e3-testsuite Documentation

AdaCore

Sep 07, 2022

Contents

`e3-testsuite` is a Python library built on top of `e3-core`. Its purpose it to provide building blocks for software projects to create testsuites in a simple way. This library is generic: for instance, the tested software does not need to use Python.

Note that this manual assumes that readers are familiar with the Python language (Python3, to be specific) and how to run scripts using its standard interpreter CPython.

CHAPTER 1

Installation

`e3-testsuite` is available on Pypi, so installing it is as simple as running:

```
pip install e3-testsuite
```

CHAPTER 2

How to read this documentation

The *Core concepts* and *Tutorial* sections are must read: the former introduces notions required to understand most of the documentation and the latter put them in practice, as a step-by-step guide to write a simple, but real world testsuite.

From there, brave/curious readers can go on until the end of the documentation, while readers with time/energy constraints can just go over the sections of interest for their needs.

CHAPTER 3

Topics

3.1 Core concepts

3.1.1 Testsuite organization

All testsuites based on `e3-testsuite` have the same organization. On one side, a set of Python scripts use classes from the `e3.testsuite` package tree to implement a *testsuite framework*, and provide an entry point to launch the testsuite. On the other side, a set of *testcases* will be run by the testsuite.

By default, the testsuite framework assumes that every testcase is materialized as a directory that contains a `test.yaml` file, and that all testcases can be arbitrarily organized in a directory tree. However, this is just a default: the testcase format is completely customizeable.

3.1.2 Test results

During execution, each testcase can produce one or several *test results*. A test result contains a mandatory status (PASS, FAIL, XFAIL, ...) to determine if the test succeeded, failed, was skipped, failed in an expected way, etc. It can also contain optional metadata to provide details about the testcase execution: output, logs, timing information, and so on.

3.1.3 Test drivers

It is very common for testsuites to have lots of very similar testcases. For instance, imagine a testsuite to validate a compiler:

- Some testcases will do unit testing: they build and execute special programs (for instance: `unittest.c`) which use the compiler as a library, expecting these program not to crash.
- All other testcases come with a set of source files (say `*.foo` source files), on which the compiler will run:
  - Some testcases will check the enforcement of language legality rules: they will run the compiler and check the presence of errors.

- Some testcases will check code generation: they will run the compiler to produce an executable, then run the executable and check its output.
- Some testcases will check the generation of debug info: they will run the compiler with special options and then check the produced debug info.

There are two strategies to implement this scheme. One can create a "library" that contain helpers to run the compiler, extract error messages from the output, etc. and put in each testcase a script (for instance `test.py`) calling these helpers to implement the checking. For "legality rules enforcement" tests, this could give for instance:

```
# test.py
from support import run_compiler
result = run_compiler("my_unit.foo")
assert result.errors == ["my_unit.foo:2:5: syntax error"]
```

```
# my_unit.foo
# Syntax error ahead:
bar(|
```

An alternative strategy is to create "test macros" which, once provided testcase data, run the desired scenario: one macro would take care of unit testing, another would check legality rules enforcement, etc. This removes the need for redundant testing code in all testcases.

`e3-testsuite` uses the latter strategy: "test macros" are called *test drivers*, and by default the entry point for a testcase is the `test.yaml` file. The above example looks instead like the following:

```
# test.yaml
driver: legality-rules
errors:
   - "my_unit.foo:2:5: syntax error"
```

```
# my_unit.foo
# Syntax error ahead:
bar(|
```

Note that when testcases are just too different, so that creating one or several test drivers does not make sense, there is still the option of creating a "generic" test drivers that only runs a testcase-provided script.

To summarize: think of test drivers as programs that run on testcase data and produce a test result to describe testcase execution. All testcases need a test driver to run.

3.2 Tutorial

Let's create a simple testsuite to put *Core concepts* in practice and introduce common APIs. The goal of this testsuite will be to write tests for the `bc`, the famous POSIX command-line calculator.

3.2.1 Basic setup

First, create an empty directory and put the following Python code in a `testsuite.py` file:

```
#! /usr/bin/env python3

import sys
```

(continues on next page)

(continued from previous page)

```
from e3.testsuite import Testsuite


class BcTestsuite(Testsuite):
    """Testsuite for the bc(1) calculator."""

    pass


if __name__ == "__main__":
    sys.exit(BcTestsuite().testsuite_main())
```

Just this already makes a functional (but useless) testsuite. Make this file executable and then run it:

```
$ chmod +x testsuite.py
$ ./testsuite.py
INFO     Found 0 tests
INFO     Summary:
_  <no test result>
```

That makes sense: we have an empty testsuite, so running it actually executes no test.

3.2.2 Creating a test driver

Most testcases will check the behavior of artithmetic computations, so we have an obvious first driver to write: it will spawn a `bc` process, passing a file that contains the arithmetic expression to evaluate to it and check that the output is as expected. Testcases using this driver just need to provide the expression input file and an expected output file.

Creating a test driver is as simple as creating a Python class that derives from `e3.testsuite.drivers.TestDriver`. However, its API is quite crude, so we will study it *later*. Let's use `e3.testsuite.drivers.diff.DiffTestDriver` instead: that class conveniently provides the framework to spawn subprocesses and check their outputs against baselines, i.e. exactly what we want to do here.

Add the following class to `testsuite.py`:

```
from e3.testsuite.driver.diff import DiffTestDriver


class ComputationDriver(DiffTestDriver):
    """Driver to run a computation through "bc" and check its output.

    This passes the "input.bc" to "bc" and check its output against the
    "test.out" baseline file.
    """

    def run(self):
        self.shell(["bc", "input.bc"])
```

The only mandatory thing to do for `ClassicTestDriver` concrete subclasses (`DiffTestDriver` is an abstract subclass) is to override the `run` method. The role of this method is to do whatever actions the test driver is supposed to do in order for testcases to exercize the tested piece of software: compile software, prepare input files, run processes, and so on.

The very goal of `DiffTestDriver` is to compare a "test output" against a baseline: the test succeeds only if both match. But what's a test output? It is up to the `DiffTestDriver` subclass to define it: for example the output of one subprocess, the concatenation of several subprocess outputs or the content of a file that the testcase produces.

Subclasses must store this test output in the `self.output` attribute, and `DiffTestDriver` will then compare it against the baseline, i.e. by default: the content of the `test.out` file.

Here, the only thing we need to do is to actually run the `bc` program on our input file. `shell` is a method inherited from `ClassicTestDriver` acting as a wrapper around Python's `subprocess` standard library module. This wrapper spawns a subprocess with an empty standard input, returns its exit code and captured output (mix of standard output/error). While the only mandatory argument is a list of strings for the command-line to run, optional arguments control how to spawn this subprocess and use its result, for instance:

- `cwd` controls the working directory for the subprocess. By default, the subprocess is run in the test working directory.
- `env` allows to control environment variables passed to the subprocess. By default: leave the testsuite environment variables unchanged.
- `catch_error`: whether to consider non-zero exit code as a test failure (true by default).
- `analyze_output`: whether to append the subprocess' output to `self.output` (true by default).

Thanks to these defaults, the above call to `self.shell` will make the test succeed only if the `bc` program prints the exact expected output and stops with exit code 0.

Now that we have a test driver, we can make `BcTestsuite` aware of it:

```
class BcTestsuite(Testsuite):
    test_driver_map = {"computation": ComputationDriver}
    default_driver = "computation"
```

The `test_driver_map` class attribute maps names to test driver classes. It allows testcases to refer to the test driver they require using these names (see the next section). `default_driver` gives the name of the default test driver, for testcases that do not specify a specific driver.

With this framework, it is now possible to write actual testcases!

3.2.3 Writing tests

As described in *Core concepts*, the standard format for testcases is: any directory that contains a `test.yaml` file. By default, the testsuite searches all directories near the Python script file that subclasses `e3.testsuite.Testsuite`. In our example, that means all directories near the `testsuite.py` file, and all nested directories.

With that in mind, let's write our first testcase: create an `addition` directory next to `testsuite.py` and fill it with testcase data:

```
$ mkdir addition
$ cd addition
$ echo "driver: computation" > test.yaml
$ echo "1 + 2" > input.bc
$ echo 3 > test.out
```

Thanks to the presence of the `addition/test.yaml` file, the `addition/` directory is considered as a testcase. Its content tells the testsuite to run it using the "computation" test driver: that driver will pick the two other files as `bc`'s input and the expected output. In practice:

```
$ ./testsuite.py
INFO     Found 1 tests
INFO     PASS         addition
INFO     Summary:
_  PASS         1
```

Note: given that the "compute" test driver is the default one, `driver: computation` in the `test.yaml` file is not necessary. We can show that with a new testcase (empty `test.yaml` file):

```
$ mkdir subtraction
$ cd subtraction
$ touch test.yaml
$ echo "10 - 2" > input.bc
$ echo 8 > test.out
$ cd ..
$ ./testsuite.py
INFO     Found 2 tests
INFO     PASS         addition
INFO     PASS         subtraction
INFO     Summary:
_  PASS         2
```

3.2.4 Commonly used testsuite arguments

So far everything worked fine. What happens when there is a test failure? Let's create a faulty testcase to find out:

```
$ mkdir multiplication
$ cd multiplication
$ touch test.yaml
$ echo "2 * 3" > input.bc
$ echo 8 > test.out
$ cd ..
$ ./testsuite.py
INFO     Found 3 tests
INFO     PASS         subtraction
INFO     PASS         addition
INFO     FAIL         multiplication: unexpected output
INFO     Summary:
_  PASS         2
_  FAIL         1
```

Instead of the expected `PASS` test result, we have a `FAIL` one with a message: `unexpected output`. Even though we can easily guess the error is that the expected output should be `6` (not `8`), let's ask the testsuite to show details thanks to the `--show-error-output/-E` option. We'll also ask the testsuite to run only that failing testcase:

```
$ ./testsuite.py -E multiplication
INFO     Found 1 tests
INFO     FAIL         multiplication: unexpected output
_--- expected
_+++ output
_@@ -1 +1 @@
_-8
_+6
INFO     Summary:
_  FAIL         1
```

On baseline comparison failure, `DiffTestDriver` creates a unified diff between the baseline (`--- expected`) and the actual output (`+++ output`) showing the difference, and the testsuite's `--show-error-output/-E` option displays it, making it easy to quickly spot the difference between the two.

Even though these 3 testcases take very little time to run, most testsuites require a lot of CPU time to run to completion. Nowadays, most working stations have several cores, so we can spawn one test per core to speedup testsuite execution time. `e3.testsuite` supports the `--jobs/-j` option to achive this. This option works the same way it does for

the `make` program: `-jN` is the default (run at most N testcases at a time, default is 1), and `-j0` tells to set N to the number of CPU cores.

3.2.5 Test execution control

There is no obvious bug in `bc` that this documentation could expect to survive for long, so let's stick with this wrong `multiplication` testcase and pretend that `bc` should return `8`. This is a known bug, and so the failure is expected for the time being. This situation occurs a lot in software: bugs often take a lot of time to fix, sometimes test failures come from bugs in upstream projects, etc.

To keep testsuite reports readable/usable, it is convenient to tag failures that are temporarily accepted as `XFAIL` rather than `FAIL`: the former is a failure that has been analyzed as acceptable for now, leaving the latter for unexpected regressions to investigate. Testcases using a driver that inherits from `ClassicTestDriver` can do that by adding a `control` entry in their `test.yaml` file. To do that, append the following to `multiplication/test.yaml`:

```
control:
- [XFAIL, "True", "erroneous multiplication: see bug #1234"]
```

When present, `control` must contain a list of 2- or 3-uplets:

- A command. Here, `XFAIL` to state that failure is expected: `FAIL` test statuses are turned into `XFAIL`, and `PASS` are turned into `XPASS`. There are two other commands available: `NONE` (the default: regular test execution), and `SKIP` (do not execute the testcase and create a `SKIP` test result).
- A Python expression as a condition guard, to decide whether the command applies to this testsuite run. Here, it always applies.
- An optional message to describe why this command is here.

The first command whose guard evaluates to true applies. We can see it in action:

```
$ ./testsuite.py -j8
INFO     Found 3 tests
INFO     XFAIL        multiplication: unexpected output (erroneous multiplication:␣
↪see bug #1234)
INFO     PASS         subtraction
INFO     PASS         addition
INFO     Summary:
_  PASS         2
_  XFAIL        1
```

You can learn more about this specific test control mechanism and even how to create your own mechanism in *e3.testsuite.control: Control test execution*.

3.3 `e3.testsuite.result`: Create test results

As presented in the *Core concepts*, each testcase can produce one or several test results. But what is a test result exactly? The answer lies in the `e3.testsuite.result` module. The starting point is the `TestResult` class it defines.

3.3.1 `TestResult`

This test result class is merely a data holder. It contains:

- the name of the test corresponding to this result;

- the test status (a `TestStatus` instance: see the section below) as well as an optional one-line message to describe it;
- various information to help post-mortem investigation, should any problem occur (logs, list of subprocesses spawned during test execution, environment variables, ...).

Even though test drivers create a default `TestResult` instance (`self.result` test driver attribute), the actual registration of test results to the testsuite report is manual: test drivers must call their `push_result` method for that. This is how a single test driver instance (i.e. a single testcase) can register multiple test results.

The typical use case for a single driver instance pushing multiple test results is for testcases that contain multiple "inputs" and 1) compile a test program 2) run that program once for each input. In this case, it makes sense to create one test result per input, describing whether the software behaves as expected for each one independently rather than creating a single result that describes whether the software behaved well for *all* inputs.

This leads to the role of the test result name (`test_name` attribute of `TestResult` instances). The name of the default result that drivers create is simply the name of the testcase. This is a nice start, since it makes it super easy for someone looking at the report to relate the `FAIL foo-bar` result to the `foo/bar` testcase. By convention, drivers that create multiple results assign them names such as `TEST_NAME.INPUT_NAME`, i.e. just put a dot between the testcase name and the name of the input that triggers the emission of a separate result.

An example may help to clarify. Imagine a testsuite for a JSON handling library, and the following testcase that builds a test program that 1) parses its JSON input 2) pretty-prints that JSON document on its output:

```
parsing/
   test.yaml
   test_program.c

   empty-array.json
   empty-array.out

   zero.json
   zero.out

   ...
```

A test driver for this testcase could do the following:

- Build `test_program.c`, a program using the library to test (no test result for that).
- Run that program on `empty-array.json`, compare its output to `empty-array.out` and create a test result for that comparison, whose name is `parsing.empty-array`.
- Likewise for `zero.json` and `zero.out`, creating a `parsing.zero` test result.
- ...

Here is the exhaustive list of `TestResult` attributes:

`test_name` Name for this test result.

`env` Dictionary for the *test environment*.

`status` *Status* for this test result.

`msg` Optional (`None` or string) short message to describe this result. Strings must not contain newlines. This message usually comes as a hint to explain why the status is not `PASS`: unexpected output, expected failure from the `test.yaml` file, etc.

`log` *Log* instance to contain free-form text, for debugging purposes. Test drivers can append content to it that will be useful for post-mortem investigations if things go wrong during the test execution.

processes List of free-form information to describe the subprocesses that the test driver spawned while running the testcase, for debugging purposes. The only constraint is that this attribute must contain YAML-serializable data.

Note: This is likely redundant with the `log` attribute, so this attribute could be removed in the future.

---

failure_reasons When the test failed, optional set of reasons for the failure. This information is used only in advanced viewers, which may highlight specifically some failure reasons. For instance, highlight crashes, that may be more important to investigate than mere unexpected outputs.

expected, out and diff Drivers that compare expected and actual output to validate a testcase should initialize these with `Log` instances to hold the expected test output (`self.expected`) and the actual test output (`self.out`). It is assumed that the test fails when there is at least one difference between both.

Note that several drivers refine expected/actual outputs before running the comparison (see for instance the *output refining mechanism*). These logs are supposed to contain the outputs actually passed to the diff computation function, i.e. *after* refining, so that whatever attemps to re-compute the diff (report production, for instance) get the same result.

If, for some reason, it is not possible to store expected and actual outputs, `self.diff` can be assigned a `Log` instance holding the diff itself. For instance, the output of the `diff -u` command.

time Optional decimal number of seconds (`float`). Test drivers can use this field to track performance, most likely the time it took to run the test. Advanced results viewer can then plot the evolution of time over software evolution.

info Key/value string mapping, for unspecified use. The only restriction is that no string can contain a newline character.

3.3.2 TestStatus

This is an `Enum` subclass, allowing to classify results: tests that passed (`TestStatus.PASS`), tests that failed (`TestStatus.FAIL`), etc. For convenience, here the list of all available statuses as described in the `result.py` module:

PASS The test has run to completion and has succeeded.

FAIL The test has run enough for the testsuite to consider that it failed.

XFAIL The test has run enough for the testsuite to consider that it failed, and that this failure was expected.

XPASS The test has run to completion and has succeeded whereas a failure was expected. This corresponds to `UOK` in old AdaCore testsuites.

VERIFY The test has run to completion, but it could not self-verify the test objective (i.e. determine whether it succeeded). This test requires an additional verification action by a human or some external oracle.

SKIP The test was not executed (it has been skipped). This is appropriate when the test does not make sense in the current configuration (for instance it must run on Windows, and the current OS is GNU/Linux).

This is equivalent to DejaGnu's UNSUPPORTED, or UNTESTED test outputs.

NOT_APPLICABLE The test has run and managed to automatically determine it can't work on a given configuration (for instance, a test scenario requires two distinct interrupt priorities, but only one is supported on the current target).

The difference with SKIP is that here, the test has started when it determined that it would not work. The definition of when a test actually starts is left to the test driver.

ERROR The test could not run to completion because it is misformatted or due to an unknown error. This is very different from FAIL, because here the problem comes more likely from the testcase or the test framework rather than the tested software.

This is equivalent to DejaGnu's UNRESOLVED test output.

3.3.3 `Log`

This class acts as a holder for strings or sequences of bytes, to be used as free-form textual logs, actual output, ... in `TestResult` instances.

The only reason to have this class instead of just holding Python's `string`/`bytes` objects is to control the serialization of these logs to YAML. Interaction wiht these should be transparent to test drivers anyway, as they are intended to be used in append-only mode. For instance, to add a line to a test result's free-form log:

```
# In this example, self.result.log is already a Log instance holding a "str"
# instance.
self.result.log += "Test failed because mandatory.txt file not found.\n"
```

3.3.4 `FailureReason`

A testcase may produce `FAIL` results for very various reasons: for instance because process output is unexpected, or because the process crashed. Since crashes may be more urgent to investigate than "mere" unexpected outputs, advanced report viewers may want to highlight them specifically.

To answer this need, test drivers can set the `.failure_reasons` attribute in `TestResult` instances to a set of `FailureReason` values. `FailureReason` is an `Enum` subclass that defines the following values:

CRASH A process crash was detected. What is a "crash" is not clearly specified: it could be for instance that a "GCC internal compiler error" message is present in the test output.

TIMEOUT A process was stopped because it timed out.

MEMCHECK The tested software triggered an invalid memory access pattern. For instance, Valgrind found a conditional jump that depends on uninitialized data.

DIFF Output is not as expected.

3.4 `e3.testsuite.driver`: Core test driver API

The first sections of this part contain information that applies to all test drivers. However, starting with the *Test fragments* section, it describes the low level `TestDriver` API, to create test drivers. You should consider using it only if higher lever APIs, such as *ClassicTestDriver* and *DiffTestDriver* are not powerful enough for your needs. Still, knowing how things work under the hood may help when issues arise, so reading this part until the end can be useful at some point.

3.4.1 Basic API

All test drivers are classes that derive directly or indirectly from `e3.testsuite.driver.TestDriver`. Instances contain the following attributes:

`env` `e3.env.BaseEnv` instance, inherited from the testsuite. This object contains information about the host/build/target platforms, the testsuite parsed command-line arguments, etc. More on this in *e3.testsuite: Testsuite API*.

test_env The testcase environment. It is a dictionary that contains at least the following entries:

- `test_name`: The name that the testsuite assigned to this testcase.
- `test_dir`: The absolute name of the directory that contains the testcase.
- `working_dir`: The absolute name of the temporary directory that this test driver is free to create (see *below*) in order to run the testcase.

Depending on how the way the testcase has been created (see *e3.testsuite.testcase_finder: Control testcase discovery*), this dictionary may contain other entries: for `test.yaml`-based tests, this will also contain entries loaded from the `test.yaml` file.

result Default `TestResult` instance for this testcase. See *e3.testsuite.result: Create test results*.

3.4.2 Test/working directories

Test drivers need to deal with two directories specific to each testcase:

Test directory This is the "source" of the testcase: the directory that contains the `test.yaml` file. Consider this repertory read-only: it is bad practice to have execution modify the source of a testcase.

Working directory In order to execute a testcase, it may be necessary to create files and directories in some temporary place, for instance to build a test program. While using Python's standard mechanism to create temporary files (`tempfile` module) is an option, `e3.testsuite` provide its own temporary directory management facility, which is more helpful when investigating failures.

Each testcase is assigned a unique subdirectory inside the testsuite's temporary directory: the testcase working directory, or just "working directory". Note that the testsuite only reserves the name of that subdirectory: it is up to test drivers to actually create it, should they need it.

Inside test driver methods, directory names are available respectively as `self.test_env["test_dir"]` and `self.test_env["working_dir"]`. In addition, two shortcut methods allow to build absolute file names inside these directories: `TestDriver.test_dir` and `TestDriver.working_dir`. Both work similarly to `os.path.join`:

```
# Absolute name for the "test.yaml" in the test directory
self.test_dir("test.yaml")

# Absolute name for the "obj/foo.o" file in the working directory
self.test_dir("obj", "foo.o")
```

Warning: What follows documents the advanced API. Only complex testsuites should need this.

3.4.3 Test fragments

The `TestDriver` API deals with an abstraction called *test fragments*. In order to leverage machines with multiple cores so that testsuites run faster, we need processings to be separated into independent parts to be scheduled in parallel. Test fragments are such independent parts: the fact that a test driver can create multiple fragments for a single testcase allows finer granularity for testcase execution parallelisation compared to "a whole testcase reserves a whole core".

When a testsuite runs, it first looks for all testcases to run, then ask their test drivers to create all the test fragments they need to execute tests. Only then, a scheduler is spawned to run test fragments with the desired level of parallelism.

This design is supposed to work with workflows such as "build test program and only then run in parallel all tests using this program". To allow this, test drivers can create dependencies between test fragments. This formalism is very similar to the dependency mechanism in build software such as `make`: the scheduler will first trigger the execution of fragments with no dependency, then of fragments with dependencies satisfied, etc.

To continue with the JSON example presented in *e3.testsuite.result: Create test results*: the test driver can create a `build` fragment (with no dependency) and then one fragment per JSON document to parse (all depending on the `build` fragment). The scheduler will first trigger the execution of the `build` fragment: once this fragment has run to completion, the scheduler will be able to trigger the execution of all other fragments in parallel.

3.4.4 Creating test drivers

As described in the *tutorial*, creating a test driver implies creating a `TestDriver` subclass. The only thing such subclasses are required to do is to provide an implementation for the `add_test` method, which acts as an entry point. Note that there should be no need to override the constructor.

This `add_test` method has one purpose: register test fragments, and the `TestDriver.add_fragment` method is available to do so. This latter method has the following interface:

```
def add_fragment(self, dag, name, fun=None, after=None):
```

dag Data structure that hold fragments and that the testsuite scheduler will use to run jobs in the correct order. The `add_test` method must forward its own `dag` argument to `add_fragment`.

name String to designate this new fragment in the current testcase.

fun Test fragment callback. It must accept two positional arguments: `previous_values` and `slot`. When this test fragment is executed, this function is called and passed as `previous_values` a mapping that contains return values from previously executed fragments. Later, other test fragments executed will see `fun`'s own return value in this record under the `name` key.

If left to `None`, `add_fragment` will fetch the test driver method called `name`.

The `slot` argument is described *below*.

after List of fragment names that this new fragment depends on. The testsuite will schedule the execution of this new fragment only after all the fragments that `after` designates have been executed. Note that its execution will happen even if one or several fragments in `after` terminated with an exception.

Let's again continue with this JSON example. It is time to roll a `TestDriver` subclass, define the appropriate `add_test` method to create test fragments.

```
from glob import glob
import subprocess

from e3.testsuite.driver import TestDriver
from e3.testsuite.result import TestResult, TestStatus


class ParsingDriver(TestDriver):

    def add_test(self, dag):
        # Register the "build" fragment, no dependency. The fragment
        # callback is the "build" method.
        self.add_fragment(dag, "build")

        # For each input JSON file in the testcase directory, create a
        # fragment to run the parser on that JSON file.
```

(continues on next page)

(continued from previous page)

```
        for json_file in glob(self.test_dir("*.json")):
            input_name = os.path.splitext(json_file)[0]
            fragment_name = "parse-" + input_name
            out_file = json_file + ".out"

            self.add_fragment(
                dag=dag,

                # Unique name for this fragment (specific to json_file)
                name=fragment_name,

                # Unique callback for this fragment (likewise)
                fun=self.create_parse_callback(
                    fragment_name, json_file, out_file
                ),

                # This fragment only needs the build to happen first
                after=["build"]
            )

    def build(self, previous_values):
        """Callback for the "build" fragment."""
        # Create the temporary directory for this testcase
        os.mkdir(self.working_dir())

        # Build the test program, writing it to this temporary directory
        # (don't ever modify the testcase source directory!).
        subprocess.check_call(
            ["gcc", "-o", "test_program", self.test_dir("test_program.c")],
            cwd=self.working_dir()
        )

        # Return True to tell next fragments that the build was successful
        return True

    def create_parse_callback(self, fragment_name, json_file, out_file):
        """
        Return a callback for a "parse" fragment applied to "json_file".
        """

        def callback(previous_values):
            """Callback for the "parse" fragments."""
            # We can't do anything if the build failed
            if not previous_values.get("build"):
                return False

            # Create a result for this specific test fragment
            result = TestResult(fragment_name, self.test_env)

            # Run the test program on the input JSON, capture its output
            with open(self.test_dir(json_file), "rb") as f:
                output = subprocess.check_output(
                   ["./test_program"],
                   stdin=f,
                   stderr=subprocess.STDOUT
                )
```

(continues on next page)

(continued from previous page)

```
        # The test passes iff the output is as expected
        with open(self.test_dir(out_file), "rb") as f:
            if f.read() == output:
                result.set_status(TestStatus.PASS)
            else:
                result.set_status(TestStatus.FAIL, "unexpected output")

        # Test fragment is complete. Don't forget to register this
        # result. No fragment depends on this one, so no-one will use
        # the return value in a previous_values mapping. Yet, return
        # True as a good practice.
        self.push_result(result)
        return True
```

Note that this driver is not perfect: calls to `subprocess.check_call` and `subprocess.check_output` may raise exceptions, for instance in `test_program.c` is missing or has a syntax error, if its execution fails for some reason. Opening the `*.out` files also assumes that the file is present. In all these cases, an unhandled exception will be propagated. The testsuite framework will catch these and create an `ERROR` test result to include the error in the report, so errors will not go unnoticed (good), but the error messages will not necessarily make debugging easy (not so good).

A better driver would catch manually likely exceptions, and create `TestResult` instances with useful information, such as the name of the current step (`build` or `parse`) and the current input JSON file (if applicable) so that testcase developpers have all the information they need to understand errors when they occur.

3.4.5 Test fragment abortion

During their execution, test fragment callbacks can raise `e3.testsuite.TestAbort` exceptions: if exception propagation reaches the callback's caller, the test fragment execution will be silently discarded. This implies no entry left in `previous_values` and, unless the callback already pushed a result (`TestDriver.push_result`), there will be no track of this fragment in the test report.

However, if a callback raises another type of uncaught exception, the testsuite creates and pushes a test result with an `ERROR` status and with the exception traceback in its log, so that this error appears in the testsuite report.

3.4.6 Test fragment slot

Each test fragment can be scheduled to run in parallel, up to the parallelism level requested when running the testsuite: `--jobs=N/-jN` testsuite argument creates `N` jobs to run fragments in parallel.

Some testsuites need to create special resources for testcases to run. For instance, the testsuite for a graphical text editor running on GNU/Linux may need to spawn `Xvfb` processes (X servers) in which the text editors will run. If the testsuite can execute `N` multiple fragments in parallel, it needs at least `N` simultaneously running servers since each text editor requires the exclusive use of a server. In other words, two concurrent tests cannot use the same server.

Make each test create its own server is possible, but starting and stopping a server is costly. In order to satisfy the above requirement and keep the overhead minimal, it would be nice to start exactly `N` servers at the beginning of the testsuite (one per testsuite job): at any time, job `J` would be the only user of server `J`, so there would be no conflict between test fragments.

This is exactly the role of the `slot` argument in test fragments callback: it is a job ID between 1 and the number `N` of testsuite jobs (included). Test drivers can use it to handle shared resources avoiding conflicts.

3.4.7 Inter-test dependencies

This section presents how to create dependencies between fragments that don't belong to the same tests. But first, a warning: the design of `e3-testsuite` is thought primarily for tests that are independent: tests not interacting so that each test can be executed and not the others. Introducing inter-test dependencies removes this restriction, which introduces a fair amount of complexity:

- The execution of tests must be synchronized so that the one that depends on another one must run after it.
- There is likely logistic to take care of so that whatever justifies the dependency is carried from one test to the other.
- A test does not depend only on what is being tested, but may also depend on what other tests did, which may make tests more fragile and complicates failure analysis.
- When a user asks to run only one test, while this test happens to depend on another one, the testsuite needs to make sure that this other test is also run.

Most of the time, these drawbacks make inter-test dependencies inappropriate, and thus better avoided. However there are cases where they are necessary. Real world examples include:

- Writing an `e3-testsuite` based test harness to exercize existing inter-dependent testcases that cannot be modified. For instance, the ACATS (Ada Conformity Assessment Test Suite) has some tests which write files and other tests that then read later on.
- External constraints require separate tests to host the validation of data produced in other tests. For instance a qualification testsuite (in the context of software certification) that needs a single test (say `report-format-check`) to check that all the outputs of a qualified tool throughout the testsuite (say output of tests `feature-A`, `feature-B`, ...) respect a given constraint.

  Notice how, in this case, the outcome of such a test depends on how the testsuite is run: if `report-format-check` detects a problem in the output from `feature-A` but not in outputs from other tests, then `report-format-check` will pass or fail depending on the specific set of tests that the testsuite is requested to run.

With these pitfalls in mind, let's see how to create inter-test dependencies. First, a bit of theory regarding the logistics of test fragments in the testsuite:

The description of the *TestDriver.add_fragment method* above mentionned a crucial data structure in the testsuite: the DAG (Directed Acyclic Graph). This graph (an instance of `e3.collections.dag.DAG`) contains the list of fragments to run as nodes and the dependencies between these fragments as edges. The DAG is then is used to schedule their execution: first execute fragments that have no dependencies, then fragments that depend on these, etc.

Each node in this graph is a `FragmentData` instance, that the `add_fragment` method creates. This class has four fields:

- `uid`, a string used as an identifier for this fragment that is unique in the whole DAG (it corresponds to the `VertexID` generic type in `e3.collections.dag`). `add_fragment` automatically creates it from the driver's `test_name` field and `add_fragment`'s own `name` argument.
- `driver`, the test driver that created this fragment.
- `name`, the `name` argument passed to `add_fragment`.
- `callback`, the `fun` argument passed to `add_fragment`.

Our goal here is, once the DAG is populated with all the `FragmentData` to run, to add dependencies between them to express scheduling constraints. Overriding the `Testsuite.adjust_dag_dependencies` method allows this: this method is called when the DAG was created and populated, and right before the scheduling and starting the execution of fragments.

As as simplistic example, suppose that a testsuite has two kinds of drivers: `ComputeNumberDriver` and `SumDriver`. Tests running with `ComputeNumberDriver` have no dependencies, while each test using `SumDriver` needs the result of all `ComputeNumberDriver` (i.e. depends on all of them). Also assume that each driver creates only one fragment (more on this later), then the following method overriding would do the job:

```
def adjust_dag_dependencies(self, dag: DAG) -> None:
    # Get the list of all fragments for...

    # ... ComputeNumberDriver
    comp_fragments = []

    # ... SumDriver
    sum_fragments = []

    # "dag.vertex_data" is a dict that maps fragment UIDs to FragmentData
    # instances.
    for fg in dag.vertex_data.values():
        if isinstance(fg.driver, ComputeNumberDriver):
            comp_fragments.append(fg)
        elif isinstance(fg.driver, SumDriver):
            sum_fragments.append(fg)

    # Pass the list of ComputeNumberDriver fragments to all SumDriver
    # instances and make sure SumDriver fragments run after all
    # ComputeNumberDriver ones.
    comp_uids = [fg.uid for fg in comp_fragments]
    for fg in sum_fragments:
        # This allows code in SumDriver to have access to all
        # ComputeNumberDriver fragments.
        fg.driver.comp_fragments = comp_fragments

        # This creates the scheduling constraint: the "fg" fragment must
        # run only after all "comp_uids" fragments have run.
        dag.update_vertex(vertex_id=fg.uid, predecessors=comp_uids)
```

Note the use of the `DAG.update_vertex` method rather than `.set_predecessors`: the former adds predecessors (i.e. preserves existing ones, that the `TestDriver.add_fragment` method already created) while the latter would override them.

Some drivers create more than one fragment: for instance `e3.testsuite.driver.BasicDriver` creates a `set_up` fragment, a `run` one, a `tear_down` one and a `analyze` one, which each fragment having a dependency on the previous one. To deal with them, `adjust_dag_dependencies` need to check the `FragmentData.name` field to get access to specific fragments:

```
# Look for the "run" fragment from FooDriver tests
if fg.name == "run" and isinstance(fg.driver, FooDriver):
   ...

# FragmentData provides a helper to do this:
if fg.matches(FooDriver, "run"):
   ...
```

3.5 `e3.testsuite.driver.classic`: Common test driver facilities

The `driver.classic` module's main contribution is to provides a `TestDriver` convenience subclass: `ClassicTestDriver`, that test driver implementations are invited to derive from.

It starts with an assumption, considered to be common to most real world use cases: testcases are atomic, meaning that the execution of each testcase is a single chunk of work that produces a single test result. This assumption allows to provide a simpler framework compared to the base `TestDriver` API, so that test drivers are easier to write.

First, there is no need to create *fragments* and handle dependencies: the minimal requirement for `ClassicTestDriver` subclasses is to define a `run` method. As you have probably guessed, its sole responsibility is to proceed to testcase execution: build what needs to be built, spawn subprocesses as needed, etc.

3.5.1 Working directory management

`ClassicTestDriver` considers that most drivers will need to create the temporary directories, and thus make it the default: before running the testcase, this driver will copy the test directory to the working directory. Subclasses can override this behavior overriding the `copy_test_directory` property. For instance, to disable this copy unconditionally:

```
class MyDriver(ClassicTestDriver):
    copy_test_directory = False
```

Alternatively, to disable it only if the `test.yaml` file contains a `no-copy` entry:

```
class MyDriver(ClassicTestDriver):
    @property
    def copy_test_directory(self):
        return not self.test_env.get("no-copy")
```

3.5.2 Output encodings

Although the concept of "test output" is not precisely defined here, `ClassicTestDriver` has provisions for the very common pattern of drivers that build a string (the test output) and that, once the test has run, analyze of the content of this output determines whether the testcase passed or failed. For this reason, the `self.output` attribute contains a `Log` instance (see *Log*).

Although drivers generally want to deal with actual strings (`str` in Python3, a valid sequence of Unicode codepoints), at the OS level, process outputs are mere sequences of bytes (`bytes` in Python3), i.e. binary data. Such drivers need to decode the sequence of bytes into strings, and for that they need to pick the appropriate *encoding* (UTF-8, ISO-8859-1, ...).

The `default_encoding` property returns the name of the default encoding used to decode process outputs (as accepted by the `str.encode()` method: `utf-8`, `latin-1`, ...). If it returns `binary`, outputs are not decoded and `self.output` is set to a `Log` instance that holds bytes.

The default implementation for this property returns the `encoding` entry from the `self.test_env` dict. If there is no such entry, it returns `utf-8` (the most commonly used encoding these days).

3.5.3 Spawning subprocesses

Spawning subprocesses is so common that this driver class provides a convenience method to do it:

```
def shell(self, args, cwd=None, env=None, catch_error=True,
          analyze_output=True, timeout=None, encoding=None):
```

This will run a subprocess given a list of command-line arguments (`args`); its standard input is redirected to `/dev/null` while both its standard output/error streams are collected as a single stream. `shell` returns a

`ProcessResult` instance once the subprocess exitted. `ProcessResult` is just a holder for process information: its `status` attribute contains the process exit code (an integer) while its `out` attribute contains the captured output.

Note that the `shell` method also automatically appends a description of the spawned subprocess (arguments, working directory, exit code, output) to the *test result log*.

Its other arguments give finer control over process execution:

`cwd` Without surprise for people familiar with process handling APIs: this argument controls the directory in which the subprocess is spawned. When left to `None`, the processed is spawned in the working directory.

`env` Environment variables to pass to the subprocess. If left to `None`, the subprocess inherit the Python interpreter's environment.

`catch_error` If true (the default), `shell` will check the exit status: if it is 0, nothing happen, however if it is anything else, `shell` raises an exception to abort the testcase with a failure (see *Exception-based execution control* for more details). If set to false, nothing special happens for non-0 exit statuses.

`analyze_output` Whether to append the subprocess output to `self.output` (see *Output encodings*). This is for convenience in test drivers based on output comparison (see *e3.testsuite.driver.diff: Test driver for actual/expected outputs*).

`timeout` Number of seconds to allow for the subprocess execution: if it lasts longer, the subprocess is aborted and its status code is set to non-zero.

If left to `None`, use instead the timeout that the `default_process_timeout` property returns. The `ClassicTestDriver` implementation for that property returns either the `timeout` entry from `self.test_env` (if present) or 300 seconds (5 minutes). Of course, subclasses are free to override this property if needed.

`encoding` Name of the encoding used to decode the subprocess output. If left to `None`, use instead the encoding that the `default_encoding` property returns (see *Output encodings*). Here, too, the default implementation returns the `encoding` entry from `self.test_env` (if present) or `utf-8`. Again, subclasses are free to override this property if needed.

`truncate_logs_threshold` Natural number, threshold to truncate the subprocess output that `shell` logs in the *test result log*. This threshold is interpreted as half the number of output lines allowed before truncation, and 0 means that truncation is disabled. If left to `None`, use the testsuite's `--truncate-logs` option.

3.5.4 Set up/analyze/tear down

The common organization for test driver execution has four parts:

1. Initialization: make sure input is valid: required files must be present (test program sources, input files), metadata is valid, start a server, and so on.
2. Execution: the meat happens here: run the necessary programs, write the necessary files, ...
3. Analysis: look at the test output and decide whether the test passed.
4. Finalization: free resources, shut down the server, ..

`ClassicTestDriver` defines four overridable methods, one for each step: `set_up`, `run`, `analyze` and `tear_down`. First, the `set_up` method is called, then the `run` one and then the `analyze` one. So far, any unhandled exception in these methods would prevent the next ones to run. Except for the `tear_down` method, which is called no matter what happens as long as the `set_up` method was called.

The following example shows how this is useful. Imagine a testsuite for a database server. We want some test drivers only to start the server (leaving the rest to testcases) while we want other test drivers to perform more involved server initialization.

```
class BaseDriver(ClassicTestDriver):
    def set_up(self):
        super().set_up()
        self.start_server()

    def run(self):
        pass  # ...

    def tear_down(self):
        self.stop_server()
        super().tear_down()

class FixturesDriver(BaseDriver):
    def set_up(self):
        super(FixturesDriver, self).set_up()
        self.install_fixtures()
```

The `install_fixtures()` call has to happen after the `start_server()` one, but before the actual test execution (`run()`). If initialization, execution and finalization all happened in `BaseDriver.run`, it would not be possible for `FixturesDriver` to insert the call at the proper place.

Note that `ClassicTestDriver` provide valid default implementations for all these methods except `run`, which subclasses have to override.

The `analyze` method is interesting: its default implementation calls the `compute_failures` method, which returns a list of error messages. If that list is empty, it considers that there is no test failure, and thus that the testcase passed. Otherwise, it considers that the test failed. In both cases, it appropriately set the status/message in `self.result` and pushes it to the testsuite report.

That means that in practice, test drivers only need to override this `compute_failures` method in order to properly analyze test output. For instance, let's consider a test driver whose `run` method spawns a supbrocess and must consider that the test succeeds iff the `SUCCESS` string appears in the output. The following would do the job:

```
class FooDriver(ClassicTestDriver):
    def run(self):
        self.shell(...)

    def compute_failures(self):
        return (["no match for SUCCESS in output"]
                if "SUCCESS" not in self.output
                else [])
```

3.5.5 Metadata-based execution control

Deciding whether to skip a testcase, or expecting a test failure are both so common that `ClassicTestDriver` provides a mechanism which makes it possible to control testcase execution thanks to metadata in that testcase.

By default, it is based on metadata from the test environment (`self.test_env`, i.e. from the `test.yaml` file), but each driver can customize this. This mechanism is described extensively in *e3.testsuite.control: Control test execution*.

3.5.6 Exception-based execution control

The `e3.testsuite.driver.classic` module defines several exceptions that `ClassicTestDriver` subclasses can use to control the execution of testcases. These exceptions are expected to be propagated from the `set_up`,

`run` and `analyze` methods when appropriate. When they are, this stops the execution of the testcase (next methods are not run). Please refer to *TestStatus* for the meaning of test statuses.

`TestSkip` Abort the testcase and push a `SKIP` test result.

`TestAbortWithError` Abort the testcase and push an `ERROR` test result.

`TestAbortWithFailure` Abort the testcase and push a `FAIL` test result, or `XFAIL` if a failure is expected (see *e3.testsuite.control: Control test execution*).

3.5.7 Colors

Long raw text logs can be difficult to read quickly. Light formatting (color, brightness) can help in this area, revealing the structure of text logs. Since it relies on the `e3-core` project, `e3-testsuite` already has the colorama project in its dependencies.

`ClassicTestDriver` subclasses can use `self.Fore` and `self.Style` attributes as "smart" shortcuts for `colorama.Fore` and `colorama.Style`: if there is a single chance for text logs to be redirected to a text file (rather than everything to be printed in consoles), colors support is disable and these two attributes yield empty strings instead of the regular console escape sequences.

The `shell` method already uses them to format the logging of subprocesses in `self.result.log`:

```
self.result.log += (
    self.Style.RESET_ALL + self.Style.BRIGHT
    + "Status code" + self.Style.RESET_ALL
    + ": " + self.Style.DIM + str(p.status) + self.Style.RESET_ALL
)
```

This will format `Status code` in bright style and the status code in dim style if formatting is enabled, and will just return `Status code:  0` without formatting when disabled.

3.5.8 Test fragment slot

Even though each testcase using a `ClassicTestDriver` subclass has a single test fragment, it can be useful for drivers to know which *slot* they are being run on. The slot is available in the `self.slot` driver attribute.

3.6 `e3.testsuite.driver.diff`: Test driver for actual/expected outputs

The `driver.diff` module defines `DiffTestDriver`, a `ClassicTestDriver` subclass specialized for drivers whose analysis is based on output comparisons. It also defines several helper classes to control more precisely the comparison process.

3.6.1 Basic API

The whole `ClassicTestDriver` API is available in `DiffTestDriver`, and the overriding requirements are the same:

- subclasses must override the `run` method;
- they can, if need be, override the `set_up` and `tear_down` methods.

Note however that unlike its parent class, it provides an actually useful `compute_failures` method override, which compares the test actual output and the output baseline:

- The test actual output is what the `self.output Log` instance holds: this is where the `analyze_output` from the *shell method* matters.
- The output baseline, which we could also call the *test expected output*, is by default the content of the `test.out` file, in the test directory. As explained *below*, this defalut can be changed.

Thanks to this subclass, writing real world test drivers requires little code. The following example just runs the `my_program` executable with arguments provided in the `test.yaml` file, and checks that its status code is 0 and that its output matches the content of the `test.yaml` file:

```
from e3.testsuite.driver.diff import DiffTestDriver


class MyTestDriver(DiffTestDriver):
    def run(self):
        argv = self.test_env.get("argv", [])
        self.shell(["my_program"] + argv)
```

3.6.2 Output encodings

See *Output encodings* for basic notions regarding string encoding/decoding concerns in `ClassicTestDriver` and all its subclasses.

In binary mode (the `default_encoding` property returns `binary`), `self.output` is initialized to contain a `Log` instance holding `bytes`. The `shell` method doesn't decode process outputs: they stay as `bytes` and thus their concatenation to `self.output` is valid. In addition, the baseline file (`test.out` by default) is read in binary mode, so in the end, `DiffTestDriver` only deals with `bytes` instances.

Conversely, in text mode, `self.output` is a `Log` instance holding `str` objects, which the `shell` method extends with decoded process outputs, and finally the baseline file is read in text mode, decoded using the same string encoding.

3.6.3 Handling output variations

In some cases, program outputs can contain unpredictible parts. For instance, the following script:

```
o = object()
print(o)
```

Can have the following output:

```
$ python foo.py
<object object at 0x7f15fbce1970>
```

... or the following:

```
$ python foo.py
<object object at 0x7f4f9e031970>
```

Although it's theoretically possible to constrain the execution environment enough to make the printed address constant, it is hardly practical. There can be a lot of other sources of output variation: printing the current date, timing information, etc.

`DiffTestDriver` provides two alternative mechanisms to handle such cases: match actual output against regular expressions, or refine outputs before the comparison.

Regexp-based matching

Instead of providing a file that contains byte-per-byte or codepoint-per-codepoint expected output, the baseline can be considered as a regular expression. With this mechanism, the following `test.out`:

```
<object object at 0x.*>
```

will match the output of the `foo.py` example script above. This relies on Python's standard `re` module: please refer to its documentation for the syntax reference and the available regexp features.

In order to switch to regexp-matching on a per-testcase basis, just add the following to the `test.yaml` file:

```
baseline_regexp: True
```

Output refining

Another option to match varying outputs is to refine them, i.e. perform substitutions to hide varying parts from the comparison. Applied to the previous example, the goal is to refine such outputs:

```
<object object at 0x7f15fbce1970>
```

To a string such as following:

```
<object object at [HEX-ADDR]>
```

To achieve this goal, the `driver.diff` module defines the following abstract class:

```
class OutputRefiner:
    def refine(self, output):
        raise NotImplementedError
```

Subclasses must override the `refine` method so that it takes the original output (`output` argument) and return the refined output. Note that depending on the encoding, `output` can be either a string (`str` instance) or binary data (`bytes` instance): in each case it must return an object that has the same type as the `output` argument.

Several very common subclasses are available in `driver.diff`:

`Substitute(substring, replacement="")` Replace a specific substring. For instance:

```
# Just remove occurences of <foo>
# (replace them with an empty string)
Substitute("<foo>")

# Replace occurences of <foo> with <bar>
Substitute("<foo>", "<bar>")
```

`ReplacePath(path, replacement="")` Replace a specific filename: `path` itself, the corresponding absolute path or the corresponding Unix-style path.

`PatternSubstitute(pattern, replacements="")` Replace anything matching the `pattern` regular expression.

Using output refiners from `DiffTestDriver` instances is very easy: just override the `output_refiners` property in subclasses to return a list of `OutputRefiner` to apply on actual outputs before comparing them with baselines.

To complete the `foo.py` example above, thanks to the following overriding:

```
@property
def output_refiners(self):
    return [PatternSubstitute("0x[0-9a-f]+", "[HEX-ADDR]")]
```

All refined outputs from `foo.py` would match the following baseline:

```
<object object at [HEX-ADDR]>
```

Note that even though refiners only apply to actual outputs by default, it is possible to also apply them to baselines. To do this, override the `refine_baseline` property:

```
@property
def refine_baseline(self):
    return True
```

This behavior is disabled by default because a very common refinment is to remove occurences of the working directory from the test output. In that case, baselines that contain the working directory (for instance `/home/user/my-testsuite/tmp/my-test`) will be refined as expected with the setup of the original testcase author, but will not on another setup (for instance when the working directory is `/tmp/testsuite-tmp-dir`).

3.6.4 Alternative baselines

`DiffTestDriver` subclasses can override two properties in order to select the baseline to use as well as the output matching mode (equality vs. regexp):

The `baseline_file` property must return a `(filename, is_regexp)` couple. The first item is the name of the baseline file (relative to the test directory), i.e. the file that contains the output baseline. The second one is a boolean that determines whether to use the regexp matching mode (if true) or the equality mode (if false).

If, for some reason (for instance: extracting the baseline is more involved than just reading the content of a file) the above is not powerful enough, it is possible instead to override the `baseline` property. In that case, the `baseline_file` property is ignored, and `baseline` must return a 3-element tuple:

1. The absolute filename for the baseline file, if any, `None` otherwise. Only a present filename allows *baseline rewriting*.
2. The baseline itself: a string in text mode, and a `bytes` instance in binary mode.
3. Whether the baseline is a regexp.

3.6.5 Automatic baseline rewriting

Often, test baselines depend on formatting rules that need to evolve over time. For example, imagine a testsuite for a program that keeps track of daily min/max temperatures. The following could be a plausible test baseline:

```
01/01/2020 260.3 273.1
01/02/2020 269.2 273.2
```

At some point, it is decided to change the format for dates. All baselines need to be rewritten, so the above must become:

```
2020-01-01 260.3 273.1
2020-01-02 269.2 273.2
```

That implies manually rewriting the baselines of potentially a lot of tests.

`DiffTestDriver` makes it possible to automatically rewrite baselines for all tests based on equality (not regexps). Of course, this is disabled by default: one needs to run it only when such pervasive output changes are expected, and baseline updates need to be carefully reviewed afterwards.

Enabling this behavior is as simple as setting `self.env.rewrite_baselines` to True in the `Testsuite` instance. The APIs to use for this are properly introduced later, in *e3.testsuite: Testsuite API*. Here is a short example, in the meantime:

```
class MyTestsuite(Testsuite):

    # Add a command-line flag to the testsuite script to allow users to
    # trigger baseline rewriting.
    def add_options(self, parser):
        parser.add_argument(
            "--rewrite", action="store_true",
            help="Rewrite test baselines according to current outputs"
        )

    # Before running the testsuite, keep track in the environment of our
    # desire to rewrite baselines. DiffTestDriver instances will pick it up
    # automatically from there.
    def set_up(self):
        super(MyTestsuite, self).set_up()
        self.env.rewrite_baselines = self.main.args.rewrite
```

Note that baseline rewriting applies only to tests that are not already expected to fail. Imagine for instance the situation described above (date format change), and the following testcase:

```
# test.yaml
control:
   - [XFAIL, "True",
      "Precision bug: max temperature is 280.1 while it should be 280.0"]
```

```
# test.out
01/01/2020 270.3 280.1
```

The testsuite must not rewrite `test.out`, otherwise the precision bug (`280.1` instead of `280.0`) will be recorded in the baseline, and thus the testcase will incorrectly start to pass (`XPASS`). But this is just a compromise: in the future, the testcase will fail not only because of the lack of precision, but also because of the bad date formatting, so in such cases, baselines must be manually updated.

3.7 `e3.testsuite.control`: Control test execution

Expecting all testcases in a testsuite to run and pass is not always realistic. There are two reasons for this.

Some tests may exercize features that make sense only on a specific OS: imagine for instance a "Windows registry edit" feature, which would make no sense on GNU/Linux or MacOS systems. It makes no sense to even run such tests when not in the appropriate environment.

In parallel: even though our ideal is to have perfect software, real world programs have many bugs. Some are easy to fix, but some are so hard that they can take days, months or even *years* to resolve. Creating testcases for bugs that are not fixed yet makes sense: such tests allow to keep track of "known" bugs, in particular when they unexpectedly pass whereas the bug is already supposed to be around. Running such tests has value, but clutters the testsuite reports, potentially hiding unexpected failures in the middle of many known ones.

For the former, it is appropriate to create SKIP test results (you can read more about test statuses in *TestStatus*). The latter is the raison d'être of the PASS/XPASS and FAIL/XFAIL distinctions: in theory all results should be PASS or XFAIL, so when looking for regressions after a software update, one only needs to look at XPASS and FAIL statuses.

3.7.1 Basic API

The need to control whether to execute testcases and how to "transform" its test status (PASS to XPASS, FAIL to XFAIL) is so common that `e3-testsuite` provides an abstraction for that: the `TestControl` class and the `TestControlCreator` interface.

Note that even though this API was initially created as a helper for *ClassicTestDriver*, it is designed separately so that it can be reused in other drivers.

`TestControl` is just a data structure to hold the decision regarding test control:

- the `skip` attribute is a boolean, specifying whether to skip the test;
- the `xfail` attribute is a boolean, telling whether a failure is expected
- the `message` attribute is an optional string: a message to convey the reason behind this decision.

The goal is to have one `TestControl` instance per test result to create.

`TestControlCreator` instances allow test drivers to instantiate `TestControl` once per test result: their `create` method takes a test driver and must return a `TestControl` instance.

The integration of this API in `ClassicTestDriver` is simple:

- In test driver subclasses, override the `test_control_creator` property to return a `TestControlCreator` instance.
- When the test is about to be executed, `ClassicTestDriver` will use this instance to get a `TestControl` object.
- Based on this object, the test will be skipped (creating a SKIP test result) or executed normally, and PASS/FAIL test result will be turned into XPASS/XFAIL if this object states that a failure is expected.

There is a control mechanism set up by default: the `ClassicTestDriver.test_control_creator` property returns a `YAMLTestControlCreator` instance.

3.7.2 `YAMLTestControlCreator`

This object creates `TestControl` instances from test environment (`self.test_env` in test driver instances), i.e. from the `test.yaml` file in most cases (the *e3.testsuite.testcase_finder: Control testcase discovery* later section describes when it's not). The idea is very simple: let each testcase specify when to skip execution/expect a failure depending on the environment (host OS, testsuite options, etc.).

To achieve this, several "verbs" are available:

`NONE` Just run the testcase the regular way. This is the default.

`SKIP` Do not run the testcase and create a SKIP test result.

`XFAIL` Run the testcase the regular way, expecting a failure: if the test passes, emit a XPASS test result, emit a XFAIL one otherwise.

Testcases can then put metadata in their `test.yaml`:

```
driver: my_driver
control:
- [SKIP, "env.build.os != 'Windows'", "Tests a Windows-specific feature"]
- [XFAIL, "True", "See bug #1234"]
```

The `control` entry must contain a list of entries. Each entry contains a verb, a Python boolean expression, and an optional message. The entries are processed in order: only the first for which the boolean expression returns true is considered. The verb and the message determine how to create the `TestControl` object.

But where does the `env` variable comes from in the example above? When evaluating a boolean expression, `YAMLTestCreator` passes it variables corresponding to the `condition_env` argument constructor argument, plus the testsuite environment (`self.env` in test drivers) as `env`. Please refer to the e3.env documentation to know more about environments, which are instances of the `AbstractBaseEnv` subclasses.

```
tcc = YAMLTestControlCreator({"mode": "debug", "cpus": 8})

# Condition expressions in driver.test_env["control"] will have access to
# three variables: mode (containing the "debug" string), cpus (containing
# the 8 integer) and env.
tcc.create(driver)
```

`ClassicTestDriver.test_control_creator` instantiates `YAMLTestControlCreator` with an empty condition environment, so by default, only `env` is available.

With the example above, a `YAMLTestControlCreator` instance will create:

- `TestControl("Tests a Windows-specific feature", skip=True, xfail=False)` on every OS but Windows;
- `TestControl("See bug #1234", skip=False, xfail=True)` on Windows.

3.8 `e3.testsuite`: Testsuite API

So far, this documentation focused on writing test drivers. Although these really are the meat of each testsuite, there are also testsuite-wide features and customizations to consider.

3.8.1 Test drivers

The *Tutorial* already covered how to register the set of test drivers in the testsuite, so that each testcase can chose which driver to use. Just creating `TestDriver` subclasses is not enough: testsuite must associate a name to each available driver.

This all happens in `Testsuite.test_driver_map`, which as usual can be either a class attribute or a property. It must contain/return a dict, mapping driver names to `TestDriver` subclasses:

```
from e3.testsuite import Testsuite
from e3.testsuite.driver import TestDriver


class MyDriver1(TestDriver):
    # ...
    pass


class MyDriver2(TestDriver):
```

(continues on next page)

(continued from previous page)

```
    # ...
    pass


class MyTestsuite(Testsuite):
    test_driver_map = {"driver1": MyDriver1, "driver2": MyDriver2}
```

This is the only mandatory customization when creating a `Testsuite` subclass. A nice optional addition is the definition of a default driver: if most testcases use a single test driver, this will make it handier to create tests.

```
class MyTestsuite(Testsuite):
    test_driver_map = {"driver1": MyDriver1, "driver2": MyDriver2}

    # Testcases that don't specify a "driver" in their test.yaml file will
    # automatically run with MyDriver2.
    default_driver = "driver2"
```

3.8.2 Testsuite environment

`Testsuite` and `TestDriver` instances all have a `self.env` attribute. This holds a `e3.env.BaseEnv` instance: the testsuite originally creates it when starting and forwards it to test drivers.

This environment holds information about the platform for which tests are running (host OS, target CPU, ... as well as parsed options from the command-line (see below). The testsuite is also free to add more information to this environment.

If a testsuite actually needs to deal with non-native targets, for instance running on GNU/Linux for x86_64 tests that involve programs for bare ARM ELF targets, then it's useful to override the `enable_cross_support` class attribute/property to return true (it returns false by default):

```
class MyTestsuite(Testsuite):
    enable_cross_support = True
```

In this case, the testsuite will add `--build`, `--host` and `--target` command-line arguments. These have the same semantics as the homonym options in GNU `configure` scripts: see The GNU configure and build system. The testsuite will then use these arguments to build the appropriate environment in `self.env`, and thus for instance `self.env.target.cpu.name` will reflect the target CPU.

3.8.3 Command-line options

Note: This section assumes that readers are familiar with Python's famous `argparse` standard package. Please read its documentation if this is the first time you hear about it.

Testsuites often have multiple operating modes. A very common mode is: does it run programs under Valgrind? Doing this has great value, as it helps finding invalid memory accesses, use of uninitialized values, etc. but comes at a great performance cost. So always using Valgrind is not realistic.

Adding a testsuite command-line option is a way to solve this problem: by default (for the most common cases: day-to-day development runs) Valgrind support is disabled, and the testsuite enables it when run with a `--valgrind` argument (used in continuous builders, for instance).

Adding testsuite options is very simple: in the `Testsuite` subclass, override the `add_options` method. It takes a single argument: the `argparse.ArgumentParser` instance that is responsible for parsing the testsuite command-line arguments. To implement the Valgrind example discussed above, we can have:

```
class MyTestsuite(Testsuite):
    def add_options(self, parser):
        parser.add_argument("--valgrind", action="store_true",
                            help="Run tests under Valgrind")
```

The result of command-line parsing, i.e. the result of `parser.parse_args()` is made available in `self.env.options`. This means that test drivers can then check for the presence of the `--valgrind` on the command line the following way:

```
class MyDriver(ClassicTestDriver):
    def run(self):
        argv = self.test_program_command_line

        # If the testsuite is run with the --valgrind option, run the test
        # program under Valgrind.
        if self.env.options.valgrind:
            argv = ["valgrind", "--leak-check=full", "-q"] + argv

        self.shell(argv)
```

3.8.4 Set up/tear down

Testsuites that need to execute arbitrary operations right before looking for tests and running them can override the `Testsuite.set_up` method. Similarly, testsuites that need to execute actions after all testcases ran to completion and after testsuite reports were emitted can override the `Testsuite.tear_down` method.

```
class MyTestsuite(Testsuite):
    def set_up(self):
        # Let the base class' set_up method do its job
        super().set_up()

        # Then do whatever is required before running testcases.
        # Note that by the time this is executed, command-line
        # options are parsed and the environment (self.env)
        # is fully initialized.

        # ...

    def tear_down(self):
        # Do whatever is required to after the testsuite has
        # run to completion.

        # ...

        # Then let the base class' tear_down method do its job
        super().tear_down()
```

3.8.5 Overriding tests subdirectory

As described in the *tutorial*, by default the testsuite looks for tests in the testsuite root directory, i.e. the directory that contains the Python script in which `e3.testsuite.Testsuite` is subclassed. Testsuites can override this

behavior with the `tests_subdir` property:

```
class MyTestsuite(Testsuite):
    @property
    def tests_subdir(self):
        return "tests"
```

This property must return a directory name that is relative to the testsuite root: testcases are looked for in all of its subdirectories.

The *next section* describes how to go deeper and change the testcase discovery process itself.

3.8.6 Changing the testcase naming scheme

Testsuite require unique names for all testcases. These name must be valid filenames: no directory separator or special character such as `:` are allowed.

By default, this name is computed from the name of the testcase directory, relative to the tests subdirectory: directory separators are just replaced with `__` (two underscores). For instance, the testcase `a/b-c/d` is assigned the `a__b-c__d` name.

Changing the naming scheme is as easy as overriding the `test_name` method, which takes the name of the test directory and must return the test name, conforming to the constraints described above:

```
class MyTestsuite(Testsuite):
    def test_name(self, test_dir):
        return custom_computation(test_dir)
```

3.9 `e3.testsuite.testcase_finder`: Control testcase discovery

In *Core concepts*, the default format for testcases is described as: any directory that contains a `test.yaml` file. This section shows the mechanisms to implement different formats.

Internally, the testsuite creates testcases from a list of `e3.testsuite.testcase_finder.ParsedTest` instances: precisely one testcase per `ParsedTest` object. This class is just a holder for the information required to create a testcase, it contains the following attributes:

`test_name` Name for this testcase, generally computed from `test_dir` using `Testsuite.test_name` (see *Changing the testcase naming scheme*). Only one testcase can have a specific name, or put differently: test names are unique.

`driver_cls` `TestDriver` subclass to instantiate for this testcase. When left to `None`, the testsuite will use the default driver (*if available*).

`test_env` Dictionary for the *test environment*.

`test_dir` Name of the directory that contains the testcase.

`test_matcher` Optional "matching name", for filtering purposes, i.e. to run the testsuite on a subset of tests. See *below*.

The next piece of code, responsible to create `ParsedTest` instances, is the `e3.testsuite.testcase_finder.TestFinder` interface. This API is very simple: `TestFinder` objects must support a `probe(testsuite, dirpath, dirnames, filenames)` method, which is called for each directory that is a candidate to be a testcase. The semantics for `probe` arguments are:

`testsuite` Testsuite instance that is looking for testcases.

dirpath Absolute name for the candidate directory to probe.

dirnames Base names for `dirpath` subdirectories.

filenames Basenames for files in `dirpath`.

When called, `TestFinder.probe` overriding methods are supposed to look at `dirpath`, `dirnames` and `filenames` to determine whether this directory contains testcases. It must return a list of `ParsedTest` instances: each one will later be used to instantiate a `TestDriver` subclass for this testcase.

Note: For backwards compatibility, `probe` methods can return `None` instead of an empty list when there is no testcase, and can return directly a `ParsedTest` instance instead of a list of one element when the probed directory contains exactly one testcase.

The default `TestFinder` instance that testsuites use come from the `e3.testsuite.testcase_finder.YAMLTestFinder` class. Its probe method is very simple: consider there is a testcase iff there is `test.yaml` is present in `filenames`. In that case, parse its YAML content, use the result as the test environment and look for a `driver` environment entry to fetch the corresponding test driver.

The `Testsuite.get_test_list` internal method is the one that takes care of running the search for tests in the appropriate directories: in the testsuite root directory, or in directories passed in argument to the testsuite, and delegates the actual "testcase decoding" to `TestFinder` instances.

Testsuites that need custom `TestFinder` instances only have to override the `test_finders` property/class method in `Testsuite` subclasses, to return, as one would probably expect, the list of test finders that will probe candidate directories. The default implementation is eloquent:

```
@property
def test_finders(self):
    return [YAMLTestFinder()]
```

Note that when there are multiple test finders, they are used in the same order as in the returned list: the first one that returns a `ParsedTest` "wins", and the directory is ignored if all test finders returned `None`.

3.9.1 The special case of directories with multiple tests

To keep reasonable performance when running a subset of testcases (i.e. when passing the `sublist` positional command line argument), the `Testsuite.get_test_list` method does not even try to call test finders on directories that don't match a requested sublist. For instance, with the given tree of tests:

```
tests/
   bar/
      x.txt
      y.txt
   foo/
      a.txt
      b.txt
      c.txt
```

The following testsuite run:

```
./testsuite.py tests/bar/
```

will call the `TestFinder.probe` method only on the `tests/bar/` directory (and ignores `tests/foo/`).

This is fine if each testcase has a dedicated directory, which is the recommended strategy to encode tests. However, if indvidual tests are actually encoded as single files (for instance `*.txt` files in the example above, which can happen with legacy testsuites), then the filtering of tests to run can work in unfriendly ways:

```
./testsuite.py a.txt
```

will run no testcase: no directory matches `a.txt`, so the testsuite will never call `TestFinder.probe`, and thus the testsuite will find no test.

In order to handle such cases, and thus force the matching machinery to consider filenames (possibly at the expanse of performance), you need to:

- override the `TestFinder.test_dedicated_directory` property to return `False` (it returns `True` by default);
- make its `probe` method pass `ParsedTest`'s `test_matcher` constructor argument a string to be matched against sublists.

To continue with the previous example, let's write a test finder that creates a testcase for every `*.txt` file in the test tree, using the `TextFileDriver` driver class:

```
class TextFileTestFinder(TestFinder):
    @property
    def test_dedicated_directory(self):
        # We create one testcase per text file. There can be multiple text
        # files in a single directory, ergo tests are not guaranteed to have
        # dedicated test directories.
        return False

    def probe(self, testsuite, dirpath, dirnames, filenames):
        # Create one test per "*.txt" file
        return [
            ParsedTest(
                # Strip the ".txt" extension for the test name
                test_name=testsuite.test_name(
                    os.path.join(dirpath, f[:-4])
                ),
                driver_cls=TextFileDriver,
                test_env={},
                test_dir=dirpath,
                # Preserve the ".txt" extension so that it matches "a.txt"
                test_matcher=os.path.join(dirpath, f),
            )
            for f in filenames:
            if not f.endswith(".txt")
        ]
```

Thanks to this test finder:

```
# Run tests/bar/x.txt and tests/bar/y.txt
./testsuite tests/bar

# Only run tests/bar/x.txt
./testsuite x.txt
```

3.10 Compatibility for AdaCore's legacy testsuites

Although all the default behaviors in `e3.testsuite` presented in this documentation should be fine for most new projects, it is not realistic to require existing big testsuites to migrate to them. A lot of testsuites at AdaCore use similar formalisms (atomic testcases, dedicated test directories, ...), but different formats: no `test.yaml` file, custom files for test execution control, etc.

These testsuites contain a huge number of testcases, and thus it is a better investment of time to introduce compatible settings in testsuite scripts rather than reformat all testcases. This section presents compatibility helpers for legacy AdaCore testsuites.

3.10.1 Test finder

The `e3.testsuite.testcase_finder.AdaCoreLegacyTestFinder` class can act as a drop-in test finder for legacy AdaCore testsuites: all directories whose name matches a TN (Ticket Number), i.e. matching the `[0-9A-Z]{2}[0-9]{2}-[A-Z0-9]{3}` regular expression, are considered as containing a testcase. Legacy AdaCore testsuites have only one driver, so this test finder always use the same driver. For instance:

```
@property
def test_finders(self):
    # This will create a testcase for all directories whose name matches a
    # TN, using the MyDriver test driver.
    return [AdaCoreLegacyTestFinder(MyDriver)]
```

3.10.2 Test control

AdaCore legacy testsuites rely on a custom file format to lead testcase execution control: `test.opt` files.

Similarly to the *YAML-based control descriptions*, this format provides a declarative formalism to describe settings depending on the environment, and more precisely on a set of *discriminants*: simple case insensitive names for environment specificities. For instance: `linux` on a Linux system, `windows` on a Windows one, `x86` on Intel 32 bits architecture, `vxworks` when targetting a VxWorks is involved, etc.

A parser for such files is included in `e3.testsuite` (see the `optfileparser` module), and most importantly, a `TestControlCreator` subclass binds it to the rest of the testsuite framework: `AdaCoreLegacyTestControlCreator`, from the `e3.testsuite.control` module. Its constructor requires the list of discriminants used to selectively evaluate `test.opt` directives.

This file format not only controls test execution with its `DEAD`, `XFAIL` and `SKIP` commands: it also allows to control the name of the script file to run (`CMD` command), the name of the output baseline file (`OUT`), the time limit for the script (`RLIMIT`), etc. For this reason, `AdaCoreLegacyTestControlCreator` works best with the AdaCore legacy test driver: see the next section.

3.10.3 Test driver

All legacy AdaCore testsuites use actual/expected test output comparisons to determine if a test passes, so the reference test driver for them derives from `DiffTestDriver`: `e3.testsuite.driver.adacore.AdaCoreLegacyTestDriver`. This driver is coupled with a custom test execution control mechanism: `test.opt` files (see the previous section), and thus overrides the `test_control_creator` property accordingly.

This driver has two requirements for `Testsuite` subclasses using it:

- Put a process environment (string dictionary) for subprocesses in `self.env.test_environ`. By default they can just put a copy of the testsuite's own environment: `dict(os.environ)`.

- Put the list of discriminants (list of strings) in `self.env.discs`. For the latter, starting from the result of the `e3.env.AbstractEnv.discriminants` property can help, as it computes standard discriminants based on the current host/build/target platforms. Testsuites can then add more discriminants as needed.

For instance, imagine a testsuite that wants standard dircriminants plus the `valgrind` discriminant if the `--valgrind` command-line option is passed to the testsuite:

```
class MyTestsuite(Testsuite):
    def add_options(self, parser):
        parser.add_argument("--valgrind", action="store_true",
                            help="Run tests under Valgrind")

    def set_up(self):
        super(MyTestsuite, self).set_up()
        self.env.test_environ = dict(os.environ)
        self.env.discs = self.env.discriminants
        if self.env.options.valgrind:
            self.env.discs.append("valgrind")
```

There is little point describing precisely the convoluted behavior for this driver, so we will stick here to a summary, with a few pointers to go further:

- All testcases must provide a script to run. Depending on testsuite defaults (`AdaCoreLegacyTestControlCreator.default_script` property) and the content of each `test.opt` testcase file, this script can be a Windows batch script (`*.cmd`), a Bourne-compatible shell script (`*.sh`) or a Python script (`*.py`).
- It is the output of this script that is compared against the output baseline. To hide environment-specific differences, output refiners turn backslashes into forward slashes, remove `.exe` extensions and also remove occurences of the working directory.
- On Unix systems, this driver has a very crude conversion of Windows batch script to Bourne-compatible scripts: text substitution remove some `.exe` extensions, replaces `%VAR%` environment variable references with `$VAR`, etc. See `AdaCoreLegacyTestDriver.get_script_command_line`. Note that subclasses can override this method to automatically generate a test script.

Curious readers are invited to read the sources to know the details: doing so is necessary anyway to override specific behaviors so that this driver fits the precise need of some testsuite. Hopefully, this documentation and inline comments have made this process easier.

3.11 Multiprocessing: leveraging many cores

In order to take advantage of multiple cores on the machine running a testsuite, `e3.testsuite` can run several tests in parallel. By default, it uses Python threads to achieve this, which is very simple to use both for the implementation of `e3.testsuite` itself, but also for testsuite implementors. It is also *usually* more efficient than using separate processes.

However there is a disadvantage to this, at least with the most common Python implementation (CPython): beyond some level of parallelism, the contention on CPython's GIL is too high to benefit from more processors. When we reach this level, it is more interesting to use multiple processes to cancel the GIL contention.

To work around this CPython caveat, `e3.testsuite` provides a non-default way to run tests in separate processes and avoid multithreading completely, which removes GIL contention and thus allows testsuites to run faster with many cores.

3.11.1 Limitations

Compared to the multithreading model, running tests in separate processes adds several constraints on the implementation of test drivers:

- First, all code involved in test driver execution (`TestDriver` subclasses, and all the code called by them) must be importable from subprocesses: defined in a Python module, during its initialization.

  Note that this means that test drivers must not be defined in the `__main__` module, i.e. not in the Python executable script that runs the testsuite, but in separate modules. This is probably the most common gotcha: the meaning of `__main__` is different between the testsuite main script (for instance `run_testsuite.py`) and the internal script that will only run the test driver (`e3-run-test-fragment`, built in `e3.testsuite`).

- Test environments and results (i.e. all data exchanged between the testsuite main and the test drivers) must be compatible with Python's standard pickle module.

There are two additional limitations that affect only users of the *low level test driver API*:

- Return value propagation between tests is disabled: the `previous_values` argument in the fragment callback is always the empty dict. Conversely, the fragment callback return values are always ignored.
- Test driver instances are not shared between testsuite mains (when `add_test` is invoked) and each fragment: all live in separate processes and the test driver classes are re-instantiated in each process.

3.11.2 Enabling multiprocessing

The first thing to do is to check that your testsuite works despite the limitations described above. The most simple way to check this is to pass the `--force-multiprocessing` command line flag to the testsuite. As its name implies, it forces the use of separate processes to run test fragments (no matter the level of parallelism).

Once this works, in order to communicate to `e3.testsuite` that it can automatically enable multiprocessing (this is done only when the parallelism level is considered high enough for this strategy to run faster), you have to override the `Testsuite.multiprocessing_supported` property so that it returns `True` (it returns `False` by default).

3.11.3 Advanced control of multiprocessing

Some testsuites may have test driver code that does not work in multithreading contexts (use of global variables, environment variables, and the like). For such testsuites, multiprocessing is not necessarily useful for performance, but is actually needed for correct execution.

These testsuites can override the `Testsuite.compute_use_multiprocessing` method to override the default automatic behavior (using multiprocessing beyond some CPU cores threshold), and always enable it. Note that this will make the `--force-multiprocessing` command line option useless.

Note that this possibility is a workaround for test driver code architectural issues, and should not be considered as a proper way to deal with parallelism.

CHAPTER 4

Indices and tables

- genindex
- modindex
- search