Department of Computer Science

Fachhochschule Dortmund

University of Applied Sciences and Arts

Bachelor Thesis

for attaining the academic degree Bachelor of Science (B. Sc.)

in the course of studies Software Engineering (Softwaretechnik dual)

Compile-time abstraction of JavaScript mocking libraries powering a domain-specific language for interaction testing

Eine domänenspezifische Sprache für Interaktionstests basierend auf der Abstraktion verschiedener JavaScript-Mocking-Libraries zur Compile-Zeit

Tim Seckinger

born 1997-09-06

supervised by Prof. Dr. Martin Hirsch

Dortmund, 2018-09-16

Contents

Versicherung Ich versichere, dass ich die vorliegende Arbeit selbständig angefertigt und mich keiner fremden Hilfe bedient sowie keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle Stellen, die wörtlich oder sinngemäß veröffentlichten oder nicht veröffentlichten Schriften und anderen Quellen entnommen sind, habe ich als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Abstract

Spockjs is a testing tool that integrates with most modern JavaScript test runners and allows users to write their tests using the declarative 'block' style popularized by the *Spock Framework* for Groovy tests. Adding to the *assertion blocks* already implemented in the *domain-specific language* of spockjs, we develop another Spock Framework-inspired block type: *Interaction blocks*. To implement this new feature in a mocking library-agnostic way, we experiment with and evaluate two different approaches to distributing the work between compile time and runtime.

German Abstract

Spockjs ist ein Testing-Tool, das mit den meisten modernen JavaScript Test-Runnern verwendet werden kann und es Nutzern ermöglicht, ihre Tests in dem deklarativen 'Block'-Stil des *Spock Framework*s für Groovy-Tests zu schreiben. Zusätzlich zu den *Assertion-Blöcken*, die in der *domänenspezifischen Sprache* von spockjs bereits implementiert sind, wird ein weiterer durch das Spock Framework inspirierter Blocktyp entwickelt: *Interaction-Blöcke*. Um diese neue Funktionalität zu implementieren und dabei von der der verwendeten Mocking-Library unabhängig zu bleiben, wird mit zwei verschiedenen Ansätzen zur Verteilung der Arbeit zwischen Compile-Zeit und Laufzeit experimentiert und diese Ansätze werden bewertet.

1 Introduction

Elaborating on the information given in the abstract, this introductory chapter states the goal of the paper in the *goal* section after motivating it in the *motivation* section and providing some technical context in the *context* section; finally, it outlines the path we take throughout the chapters to achieve our goal in the *structure* section.

1.1 Motivation

In software development, automated tests are one of the key tools to achieve a high level of software quality. Writing tests efficiently itself requires good support from the programming language, standard library, or third party tools. The two types of such testing tools we will primarily look at are

- *assertion libraries* and
- *(stubbing and) mocking libraries*.

Both can be standalone or part of a greater testing framework or utility library. The former are used to specify a part of the *test oracle*, deciding whether a test case is considered to pass successfully or fail with an error. The latter are particularly important for substituting subsystems in component testing, and mock expectations can also form another part of the test oracle.

Assertions are often encoded as chained function calls, sometimes called *BDD (behavior-driven development)* style, or in a more classical style as plain calls to an assert function: [LC18b][LC18a]

```
expect(1).to.equal(1);
assert(1 === 1);
```

Mocking libraries tend to follow a somewhat similar API design approach: [JRC18c][JRC18a]

```
const doubleStub = stub().withArgs(21).returns(42);
const myMock = mock(myApi).expects("method")
                          .once().withArgs(42, 1337);
```

Using these libraries to abstract away the implementation details of our test oracles is key to allowing test authors to focus only on the inputs and expectations in a test case. But the libraries do not eliminate all boilerplate code. For example, we still need to repeat the `assert` call for each assertion in a test case that contains multiple of them, which may be a very minor annoyance, but can still be considered excess boilerplate.

More significantly, stubbing and mocking does not look natural at all. Why do we need to declare stub and mock calls like

```
mock(myApi).expects("method").withArgs(42, 1337)
```

instead of the ordinary

```
myApi.method(42, 1337)
```

that we know from the regular method call syntax as defined by the language?

1.2 Context

The popular testing framework *Spock Framework* for the Groovy programming language takes a route that significantly differs from the usage of classic testing helper libraries. It defines a *domain-specific language (DSL)* for the test cases that allows expressing assertions and stub / mock interactions in a very concise way. This DSL cannot be implemented entirely by a 'userspace' library, therefore, Spock hooks into the compiler and transforms the syntax tree prior to test execution.

Spock allows expressing assertions in `expect` blocks, or alternatively in `when-then` blocks: [NC18]

```
expect:
Math.max(1, 2) == 2
// ---
when:
stack.push(elem)
then:
stack.size() == 1
```

Stub and mock interactions can be expressed in the same fashion: [NC18]

```
then:
// return "ok" for every call to receive
subscriber.receive(_) >> "ok"
// expect 1 call to receive with argument "hello"
1 * subscriber2.receive("hello")
```

Spockjs is a work-in-progress implementation of Spock-style testing for JavaScript. [SC18c] The latest version at the time of writing supports use of the `expect` and `when-then` labels to express assertions, but has not yet implemented any of the more advanced features of Spock. Spockjs is test runner-agnostic by hooking into the popular JavaScript compiler babel, which integrates with most modern test runners. [McK+18d]

1.3 Goal

Our goal is to complement the existing spockjs functionality with basic support for Spock-style interaction testing. In addition to preserving the property of test runner-agnosticism, the interaction testing implementation shall also be agnostic of the mocking library in use — unlike the original Spock, which ships with its own mock / stub object implementation. This design provides the benefits

- that users can still access specific features of their library of choice past spockjs, in particular features that integrate well with their test runner and other tools, and
- that users can choose to quit using spockjs at any time and even use an automated escape hatch to return to using their library directly, like it is already possible with assertions in spockjs, thus avoiding a form of 'technology lock-in'.

To achieve this kind of flexibility, we experiment with and evaluate two different approaches:

1. Immediately compiling the interaction declarations to calls to the target mocking library, and
2. compiling the interaction declarations to calls to our own uniform interface while providing implementations of that interface for each target mocking library.

Because a comparison uncovering advantages and disadvantages of these two architectures as well as techniques to tame their individual challenges may also be valuable for the implementation of other DSLs using the custom compilation approach, we make highlighting those discoveries an explicit goal of ours too.

1.4 Structure

In the following Chapter 2, we will highlight the aspects of automated software testing that our later work will touch on, including test assertions, stubs and mocks.

Afterwards, in Chapter 3, we will take a closer look at the implementation of DSLs in general and at the DSL of the Spock Framework in particular. To explain the very foundation of how the Spock Framework manages to implement its DSL, we will work out the basics of syntax tree traversal and transformation in Chapter 4.

In Chapter 5, we will see how the current version of spockjs applies those techniques in order to implement functionality similar to the original Spock Framework for the JavaScript ecosystem, and which other features it provides. Chapter 6 will then tackle the groundwork that is necessary to add interaction testing capabilities to spockjs while supporting the use of multiple mocking libraries.

Chapters 7 and 8 describe two implementations of the full library-agnostic interaction testing feature set, following the *direct compilation* and *runtime dispatch* approaches, respectively. In Chapter 9, we wrap up by comparing the approaches, both with regard to their specific upsides and downsides in the context of spockjs interaction testing, and from a more general standpoint of compile-time versus runtime work.

2 Automated software testing

Many software projects these days strive to automate software quality control as much as possible. In particular, dynamic testing is a popular method to check whether the software fulfills its functional requirements. A wide variety of test runners, frameworks and support libraries are available and used to optimize the test creation workflow, and the wider project infrastructure is adapted to execute automated tests frequently within continuous integration pipelines.

In this chapter, we will introduce some basic concepts that are important for understanding automated software testing and the remaining chapters of this paper.

2.1 Test oracles

Test oracles are the part of a test that determines whether the test has failed or succeeded. [How78] In xUnit frameworks (such as xUnit.net, JUnit, and many more), they are typically encoded as *assertions*. [Mes04] There, they most commonly occur in test case bodies, but may also appear in other parts of the xUnit test structure, such as test fixture teardown functions to share an assertion across an entire test suite.

Assertions verify that a logical expression, which is expected to be an *invariant* at that point in the program, is indeed truthy during execution. If it is not, the assertion will cause the test to fail, usually by interrupting the control flow by means of throwing an exception:

```
assert(1 === 1); // assert(invariant) is no-op
assert(1 === 2); // throws an exception
assert(2 === 2); // never executed
```

2.2 Assertion helpers

Many programming languages and standard libraries natively provide a mechanism for assertions in tests or sometimes even in production code through syntax features, such as Python's `assert` statement, [17a, Chapter 7.3: The assert statement] or through functions, such as Node.js' `assert` module. [18e, Chapter: Assert]

In addition to those typically rather primitive helpers, there are many third-party libraries capable of intuitively performing more complex or specific assertions, as demonstrated by this *Chai* snippet: [LC18b, Chapter: API Reference — Language Chains — .not]

```
expect([1, 2]).to.be.an('array').that.does.not.include(3);
```

Examples include asserting [LC18b][Wal+18]

- deep equality,
- regular expression matches,
- emptiness of a collection, and
- occurrence in a collection,
  - optionally in a specific order.

2.3 Test drivers

Test drivers control the test execution from the outside [PS96] as shown in Figure 1. They can replace high-level system components that will at some point integrate the *test subject*, but have not yet been implemented, and thus they are particularly useful in conjunction with a bottom-up software development approach, where small components at the bottom of the hierarchy are first designed in their entirety and later integrated to form the software product as a whole.

```
1  import test from "ava";
2
3  import add from "./add";
4
5  test("adds two positive integers", t ⇒ {
6    t.is(add(1, 2), 3);
7  });
8
9  test("adds a positive and a negative integer", t ⇒ {
10   t.is(add(1, -2), -1);
11 });
```

Figure 1: An example test driver that uses the testing framework AVA [Wub+18] to execute two test cases for the test subject `add`.

Test drivers need not replace a future system component. They can also be used in place of components that have already been implemented, if it is still desired that the components it depends on should be tested on their own. They can also be used in place of components that will never be implemented because they are out of scope for the system being developed, in particular if the test drivers replace the human users who interact with the top-level components of the system in production themselves. [MSB11]

2.4 Test stubs

Test stubs are the inverse counterpart to test drivers. They replace components that the test subject depends on and provide canned answers to calls from the test subject to the stubbed component [Fow07] as shown in Figure 2. They are particularly useful in top-down software development, where the high-level components are implemented first and testing them requires stubs for their dependencies that are not yet implemented in full detail.

```
1 const subtract = add ⇒ (a, b) ⇒ add(a, -b);
2
3 const addStub = (a, b) ⇒ (a === 3 && b === -2 ? 1 : 0);
4
5 assert(subtract(addStub)(3, 2) === 1);
```

Figure 2: In this somewhat contrived stubbing example, the `add` function, a dependency of the `subtract` function defined in line 1, is being stubbed with the `addStub` function defined in line 3, which does not actually perform any addition logic, but instead only responds to a certain call with a canned answer. This way, `subtract` can be tested in isolation in line 5, without an actual implementation of `add`.

```
1 const subtract = add ⇒ (a, b) ⇒ add(a, -b);
2
3 const addCalls = [];
4 const addMock = (a, b) ⇒ (
5   addCalls.push({ a, b }), a === 3 && b === -2 ? 1 : 0
6 );
7
8 assert(subtract(addMock)(3, 2) === 1);
9 assert.deepStrictEqual(addCalls, [{ a: 3, b: -2 }]);
```

Figure 3: Using a mock defined in lines 4 to 6 to verify in line 9 that the `subtract` function defined in line 1 actually made a call to `add`. Note that this concrete example would typically be considered a significant overspecification in practice.

2.5 Mocks

There is no clear consensus on the meaning and differentiation of the terms ‘stub’ and ‘mock’, however, a common definition — and the one that we will use — is that mocks extend stubs to not only answer to calls with predefined responses, but also to place equally predefinable assertions on the calls that the test subjects performs on the mock, [Fow07] thus becoming part of the test oracle. Figure 3 shows the same example as for stubs before, but using a mock that introduces an assertion on the behavior of the test subject. Many mocking libraries choose to not make a distinction at all in practice and have their mock objects act as dummies, fakes, stubs, spies and mocks — however those may be distinguished — at once. [NC18][18d]

```
const subtract = add ⇒ (a, b) ⇒ add(a, -b);

const addMock = jest.fn(
  (a, b) ⇒ (a === 3 && b === -2 ? 1 : 0)
);

expect(subtract(addMock)(3, 2)).toBe(1);
expect(addMock).toBeCalledWith(3, -2);
```

Figure 4: Getting rid of mocking boilerplate by using a mocking library, in this example the one provided by the test runner *Jest*. [18a] Note that unlike the previous 'vanilla' example, this will allow additional calls to the mock as long as the first one is performed with matching arguments because of the way Jest implements the call matching.

Figure 4 shows how a mocking library can greatly simplify the mocking example by eliminating the manual argument capturing from the test code and providing call verification helper functions.

2.6 Summary

In this chapter, we have learned what test oracles are and we have seen how they are commonly manifested in assertions. We have also illustrated the use of test drivers and stubs in component testing and distinguished mocks from stubs.

In the following chapter, we will take a look at how the Spock Framework implements assertions and mocking, including stubbing.

3 The Spock Framework

The Spock Framework is a testing framework for the Groovy programming language and other languages that run on the *Java Virtual Machine (JVM)*. [NC18] It is one of the most used testing tools in the Groovy ecosystem. [18b, Chapter 3.6: Testing Guide] Its most remarkable feature is the concise and expressive, but somewhat unusual DSL for writing tests, which we will examine in this chapter.

3.1 Domain-specific languages

A *domain-specific language (DSL)*, as opposed to a general-purpose language like C or Java, targets a very specific application purpose. Examples from various areas of software development are:

- *Markdown* for text formatting, [Gru04]
- the *Gradle Build Language* for build configuration, [17b] and
- the *Spock specification language* for test cases. [NC18]

These three languages can be organized in three rough categories, distinguished by the way they are implemented:

1. Standalone DSLs such as Markdown,
2. entirely 'userspace' DSLs such as the Gradle Build Language, and
3. as a middle ground, custom compilation DSLs such as the Spock specification language.

Standalone DSLs Standalone DSLs are implemented by defining a wholly new lexical and syntactical grammar and the language semantics. One can then use a parser generator to obtain a parser for the language and include it in their software.

The major disadvantages of this approach for a testing DSL are

- that the language would have to be enormously complex in order to be able to replace the regular Groovy language
  - or it would have to be embedded in strings in the test cases, and
- that there would be no tooling support for the language that provides autocompletion, formatting, error checking, and other assistance like there is for widespread existing languages.

Userspace DSLs *Userspace DSLs* have recently become popular. They are implemented entirely using features of the host language. In particular, some languages are explicitly designed to support the creation of DSLs well, such as Groovy [18b, Chapter 3.15: Domain-Specific Languages] and, more recently, Kotlin [18c, Chapter: Type-Safe Builders]. Language features that can aid the creation of DSLs are [18b, Chapter 3.15: Domain-Specific Languages]

- optional parentheses around call arguments,
- operator overloading,
- closure delegates, and
- compilation customization for advanced DSLs of the last category.

Custom compilation DSLs *Custom compilation DSLs* also adhere to the syntax of an existing language, but they offer more flexibility, because they are not restricted to runtime language features. For example, labels in the code cannot be given additional semantics at runtime by a library, which is something that, as we will see, the Spock Framework needs to do.

But unlike a standalone language, existing tools can still handle the code written in the custom compilation DSL, because all of it is valid code in the language it builds upon — although some tools may be confused by the unorthodox usage of language features and become less helpful.

The compile-time portion of the implementation of this kind of DSL primarily employs AST transformations, which we will learn about in Chapter 4. Groovy has extensive support for custom compilation and these transformations, [18b, Chapter 3.15.6: Compilation customizers] enabling the Spock Framework to implement its testing DSL cleanly.

3.2 Spock testing DSL basics

Spock test cases consist of *blocks*. These blocks are started by a label, such as `setup:`, `expect:`, and `then:`. Any code in a test case before a block is declared implicitly becomes a *setup* block. [NC18, Chapter: Spock Primer - Feature Methods - Blocks] The various types of blocks available are shown in Figure 5.

Expect and *then* blocks can contain assertions formed by simple expressions. An assertion passes if the expression evaluates to a truthy value and fails if it evaluates to a falsy value. Simple functional programming-style assertions usually look cleaner using the expect style as shown in lines 2 to 5; code with side effects should be executed in a `when` block, followed by a `then` block verifying that the effects have resulted in the expected program state as shown in lines 16 to 26. It is also possible to assert that an exception has been thrown in the preceding `when` block using `thrown`, which can be seen in the `when-then` example test case.

The blocks *setup* (with its alias *given*) and *cleanup*, as shown in lines 7 to 14, are offered to run code before and after the assertions. `cleanup` blocks are particularly useful because they are *always* run, even if the previously executed blocks threw an exception. [NC18, Chapter: Spock Primer - Feature Methods - Blocks] This functionality makes them useful for tasks such as restoring the previous state after I/O operations, as shown in the setup-cleanup example test case.

3.3 Data-driven testing

Spock also provides a special syntax for *data-driven testing*, shown in the *where* example in lines 28 to 37 of Figure 5. Using the Data-driven testing features of the Spock Framework, the same test code can easily be run multiple times, but passing different inputs to the system under test and expecting different outputs from it. [NC18, Chapter: Data Driven Testing]

The given example declares the test data using a *data table*, resulting in a very compact representation of the data, separating the data points in the rows and the variable names in the header with a single pipe (|). A double pipe (||) is used to separate inputs (a / b) and expected values (r) clearly; this difference is only visual, Spock treats single and double pipes in data tables equally. [NC18, Chapter: Data Driven Testing - Syntactic Variations]

```groovy
class HelloSpockSpec extends Specification {
  def 'expect'() {
    expect:
    1 + 2 == 3
  }

  def 'setup-cleanup'() {
    setup:
    final file = new File('/tmp/example')
    file.createNewFile()
    // ...
    cleanup:
    file.delete()
  }

  def 'when-then'() {
    given:
    final stack = new ArrayDeque<Integer>()

    when:
    stack.pop()

    then:
    thrown NoSuchElementException
    stack.empty
  }

  def 'where'() {
    expect:
    Math.max(a, b) == r

    where:
    a | b || r
    1 | 3 || 3
    7 | 4 || 7
    0 | 0 || 0
  }

  def 'stub'() {
    // ...
    setup:
    subscriber0.receive(_) >> true
    subscriber1.receive(_) >> true

    expect:
    publisher.publish("msg") == true
  }

  def 'mock'() {
    when:
    publisher.publish("msg")

    then:
    1 * subscriber0.receive("msg")
    1 * subscriber1.receive("msg")
  }
}
```

Figure 5: An example specification that uses the Spock Framework, partially adapted from examples in its reference documentation. [NC18] Imports are not shown.

As an alternative to data tables, *data pipes* such as `a << [1, 7, 0]` can be used, additionally allowing users to generate data points dynamically, which cannot be achieved with data tables. However, since data-driven testing is not a primary concern of this paper, we will not go into more detail on data tables and data pipes and instead refer to the Spock Framework documentation. [NC18]

3.4 Interaction testing

The Spock Framework comes with an own library for creating test mocks (which are at the same time also test stubs). While it is possible to use any such library written in Java or another language that can be compiled to JVM bytecode in conjunction with the Spock Framework, only mock objects created using Spock can be used with its special syntax for declaring mock interactions. Mock objects can be created in the following two equivalent ways: [NC18, Chapter: Interaction Based Testing - Creating Mock Objects]

```
Subscriber subscriber0 = Mock()
final subscriber1 = Mock(Subscriber)
```

Stubbing To set up a mock object for stubbing, the *right shift operator* (`>>`) can be used:

```
subscriber0.receive("msg") >> true
```

This *interaction declaration* instructs the Spock Framework that when the method `receive` is called with the sole argument `"msg"` on the mock object `subscriber0`, the call should return `true`. Interaction declarations for stubbing usually appear in a `setup` block to prepare the mock object for usage in the following `when` or `expect` block.

The 'stub' example in lines 39 to 47 of Figure 5 also uses the special underscore (`_`) identifier. Used as a method argument, it denotes 'any argument' and thus makes the stub from the example return `true` on every single-argument call to its `receive` method.

Mocking To setup up expectations on a mock object, the *multiplication operator* (`*`) can be used as shown in lines 49 to 56 of Figure 5:

```
1 * subscriber0.receive("msg")
```

This is also an interaction declaration. It asserts that the method `receive` was called **exactly once** with the sole argument `"msg"` on the mock object `subscriber0`. Interaction declarations for mocking usually appear in a `then` block to verify that the expected calls to the mock object have been made in a the preceding `when` block.

The underscore placeholder could also be used here, either as the method argument, or even as the cardinality on the left-hand side of the multiplication operation, which denotes 'any number of invocations, including zero'. That would usually makes it a worthless interaction declaration, however, it can be useful to allow certain interactions when followed by a declaration that forbids a superset of those interactions. [NC18, Chapter: Interaction Based Testing - Mocking - Strict Mocking]

Advanced interaction declarations The interaction testing DSL of the Spock Framework is capable of much more than the functionality described here, but we will not implement or need to know about any of those advanced interaction testing features. For the sake of completeness and to suggest some material for future experimentation, we will briefly mention some of that functionality.

Stubbing and mocking can be combined within an interaction declaration:

```
1 * subscribe0.receive("msg") >> true
```

This way, the test can both assert that a method call occurs in the `when` block and stub a return value for that same call. Declaring the stub and mock interactions separately will not result in the desired behavior — only one interaction declaration will be matched when the call occurs, [NC18, Chapter: Interaction Based Testing - Combining Mocking and Stubbing] so either the call will return the default value as if there was no stubbing or the mock will fail to record the call for verification.

Other features include [NC18, Chapter: Interaction Based Testing]

- *spy* objects,
- stubs returning different values for each invocation,
  - even dynamically computed ones,
- declaring interactions on mock object creation,
- order-sensitive interaction expectations,

- cardinality ranges in mock interaction declarations,
- matching method names using regular expressions, and
- various advanced method argument constraints, such as
  - matching all arguments *except* a certain value,
  - matching arguments by type,
  - matching an arbitrary number of arguments, and
  - dynamically matching arguments using a closure.

3.5 Summary

In this chapter, we have examined the Spock Framework with regard to its testing DSL. We looked at how it can help structure your test cases well, how it implements assertions and data-driven testing, and in particular its interaction testing capabilities, which will serve as a starting point for designing and implementing similar features for spockjs.

We have also identified it as a DSL that is implemented via custom compilation and looked at the specific characteristics and advantages of this implementation concept for a testing language like the one Spock provides and the one that we will develop.

4 Abstract syntax trees

In this chapter, we will introduce the concept of abstract syntax trees, which is important for domain-specific and general-purpose languages alike, and look at how these trees can be put to use.

An *abstract syntax tree (AST)* represents the syntactic structure of program source code. It is a datastructure used to store source code internally by various tools, such as compilers, code formatters, and integrated development environments. It can be built from the source code tokens using a *parser*. Unlike a *concrete syntax tree*, an abstract syntax tree contains insufficient information to perform the inverse operation and recreate the original source code from the tree. [Jon03]

Its *vertices (nodes)* each store a node type, which can be discriminated upon in an implementation language-specific manner. Common examples for node types include

- *binary expressions*,
- *identifiers*, and
- *literals*.

Vertices can also store additional attributes, depending on their type. Examples of additional attributes include

- the *operator* used in a binary expression (although some AST formats instead use different node types for different binary operations [Jon03]),
- the *name* of an identifier, and
- the *value* of a literal.

The edges are labeled to indicate the relation between parent and child nodes, with the parent node type determining which edge labels are allowed. Examples of edge labels include

- the *test* of an if statement,
- its *consequent*, and
- its optional *alternate* (‘else block’)

There may also be multiple ordered children connected to their parent with edges of the same label, such as the *body* label for a *block statement* parent.

4.1 ESTree

We will use the ESTree AST format whenever we operate on ASTs. ESTree is the de facto community standard for ECMAScript ASTs, used in various popular tools, including the parsers ‘Acorn’ and ‘Esprima’, the linter ESLint, and the compiler Babel, which we will work with extensively. [Her+18]

The ESTree specification contains an interface for each node type, as well as abstract interfaces that others inherit from. All node type interfaces inherit — directly or indirectly — from the abstract `Node` interface: [Her+18]

```
interface Node {
  type: string;
  loc: SourceLocation | null;
}
```

The `loc` attribute contains information about the position of the source code section that corresponds to the AST node, but we can disregard it for now. The `type` attribute serves as the discriminator, with a unique value assigned to each node type that can be used to determine the concrete node type of a given node object.

The `IfStatement` is an example of such a concrete node type: [Her+18]

```
interface IfStatement <: Statement {
    type: "IfStatement";
    test: Expression;
    consequent: Statement;
    alternate: Statement | null;
}
```

An instance of it with its corresponding source code is shown in Figure 6.

Source code:

```
if (1 == 2) {
  console.log(42);
} else {
  console.log(1337);
}
```

ESTree node:

```
{
  "type": "IfStatement",
  "start": 0,
  "end": 65,
  "loc": {
    "start": {
      "line": 1,
      "column": 0
    },
    "end": {
      "line": 5,
      "column": 1
    }
  },
  "test": {
    "type": "BinaryExpression"
  },
  "consequent": {
    "type": "BlockStatement"
  },
  "alternate": {
    "type": "BlockStatement"
  }
}
```

Figure 6: An if statement in source code and its ESTree node, serialized as JSON. Its child nodes have been shortened; they include only the node types.

4.2 Traversal and transformation

In the following, we will illustrate how an AST can be traversed and transformed by defining a simple example transformation. We will swap the sides of each *strict equality comparison* (`===`) that occurs in the tree. The node objects that we operate on are shaped as defined by the ESTree specification.

4.2.1 Traversal

In object-oriented languages, different tree node types are represented as different classes that directly or indirectly implement the `Node` interface. There are multiple ways to navigate the tree, each resulting in a distinct order of node occurence. The most common algorithms for tree traversal in general are

- *Depth-first search (DFS)*, which can be performed
  - *Pre-order*,
  - *In-order*, or
  - *Post-order*, and
- *Breadth-first search (BFS)*.

However, we can disregard the traversal order for our example transformation (and for the other transformations in this paper), because we will only modify single nodes regardless of the context they appear in, and we do not hold any state across nodes.

Instead, we are interested in the structure of the code that manipulates AST nodes. There are multiple ways to switch between different actions depending on the type of the node we are currently processing.

Type switching The primitive approach shown in Figure 7 is to simply switch between code paths depending on the node type, which we can retrieve from the `type` property — or by using an `instanceof` operator with many other languages and AST formats.

```
switch (node.type) {
  case 'Identifier':
    console.log(node.name);
    break;
  case 'Literal':
    console.log(node.value);
    break;
  // ...
  default:
    throw new Error(`unknown node type '${node.type}'`);
}
```

Figure 7: Switching between different node types.

```
interface Node {
  handle(): void;
}

class Identifier implements Node {
  handle() {
    console.log(this.name);
  }
}

class Literal implements Node {
  handle() {
    console.log(this.value);
  }
}
```

Figure 8: Defining handlers on each node type.

However, this approach is generally considered bad design in the world of *object-oriented programming (OOP)*, [PJ98] requires type casts in languages that are neither dynamic nor structurally-typed, and needs a fallback case instead of verifying that all types are covered at compile time.

Abstract handler methods An alternative approach that better adheres to *object-oriented design (OOD)* principles is to define the handler method in the `Node` interface as shown in Figure 8. [PJ98] We implement it in each node type class and let the language dynamically dispatch for us.

The major disadvantage of this approach is that all handlers must be defined on the node types themselves. This becomes impractical if we want to apply multiple transformations to the AST, each with their own node handler logic, or even if we just want to define multiple handlers for totally different operations on the tree, such as machine code generation, bytecode generation, tree serialization, and pretty-printing.

Visitor pattern The final and most popular approach is to employ the *visitor pattern* as shown in Figure 9. [Gam+95] After dispatching to the `accept` method of the current node type, we further delegate to the *visitor* object in lines 7 and 13, which provides a method to handle each node type, including the one for the now well-known type of the current node `this`.

This approach is both immaculate OOD and fulfills the requirement to decouple the node types from their handlers, of which there may be a large quantity, serving different purposes. While traversing the tree we can now apply each visitor (or a list of visitors) to each node.

4.2.2 Transformation

Our strict equality swap visitor will only apply to `BinaryExpression`s, of which strict equality comparison expressions are a subset. We filter any other kind of binary operation; only for strict equality comparisons we swap the left subexpression with the right one:

```
if (node.operator === '===') {
  [node.left, node.right] = [node.right, node.left];
}
```

Other common transformations include [KC17]

- replacing a node with a new node (that is still valid with regard to its parent),
  - in particular, wrapping a node into another node by replacing it with a new node that has the old one as its child,
- inserting a sibling node,
- removing a node, and
- renaming a binding.

```
interface Node {
  accept(v: Visitor): void;
}

class Identifier implements Node {
  accept(v: Visitor) {
    v.visitIdentifier(this);
  }
}

class Literal implements Node {
  accept(v: Visitor) {
    v.visitLiteral(this);
  }
}


interface Visitor {
  visitIdentifier(identifier: Identifier): void;
  visitLiteral(literal: Literal): void;
}

class LogVisitor implements Visitor {
  visitIdentifier(identifier: Identifier) {
    console.log(identifier.name);
  }

  visitLiteral(literal: Literal) {
    console.log(identifier.value);
  }
}
```

Figure 9: The visitor pattern.

4.3 Summary

In this chapter, we have looked at how a program can be represented as an abstract syntax tree (AST) and what parts this type of data structure consists of. We have learned how to encode ASTs as concrete objects according to the ESTree specification, which will be useful when we work with the compiler Babel.

We have also examined some patterns that can be used to execute different pieces of code based on the types of tree nodes that we encounter during tree traversal and have found the visitor pattern to be particularly useful for this task. Finally, we have looked at some of the tree mutation operations that are commonly performed in visitors during tree traversal.

5 The state of spockjs

In this chapter, we will examine the current state of spockjs, primarily paying attention to

- its features,
- their implementation,
- its architecture, and
- its future roadmap.

5.1 Assertion blocks

Assertion blocks are the only feature of the original Spock Framework that spockjs has implemented so far. A simple assertion can be realized using spockjs in the same way it would look using the original Spock: [SC18c]

```
expect: 'a' + 'b' == 'ab';
```

In ESTree terms, this is a *LabeledStatement* with an *Identifier* named *'expect'* as its *label* and an *ExpressionStatement* as its body, containing the strict equality *BinaryExpression*.

Unlike Spock, to write an assertion block with multiple assertions in spockjs, the *ExpressionStatement*s must be wrapped in a *BlockStatement*: [SC18c]

```
expect: {
  'a' + 'b' == 'ab';
  1 < 2;
}
```

This allows users to continue with ordinary statements after an assertion block instead of implicitly assuming that everything in a test case that comes after a label is an assertion. It also fits the representation in the AST better, since all assertions are children in the *body* of the *LabeledStatement* that represents the assertion block instead of its following siblings inside the surrounding method body or other *BlockStatement*. In consequence, we also avoid a weird formatting style of the assertion block that an automatic formatter might otherwise produce:

```
expect:
  'a' + 'b' == 'ab';
1 < 2;
// or
expect: 'a' + 'b' == 'ab';
1 < 2;
```

Using the alternative *when-then-style* for code with side effects is supported as well, with `then` and `expect` blocks being treated in exactly the same way:

```
when: {
  abc.setXyz(1);
}

then: {
  abc.getXyz() == 1;
}
```

The `when` block does not have any special meaning here, its only purpose is to improve readability of the test case but could in theory be replaced with just its contents. [SC18c]

Using statements other than *ExpressionStatements* and *BlockStatements* consisting of *ExpressionStatements* — such as control structures — in an assertion block is not permitted. For instance, the following code would not compile with spockjs:

```
expect: {
  if (x < 1) x == 0.5;
  else x == 2;
}
```

However, assertion blocks can be nested into other structures, so the desired behavior from this example can be achieved in the following way: [SC18c]

```
if (x < 1) expect: x ≡ 0.5;
else expect: x ≡ 2;
```

5.2 Installation

As noted earlier, spockjs is compatible with most modern JavaScript test runners — it is *not* a test runner by itself, which sets it apart from the Spock Framework. Instead, spockjs ships in the form of a plugin for the popular JavaScript compiler Babel, [McK+18a] which can be set up to work with most modern test runners. [McK+18d] We will look at how spockjs uses Babel to implement assertion blocks later in the chapter.

5.2.1 Configuration

Spockjs is designed to work well by default, but does offer a few configuration options, mostly to turn miscellaneous features on or off. [SC18c]

powerAssert This on-by-default option causes spockjs to generate detailed error messages when an assertion fails using an external library provided by the *power-assert* project. [WC18] The following is an example of how such an error message can look: [SC18c]

```
assert(3 * 3 ⩾ 4 * 4)
         |   |    |
         |   |    16
         9   false
```

autoImport This on-by-default option causes spockjs to automatically import necessary modules — such as the `assert` core module — into the test file. Without this feature, users would need to manually add an import to each test file, which is easy to forget and might cause other tools to warn about the import being supposedly unused.

staticTruthCheck This off-by-default option causes spockjs to use the static analysis capabilities of Babel in order to detect assertions that can be inferred to always be truthy or always be falsy, which indicates that the assertion might not provide any value to the test case, and throw an error without even executing the test. The following is an example of such a useless assertion:

```
const x = 1;
expect: x === 1;
```

assertFunctionName This string option can override the name of the assert function spockjs uses, which can be useful if *autoImport* is not used, or for the *escape route* that spockjs provides for users who want to stop using spockjs without rewriting all their tests, which we will get back to in a later chapter. By default, the name of the assert function is *'assert'* or whichever free identifier is chosen for the automatic import.

presets This array of strings can be used to apply so-called *presets*, which can hook into the transformation logic of spockjs and customize its behavior in various ways. We will analyze the preset mechanism in greater detail when talking about the architecture of spockjs. Spockjs currently provides two 'official' presets:

- *@spockjs/runner-jest* ensures helpful assertion error messages when using the test runner Jest, [18a] which would otherwise hide too many details due to its special handling of assertion errors.
- *@spockjs/runner-ava* eliminates the necessity to set an AVA [Wub+18] configuration option that makes test cases fail erroneously if not set, and it causes assertion error messages to look exactly the same as if native AVA assertions were used.

5.3 Babel

Babel is a popular community-developed JavaScript compiler that is used and sponsored by many of the most prominent tech companies and has acquired more than 28000 stars on GitHub as of mid-July 2018. [McK+18c] Started as '6to5', serving the purpose of transforming ECMAScript 2015 (at that time named ECMAScript 6) code to ECMAScript 5 code that would

run in all browsers and on all platforms regardless of their ECMAScript 2015 support, Babel can now not only transform input code using more recent versions of ECMAScript such as ECMAScript 2018 and even ES.Next proposals that are not yet in the standard, but also apply arbitrary plugins developed by the community that can transform the code to their liking. [DC18] [WC18]

In the following, we will acquire some basic knowledge on how to develop Babel plugins. For a more extensive and detailed view that would go beyond the scope of this paper, both the author and the official Babel documentation recommend the Babel Plugin Handbook. [KC17]

5.3.1 Plugin visitors

. Babel plugins apply AST transformations in *visitors* as introduced in Chapter 4. A Babel plugin is essentially no more than a module that exports a function returning an object with arbitrary visitor functions. The following is the full Babel plugin to accomplish the strict equality swap example transformation as described in Chapter 4:

```
1 export default ({ types: t }) => ({
2   visitor: {
3     BinaryExpression({ node }) {
4       if (node.operator === "==") {
5         [node.left, node.right] = [node.right, node.left];
6       }
7     }
8   }
9 });
```

The `types` property of the object passed to the plugin in line 1 is not used in this particular plugin, but is usually one of the most important helpers, providing creation methods for each type of node and the corresponding type check methods:

```
t.isIdentifier(t.identifier('something')) === true
```

The visitor object returned from the plugin can contain a visitor method for each node type (capitalized), but it may be partial, i.e. it does not have to implement methods for all of the more than 200 node types. In this case, we only provide a visitor for binary expressions in lines 3 to 7. The visitor method receives the current `path` object as its first parameter,

which we will later take a closer look at, but in this case, we extract only the binary expression node from it in line 3. There is also a second `state` parameter that we can disregard here, but it is for example needed to access user-supplied configuration options that a plugin might offer.

The remaining part of the plugin inside the binary expression visitor is the code developed in Chapter 4 and performs the transformation as already explained there.

5.3.2 Nodes vs. paths

Visitor methods receive *path* objects when they are called while the AST is traversed. Unlike nodes, which almost exactly reflect the node objects as defined by ESTree, paths are aware of their context in the tree and, in particular, their parents. This means that while a path can be used to reach and manipulate any part of the tree, including those above the corresponding node, a node can only be used to inspect and manipulate its children.

We have already seen how to navigate and manipulate the primitive node objects, so let us now take a quick look at the more powerful path objects, covering only a select few of their many functions that are relevant to spockjs.

To retrieve child paths, `get` can be used, specifying the same property access path that would be taken on a node as a string argument. The following example describes the path navigations using `get` on the left by comparing them to their corresponding node navigations on the right:

```
path.get('left').node   ≡≡ path.node.left
path.get('body.0').node ≡≡ path.node.body[0]
```

While there is no reason to use the more cumbersome path navigation in this example, we need to use it if we want to call methods on the child path, some of which are named in the following. There is no way to recover the path object if we only have its corresponding node object.

Spockjs frequently uses the `replace` method, which discards the entire node that belongs to the path and replaces it with a given node, and its sibling method `replaceWithMultiple`, which replaces it with all nodes in a given array, provided that the surrounding context allows that to happen.

In fatal error cases, the utility method `buildCodeFrameError` is highly useful to bail with an error message that tells the user where in the source code the bit that the plugin doesn't understand is located.

5.4 Implementation

In this section, we examine the existing implementation of assertion blocks in spockjs, broken down to

1. the detection and processing of blocks in general, and
2. assertion transformation specifically.

5.4.1 Blocks

The *@spockjs/babel-plugin-spock* package is the entry point that exposes a Babel plugin — we will talk about the multi-package structure of spockjs later in the chapter. The full source code of this package is shown in Figure 10.

The Babel plugin in lines 39 to 54 consists of just a single visitor method for labeled statements. The visitor checks whether the label name of the current statements is one of the known labels that designate an assertion block — `['expect', 'then']` — and if so, it calls the *transformLabeledBlockOrSingle* helper defined in lines 11 to 37 to apply the *assertifyStatement* transformation to each statement in the case of a block statement contained inside the labeled statement, or otherwise to the single statement contained inside the labeled statement.

Note that the *assertifyStatement* transformation function, which comes from the *@spockjs/assertion-block* package that we will address after the general block handling, is a higher-order function with the signature

```
(babel, state, config) ⇒ (
  statementPath: NodePath<BabelTypes.Statement>
) ⇒ void;
```

, so after applying the first three arguments, it can be used any number of times on different statement paths.

The *transformLabeledBlockOrSingle* helper does exactly that for us, applying the transformation to the statements in a block statement in line 26, or to a single statement in line 32. It also removes the labels by replacing the labeled statement with its body in line 29 or 35, respectively, in order to produce cleaner code. We will be able to reuse this helper when implementing interaction blocks later on.

```ts
import { PluginObj } from '@babel/core';
import { NodePath } from '@babel/traverse';
import * as BabelTypes from '@babel/types';

import { extractConfigFromState } from '@spockjs/config';

import assertifyStatement, {
  labels as assertionBlockLabels,
} from '@spockjs/assertion-block';

const transformLabeledBlockOrSingle = (
  transform: (statementPath: NodePath<BabelTypes.Statement>) => void,
  path: NodePath<BabelTypes.LabeledStatement>,
): void => {
  const labeledBodyPath = path.get('body') as NodePath<BabelTypes.Statement>;

  switch (labeledBodyPath.type) {
    case 'BlockStatement':
      // power-assert may add statements in betweeen,
      // so never reuse body array
      const statementPaths = () =>
        (labeledBodyPath.get('body') as any) as NodePath<
          BabelTypes.Statement
        >[];

      statementPaths().forEach(transform);

      // remove label
      path.replaceWithMultiple(statementPaths().map(stmtPath => stmtPath.node));
      break;
    default:
      transform(labeledBodyPath);

      // remove label
      path.replaceWith(labeledBodyPath);
  }
};

export default (babel: { types: typeof BabelTypes }): PluginObj => ({
  visitor: {
    LabeledStatement(path, state) {
      const config = extractConfigFromState(state);
      const label = path.node.label.name;

      // assertion block
      if (assertionBlockLabels.includes(label)) {
        transformLabeledBlockOrSingle(
          assertifyStatement(babel, state, config),
          path,
        );
      }
    },
  },
});
```

Figure 10: The source code of *@spockjs/babel-plugin-spock* before any interaction testing capabilities were added.

```
1  (statementPath: NodePath<BabelTypes.Statement>) ⇒ {
2    if (statementPath.isExpressionStatement()) {
3      const statement = statementPath.node;
4      const origExpr = statement.expression;
5
6      const assertIdentifier = t.identifier('assert');
7      const newExpr = t.callExpression(assertIdentifier, [origExpr]);
8
9      statement.expression = newExpr;
10   } else {
11     throw statementPath.buildCodeFrameError(
12       `Expected an expression statement, but got a statement of type ${
13         statementPath.type
14       }`,
15     );
16   }
17 };
```

Figure 11: A stripped-down version of *@spockjs/assertion-block* without powerAssert, autoImport, staticTruthCheck, assertFunctionName, preset post processing, line/column number handling, and the enclosing function with the babel, state and config parameters.

5.4.2 Assertions

The *@spockjs/assertion-block* exports the aforesaid function that can be used to *assertify* a statement, converting it from an unused expression statement to a function call and passing the result of evaluating the expression to the assert function. Figure 11 shows the core part of the package that performs this transformation, while omitting some miscellaneous code that is not essential to the assertion block concept.

First of all, if the statement passed to us is not an *ExpressionStatement*, we will not be able to use it as an argument to an assert function call, so we bail with a compile-time error in lines 11 to 15. Otherwise, we generate a *CallExpression* with the *Identifier* 'assert' as its *callee*, and the original expression as its single argument in lines 6 to 7 and update the statement to our new expression in line 9.

Figure 12 shows the AST after such a transformation, with the nodes and edges that have been newly inserted highlighted in teal, for the example input code

```
expect: true;
```

that is transformed to:

```
expect: assert(true);
```

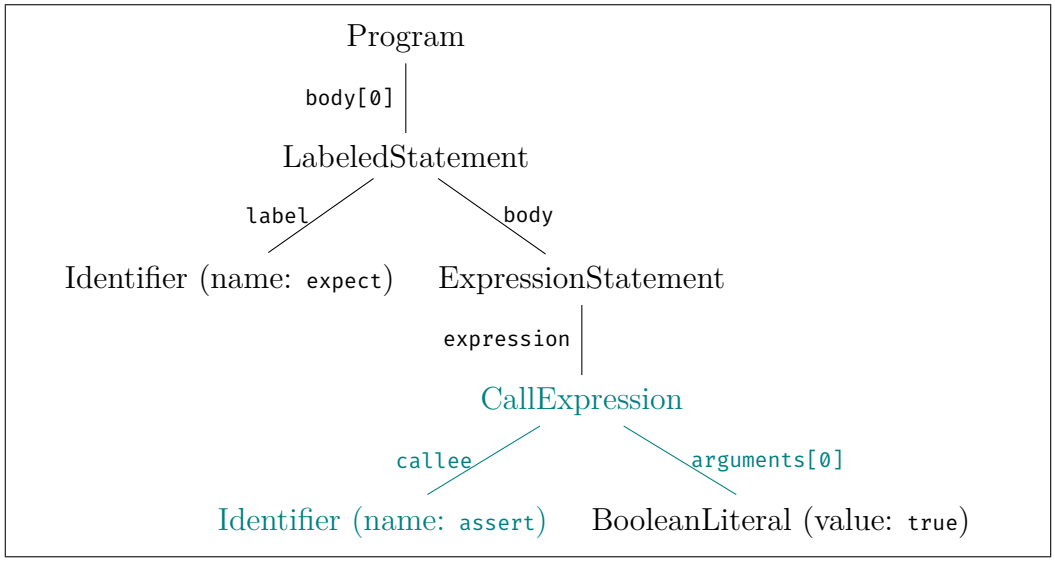

Figure 12: An exemplary AST after an assertion block transformation, with newly inserted nodes and edges highlighted in teal.

5.5 Architecture

Although the entire codebase of spockjs resides in a single source code repository (*monorepo*), it is organized in and distributed as various packages on the *npm (Node.js package manager)* registry, [18f] the de facto standard place for publishing JavaScript code. The packages are all published in the *@spockjs* namespace. We have already seen two of these packages: *@spockjs/babel-plugin-spock* and *@spockjs/assertion-block*. There are a total of nine packages, and we will add some more when implementing interaction testing.

Inside the repository, a package *@spockjs/name* is located in a *packages/name* directory. The 'workspaces' feature [18h] of the package manager *Yarn* links them all together during development so changes across multiple packages do not require first publishing the changes to a dependency before working on the changes to a dependent. In addition, the tool *Lerna* [Sto+18] is used to perform tasks such as builds, changelog generation, version bumps, and releases on multiple packages at once.

Presets The spockjs *preset* system is a particularly useful application for the multi-package architecture. We have already mentioned the `presets` configuration option briefly. A preset is a package that exports an arbitrary number of *hooks* that can tap into the process of transformation and perform additional manipulations. The presets specified in the configuration will be dynamically imported during compilation. This split also has the effect that users who add *@spockjs/babel-plugin-spock* as a dependency will only receive the few core packages of spockjs; if they want additional features, they can seperately install packages like *@spockjs/preset-runner-jest* and add them to the presets array.

The *@spockjs/preset-runner-jest* preset, for example, improves the experience for developers that use spockjs with the Jest test runner. Jest reports `AssertionError`s in tests uniquely by printing some additional information based on the assert function. Unfortunately, this information is confusing for users of spockjs, who never wrote an assert function call, because it was generated for them behind the scenes. To make Jest print only the error message supplied by spockjs, the preset exports the *@spockjs/assertion-post-processor-regular-errors* hook, which wraps the assertion in a try-catch block that catches the `AssertionError` and rethrows a regular `Error`:

```
try {
  assert(true);
} catch(e) {
  if (e instanceof AssertionError) {
    throw new Error(e.message);
  }
  throw e;
}
```

While implementing interaction testing, we will make further use of the preset mechanism to let users specify the mocking library to be used in the code generated by spockjs.

5.6 Roadmap

Spockjs has a lot of potential for additional features in the form of new block types. Conceivable enhancements are

- a block to perform the common task of asserting deep equality on objects and arrays, e.g. `expect: deepEq: a, b;`,

- a block to assert that the statements inside it throw an error, e.g. `expect: throws: fn();`,
- a block to capture a snapshot of a value, given that the test runner supports snapshot testing, e.g. `snapshot: a;`, and
- combinations of those, e.g. `snapshot: throws: fn();` to assert that an error is thrown and capture a snapshot of it.

The largest upcoming enhancement is the interaction testing capabilities similar to the original Spock Framework. We will lay the foundation for those in the next chapter.

5.7 Summary

In this chapter, we looked at both the concept and the rough implementation of spockjs assertion blocks. We learned some of the basics of Babel plugins along the way. The part that is not specific to assertion blocks, but applies to blocks in general, can be reused for implementing interaction blocks. The architecture of spockjs will allow us to do this without touching any of the packages that are only responsible for the existing assertion block transformation.

To get the whole picture, we also learned how to install and configure spockjs for a project, and what features might be coming up next, one of which is the interaction testing capabilities we will deal with in the next chapter.

6 Making interactions work

Like the assertion block DSL, our interaction block DSL will be heavily inspired by the Spock Framework. However, we will only design and start implementing a subset of its capabilities in this chapter and in the scope of this paper, restraining ourselves to the simplest mock and stub interactions and the combination of those two in one interaction.

6.1 Syntax

Like assertion blocks, interaction declaration blocks are also introduced by a labeled statement. The label name may be either *mock* or *stub*, with no technical distinction being made between those two — although users are, of course, urgently reminded not to use them inappropriately.

The body of such a labeled statement — or, in the case of a block statement, each statement contained within it — may have the shape of a *Mock-*, *Stub-*, or *CombinedInteractionDeclaration* as shown in Figure 13. The multiplication and right shift operators used in those interaction declarations are the same that the Spock Framework uses for interaction declarations.

```
MockInteractionDeclaration:
cardinality * call;


StubInteractionDeclaration:
call >> returnValue;


CombinedInteractionDeclaration:
cardinality * call >> returnValue;
```

Figure 13: The source code structure of a Mock-, Stub-, or CombinedInteractionDeclaration

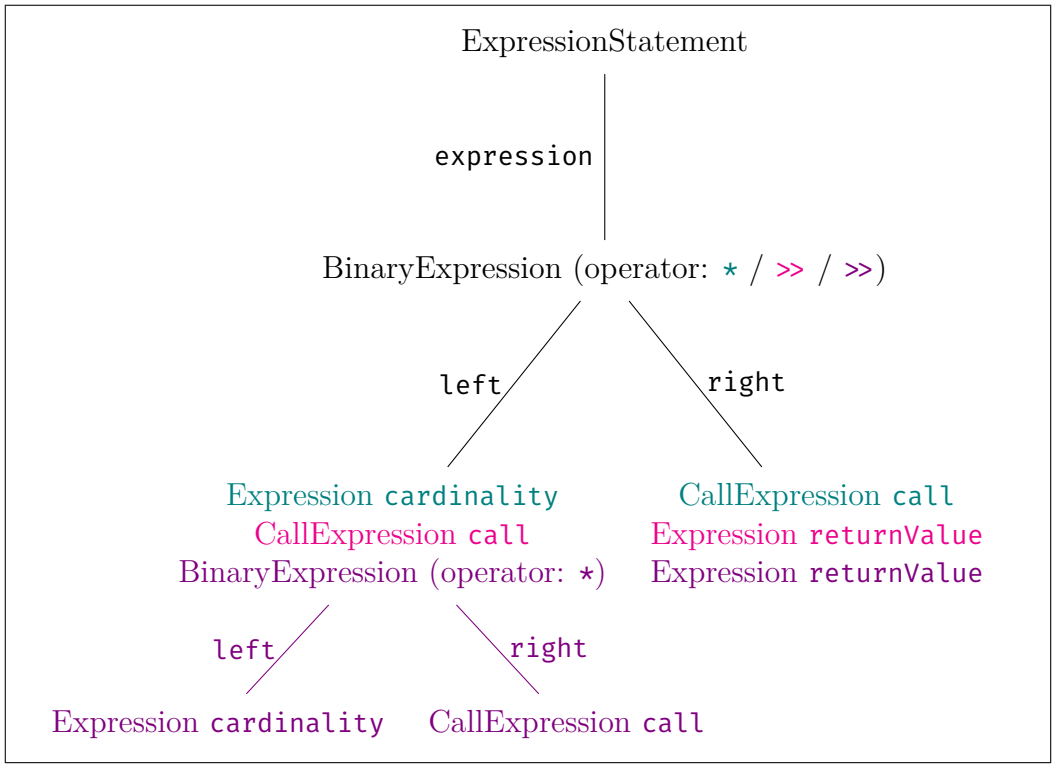

Figure 14: The AST of a Mock-, Stub-, or CombinedInteractionDeclaration, matching colors with their corresponding source code structures in Figure 13. The *calls* are collapsed to save space; their homogeneous subtrees are shown in detail in Figure 15.

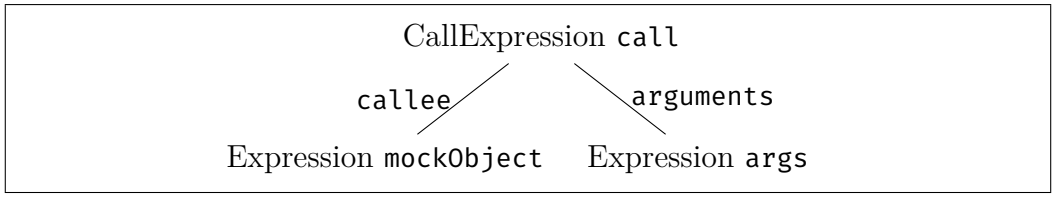

Figure 15: The AST of a *call* as it appears in multiple places in Figure 14.

Figure 14 shows the corresponding AST structure for all three variants, distinguished by color. Later in this chapter, we will build a parser that can extract the interesting parts of a declaration AST that can be used to generate mocking library calls that represent the given interaction declarations. The terminal nodes we need to extract are

- the *mockObject*,
- the *args*,
- the *cardinality* (for mock interactions), and
- the *returnValue* (for stub interactions).

The *mockObject* and the *args* always appear in the same `CallExpression` *call* that is shown in Figure 15, although that call may be attached to three different sections of the AST.

In addition to interaction declaration blocks, there are also interaction verification blocks that are used to verify that the declared calls on a mock have actually occurred:

```
verify: mockObject;
```

This explicit verification command is required to be test runner-agnostic, because we would need to know the test case structure for automatic insertion of verification statements. We could still allow their omission by implementing automatic insertion in test runner presets at some point.

6.2 Integration

In its top-level *index.ts* file, our new *@spockjs/interaction-block* package exports a function with the signature:

```
(t, config) ⇒ ({
  declare: (statementPath: NodePath<BabelTypes.Statement>) ⇒ void,
  verify: (statementPath: NodePath<BabelTypes.Statement>) ⇒ void,
});
```

This signature is reminiscent of *@spockjs/assertion-block* ('assertifyStatement'), because it, too, performs the task of transforming a declarative block into something that actually executes meaningful code. However, this package can transform two distinct types of blocks — interaction declaration and interaction verification blocks — so after applying the `t` and `config` argu-

ments that do not change throughout the compilation, there are both the *declare* and the *verify* function available to be applied to individual blocks with the respective labels *mock/stub* and *verify*, which are also exported by the package as *declarationLabels* and *verificationLabel*.

To integrate the new interaction block transformation package into the central *@spockjs/babel-plugin-spock* Babel plugin package — similarly to the assertion block transformation package — we first import all of its exported values:

```
import transformInteractionDeclarationStatement, {
  declarationLabels as interactionDeclarationLabels,
  verificationLabel as interactionVerificationLabel,
} from '@spockjs/interaction-block';
```

Then, we add if statements that handle the new block types right below the assertion block handling (after line 51 in the plugin source code that was previously shown in Figure 10):

```
1  // interaction block
2  else if (interactionDeclarationLabels.includes(label)) {
3    transformLabeledBlockOrSingle(
4      transformInteractionDeclarationStatement(babel.types, config).declare,
5      path,
6    );
7  } else if (label === interactionVerificationLabel) {
8    transformLabeledBlockOrSingle(
9      transformInteractionDeclarationStatement(babel.types, config).verify,
10     path,
11   );
12 }
```

6.3 Interaction declaration parsing

The first step of any *declare* transformation is parsing the interaction declaration. The three different interaction declaration types defined by the syntax are represented as a tagged union `InteractionDeclaration` with the discriminator property `kind` as shown in lines 7 and 12 and 17 of Figure 16. The parser we will now build has the following function signature:

```
(NodePath<BinaryExpression>) ⇒ InteractionDeclaration;
```

In the following, we will use pseudocode to build the parser top-to-bottom. The full source code is included in the appendix.

```
export interface BaseInteractionDeclaration {
  mockObject: BabelTypes.Expression;
  args: (BabelTypes.Expression | BabelTypes.SpreadElement)[];
}
export interface MockInteractionDeclaration
  extends BaseInteractionDeclaration {
  kind: 'mock';
  cardinality: BabelTypes.Expression;
}
export interface StubInteractionDeclaration
  extends BaseInteractionDeclaration {
  kind: 'stub';
  returnValue: BabelTypes.Expression;
}
export interface CombinedInteractionDeclaration
  extends BaseInteractionDeclaration {
  kind: 'combined';
  cardinality: BabelTypes.Expression;
  returnValue: BabelTypes.Expression;
}
export type InteractionDeclaration =
  | MockInteractionDeclaration
  | StubInteractionDeclaration
  | CombinedInteractionDeclaration;
```

Figure 16: The tagged union type `InteractionDeclaration`.

The entry point function `parseInteractionDeclaration` looks at the operator used in the top-level binary expression and decides whether the input is a mock interaction declaration with operator `'*'` (line 3) or a stub or combined interaction declaration with operator `'>>'` (line 5):

```
1 parseInteractionDeclaration(expressionPath):
2   switch (expressionPath.node.operator)
3     case '*'
4       return parseMockInteractionDeclaration(expressionPath)
5     case '>>'
6       return parseStubInteractionDeclaration(expressionPath)
7     default
8       throw 'unexpected operator'
```

The sub-parser function `parseMockInteractionDeclaration` as shown below simply uses the left side of the binary expression as the *cardinality* in line 5 and parses the right side of the binary expression as a *call* in line 3:

```
1 parseMockInteractionDeclaration(expressionPath):
2   return
3     mockObject, args = parseCall(expressionPath.get('right'))
4     kind = 'mock'
5     cardinality = expressionPath.node.left
```

The sub-parser function `parseStubInteractionDeclaration` as shown below behaves similarly for a pure stub interaction declaration, using the right side of the binary expression as the *returnValue* in line 10 or 15 and parsing the left side of the binary expression as a *call* in line 13. However, if the left side is a binary expression with operator `'*'`, a combined interaction declaration was used, and the left side is instead parsed as a mock interaction declaration in lines 7 to 8:

```
1  parseStubInteractionDeclaration(expressionPath):
2    let leftPath = expressionPath.get('left')
3    let returnValue = expressionPath.node.right
4    if (leftPath.isBinaryExpression())
5      if (leftPath.node.operator == '*')
6        return
7          mockObject, args, cardinality =
8            parseMockInteractionDeclaration(leftPath)
9          kind = 'combined'
10         returnValue = returnValue
11     throw 'unexpected operator'
12   return
13     mockObject, args = parseCall(leftPath)
14     kind = 'stub'
15     returnValue = returnValue
```

The sub-parser function `parseCall` as shown below simply extracts the *callee* as the *mockObject* in line 4 and the *arguments* as the *args* in line 5 from a *CallExpression*:

```
1  parseCall(expressionPath):
2    if (expressionPath.isCallExpression())
3      return
4        mockObject = expressionPath.node.callee
5        args = expressionPath.node.arguments
6    throw 'unexpected expression type'
```

6.4 Multi-library support

Unlike the common operation of parsing interaction declarations, generating statements that perform the declaration after parsing as well as generating statements that perform the verification are more complex tasks that require proper abstractions in order to work with multiple supported mocking libraries.

6.4.1 Purpose

There are multiple reasons why we want to provide support for different mocking libraries as opposed to supporting just a single one or even shipping our own.

Distinctive library features Users may wish to use special functionality provided by their mocking library of choice alongside the usage through spockjs. This is particularly common for libraries that are included in a test runner and designed to integrate well with it.

For example, Jest [18a] allows replacing the module `module.js` with a mock implementation `__mocks__/module.js` wherever the module is used by simply calling `jest.mock('./module.js')` in a test. [18d]

Avoiding technology lock-in Forcing the usage of a specific mocking library, possibly even one that is only used for spockjs, would also significantly increase the severity of technology lock-in imposed by spockjs by preventing users from returning to direct usage of the mocking library if they wish to stop using spockjs.

Indeed, avoiding this has been a focus for the implementation of assertion blocks. The spockjs documentation has a section dedicated to an automated escape hatch that allows users to transform *(codemod)* all of their tests to make conventional assertion calls, allowing them to drop spockjs without first undergoing an enormous refactoring effort. [SC18c]

6.4.2 Library selection and semantics

Telling spockjs which mocking library to use should take no more than enabling a preset in the configuration. However, at least in the first version, spockjs does not aim to be fully opaque by hiding the mocking library in use as an 'implementation detail'. This means that users will still need to be aware about the mocking library in use.

For example, interaction declarations do not have to carry the same semantics regardless of which mocking library is used. If, for instance, declaring the same interaction twice is handled by overriding the first interaction by one backing library, another library may instead handle it by 'queueing' the declarations and expecting the calls to occur in succession.

The reasons for this decision are

- that ensuring consistent behavior across mocking libraries would mean reimplementing large parts of the libraries ourselves, using the library primitives but not features which may diverge for different libraries, such as their interaction matching capabilities,
- that future advanced features of the spockjs interaction block syntax may not be supported with all mocking libraries, and
- that users would still need to instantiate mock objects themselves, so the backing library would not be fully opaque anyway.

Our implementations of interaction blocks for spockjs will initially provide presets for the Sinon.JS mocking library [JRC18b] and the built-in mocking library of the Jest test runner [18a]. As a reference for the semantics of spockjs when used with those libraries, test cases for interaction blocks backed by the respective libraries are included in the appendix. Those test cases specify the desired behavior in greater detail than the intuition obtained from the Spock Framework interaction testing semantics.

6.4.3 Approaches

We will try out two different approaches to implementing interaction declaration and verification across different mocking libraries.

Direct compilation The first and perhaps most obvious approach, *direct compilation*, works by generating code that calls functions provided by the mocking library in use.

A mocking library preset exports an *interaction processor*, which generates a statement for a declaration based on the parsed *InteractionDeclaration*, and a statement for a verification based on the *mock object*.

For example, the interaction declaration

```
mock: 1 * fn();
```

might be transformed to

```
expect(fn).withoutArgs().times(1);
```

for a fictional mocking library. The interaction verification

```
verify: fn;
```

might be transformed to:

```
verify(fn);
```

Runtime dispatch The second approach, *runtime dispatch*, works by generating code that always makes the same call, but swaps out the *interaction runtime adapter* that accepts the call depending on the mocking library in use. Each runtime adapter knows how to translate the generic declaration and verification calls to calls for the mocking library that it provides support for.

A mocking library preset exports the name of the module that exports the interaction runtime adapter, so it can be imported in the generated code.

The interaction declaration

```
mock: 1 * fn();
```

might be transformed to

```
interactionRuntimeAdapter.declare({
  kind: 'mock',
  mockObject: fn,
  args: [],
  cardinality: 1,
});
```

and the interaction verification

```
verify: fn;
```

might be transformed to:

```
interactionRuntimeAdapter.verify(mock)
```

In addition, an import

```
import interactionRuntimeAdapter
  from '@spockjs/interaction-runtime-adapter-some-lib';
```

needs to be added at the top of the file to select the mocking library. The actual calls to the mocking library happen inside the adapter with this approach.

A variant of this approach would be to compile to calls to the interface of a common mocking library and provide adapters that translate from that library to others. Such adapters could perhaps be more reusable for other purposes outside of spockjs, but would take a lot of flexibility away from spockjs.

6.5 Summary

In this chapter, we defined the scope of the interaction testing capabilities that we want to add to spockjs and presented the concrete syntax for them.

We explained why and how users should be able to choose their preferred mocking library to use in conjunction with spockjs and presented two ways to implement the interaction testing capabilities while preserving that choice for our users.

We will go both of those ways in the following chapters in order to compare them afterwards. The interaction parser and the integration into the spockjs Babel plugin constitute the portion of the code that is common to both approaches. In this chapter, we also implemented that common portion already.

7 Direct compilation

In this chapter, we will develop two *interaction processors* that implement interaction blocks for the Sinon.JS and Jest mocking libraries to try out the direct compilation approach. We will also finalize *@spockjs/interaction-block* by adding code to delegate to one of the interaction processors at compile time. This chapter will not yet evaluate the approach; evaluation is set aside until both approaches have been implemented.

7.1 Presets

With the direct compilation approach, the two mocking library presets *@spockjs/preset-sinon-mocks* and *@spockjs/preset-jest-mocks* each have to export an array of *interaction processors*. An interaction processor has the following type signature:

```
(
  t: typeof BabelTypes,
  config: InternalConfig
) ⇒ {
  primary: boolean;
  declare(
    interaction: InteractionDeclaration
  ): BabelTypes.Statement;
  verify(
    mockObject: BabelTypes.Expression
  ): BabelTypes.Statement;
};
```

The signature somewhat resembles that of the *@spockjs/interaction-block* package itself in that it also offers the *declare* and *verify* operations. This is because *@spockjs/interaction-block*, aside from parsing interaction declarations, mostly just delegates them through with this approach, as we will see when finishing the implementation of that package soon.

The interaction processor property *primary* indicates whether the processor is meant for standalone use (`true`), as is the case for processors implementing mocking library bindings, which we deal with in this paper, or an auxiliary processor (`false`), such as one that provides better integration with a test runner, which we disregard in this paper. Users will see an error when attempting to use interaction blocks without a primary interaction processor.

The Sinon.JS and Jest mocking library presets do not contain the corresponding interaction processors themselves. Instead, like all such components of spockjs, including the similar assertion post processors, the interaction processors exist in the separate packages *@spockjs/interaction-processor-sinon-mocks* and *@spockjs/interaction-processor-jest-mocks*, which allows for them to be easily combined in other presets. The *@spockjs/preset-sinon-mocks* and *@spockjs/preset-jest-mocks* packages simply import the interaction processors from those separate packages and re-export them in single-element arrays.

7.2 Interaction processor integration

We have created a parser for interaction declaration blocks in the previous chapter. This parser already makes up the largest portion of the *@spockjs/interaction-block* package. We can now implement the rest of the package.

The outer function shown below, having received the Babel *types* helper and the spockjs configuration object in line 1, extracts the interaction processors from the configuration and prepares them for use in a `processors` array in lines 2 to 4. It also validates that the config has at least one primary interaction processor in lines 5 to 7 to make sure the user did not forget to enable a preset for their mocking library:

```
1  export default (t: typeof BabelTypes, config: InternalConfig) ⇒ {
2    const processors = config.hooks.interactionProcessors.map(processor ⇒
3      processor(t, config),
4    );
5    if (!processors.some(({ primary }) ⇒ primary)) {
6      throw new Error(/* ... */);
7    }
8
9    return {
10     declare, // ...
11     verify, // ...
12   }
13 }
```

Because there may be auxiliary interaction processors in addition to the single usual primary processor, we allow the configured presets to specify an array of more than one interaction processor. All specified interaction processors will be applied to interaction declarations and verifications and all of the AST nodes they generate will be inserted sequentially.

The *declare* function as shown below first ensures that the interaction declaration block has the expected structure of a binary expression statement in lines 2 to 7. If it does, in line 8 the parser is called to obtain the interaction declaration, which is then passed to every interaction processor in line 10. The collective statement nodes returned from the declaration functions of the processors are used to replace the interaction declaration block in lines 9 to 11. The following code shows the *declare* function implementation, eliding error case handling:

```
1  (statementPath: NodePath<BabelTypes.Statement>) ⇒ {
2    if (statementPath.isExpressionStatement()) {
3      const expressionPath = statementPath.get('expression') as NodePath<
4        BabelTypes.Expression
5      >;
6
7      if (expressionPath.isBinaryExpression()) {
8        const declaration = parseInteractionDeclaration(expressionPath);
9        statementPath.replaceWithMultiple(
10         processors.map(({ declare }) ⇒ declare(declaration)),
11       );
12     }
13   }
14 }
```

Even more simply, the *verify* function just replaces the interaction verification block with the collective statement nodes returned from the verification functions of the processors applied to the mock object expression:

```
1  (statementPath: NodePath<BabelTypes.Statement>) ⇒ {
2    if (statementPath.isExpressionStatement()) {
3      statementPath.replaceWithMultiple(
4        processors.map(({ verify }) ⇒ verify(statementPath.node.expression)),
5      );
6    }
7  }
```

7.3 Templates

In order to generate code that calls the Sinon.JS or Jest API effectively, we will need some more tooling. We have already created some AST nodes in other places such as identifiers, and even non-leaf nodes such as call expressions, but structures like the chained calls to the fluent Sinon.JS API

require *a lot* of nodes. Creating nodes via `t.identifier` and the like was fine for the so far mostly small structures like identifiers and simple call expressions, but it would be massively cumbersome to handle for larger tree structures.

This is where *babel-template* comes into play. *babel-template* allows us to create AST nodes by writing them as source code: [McK+18b]

```
const buildRequire = template(`
  const IMPORT_NAME = require(SOURCE);
`);

const requireDecl = buildRequire({
  IMPORT_NAME: t.identifier('myModule'),
  SOURCE: t.stringLiteral('my-module'),
});
```

This capability is also known as *quasiquotes* or *quasiquotation*. [Baw99] Parts of the AST — in the example above, `IMPORT_NAME` and `SOURCE` — can be dynamically substituted with other nodes. We will use this feature extensively for the implementation of both the Sinon.JS and the Jest interaction processors with the direct compilation approach.

7.4 Sinon.JS

As we will see, Sinon.JS (*Sinon*) provides a quite rich stub and mock API that we can leverage to make compiling interaction blocks to Sinon calls very straightforward.

On the other hand, Sinon is peculiar in that it has a notion of mock objects with methods on them as an alternative to plain mock functions: [JRC18a]

```
const obj = { method: () => {} };
const mock = sinon.mock(obj);
mock.expects("method").once().throws();
```

If we see a member expression like `obj.method()` instead of just `func()`, we can assume that it is such a mock object and need to make sure that we generate the proper `expects` call like in the example above. For plain mock functions, we can skip the `expects`:

```
mockFunc.once().throws();
```

In the following, we will focus on the most complex case: a combined interaction declaration (i.e. mocking and stubbing) with a function that is a member of a mock object, e.g.:

```
1 * mock.method(42) >> 1337;
```

All the other cases with stubbing/mocking only or simple functions instead of mock objects are effectively just boiled down versions of this case. As usual, the full source code of *@spockjs/interaction-processor-sinon-mocks* can be found in the appendix, along with test cases that verify the behavior of spockjs in conjunction with Sinon for both implementation approaches.

7.4.1 Declaration

The *declare* function for this case starts by checking whether the interaction declaration is of type *combined* in line 4, checking whether the mock object is a member expression in line 7, and extracting the data from the declaration and member expression in lines 2 and 5 and 8 to 12:

```
1  interaction ⇒ {
2    const { mockObject, args } = interaction;
3
4    if (interaction.kind === 'combined') {
5      const { cardinality, returnValue } = interaction;
6
7      if (t.isMemberExpression(mockObject)) {
8        const {
9          object: mock,
10         property: method,
11         computed
12       } = mockObject;
```

Before generating code, we need to do one more thing. Because the member expression could have a computed property name (`a["b"]` instead of `a.b`) we cannot always simply wrap the name of the identifier in a string literal and pass that to the Sinon *expects* function as shown in lines 18 to 20. In the case of a computed property name, we instead stringify it and pass it to *expects* in lines 14 to 17:

```
13       const methodName = computed
14         ? t.callExpression(
15             t.identifier('String'),
16             [method]
```

```
17            )
18          : t.stringLiteral(
19              (method as Identifier).name
20            );
```

Finally, we instantiate the Babel template for a combined interaction declaration with the nodes we have extracted and computed:

```
21        return declareMockAndStubInteraction({
22          MOCK: mock,
23          METHOD_NAME: methodName,
24          ARGS: args,
25          CARDINALITY: cardinality,
26          RETURN_VALUE: returnValue,
27        });
28      }
29    }
30  }
```

The corresponding template is:

```
1  const declareMockAndStubInteraction = template(`
2    MOCK
3      .expects(METHOD_NAME)
4      .withArgs(ARGS)
5      .atLeast(CARDINALITY)
6      .atMost(CARDINALITY)
7      .returns(RETURN_VALUE);
8  `);
```

The other templates for the simpler cases

- omit the *expects* for plain mock functions,
- omit the *atLeast* and *atMost* for stub-only declarations, or
- omit the *returns* for mock-only declarations.

We could generate the templates dynamically instead of writing them out for all the cases, but it does not appear to be worthwile for the small number of cases. Also, mocks and stubs are considered different object types by Sinon, so even though parts of the API — like *withArgs* — are identical, it would be awkward to treat them as the same thing and just plug in function calls to our liking.

7.4.2 Verification

We recorded all declarations by converting them into a format understood by Sinon and passing them to the Sinon mock object API. Given that the mock object now already holds all the state that is required to compare the expected and actual calls, we can simply generate a call to the *verify* function of the mock object using the following template:

```
const verify = template(`
  MOCK.verify();
`);
```

7.5 Jest

Jest out of the box only provides a few simple and non-chainable mock function matchers, such as `expect(mockFn).toBeCalled()` and the more specific `.toBeCalledWith(arg0, arg1)`. However, Jest mocks do record all of the calls to them and provide access to them via `mockFn.mock.calls`. We can build our own simplistic call matching logic on top of that interface, although it means that we will need to generate a lot more and a lot more complex code compared to the Sinon interaction processor. Note that Jest mock functions (*mocks*) are also used as stubs; Jest has no pure stubs.

The basic idea of what the generated code should do in order to be able to memorize interaction declarations and later access their return values for stubbing and verify them by comparing to the actual calls that occurred for mocking is the following:

1. Store all declarations — including arguments, cardinality and return value — in an array hidden inside the mock. In this case, the key to access the array on the mock will be:

   ```
   const symbol = 'Symbol.for("spockjsInteractDecls")';
   ```

2. When the function is called, stub out its return value by looking for an interaction declaration with the correct arguments in the store and using the return value specified in that declaration.

3. When the mock is verified, iterate through those of its interaction declarations that have a cardinality specified (i.e. that are mocking or combined declarations) and for each one of them, ensure that exactly the specified number of calls with the correct arguments has occurred.

7.5.1 Declaration

The Jest interaction processor generates not just a single statement like the one for Sinon, but a block statement composed of two parts: The *declareInteraction* template and the *initStub* template.

The *declareInteraction* template shown below first performs a sanity check to ensure that the *mockObject* from the user's interaction declaration is actually a Jest mock function and prints an error if that is not the case. Afterwards, it adds a declaration to the store of the mock in lines 3 to 10, if necessary initializing the store with an empty array beforehand in line 4:

```
1  const declareInteraction = template(`
2    // sanity check ...
3    MOCK[${symbol}] = [
4      ...(MOCK[${symbol}] || []),
5      {
6        args: ARGS,
7        cardinality: CARDINALITY,
8        returnValue: RETURN_VALUE,
9      }
10   ];
11 `);
```

When the template is instantiated for a concrete interaction declaration, the mock, args, cardinality, and return value placeholders are filled with the AST nodes from the interaction declaration, with the possibility of one of the latter two being `undefined` identifiers if the declaration was not a combined interaction declaration.

The *initStub* template shown below uses the *mockImplementation* feature of Jest mock functions to inject behavior into the mock function. Whenever it is called after this initialization, it will look for an interaction declaration with matching arguments in the store in lines 5 to 7 and return the associated return value for that interaction, defaulting to `undefined` if no declaration matches:

```
1  const initStub = template(`
2    STUB.mockImplementation(
3      (...actual) ⇒
4        (
5          STUB[${symbol}].find(({ args: expected }) ⇒
6            ${deepStrictEqual}(actual, expected),
7          ) || {}
8        ).returnValue,
9    );
10 `);
```

The *deepStrictEqual* helper used in line 6 is an imported function that compares the contents of object and array structures instead of their shallow identities, so that for instance the following declaration and call would match:

```
stub: s({}) >> 42;
s({}) ⩵ 42;
```

7.5.2 Verification

The *verify* template shown below also performs the sanity check to ensure that the given value is actually a Jest mock function and afterwards applies the verification algorithm as described earlier:

1. Retrieve the interaction declarations from the mock (line 3).
2. Filter out all the interaction declarations that have no cardinality (i.e. that are meant for stubbing only, line 4).
3. For each remaining interaction declaration (line 5):
   a) Retrieve the calls that actually occurred from the mock (line 6).
   b) Filter out calls with arguments that differ from the *args* of the interaction declaration (lines 6 to 8).
   c) Count the remaining calls (line 8).
   d) If the count is not equal to the *cardinality* of the interaction declaration (line 9), throw an error (lines 10 to 15).

The following template implements this algorithm:

```
1  const verify = template(`
2    // sanity check ...
3    (MOCK[${symbol}] || [])
4      .filter(({ cardinality }) ⇒ cardinality ≠ null)
5      .forEach(({ args: expected, cardinality: expectedTimes }) ⇒ {
6        const __spockjs_actualTimes = MOCK.mock.calls.filter(actual ⇒
7          ${deepStrictEqual}([ ...actual], [ ...expected]),
8        ).length;
9        if (__spockjs_actualTimes ≠≠ expectedTimes) {
10         const __spockjs_args = ${prettyFormat}(expected);
11         throw new Error(
12           \`Expected \${expectedTimes} call(s) to mock '\${MOCK_NAME}' \` +
13             \`with arguments \${__spockjs_args}, \` +
14             \`but received \${__spockjs_actualTimes} such call(s).\`,
15         );
16       }
17     });
18 `);
```

When instantiating the template, the *MOCK_NAME* placeholder in line 12 will be replaced with a string literal containing the printed *mockObject* AST node. The error message is thus made more useful by referring to the mock like it was specified in the original interaction declaration. For example, the code

```
const mock = jest.fn();
mock: 1 * mock();
verify: (true ? mock: other);
```

would result in the error message:

```
Expected 1 call(s) to mock '(true ? mock: other)' ...
```

The full code of *@spockjs/interaction-processor-jest-mocks*, including the template instantiation, is included in the appendix.

7.6 Summary

In this chapter, we implemented interaction block handling with the backing mocking libraries Sinon and Jest using the first of our two conceived approaches.

With this approach, library-specific *interaction processors* do most of the work, using *babel-template*, which we also learned about in this chapter, to generate chunks of code that call APIs provided by the mocking library and optionally performing arbitrary additional computations, such as in the case of Jest.

8 Runtime dispatch

In this chapter, we will develop alternative interaction block implementations for the Sinon.JS and Jest mocking libraries using *interaction runtime adapters* as well as their integration into the Babel plugin in the *@spockjs/interaction-block* package. Unlike the direct compilation method, interaction blocks always compile to the same calls, but the callee is a runtime adapter imported from a different package depending on which mocking library calls should be delegated to. Direct comparisons between this approach and the direct compilation approach are saved for the following chapter; this chapter merely describes the approach without evaluating it.

8.1 Presets

With the runtime dispatch approach, mocking library presets each have to export the module name of an *interactionRuntimeAdapter*. When interaction blocks are used, this module name will be used as the source for importing the adapter that accepts the generated declaration and verification calls.

An interaction runtime adapter exports the *declare* and *verify* functions with the following type signatures:

```
let declare: (
  declaration: RuntimeInteractionDeclaration,
) ⇒ void;
let verify: (
  mockObject: any
) ⇒ void;
```

Note that the *RuntimeInteractionDeclaration* type of the *declare* parameter is not the same as the *InteractionDeclaration* type that the declaration parser has as its output type, which is called *CompileTimeInteractionDeclaration* with this approach because of this distinction. Instead of the AST

nodes of a *CompileTimeInteractionDeclaration*, a *RuntimeInteractionDeclaration* contains the actual runtime values of an interaction declaration. For example, its *cardinality* is of type *number* instead of *Expression* and its *mockObject* is of type *any* instead of *Expression*.

Of course, the Sinon.JS and Jest mocking library presets do not contain the corresponding interaction processors themselves with this approach either. The *interactionRuntimeAdapter* module names they export are those of the interaction runtime adapter packages *@spockjs/interaction-runtime-adapter-sinon* and *@spockjs/interaction-runtime-adapter-jest*.

8.2 Interaction runtime adapter integration

With the runtime dispatch approach, the *@spockjs/interaction-block* package — apart from the parser — looks very different and has to do a lot more work.

One part that looks somewhat similar is the outer function:

```
export default (t: typeof BabelTypes, config: InternalConfig) ⇒ {
  const {
    hooks: { interactionRuntimeAdapter, interactionVerificationPostProcessors },
  } = config;
  const postProcessors = interactionVerificationPostProcessors.map(processor ⇒
    processor(t, config),
  );

  if (!interactionRuntimeAdapter) {
    throw new Error(/* ... */);
  }

  return {
    declare, // ...
    verify, // ...
  };
};
```

Instead of ensuring there is at least one primary interaction processor, we now ensure that there is an interaction runtime adapter defined.

One might notice that there are still some sort of interaction processors in this implementation, called *interactionVerificationPostProcessors*. These are still required as a kind of replacement for the non-primary interaction processors from the direct compilation approach, because for example *@spockjs/preset-runner-ava* needs to add a passing assertion after the mock verification because AVA makes tests fail if they run without performing an assertion through AVA.

8.2.1 Declaration serialization

The *declare* function does quite a lot more than just delegating through with the runtime dispatch approach. After the usual AST structure checks and the call to the parser, we need to *serialize* the compile-time interaction declaration to AST nodes that will create a runtime interaction declaration object to be passed to the interaction runtime adapter when the code is executed. Generating the adapter call is simple:

```
1  statementPath.replaceWith(
2    t.callExpression(
3      addNamed(
4        statementPath,
5        'declare',
6        interactionRuntimeAdapter,
7      ) as BabelTypes.Expression,
8      [t.objectExpression(interactionDeclarationProperties)],
9    ),
10 );
```

Babel provides helper functions to conveniently create imports in its *babel-helper-module-imports* package. We use the *addNamed* function to generate an import like

```
import {
  declare,
} from '@spockjs/interaction-runtime-adapter-example';
```

in lines 3 to 7. Its return value is an expression evaluating to the imported value, which is the *declare* function of the interaction runtime adapter, so we can generate a call to that function with a runtime interaction declaration as the argument in line 8.

Obtaining the *interactionDeclarationProperties* requires some more work. The first two properties are simple — the *kind* and *args* property nodes are created according to the following pattern:

```
t.objectProperty(
  t.identifier('kind'),
  t.stringLiteral(interactionDeclaration.kind),
)
```

A *cardinality* property is added in the same way, but only if the interaction declaration is of type *mock* or *combined*. A *returnValue* property is added in the same way, but only if the interaction declaration is of type *stub* or *combined*.

Finally, the *mockObject* property is a bit more complex. When implementing the first approach, we have already noticed that for the Sinon.JS mocking library we need to distinguish between member expressions and other expressions in the mock object, because the former must result in an *expects* call to Sinon.JS. This distinction cannot be deferred to the runtime, because there is no way for the runtime adapter to find out how the *mockObject* value passed in the runtime interaction declaration was obtained.

So we introduce another property on the *RuntimeInteractionDeclaration* types: The *methodName*. If the mock object was specified as a member expression, the *mockObject* property will only contain the object that the member access occurs on, while the *methodName* property specifies the name of the member of that object. Otherwise, *methodName* will be `undefined`. This distinction is implemented mostly in the same way as in the direct compilation approach; the full source code of *@spockjs/interaction-block* can be found in the appendix.

8.2.2 Verification

The *verify* function is a lot simpler because there is no complex data structure to serialize — all the interaction runtime adapter needs is the mock object. After verifying that the statement is an expression statement, we generate an interaction runtime adapter call with the corresponding import similar to the *declare* function:

```
1  statementPath.replaceWithMultiple([
2    t.callExpression(
3      addNamed(
4        statementPath,
5        'verify',
6        interactionRuntimeAdapter,
7      ) as BabelTypes.Expression,
8      [statementPath.node.expression],
9    ),
10   ...postProcessors.map(postProcess ⇒
11     postProcess(statementPath.node.expression),
12   ),
13 ]);
```

The only significant differences are that the argument passed to the adapter is a lot simpler to generate (line 8) and that we insert additional statements generated by the post processors for reasons explained earlier in lines 10 to 12.

8.3 Sinon.JS

To implement *@spockjs/interaction-runtime-adapter-sinon*, we use the same Sinon.JS (*Sinon*) APIs as with the direct compilation approach, but because we are now operating at runtime, we can call them directly instead of generating code that does so. We can also simply branch to handle different kinds of interaction declarations and mock objects instead of building templates for all combinations.

The following code implements the *declare* function of the adapter:

```
1  declaration ⇒ {
2    let mock = declaration.mockObject;
3
4    const { methodName } = declaration;
5    if (methodName ≠ null) {
6      mock = mock.expects(methodName);
7    }
8    mock = mock.withArgs( ... declaration.args);
9
10   if (declaration.kind ≡ 'mock' || declaration.kind ≡ 'combined') {
11     const { cardinality } = declaration;
12     mock = mock.atLeast(cardinality).atMost(cardinality);
13   }
14
15   if (declaration.kind ≡ 'stub' || declaration.kind ≡ 'combined') {
16     mock = mock.returns(declaration.returnValue);
17   }
18 };
```

In lines 4 to 7, we handle mock objects with methods in Sinon by calling the *expects* function if the *methodName* declaration property that we introduced for this special case is set. In line 8, we restrict the call matching to calls with the correct arguments as specified in the declaration. In lines 10 to 17, depending on the declaration kind, we tell Sinon to expect a certain number of matching calls, or to stub the return values on call, or to do both. The *verify* function is as simple as:

```
1  mock ⇒ {
2    mock.verify();
3  };
```

Additionally, we could introduce checks in the future to ensure that the given mock objects are really Sinon mocks — especially now that performing additional steps at runtime does not require adding code to a template to insert them at every interaction block — and throw an error with a helpful message if this is not the case instead of the obscure type errors that the adapter might run into if a user mistakenly attempts to declare or verify interactions on a value that is not a Sinon mock.

8.4 Jest

To implement *@spockjs/interaction-runtime-adapter-jest*, we use the same custom interaction declaration management on top of the Jest mock primitive that we already developed with the direct compilation approach. We will not repeat the code here, as it is very similar to the template contents. The full source code of this implementation, along with test cases for both implementation approaches of Jest-backed spockjs interaction blocks, is included in the appendix.

One small difference is that the *isMockFunction* check is converted from a negative condition with early return error handling to a positive condition with else block error handling. Because unlike the template literal, the adapter code can be type-checked, this allows *isMockFunction* to act as a *type guard* so the TypeScript type checker knows that the value must be a Jest mock inside the if block.

We also need to incorporate some extra logic compared to the direct compilation approach, because the generic interaction block handler always rewrites declarations on a member expression with the *methodName* property, not only when Sinon is used. Even though Jest does not need this special case, we have to handle it at the start of the *declare* function:

```
1 const { args, methodName } = declaration;
2 let { mockObject } = declaration;
3
4 if (methodName ≠ null) {
5   mockObject = mockObject[methodName];
6 }
```

Afterwards, the *mockObject* will be the evaluated result of the entire expression again, regardless of whether it was disassembled as a member expression, or directly passed to us as a generic expression. We continue the rest of the *declare* function, as well as the *verify* function, with essentially the same code as in the direct compilation approach.

8.5 Summary

In this chapter, we implemented interaction block handling using our second approach, in which the generated code does not depend on the mocking library in use, expect for the runtime adapter handling the interaction declarations and verifications, which can be swapped to use different mocking libraries.

The library-independent part did a lot more work with this approach, but in turn, developing individual library adapters became easier. Adding to that, much of the library-specific code from the first approach could be reused with slight modifications.

9 Conclusion

Having implemented both the *direct compilation* approach and the *runtime dispatch* approach to building an abstraction for interaction block handling over multiple mocking libraries, we now have a sufficiently concrete view on how their implementations differ in detail to compare the approaches directly. In the following, we compare the slighty less obvious and intuitive runtime dispatch approach to the direct compilation approach.

9.1 Runtime dispatch: The good

The runtime dispatch approach provides a significantly better experience for development of spockjs itself, making it easier to maintain and scale. Writing most of the library-specific code in interaction processor template strings gets cumbersome rather quickly, while interaction runtime adapter code can be written and type-checked like any other spockjs code, barely even revealing that it is executed at test runtime instead of test compile time.

For mocking libraries like Jest that do not provide APIs implementing sufficient interaction matching, thus requiring us to implement much of it ourselves, lackluster developer experience (*DX*) can rapidly become a limiting factor to both quantity and quality of the functionality that we can provide for those libraries.

It was already annoying to implement the cardinality checks in Jest mock verification. Imagine implementing and debugging an algorithm to match *ordered* mock expectations in the templates of an interaction processor. Imagine adding improved error messages for all kinds of mistakes users of spockjs might make using nested template strings with escaped characters all over. In a runtime adapter, these tasks can be performed using plain TypeScript code.

It is also easier to write the code *well* this way. For instance, while both approaches use *Symbols* [18g] to 'hide' interaction declarations inside a Jest mock instance, the direct compilation approach uses a named Symbol retrieved by the code `Symbol.for('spockjs ... ')` to access the declarations everywhere, while the runtime adapter approach can just generate a `Symbol()` once and use that to access the declarations everywhere, giving it a Symbol that is guaranteed to be unique, which is arguably a cleaner way that follows best practices.

If this were a requirement to be implemented with the direct compilation approach, which does not have the ability to easily hold global state like a runtime adapter does in the top-level scope of its module, we would need to either generate a Symbol constant into the top-level scope of each test file — which is easily done using Babel, but ceases to work if a mock object is passed around across test files because they would each have their own distinct Symbol — or we would need to store the Symbol itself somewhere in the global scope, which is not really cleaner or safer than using a named symbol as is being done in the current direct compilation implementation.

To improve upon the general spockjs DX issues of direct compilation, it might be viable to build advanced infrastructure that allows us to move template contents into separate files that can be type-checked and otherwise used regularly in development, transform those, parse them as templates and perform the substitutions. The viability of such intricate infrastructure extensions is questionable compared to using runtime dispatch right away, which appears to be the cleaner concept for optimizing the DX.

9.2 Runtime dispatch: The bad

As far as spockjs DX is concerned, there is also one negative aspect to be remarked about runtime dispatch. The spockjs integration tests now need yet another transpilation step, resulting in even more complexity in the development setup. In addition to

- transforming the spockjs test files with spockjs itself (which is necessary because spockjs is *self-hosting* in that its very own tests use spockjs blocks as well),
- transforming the spockjs TypeScript source files in order to perform aforesaid transformation, and

- transforming test files in the spawned test runner child process (because *integration* tests always perform a full test runner execution like a user would),

we now also need the spawned child process to be capable of compiling TypeScript on-the-fly in the runtime adapter modules imported by the test files.

As confusing as this may sound, it is somewhat specific to the spockjs development infrastructure, it is not an unsolvable problem in development, and it does not arise at all in production, where the spockjs TypeScript source code is transpiled ahead-of-time. To see how the the various transpilation steps used during spockjs development are set up, refer to the top-level and integration test directories of spockjs. [SC18b]

A more significant downside to the runtime adapter approach is that the automated escape hatch for users who want to stop using spockjs mentioned in Chapter 6 can no longer be used to obtain test files that work completely independent of spockjs. With the direct compilation approach, at least for test runners like Sinon, the escape hatch will just replace the interaction blocks with the corresponding Sinon API calls. Even for Jest, it will produce working tests by inserting all of the interaction matching logic all over the tests, although that of course does not leave the tests in a maintainable state. With the runtime adapter approach, the escape hatch will leave the tests with imports and calls to a spockjs interaction runtime adapter in them, which some may still consider valuable because the tests no longer need a transformation step for spockjs blocks, but it does not fully meet the purpose of the escape hatch, that is removing all traces of spockjs as if it was never used.

While the runtime dispatch approach is arguably more comfortable for spockjs development, it lags behind a bit in achieving great user experience (*UX*). Showing a snippet of the original code in an error message, like the direct compilation Jest interaction processor does, is not possible in a runtime adapter unless we serialize some more information from the AST in the library-agnostic interaction block handling routine. This leads us to the more general problem space of information loss during transition from compile time to runtime.

9.3 Information loss

During compilation of tests with the spockjs Babel plugin, we naturally incur some information loss. At runtime, we will no longer have access to the AST data. If we need to access that data, for example to print better error messages, we have to explicitly preserve the information during compilation, which can be done to different degrees, such as

- serializing pieces of information as they are needed at runtime, e.g. `"mockName": "(0 || mock)"`,
- serializing the *mockObject* AST node, which can be printed or used in other ways at runtime, and
- serializing the entire interaction declaration subtree for maximum information preservance, which is usually overkill and can have a noticeable performance impact.

We also need to be wary about self-inflicted information loss, avoiding it before it becomes another problem on top of natural information loss. Suppose that Sinon had no *atLeast* and *atMost* functions for specifying cardinality n, instead requiring users to register the same mock expectation n times. The correct handling of this situation would be to loop n times at runtime to register those expectations with Sinon. If we instead decided to unfold the declarations at compile time so that `2 * fn()` becomes the same as `1 * fn(); 1 * fn()` in our internal declaration data structure, we could no longer use any opportunity to specify the cardinality directly offered by other mocking libraries, and we would be forced to always use complex queueing declaration semantics as described in Chapter 6, including for our own declaration matching implementations such as the Jest adapter. Deferring the computation of derived data representations to runtime can save us from a lot of trouble.

The kind of self-inflicted information loss we described here is always avoidable; it is just a matter of designing the right abstractions and data structures. The natural information loss during compilation is avoided entirely with the direct compilation approach by doing all the work at compile time straight away; if we urgently need the naturally lost information, the runtime dispatch approach requires extra serialization effort for all of this information. As long as we do not need large parts of this information though, runtime dispatch is a lot easier to handle for us developers.

Hybrids Of course, another option could be building a bridge between the two approaches. With the runtime dispatch approach, it is still possible to introduce processors that hook into the runtime adapter call generation, serializing additional compile-time information to be passed into the adapter or directly generating other statements.

Moreover, to fully 'mix' both approaches, we can *embed* runtime dispatch into direct compilation. This is done by making the *@spockjs/interaction-block* implementation that generates adapter calls in the runtime dispatch approach just another *processor* of the direct compilation approach. This way, e.g. the Sinon preset can just specify its *interaction-processor-sinon-mocks* directly, while e.g. the Jest preset can specify the generic processor that serializes and then delegates to a runtime adapter with the *interaction-runtime-adapter-jest* module name as an argument.

9.4 Spockjs

Every project that needs to generate a lot of complex code at compile time will need to make the decision whether do defer this work to runtime. As a guiding principle, setting aside other considerations like compilation and runtime performance cost, which may be highly relevant in some other contexts, we recommend basing this choice on the amount of required information that is contained in the AST but is not present at runtime, because every such piece of information will necessitate more serialization effort to transfer it from compile time to runtime, adding more complexity to the low baseline of implementations like our runtime dispatch.

In spockjs, we will go with the cleaner runtime dispatch approach for now, eventually pulling out some parts of the logic to compile time and going some of the way towards a hybrid approach once the small improvements that require compile-time information have accumulated and gotten out of hand.

Other future features of spockjs, such as the snapshot testing blocks with support for the snapshot implementations of multiple test runners, will require the same kind of abstraction that we built for mocking libaries, albeit with much less complexity. We will use our findings in this paper to estimate the complexity of compile time-centric versus runtime-centric approaches to implementing those features without fully implementing all of the approaches like we did for interaction testing support.

Bibliography

[17a] *The Python Language Reference*. 3.6.4. Python Software Foundation. Dec. 2017. URL: http://web.archive.org/web/20171228181357/https://docs.python.org/3/reference/index.html (visited on 12/28/2017).

[17b] *Writing Build Scripts - Gradle User Guide*. 4.1. Gradle Inc. Aug. 2017. URL: https://web.archive.org/web/20170816161353/https://docs.gradle.org/current/userguide/writing_build_scripts.html (visited on 08/16/2017).

[18a] *facebook/jest: Delightful JavaScript Testing*. Facebook, Inc. June 2018. URL: https://github.com/facebook/jest/tree/9c1c3b1f072175e434943d1eb99d84b4d72ed458 (visited on 06/13/2018).

[18b] *Groovy Language Documentation*. 2.4.15. The Apache Software Foundation. May 2018. URL: https://web.archive.org/web/20180510231817/http://docs.groovy-lang.org/latest/html/documentation/index.html (visited on 05/10/2018).

[18c] *Kotlin Language Documentation*. JetBrains s.r.o. June 2018. URL: https://web.archive.org/web/20180616181208/https://kotlinlang.org/docs/kotlin-docs.pdf (visited on 06/16/2018).

[18d] *Mock Functions — Jest*. Facebook, Inc. May 2018. URL: https://web.archive.org/web/20180530003314/https://facebook.github.io/jest/docs/en/mock-functions.html (visited on 05/30/2018).

[18e] *Node.js Documentation*. 10.4.0. Node.js Foundation. June 2018. URL: https://web.archive.org/web/20180611120802/https://nodejs.org/api/all.html (visited on 06/11/2018).

[18f] *npm*. npm, Inc. July 2018. URL: https://web.archive.org/web/20180716202453/https://www.npmjs.com/ (visited on 07/16/2018).

[18g] *Symbol - MDN Web Docs Glossary*. Mozilla Corporation. Aug. 2018. URL: https://web.archive.org/web/20180811150650/https://developer.mozilla.org/en-US/docs/Glossary/Symbol (visited on 08/11/2018).

[18h] *Workspaces | Yarn*. Facebook, Inc. June 2018. URL: https://web.archive.org/web/20180718223353/https://yarnpkg.com/lang/en/docs/workspaces/ (visited on 07/18/2018).

[Baw99] *Quasiquotation in Lisp*. 1999, pp. 4–12. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.5.2566.

[DC18] Kent C. Dodds and Contributors. *kentcdodds/babel-plugin-preval: Pre-evaluate code at build-time*. July 2018. URL: https://github.com/kentcdodds/babel-plugin-preval/tree/b0f94b9ee79da47e485d95e8253749596440ea6d (visited on 07/16/2018).

[Fow07] Martin Fowler. *Mocks Aren't Stubs*. Jan. 2007. URL: https://web.archive.org/web/20180528203204/https://martinfowler.com/articles/mocksArentStubs.html (visited on 05/28/2018).

[Gam+95] Erich Gamma et al. *Design patterns: elements of reusable object-oriented software*. Pearson Education India, 1995.

[Gru04] John Gruber. *Markdown*. Dec. 2004. URL: https://web.archive.org/web/20180601070129/https://daringfireball.net/projects/markdown/ (visited on 06/01/2018).

[Her+18] Dave Herman et al. *The ESTree Spec*. May 2018. URL: https://github.com/estree/estree/tree/df2290b23a652e1ec354b6775459a3b42e22f108 (visited on 05/31/2018).

[How78] William E Howden. "Theoretical and empirical studies of program testing". In: *IEEE Transactions on Software Engineering* 4 (1978), pp. 293–298.

[Jon03] Joel Jones. "Abstract syntax tree implementation idioms". In: *Proceedings of the 10th conference on pattern languages of programs (plop2003)*. 2003, pp. 1–10.

[JRC18a] Christian Johansen, Morgan Roderick, and Contributors. *Mocks - Sinon.JS*. June 2018. URL: https://web.archive.org/web/20180617135730/http://sinonjs.org/releases/v6.0.0/mocks/ (visited on 06/17/2018).

[JRC18b] Christian Johansen, Morgan Roderick, and Contributors. *sinonjs/sinon: Test spies, stubs and mocks for JavaScript*. Aug. 2018. URL: https://github.com/sinonjs/sinon/tree/f158f4fbe2d40805ecba3213039817e5eaf10006 (visited on 08/10/2018).

[JRC18c] Christian Johansen, Morgan Roderick, and Contributors. *Stubs - Sinon.JS*. June 2018. URL: https://web.archive.org/web/20180617135700/http://sinonjs.org/releases/v6.0.0/stubs/ (visited on 06/17/2018).

[KC17] James Kyle and Contributors. *Babel Plugin Handbook*. Dec. 2017. URL: https://github.com/jamiebuilds/babel-handbook/blob/c6d34a64e212b1ffa63eca1e79e033c9b8f8bf54/translations/en/plugin-handbook.md (visited on 05/31/2018).

[LC18a] Jake Luer and Contributors. *Assert — Chai*. June 2018. URL: https://web.archive.org/web/20180617130025/http://www.chaijs.com/api/assert/ (visited on 06/17/2018).

[LC18b] Jake Luer and Contributors. *Expect / Should — Chai*. June 2018. URL: https://web.archive.org/web/20180611124604/http://www.chaijs.com/api/bdd/ (visited on 06/11/2018).

[McK+18a] Sebastian McKenzie et al. *Babel · The compiler for next generation JavaScript*. July 2018. URL: https://web.archive.org/web/20180701220534/https://babeljs.io/ (visited on 07/01/2018).

[McK+18b] Sebastian McKenzie et al. *babel-template*. Aug. 2018. URL: https://web.archive.org/web/20180813184223/https://babeljs.io/docs/en/next/babel-template.html (visited on 08/13/2018).

[McK+18c] Sebastian McKenzie et al. *babel/babel: Babel is a compiler for writing next generation JavaScript*. July 2018. URL: https://github.com/babel/babel/tree/935533cff3c15ca1c779293cc880bbd983402462 (visited on 07/16/2018).

[McK+18d] Sebastian McKenzie et al. *Using Babel - How to use Babel with your tool of choice.* June 2018. URL: https://web.archive.org/web/20180617160806/https://babeljs.io/en/setup (visited on 06/17/2018).

[Mes04] Gerard Meszaros. "A pattern language for automated testing of indirect inputs and outputs using xunit". In: *Proceedings of 11th Conference on Pattern Languages of Programs (PLoP2004)*. 2004.

[MSB11] Glenford J Myers, Corey Sandler, and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[NC18] Peter Niederwieser and Contributors. *Spock Framework Reference Documentation*. 1.1. Jan. 2018. URL: https://web.archive.org/web/20180119193934/http://spockframework.org/spock/docs/1.1/all_in_one.html (visited on 01/19/2018).

[PJ98] Jens Palsberg and C Barry Jay.
"The essence of the visitor pattern". In: *Computer Software and Applications Conference, 1998. COMPSAC'98. Proceedings. The Twenty-Second Annual International*. IEEE. 1998, pp. 9–15.

[PS96] Jan Peleska and Michael Siegel.
"From testing theory to test driver implementation".
In: *International Symposium of Formal Methods Europe*. Springer. 1996, pp. 538–556.

[SC18a] Tim Seckinger and Contributors.
spockjs interaction blocks implemented via direct compilation.
Aug. 2018.
URL: `https://github.com/spockjs/spockjs/tree/experimental-mocking-direct-compilation`
(visited on 08/16/2018).

[SC18b] Tim Seckinger and Contributors.
spockjs interaction blocks implemented via runtime dispatch.
Aug. 2018.
URL: `https://github.com/spockjs/spockjs/tree/experimental-mocking-runtime-dispatch` (visited on 08/16/2018).

[SC18c] Tim Seckinger and Contributors. *spockjs/spockjs: Structured JavaScript test cases, inspired by Spock Framework*.
July 2018.
URL: `https://github.com/spockjs/spockjs/tree/a1b488cd33bc354a1d68fc9c37579386424d8a43`
(visited on 07/18/2018).

[Sto+18] Daniel Stockman et al. *lerna/lerna: A tool for managing JavaScript projects with multiple packages*. July 2018.
URL: `https://github.com/lerna/lerna/tree/57ff865c0839df75dbe1974971d7310f235e1109`
(visited on 07/20/2018).

[Wal+18] Joe Walnes et al. *JavaHamcrest API*. 2.0.0.0. June 2018.
URL: `https://web.archive.org/web/20180611143826/http://hamcrest.org/JavaHamcrest/javadoc/2.0.0.0/index-all.html` (visited on 06/11/2018).

[WC18] Takuto Wada and Contributors.
power-assert-js/power-assert: Power Assert in JavaScript.
July 2018. URL: https://github.com/power-assert-js/power-assert/tree/161251839f9139b30fe79b8705e77f19e742e316
(visited on 07/01/2018).

[Wub+18] Mark Wubben et al.
avajs/ava: Futuristic JavaScript test runner. June 2018.
URL: https://github.com/avajs/ava/tree/09f2d87540ff5ab9d8c4875d141312dc6d3fb80b
(visited on 06/13/2018).

Appendix

The appendix contains those parts of the test case definitions and production code of the spockjs interaction block implementation that are directly relevant to this paper. The full source code — with a perhaps better reading experience — is available online. [SC18a] [SC18b]

Test cases

This section lists the test cases that an interaction block implementation needs to pass. This encompasses

- test cases with the preset for Jest mocks enabled,
- test cases with the preset for Sinon.JS mocks enabled, and
- common test cases for invalid interaction blocks.

Each test case — apart from those for invalid blocks — specifies the code that is transformed, and the result of executing the transformed code:

- Its return value,
- that it throws (with a sensible error message), or
- that it does not throw.

Jest stubs

stubs without parameters

```
1 const s = jest.fn();
2 stub: s() >> 42;
3 return s();
```

Returns: `42`

matches the stub call with the correct arguments

```
const s = jest.fn();
stub: {
  s('a', 'b') >> 41;
  s('b', 'c') >> 42;
  s('b', 'c', 'd') >> 43;
}
return s('b', 'c');
```

Returns: `42`

does not match a stub call with missing arguments

```
const s = jest.fn();
stub: s('a') >> 42;
return s();
```

Returns: `undefined`

does not match a stub call with missing arguments

```
const s = jest.fn();
stub: s('a') >> 42;
return s('b');
```

Returns: `undefined`

Jest mocks

reports too few calls

```
const mock = jest.fn();
mock: 2 * mock();

mock();

verify: mock;
```

Throws

reports no calls at all

```
const mock = jest.fn();
mock: 1 * mock();

verify: mock;
```

Throws

reports too many calls

```
const mock = jest.fn();
mock: 0 * mock();

mock();

verify: mock;
```

Throws

does not accept a call with missing arguments

```
const mock = jest.fn();
mock: 1 * mock(42);

mock();

verify: mock;
```

Throws

does not accept a call with incorrect arguments

```
const mock = jest.fn();
mock: 1 * mock(42);

mock(1337);

verify: mock;
```

Throws

does not throw if the calls were correct

```
const mock = jest.fn();
mock: 2 * mock(42);

mock(42);
mock(42);

verify: mock;
```

Does not throw

does not care about unspecified calls

```
const mock = jest.fn();
mock: 1 * mock(42);

mock(42);
mock(1337);

verify: mock;
```

Does not throw

does not care about mocks without interaction declarations

```
const mock = jest.fn();

mock(42);
mock(1337);

verify: mock;
```

Does not throw

reports too few calls with one of multiple argument lists

```
const mock = jest.fn();
mock: 2 * mock();
mock: 2 * mock(42);

mock();
mock();
mock(42);

verify: mock;
```

Throws

works with a member expression mock

```
const mock = { method: jest.fn() };
mock: 2 * mock.method();

mock.method();

verify: mock.method;
```

Throws

works with a computed mock object

```
const mock = jest.fn();
mock: 2 * mock();

mock();

verify: 0 || mock;
```

Throws

throws if the callee in an interaction declaration is not a Jest mock function

```
const fn = () => {};
mock: 2 * fn();
```

Throws

throws if the value in an interaction verification is not a Jest mock function

```
const fn = () ⇒ {};
verify: fn;
```

Throws

Jest combined

reports too few calls

```
const mock = jest.fn();
mock: 1 * mock('a') >> 1337;
mock: 4 * mock() >> 42;

mock();
mock('a');
mock();
mock();

verify: mock;
```

Throws

returns the values associated with the correct calls

```
const mock = jest.fn();
mock: 1 * mock('a') >> 1337;
mock: 4 * mock() >> 42;

const one = mock();
const two = mock('a');
const three = mock();
const four = mock();

return [one, two, three, four];
```

Returns: `[42, 1337, 42, 42]`

Sinon stubs

stubs without parameters

```
const s = sinon.stub();
stub: s() >> 42;
return s();
```

Returns: 42

matches the stub call with the correct arguments

```
const s = sinon.stub();
stub: {
  s('a', 'b') >> 41;
  s('b', 'c') >> 42;
  s('b', 'c', 'd') >> 43;
}
return s('b', 'c');
```

Returns: 42

does not match a stub call with missing arguments

```
const s = sinon.stub();
stub: s('a') >> 42;
return s();
```

Returns: undefined

does not match a stub call with incorrect arguments

```
const s = sinon.stub();
stub: s('a') >> 42;
return s('b');
```

Returns: undefined

Sinon mocks

reports too few calls

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 2 * mock.method();

obj.method();

verify: mock;
```

Throws

reports no calls at all

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method();

verify: mock;
```

Throws

reports too many calls

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 0 * mock.method();

obj.method();

verify: mock;
```

Throws

does not accept a call with missing arguments

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method(42);

obj.method();

verify: mock;
```

Throws

does not accept a call with incorrect arguments

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method(42);

obj.method(1337);

verify: mock;
```

Throws

does not throw if the calls were correct

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 2 * mock.method(42);

obj.method(42);
obj.method(42);

verify: mock;
```

Does not throw

does not care about unspecified methods

```
const obj = { method: () => {}, other: () => {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method(42);

obj.method(42);
obj.other();

verify: mock;
```

Does not throw

reports the occurrence of an unspecified call

```
const obj = { method: () => {}, other: () => {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method(42);

obj.method(42);
obj.other();

verify: mock;
```

Throws

reports too few calls on one of multiple methods

```
const obj = { method: () => {}, other: () => {} };

const mock = sinon.mock(obj);
mock: 2 * mock.method();
mock: 2 * mock.other();

obj.method();
obj.method();
obj.other();

verify: mock;
```

Throws

reports too few calls with one of multiple argument lists

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 2 * mock.method();
mock: 2 * mock.method(42);

obj.method();
obj.method();
obj.method(42);

verify: mock;
```

Throws

reports too few calls on one of multiple identical interactions

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 2 * mock.method();
mock: 2 * mock.method();

obj.method();
obj.method();
obj.method();

verify: mock;
```

Throws

works with a computed method name

```
const obj = { method: () => {} };

const mock = sinon.mock(obj);
mock: 2 * mock['method']();

obj.method();

verify: mock;
```

Throws

works with a function expectation

```
const mock = sinon.mock('mock');
mock: 2 * mock();

mock();

verify: mock;
```

Throws

Sinon combined

reports too few calls

```
const obj = { method: () ⇒ {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method('a') >> 1337;
mock: 2 * mock.method() >> 42;
mock: 2 * mock.method() >> 43;

obj.method();
obj.method('a');
obj.method();
obj.method();

verify: mock;
```

Throws

returns the values associated with the correct calls

```
const obj = { method: () ⇒ {} };

const mock = sinon.mock(obj);
mock: 1 * mock.method('a') >> 1337;
mock: 2 * mock.method() >> 42;
mock: 2 * mock.method() >> 43;

const one = obj.method();
const two = obj.method('a');
const three = obj.method();
const four = obj.method();

return [one, two, three, four];
```

Returns: `[42, 1337, 42, 43]`

reports too few calls with a function expectation

```
const mock = sinon.mock('mock');
mock: 2 * mock() >> 42;

mock();

verify: mock;
```

Throws

returns the correct value with a function expectation

```
const mock = sinon.mock('mock');
mock: 2 * mock() >> 42;

return mock();
```

Returns: `42`

Invalid block

throws if a verification is not an expression statement

```
verify: if(true);
```

throws if a declaration is not an expression statement

```
stub: if(true);
```

throws if a declaration is a non-binary expression statement

```
stub: 42;
```

throws if a declaration does not have a function call

```
stub: asdf >> 42;
```

throws if a declaration has a binary expression with the wrong operator

```
stub: asdf() + 42;
```

throws if a combined declaration has a nested binary expression with the wrong operator

```
stub: 1 + asdf() >> 42;
```

Implementation approaches

This section contains the essential sections of spockjs source code that implement interaction blocks via direct compilation and runtime dispatch, respectively. Parts that are identical or very similar in both implementations are consolidated in the 'common' subsection.

Common

packages/babel-plugin-spock/src/index.ts

```
import { PluginObj } from '@babel/core';
import { NodePath } from '@babel/traverse';
import * as BabelTypes from '@babel/types';

import { extractConfigFromState } from '@spockjs/config';

import assertifyStatement, {
  labels as assertionBlockLabels,
} from '@spockjs/assertion-block';
import transformInteractionDeclarationStatement, {
```

```
  declarationLabels as interactionDeclarationLabels,
  verificationLabel as interactionVerificationLabel,
} from '@spockjs/interaction-block';

const transformLabeledBlockOrSingle = (
  transform: (statementPath: NodePath<BabelTypes.Statement>) => void,
  path: NodePath<BabelTypes.LabeledStatement>,
): void => {
  const labeledBodyPath = path.get('body') as NodePath<BabelTypes.Statement>;

  switch (labeledBodyPath.type) {
    case 'BlockStatement':
      // power-assert may add statements in betweeen,
      // so never reuse body array
      const statementPaths = () =>
        (labeledBodyPath.get('body') as any) as NodePath<
          BabelTypes.Statement
        >[];

      statementPaths().forEach(transform);

      // remove label
      path.replaceWithMultiple(statementPaths().map(stmtPath => stmtPath.node));
      break;
    default:
      transform(labeledBodyPath);

      // remove label
      if (!path.removed) {
        path.replaceWith(labeledBodyPath);
      }
  }
};

export default (babel: { types: typeof BabelTypes }): PluginObj => ({
  visitor: {
    LabeledStatement(path, state) {
      const config = extractConfigFromState(state);
      const label = path.node.label.name;

      // assertion block
      if (assertionBlockLabels.includes(label)) {
        transformLabeledBlockOrSingle(
          assertifyStatement(babel, state, config),
          path,
        );
      }

      // interaction block
      else if (interactionDeclarationLabels.includes(label)) {
        transformLabeledBlockOrSingle(
          transformInteractionDeclarationStatement(babel.types, config).declare,
          path,
        );
      } else if (label === interactionVerificationLabel) {
        transformLabeledBlockOrSingle(
          transformInteractionDeclarationStatement(babel.types, config).verify,
          path,
        );
      }
    },
  },
});
```

packages/interaction-block/src/parser.ts

```
import { NodePath } from '@babel/traverse';
import { BinaryExpression, Expression, Node } from '@babel/types';
import {
  BaseInteractionDeclaration,
```

```
5     CombinedInteractionDeclaration,
6     InteractionDeclaration,
7     MockInteractionDeclaration,
8     StubInteractionDeclaration,
9   } from '@spockjs/config';
10
11  const mockOperator = '*';
12  const stubOperator = '>>';
13
14  type InteractionDeclarationParser<
15    E extends Node,
16    I extends BaseInteractionDeclaration
17  > = (expressionPath: NodePath<E>) => I;
18
19  const parseCall: InteractionDeclarationParser<
20    Expression,
21    BaseInteractionDeclaration
22  > = (expressionPath: NodePath) => {
23    if (expressionPath.isCallExpression()) {
24      const {
25        node: { callee: mockObject, arguments: args },
26      } = expressionPath;
27      return {
28        mockObject,
29        args,
30      };
31    }
32    throw expressionPath.buildCodeFrameError(
33      `Expected a call expression, but got an expression of type ${
34        expressionPath.type
35      }`,
36    );
37  };
38
39  const parseMockInteractionDeclaration: InteractionDeclarationParser<
40    BinaryExpression,
41    MockInteractionDeclaration
42  > = expressionPath => ({
43    ...parseCall(expressionPath.get('right') as NodePath<Expression>),
44    kind: 'mock',
45    cardinality: expressionPath.node.left,
46  });
47
48  const parseStubInteractionDeclaration: InteractionDeclarationParser<
49    BinaryExpression,
50    StubInteractionDeclaration | CombinedInteractionDeclaration
51  > = expressionPath => {
52    const leftPath = expressionPath.get('left') as NodePath<Expression>;
53    const {
54      node: { right: returnValue },
55    } = expressionPath;
56
57    if (leftPath.isBinaryExpression()) {
58      const {
59        node: { operator },
60      } = leftPath;
61      if (operator === mockOperator) {
62        return {
63          ...parseMockInteractionDeclaration(leftPath),
64          returnValue,
65          kind: 'combined',
66        };
67      }
68      throw new Error(
69        `Expected operator '${mockOperator}' (for combined mocking and stubbing), ` +
70          `but got operator '${operator}'`,
71      );
72    }
73
74    return {
75      ...parseCall(leftPath),
76      returnValue,
77      kind: 'stub',
```

```
  };
};

const parseInteractionDeclaration: InteractionDeclarationParser<
  BinaryExpression,
  InteractionDeclaration
> = expressionPath => {
  const { node: expression } = expressionPath;
  const { operator } = expression;

  switch (operator) {
    case mockOperator:
      return parseMockInteractionDeclaration(expressionPath);
    case stubOperator:
      return parseStubInteractionDeclaration(expressionPath);
    default:
      throw expressionPath.buildCodeFrameError(
        `Expected operator '${mockOperator}' (for mocking) ` +
          `or '${stubOperator}' (for stubbing), ` +
          `but got operator '${operator}'`,
      );
  }
};

export default parseInteractionDeclaration;
```

experimental-mocking-direct-compilation

packages/config/src/hooks/interaction.ts

```
import * as BabelTypes from '@babel/types';

import { InternalConfig } from '..';

export interface BaseInteractionDeclaration {
  mockObject: BabelTypes.Expression;
  args: (BabelTypes.Expression | BabelTypes.SpreadElement)[];
}
export interface MockInteractionDeclaration extends BaseInteractionDeclaration {
  kind: 'mock';
  cardinality: BabelTypes.Expression;
}
export interface StubInteractionDeclaration extends BaseInteractionDeclaration {
  kind: 'stub';
  returnValue: BabelTypes.Expression;
}
export interface CombinedInteractionDeclaration
  extends BaseInteractionDeclaration {
  kind: 'combined';
  cardinality: BabelTypes.Expression;
  returnValue: BabelTypes.Expression;
}
export type InteractionDeclaration =
  | MockInteractionDeclaration
  | StubInteractionDeclaration
  | CombinedInteractionDeclaration;

export type InteractionProcessor = (
  t: typeof BabelTypes,
  config: InternalConfig,
) => {
  /**
   * Whether the processor is meant for standalone use (`true`),
   * as is the case for those implementing mocking library bindings,
   * or it is an auxiliary processor (`false`),
   * such as one that provides better integration with a test runner.
   * Users will see an error when attempting to use interaction blocks
   * without a primary interaction processor.
```

```
     */
    primary: boolean;
    declare(interaction: InteractionDeclaration): BabelTypes.Statement;
    verify(mockObject: BabelTypes.Expression): BabelTypes.Statement;
  };
```

packages/interaction-block/src/index.tx

```
import { NodePath } from '@babel/traverse';
import * as BabelTypes from '@babel/types';

import { InternalConfig } from '@spockjs/config';

import parseInteractionDeclaration from './parser';

export default (t: typeof BabelTypes, config: InternalConfig) => {
  const processors = config.hooks.interactionProcessors.map(processor =>
    processor(t, config),
  );
  if (!processors.some(({ primary }) => primary)) {
    throw new Error(
      'Found an interaction declaration, ' +
        'but no preset defines a primary interactionProcessor.\n' +
        'You need to enable a preset for your mocking library.',
    );
  }

  return {
    declare: (statementPath: NodePath<BabelTypes.Statement>) => {
      if (statementPath.isExpressionStatement()) {
        const expressionPath = statementPath.get('expression') as NodePath<
          BabelTypes.Expression
        >;

        if (expressionPath.isBinaryExpression()) {
          const declaration = parseInteractionDeclaration(expressionPath);
          statementPath.replaceWithMultiple(
            processors.map(({ declare }) => declare(declaration)),
          );
        } else {
          throw expressionPath.buildCodeFrameError(
            `Expected a binary expression, but got an expression of type ${
              expressionPath.type
            }`,
          );
        }
      } else {
        throw statementPath.buildCodeFrameError(
          `Expected an expression statement, but got a statement of type ${
            statementPath.type
          }`,
        );
      }
    },

    verify: (statementPath: NodePath<BabelTypes.Statement>) => {
      if (statementPath.isExpressionStatement()) {
        statementPath.replaceWithMultiple(
          processors.map(({ verify }) => verify(statementPath.node.expression)),
        );
      } else {
        throw statementPath.buildCodeFrameError(
          `Expected an expression statement, but got a statement of type ${
            statementPath.type
          }`,
        );
      }
    },
  };
};
```

```

export const declarationLabels = ['stub', 'mock'];
export const verificationLabel = 'verify';
```

packages/preset-sinon-mocks/src/index.ts

```
import { InteractionProcessor } from '@spockjs/config';

import interactionProcessor from '@spockjs/interaction-processor-sinon-mocks';

export const assertionPostProcessors = [];

export const interactionProcessors: InteractionProcessor[] = [
  interactionProcessor,
];
```

packages/preset-jest-mocks/src/index.ts

```
import { InteractionProcessor } from '@spockjs/config';

import interactionProcessor from '@spockjs/interaction-processor-jest-mocks';

export const assertionPostProcessors = [];

export const interactionProcessors: InteractionProcessor[] = [
  interactionProcessor,
];
```

packages/interaction-processor-sinon-mocks/src/index.ts

```
import template from '@babel/template';
import { ExpressionStatement, Identifier } from '@babel/types';

import { InteractionProcessor } from '@spockjs/config';

// Sinon stubs do not implement `withExactArgs`,
// so we do not use it on mocks either for consistency reasons
// and just have to accept that excess arguments will go unnoticed
const declareStubInteraction = template(`
  STUB
    .withArgs(ARGS)
    .returns(RETURN_VALUE);
`);
const declareMockInteraction = template(`
  MOCK
    .expects(METHOD_NAME)
    .withArgs(ARGS)
    .atLeast(CARDINALITY)
    .atMost(CARDINALITY);
`);
const declareMockFunctionInteraction = template(`
  MOCK
    .withArgs(ARGS)
    .atLeast(CARDINALITY)
    .atMost(CARDINALITY);
`);
const declareMockAndStubInteraction = template(`
  MOCK
    .expects(METHOD_NAME)
    .withArgs(ARGS)
    .atLeast(CARDINALITY)
    .atMost(CARDINALITY)
    .returns(RETURN_VALUE);
```

```
`);
const declareMockAndStubFunctionInteraction = template(`
  MOCK
    .withArgs(ARGS)
    .atLeast(CARDINALITY)
    .atMost(CARDINALITY)
    .returns(RETURN_VALUE);
`);

const verify = template(`
  MOCK.verify();
`);

const processor: InteractionProcessor = (t, config) => ({
  primary: true,

  declare: interaction => {
    const { mockObject, args } = interaction;

    if (interaction.kind === 'mock' || interaction.kind === 'combined') {
      const { cardinality } = interaction;

      if (t.isMemberExpression(mockObject)) {
        const { object: mock, property: method, computed } = mockObject;
        const methodName = computed
          ? t.callExpression(t.identifier('String'), [method])
          : t.stringLiteral((method as Identifier).name);

        if (interaction.kind === 'combined') {
          // combined mocking and stubbing
          const { returnValue } = interaction;
          return declareMockAndStubInteraction({
            MOCK: mock,
            METHOD_NAME: methodName,
            ARGS: args as any,
            CARDINALITY: cardinality,
            RETURN_VALUE: returnValue,
          }) as ExpressionStatement;
        }

        // mocking
        return declareMockInteraction({
          MOCK: mock,
          METHOD_NAME: methodName,
          ARGS: args as any,
          CARDINALITY: cardinality,
        }) as ExpressionStatement;
      }

      if (interaction.kind === 'combined') {
        // combined mocking and stubbing with a plain function
        const { returnValue } = interaction;
        return declareMockAndStubFunctionInteraction({
          MOCK: mockObject,
          ARGS: args as any,
          CARDINALITY: cardinality,
          RETURN_VALUE: returnValue,
        }) as ExpressionStatement;
      }

      // mocking with a plain function
      return declareMockFunctionInteraction({
        MOCK: mockObject,
        ARGS: args as any,
        CARDINALITY: cardinality,
      }) as ExpressionStatement;
    }

    // stubbing
    const { returnValue } = interaction;
    return declareStubInteraction({
      STUB: mockObject,
      ARGS: args as any,
```

```
      RETURN_VALUE: returnValue,
    }) as ExpressionStatement;
  },

  verify: mockObject => {
    return verify({
      MOCK: mockObject,
    }) as ExpressionStatement;
  },
});

export default processor;
```

packages/interaction-processor-jest-mocks/src/index.ts

```
import print from '@babel/generator';
import template from '@babel/template';
import { ExpressionStatement, Statement } from '@babel/types';

import { InteractionProcessor } from '@spockjs/config';

const symbol = 'Symbol.for("spockjsInteractionDeclarations")';
const deepStrictEqual = 'require("deep-strict-equal")';
const prettyFormat = 'require("pretty-format")';

const declareInteraction = template(`
  if(!jest.isMockFunction(MOCK)) {
    const __spockjs_mock = ${prettyFormat}(MOCK);
    throw new Error(\`Expected the callee in an interaction declaration \` +
      \`to be a Jest mock function, but '\${MOCK_NAME}' is \${__spockjs_mock}\`);
  }
  MOCK[${symbol}] = [
    ...(MOCK[${symbol}] || []),
    {
      args: ARGS,
      cardinality: CARDINALITY,
      returnValue: RETURN_VALUE,
    }
  ];
`);
const initStub = template(`
  STUB.mockImplementation(
    (...actual) =>
      (
        STUB[${symbol}].find(({ args: expected }) =>
          ${deepStrictEqual}(actual, expected),
        ) || {}
      ).returnValue,
  );
`);
const verify = template(`
  if(!jest.isMockFunction(MOCK)) {
    const __spockjs_mock = ${prettyFormat}(MOCK);
    throw new Error(\`Expected the value in an interaction verification \` +
      \`to be a Jest mock function, but '\${MOCK_NAME}' is \${__spockjs_mock}\`);
  }
  (MOCK[${symbol}] || [])
    .filter(({ cardinality }) => cardinality !== null)
    .forEach(({ args: expected, cardinality: expectedTimes }) => {
      const __spockjs_actualTimes = MOCK.mock.calls.filter(actual =>
        ${deepStrictEqual}([...actual], [...expected]),
      ).length;
      if (__spockjs_actualTimes !== expectedTimes) {
        const __spockjs_args = ${prettyFormat}(expected);
        throw new Error(
          \`Expected \${expectedTimes} call(s) to mock '\${MOCK_NAME}' \` +
            \`with arguments \${__spockjs_args}, \` +
            \`but received \${__spockjs_actualTimes} such call(s).\`,
        );
```

```
      }
    });
`);

const processor: InteractionProcessor = (t, config) => ({
  primary: true,

  declare: interaction => {
    const undefinedIdentifier = t.identifier('undefined');

    const { mockObject, args } = interaction;
    const cardinality =
      interaction.kind === 'stub'
        ? undefinedIdentifier
        : interaction.cardinality;
    const returnValue =
      interaction.kind === 'mock'
        ? undefinedIdentifier
        : interaction.returnValue;

    return t.blockStatement([
      ...((declareInteraction({
        MOCK: mockObject,
        ARGS: t.arrayExpression(args),
        CARDINALITY: cardinality,
        RETURN_VALUE: returnValue,
        MOCK_NAME: t.stringLiteral(print(mockObject).code),
      }) as any) as Statement[]),
      initStub({ STUB: mockObject }) as ExpressionStatement,
    ]);
  },

  verify: mockObject => {
    return t.blockStatement((verify({
      MOCK: mockObject,
      MOCK_NAME: t.stringLiteral(print(mockObject).code),
    }) as any) as Statement[]);
  },
});

export default processor;
```

experimental-mocking-runtime-dispatch

Note that with this approach the `{Base,Mock,Stub,Combined,}InteractionDeclaration` types are renamed to `CompileTime{Base,Mock,Stub,Combined,}InteractionDeclaration` to distinguish them from the *RuntimeInteractionDeclaration* types.

packages/config/src/hooks/interaction/compile-time.ts

```
import * as BabelTypes from '@babel/types';

import { InternalConfig } from '../..';

export interface CompileTimeBaseInteractionDeclaration {
  mockObject: BabelTypes.Expression;
  args: (BabelTypes.Expression | BabelTypes.SpreadElement)[];
}
export interface CompileTimeMockInteractionDeclaration
  extends CompileTimeBaseInteractionDeclaration {
  kind: 'mock';
  cardinality: BabelTypes.Expression;
}
```

```ts
export interface CompileTimeStubInteractionDeclaration
  extends CompileTimeBaseInteractionDeclaration {
  kind: 'stub';
  returnValue: BabelTypes.Expression;
}
export interface CompileTimeCombinedInteractionDeclaration
  extends CompileTimeBaseInteractionDeclaration {
  kind: 'combined';
  cardinality: BabelTypes.Expression;
  returnValue: BabelTypes.Expression;
}
export type CompileTimeInteractionDeclaration =
  | CompileTimeMockInteractionDeclaration
  | CompileTimeStubInteractionDeclaration
  | CompileTimeCombinedInteractionDeclaration;

export type InteractionVerificationPostProcessor = (
  t: typeof BabelTypes,
  config: InternalConfig,
) => (mockObject: BabelTypes.Expression) => BabelTypes.Statement;
```

packages/config/src/hooks/interaction/runtime.ts

```ts
export interface RuntimeBaseInteractionDeclaration {
  mockObject: any;
  /**
   * For a call like `mock.method()`,
   * mockObject will be just `mock`,
   * and methodName will be `'method'`.
   */
  methodName?: string;
  args: any[];
}
export interface RuntimeMockInteractionDeclaration
  extends RuntimeBaseInteractionDeclaration {
  kind: 'mock';
  cardinality: number;
}
export interface RuntimeStubInteractionDeclaration
  extends RuntimeBaseInteractionDeclaration {
  kind: 'stub';
  returnValue: any;
}
export interface RuntimeCombinedInteractionDeclaration
  extends RuntimeBaseInteractionDeclaration {
  kind: 'combined';
  cardinality: number;
  returnValue: any;
}
export type RuntimeInteractionDeclaration =
  | RuntimeMockInteractionDeclaration
  | RuntimeStubInteractionDeclaration
  | RuntimeCombinedInteractionDeclaration;

export type InteractionDeclarationRuntimeAdapter = (
  declaration: RuntimeInteractionDeclaration,
) => void;
export type InteractionVerificationRuntimeAdapter = (mockObject: any) => void;
```

packages/interaction-block/src/index.ts

```ts
import { addNamed } from '@babel/helper-module-imports';
import { NodePath } from '@babel/traverse';
import * as BabelTypes from '@babel/types';

import { InternalConfig } from '@spockjs/config';

```

```
import parseInteractionDeclaration from './parser';

export default (t: typeof BabelTypes, config: InternalConfig) => {
  const {
    hooks: { interactionRuntimeAdapter, interactionVerificationPostProcessors },
  } = config;
  const postProcessors = interactionVerificationPostProcessors.map(processor =>
    processor(t, config),
  );

  if (!interactionRuntimeAdapter) {
    throw new Error(
      'Found an interaction declaration, ' +
        'but no preset defines an interaction runtime adapter.\n' +
        'You need to enable a preset for your mocking library.',
    );
  }

  return {
    declare: (statementPath: NodePath<BabelTypes.Statement>) => {
      if (statementPath.isExpressionStatement()) {
        const expressionPath = statementPath.get('expression') as NodePath<
          BabelTypes.Expression
        >;

        if (expressionPath.isBinaryExpression()) {
          const interactionDeclaration = parseInteractionDeclaration(
            expressionPath,
          );

          // serialize interaction declaration to object properties

          let interactionDeclarationProperties = [
            t.objectProperty(
              t.identifier('kind'),
              t.stringLiteral(interactionDeclaration.kind),
            ),
            t.objectProperty(
              t.identifier('args'),
              t.arrayExpression(interactionDeclaration.args),
            ),
          ];

          const { mockObject } = interactionDeclaration;
          if (t.isMemberExpression(mockObject)) {
            interactionDeclarationProperties = [
              ...interactionDeclarationProperties,
              t.objectProperty(t.identifier('mockObject'), mockObject.object),
              t.objectProperty(
                t.identifier('methodName'),
                mockObject.computed
                  ? mockObject.property
                  : t.stringLiteral(
                      (mockObject.property as BabelTypes.Identifier).name,
                    ),
              ),
            ];
          } else {
            interactionDeclarationProperties = [
              ...interactionDeclarationProperties,
              t.objectProperty(t.identifier('mockObject'), mockObject),
            ];
          }

          // kinds
          if (
            interactionDeclaration.kind === 'mock' ||
            interactionDeclaration.kind === 'combined'
          ) {
            interactionDeclarationProperties = [
              ...interactionDeclarationProperties,
              t.objectProperty(
                t.identifier('cardinality'),
```

```
                interactionDeclaration.cardinality,
              ),
            ];
          }
          if (
            interactionDeclaration.kind === 'stub' ||
            interactionDeclaration.kind === 'combined'
          ) {
            interactionDeclarationProperties = [
              ...interactionDeclarationProperties,
              t.objectProperty(
                t.identifier('returnValue'),
                interactionDeclaration.returnValue,
              ),
            ];
          }

          statementPath.replaceWith(
            t.callExpression(
              addNamed(
                statementPath,
                'declare',
                interactionRuntimeAdapter,
              ) as BabelTypes.Expression,
              [t.objectExpression(interactionDeclarationProperties)],
            ),
          );
        } else {
          throw expressionPath.buildCodeFrameError(
            `Expected a binary expression, but got an expression of type ${
              expressionPath.type
            }`,
          );
        }
      } else {
        throw statementPath.buildCodeFrameError(
          `Expected an expression statement, but got a statement of type ${
            statementPath.type
          }`,
        );
      }
    },

    verify: (statementPath: NodePath<BabelTypes.Statement>) => {
      if (statementPath.isExpressionStatement()) {
        statementPath.replaceWithMultiple([
          t.callExpression(
            addNamed(
              statementPath,
              'verify',
              interactionRuntimeAdapter,
            ) as BabelTypes.Expression,
            [statementPath.node.expression],
          ),
          ...postProcessors.map(postProcess =>
            postProcess(statementPath.node.expression),
          ),
        ]);
      } else {
        throw statementPath.buildCodeFrameError(
          `Expected an expression statement, but got a statement of type ${
            statementPath.type
          }`,
        );
      }
    },
  };
};

export const declarationLabels = ['stub', 'mock'];
export const verificationLabel = 'verify';
```

packages/preset-sinon-mocks/src/index.ts

```ts
import { InteractionVerificationPostProcessor } from '@spockjs/config';

export const assertionPostProcessors = [];

export const interactionRuntimeAdapter =
  '@spockjs/interaction-runtime-adapter-sinon';
export const interactionVerificationPostProcessors:
  InteractionVerificationPostProcessor[] = [];
```

packages/preset-jest-mocks/src/index.ts

```ts
import { InteractionVerificationPostProcessor } from '@spockjs/config';

export const assertionPostProcessors = [];

export const interactionRuntimeAdapter =
  '@spockjs/interaction-runtime-adapter-jest';
export const interactionVerificationPostProcessors:
  InteractionVerificationPostProcessor[] = [];
```

packages/interaction-runtime-adapter-sinon/src/index.ts

```ts
import {
  InteractionDeclarationRuntimeAdapter,
  InteractionVerificationRuntimeAdapter,
} from '@spockjs/config';

export const declare: InteractionDeclarationRuntimeAdapter = declaration => {
  let mock = declaration.mockObject;

  const { methodName } = declaration;
  if (methodName != null) {
    mock = mock.expects(methodName);
  }
  mock = mock.withArgs(...declaration.args);

  if (declaration.kind === 'mock' || declaration.kind === 'combined') {
    const { cardinality } = declaration;
    mock = mock.atLeast(cardinality).atMost(cardinality);
  }

  if (declaration.kind === 'stub' || declaration.kind === 'combined') {
    mock = mock.returns(declaration.returnValue);
  }
};
export const verify: InteractionVerificationRuntimeAdapter = mock => {
  mock.verify();
};
```

packages/interaction-runtime-adapter-jest/src/index.ts

```ts
import {
  InteractionDeclarationRuntimeAdapter,
  InteractionVerificationRuntimeAdapter,
} from '@spockjs/config';
import deepStrictEqual from 'deep-strict-equal';
import prettyFormat from 'pretty-format';

const interactionsSymbol = Symbol();
interface Interaction {
  args: any[];
```

```
    cardinality?: number;
    returnValue?: any;
  }
  interface InteractionsStore {
    [interactionsSymbol]?: Interaction[];
  }

  export const declare: InteractionDeclarationRuntimeAdapter = declaration => {
    const { args, methodName } = declaration;
    let { mockObject } = declaration;

    if (methodName != null) {
      mockObject = mockObject[methodName];
    }

    if (jest.isMockFunction(mockObject)) {
      const interactionsStore = mockObject as InteractionsStore;

      const interactions: Interaction[] = (interactionsStore[
        interactionsSymbol
      ] = [
        ...(interactionsStore[interactionsSymbol] || []),
        {
          args,
          ...(declaration.kind === 'mock' || declaration.kind === 'combined'
            ? { cardinality: declaration.cardinality }
            : {}),
          ...(declaration.kind === 'stub' || declaration.kind === 'combined'
            ? { returnValue: declaration.returnValue }
            : {}),
        },
      ]);

      mockObject.mockImplementation(
        (...actual) =>
          (
            interactions.find(({ args: expected }) =>
              deepStrictEqual(actual, expected),
            ) || { returnValue: undefined }
          ).returnValue,
      );
    } else {
      throw new Error(
        'Expected the callee in an interaction declaration to be a Jest mock function, ' +
          `but received ${prettyFormat(mockObject)}.`,
      );
    }
  };
  export const verify: InteractionVerificationRuntimeAdapter = mockObject => {
    if (jest.isMockFunction(mockObject)) {
      const interactionsStore = mockObject as InteractionsStore;

      (interactionsStore[interactionsSymbol] || [])
        .filter(({ cardinality }) => cardinality !== null)
        .forEach(({ args: expected, cardinality: expectedTimes }) => {
          const actualTimes = mockObject.mock.calls.filter(actual =>
            deepStrictEqual([...actual], [...expected]),
          ).length;
          if (actualTimes !== expectedTimes) {
            throw new Error(
              `Expected ${expectedTimes} call(s) to mock ${mockObject.getMockName()}` +
                `with arguments ${prettyFormat(expected)}, ` +
                `but received ${actualTimes} such call(s).`,
            );
          }
        });
    } else {
      throw new Error(
        'Expected the value in an interaction verification to be a Jest mock function,' +
          ` but received ${prettyFormat(mockObject}.`,
      );
    }
  };
```