Bootstrapping Boost.Test using a macro DSL

guy@waftex.com
@guyboltonking

Slightly inaccurate title, but the truthful one sounds silly

Bootstrapping your Boost.Test tests using a macro DSL

guy@waftex.com
@guyboltonking

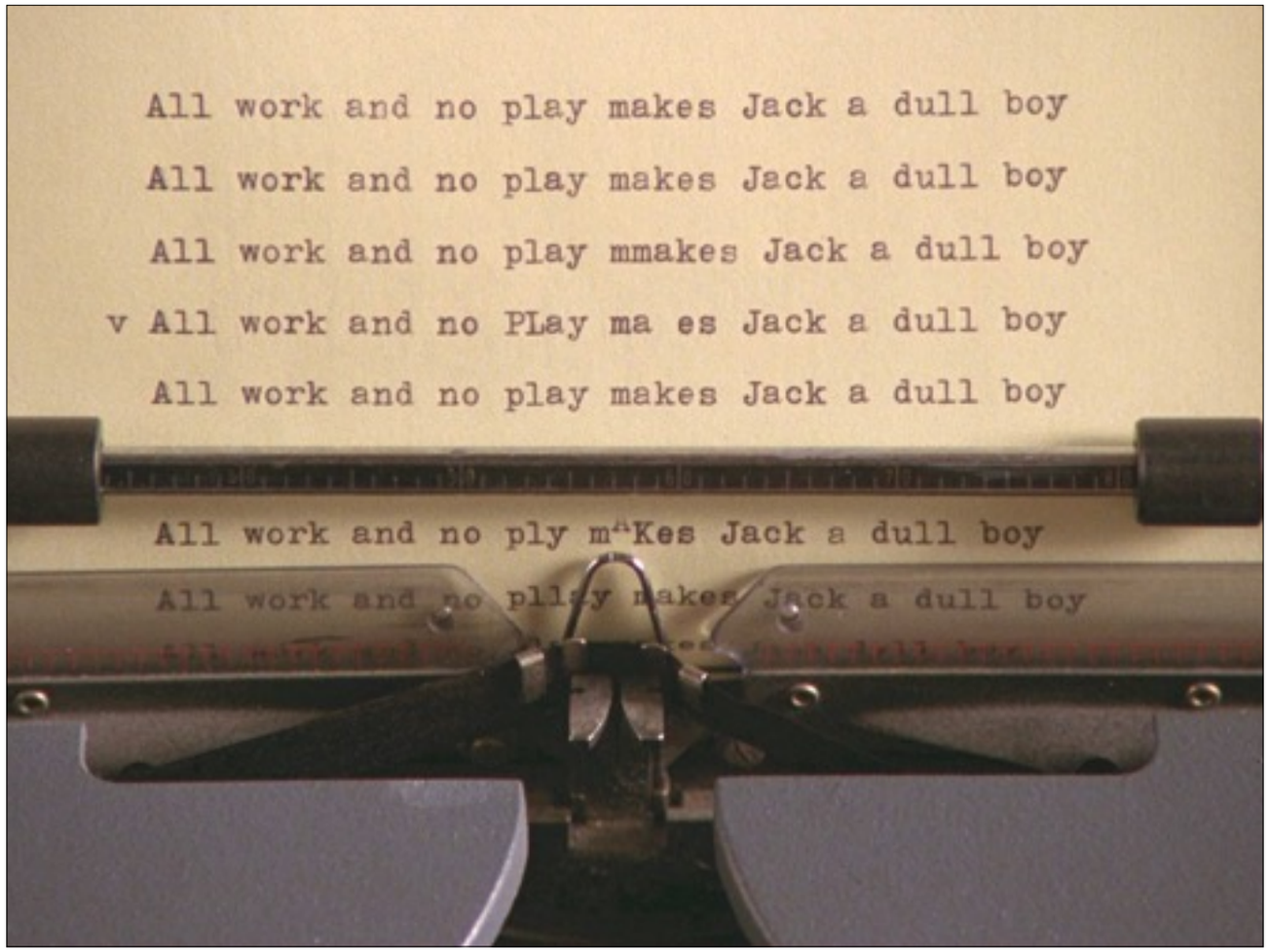
How much wood would a woodchuck chuck etc.

That is, however, what I’m going to talk about; the presumed constraint is that you’re using Boost.Test.

This page intentionally left blank

The blank page

If you start coding tests without thinking about what you're doing you'll end up writing tests you don't need leading to functionality you don't need and then...



madness
So: how do we get started?
The language of writing BDD acceptance tests can help us here.

```
Scenario: popping a non empty stack

  Given  a stack with one member
  When  the stack is popped
  Then  the stack becomes empty
  And  the stack size is reduced by 1
```

Here’s an example for a simple stack.
We’re going to ignore the argument that doing BDD without a domain expert is not useful; the language helps us to focus on what we’re trying to do.
We can turn that into code with a scattering of syntax:

```
Scenario_(popping_a_non_empty_stack)

  Given_(a_stack_with_one_member)
  When_(the_stack_is_popped)
  Then_(the_stack_becomes_empty) {}
  And_(the_stack_size_is_reduced_by_1) {}

  End_
  End_

End_
```

This code compiles and runs! You'll get some warnings about not checking any assertions, but it _does_ work. This is good; we've got ourselves to point where we have to fill in the blanks with the stuff that fulfils our statements.

```
struct before_each {
  Stack<int> stack;
};

Scenario_(popping_a_non_empty_stack)

  Given_(a_stack_with_one_member)
    Before_each_(stack.push(1))

    When_(the_stack_is_popped)
      struct before_each: outer_before_each {
        std::size_t size_before_pop;
        before_each(): size_before_pop(stack.size()) {
          stack.pop();
        }
      };
      Then_(the_stack_becomes_empty)
      {
        BOOST_CHECK(stack.empty());
      }
      And_(the_stack_size_is_reduced_by_1)
      {
        BOOST_CHECK_EQUAL(stack.size(), size_before_pop - 1);
      }
    End_
  End_

End_
```

And this is it.
[click] You can still see our original test description; we've just filled in the blanks
What's in those blanks?
before_each is a fixture. When writing tests in this style, _all_ fixtures are called before_each, and each suite and test implicitly uses a fixture called before_each. If you don't define it, an empty one will be used.
Before_each_ is shorthand for declaring a fixture with no member data, with the argument as the contents of the ctor.
If you declare before_each inside a suite, inherit from outer_before_each so you get the outer fixture (this isn't wrapped in a macro because it's a) important to be aware that we have a struct here, and b) it allows us to trivially mix-in other helper classes)
And then here are our tests.

```
struct before_each {
  Stack<int> stack;
};

Scenario_(popping_a_non_empty_stack)

  Given_(a_stack_with_one_member)
    Before_each_(stack.push(1))

    When_(the_stack_is_popped)
      struct before_each: outer_before_each {
        std::size_t size_before_pop;
        before_each(): size_before_pop(stack.size()) {
          stack.pop();
        }
      };
      Then_(the_stack_becomes_empty)
      {
        BOOST_CHECK(stack.empty());
      }
      And_(the_stack_size_is_reduced_by_1)
      {
        BOOST_CHECK_EQUAL(stack.size(), size_before_pop - 1);
      }
    End_
  End_

End_
```

And this is it.
[click] You can still see our original test description; we've just filled in the blanks
What's in those blanks?
before_each is a fixture. When writing tests in this style, _all_ fixtures are called before_each, and each suite and test implicitly uses a fixture called before_each. If you don't define it, an empty one will be used.
Before_each_ is shorthand for declaring a fixture with no member data, with the argument as the contents of the ctor.
If you declare before_each inside a suite, inherit from outer_before_each so you get the outer fixture (this isn't wrapped in a macro because it's a) important to be aware that we have a struct here, and b) it allows us to trivially mix-in other helper classes)
And then here are our tests.

```
Entering test suite "SCENARIO_popping_a_non_empty_stack"
Entering test suite "given_a_stack_with_one_member"
Entering test suite "when_the_stack_is_popped"
Entering test case "then_the_stack_becomes_empty"
Leaving test case "then_the_stack_becomes_empty"; testing time: 76mks
Entering test case "and_the_stack_size_is_reduced_by_1"
Leaving test case "and_the_stack_size_is_reduced_by_1"; testing time: 79mks
Leaving test suite "when_the_stack_is_popped"
Leaving test suite "given_a_stack_with_one_member"
Leaving test suite "SCENARIO_popping_a_non_empty_stack"
```

The default boost test output is uninspiring; let’s see what we can do about that.

```
SCENARIO popping a non empty stack
   given a stack with one member
     when the stack is popped
       then the stack becomes empty
       and the stack size is reduced by 1
```

So here we've removed the noise and are showing the test hierarchy

Implementation

Boost.Test Building Blocks

```
BOOST_AUTO_TEST_SUITE(suite_name)
   namespace suite_name {

BOOST_AUTO_TEST_SUITE_END()
   }

BOOST_FIXTURE_TEST_CASE(test_name, fixture_class_name)
   struct test_name: public fixture_class_name {
     void test_method();
   };
   void test_name::test_method()
```

Building blocks: Boost.Test has fixtures, suites, and test-cases.
Note that I'm ignoring the test registration machinery: it's not relevant to what we're attempting.
A suite is just a namespace
A test-case is a class, with a single method that contains your code. It inherits from the supplied fixture, which means we get setup/teardown using RAII.

before_each

```
#define GWT_TEST_SUITE(test) \
  BOOST_AUTO_TEST_SUITE(test); \
  typedef before_each outer_before_each;

#define GWT_TEST_CASE(test) \
  BOOST_FIXTURE_TEST_CASE(test, before_each)
```

These two macros allow us to call all our fixtures before_each, and allow chaining them to fixtures in outer scope

Given/When/Then

```
#define Scenario_(x) GWT_TEST_SUITE(SCENARIO_ ## x)

#define Given_(x) GWT_TEST_SUITE(given_ ## x)
#define When_(x) GWT_TEST_SUITE(when_ ## x)

#define Then_(x) GWT_TEST_CASE(then_ ## x)
#define And_(x) GWT_TEST_CASE(and_ ## x)

#define End_ BOOST_AUTO_TEST_SUITE_END()
```

We can now define Given/When/Then, and we’re done.

Bells and Whistles

- Logger
  - Clearer formatting
  - Print filter string for failing tests
- Spec-style testing

```
#define Describe_(x) GWT_TEST_SUITE(x ## _)
#define Context_(x) GWT_TEST_SUITE(x)
#define It_(x) GWT_TEST_CASE(x)
```

The Logger is much more code than the actual macros; see the repo on github
Describe has a trailing _ to allow us to say Describe(class-name) without clashing with the class-name

Questions?

Questions?

- Why not use Catch?
  - Because reasons

github.com/guyboltonking/bdd-with-boost-test

guy@waftex.com
@guyboltonking