igloo - BDD Style Unit Testing for C++

Igloo test applications are created as command line executables that either succeeds with a return value of zero, or fails with a return value corresponding to the number of failing tests.

Test Application Basics

Your test applicaton should look as follows:

```
#include <igloo/igloo.h>
using namespace igloo;

int main(int argc, const char *argv[])
{
return TestRunner::RunAllTests(argc, argv);
}
```

This will automatically run all registered tests in the application.

Command Line Switches

If you pass argc and argv to TestRunner::RunAllTests(), your test application supports the following command line switches:

```
--version                          Print version of Igloo and exit.
--help                             Print available command line switches.
--output=[default|vs|color|xunit]  Select output format.
    default: Igloo's default output format
    vs:      Visual Studio's output format
    color:   Colored output format
    xunit:   XUnit style output format
```

Contexts

Igloo considers a test application to be an executable specification of the application that’s under test. The test application contains one or more contexts that each describe a state of the application under test and a set of specifications for what should hold true in that context.

```
#include <igloo/igloo.h>
using namespace igloo;

Context(a_newly_started_game)
{
Spec(should_have_an_empty_board)
{
Assert::That(game.Positions(), Has().All().EqualTo(EmptyPosition));
}

Game game;
};
```

Alternatives

Besides Context/Spec there are other syntaxes available out of the box: Describe/It and When/Then. You enable these by including igloo_alt.h instead of igloo.h. If you like to, you can even define your own syntax for describing a test.

```
#include <igloo/igloo_alt.h>
using namespace igloo;

Describe(a_newly_started_game)
{
It(has_an_empty_board)
{
Assert::That(game.Positions(), Has().All().EqualTo(EmptyPosition));
}

Game game;
};

When(a_new_game_is_started)
{
Then(it_should_have_an_empty_board)
{
Assert::That(game.Positions(), Has().All().EqualTo(EmptyPosition));
}

Game game;
};
```

Nested Contexts

Igloo enables you to create nested contexts. The inner context inherits and augments the properties of the outer context. This is a powerful feature that lets you organize your contexts in a way that enables you to create just the right amount of setup for each context.

```
#include <igloo/igloo_alt.h>
using namespace igloo;

Describe(a_newly_started_game)
{
It(has_an_empty_board)
{
Assert::That(game.Positions(), Has().All().EqualTo(EmptyPosition));
}

Describe(player_one_is_selected_to_start)
{
void SetUp()
{
Root().game.Select(PlayerOne);
}

It(is_player_ones_turn)
{
Assert::That(Root().game.NextPlayer(), Equals(PlayerOne));
}
};

Game game;
};
```

Accessing Parent Contexts

A nested context can access members of its parent contexts by using the following methods:

Parent() accesses the direct parent context of the current context.

Root() accesses the root (outermost) context. Most of the time all of the members needed by a set of nested contexts are declared in the root context. No matter how deep your nesting goes, you can access those members by using this method.

Set Up and Tear Down

Igloo creates a new context before each call to a Spec. This ensures that each Spec is executed in a fresh environment. Sometimes you might need to perform additional setup before each call, and additional cleanup after each call. You can do this by overriding the methods SetUp and TearDown in your context.

```
#include <igloo/igloo.h>
using namespace igloo;

Context(NameOfContext)
{
void SetUp() { /* Setup code. Called before each Spec. */ }
void TearDown() { /* Tear down code. Called after each Spec. */ }

// ...
};
```

Context SetUp and TearDown

Sometimes you need to setup an environment that is the same for all specs in a context, and that takes a bit too long to setup before each call. In this case you can use Igloo’s SetUpContext and TearDownContext methods to set up static members used in the specs later on.

```
Context(name_of_context)
{
static void SetUpContext() { /* Called once before any Spec. */ }
static void TearDownContext() { /* Called once after all Specs. */ }

// ...

static MyEnvironment common_stuff;
};
```

Running a Subset of the Tests

Specifying which tests to run

By appending “_Only” to a context or a spec, igloo will only run those contexts and specs that has the “_Only” suffix. If a context is marked as “_Only” it will also run all its nested contexts.

```
Context_Only(name_of_context)
{
// This context will be run

Context(name_of_nested_context)
{
// This context will be run as well
};
};

Context(another_context)
{
Spec_Only(my_spec)
{
// This spec will be run
}

Spec(my_other_spec)
{
// This will not be run.
}
};

Context(yet_another_context)
{
// This context will not be run
};
```

Specifying which tests to skip

You can append “_Skip” to contexts and specs to exclude them from the test runs.

```
Context_Skip(skip_this_context)
{
};

Context(a_context)
{
Spec_Skip(skip_this_spec)
{
}
};
```

Test Listeners

You can listen to events during a test run to perform additional work such as logging, statistics, or additional setup and teardown. You do this by deriving from TestListener:

```
class TestListener
{
public:
// Called when a test run is about to begin.
virtual void TestRunStarting() = 0;

// Called after a test run has been completed.
// The TestResults parameter contains information about the
// test run, and stores collections of all succeeded and failed specs.
virtual void TestRunEnded(const TestResults& results) = 0;

// Called before the specs in a context is about to be called.
// If you’ve added ContextAttributes to your contexts you can read these here.
//
// Context(MyContext)
// {
// ContextAttribute("category", "my category")
//
// // ...
// };
//
// void ContextRunStarting(const ContextBase& context)
// {
// const std::string& category = context.GetAttribute("context");
//
// // ...
// }
virtual void ContextRunStarting(const ContextBase& context) =
0;

// Called after all specs have been called for a context.
virtual void ContextRunEnded(const ContextBase& context) = 0;

// Called before a spec is about to be executed.
virtual void SpecRunStarting(const ContextBase& context, const std::string& specName) = 0;

// Called after a spec has been successfully executed.
virtual void SpecSucceeded(const ContextBase& context, const std::string& specName) = 0;

// Called after a spec has failed.
virtual void SpecFailed(const ContextBase& context, const std::string& specName) = 0;
};
```

Registering a TestListener

The following version of main creates and registers a test listener with Igloo:

```
int main()
{
DefaultTestResultsOutput output;
TestRunner runner(output);

MyTestListener listener;
runner.AddListener(&listener);

runner.Run();
}
```

Igloo is maintained by joakimkarlsson | Twitter: @jhkarlsson