Spekl Documentation

Release 0.0.6

John L. Singleton

February 16, 2016

Contents:

CHAPTER 1

Getting Started with Spekl

How can we know that our software does what it is supposed to do? Techniques like unit testing are good for increasing our confidence that a program does what it is supposed to do, but ultimately they are weak approximations. Often times it's impossible to encode all the possible edge cases into a unit test, and if it is possible, it may be extremely time consuming to do so.

What do we do? Enter Formal Methods.

Formal Methods is an area of Computer Science that aims to address the problem of verifying what software does by using mathematical models and techniques. It's also a term that strikes fear into the hearts of productive engineers everywhere.

Most Formal Methods techniques involve specifications, static checkers, runtime assertion checkers, and SMT-solvers. However, getting them to work together is often difficult and error-prone. Spekl is a platform for streamlining the process of authoring, installing, and using specifications and formal methods tools.

CHAPTER 2

Installation

Installing Spekl is easy. To install, simply download the installer for your platform from Spekl's releases releases page and run it.

At the moment, users on Linux and OSX are required to have the Git binaries installed on your path. This requirement will be lifted in future versions of Spekl. Windows users don't need to have Git installed.

CHAPTER 3

Your First Verification Project

In this section we're going to see just how easy it is to verify your programs using Spekl. Spekl supports lots of different tools like OpenJML, SAW, and FindBugs. In this section we are going to perform extended static checking on a small application to show you how easy Spekl makes the verification process.

To start, create a new directory you want to store this example in:

```
~ » mkdir my-project
~ » cd my-project
```

Next, initialize the project in that directory. You can do this with the `spm init` command. This command is interactive but for this example we are going to just accept the defaults `spm init` uses.

```
~ » spm init
[spm] INFO  - [command-init]
[spm] INFO  - [new-project] Creating new verification project...
Project Name? [default: my project]
Project Id? [default: my.project]
Project Version? [default: 0.0.1]
#
# Basic Project Information
#
name            : my project
project-id      : my.project
version         : 0.0.1

#
# Checks
#
# hint: Use spm add <your-tool> to add new checks here
#

##
## Example
##
# checks :
#   - name        : openjml-esc
#     description : "OpenJML All File ESC"
#     language    : java              # might not need this, because it is implied by the tool
#     paths       : [MaybeAdd.java]

#     tool:
#       name      : openjml-esc
#       version   : 0.0.3
```

```
#       pre_check :  # stuff to do before a check
#       post_check:  # stuff to do before a check

#     # specs:
#     #   - name: java-core
#     #     version: 1.1.1


Does this configuration look reasonable? [Y/n] y
[spm] INFO  - [new-project] Writing project file to spekl.yml
[spm] INFO  - [new-project] Done.
```

This command creates a file called `spekl.yml` in the directory you execute `spm init` in. Edit that file to look like the listing, below.

```
#
# Basic Project Information
#
name             : my project
project-id       : my.project
version          : 0.0.1

checks :
  - name         : openjml-esc
    description  : "OpenJML All File ESC"
    paths        : [MaybeAdd.java]

    tool:
      name       : openjml-esc
      pre_check :  # stuff to do before a check
      post_check:  # stuff to do before a check
```

What did we do in the listing, above? In the checks section we defined a check called `openjml-esc`. This is the extended static checker provided by OpenJML, a tool that is able to check programs written in the JML Specification Language. You don't need to know JML to follow this example, but JML is an excellent modeling language that is widely known (meaning, you should probably learn it).

Continuing with the example above, we defined just one check here. Note that we have specified that we want to use OpenJML declaratively — we haven't specified *how* to use OpenJML. Also note that OpenJML depends on things like SMT solvers which may be difficult for new users to configure. We haven't needed to specify anything about them, either.

Note that in the `paths` element we specified that we want to check the file `MaybeAdd.java`. We'll create this file next. Note that the `paths` element can contain a comma-separated list of paths that may contain wildcards. You use this to specify the files you want to run a given check on.

Next, put the following text into the file `MaybeAdd.java` in the current directory

```
public class MaybeAdd {

  //@ requires 0 < a && a < 1000;
  //@ requires 0 < b && b < 1000;
  //@ ensures  0 < \result;
  public static int add(int a, int b){
      return a-b;
  }


  public static void main(String args[]){
```

```
      System.out.println(MaybeAdd.add(1,2));

  }

}
```

In this minimal class you can see that we wrote a minimal example that (wrongly) adds two integers. Let's see what happens when we run this example with Spekl. To do that, first let's tell Spekl to install our tools:

```
~ » spm install
```

This command will kick off an installation process that will install `z3`, `openjml`, and `openjml-esc`. The output will look like the following:

```
[spm] INFO  - [command-install] Finding package openjml-esc in remote repository
[spm] INFO  - [command-install] Starting install of package openjml-esc (version: 1.7.3.20150406-5)
[spm] INFO  - [command-install] Examining dependencies...
[spm] INFO  - [command-install] Will install the following missing packages:
[spm] INFO  - [command-install] -  openjml (version: >= 1.7.3 && < 1.8)
[spm] INFO  - [command-install] -  z3 (version: >= 4.3.0 && < 4.3.1)
[spm] INFO  - [command-install] Finding package openjml in remote repository
[spm] INFO  - [command-install] Starting install of package openjml (version: 1.7.3.20150406-1)
[spm] INFO  - [command-install] Examining dependencies...
[spm] INFO  - [command-install] Installing package openjml (version: 1.7.3.20150406-1)
[spm] INFO  - [command-install] Downloading Required Assets...
openjml-dist         : [================================================] 100%
[spm] INFO  - [command-install] Running package-specific installation commands
[spm] INFO  - [command-install-scripts]  Unpacking the archive...
[spm] INFO  - [command-install] Performing cleanup tasks...
[spm] INFO  - [command-install] Cleaning up resources for asset  openjml-dist
[spm] INFO  - [command-install] Writing out package description...
[spm] INFO  - [command-install] Completed installation of package openjml (version: 1.7.3.20150406-1)
[spm] INFO  - [command-install] Finding package z3 in remote repository
[spm] INFO  - [command-install] Starting install of package z3 (version: 4.3.0-2)
[spm] INFO  - [command-install] Examining dependencies...
[spm] INFO  - [command-install] Installing package z3 (version: 4.3.0-2)
[spm] INFO  - [command-install] Downloading Required Assets...
Z3 Binaries for Windows : [================================================] 100%
[spm] INFO  - [command-install] Running package-specific installation commands
[spm] INFO  - [command-install-scripts]  Unpacking Z3...
[spm] INFO  - [command-install] Performing cleanup tasks...
[spm] INFO  - [command-install] Cleaning up resources for asset  Z3 Binaries for Windows
[spm] INFO  - [command-install] Writing out package description...
[spm] INFO  - [command-install] Completed installation of package z3 (version: 4.3.0-2)
[spm] INFO  - [command-install] Installing package openjml-esc (version: 1.7.3.20150406-5)
[spm] INFO  - [command-install] Downloading Required Assets...
[spm] INFO  - [command-install] Running package-specific installation commands
[spm] INFO  - [command-install] Performing cleanup tasks...
[spm] INFO  - [command-install] Writing out package description...
[spm] INFO  - [command-install] Completed installation of package openjml-esc (version: 1.7.3.201504
[spm] INFO  - [command-install] Installing specs....
[spm] INFO  - [command-install] Done. Use `spm check` to check your project.
```

After that completes, we can run a check with the following command:

```
~ » spm check
```

The output from the check will look like the following:

```
[spm] INFO  - [command-check] Running all checks for project...
[spm] INFO  - [command-check] Running check: OpenJML All File ESC
[spm] INFO  - Configuring solver for Z3...
[spm] INFO  - Running OpenJML in ESC Mode...

.\MaybeAdd.java:7: warning: The prover cannot establish an assertion (Postcondition: .\MaybeAdd.java
        return a-b;
        ^
.\MaybeAdd.java:5: warning: Associated declaration: .\MaybeAdd.java:7:
    //@ ensures  0 < \result;
        ^
2 warnings
```

As you can see in the output above, the extended static checker has correctly detected that our implementation did not satisfy the specification. Let's fix that. To do that, replace the − operation in the `MaybeAdd` class with +. Your listing should look like the following:

```
public class MaybeAdd {

  //@ requires 0 < a && a < 1000;
  //@ requires 0 < b && b < 1000;
  //@ ensures  0 < \result;
  public static int add(int a, int b){
      return a+b;
  }


  public static void main(String args[]){

      System.out.println(MaybeAdd.add(1,2));

  }

}
```

Let's see if this works now:

```
~ » spm check
```

The output from the check will look like the following:

```
[spm] INFO  - [command-check] Running all checks for project...
[spm] INFO  - [command-check] Running check: OpenJML All File ESC
[spm] INFO  - Configuring solver for Z3...
[spm] INFO  - Running OpenJML in ESC Mode...
```

Since OpenJML didn't emit any errors, it means that the code we wrote satisfies the specifications.

CHAPTER 4

Next Steps

This is just a sample of the many things you can do with Spekl. As a user of Spekl most of your work will consist of adding and running checks. To browse some of the available checks, head over to the recipes section, here: *What are Spekl Recipes?*

CHAPTER 5

What are Spekl Recipes?

Spekl makes it easy to drop in new verification checks into your projects. To that end we've created an easy to use reference of available checks in Spekl we call "Recipes." Browse the sections below to find out the kinds of checks you can add to your programs.

CHAPTER 6

OpenJML

OpenJML is a suite of tools for editing, parsing, type-checking, verifying (static checking), and run-time checking Java programs that are annotated with JML statements stating what the program's methods are supposed to do and the invariants the data structures should obey. JML annotations state preconditions, postconditions, invariants and the like about a method or class; OpenJML's tools will then check that the implementation and the specifications are consistent.

The Java Modeling Language (JML) is a behavioral interface specification language (BISL) that can be used to specify the behavior of Java modules. It combines the design by contract approach of Eiffel and the model-based specification approach of the Larch family of interface specification languages, with some elements of the refinement calculus.

More About this Tool:

- JML Specification Language
- OpenJML Project Homepage

6.1 Runtime Assertion Checking

Runtime Assertion Checking works by first compiling your program and then running it with the assertions added to your program. The listing below shows both checks configured. Note that the jml-java-7 specs have been configured. In the example below, we output all of the compiled classes to the `out` directory. This directory can be any directory you like, just make sure it exists before running `spm check`.

```
checks :
  - name        : openjml-rac-compile
    description : "OpenJML All File RAC Compile"
    check       : rac-compile
    paths       : [MaybeAdd.java]
    classpath   : []
    out         : out       # the compile output directory

    tool:
      name      : openjml-rac

    specs:
      - name: jml-java-7

  - name        : openjml-rac-run
    description : "OpenJML All File RAC Check"
    check       : rac-check
    main        : MaybeAdd  # your main class
```

```
    paths       : [MaybeAdd.java]
    classpath   : []
    out         : out       # the compile output directory

    tool:
      name      : openjml-rac
```

6.2 Extended Static Checking

Extended Static Checking is much more extensive in the types of errors it can catch but is also generally harder to write specifications for. Like the Runtime Assertion Checker, this check supports a `classpath` attribute that you can use to add classpath elements needed to resolve all the classes in your project.

```
checks :
  - name        : openjml-esc
    description : "OpenJML All File ESC"
    paths       : [MaybeAdd.java]

    tool:
      name      : openjml-esc
```

CHAPTER 7

FindBugs

FindBugs uses static analysis to inspect Java bytecode for occurrences of bug patterns. Static analysis means that FindBugs can find bugs by simply inspecting a program's code: executing the program is not necessary. This makes FindBugs very easy to use: in general, you should be able to use it to look for bugs in your code within a few minutes of downloading it. FindBugs works by analyzing Java bytecode (compiled class files), so you don't even need the program's source code to use it. Because its analysis is sometimes imprecise, FindBugs can report false warnings, which are warnings that do not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.

More About this Tool:

- FindBugs Project Homepage

7.1 Run FindBugs and Generate HTML Reports

```
checks :
  - name        : findbugs-html
    description : "FindBugs HTML Report"
    check       : html
    paths       : [A.class]  # your class files

    tool:
      name      : findbugs
```

7.2 Run FindBugs and Generate XML Reports

```
checks :
  - name        : findbugs-xml
    description : "FindBugs XML Report"
    check       : xml
    paths       : [A.class]  # your classfiles

    tool:
      name      : findbugs
```

CHAPTER 8

SAW

The Software Analysis Workbench (SAW) provides the ability to formally verify properties of code written in C, Java, and Cryptol. It leverages automated SAT and SMT solvers to make this process as automated as possible, and provides a scripting language, called SAW Script, to enable verification to scale up to more complex systems.

More About this Tool:

- Galois Homepage
- SAW Project Homepage

8.1 Verify that Two Implementations are Equivalent

```
checks :
  - name        : saw
    description : "SAW"
    check       : equiv-c
    paths       : [] #
    reference:
      file     : ffs_ref.c   # the reference file
      function : ffs_ref     # the reference function
    test:
      file     : ffs_test.c  # the file to check
      function : ffs_test    # the function to check

    tool:
      name      : saw
```

CHAPTER 9

Checker Framework

Are you tired of null pointer exceptions, unintended side effects, SQL injections, concurrency errors, mistaken equality tests, and other run-time errors that appear during testing or in the field?

The Checker Framework enhances Java's type system to make it more powerful and useful. This lets software developers detect and prevent errors in their Java programs. The Checker Framework includes compiler plug-ins ("checkers") that find bugs or verify their absence. It also permits you to write your own compiler plug-ins.

More About this Tool:

- Checker Framework Homepage

9.1 Nullness Checker

Note that this checker supports an optional `classpath` element, which is a list of paths to add to the classpath. Also, if you do not want your class files written to the same directory in which they reside, you may use the `out` option to specify an output directory. Note that it must exist before running `spm check`.

See the Nullness Checker Documentation

```
checks :
  - name        : checker-framework-nullness
    check       : nullness
    description : "Checker Framework Nullness Check"
    paths       : [MaybeAdd.java]

    tool:
      name      : checker-framework
```

9.2 Interning Checker

Note that this checker supports an optional `classpath` element, which is a list of paths to add to the classpath. Also, if you do not want your class files written to the same directory in which they reside, you may use the `out` option to specify an output directory. Note that it must exist before running `spm check`.

See the Interning Checker Documentation

```
checks :
  - name        : checker-framework-interning
    check       : interning
    description : "Checker Framework Interning Check"
```

```
    paths        : [MaybeAdd.java]

    tool:
      name       : checker-framework
```

CHAPTER 10

CheckLT: Taint Checking for Mere Mortals

CheckLT is a program verification tool for Java which can help you use taint tracking to find defects in your software. CheckLT provides an easy to install verification toolset, a simple, non-invasive syntax for annotating programs, and a dynamically configurable security lattice.

More About this Tool:

- CheckLT Homepage

Note that this checker supports an optional `classpath` element, which is a list of paths to add to the classpath. Also, if you do not want your class files written to the same directory in which they reside, you may use the `out` option to specify an output directory. Note that it must exist before running `spm check`.

Note that for this to work, you must define a `security.xml` in the root directory of your project as described by the CheckLT documentation.

```
checks :
  - name        : checklt
    description : "CheckLT Lattice Tainting Check"
    paths       : [MaybeAdd.java]

    tool:
      name      : checklt
```

CHAPTER 11

Tool Authoring Guide

CHAPTER 12

Specification Authoring Guide

Specification authoring is central to the design of Spekl. In this section you will learn how to author and publish specifications with the help of Spekl.

CHAPTER 13

Authoring Standalone Specifications

The first, and easiest to understand mode of authoring is the case of authoring standalone specifications. You can think of standalone specifications as a specification project, in which you provide only the contents of your specifications. Let's dive right in.

13.1 Creating the Project

For this section, let's pretend that we are authoring specifications for a fictional project called `MaybeAdd`. To start, let's create a project:

```
~ » mkdir maybe-add-specs
~ » spm init spec
```

The `spm` tool will ask you a series of questions to set up this project. There are a few important details to understand here:

- You should specify a unique name for the project. I'm going to call mine `maybe-add-specs` – but you'll have to pick something else since this command will register your project with the Spekl central repository.
- You'll need a valid GitHub account. When the wizard prompts you for a username, make sure you give your GitHub username. Also, the email and full name you specify here needs to be the email and full name you use on GitHub.
- After you create your project, you will get a email from GitHub asking you to join a team. **Accept this invite**. You'll need to do this before you can publish your specification.

After this command completes, you'll have a single file in your current directory called `package.yml`. Let's have a look at that file, now.

```
name        : maybe-add-specs          # name of the package
version     : 0.0.1       # version of the package
kind        : spec                     # one of tool or spec(s)
description : a short description

author:
  - name: John L. Singleton
    email: jsinglet@gmail.com
```

Note that the `package.yml` file supports all the usual configuration elements supported in tool authoring. You can enforce environmental conditions with the `assumes` configuration element, you can create dependencies with the `depends` element and you can add assets and installation commands with the `assets` and `install` configuration elements.

In general a specification library won't do these things but if you need to, you can see the section in the tool authoring guide on configuring these elements. See the *Tool Authoring Guide* for more details.

Next you will want to start adding specifications to your library. As an example, we'll add a specification in the `JML` language called `MaybeAdd`. To do this, create a file in the current directory called `MaybeAdd.jml`.

```
public class MaybeAdd {

    //@ requires 0 < a && a < 1000;
    //@ requires 0 < b && b < 1000;
    //@ ensures  \result > 0;
    public static int add(int a, int b);

}
```

Ok, you've just authored your first specification with Spekl! Let's publish your spec. To do that, execute the `spm publish` command:

```
~ » spm publish
```

As long as you met the requirements we mentioned earlier in this section, you should now have a freshly published project. You can continue working on this project by editing and doing a `spm publish` at any time. Every time you publish, however, make sure to increment the version number in your `package.yml` file. If you don't Spekl will not allow you to publish.

CHAPTER 14

Authoring Inline Specifications

In the last section we learned how to create stand alone specifications. While this is the authoring mode most useful for projects wishing to import existing specifications into Spekl, the more general case for using Spekl is when one wants to author specifications for a codebase that is under development. In this section we are going to learn how to do that. We'll continue with the example we've used in other sections, the `MaybeAdd` example.

To start, let's create a normal verification project:

```
~ » mkdir new-project
~ » cd new-project
~ » spm init
```

Next, create the file `MaybeAdd.java` with the following content:

```
public class MaybeAdd {

  public static int add(int a, int b){
      return a-b;
  }


  public static void main(String args[]){

      System.out.println(MaybeAdd.add(1,2));

  }

}
```

Note that this file does not contain specifications. Next, edit your `spekl.yml` file to be set up to do Runtime Assertion Checking with OpenJML:

```
checks :
  - name        : openjml-rac-compile
    description : "OpenJML All File RAC Compile"
    check       : rac-compile
    paths       : [MaybeAdd.java]
    classpath   : []
    out         : out       # the compile output directory

    tool:
      name      : openjml-rac

    specs:
      - name: jml-java-7
```

```
  - name        : openjml-rac-run
    description : "OpenJML All File RAC Check"
    check       : rac-check
    main        : MaybeAdd  # your main class
    paths       : [MaybeAdd.java]
    classpath   : []
    out         : out       # the compile output directory

    tool:
      name      : openjml-rac
```

Next, we want to install all of the tools for this project:

```
~ » spm install
```

At this point, if we run `spm check`, we will get no errors for our code. That is because there are no specifications attached. Suppose that we'd like to create a new specification library for our project that we are working on. To do that, we execute the `spm init spec` command. Since we are in a directory with a `spekl.yml` file, Spekl will detect that we want to do an *inline specification*. This will create the specification in the `.spm` directory. Here's what the command sequence looks like:

```
$ spm init spec
[spm] INFO  - [command-init]
[spm] INFO  - [new-spec] Creating new spec project...
Spec Name? [default: my spec] my-spec-1
Spec Description? [default: a short description]
Version? [default: 0.0.1]
Author Name? [default: Some User] John L. Singleton
Author Email? [default: user@email.com] jsinglet@gmail.com
Username? (not stored) [default: someuser] xxxxxxxxxx
name        : my-spec-1          # name of the package
version     : 0.0.1       # version of the package
kind        : spec                    # one of tool or spec(s)
description : a short description

author:
  - name: John L. Singleton
    email: jsinglet@gmail.com


Does this configuration look reasonable? [Y/n] y
[spm] INFO  - [new-spec] Writing configuration file to: .spm\my-spec-1-0.0.1\package.yml
[spm] INFO  - [backend-init-at] Creating SPM repository connection...
[spm] INFO  - [new-spec] Done.
```

Note that Spekl created the spec in the `.spm/my-spec-1-0.0.1` directory. This functions exactly like the standalone specifications in the previous section but it can be authored alongside the project you are currently working on.

Let's add some specifications to the library. To do that, create the file `MaybeAdd.jml` with the following content in the newly created directory under the `.spm` directory:

```
public class MaybeAdd {

    //@ requires 0 < a && a < 1000;
    //@ requires 0 < b && b < 1000;
    //@ ensures  \result > 0;
    public static int add(int a, int b);
```

```
}
```

Next, update your `spekl.yml` file to contain a reference to your new specification library.

```
checks :
  - name        : openjml-rac-compile
    description : "OpenJML All File RAC Compile"
    check       : rac-compile
    paths       : [MaybeAdd.java]
    classpath   : []
    out         : out       # the compile output directory

    tool:
      name      : openjml-rac

    specs:
      - name: jml-java-7
      - name: my-spec-1 #  <-- added my-spec here

  - name        : openjml-rac-run
    description : "OpenJML All File RAC Check"
    check       : rac-check
    main        : MaybeAdd  # your main class
    paths       : [MaybeAdd.java]
    classpath   : []
    out         : out       # the compile output directory

    tool:
      name      : openjml-rac
```

Now, let's try to run a check:

```
$ spm check


[spm] INFO  - [command-check] Running all checks for project...
[spm] INFO  - [command-check] Running check: OpenJML All File RAC Compile
[spm] INFO  - Running OpenJML RAC Compile...
.spm\my-spec-1-0.0.1\MaybeAdd.jml:5: error: The token \result is illegal or not implemented for a typ
    //@ \result == a+b;
        ^
Note: .spm\jml-java-7-1.7-2\java\util\Arrays.jml uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
1 error

[spm] INFO  - [command-check] Running check: OpenJML All File RAC Check
[spm] INFO  - Running OpenJML RAC Program...
```

Now, as you can see, Spekl correctly picks up your specification library. You can create as many specification libraries as you want and add them to your project so as to specify different portions of your codebase. This enables you to use different tools, different checks, and different languages all within the same project.

As with stand alone projects, you can publish your changes by going into your newly created specification directory under `.spm` and typing `spm publish`.

CHAPTER 15

Where Do Published Specs Go?

After you've published your specs, you may want access them directly. You can find your spec on GitHub at http://github.com/Spekl/<your spec>

CHAPTER 16

Advanced Topics in Specification Authoring

For the more advanced reader, the following sections contain some topics that pertain to some of Spekl's more advanced features for specification authoring.

16.1 Layering Specifications

Many times in specification writing you'll want to modify the specification of some existing specification – but use the other parts of the specification. Spekl allows you do do this via *Specification Extension*. In extension, you base your specifications off of a pre-existing specification. Any modifications made to the upstream specification are then automatically propagated down to your specification.

For example, let's think about specifying the Java API. Java 7 contains many of the same functions and classes as Java 4, but with some additions and changes. In turn, Java 6 could be based on Java 5 and so on. In Spekl, this is a perfect example of a specification hierarchy. More formally, you can think of a specification hierarchy as a chain of the form:

$$\overrightarrow{SH} = \{(S_\bot, H_\bot), \ldots, (S_\top, H_\top)\}$$

Where $\top$ is the top of the hierarchy and $\bot$ is the bottom. It is ordered by the relation given here:

$$(S', H') \sqsupseteq (S, H) \iff S' <: S \wedge \exists \delta \in H' \ : \ \delta \in H$$

To specify that a specification should extend another specification, you use the `spm extend` command. During the process of creation you give the name of a specification that exists.

When the creation process is over, you will now have a freshly created specification project that is based on the upstream specification. Any changes you make to your specification will be local to only your specification, but now you will have the ability to `refresh` your specification with respect to the upstream specification.

16.2 Refreshing Layered Specifications

Once you have created a downstream specification with the `spm extend` command, you can either explicitly update your specification with the changes in the upstream specification or allow it to happen automatically at check time.

To update your specification with the most recent work on the upstream specification, use the following command:

```
~ » spm refresh
```

This command will walk up the entire specification hierarchy and refresh your specification if there has been any upstream work. Note that if you do a `spm publish`, these changes will become part of the permanent history of your specification library (and will no longer need to be refreshed).

CHAPTER 17

Frequently Asked Questions

17.1 I just installed Spekl but it's not appearing on my path. Do I need to manually add it?

Spekl's installer automatically adds the path to `spm` to your path. On platforms like Windows and OSX, the easiest way to refresh your path is to restart the shell you use (CMD.EXE or Cygwin, Terminal.app, etc). This should enable you to use the `spm` tool on the command line. On Linux platforms the path is not automatically updated and must be added to your `.profile` or shell init scripts. In case the automatic path modification doesn't work (or if you are on Linux), the following list below details where Spekl is installed on various platforms:

- Windows: `C:\Program Files (x86)\spm`
- OSX: `/Applications/spm`
- Linux: `/opt/spm`

17.2 I need to update my publishing information/authentication info, how do I do it?

All authentication information is stored under the `author` configuration element. See the next question (*My project has multiple authors, how can we all author together?*) for more information on how to get this to work.

17.3 My project has multiple authors, how can we all author together?

The author attribute in the configuration supports any number of authors. For it to work, you must first add the user to your team on GitHub. To do that, go to the team administration page for your project: https://github.com/orgs/Spekl/teams

After you add the user to your team, fill in their name and email in the author configuration element, for example:

```
name        : maybe-add-specs          # name of the package
version     : 0.0.1       # version of the package
kind        : spec                     # one of tool or spec(s)
description : a short description

author:
```

```
  - name: John L. Singleton
    email: jsinglet@gmail.com
  - name: Some Other User
    email: otheruser@gmail.com
```

When you run `spm publish` you will be prompted to select the user you want to use for authentication.

17.4 How Do I Update My Specs/Tools After I Publish Them?

If you still have the local directory you created the project in you can just use your normal `spm publish` commands.

If you are on a different machine (or you'd like to give access to someone else) just go to our Spekl page on GitHub, here: http://github.com/Spekl. Find your repo and do a normal `git clone` as with any other Git repository. Note however that you should not do an explicit `git push` – rely on `spm publish` do do that work for you.

17.5 Help! Something is Broken! Spekl Isn't Working, etc

The first thing you should always do is delete the `.spm` directory and try installing with `spm install` again. If that doesn't work, please open an issue over on our GitHub Issue Tracker.

CHAPTER 18

Indices and tables

- genindex
- modindex
- search