrapper extension of `Statement`, which can implement additional contributions by overriding `evaluate`:

```java
public class MyRule implements TestRule {
    @Override
    public Statement apply( Statement base, Description description ) {
        return new MyStatement( base );
    }
}

class MyStatement extends Statement {
    private final Statement base;

    MyStatement( Statement base ) {
        this.base = base;
    }

    @Override
    public void evaluate() throws Throwable {
        System.out.println( "before" );
        try {
            base.evaluate();
        } finally {
            System.out.println( "after" );
        }
    }
}
```

The preceding listing shows how a statement adapter works. Embedding the delegating `evaluate` call into a try-finally block ensures that, no matter what happens during the invocation, the console output 'before' and 'after' gets written. `MyRuleTest` confirms that our custom extension `MyRule` can be used in the same way as `TemporaryFolder`:

```java
public class MyRuleTest {

    @Rule
```
public MyRule myRule = new MyRule();

@Test
public void testRun() {
    System.out.println( "during" );
}

Launching the test case leads to the output depicted in the following image. This proves that our example rule works as expected. The test execution gets intercepted and modified by our rule to print before and after around the during of the test method.

Doing something before and after a test run is particularly typical for rules dealing with external resources, such as files, sockets, servers, database connections, and so on. So, it isn't surprising that there is a common super class for such use cases. The current example could be rewritten by extending ExternalResource, as indicated here:

    public class MyRule extends ExternalResource {

        @Override
        protected void before() {
            System.out.println( "before" );
        }

        @Override
        protected void after() {
            System.out.println( "after" );
        }
    }

Now that the very basics of rule development have been understood, we'll go a step further and cover a more advanced, yet very popular, pattern.
Reducing Boilerplate with JUnit Rules

Configuring the fixture with annotations

Up to this point, we apply rules if we want to take care of a certain aspect that relates to all tests of a test case. Special fixture adjustments are done within each test method by calling the rule's helper methods. But sometimes, these configurations may not add much to the comprehension of the test's purpose or an aspect may not be related to all tests.

In these instances, it might be appropriate to combine a rule with an additional control annotation to reduce, for example, the clutter within a test. Remember the @Mock annotation mentioned in the preceding chapter? We've heard about the MockitoRule picking up marked fields and created test double instances on the fly—before the actual test execution started.

Later, on we'll encounter another useful application, but first, let's examine how we can evolve MyRule to enable test-specific settings by means of annotations. To do so, we introduce a method-related annotation, MyRuleConfiguration, which accepts a string value as a parameter:

```java
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface MyRuleConfiguration {
    String value() default "";
}
```

It's possible to access test method annotations by the Description argument, which we haven't used until now. This allows us to supplement the console output with the specified configuration value. The MyConfigurableRule variant here illustrates the details:

```java
public class MyConfigurableRule implements TestRule {

    @Override
    public Statement apply( Statement base, Description description ) {
        return new MyConfigurableStatement( base, description );
    }
}
```

```java
class MyConfigurableStatement extends Statement {

    private final Description description;
    private final Statement base;
```
MyConfigurableStatement( Statement base,
    Description description )
{
    this.description = description;
    this.base = base;
}

@Override
public void evaluate() throws Throwable {
    String configuration = getConfiguration();
    System.out.println( "before [" + configuration + "]" );
    try {
        base.evaluate();
    } finally {
        System.out.println( "after [" + configuration + "]" );
    }
}

private String getConfiguration() {
    return description
        .getAnnotation( MyRuleConfiguration.class )
            .value();
}

This is all we have to do to bring MyRuleConfiguration into action. In the next
snippet, MyConfigurableRuleTest shows its appropriate use:

[ Note the additional annotation on the testRun method. ]

public class MyConfigurableRuleTest {

    @Rule
    public MyConfigurableRule myConfigurableRule
        = new MyConfigurableRule();

    @Test
    @MyRuleConfiguration( "myConfigurationValue" )
    public void testRun() {
        System.out.println( "during" );
    }
}
In the end, the console output will respect the configuration value, as displayed in the following image:

![Console Output](image)

The MyRuleTest console output with annotation

Being able to write our own rules gives us a pretty good insight into how they are working. All the more, it's interesting to learn what some out-of-the-box implementations are capable of.

**Working with advanced concepts**

JUnit comes with a set of readymade rules, taking away some of the burden of common test-related development tasks, and we’ve already encountered a few of them. But the tool offers additional capabilities, allowing us to apply and combine rules in a way so as to meet special requirements, at which we'll have a look in the following section.

**Using ClassRules**

At times, integration tests need access to external resources, which can be expensive to establish. If these resources do not contribute more to the precondition of any test, rather than being an environmental invariant, testing individual creations and disposing of them might be a waste.

Think of an application server necessary as the infrastructure to provide the REST services you intend to validate. The application server does not contribute any test-specific state. It simply has to be there to be able to deploy, test, and undeploy REST resources on the fly. Because of this, it would be desirable to perform the following actions:

1. Start the server *once* before the service tests are executed.
2. Stop it as soon as these tests are done.
JUnit supports this with the aid of class rules. These are public, static fields of test cases annotated with @ClassRule. Eagerly created, they also have to be a subtype of TestRule. They can affect the operation of a whole test case, which means the execution of the complete sequence of its tests is embedded into a rule's Statement#evaluate call.

For better understanding, let's sketch the preceding application server use case. Consider ServerRule, which is responsible for starting and stopping a server instance. It would extend ExternalResource and may accept a port as a constructor argument. We do not really work with a server here but rather indicate the life cycle events with console output messages:

```java
public class ServerRule extends ExternalResource {

    private final int port;

    public ServerRule( int port ) {
        this.port = port;
    }

    @Override
    protected void before() throws Throwable {
        System.out.println( "start server on port: " + port );
    }

    @Override
    protected void after() {
        System.out.println( "stop server on port: " + port );
    }
}
```

Moreover, we provide MyServerTest to represent a set of tests that expects the server instance to be up and running. Again, we indicate this only with messages written to the console.

```java
public class MyServerTest {

    @Rule
    public TestName name = new TestName();

    @Test
```

Note how we use the TestName rule to determine the name of a test.
Reducing Boilerplate with JUnit Rules

```java
public void runFirstServerTest() {
    System.out.println( name.getMethodName() );
}

@Test
public void runSecondServerTest() {
    System.out.println( name.getMethodName() );
}
```

Due to the fact that any subclass of `ParentRunner` will support class rules, it's feasible to aggregate all server-dependent test cases. `ServerIntegrationTestSuite` uses the `ClasspathSuite` runner introduced in Chapter 5, Using Runners for Particular Testing Purposes, to do so:

```java
@RunWith( ClasspathSuite.class )
@ClassnameFilters( { ".*ServerTest" } )
public class ServerIntegrationTestSuite {

    @ClassRule
    public static ServerRule serverRule = new ServerRule( 4711 );
}
```

The suite will first "start" our server, then pick up any test cases prefixed with `ServerTest`, run all tests of these classes, and finally "stop" the server. The following screenshot of a test launch's output confirms the expected behavior:

![ServerIntegrationTestSuite output](image)

Although this approach has its advantages, there is a little downside too. Using class rules on suites, as explained previously, leads to a situation where a single test case loses its autonomy. It implies that running `MyServerTest` standalone isn't possible anymore. But you might have a lot of server-related test cases. Then, the overhead of starting and stopping the server for each of those separately might be too high, which, in turn, can justify this kind of suite solution.
The ordering of rule execution

Of course, it is feasible to employ more than one rule in a test case. In this instance, the rules' statement adapters get nested. However, in which sequence this will happen is undetermined. But sometimes, correct ordering is crucial. Consider, for example, a rule that depends on our server rule. It might deploy a service needed by all of the suite's aggregated tests. As a consequence, we would run into an error if the server isn't started first.

For such requirements, JUnit provides the RuleChain utility. Being an implementation of TestRule, it's registered just as any other rule. It allows you to compound rules in a way that defines their nesting order. The following listing enhances our server integration test suite to illustrate the usage:

```java
@RunWith(ClasspathSuite.class)
@ClassnameFilters( { ".*ServerTest" } )
public class ServerIntegrationTestSuite {

    @ClassRule
    public static TestRule chain = RuleChain
        .outerRule( new ServerRule( 4711 ) )
        .around( new MyRule() );
}
```

Rule chains are configured fluently. The outerRule static method marks the start of a chain, and each around call adds a nested rule instance. Running the suite and observing the console reveals how the MyRule-related messages are clamped within the ServerRule output but spans the succession of tests. The following screenshot shows this:

![The MyServerTest console output with rule chaining](image)

For a complete list and description of all built-in rules, please refer to the eponymic section of the JUnit documentation wiki at https://github.com/junit-team/junit/wiki/Rules.
Reducing Boilerplate with JUnit Rules

Now, we'll continue with the introduction of useful little helpers provided by third-party vendors.

Employing custom solutions

Given the variety of possible applications, it's no wonder that there are custom rule solutions for both routine and fancy demands. Because of this, the last section of the chapter will dwell on noteworthy third-party implementations for each area.

Working with system settings

A source of constant burden is environment-specific functionality and state handling. Most of it is made accessible via the java.lang.System class. Setting and resetting system properties, capturing system output and so on produces quite a bit of overhead of boilerplate code. Luckily, a third-party utility called System Rules, [SYSRUL], comes to the rescue. It encapsulates the redundancies and supplies several rules to accomplish the various tasks efficiently.

Let's have a quick look at some of the more common challenges and see how system rules will help. If you need to ensure that a particular system property is removed before test execution, you can use the ClearSystemProperty test helper. This rule deletes a property before a test run and restores the original value after a test run. The following example clears the predefined location of the temporary directory:

```java
public class ClearPropertiesExample {
    private static final String JAVA_IO_TMPDIR = "java.io.tmpdir";

    @Rule
    public final ClearSystemProperties clearTempDirRule
    = new ClearSystemProperties( JAVA_IO_TMPDIR );

    @Test
    public void checkTempDir() {
        assertNull( System.getProperty( JAVA_IO_TMPDIR ) );
    }
}
```

Maybe more frequently, you want to define or override a particular system property for all tests. This can be achieved with the ProvideSystemProperty rule. Again, after a test run, the original value gets restored. The next example illustrates how you could alter (falsify) the value of the temporary directory property:

```java
public class ProvideSystemPropertyExample {

    @Rule
    public final ProvideSystemProperties provideTempDirRule
    = new ProvideSystemProperties( JAVA_IO_TMPDIR, "C:\tmp" );

    @Test
    public void checkTempDir() {
        assertEquals( "C:\tmp", System.getProperty( JAVA_IO_TMPDIR ) );
    }
}
```
private static final String JAVA_IO_TMPDIR = "java.io.tmpdir";
private static final String MY_TMPDIR = "/path/to/my/tmpdir";

@Rule
public final ProvideSystemProperty provideCustomTempDirRule
    = new ProvideSystemProperty( JAVA_IO_TMPDIR, MY_TMPDIR );

@Test
public void checkTempDir() {
    assertEquals( MY_TMPDIR,
        System.getProperty( JAVA_IO_TMPDIR ) );
}
}

But presumably, the most interesting is the RestoreSystemProperties helper. With this in place, you can simply set or change any property during a test. After a test run, the original values get restored. As this usage simply requires an appropriate rule definition, we omit any code snippets here.

Another common use case is that of capturing content that gets written to System.out or System.err print streams. For this purpose, the SystemOutRule and SystemErrRule rules are made available. They provide several modes configurable by the methods enableLog, mute, and muteForSuccessfulTests. The first one activates the capturing of content, the second prevents write through to the original stream. The last one does the same but allows messages to pass in the event of a test failure. The following snippet shows how to capture output but suppresses message depiction for successful tests:

```java
public class CaptureSystemOutputExample {

    private static final String OUTPUT = "output";

    @Rule
    public final SystemOutRule systemOutRule
        = new SystemOutRule().enableLog().muteForSuccessfulTests();

    @Test
    public void captureSystemOutput() {
        System.out.print( OUTPUT );

        assertEquals( OUTPUT, systemOutRule.getLog() );
    }
}
```
The last system helper we'll examine here allows the provision of text input. This can be read from the `System.in` input stream during a test. The `TextFromStandardInputStream` rule gets created by the `emptyStandardInputStream` factory method, as shown in the next code. It provides several methods to supply the content that should be dispatched to the input stream. The example uses `provideLines` to this end:

```java
public class ProvideSystemInputExample {

    private static final String INPUT = "input";

    @Rule
    public final TextFromStandardInputStream systemInRule = TextFromStandardInputStream.emptyStandardInputStream();

    @Test
    public void stubInput() {
        systemInRule.provideLines( INPUT );

        assertEquals( INPUT, readLine( System.in ) );
    }

    private String readLine( InputStream inputStream ) {
        return new Scanner( inputStream ).nextLine();
    }
}
```

For a complete list and profound description of all system rules, please refer to the library's documentation at [http://stefanbirkner.github.io/system-rules/index.html](http://stefanbirkner.github.io/system-rules/index.html).

We'll conclude this chapter with a utility that can come in handy when you run into trouble with platform-specific functionality for example.

**Ignoring tests conditionally**

JUnit offers the possibility to ignore single tests. To do so, you mark a test method additionally with `@Ignore`. Different than simply commenting or deleting `@Test`, runners will report the number of skipped tests and tag them appropriately within the UI result view. It's even feasible to record the reason why a test is being ignored using the annotation's optional string parameter.
But why should we wish to tear a hole in our safety net by ignoring a test? Well, basically we shouldn't. Nevertheless there are situations where we might opt for temporarily skipping certain tests. Think about nondeterministic tests for example, which fail only intermittently when running our complete suite of tests.

"The trouble with non-deterministic tests is that when they go red, you have no idea whether it's due to a bug, or just part of the non-deterministic behavior."

"Initially people will look at the failure report and notice that the failures are in non-deterministic tests, but soon they'll lose the discipline to do that. Once that discipline is lost, then a failure in the healthy deterministic tests will get ignored too."

– [FOWL11]

Thus, it's better to deactivate these tests temporarily, just in case we are not able to fix the problem on the spot. Fowler describes strategies on how to put such tests into quarantine. A low-level approach might include marking them with @Ignore. The advantage compared to deleting the @Test annotation is that ignored tests are reported and, hence, pop up as reminders of future work items.

However, there can be reasons that call for a more fine-grained control of when to skip a test. Some time ago, while working on an SWT-based UI, we ran into a platform-related issue. It turned out that on non-Windows platforms asserting whether an SWT widget has got the input focus does not work with automated tests. But we thought that to have a test up and running on one platform is better than nothing. Hence, we decided to ignore the focus-related tests on non-Windows systems for the time being.

In JUnit, assumptions are the built-in means to skip tests that aren't meaningful. Assume statements throw AssumptionViolatedException if a given condition isn't met. The default runner marks a test with an unfulfilled assumption as skipped. Have a look at the following snippet that illustrates the principle:

```java
public class AssumptionTest {

@Test
public void ignored() {
    Assume.assumeFalse( true );
    // statements below this line are skipped
}

@Test
public void executed() {
    Assume.assumeTrue( true );
    // statements below this line are executed
}
}
```
Reducing Boilerplate with JUnit Rules

The following screenshot shows how a test run with a failed assumption gets depicted in the UI. You can see clearly how the test named ignored is marked as skipped:

![Screenshot](image)

But this approach tends to mingle test code with unrelated aspects. It seems more natural to separate the decision as to whether a test is to be ignored from the test's body. This is the notion behind ConditionalIgnoreRule, which uses the @ConditionalIgnore annotation to reach this goal, [HERR13]. ConditionalIgnoreTest demonstrates the concept as follows:

```java
public class ConditionalIgnoreTest {

    @Rule
    public ConditionalIgnoreRule rule = new ConditionalIgnoreRule();

    @Test
    @ConditionalIgnore(condition = NotRunningOnWindows.class)
    public void focus() {
        // ...
    }
}

class NotRunningOnWindows implements IgnoreCondition {
    public boolean isSatisfied() {
        return !System.getProperty("os.name").startsWith("Windows");
    }
}
```
@ConditionalIgnore requires a condition argument, pointing to a class that implements IgnoreCondition. IgnoreConditionRule picks up the annotation at runtime as described previously. It creates an instance of the condition-defining type and decides, based on the result of its isSatisfied method, whether a test should be skipped.

If so, AssumptionViolatedException is thrown. Therefore the ConditionalIgnore annotation has basically the same effect that an unmet Assume condition would have. It has the slight advantage that @Before and @After callbacks are also skipped.

Now that you know how to stick to the rules, let's shortly recap what you've learned in this chapter.

Summary
In this chapter, you acquired profound knowledge of the mechanics and capabilities of JUnit rules. You saw what it takes to write your own extension and learned how to evolve self-made rules by means of annotations. After that, you were imparted the usage of class rules on test suites and discussed the pros and cons of this approach. Besides, you were shown how rules can be nested in case your tests depend on a well-defined execution order. To round out the topic, we introduce third-party vendor solutions for common testing-related tasks.

The next chapter is devoted to the various available verification concepts. It will put the built-in assert functionality on a test and explain the assets and drawbacks of the most popular alternatives.
Where to buy this book

You can buy Testing with JUnit from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.