Subplot

The Subplot project

2022-04-10 09:02

Contents

1 Introduction

Subplot is software to help capture and communicate acceptance criteria for software and systems, and how they are verified, in a way that's understood by all project stakeholders. The current document contains the acceptance criteria for Subplot itself, and its architecture.

The acceptance criteria are expressed as *scenarios*, which roughly correspond to use cases. The scenario as accompanied by explanatory text to explain things to the reader. Scenarios use a given/when/then sequence of steps, where each step is implemented by code provided by the developers of the system under test. This is very similar to the Cucumber[1] tool, but with more emphasis on producing a standalone document.

1.1 Acceptance criteria and acceptance tests

We define the various concepts relevant to Subplot as follows:

- **Acceptance criteria**: What the stakeholders require of the system for them to be happy with it and use it.
- **Stakeholder**: Someone with a keen interest in the success of a system. They might be a paying client, someone who uses the system, or someone

[1] `https://en.wikipedia.org/wiki/Cucumber_(software)`

involved in developing the system. Depending on the system and project, some stakeholders may have a bigger say than others.

- **Acceptance test**: How stakeholders verify that the system fulfills the acceptance criteria, in an automated way. Some criteria may not be possible to verify automatically.
- **Scenario**: In Subplot, the acceptance criteria are written as freeform prose, with diagrams, etc. The scenarios, which are embedded blocks of Subplot scenario language, capture the mechanisms of verifying that criteria are met - the acceptance tests - showing step by step how to determine that the software system is acceptable to the stakeholders.

1.2 A basic workflow for using Subplot

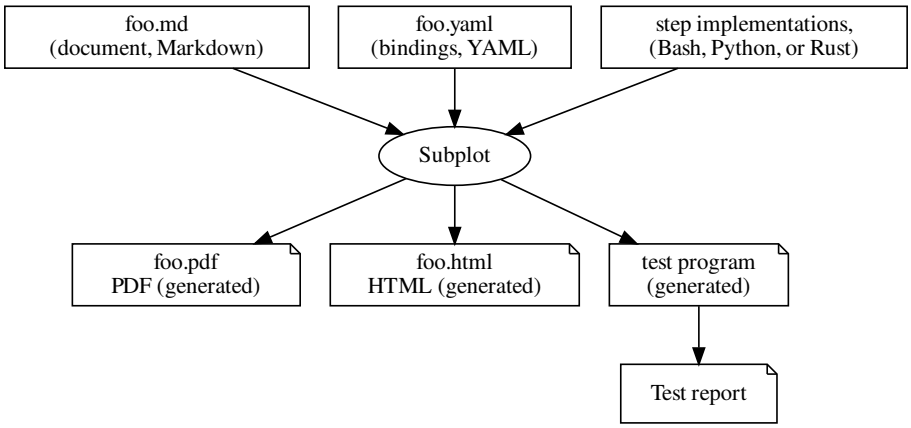
We recommend the following initial approach to using Subplot, which you can vary based on your particular needs and circumstances.

1. Start with a small acceptance document that you think expresses some useful requirements.
2. Write some acceptance criteria and have them agreed among the stakeholders.
3. Write scenarios to verify that the criteria are met, and have those scenarios agreed by the stakeholders.
4. Write bindings and test functions, so that as the code is written it can be tested against the acceptance criteria.
5. Iterate on this in short cycles to maximise discussion and stakeholder buy-in.

You definitely want to keep the subplot document source code in version control. You certainly need to have people who can write technical text that's aimed at all your stakeholders.

1.3 Subplot architecture

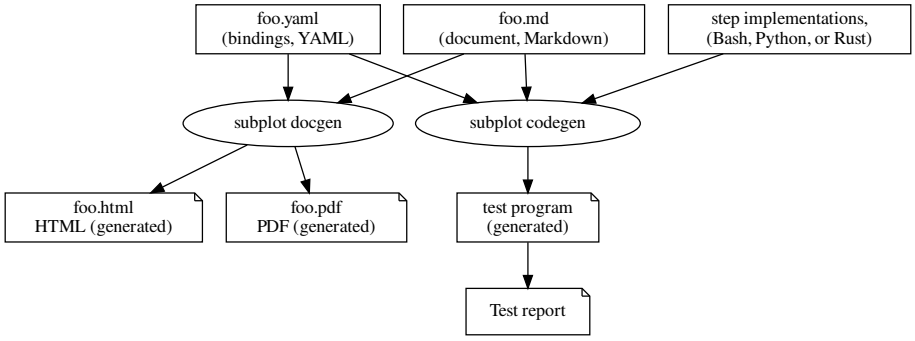
Subplot reads an input document, in Markdown, and generates a typeset output document, as PDF or HTML, for all stakeholders to understand. Subplot also generates a test program, in Python, that verifies the acceptance criteria are met, for developers and testers and auditors to verify the system under test meets its acceptance criteria. The generated program uses code written by the Subplot user to implement the verification steps. The graph below illustrates this and shows how data flows through the system.



Subplot uses the Pandoc[2] software for generating PDF and HTML output documents. In fact, any output format supported by Pandoc can be requested by the user. Depending on the output format, Pandoc may use, for example, LaTeX. Subplot interprets parts of the Markdown input file itself.

Subplot actually consists mainly of two separate programs: **subplot docgen** for generating output documents, and **subplot codegen** for generating the test program. There are a couple of additional tools (**subplot metadata** for reporting meta data about a Subplot document, and **subplot-filter** for doing the document generation as a Pandoc filter).

Thus a more detailed architecture view is shown below.



1.4 A fairy tale of acceptance testing

The king was upset. This naturally meant the whole court was in a tizzy and chattering excitedly at each other, while trying to avoid the royal wrath.

"Who will rid me of this troublesome chore?" shouted the king, and quaffed a flagon of wine. "And no killing of priests, this time!"

[2] `https://pandoc.org/`

The grand hall's doors were thrown open. The grand wizard stood in the doorway, robe, hat, and staff everything, but quite still. After the court became silent, the wizard strode confidently to stand before the king.

"What ails you, my lord?"

The king looked upon the wizard, and took a deep breath. It does not do to shout at wizards, for they control dragons, and even kings are tasty morsels to the great beasts.

"I am tired of choosing what to wear every day. Can't you do something?"

The wizard stoked his long, grey beard. He turned around, looked at the magnificent outfits worn by members of the court. He turned back, and looked at the king.

"I believe I can fix this. Just to be clear, your beef is with having to choose clothing, yes?"

"Yes", said the king, "that's what I said. When will you be done?"

The wizard raised his staff and brought it back down again, with a loud bang.

"Done" said the wizard, smugly.

The king was amazed and started smiling, until he noticed that everyone, including himself, was wearing identical burlap sacks and nothing on their feet. His voice was high, whiny, like that of a little child.

"Oh no, that's not at all what I wanted! Change it back! Change it back now!"

The morale of this story is to be clear and precise in your acceptance criteria, or you might get something other than what you really, really wanted.

1.5 Motivation for Subplot

Keeping track of requirements and acceptance criteria is necessary for all but the simplest of software projects. Having all stakeholders in a project agree to them is crucial, as is that all agree how it is verified that the software meets the acceptance criteria. Subplot provides a way for documenting the shared understanding of what the acceptance criteria are and how they can be checked automatically.

Stakeholders in a project may include:

- those who pay for the work to be done; this may be the employer of the developers for in-house projects ("*customer*")
- those who use the resulting systems, whether they pay for it or not ("*user*")
- those who install and configure the systems and keep them functional ("*sysadmin*")
- those who support the users ("*support*")
- those who test the project for acceptability ("*tester*")

- those who develop the system in the first place ("*developer*")

The above list is incomplete and simplistic, but suffices as an example.

All stakeholders need to understand the acceptance criteria, and how the system is evaluated against the criteria. In the simplest case, the customer and the developer need to both understand and agree so that the developer knows when the job is done, and the customer knows when they need to pay their bill.

However, even when the various stakeholder roles all fall upon the same person, or only on people who act as developers, the Subplot tooling can be useful. A developer would understand acceptance criteria expressed only in code, but doing so may take time and energy that are not always available. The Subplot approach aims to encourage hiding unnecessary detail and documenting things in a way that is easy to understand with little effort.

Unfortunately, this does mean that for a Subplot output document to be good and helpful, writing it will require effort and skill. No tool can replace that.

1.6 Using this document to verify Subplot works

This document ("subplot") can be used to verify Subplot itself from its source tree or an installed Subplot. The default is to test Subplot from the source tree, and the `./check` script does that. You can run this in the source tree to build Subplot and then verify it using itself:

```
$ cargo build -q
$ cargo run --bin subplot codegen -- subplot.md -o test.py
$ python3 test.py
... much output
OK, all scenarios finished successfully
$
```

To test an installed Subplot, generate the test program, and tell the test program where Subplot is installed. Again, in the Subplot source tree:

```
$ cargo build -q
$ cargo run --bin subplot codegen -- subplot.md -o test.py
$ python3 test.py  --env SUBPLOT_DIR=/usr/local/bin
... much output
OK, all scenarios finished successfully
$
```

You can do this with an installed Subplot as well:

```
$ cargo clean
$ /usr/local/bin/subplot codegen subplot.md -o test.py
$ python3 test.py --env SUBPLOT_DIR=/usr/local/bin
... much output
```

```
OK, all scenarios finished successfully
$
```

The generated test program is self-standing, and can be run from anywhere. However, to generate it you need to be in the Subplot source tree. You can move it elsewhere after generating it, you if you prefer.

2 Requirements

This chapter lists requirements for Subplot. These requirements are not meant to be automatically verifiable. For specific, automatically testable acceptance criteria, see the later chapter with acceptance tests for Subplot[3].

Each requirement here is given a unique mnemonic id for easier reference in discussions.

UnderstandableTests Acceptance tests should be possible to express in a way that's easily understood by all stakeholders, including those who are not software developers.

Done but requires the Subplot document to be written with care.

EasyToWriteDocs The markup language for writing documentation should be easy to write.

Done by using Markdown.

AidsComprehension The formatted human-readable documentation should use good layout and typography to enhance comprehension.

In progress — typesetting via Pandoc works, but may need review and improvement.

CodeSeparately The code to implement the acceptance criteria should not be embedded in the documentation source, but be in separate files. This makes it easier to edit without specialised tooling.

Done by keeping scenario step implementations in a separate file.

AnyProgammingLanguage The developers implementing the acceptance tests should be free to use a language they're familiar and comfortable with. Subplot should not require them to use a specific language.

Not done — only Python supported at the moment.

FastTestExecution Executing the acceptance tests should be fast.

Not done — the generated Python test program is simplistic and linear.

NoDeployment The acceptance test tooling should assume the system under test is already deployed and available. Deploying is too big of a problem

[3] `#acceptance`

space to bring into the scope of acceptance testing, and there are already good tools for deployment.

Done by virtue of letting those who implement the scenario steps worry about it.

MachineParseableResults The tests should produce a machine parseable result that can be archived, post-processed, and analyzed in ways that are of interest to the project using Subplot. For example, to see trends in how long tests take, how often tests fail, to find regressions, and to find tests that don't provide value.

Not done — the generated test program is simplistic.

3 Subplot input language

Subplot reads three input files, each in a different format:

- The document file, which uses the Markdown dialects understood by Pandoc.
- The bindings file, in YAML.
- The functions file, in Bash or Python.

Subplot interprets marked parts of the input document specially. It does this via the Pandoc abstract syntax tree, rather than text manipulation, and thus anything that Pandoc understands is understood by Subplot. We will not specify Pandoc's dialect of Markdown here, only the parts Subplot pays attention to.

3.1 Scenario language

The scenarios are core to Subplot. They express what the detailed acceptance criteria are and how they're verified. The scenarios are meant to be understood by both all human stakeholders and the Subplot software. As such, they are expressed in a somewhat stilted language that resembles English, but is just formal enough that it can also be understood by a computer.

A scenario is a sequence of steps. A step can be setup to prepare for an action, an action, or an examination of the effect an action had. For example, a scenario to verify that a backup system works might look like the following:

```
~~~scenario
given a backup server
when I make a backup
and I restore the backup
then the restored data is identical to the original data
~~~
```

This is not magic. The three kinds of steps are each identified by the first word in the step.

- `given` means it's a step to set up the environment for the scenario
- `when` means it's a step with the action that the scenario verifies
- `then` means it's a step to examine the results of the action

The `and` keyword is special in that it means the step is the same kind as the previous step. In the example, on line 4, it means the step is a `when` step.

Each step is implemented by a bit of code, provided by the author of the subplot document. The step is *bound* to the code via a binding file, via the text of the step: if the text is like this, then call that function. Bindings files are described in detail shortly below.

The three kinds of steps exist to make scenarios easier to understand by humans. Subplot itself does not actually care if a step is setup, action, or examination, but it's easier for humans reading the scenario, or writing the corresponding code, if each step only does the kind of work that is implied by the kind of step it's bound to.

3.1.1 Using Subplot's language effectively

Your subplot scenarios will be best understood when they use the subplot language in a consistent fashion, within and even across *different* projects. As with programming languages, it's possible to place your own style on your subplots. Indeed, there is no inherent internal implementation difference between how `given`, `when` and `then` steps are processed (other than that `given` steps often also have cleanup functions associated with them).

Nonetheless we have some recommendations about using the Subplot language, which reflect how we use it in Subplot and related projects.

When you are formulating your scenarios, it is common to try and use phraseology along the lines of *if this happens then that is the case* but this is not language which works well with subplot. Scenarios describe what will happen in the success case. As such we don't construct scenarios which say *if foo happens then the case fails*, instead we say *when I do the thing then foo does not happen*. This is a subtle but critical shift in the construction of your test cases which will mean that they map more effectively to scenarios.

Scenarios work best when they describe how some entity (human or otherwise) actually goes about successfully achieving their goal. They start out by setting the scene for the goal (`given`) they go on to describe the actions/activity undertaken in order for the goal to be achieved (`when`) and they describe how the entity knows that the goal has been achieved (`then`). By writing in this active goal-oriented fashion, your scenarios will flow better and be easier for all stakeholders to understand.

In general you should use `given` statements where you do not wish to go into the detail of what it means for the statement to have been run, you simply wish to inform the reader that some precondition is met. These statements are often

best along the lines of `given a setup which works` or `given a development enviroment` or somesuch.

The `when` statements are best used to denote **active** steps. These are the steps which your putative actors or personae use to achieve their goals. These often work best in the form `when I do the thing` or `when the user does the thing`.

The `then` statements are the crux of the scenario, they are the **validation** steps. These are the steps which tell the reader of the scenario how the actor knows that their action (the `when` steps) has had the desired outcome. This could be of the form `then some output is present` or `then it exits successfully`.

With all that in mind, a good scenario looks like

given the necessary starting conditions
when I do the required actions
then the desired outcome is achieved

Given all that, however, it's worth considering some pitfalls to avoid when writing your scenarios.

It's best to avoid overly precise or overly technical details in your scenario language (unless that's necessary to properly describe your goal etc.) So it's best to say things like `then the output file is valid JSON` rather than `then the output file contains {"foo": "bar", "baz": 7}`. Obviously if the actual values are important then again, statements such as `then the output file has a key "foo" which contains the value "bar"` or similar.

Try not to change "person" or voice in your scenarios unless there are multiple entities involved in telling your stories. For example, if you have a scenario statement of `when I run fooprogram` do not also have statements in the passive such as `when fooprogram is run`. It's reasonable to switch between `when` and `then` statements (`then the output is good`) but try not to have multiple `then` statements which switch it up, such as `then I have an output file`, `and the output file is ok`.

If you're likely to copy-paste your scenario statements around, do not use `and` as a scenario keyword, even though it's valid to do so. Instead start all your scenario statements with the correct `given`, `when`, or `then`. The typesetter will deal with formatting that nicely for you.

3.2 Document markup

Subplot uses Pandoc[4], the universal document converter, to parse the Markdown file, and thus understands the variants of Markdown that Pandoc supports. This includes traditional Markdown, CommonMark, and GitHub-flavored Markdown.

[4] `https://pandoc.org/`

Subplot extends Markdown by treating certain certain tags for fenced code blocks[5] specially. A scenario, for example, would look like this:

```
```scenario
given a standard setup
when peace happens
then everything is OK
```
```

The `scenario` tag on the code block is recognized by Subplot, which will typeset the scenario (in output documents) or generate code (for the test program) accordingly. Scenario blocks do not need to be complete scenario. Subplot will collect all the snippets into one block for the test program. Snippets under the same heading belong together; the next heading of the same or a higher level ends the scenario.

For embedding test data files in the Markdown document, Subplot understands the `file` tag:

```
~~~{#filename .file}
This data is accessible to the test program as 'filename'.
~~~
```

The `.file` attribute is necessary, as is the identifier, here `#filename`. The generated test program can access the data using the identifier (without the #). The mechanism used is generic to Pandoc, and can be used to affect the typesetting by adding more attributes. For example, Pandoc can typeset the data in the code block using syntax highlighting, if the language is specified: `.markdown`, `.yaml`, or `.python`, for example.

Subplot also understands the `dot` and `roadmap` tags, and can use the Graphviz dot program, or the roadmap[6] Rust crate, to produce graphs. These can useful for describing things visually.

When typesetting files, Subplot will automatically number the lines in the file so that documentation prose can refer to sections of embedded files without needing convoluted expressions of positions. However if you do not want that, you can annotate the file with `.noNumberLines`.

For example...

```
~~~{#numbered-lines.txt .file}
This file has numbered lines.

This is line number three.
~~~

~~~{#not-numbered-lines.txt .file .noNumberLines}
```

[5] https://pandoc.org/MANUAL.html#fenced-code-blocks
[6] https://crates.io/search?q=roadmap

```
This file does not have numbered lines.

This is still line number three, but would it be obvious?
~~~
```

. . . renders as:

File: **numbered-lines.txt**

```
1 This file has numbered lines.
2
3 This is line number three.
```

File: **not-numbered-lines.txt**

```
This file does not have numbered lines.

This is still line number three, but would it be obvious?
```

3.3 Document metadata

Pandoc supports, and Subplot makes use of, a YAML metadata block[7] in a Markdown document. This can and should be used to set the document title, authors, date (version), and can be used to control some of the typesetting. Crucially for Subplot, the bindings and functions files are named in the metadata block, rather than Subplot deriving them from the input file name.

As an example, the metadata block for the Subplot document might look as follows. The `---` before and `...` after the block are mandatory: they are how Pandoc recongizes the block.

```
1  ---
2  title: "Subplot"
3  author: The Subplot project
4  date: work in progress
5  bindings:
6  - subplot.yaml
7  impls:
8    python:
9      - subplot.py
10 ...
```

There can be more than one bindings or functions file: use a YAML list.

3.4 Bindings file

The bindings file binds scenario steps to code functions that implement the steps. The YAML file is a list of objects (also known as dicts or hashmaps or key/value

[7] `https://pandoc.org/MANUAL.html#extension-yaml_metadata_block`

pairs), specifying a step kind (given, when, then), a pattern matching the text of the step and optionally capturing interesting parts of the text. Each binding may contain a type map which tells subplot the types of the captures in the patterns so that they can be validated to some extent, and a binding will list some number of implementations, each of which is specified by the name of the language (template) it is for, and then the name of a function that implements the step, optionally with the name of a function to call to clean up a scenario which includes that step.

There are some flexibilities in bindings, futher details can be found below:

1. Patterns can be simple or full-blown Perl-compatible regular expresssions (PCRE[8]).
2. Bindings *may* have type maps. Without a type map, all captures are considered to be short strings (words).
3. Bindings *may* have as many or as few implementations as needed. A zero `impl` binding will work for `docgen` but will fail to `codegen`. This can permit document authors to prepare bindings without knowing how an engineer might implement it.

```
- given: "a standard setup"
  impl:
    python:
      function: create_standard_setup
- when: "{thing} happens"
  impl:
    python:
      function: make_thing_happen
  types:
    thing: word
- when: "I say (?P<sentence>.+) with a smile"
  regex: true
  impl:
    python:
      function: speak
- then: "everything is OK"
  impl:
    python:
      function: check_everything_is_ok
```

In the example above, there are four bindings and they all provide Python implementation functions:

- A binding for a "given a standard setup" step. The binding captures no part of the text, and causes the `create_standard_setup` function to be called.

[8] `https://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions`

- A binding for a "when" step consisting of one word followed by "happens". For example, "peace", as in "then peace happens". The word is captured as "thing", and given to the `make_thing_happen` function as an argument when it is called.
- A binding for a "when" followed by "I say", an arbitrary sentence, and then "with a smile", as in "when I say good morning to you with a smile". The function `speak` is then called with capture named "sentence" as "good morning to you".
- A binding for a "then everything is OK" step, which captures nothing, and calls the `check_everything_is_ok` function.

3.4.1 Simple patterns

The simple patterns are of the form `{name}` and match a single word consisting of printable characters. This can be varied by adding a suffix, such as `{name:text}` which matches any text. The following kinds of simple patterns are supported:

- `{name}` or `{name:word}` – a single word. As a special case, the form `{name:file}` is also supported. It is also a single word, but has the added constraint that it must match an embedded file's name.
- `{name:text}` – any text
- `{name:int}` – any whole number, including negative
- `{name:uint}` – any unsigned whole number
- `{name:number}` – any number

A pattern uses simple patterns by default, or if the `regex` field is set to false. To use regular expressions, `regex` must be set to true. Subplot complains if typical regular expression characters are used, when simple patterns are expected, unless `regex` is explicitly set to false.

3.4.2 Regular expression patterns

Regular expression patterns are used only if the binding `regex` field is set to `true`.

The regular expressions use PCRE[9] syntax as implemented by the Rust regex[10] crate. The `(?P<name>pattern)` syntax is used to capture parts of the step. The captured parts are given to the bound function as arguments, when it's called.

3.4.3 The type map

Bindings may also contain a type map. This is a dictionary called `types` and contains a key-value mapping from capture name to the type of the capture. Valid types are listed above in the simple patterns section. In addition to simple patterns, the type map can be used for regular expression bindings as well.

[9] `https://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions`
[10] `https://crates.io/crates/regex`

When using simple patterns, if the capture is given a type in the type map, and also in the pattern, then the types must match, otherwise subplot will refuse to load the binding.

Typically the type map is used by the code generators to, for example, distinguish between `"12"` and `12` (i.e. between a string and what should be a number). This permits the generated test suites to use native language types directly. The `file` type, if used, must refer to an embedded file in the document; subplot docgen will emit a warning if the file is not found, and subplot codegen will emit an error.

3.4.4 The implementation map

Bindings can contain an `impl` map which connects the binding with zero or more language templates. If a binding has no `impl` entries then it can still be used to `docgen` a PDF or HTML document from a subplot document. This permits a workflow where requirements owners / architects design the validations for a project and then engineers implement the step functions to permit the validations to work.

Shipped with subplot are a number of libraries such as `files` or `runcmd` and these libraries are polyglot in that they provide bindings for all supported templates provided by subplot.

Here is an example of a binding from one of those libraries:

```
- given: file {embedded_file}
  impl:
    rust:
      function: subplotlib::steplibrary::files::create_from_embedded
    python:
      function: files_create_from_embedded
  types:
    embedded_file: file
```

3.4.5 Embedded file name didn't match

given file **badfilename.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen badfilename.md -o foo.pdf**
then file **foo.pdf** exists
when I try to run **subplot codegen --run badfilename.md -o test.py**
then command fails

File: **badfilename.md**

```
---
title: Bad filenames in matched steps do not permit codegen
bindings: [b.yaml]
impls:
  python: [f.py]
...

# Bad filename

```scenario
given file missing.md
```
```

3.5 Functions file

Functions implementing steps are supported in Bash and Python. The language is chosen by setting the `template` field in the document YAML metadata to `bash` or `python`.

The functions files are not parsed by Subplot at all. Subplot merely copies them to the output. All parsing and validation of the file is done by the programming language being used.

The conventions for calling step functions vary by language. All languages support a "dict" abstraction of some sort. This is most importantly used to implement a "context" to store state in a controlled manner between calls to step functions. A step function can set a key to a value in the context, or retrieve the value for a key.

Typically, a "when" step does something, and records the results into the context, and a "then" step checks the results by inspecting the context. This decouples functions from each other, and avoids having them use global variables for state.

3.5.1 Bash

The step functions are called without any arguments.

The context is managed using shell functions provided by the Bash template:

- `ctx_set key value`
- `ctx_get key`

Captured values from scenario steps are passed in via another dict and accessed using another function:

- `cap_get key`

Similarly, there's a dict for the contents of embedded data files:

- `files_get filename`

The template provides assertion functions: `assert_eq`, `assert_contains`.

Example:

```
_run()
{
    if "$@" < /dev/null > stdout 2> stderr
    then
        ctx_set exit 0
    else
        ctx_set exit "$?"
    fi
    ctx_set stdout "$(cat stdout)"
    ctx_set stderr "$(cat stderr)"
}

run_echo_without_args()
{
    _run echo
}

run_echo_with_args()
{
    args="$(cap_get args)"
    _run echo "$args"
}

exit_code_is()
{
    actual_exit="$(ctx_get exit)"
    wanted_exit="$(cap_get exit_code)"
    assert_eq "$actual_exit" "$wanted_exit"
}

stdout_is_a_newline()
{
    stdout="$(ctx_get stdout)"
    assert_eq "$stdout" "$(printf '\n')"
}

stdout_is_text()
{
    stdout="$(ctx_get stdout)"
    text="$(cap_get text)"
    assert_contains "$stdout" "$text"
}
```

```
stderr_is_empty()
{
    stderr="$(ctx_get stderr)"
    assert_eq "$stderr" ""
}
```

3.5.2 Python

The context is implemented by a dict-like class.

The step functions are called with a `ctx` argument that has the current state of the context, and each capture from a step as a keyword argument. The keyword argument name is the same as the capture name in the pattern in the bindings file.

The contents of embedded files are accessed using a function:

- `get_file(filename)`

Example:

```
import json

def exit_code_is(ctx, wanted=None):
    assert_eq(ctx.get("exit"), wanted)

def json_output_matches_file(ctx, filename=None):
    actual = json.loads(ctx["stdout"])
    expected = json.load(open(filename))
    assert_dict_eq(actual, expected)

def file_ends_in_zero_newlines(ctx, filename=None):
    content = open(filename, "r").read()
    assert_ne(content[-1], "\n")
```

3.6 Comparing the scenario runners

Currently Subplot ships with three scenario runner templates. The Bash, Python, and Rust templates. The first two are fully self-contained and have a set of features dictated by the Subplot version. The latter is tied to how Cargo runs tests. Given that, this comparison is only considered correct against the version of Rust at the time of publishing a Subplot release. Newer versions of Rust may introduce additional functionality which we do not list here. Finally, we do not list features here which are considered fundamental, such as "runs all the scenarios" or "supports embedded files" since no template would be considered for release if it did not do these things. These are the differentiation points.

```
| Feature                      | Bash                                      | Python
| ---------------------------- | ----------------------------------------- | ----------
```

```
| Isolation model               | Subprocess                                | Subprocess
| Parallelism                   | None                                      | None
| Passing environment variables | CLI                                       | CLI
| Execution order               | Fixed order                               | Randomised
| Run specific scenarios        | Simple substring check                    | Simple substrin
| Diagnostic logging            | Writes to stdout/stderr per normal shell  | Supports compre
| Stop-on-failure               | Stops on first failure                    | Stops on first
| Data dir integration          | Cleans up only on full success            | Cleans up each
```

4 Acceptance criteria for Subplot

Add the acceptance criteria test scenarios for Subplot here.

4.1 Test data shared between scenarios

The 'smoke-test' scenarios below test Subplot by running it against specific input files. This section specifies the bindings and functions files, which are generated and cleaned up on the fly. They're separate from the scenarios so that the scenarios are shorter and clearer, but also so that the input files do not need to be duplicated for each scenario.

File: **simple.md**

````
---
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# Simple
This is the simplest possible test scenario

```scenario
given precondition foo
when I do bar
then bar was done
```
````

File: **b.yaml**

```
- given: precondition foo
  impl:
    python:
      function: precond_foo
    bash:
      function: precond_foo
```

```
- when: I do bar
  impl:
    python:
      function: do_bar
    bash:
      function: do_bar
- when: I do foobar
  impl:
    python:
      function: do_foobar
    bash:
      function: do_foobar
- then: bar was done
  impl:
    python:
      function: bar_was_done
    bash:
      function: bar_was_done
- then: foobar was done
  impl:
    python:
      function: foobar_was_done
    bash:
      function: foobar_was_done
- given: file {filename}
  impl:
    python:
      function: provide_file
    bash:
      function: provide_file
  types:
    filename: file
```

File: **f.py**

```
def precond_foo(ctx):
    ctx['bar_done'] = False
    ctx['foobar_done'] = False
def do_bar(ctx):
    ctx['bar_done'] = True
def bar_was_done(ctx):
    assert_eq(ctx['bar_done'], True)
def do_foobar(ctx):
    ctx['foobar_done'] = True
def foobar_was_done(ctx):
    assert_eq(ctx['foobar_done'], True)
```

4.1.1 Smoke test

The scenario below uses the input files defined above to run some tests to verify that Subplot can build a PDF and an HTML document, and execute a simple scenario successfully. The test is based on generating the test program from an input file, running the test program, and examining the output.

given file **simple.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** exists
when I run **subplot docgen simple.md -o simple.html**
then file **simple.html** exists
when I run **subplot codegen --run simple.md -o test.py**
then scenario "**Simple**" was run
and step "**given precondition foo**" was run
and step "**when I do bar**" was run
and step "**then bar was done**" was run
and command is successful

4.2 No scenarios means codegen fails

If you attempt to `subplot codegen` on a document which contains no scenarios, the tool will fail to execute with a reasonable error message.

given file **noscenarios.md**
and an installed subplot
when I try to run **subplot codegen noscenarios.md -o test.py**
then command fails
and stderr contains "**no scenarios were found**"

File: **noscenarios.md**

```
---
title: No scenarios in here
impls: { python: [] }
...

# This is a title

But there are no scenarios in this file, and thus nothing can be generated in a test suite.
```

4.3 Keywords

Subplot supports the keywords **given**, **when**, and **then**, and the aliases **and** and **but**. The aliases stand for the same (effective) keyword as the previous step

in the scenario. This chapter has scenarios to check the keywords and aliases in various combinations.

4.3.1 All the keywords

given file **allkeywords.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen allkeywords.md -o foo.pdf**
then file **foo.pdf** exists
when I run **subplot codegen --run allkeywords.md -o test.py**
then scenario "**All keywords**" was run
and step "**given precondition foo**" was run
and step "**when I do bar**" was run
and step "**then bar was done**" was run
and command is successful

File: **allkeywords.md**

```
---
title: All the keywords scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# All keywords

This uses all the keywords.

```scenario
given precondition foo
when I do bar
and I do foobar
then bar was done
but foobar was done
```
```

4.3.2 Keyword aliases in output

We support **and** and **but** in input lines, and we always render scenarios in output so they are used when they are allowed. This scenario verifies that this happens.

given file **aliases.md**
and file **b.yaml**
and file **f.py**

and an installed subplot
when I run **subplot docgen aliases.md -o aliases.html**
then command is successful
and file **aliases.html** matches regex /**given**<[^>]*> **precondition foo**/
and file **aliases.html** matches regex /**when**<[^>]*> **I do bar**/
and file **aliases.html** matches regex /**and**<[^>]*> **I do foobar**/
and file **aliases.html** matches regex /**then**<[^>]*> **bar was done**/
and file **aliases.html** matches regex /**and**<[^>]*> **foobar was done**/

File: **aliases.md**

```
---
title: Keyword aliases
bindings: [b.yaml]
functions: [f.py]
...

# Aliases

```scenario
given precondition foo
when I do bar
when I do foobar
then bar was done
then foobar was done
```
```

4.3.3 Misuse of continuation keywords

When continuation keywords (`and` and `but`) are used, they have to not be the first keyword in a scenario. Any such scenario will fail to parse because subplot will be unable to determine what kind of keyword they are meant to be continuing.

given file **continuationmisuse.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen continuationmisuse.md -o foo.pdf**
then file **foo.pdf** exists
when I try to run **subplot codegen --run continuationmisuse.md -o test.py**
then command fails

File: **continuationmisuse.md**

```
---
title: Continuation keyword misuse
bindings: [b.yaml]
impls:
```

```
  python: [f.py]
...

# Continuation keyword misuse

This scenario should fail to parse because we misuse a
continuation keyword at the start.

```scenario
and precondition foo
when I do bar
then bar was done
```
```

4.4 Title markup

It is OK to use markup in document titles, in the YAML metadata section. This scenario verifies that all markup works.

given file **title-markup.md**
and an installed subplot
when I run **subplot docgen title-markup.md -o foo.pdf**
then file **foo.pdf** exists

File: **title-markup.md**

```
---
title: This _uses_ ~~all~~ **most** inline `markup`
subtitle: H~2~O is not 2^10^
impls: { python: [] }
...

# Introduction
```

4.5 Empty lines in scenarios

This scenario verifies that empty lines in scenarios are ignored.

given file **emptylines.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen emptylines.md -o emptylines.pdf**
then file **emptylines.pdf** exists
when I run **subplot docgen emptylines.md -o emptylines.html**
then file **emptylines.html** exists
when I run **subplot codegen --run emptylines.md -o test.py**
then scenario "**Simple**" was run

and step "**given precondition foo**" was run
and step "**when I do bar**" was run
and step "**then bar was done**" was run
and command is successful

File: **emptylines.md**

```
---
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# Simple
This is the simplest possible test scenario

```scenario
given precondition foo

when I do bar

then bar was done

```
```

4.6 Automatic cleanup in scenarios

A binding can define a cleanup function, which gets called at the end of the scenario in reverse order for the successful steps. If a step fails, all the cleanups for the successful steps are still called. We test this for every language template we support.

File: **cleanup.yaml**

```
- given: foo
  impl:
    python:
      function: foo
      cleanup: foo_cleanup
    bash:
      function: foo
      cleanup: foo_cleanup
- given: bar
  impl:
    python:
      function: bar
      cleanup: bar_cleanup
```

```
    bash:
      function: bar
      cleanup: bar_cleanup
- given: failure
  impl:
    python:
      function: failure
      cleanup: failure_cleanup
    bash:
      function: failure
      cleanup: failure_cleanup
```

File: **cleanup.py**

```
def foo(ctx):
   pass
def foo_cleanup(ctx):
   pass
def bar(ctx):
   pass
def bar_cleanup(ctx):
   pass
def failure(ctx):
   assert 0
def failure_cleanup(ctx):
   pass
```

File: **cleanup.sh**

```
foo() {
    true
}
foo_cleanup() {
    true
}
bar() {
    true
}
bar_cleanup() {
    true
}
failure() {
   return 1
}
failure_cleanup() {
   true
}
```

4.6.1 Cleanup functions gets called on success (Python)

given file **cleanup-success-python.md**
and file **cleanup.yaml**
and file **cleanup.py**
and an installed subplot
when I run **subplot codegen --run cleanup-success-python.md -o test.py**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and command is successful

File: **cleanup-success-python.md**

```
---
title: Cleanup
bindings: [cleanup.yaml]
impls:
  python: [cleanup.py]
...

# Cleanup

~~~scenario
given foo
given bar
~~~
```

4.6.2 Cleanup functions get called on failure (Python)

given file **cleanup-fail-python.md**
and file **cleanup.yaml**
and file **cleanup.py**
and an installed subplot
when I try to run **subplot codegen --run cleanup-fail-python.md -o test.py**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and cleanup for "**given failure**" was not run
and command fails

File: **cleanup-fail-python.md**

```
---
title: Cleanup
bindings: [cleanup.yaml]
impls:
  python: [cleanup.py]
```

```
...

# Cleanup

~~~scenario
given foo
given bar
given failure
~~~
```

4.6.3 Cleanup functions gets called on success (Bash)

given file **cleanup-success-bash.md**
and file **cleanup.yaml**
and file **cleanup.sh**
and an installed subplot
when I run **subplot codegen --run cleanup-success-bash.md -o test.sh**
then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and command is successful

File: **cleanup-success-bash.md**

```
---
title: Cleanup
bindings: [cleanup.yaml]
impls:
  bash: [cleanup.sh]
...

# Cleanup

~~~scenario
given foo
given bar
~~~
```

4.6.4 Cleanup functions get called on failure (Bash)

If a step fails, all the cleanups for the preceding steps are still called, in reverse order.

given file **cleanup-fail-bash.md**
and file **cleanup.yaml**
and file **cleanup.sh**
and an installed subplot
when I try to run **subplot codegen --run cleanup-fail-bash.md -o test.sh**

then scenario "**Cleanup**" was run
and step "**given foo**" was run, and then step "**given bar**"
and cleanup for "**given bar**" was run, and then for "**given foo**"
and cleanup for "**given failure**" was not run
and command fails

File: **cleanup-fail-bash.md**

```
---
title: Cleanup
bindings: [cleanup.yaml]
impls:
  bash: [cleanup.sh]
...

# Cleanup

~~~scenario
given foo
given bar
given failure
~~~
```

4.7 Temporary files in scenarios in Python

The Python template for generating test programs supports the `--save-on-failure` option. If the test program fails, it produces a dump of the data directories of all the scenarios it has run. Any temporary files created by the scenario using the usual mechanisms need to be in that dump. For this to happen, the test runner must set the `TMPDIR` environment variable to point at the data directory. This scenario verifies that it happens.

given file **tmpdir.md**
and file **tmpdir.yaml**
and file **tmpdir.py**
and an installed subplot
when I run **subplot codegen --run tmpdir.md -o test.py**
then command is successful
and scenario "**TMPDIR**" was run
and step "**then TMPDIR is set**" was run

File: **tmpdir.md**

```
---
title: TMPDIR
bindings: [tmpdir.yaml]
impls:
  python: [tmpdir.py]
```

```
...

# TMPDIR

~~~scenario
then TMPDIR is set
~~~
```

File: **tmpdir.yaml**

```
- then: TMPDIR is set
  impl:
    python:
      function: tmpdir_is_set
```

File: **tmpdir.py**

```
import os
def tmpdir_is_set(ctx):
    assert_eq(os.environ.get("TMPDIR"), os.getcwd())
```

4.8 Capturing parts of steps for functions

A scenario step binding can capture parts of a scenario step, to be passed to the function implementing the step as an argument. Captures can be done using regular expressions or "simple patterns".

4.8.1 Capture using simple patterns

given file **simplepattern.md**
and file **simplepattern.yaml**
and file **capture.py**
and an installed subplot
when I run **subplot codegen --run simplepattern.md -o test.py**
then scenario "**Simple pattern**" was run
and step "**given I am Tomjon**" was run
and stdout contains "**function got argument name as Tomjon**"
and command is successful

File: **simplepattern.md**

```
---
title: Simple pattern capture
bindings: [simplepattern.yaml]
impls:
  python: [capture.py]
...

# Simple pattern
```

```
~~~scenario
given I am Tomjon
~~~
```

File: **simplepattern.yaml**

```
- given: I am {name}
  impl:
    python:
      function: func
```

File: **capture.py**

```
def func(ctx, name=None):
    print('function got argument name as', name)
```

4.8.2 Simple patterns with regex metacharacters: forbidden case

Help users to avoid accidental regular expression versus simple pattern confusion. The rule is that a simple pattern mustn't contain regular expression meta characters unless the rule is explicitly marked as not being a regular expression pattern.

given file **confusedpattern.md**
and file **confusedpattern.yaml**
and file **capture.py**
and an installed subplot
when I try to run **subplot codegen --run confusedpattern.md -o test.py**
then command fails
and stderr contains "**simple pattern contains regex**"

File: **confusedpattern.md**

```
---
title: Simple pattern capture
bindings: [confusedpattern.yaml]
impls:
  python: [capture.py]
...

# Simple pattern

~~~scenario
given I* am Tomjon
~~~
```

File: **confusedpattern.yaml**

```
- given: I* am {name}
```

```
  impl:
    python:
      function: func
```

4.8.3 Simple patterns with regex metacharacters: allowed case

given file **confusedbutok.md**
and file **confusedbutok.yaml**
and file **capture.py**
and an installed subplot
when I run **subplot codegen --run confusedbutok.md -o test.py**
then command is successful

File: **confusedbutok.md**

```
---
title: Simple pattern capture
bindings: [confusedbutok.yaml]
impls:
  python: [capture.py]
...

# Simple pattern

~~~scenario
given I* am Tomjon
~~~
```

File: **confusedbutok.yaml**

```
- given: I* am {name}
  impl:
    python:
      function: func
  regex: false
```

4.8.4 Capture using regular expressions

given file **regex.md**
and file **regex.yaml**
and file **capture.py**
and an installed subplot
when I run **subplot codegen --run regex.md -o test.py**
then scenario "**Regex**" was run
and step "**given I am Tomjon**" was run
and stdout contains "**function got argument name as Tomjon**"
and command is successful

File: **regex.md**

```
---
title: Regex capture
bindings: [regex.yaml]
impls:
  python: [capture.py]
...

# Regex

~~~scenario
given I am Tomjon
~~~
```

File: **regex.yaml**

```
- given: I am (?P<name>\S+)
  impl:
    python:
      function: func
  regex: true
```

4.9 Recall values for use in later steps

It's sometimes useful to use a value remembered in a previous step. For example, if one step creates a resource with a random number as its name, a later step should be able to use it. This happens in enough projects that Subplot's Python template has support for it.

The Python template has a `Context` class, with methods `remember_value`, `recall_value`, and `expand_values`. These values are distinct from the other values that can be stored in a context. Only explicitly remembered values may be recalled or expanded so that expansions don't accidentally refer to values meant for another purpose.

given file **values.md**
and file **values.yaml**
and file **values.py**
and an installed subplot
when I run **subplot codegen values.md -o test.py**
and I run **python3 test.py**
then command is successful

File: **values.md**

```
---
title: Values
bindings: [values.yaml]
impls:
  python: [values.py]
```

```
...


# Values

~~~scenario
when I remember foo as bar
then expanded "${foo}" is bar
~~~
```

File: **values.yaml**

```
- when: I remember {name} as {value}
  impl:
    python:
      function: remember

- then: expanded "{actual}" is {expected}
  impl:
    python:
      function: check
```

File: **values.py**

```
def remember(ctx, name=None, value=None):
    ctx.remember_value(name, value)

def check(ctx, expected=None, actual=None):
    assert_eq(ctx.expand_values(actual), expected)
```

4.10 Set environment variables in generated test programs

The generated test programs run each scenario with a fixed, almost empty set of environment variables. This is so that tests are more repeatable and less dependent on any values accidentally set by the developers.

However, sometimes it's helpful for the user to be able to set environment variables for the scenarios. For example, if the scenarios test locally built binaries that may be installed anywhere, the installation directory should be added to the PATH variable so that scenarios can invoke the scripts easily.

The scenario in this section verifies that the Python test program generated by `subplot codegen` accepts the option `--env NAME=VALUE`.

There is currently no equivalent functionality for the generated Bash test program. Patches for that are welcome.

given file **env.md**
and file **env.yaml**
and file **env.py**

and an installed subplot
when I run **subplot codegen env.md -o test.py**
and I try to run **python3 test.py**
then command fails
when I try to run **python3 test.py --env FOO=foo**
then command fails
when I try to run **python3 test.py --env FOO=bar**
then command is successful

File: **env.md**

```
---
title: Environment variables
bindings: [env.yaml]
impls:
  python: [env.py]
...

# Test
~~~scenario
then environment variable FOO is set to "bar"
~~~
```

File: **env.yaml**

```
- then: environment variable {name} is set to "{value:text}"
  impl:
    python:
      function: is_set_to
```

File: **env.py**

```
import os, sys
def is_set_to(ctx, name=None, value=None):
  sys.stderr.write(f"{name}={os.environ.get(name)!r}\n")
  assert os.environ.get(name) == value
```

4.11 Avoid changing typesetting output file needlessly

4.11.1 Avoid typesetting if output is newer than source files

This scenario make sure that if docgen generates the bitwise identical output to the existing output file, it doesn't actually write it to the output file, including its timestamp. This avoids triggering programs that monitor the output file for changes.

given file **simple.md**
and file **b.yaml**
and file **f.py**
and an installed subplot

when I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I wait until **1** second has passed
and I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** has same metadata as before
and only files **simple.md, b.yaml, f.py, simple.pdf** exist

4.11.2 Do typeset if output is older than markdown

given file **simple.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I wait until **1** second has passed
and I touch file **simple.md**
and I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** has changed from before

4.11.3 Do typeset if output is older than functions

given file **simple.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I wait until **1** second has passed
and I touch file **f.py**
and I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** has changed from before

4.11.4 Do typeset if output is older than bindings

given file **simple.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** exists
when I remember metadata for file **simple.pdf**
and I wait until **1** second has passed
and I touch file **b.yaml**

and I run **subplot docgen simple.md -o simple.pdf**
then file **simple.pdf** has changed from before

4.12 Document structure

Subplot uses chapters and sections to keep together scenario snippets that form a complete scenario. The lowest level heading before a snippet starts a scenario and is the name of the scenario. If there are subheadings, they divide the description of the scenario into parts, but don't start a new scenario. The next heading at the same or a higher level starts a new scenario.

4.12.1 Lowest level heading is name of scenario

given file **scenarioislowest.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot codegen --run scenarioislowest.md -o test.py**
then scenario "**heading 1.1.1**" was run
and command is successful

File: **scenarioislowest.md**

```
1
2  ---
3  title: Test scenario
4  bindings: [b.yaml]
5  impls:
6    python: [f.py]
7  ...
8
9  # heading 1
10 ## heading 1.1
11 ### heading 1.1.1
12
13 ```scenario
14 given precondition foo
15 ```
```

4.12.2 Subheadings don't start new scenario

given file **subisnotnewscenario.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot codegen --run subisnotnewscenario.md -o test.py**

then scenario "**heading 1.1a**" was run
and command is successful

File: **subisnotnewscenario.md**

```

---
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# heading 1
## heading 1.1a

```scenario
given precondition foo
```

heading 1.1.1
heading 1.1.2
```

4.12.3 Next heading at same level starts new scenario

given file **samelevelisnewscenario.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot codegen --run samelevelisnewscenario.md -o test.py**
then scenario "**heading 1.1.1**" was run
and scenario "**heading 1.1.2**" was run
and command is successful

File: **samelevelisnewscenario.md**

```

---
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# heading 1
## heading 1.1
### heading 1.1.1

```

```scenario
given precondition foo
```
heading 1.1.2

```scenario
given precondition foo
```

4.12.4 Next heading at higher level starts new scenario

given file **higherisnewscenario.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot codegen --run higherisnewscenario.md -o test.py**
then scenario "**heading 1.1.1**" was run
and scenario "**heading 1.2**" was run
and command is successful

File: **higherisnewscenario.md**

```

---
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# heading 1
## heading 1.1
### heading 1.1.1

```scenario
given precondition foo
```
heading 1.2

```scenario
given precondition foo
```
```

4.12.5 Document titles

The document and code generators require a document title, because it's a common user error to not have one, and Subplot should help make good documents.

The Pandoc filter, however, mustn't require a document title, because it's used for things like formatting websites using ikiwiki, and ikiwiki has a different way of specifying page titles.

4.12.5.1 Document generator gives an error if input document lacks title

given file **notitle.md**
and an installed subplot
when I try to run **subplot docgen notitle.md -o foo.md**
then command fails

File: **notitle.md**

```
---
bindings: [b.yaml]
functions: [f.py]
...


# Introduction

This is a very simple Markdown file without a YAML metadata block,
and thus also no document title.

```scenario
given precondition foo
when I do bar
then bar was done
```

4.12.5.2 Code generator gives an error if input document lacks title

given file **notitle.md**
and an installed subplot
when I try to run **subplot codegen --run notitle.md -o test.py**
then command fails

4.12.5.3 Subplot accepts title and headings with inline markup

Markdown allows using any inline markup in document titles and chapter and section headings. Verify that Subplot accepts them.

given file **fancytitle.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I try to run **subplot docgen fancytitle.md -o foo.md**
then command is successful
when I try to run **subplot codegen fancytitle.md -o foo.md**
then command is successful
```

File: **fancytitle.md**

```
---
title: Plain *emph* **strong** ~~strikeout~~ superscript^10^ subscript~10~
bindings: [b.yaml]
impls:
  python: [f.py]
...


# `code` [smallcaps]{.smallcaps} $$2^10$$

## "double quoted"
## 'single quoted'
## <b>raw inline</b>
## <span>span</span>
## ![alt](image.jpg)
## footnote[^1]

[^1]: footnote

This is a very simple Markdown file that uses every kind of inline
markup in the title and chapter heading.

To satisfy codegen, we *MUST* have a scenario here
~~~~scenario
when I do bar
then bar was done
~~~~
```

4.13 Running only chosen scenarios

To make the edit-test loop more convenient for the test programs generated by Subplot, we allow the user to specify patterns for scenarios to run. Default is to run all scenarios.

4.13.1 Running only chosen scenarios with Python

This verifies that the generated Python test program can run only chosen scenarios.

given file **twoscenarios-python.md**
and file **b.yaml**
and file **f.py**
and an installed subplot
when I run **subplot codegen twoscenarios-python.md -o test.py**
and I run **python3 test.py on**

then scenario "**One**" was run
and scenario "**Two**" was not run
and command is successful

File: **twoscenarios-python.md**

```
---
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
...

# One

```scenario
given precondition foo
when I do bar
then bar was done
```

Two

```scenario
given precondition foo
when I do bar
then bar was done
```
```

4.13.2 Running only chosen scenarios with Bash

This verifies that the generated Bash test program can run only chosen scenarios.

given file **twoscenarios-bash.md**
and file **b.yaml**
and file **f.sh**
and an installed subplot
when I run **subplot codegen twoscenarios-bash.md -o test.sh**
and I run **bash test.sh on**
then scenario "**One**" was run
and scenario "**Two**" was not run
and command is successful

File: **twoscenarios-bash.md**

```
---
title: Test scenario
bindings: [b.yaml]
```

```
impls:
  bash: [f.sh]
...

# One

```scenario
given precondition foo
when I do bar
then bar was done
```

Two

```scenario
given precondition foo
when I do bar
then bar was done
```
```

File: **f.sh**

```
precond_foo() {
    ctx_set bar_done 0
    ctx_set foobar_done 0
}

do_bar() {
    ctx_set bar_done 1
}

do_foobar() {
    ctx_set foobar_done 1
}

bar_was_done() {
    actual="$(ctx_get bar_done)"
    assert_eq "$actual" 1
}

foobar_was_done() {
    actual="$(ctx_get foobar_done)"
    assert_eq "$actual" 1
}
```

4.14 Document metadata

Some document metadata should end up in the typeset document, especially the title, authors. The document date is more complicated, to cater to different use cases:

- a work-in-progress document needs a new date for each revision
  - maintaining the `date` metadata field manually is quite tedious, so Subplot provides it automatically using the document source file modification time
  - some people would prefer a `git describe` or similar method for indicating the document revision, so Subplot allows the date to be specified via the command line
- a finished, reviewed, officially stamped document needs a fixed date
  - Subplot allows this to be written as the `date` metadata field

The rules for what Subplot uses as the date or document revision information are, then:

- if there is `date` metadata field, that is used
- otherwise, if the user gives the `--date` command line option, that is used
- otherwise, the markdown file's modification time is used

4.14.1 Date given in metadata

This scenario tests that the `date` field in metadata is used if specified.

given file **metadate.md**
and an installed subplot
when I run **subplot docgen metadate.md -o metadate.html**
then file **metadate.html** exists
and file **metadate.html** contains "**<title>The Fabulous Title</title>**"
and file **metadate.html** contains "**Alfred Pennyworth**"
and file **metadate.html** contains "**Geoffrey Butler**"
and file **metadate.html** contains "**WIP**"

File: **metadate.md**

```
---
title: The Fabulous Title
author:
- Alfred Pennyworth
- Geoffrey Butler
date: WIP
...
# Introduction
This is a test document. That's all.
```

4.14.2 Date given on command line

This scenario tests that the `--date` command line option is used.

given file **dateless.md**
and an installed subplot
when I run **subplot docgen dateless.md -o dateoption.html --date=FANCYDATE**
then file **dateoption.html** exists
and file **dateoption.html** contains "**<title>The Fabulous Title</title>**"
and file **dateoption.html** contains "**Alfred Pennyworth**"
and file **dateoption.html** contains "**Geoffrey Butler**"
and file **dateoption.html** contains "**FANCYDATE**"

File: **dateless.md**

```
---
title: The Fabulous Title
author:
- Alfred Pennyworth
- Geoffrey Butler
...
# Introduction
This is a test document. It has no date metadata.
```

4.14.3 No date anywhere

This scenario tests the case of no metadata `date` and no command line option, either. The date in the typeset document shall come from the modification time of the input file, and shall have the date in ISO 8601 format, with time to the minute.

given file **dateless.md**
and file **dateless.md** has modification time **2020-02-26 07:53:17**
and an installed subplot
when I run **subplot docgen dateless.md -o mtime.html**
then file **mtime.html** exists
and file **mtime.html** contains "**<title>The Fabulous Title</title>**"
and file **mtime.html** contains "**Alfred Pennyworth**"
and file **mtime.html** contains "**Geoffrey Butler**"
and file **mtime.html** contains "**2020-02-26 07:53**"

4.14.4 Missing bindings file

If a bindings file is missing, the error message should name the missing file.

given file **missing-binding.md**
and an installed subplot
when I try to run **subplot docgen missing-binding.md -o foo.html**

then command fails
and stderr contains "**: missing-binding.yaml:**"

File: **missing-binding.md**

```
---
title: Missing binding
bindings: [missing-binding.yaml]
...
```

4.14.5 Missing functions file

If a functions file is missing, the error message should name the missing file.

given file **missing-functions.md**
and file **b.yaml**
and an installed subplot
when I try to run **subplot codegen --run missing-functions.md -o foo.py**
then command fails
and stderr contains "**: missing-functions.py:**"

File: **missing-functions.md**

```
---
title: Missing functions
bindings: [b.yaml]
impls:
  python: [missing-functions.py]
...
```

4.14.6 Extracting metadata from a document

The **subplot metadata** program extracts metadata from a document. It is useful to see the scenarios, for example. For example, given a document like this:

subplot metadata would extract this information from the **simple.md** example:

```
title: Test scenario
bindings: [b.yaml]
impls:
  python: [f.py]
scenario Simple
```

This scenario check subplot metadata works. Note that it requires the bindings or functions files.

given file **images.md**
and file **b.yaml**
and file **other.yaml**
and file **f.py**
and file **other.py**

and file **foo.bib**
and file **bar.bib**
and file **expected.json**
and an installed subplot
when I run **subplot metadata images.md**
then stdout contains "**source: images.md**"
and stdout contains "**source: b.yaml**"
and stdout contains "**source: other.yaml**"
and stdout contains "**source: f.py**"
and stdout contains "**source: other.py**"
and stdout contains "**source: foo.bib**"
and stdout contains "**source: bar.bib**"
and stdout contains "**source: image.gif**"
and stdout contains "**bindings: b.yaml**"
and stdout contains "**bindings: other.yaml**"
and stdout contains "**functions[python]: f.py**"
when I run **subplot metadata images.md -o json**
then JSON output matches **expected.json**

File: **images.md**

```
---
title: Document refers to external images
bindings:
- b.yaml
- other.yaml
impls:
  python:
    - f.py
    - other.py
bibliography: [foo.bib, bar.bib]
...

![alt text](image.gif)
```

File: **other.yaml**

```
[]
```

File: **other.py**

```

```

File: **foo.bib**

```
@book{foo2020,
 author    = "James Random",
 title     = "The Foo book",
 publisher = "The Internet",
 year      =  2020,
```

```
 address   = "World Wide Web",
}
```

File: **bar.bib**

```
@book{foo2020,
 author    = "James Random",
 title     = "The Bar book",
 publisher = "The Internet",
 year      =  2020,
 address   = "World Wide Web",
}
```

File: **expected.json**

```
{
    "title": "Document refers to external images",
    "sources": [
      "b.yaml",
      "bar.bib",
      "f.py",
      "foo.bib",
      "image.gif",
      "images.md",
      "other.py",
      "other.yaml"
    ],
    "binding_files": [
      "b.yaml",
      "other.yaml"
    ],
    "impls": {
    "python": [
        "f.py",
        "other.py"
      ]
  },
    "bibliographies": [
      "bar.bib",
      "foo.bib"
    ],
    "files": [],
    "scenarios": []
}
```

4.15 Embedded files

Subplot allows data files to be embedded in the input document. This is handy for small test files and the like.

Handling of a newline character on the last line is tricky. Pandoc doesn't include a newline on the last line. Sometimes one is needed—but sometimes it's not wanted. A newline can be added by having an empty line at the end, but that is subtle and easy to miss. Subplot helps the situation by allowing a `add-newline=` class to be added to the code blocks, with one of three allowed cases:

- no `add-newline` class—default handling: same as `add-newline=auto`
- `add-newline=auto`—add a newline, if one isn't there
- `add-newline=no`—never add a newline, but keep one if it's there
- `add-newline=yes`—always add a newline, even if one is already there

The scenarios below test the various cases.

4.15.1 Extract embedded file

This scenario checks that an embedded file can be extracted, and used in a subplot.

given file **embedded.md**
and an installed subplot
when I run **subplot docgen embedded.md -o foo.html**
then file **foo.html** exists
and file **foo.html** matches regex **/embedded\.txt/**

File: **embedded.md**

```
---
title: One embedded file
...

~~~{#embedded.txt .file}
This is the embedded file.
~~~
```

4.15.2 Extract embedded file, by default add missing newline

This scenario checks the default handling: add a newline if one is missing.

given file **default-without-newline.txt**
then **default-without-newline.txt** ends in one newline

File: **default-without-newline.txt**

```
This file does not end in a newline.
```

4.15.3 Extract embedded file, by default do not add a second newline

This scenario checks the default handling: if content already ends in a newline, do not add another newline.

given file **default-has-newline.txt**
then **default-has-newline.txt** ends in one newline

File: **default-has-newline.txt**

```
This file ends in a newline.
```

4.15.4 Extract embedded file, automatically add missing newline

Explicitly request automatic newlines, when the file does not end in one.

given file **auto-without-newline.txt**
then **auto-without-newline.txt** ends in one newline

File: **auto-without-newline.txt**

```
This file does not end in a newline.
```

4.15.5 Extract embedded file, do not automatically add second newline

Explicitly request automatic newlines, when the file already ends in one.

given file **auto-has-newline.txt**
then **auto-has-newline.txt** ends in one newline

File: **auto-has-newline.txt**

```
This file ends in a newline.
```

4.15.6 Extract embedded file, explicitly add missing newline

Explicitly request automatic newlines, when the file doesn't end with one.

given file **add-without-newline.txt**
then **add-without-newline.txt** ends in one newline

File: **add-without-newline.txt**

```
This file does not end in a newline.
```

4.15.7 Extract embedded file, explicitly add second newline

Explicitly request automatic newlines, when the file already ends with one.

given file **add-has-newline.txt**
then **add-has-newline.txt** ends in two newlines

File: **add-has-newline.txt**

```
This file ends in a newline.
```

4.15.8 Extract embedded file, do not add missing newline

Explicitly ask for no newline to be added.

given file **no-adding-without-newline.txt**
then **no-adding-without-newline.txt** does not end in a newline

File: **no-adding-without-newline.txt**

```
This file does not end in a newline.
```

4.15.9 Fail if the same filename is used twice

given file **onefiletwice.md**
and an installed subplot
when I try to run **subplot docgen onefiletwice.md -o onefiletwice.html**
then command fails
and file **onefiletwice.html** does not exist

File: **onefiletwice.md**

````
---
title: Two embedded files with the same name
...

```{#filename .file}
This is the embedded file.
```

```{#filename .file}
This is another embedded file, and has the same name.
```
````

4.15.10 Fail if two filenames only differ in case

given file **casediff.md**
and an installed subplot
when I try to run **subplot docgen casediff.md -o casediff.html**
then command fails
and file **casediff.html** does not exist

File: **casediff.md**

````
---
title: Two embedded files with names differing only in case
...

```{#filename .file}
````
````

```
This is the embedded file.
```

```{#FILENAME .file}
This is another embedded file, and has the same name in uppercase.
```
```

### 4.15.11 Fail if embedded file isn't used

This scenario checks that we get warnings, when using a subplot with embedded files that aren't used.

*given* file **unusedfile.md**
*and* an installed subplot
*when* I try to run **subplot docgen unusedfile.md -o unusedfile.html**
*then* command is successful
*and* file **unusedfile.html** exists
*and* stderr contains "**thisisnotused.txt**"

File: **unusedfile.md**

```
---
title: Embedded file is not used by a scenario
...

```{#thisisnotused.txt .file}
This is the embedded file.
```
```

## 4.16 Steps must match bindings

Subplot permits the binding author to define arbitrarily complex regular expressions for binding matches. In order to ensure that associating steps to bindings is both reliable and tractable, a step must match *exactly one* binding.

File: **badbindings.yaml**

```
- given: a binding
  impl:
    python:
      function: a_binding
- given: a (?:broken)? binding
  impl:
    python:
      function: a_broken_binding
  regex: true
- given: a capitalised Binding
  impl:
    python:
```
```

```
      function: os.getcwd
  case_sensitive: true
```

4.16.1 Steps which do not match bindings do not work

File: **nobinding.md**

```
---
title: No bindings available
bindings:
- badbindings.yaml
...
# Broken scenario because step has no binding

```scenario
given a missing binding
then nothing works
```
```

given file **nobinding.md**
and file **badbindings.yaml**
and an installed subplot
when I try to run **subplot codegen --run nobinding.md -o test.py**
then command fails

4.16.2 Steps which do not case-sensitively match sensitive bindings do not work

File: **casemismatch.md**

```
---
title: Case sensitivity mismatch
impls: { python: [] }
bindings:
- badbindings.yaml
...
# Broken scenario because step has a case mismatch with sensitive binding

```scenario
given a capitalised binding
```
```

given file **casemismatch.md**
and file **badbindings.yaml**
and an installed subplot
when I try to run **subplot codegen --run casemismatch.md -o test.py**
then command fails

4.16.3 Steps which match more than one binding do not work

File: **twobindings.md**

```
---
title: Two bindings match
bindings:
- twobindings.yaml
impls:
  python: [a_function.py]
...
# Broken scenario because step has two possible bindings

```scenario
given a binding
```
```

File: **twobindings.yaml**

```
- given: a {xyzzy}
  impl:
    python:
      function: a_function
- given: a {plugh}
  impl:
    python:
      function: a_function
```

File: **a_function.py**

```
def a_function(ctx):
    assert 0
```

given file **twobindings.md**
and file **twobindings.yaml**
and file **a_function.py**
and an installed subplot
when I try to run **subplot codegen --run twobindings.md -o test.py**
then command fails
and stderr contains "**xyzzy**"
and stderr contains "**plugh**"

4.16.4 List embedded files

The `subplot metadata` command lists embedded files in its output.

given file **two-embedded.md**
and an installed subplot
when I run **subplot metadata two-embedded.md**

then stdout contains "**foo.txt**"
and stdout contains "**bar.yaml**"

File: **two-embedded.md**

```
---
title: Two embedded files
...

~~~{#foo.txt .file}
~~~

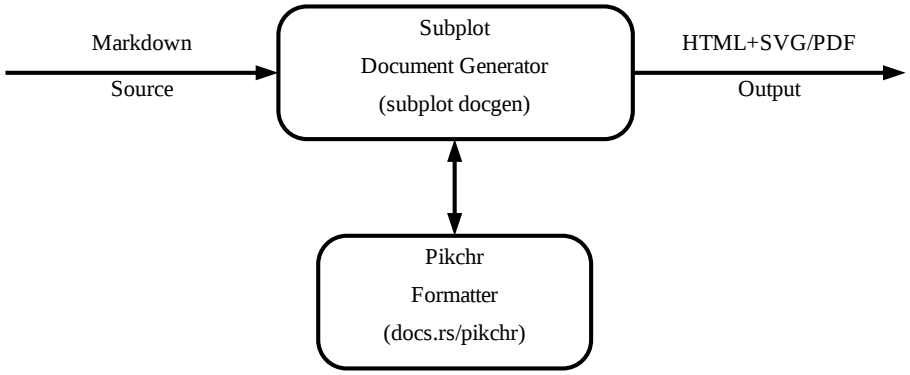
~~~{#bar.yaml. .file}
~~~
```

4.17 Embedded graphs

Subplot allows embedding markup to generate graphs into the Markdown document.

4.17.1 Pikchr

Pikchr[11] is a diagramming library which implements a Pic-like diagram language. It allows the conversion of textual descriptions of arbitrarily complex diagrams into SVGs such as this one.



The scenario checks that a graph is generated and embedded into the HTML output, and is not referenced as an external image.

given file **pikchr.md**
and an installed subplot
when I run **pandoc --filter subplot-filter pikchr.md -o pikchr.html**
then file **pikchr.html** matches regex /**img src="**/

The sample input file **pikchr.md**:

[11] `https://pikchr.org/`

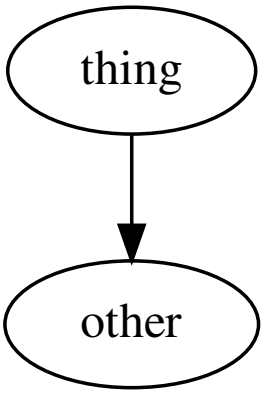
File: **pikchr.md**

```
This is an example markdown file that embeds a simple Pikchr diagram.

~~~pikchr
arrow right 200% "Markdown" "Source"
box rad 10px "Markdown" "Formatter" "(docs.rs/markdown)" fit
arrow right 200% "HTML+SVG" "Output"
arrow <-> down 70% from last box.s
box same "Pikchr" "Formatter" "(docs.rs/pikchr)" fit
~~~
```

4.17.2 Dot

Dot is a program from the Graphviz[12] suite to generate directed graphs, such as this one.



The scenario checks that a graph is generated and embedded into the HTML output, not referenced as an external image.

given file **dot.md**
and file **b.yaml**
and an installed subplot
when I run **pandoc --filter subplot-filter dot.md -o dot.html**
then file **dot.html** matches regex /**img src="**/

The sample input file **dot.md**:

File: **dot.md**

```
This is an example Markdown file, which embeds a graph using dot markup.

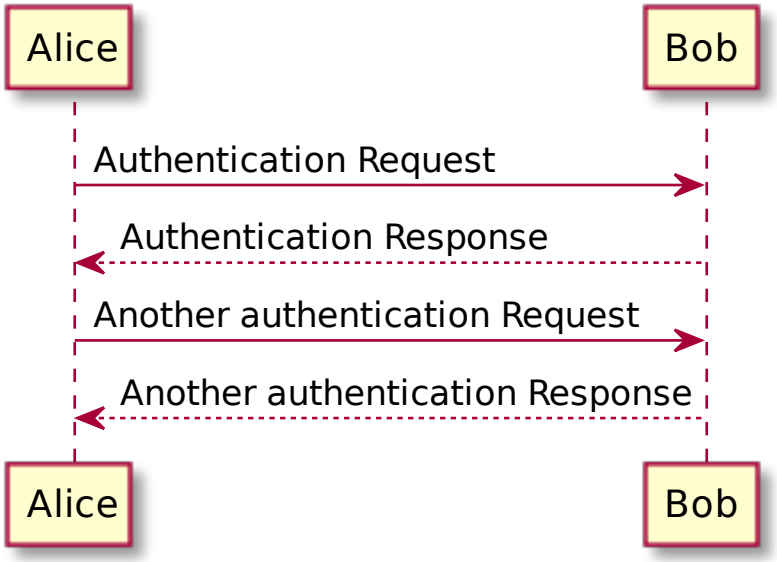
~~~dot
digraph "example" {
```

[12] http://www.graphviz.org/

```
thing -> other
}
~~~
```

4.17.3 PlantUML

PlantUML[13] is a program to generate various kinds of graphs for describing software, such as this one:



The scenario below checks that a graph is generated and embedded into the HTML output, not referenced as an external image.

given file **plantuml.md**
and file **b.yaml**
and an installed subplot
when I run **pandoc --filter subplot-filter plantuml.md -o plantuml.html**
then file **plantuml.html** matches regex /**img src="**/

The sample input file **plantuml.md**:

File: **plantuml.md**

```
This is an example Markdown file, which embeds a graph using
PlantUML markup.

~~~plantuml
@startuml
Alice -> Bob: Authentication Request
```

[13] https://plantuml.com/

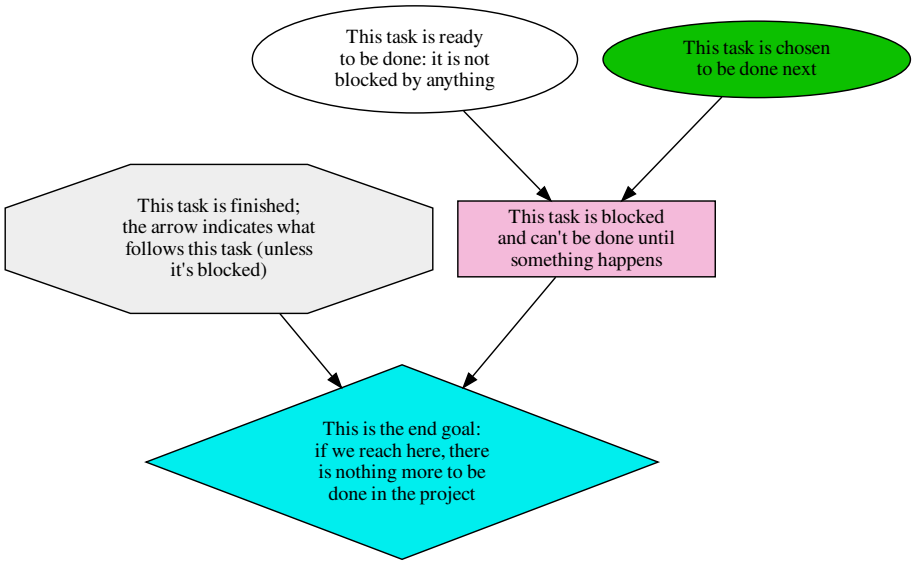
```
Bob --> Alice: Authentication Response

Alice -> Bob: Another authentication Request
Alice <-- Bob: Another authentication Response
@enduml
~~~
```

4.17.4 Roadmap

Subplot supports visual roadmaps using a YAML based markup language, implemented by the roadmap[14] Rust library. The library converts the roadmap into dot, and that gets rendered as SVG and embedded in the output document by Subplot.

An example:



This scenario checks that a graph is generated and embedded into the HTML output, not referenced as an external image.

given file **roadmap.md**
and file **b.yaml**
and an installed subplot
when I run **pandoc --filter subplot-filter roadmap.md -o roadmap.html**
then file **roadmap.html** matches regex /**img src="**/

The sample input file **roadmap.md**:

File: **roadmap.md**

[14] `https://crates.io/search?q=roadmap`

```
This is an example Markdown file, which embeds a roadmap.

~~~roadmap
goal:
  label: |
    This is the end goal:
    if we reach here, there
    is nothing more to be
    done in the project
  depends:
  - finished
  - blocked

finished:
  status: finished
  label: |
    This task is finished;
    the arrow indicates what
    follows this task (unless
    it's blocked)

ready:
  status: ready
  label: |
    This task is ready
    to be done: it is not
    blocked by anything

next:
  status: next
  label: |
    This task is chosen
    to be done next

blocked:
  status: blocked
  label: |
    This task is blocked
    and can't be done until
    something happens
  depends:
  - ready
  - next
~~~
```

4.17.5 Class name validation

When Subplot loads a document it will validate that the block classes match a known set. Subplot has a built-in set which it treats as special, and it knows some pandoc-specific classes and a number of file type classes.

If the author of a document wishes to use additional class names then they can include a `classes` list in the document metadata which subplot will treat as valid.

given file **unknown-class-name.md**
and file **known-class-name.md**
and file **b.yaml**
and an installed subplot
when I try to run **subplot docgen unknown-class-name.md -o unknown-class-name.html**
then command fails
and file **unknown-class-name.html** does not exist
and stderr contains "**Unknown classes found in the document: foobar**"
when I run **subplot docgen known-class-name.md -o known-class-name.html**
then file **known-class-name.html** exists

File: **unknown-class-name.md**

```
---
title: A document with an unknown class name
...

```foobar
This content is foobarish
```
```

File: **known-class-name.md**

```
---
title: A document with a previously unknown class name
classes:
- foobar
...

```foobar
This content is foobarish
```
```

4.18 Using as a Pandoc filter

Subplot can be used as a Pandoc *filter*, which means Pandoc can allow Subplot to modify the document while it is being converted or typeset. This can useful in

a variety of ways, such as when using Pandoc to improve Markdown processing in the ikiwiki[15] blog engine.

The way filters work is that Pandoc parses the input document into an abstract syntax tree, serializes that into JSON, gives that to the filter (via the standard input), gets a modified abstract syntax tree (again as JSON, via the filter's standard output).

Subplot supports this via the **subplot-filter** executable. It is built using the same internal logic as Subplot's docgen. The interface is merely different to be usable as a Pandoc filter.

This scenarios verifies that the filter works at all. More importantly, it does that by feeding the filter a Markdown file that does not have a YAML metadata block. For the ikiwiki use case, that's what the input files are like.

given file **justdata.md**
and an installed subplot
when I run **pandoc --filter subplot-filter justdata.md -o justdata.html**
then file **justdata.html** matches regex /**does not have a YAML metadata**/

The input file **justdata.md**:

File: **justdata.md**

```
This is an example Markdown file.
It does not have a YAML metadata block.
```

4.19 Extract embedded files

`subplot extract` extracts embedded files from a subplot file.

given file **embedded-file.md**
and file **expected.txt**
and an installed subplot
when I run **subplot extract embedded-file.md foo.txt -d .**
then files **foo.txt** and **expected.txt** match

File: **embedded-file.md**

```
---
title: Embedded file
...

~~~{#foo.txt .file}
This is a test file.
~~~
```

File: **expected.txt**

[15] `http://ikiwiki.info/`

```
1 This is a test file.
```