Pragmatic Unit Testing in C & C++

by Matt Hargett
<plaztiksyke@gmail.com>

Summary

- The only reason to do unit testing is for sustainable, competitive business advantage
- Unit testing is the most reliable route to a modular/OO design, in my experience
- Unit testing is best used in conjunction with automated integration and system testing
- Focusing on ease of consumption and maintenance of unit tests is key
  - C/C++ is not a valid excuse for messy syntax

Agenda

- What unit testing is/is not
- Breaking dependencies
  - Compile time
  - Link time
  - Preprocessor time
- Writing unit tests to be read
- Mocking
  - Link time
  - Runtime
- Performance

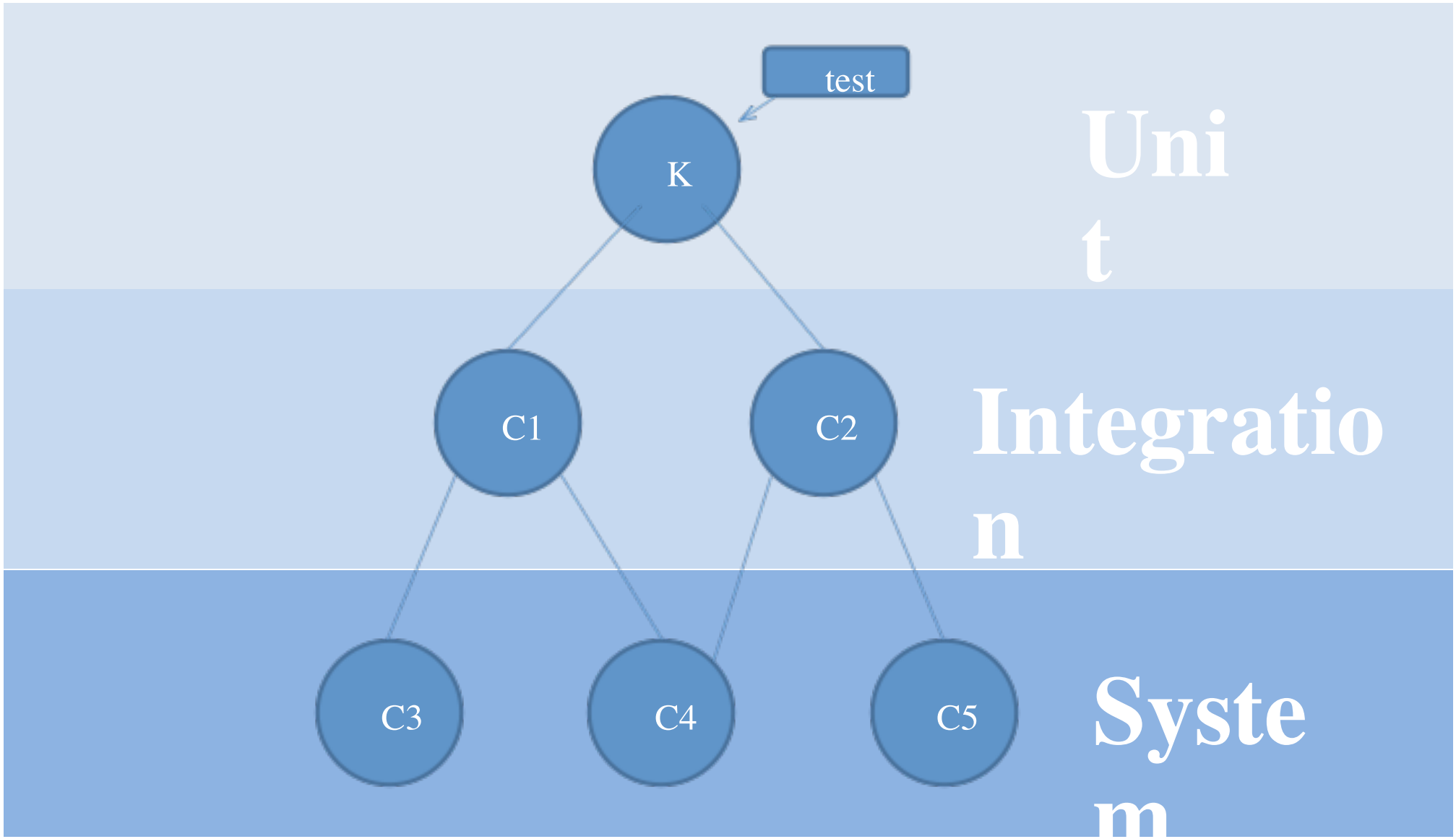
What unit testing is

- Worth mentioning, because the concept has gotten fuzzier over time
  - "a method by which individual units of source code are tested to determine if they are fit for use"
- Modularity is a requirement for real unit tests
  - "the degree to which a system's components may be separated and recombined"
  - Separating interface from implementation
- Testing things that could possibly break
- Fast feedback that you can get while coding

What unit testing is not

- Producing a test binary that can only run in a special environment
- Tests that run in developer environment, but link against the entire system
- An excuse to treat test code differently
  - Keep duplication low
- An excuse to roll your own framework
- An academic distraction of exercising all inputs

Types of developer testing



Breaking Dependencies

- Necessary to mock/stub collaborators
- Mocking collaborators reduces surface area to read/debug when a test fails
- Introducing seams increases modularity
- Which can make it easier to add features, do rapid experiments, and generally innovate

Breaking Dependencies: Compile Time

- Problem: Collaborating functions are in a header file
- Solution: Move them into a source file for link-time substitutability
- Solution: Create a mock header file and put its path on the front of your unit test compile

Breaking Dependencies: Move body

- Move target function/method bodies from header files to source files
  - Make a new file and add to build, if need be
  - Speeds up compile, can reduce transitive includes
  - Keeps interface and implementation separate
- What about inline functions?
- What about template functions?

Move Body: Template functions

```cpp
template<typename T>
void parse(T reader) {
  reader.read();
}

struct K { void read(); };
struct L { void read(); };
struct M { void parallel_read(); };

int main(void) {
  K k;
  L l;
  M m;
  parse(k);
  parse(l);
  parse(m); // error
}
```

```cpp
struct Readable {
  virtual void read(void) = 0;
};

void parse(Readable& reader) {
  reader.read();
}

struct K : public Readable { virtual void read(); };
struct L : public Readable { virtual void read(); };
struct M { void parallel_read(); };

int main(void) {
  K k;
  L l;
  M m;
  parse(k);
  parse(l);
  parse(m); // error
}
```

- Sometimes templates are implicit interfaces
- Or, Extract problematic part to separate class

Mock header

- A third-party header contains a non-trivial collaborating class/function
- mkdir ~/src/mock_include
- cp ~/src/include/trouble.h
- Gut the trouble.h copy and stub/mock only the body you need
- Put –Imock_include on the front of your unit test build command

Breaking Dependencies: Link time

- Link only against the object files your unit test needs
  - The link errors will tell you what to do
  - Aggressively mock non-trivial collaborators
- If transitive dependencies starts to balloon
  - Aggressively mock non-trivial collaborators
- What about static ctors?

Breaking Dependencies: Preprocessor

- Problem: A third-party header calls a non-trivial collaborating function
- Solution: Override non-trivial function calls by defining the symbols in your unit test build

Breaking Dependencies: Redefine symbols

- Redefine the symbol on the unit test build commandline
  - -Dtroublesome_func=mock_troublesome_func
- Then supply a link-time mock

Optimize for readability

- Tests are executable documentation
- They will be read many more times than they are written
- Favor fluent syntax that reads left-to-right
- Passing tests should have very little output
- Failing tests should provide good clues to avoid context switches to debugger/editor

Readability Example: cgreen

- http://cgreen.sf.net
- Cross-language C/C++
- No weird code generators – pure C/C++

```
#include "stream.h"
#include <cgreen/cgreen.h>

Describe(TcpParser);

BeforeEach(TcpParser) {}
AfterEach(TcpParser) {}

Ensure(TcpParser, rejects_null_stream) {
  TcpParser parser;

  assert_that(parser_read(parser, NULL), is_null);
  assert_that(parser_count(parser), is_equal_to(0));
}

$ ./cgreen-runner tcp_parser_test.so

TcpParser -> rejects_null_stream

Expected [parser_read(parser, NULL)] to [be null]

Expected [parser_count(parser)] to [equal] [0]
    Expected: [1]
    Actual:   [0]
```

Mocking: Link time

- Cgreen also supplies a mock framework

```
#include "stream.h"
#include <cgreen/cgreen.h>
#include <cgreen/mocks.h>

using namespace cgreen;

int stream_read(stream* input) {
  return (int)mock(input);
}

Describe(TcpParser);

BeforeEach(TcpParser) {}
AfterEach(TcpParser) {}

Ensure(TcpParser, stops_reading_at_end_of_stream) {
  TcpParser parser;
  const int END_OF_STREAM = -1;

  expect(stream_read,
    when(input, is_non_null),
    will_return(END_OF_STREAM));

  parser.next_chunk();
}

$ ./cgreen-runner tcp_parser_test.so

TcpParser -> stops_reading_at_end_of_stream

Got more calls than expected to [stream_read]
```

Mocking: Runtime

- Function pointers
  - just point to your link-time mocks
- C++ interfaces
  - Mockitopp!

```
#include <cgreen/cgreen.h>
#include <mockitopp/mockitopp.h>

using namespace cgreen;
using namespace mockitopp;
using namespace mockitopp::matcher;

Describe(TcpParser);
BeforeEach(TcpParser) {}
AfterEach(TcpParser) {}

struct Stream { virtual int read(int) = 0; }

Ensure(TcpParser, starts_read_at_zero) {
  mock_object<Stream> mockStream;

  mockStream.expect(&Stream::read).
    when(equal<int>(0)).
    thenReturn(0);

  Stream& stream = mockStream.getInstance();
  TcpParser parser(stream);
  parser.nextChunk(); // calls steam.read()
}
```

Breaking Dependencies: Static class

- Problem: static class makes mocking irritating
- Solution:
  - Make the static methods instance methods
  - Extract a pure virtual base class for methods
  - Make original class a singleton that returns pure virtual base class
  - Update callers
    - %s/Class::MethodName/Class::instance()->MethodName/g

Breaking Dependencies: Static -> Singleton

```
struct Statistics {
  static int average_bps();
  static int average_bps_;
  // 50 more like this. Literally.
}

int main(void) {
  ApplicationProxy proxy;
  proxy.simulate();

  assert(Statistics::average_bps() == 1);
}
```

```
struct GlobalStatistics : public Statistics {
public:
  virtual int average_bps() = 0;
};

struct GlobalStatistics : public Statistics {
public:
  static Statistics& instance() { return *instance_; }
  virtual int average_bps();

private:
  static Statistics* instance_ = new GlobalStatistics();
  int average_bps_;
};

int main(void) {
  mock_object<Statistics> mockStats;
  mockStats.expect(&Statistics::average_bps).
      when().
      thenReturn(1);

  Statistics& stats = GlobalStatistics::instance();
  ApplicationProxy proxy(stats);
  int result = proxy.simulate();

  assert(result == 31337);
}
```

Performance

- PROVE IT WITH A PROFILER OR BENCHMARK
- Improving modularity/testability doesn't have to mean decreased performance
  - Link Time Optimization
    - Optimizes across object file boundaries
  - De-virtualization
    - Tracks concrete types/function pointers and optimizes across type boundaries
  - Profile Guided Optimization
    - Can easily get back your loss

Thanks!

Questions?

Music: http://themakingofthemakingof.com

Music also on iTunes, Amazon, Google Play

Twitter: @syke

Email: plaztiksyke@gmail.com

Links

- Cgreen
  - http://cgreen.sf.net
- Mockitopp
  - http://code.google.com/p/mockitopp/
- Highly recommend tracking their repositories and building from source in your build