C++ NOW 2017

TOWARDS PAINLESS TESTING

Kris Jusiak, Quantlab Financial

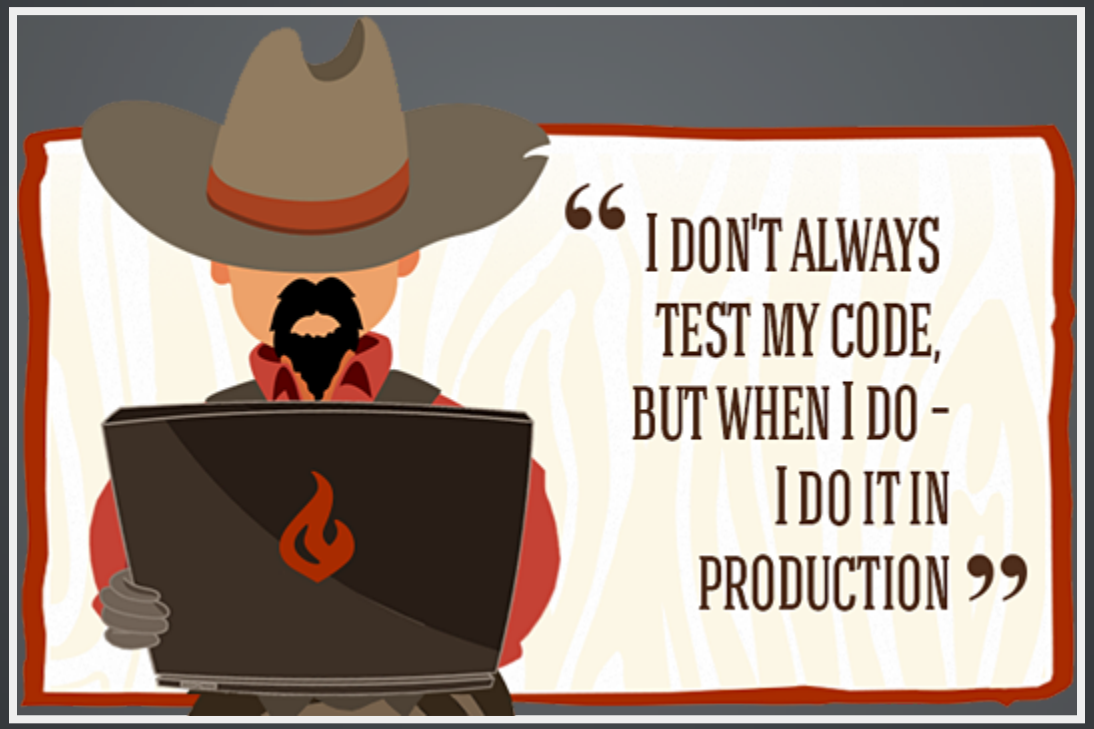
kris@jusiak.net | @krisjusiak | linkedin.com/in/kris-jusiak

"THE ONLY WAY TO GO FAST IS TO GO WELL"
UNCLE BOB

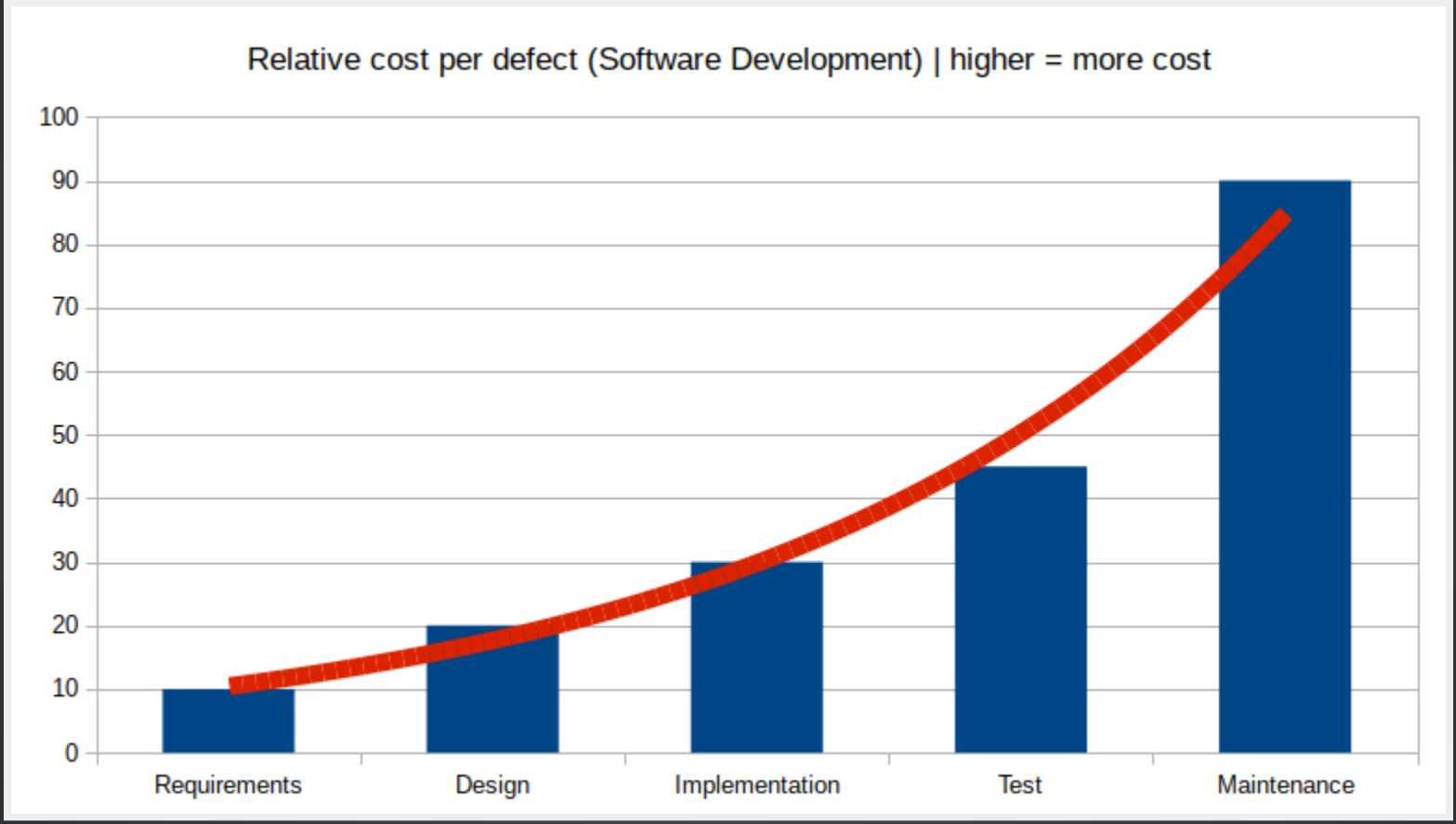
AGENDA

- Testing
  - Why?
  - How And When?
    - Frameworks
    - Mocking
    - Writing A Testable Code
      - Single Responsibility Principle
      - Dependency Inversion Principle
        - Automatic Mocks Injection
      - TDD/BDD
- Showcase
- C++2X...

WHY TESTING IS IMPORTANT?



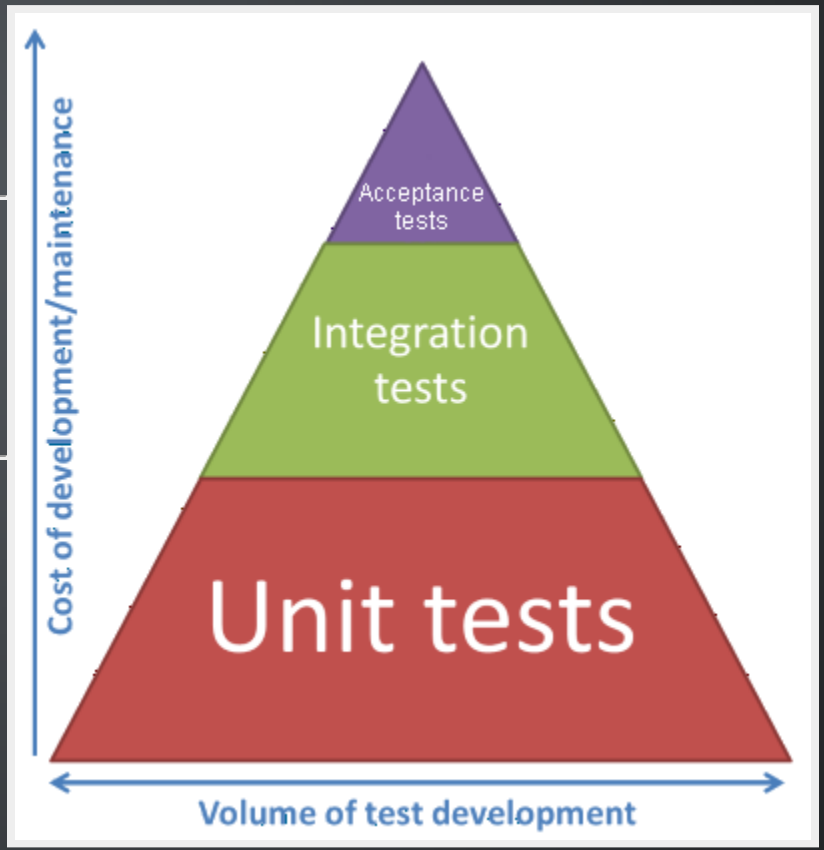
IT'S ALL ABOUT MONEY!



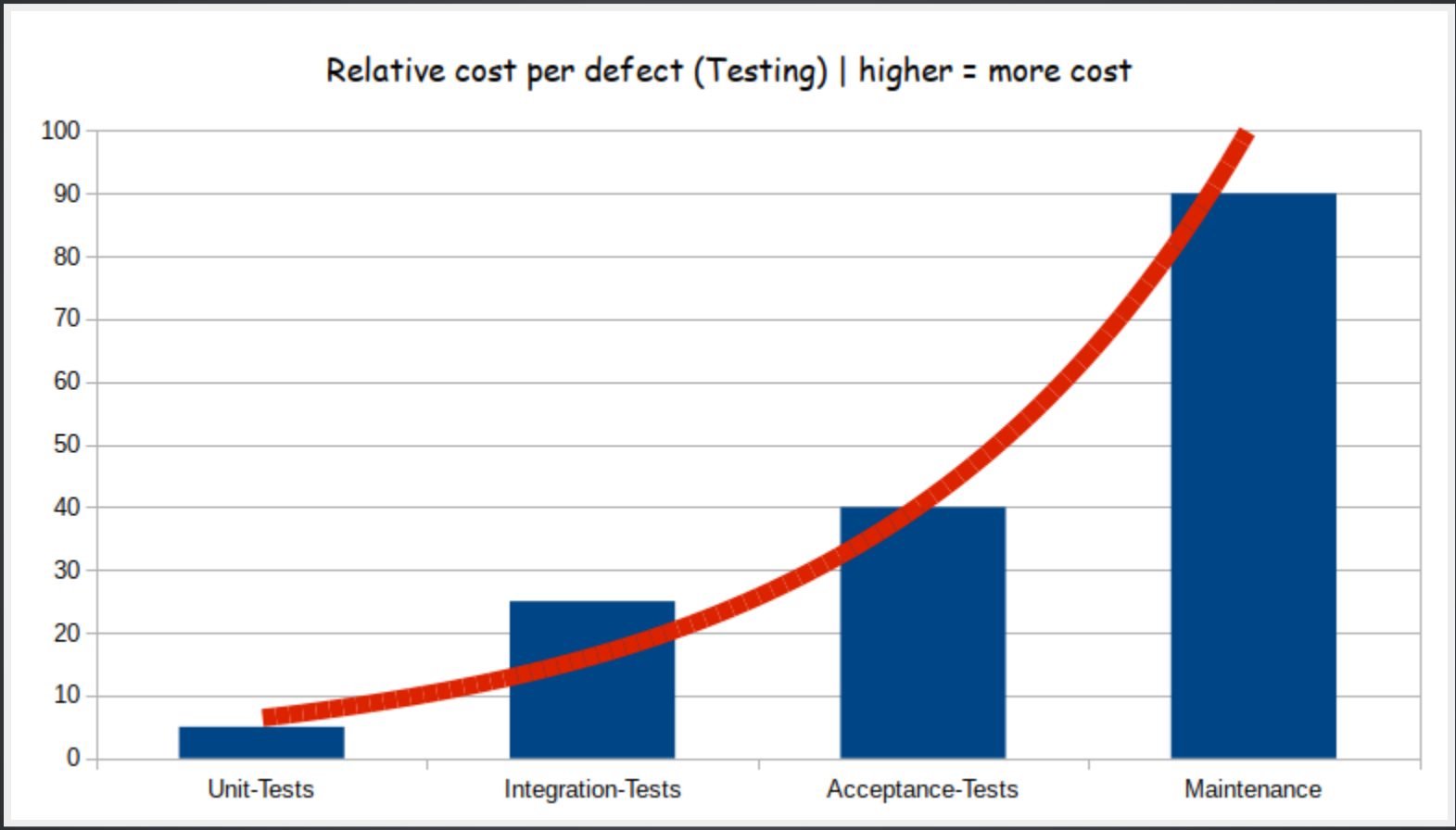
FINDING BUGS IN PRODUCTION IS REALLY EXPENSIVE!

FIRST-CLASS TESTS

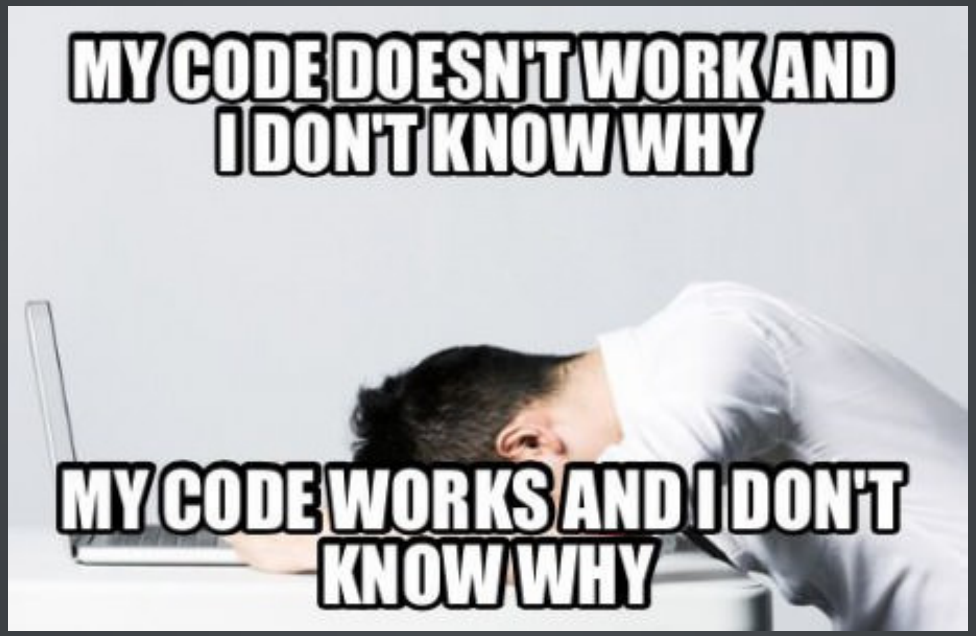
- Acceptance-Tests - 5-10%
  Written by the business
- Integration-Tests - 10-20%
  Written by architects
- Unit-Tests (70-85%)
  Written by programmers



COST PER DEFECT AND FIRST-CLASS TESTS



"DON'T CLING TO A MISTAKE JUST BECAUSE YOU SPENT A LOT OF TIME AND MONEY MAKING IT!"



TEST YOUR CODE AND AVOID DEBUGGING!

TESTING - GOALS

Quality	Testing has to improve the quality of the software
Simplicity	Testing can't be hard
Expressiveness	Tests should express `what` and not `how`
Equality	Test code should be treated as `production code`
Performance	Testable code should not affect runtime performance

To achieve those goals...

CONSIDER USING A GOOD* TESTING FRAMEWORK

GOOD*

- Easy to add and run new tests (automatic test registration)
- Proper assertions system (useful output / support for non-trivial comparisons)
- Feature reach (test suites/fixtures/different outputs)

TESTING IN C++ WITHOUT A FRAMEWORK

```
#include <cassert> // no much testing facilities in the standard

void test_should_add_2_numbers() {
  assert(4 == add(2, 2)); // *no proper asserts
                          // *no nice outputs
}

int main() {
  test_should_add_2_numbers(); // *no automatic test registration
}
```

```
$CXX -g tests.cpp && ./a.out # *no way to specify which tests to run
```

SOLUTION -> C++ TESTING FRAMEWORKS

BOOST.TEST

```
BOOST_AUTO_TEST_CASE(should_add_2_numbers) {
  BOOST_CHECK_EQUAL(4, add(2, 2));
}
```

CATCH

```
TEST_CASE("Numbers can be added", "[add]") {
  // set-up
  SECTION("should add 2 numbers") { REQUIRE(4 == add(2, 2)); }
  // tear-down
}
```

SOLUTION -> C++ TESTING FRAMEWORKS

GOOGLETEST

```
TEST(AddTest, should_add_2_numbers) {
  EXPECT_EQ(4, add(2, 2));
}
```

GOOGLETEST/GUNIT.GTEST

```
GTEST("Can add numbers") { // it works also with GoogleTest
  // set-up                // types `GTEST(AddTests)`
  SHOULD("add 2 numbers") { EXPECT(4 == add(2, 2)); }
  // tear-down
}
```

GUNIT.GTEST-LITE (IT WILL BE USED ON THE SLIDES)

```
"should add 2 numbers"_test { // -gnu-string-literal
  EXPECT(4 == add(2, 2));     // -operator-template
}
```

GUNIT.GTEST-LITE

```
"Calculator"_test = [] {
  Calculator calc{}; // set-up

  SHOULD("add 2 numbers") {        // 1. set-up
    EXPECT(4 == calc.add(2, 2));   // 2. should add 2 numbers
  }                                // 3. tear-down

  SHOULD("sub 2 numbers") {        // 1. set-up
    EXPECT(0 == calc.sub(2, 2));   // 2. should sub 2 numbers
  }                                // 3. tear-down

  // tear-down
};
```

GUnit.GTest	https://github.com/cpp-testing/GUnit#GTest
GUnit.GTest-Lite	https://github.com/cpp-testing/GUnit#GTest-Lite

CONSIDER USING A GOOD* MOCKING FRAMEWORK

GOOD*

- Has to be easier to use than hand written stubs/fakes!
- Limited boilerplate (especially macros)
- Useful error messages in case of unexpected calls
- Support for mocking Interfaces/Templates/Concepts/Type-Erasure

STUBS VS FAKES VS MOCKS

Fake	An object with limited capabilities
Stubs	An object that provides predefined answers to method calls and record calls
Mocks	An object on which you set expectations which are verified by itself

https://martinfowler.com/articles/mocksArentStubs.html

MOCKING INTERFACES - THE STORY SO FAR - GOOGLEMOCK

```
class IReader {
public:
  virtual ~interface() = default;
  virtual int read() const;
};
```

```
/**
 * Boilerplate
 */
class MockReader : public IReader {
public:
  MOCK_CONST_METHOD1(read, int()); // See the bug?
};
```

```
"should read 42"_test = [] {
  MockReader reader{};
  EXPECT_CALL(reader, read()).WillOnce(Return(42));
  EXPECT(42 == reader.read(42));
};
```

MOCKING INTERFACES - MODERN ALTERNATIVES

- Mockator / Eclipse CDT
- HippoMocks
- FakeIt
- ...
- **GUnit.GMock (based on GoogleMock)**

MOCKING INTERFACES - GUNIT.GMOCK

```
"should read 42"_test = [] {
  GMock<IReader> reader{}; // no hand written macros (no boilerplate)
                           // Not a standard solution
                           //   (vtable manipulation involved)

  EXPECT_CALL(reader, read).WillOnce(Return(42));

  EXPECT(42 == reader.read());
};
```

MOCKING INTERFACES - GUNIT.GMOCK - HOW?

```
template <class T>
class GMock { // no inheritance
 static_assert(is_complete<T>{} && is_polymorphic<T>{} &&
               has_virtual_destructor<T>{});

 detail::vtable<T> vtable{};
 detail::byte _[sizeof(T)]{};

public:
 template <class TName, class R, class B, class... TArgs>
 decltype(auto) call_(R (B::*f)(TArgs...), Matcher<TArgs>&&... args);

private:
 flat_map<string, unique_ptr<UntypedFunctionMockerBase>> fs{};
};
```

MOCKING INTERFACES - GUNIT.GMOCK - HOW?

VTABLE

```
/**
 * Itanium C++ ABI - https://mentorembedded.github.io/cxx-abi/abi.html
 * @tparam T interface type
 */
template <class T>
class vtable {
 public:
  vtable(void *f, void *dtor);
  ~vtable();
  void set(std::size_t offset, void *f);

 private:
  void **vptr{};
}
```

USAGE

```
#define EXPECT_CALL(obj, call)
  ((obj).template call_<#call>().
    InternalExpectedAt(__FILE__, __LINE__, #obj, #call)
```

MOCKING TEMPLATES - GUNIT.GMOCK

WE ARE BACK TO SQUARE ONE -> MACROS

```
template<class TReader>
auto read(TReader& reader) { return reader.read(); }
```

```
class MockReader { // no inheritance
public:
  MOCK_CONST_METHOD0(read, int());
};
```

```
"should read 42"_test = [] {
  MockReader reader{};

  // At least we can use the same front-end
  EXPECT_CALL(reader, read).WillOnce(Return(42));

  read(reader);

  EXPECT(42 == reader.read);
};
```

MOCKING CONCEPTS - GUNIT.GMOCK

CONCEPTS LITE - NOT YET :(

```
template <class T>
concept bool Readable =
  CopyConstructible<T> &&
  CopyAssignable<T> &&
  requires(T t) {
    { t.read() -> int }
  }
};
```

BUT WITH CONCEPT EMULATION (C++14) - YES!

```
template <class T>
const auto Readable =
  CopyConstructible<T> &&
  CopyAssignable<T> &&
  Callable<T, int()>($(read)); // expose read for mocking!
```

MOCKING CONCEPTS - GUNIT.GMOCK

VIRTUAL CONCEPTS

Functionality	Description
Constraint checking	Like Concepts-Lite
Type-Erasure	Like Boost.TypeErasure
Mocking	Like GoogleMock just for concepts

https://github.com/boost-experimental/vc

MOCKING CONCEPTS - GUNIT.GMOCK

```
"should read 42"_test = [] {
  GMock<Readable> reader{}; // concept based!

  EXPECT_CALL(reader, read).WillOnce(Return(42));

  EXPECT(42 == reader.read());
};
```

EXACTLY THE SAME AS WITH INTERFACES!

MOCKING CONCEPTS - GUNIT.GMOCK - HOW?

```
Callable<T, int()>($(read)) ]--> Constraint
          \_  \_____    \___-> name
            \       \
$(name) [](auto t, auto r, auto... args) { // expression
 struct { // inherit from
  static auto constraint() {
   return [](const auto &self, decltype(args)... args)
     -> decltype(self.name(args...)){};
  }
  auto name(decltype(args)... args) {
   return static_cast<decltype(t) *>(this) // static polymorphism
     ->template call_<name, typename decltype(r)::type>(args...);
  }
 } _; return _; // local struct
}
```

```
template<class... TConstraints>
class gmock_impl : decltype(std::declval<TConstraints::expression>()(
                              TConstraints::args...))... (
public:
 gmock_impl() = default;

 template <class TName, class R, class... TArgs>
 decltype(auto) call_(TArgs &&... args); // calls mocked impl...
};
```

TYPE-ERASURE (DYNAMIC DISPATCH WITHOUT INHERITANCE)

```
class FileReader { // no inheritance
public:
  explicit FileReader(std::string_view);
  int read();
};

class StreamReader { // no inheritance
public:
  int read();
};
```

ANY - CAN STORE ANYTHING WHICH SATISFIES THE CONCEPT

```
any<Readable> reader = FileReader{"file.txt"};
reader = StreamReader{};
```

MOCKING TYPE-ERASURE

```
"should read 42"_test = [] {
  GMock<Readable> reader{}; // concept based

  EXPECT_CALL(reader, read).WillOnce(Return(42));

  EXPECT(42 == reader.read());
};
```

EXACTLY THE SAME AS WITH INTERFACES AND CONCEPTS!

MOCKING TYPE-ERASURE - HOW?

```
template <class TConcept, class... Ts>
class any_impl : decltype(std::declval<TConstraints::expression>()(
                                TConstraints::args...))... (
public:
  template <class T>
  any_impl(T t) : poly_{
    t, dyno::make_concept_map(Ts::name{} = Ts::expr()...)
  } {}

  template <class TName, class R, class... TArgs>
  auto call_(TArgs&&... args) {
    return poly_.virtual_(TName{})(poly_, args...);
  }

private:
  dyno::poly<Tconcept> poly_{};
};
```

https://github.com/ldionne/dyno

MOCKING REFERENCES

ogleMock	https://github.com/google/googletest
nit.GMock	https://github.com/cpp-testing/GUnit#GMock

- Features
  - No more hand written mocks!
  - Support for mocking concepts/type_erasure
  - Support for more than 10 parameters
  - Support for std::unique_ptr without any tricks
  - Support for overloaded operators
  - Support for mocking classes with constructors
  - 100% Compatible with Google Mocks

CONSIDER WRITING SOLID INSTEAD OF STUPID CODE

- **S** Single Responsibility
- **O** Open-close
- **L** Liskov substitution
- **I** Interface segregation
- **D** Dependency inversion

- **S** ~~Singleton~~
- **T** ~~Tight Coupling~~
- **U** ~~Untestability~~
- **P** ~~Premature Optimization~~
- **I** ~~Indescriptive Naming~~
- **D** ~~Duplication~~

FEATURE: PRINT A VALUE FROM A FILE

KISS - ~~KEEP IT SIMPLE~~, STUPID

```
int main() {
  auto value = 0;
  {
    std::ifstream file{"input.txt"};
    assert(file.good());
    file >> value;
  }
  std::cout << value << '\n';
}
```

UNIT-TESTING? - GIVE ME A BREAK!

A FEW ITERATIONS LATER...

STUPID VS ~~SOLID~~

```
class Manager { // Indescriptive Naming (God object)
public:
  void printValue(int);
  int readValue();
  void update();
  void reset();
  ...
};
```

```
class App {
public:
  App()
    : manager(std::make_unique<Manager>()); // Untestability
  { }                                       // Tight Coupling

  __attribute__((always_inline)) void run() { // Premature Optimization
    Logger::instance() // Singleton
      << "run:" << manager()->readValue() << '\n';

    manager->printValue(
      manager->readValue() // Duplication
    )
  }
private:
  std::unique_ptr<Manager> manager;
};
```

APP - UNIT-TESTING?

```
"should print read value"_test = [] {
  // given
  App app{};

  // when
  app.run();

  // then
  // Ideas? How to fake manager? #define private public?
};
```

WE CAN ONLY DO INTEGRATION-TESTING OR BLACK BOX TESTING HERE!

A FEW ITERATIONS LATER...

(AFTER SOME SOLID COURSES)

SINGLE RESPONSIBILITY PRINCIPLE

A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE

APP - SINGLE RESPONSIBILITY PRINCIPLE

```
/**
 * Responsibility: Can read a value
 */
class Reader {
 public:
   int read() const;
};
```

```
/**
 * Responsibility: Can print a value
 */
class Printer {
 public:
   void print(int i);
};
```

STUPID VS SOLID - SINGLE RESPONSIBILITY PRINCIPLE

```
class App {
public:
  void run() {
    const auto value = reader.read();
    Logger << "run:" << value << '\n';
    printer.print(value); // Law of Demeter
                          // "Only talk to your immediate friends"
  }

private:
  Reader& reader;
  Printer& printer;
  Logger& logger;
};
```

APP - UNIT-TESTING?

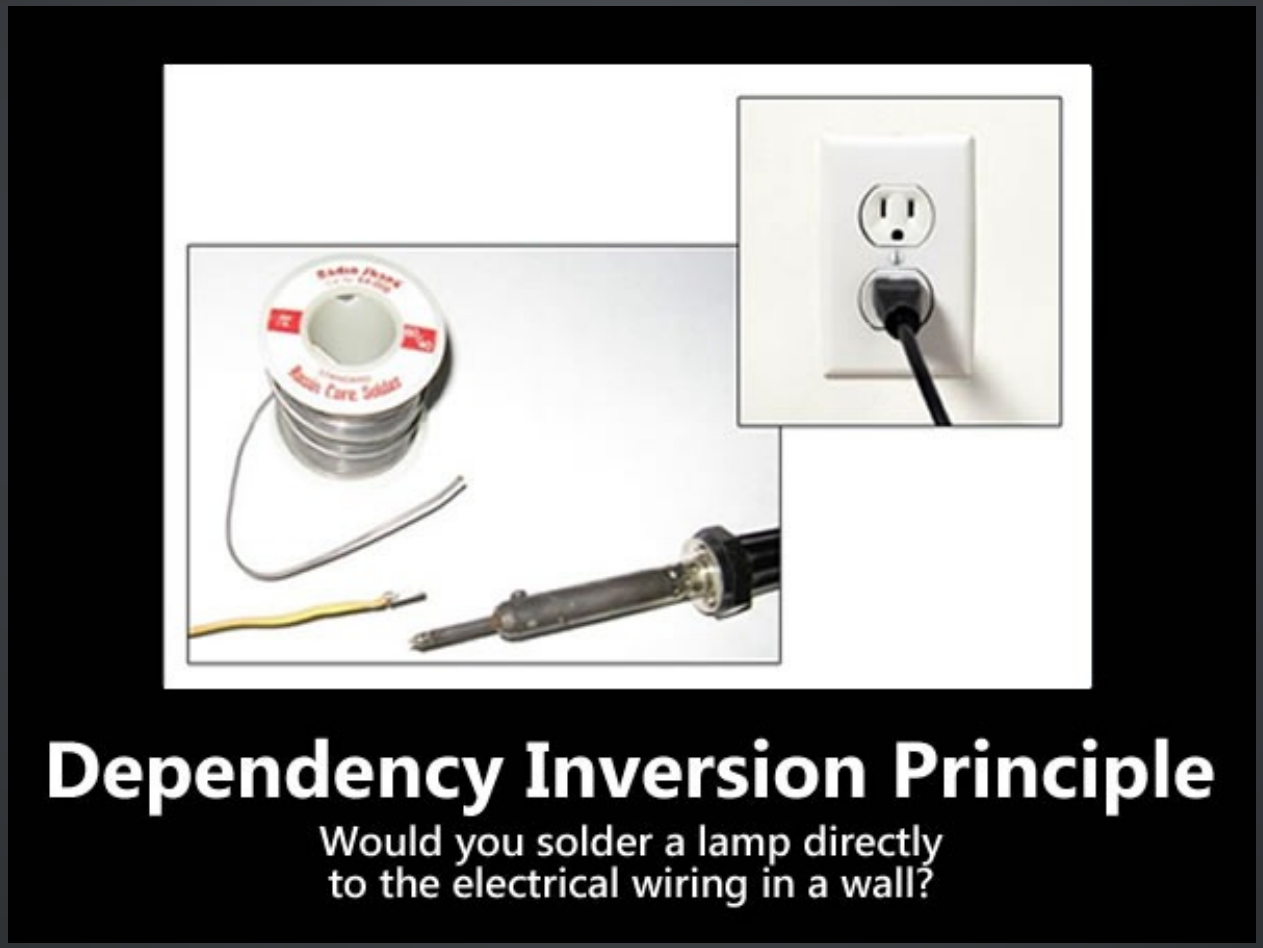
```
"should print read value"_test = [] {
  // given
  App app{};

  // when
  app.run();

  // then
  // ??? still not quite there!
};
```

A FEW ITERATIONS LATER...

(AFTER SOME MORE SOLID COURSES)

DEPENDENCY INVERSION PRINCIPLE

HIGH-LEVEL MODULES SHOULD NOT DEPEND ON LOW-LEVEL MODULES. BOTH SHOULD DEPEND ON ABSTRACTIONS, NOT ON IMPLEMENTATIONS



APP - DEPENDENCY INVERSION PRINCIPLE

```
class IReader {
public:
  virtual ~IReader() = default;
  virtual int read() const = 0;
};
```

```
class IPrinter {
public:
  virtual ~IPrinter() = default;
  virtual void print(int);
};
```

```
class FileReader final
 : public IReader {
public:
  int read() const override;
};
```

```
class ConsolePrinter final
  : public IPrinter {
public:
  void print(int) override;
};
```

WE CAN HAVE DIFFERENT IMPLEMENTATIONS OF READERS/PRINTERS/LOGGERS!

~~STUPID~~ VS SOLID - DEPENDENCY INVERSION PRINCIPLE

```
class App {
public:
  /**
   * Dependency Injection
   *   "Don't call us, we'll call you", Hollywood principle
   */
  App(IReader& reader, IPrinter& printer, ILogger& logger);

  void run() {
    const auto value = reader.read();
    Logger << "run:" << value << '\n';
    printer.print(value);
  }

private:
  IReader& reader; IPrinter& printer; ILogger& logger;
};
```

APP - UNIT-TESTING - POSSIBLE BUT WITH SOME BOILERPLATE!

```
"should print read value"_test = [] {
  // Some boilerplate introduced!
  NiceGMock<ILogger> logger{};       // Ignore an uninteresting call
  StrictGMock<IReader> reader{};     // Fail on uninteresting call
  StrictGMock<IPrinter> printer{};   // Fail on uninteresting call
  App app{reader, printer, logger}; // Wiring!

  EXPECT_CALL(reader, read).WillOnce(Return(42));
  EXPECT_CALL(printer, print, 42);

  app.run();
};
```

IT WORKS BUT IT'S JAVA-ISH

- Heap
- Reference/Pointer semantics
- Inheritance
  - "Inheritance Is The Base Class of Evil" Sean Parent
- Dynamic Dispatch
- ~Performance
  - final keyword and devirtualization - https://godbolt.org/g/e8olYN

C++ AIN'T JAVA - ZERO OVERHEAD ABSTRACTIONS, PLEASE!

TEMPLATES/CONCEPTS

APP - CONCEPTS

```
template <class T>
const auto Readable =
  CopyConstructible<T> &&
  CopyAssignable<T> &&
  Callable<T, int()>($(read));
```

```
template <class T>
const auto Printable =
  CopyConstructible<T> &&
  CopyAssignable<T> &&
  Callable<T, void(int)>($(print));
```

```
// template<Readable TReader, Printable TPrinter> // Concepts-Lite
template<class TReader, class TPrinter, class TLogger>
class App {
  static_assert(Readable<TReader>() &&
                Printable<TPrinter>() &&
                Loggable<TLogger>());

public:
  App(TReader& reader, TPrinter& printer, TLogger& logger);

  void run() {
    printer.print(reader.read());
  }

private:
  TReader& reader; TPrinter& printer; TLogger& logger;
};
```

APP - UNIT-TESTING - CONCEPTS

```
"should print read value"_test = [] {
  NiceGMock<Loggable> logger{};       // Ignore an uninteresting call
  StrictGMock<Readable> reader{};     // Fail on uninteresting call
  StrictGMock<Printable> printer{};   // Fail on uninteresting call
  App app{reader, printer, logger};   // C++17 template argument
                                      // deduction for class templates

  EXPECT_CALL(reader, read).WillOnce(Return(42));
  EXPECT_CALL(printer, print, 42);

  app.run();
};
```

PRETTY MUCH THE SAME AS FOR INTERFACES!

BUT WHAT IF DON'T KNOW THE DEPENDENCY AT COMPILE-TIME?

Note, that most dependencies are known at compile time

APP - TYPE-ERASURE - VIRTUAL CONCEPTS (DYNAMIC DISPATCH WITHOUT INHERITANCE)

```
class App {
public:
  App(any<Readable> reader
    , any<Printable> printer
    , any<Loggable> logger);

  void run() {
    printer.print(reader.read());
  }

private:
  any<Readable> reader;   // `virtual Readable` with virtual concepts!
  any<Printable> printer; // `virtual Readable` with virtual concepts!
  any<Loggable> logger;   // `virtual Readable` with virtual concepts!
};
```

APP - UNIT-TESTING - TYPE-ERASURE

```
"should print read value"_test = [] {
  NiceGMock<Loggable> logger{};       // Ignore an uninteresting call
  StrictGMock<Readable> reader{};     // Fail on uninteresting call
  StrictGMock<Printable> printer{};   // Fail on uninteresting call
  App app{reader, printer, logger};

  EXPECT_CALL(reader, read).WillOnce(Return(42));
  EXPECT_CALL(printer, print, 42);

  app.run();
};
```

THE SAME TEST AS WITH CONCEPTS!

BUT WHAT ABOUT ALL THIS WIRING/PLUMBING BOILERPLATE INTRODUCED BY `SOLID`?

DI - WIRING MESS

TEST

```
"should print read value"_test = [] {
  NiceGMock<Loggable> logger{};       // WIRING!
  StrictGMock<Readable> reader{};     // WIRING!
  StrictGMock<Printable> printer{};   // WIRING!
  App app{reader, printer, logger};   // WIRING!
};
```

MAIN - COMPOSITION ROOT (UNIQUE LOCATION WHERE MODULES ARE COMPOSED TOGETHER)

```
int main() {
  auto file = std::fstream{"input.txt"};                // WIRING!
  auto reader = std::make_unique<FileReader>(file);     // WIRING!
  auto printer = std::make_shared<Printer>(std::cout); // WIRING!
  App app{reader, printer};                             // WIRING!
}
```

SOLID - PRODUCES TESTABLE CODE AND BETTER DESIGN

BUT

- SINGLE RESPONSIBILITY PRINCIPLE

  =>

  - A LOT OF CLASSES

    =>

    - WIRING MESS

      =>

      - HARD TO MAINTAIN + LAZY PROGRAMMERS (99%)

        =>

        - HACKS/WORKAROUNDS (~~SINGLE RESPONSIBILITY~~)

SOLUTION (WORLD WITHOUT STATIC FACTORIES)

SIMPLIFY/REMOVE THE WIRING MESS

BY AUTOMATING DEPENDENCY INJECTION?

=>

[BOOST].DI

CONSIDER USING DEPENDENCY INJECTION FRAMEWORK TO AVOID THE WIRING MESS AND INJECT MOCKS AUTOMATICALLY

[BOOST].DI - DEPENDENCY INJECTION FRAMEWORK

```
#include <boost/di.hpp>
namespace di = boost::di;
```

```
class App {
 public:
  App(const Reader&, Printer&); //[Boost].DI
  void run();                   // deduces constructors parameters
                                // dedcues required scope for them
 private:                       // create dependencies and inject them
  Reader& reader;
  Printer& printer;
};
```

CREATE APP (OBJECT GRAPH) USING [BOOST].DI

```
int main() {
  auto app = di::make<App>();
  app.run();
}
```

[BOOST].DI - RESOLVE DEPENDENCIES TEMPLATES/CONCEPTS/INTERFACES/TYPE_ERASURE

```
template<class TReader = Readable, class TSize = class Size>
class App {                                                  ^
 public:                                                     |
  App(const TReader&, std::unique_ptr<IPrinter>, config);    |
};           ^                              ^          ^     |
             |                              \-------\  |     |
             \------------------------------\   |   |  |     |
int main() {                                |   |   |  |     |
  const auto injector = di::make_injector(  |   |   |  |     |
     di::bind<Readable>.to<FileReader>(),   --/   |   |  |     |
     di::bind<IPrinter>.to<ConsolePrinter>(), ----/   |  |     |
     config{"127.0.0.1", 8080}, --------------------/  |     |
     di::bind<class Size>.to<int_<42>>() ---------------/
  );

  di::make<App>(injector).run();
}
```

HOW IS THAT HELPING WITH THE WIRING MESS?

[BOOST].DI - AUTOMATIC INJECTION (OBJECT GRAPH)

MANUAL DI - WIRING MESS

```
App::App(const Reader& reader, Printer& printer);
```

```
int main() {
  auto reader = Reader{};           // WIRING!
  auto printer = Printer{};         // WIRING!
  auto app = App{reader, printer}; // WIRING!
  app.run();
}
```

[BOOST].DI

```
int main() {
  di::make<App>().run();
}
```

- [Boost].DI will also boost your performance
  - No run-time overhead (wiring at compile time)
  - Cache friendly object layout (whole graph is known at compile time)

[BOOST].DI - REFACTORING FOR FREE!

LET'S CHANGE `APP` CONSTRUCTOR

BEFORE

```
App(const Reader& reader, Printer& printer)
  : reader(reader))
  , printer(printer)
{ }
```

AFTER

```
App(Printer& printer, std::unique_ptr<Reader> reader)
 : printer(printer)
 , reader(std::move(reader))
{}
```

[BOOST].DI - REFACTORING FOR FREE!

MANUAL DI - WIRING MESS

```
int main() {
  auto reader = std::make_unique<MyReader>{}; // DIFF Wiring!
  auto printer = ConsolePrinter{};            // Wiring
  auto app = App{printer, std::move(reader)}; // DIFF Wiring!
  app.run();
}
```

[BOOST].DI - NO CHANGES!

```
int main() {
  di::make<App>().run(); // same as before!
}
```

[BOOST].DI - INTEGRATION TESTING FOR FREE!

PRODUCTION

```
const auto config = [] {
  return di::make_injector(
    di::bind<Readable>.to<FileReader>(),
    di::bind<Printable>.to<ConsolePrinter>(),
    di::bind<DataBase>.to<SQL>()
  );
};

int main() {
  di::make<App>(config)().run();
}
```

INTEGRATION-TESTING

```
"should"_test = [] {
  auto app = di::make<App>(
    config, // production config
    di::bind<DataBase>.to<FakeDataBase>() [di::override]
  );

  app.run(); // fake data base will be used!
```

[BOOST].DI/GUNIT.GMOCK - AUTOMATIC MOCKS INJECTION

LET'S GET RID OFF SOME BOILERPLATE!

```
GMock<Readable> reader{};     // WIRING!
GMock<Printable> printer{};   // WIRING!
GMock<Loggable> logger{};     // WIRING!
...
```

TEST - MANUAL MOCKS INJECTION

```
"should print read value"_test = [] {
  GMock<Readable> reader{};                        // Boilerplate
  GMock<Printable> printer{};                      // Boilerplate
  App<Readable, Printable> app{reader, printer};   // Boilerplate

  EXPECT_CALL(reader, read).WillOnce(Return(42));
  EXPECT_CALL(printer, print, 42);

  app.run();
};
```

TEST - AUTOMATIC MOCKS INJECTION

```
"should print read text"_test = [] {
  auto [app, mocks] = testing::make<App>();// creates System Under Test
                                           // and mocks

  EXPECT_CALL(mocks<Readable)>(), read).WillOnce(Return(42));
  EXPECT_CALL(mocks<Printable>(), print, 42);

  app.run();
};
```

LET'S CHANGE THE APP A BIT

TEST - MANUAL MOCKS INJECTION (LOC CHANGED: 5/6, LOC ADDED: 1)

```
"should print read value"_test = [] {
  auto reader = std::make_shared<GMock<Readable>>(); // DIFF!
  auto printer = std::make_unique<GMock<Printable>>(); // DIFF!
  auto printer_ptr = printer.get(); // HACK!
  App<Printable, Readable> app{reader, std::move(printer)}; // DIFF!

  EXPECT_CALL(*reader, read).WillOnce(Return(42)); // Dereference!
  EXPECT_CALL(*printer_ptr, print, 42); // HACK!

  app.run();
};
```

TEST - AUTOMATIC MOCKS INJECTION (LOC CHANGED: 0/6, LOC ADDED: 0)

```
"should print read text"_test = [] {
  auto [app, mocks] = testing::make<App>(); // SAME OLD, SAME OLD!

  EXPECT_CALL(mocks<Readable>(), read).WillOnce(Return(42));
  EXPECT_CALL(mocks<Printable>(), print, 42);

  app.run();
};
```

AUTOMATIC MOCKS INJECTION - HOW?

```
template<class TArgs, class TMocks>
struct mocker {
  const TArgs& args;
  TMocks& mocks;

  template<class T> requires std::is_polymorphic<raw_t<T>>{}
  operator T() const {
    return mocks.add<T>(); // create a new GMock<T>
  }

  template<class T> requires !std::is_polymorphic<raw_t<T>>{}
  operator T() const {
    return args.get<T>(); // get the arg
  }
};
```

```
template <class T, class... TMocks>
auto make(TArgs &&... args) {
  std::tuple<TArgs...> args{args...};
  mocks_t mocks{};

  return std::pair{
    T{mocker{args, mocks}...} // depends on sizeof of
  , mocks_t                   // is_constructible<T, mocker...>
  };
}
```

Okay, but isn't that hiding a bad design (too many dependencies)?

SOLUTION

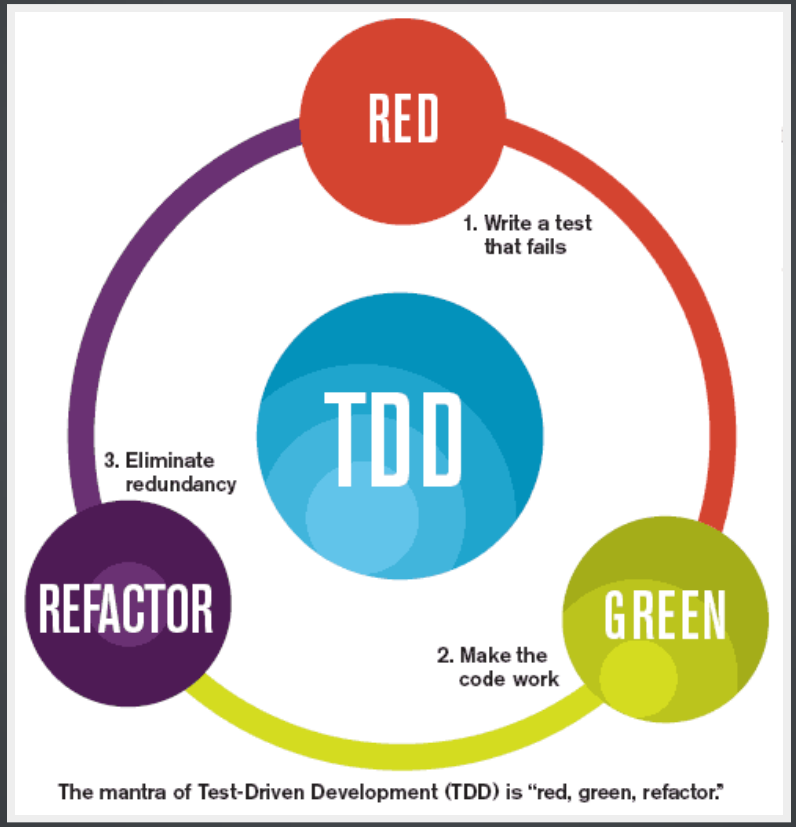
=>

COMBINE IT WITH TEST DRIVEN DEVELOPMENT

CONSIDER WRITING TESTS BEFORE THE IMPLEMENTATION

TDD	Test Driven Development	Unit Tests
BDD	Behaviour Test Driven Development	Integration/Acceptance Tests

TDD MANTRA - RED-GREEN-REFACTOR



The mantra of Test-Driven Development (TDD) is "red, green, refactor."

THE THREE RULES OF TDD

1. You are not allowed to write any production code unless it is to make a failing unit test pass
2. You are not allowed to write any more of a unit test than is sufficient to fail
   - Compilation failures are failures
3. You are not allowed to write any more production code than is sufficient to pass the one failing unit test

TEST DRIVEN DEVELOPMENT (TDD)

- We only implement what we needed (no over engineering)
- There is no untested code (it implies 100% coverage)
- We ALWAYS know that the code works (regression, refactoring)
- Unit tests are runnable documentation of the code (requirements, example usage)
- Loosely coupled code (no globals/singletons/god objects)

Especially useful with `constexpr` algorithms!

BEHAVIOUR DRIVEN DEVELOPMENT (BDD)

LANGUAGE

```
"[TDD] add numbers"_test = [] {
  EXPECT_CALL(sum, add(2, 2)).
    WillOnce(Return(4));
  c.add(2, 2);
  EXPECT(count == 4);
};
```

```
"[BDD} should add 2 numbers" = [] {
  GIVEN(sum, add(2, 2)).
    WillOnce(Return(4));
  WHEN(calc.add(2, 2));
  THEN(count).should_be(4);
};
```

SCOPE

	Test
TDD	Locked Implementation
BDD	Locked Behaviour

BDD (GIVEN/WHEN/THEN) - EXAMPLE

Gherkin/Cucumber

```
Feature: File Viewer

Scenario 1: Value from a file is displayed
 Given I have a file with a 42 value in it
   And the App is created
 When The App runs
 Then The 42 should be printed
```

test/features/file_viewer

BDD - GHERKIN/CUCUMBER (ACCEPTANCE TESTING)

```
GIVEN("^I have a file with a (\\d+) in it$") {
  REGEX_PARAM(int, n);
  ScenarioScope<AppCtx> context{};
  context->app.push(n);
}
```

```
GIVEN("^The App is created$") {
  ScenarioScope<AppCtx> context{};
  std::cout.rdbuf(context->app.buffer.rdbuf()); // redirect cout
  context->app = di::make<App>(production_wiring);
}
```

```
WHEN("^The App runs") {
  ScenarioScope<AppCtx> context{};
  context->app.run();
}
```

```
THEN("^The (\\d+) should be printed") {
  REGEX_PARAM(int, expected);
  ScenarioScope<AppCtx> context{};
  specify(context->buffer, should.equal(expected));
}
```

file_viewer.cpp

BDD - GHERKIN/CUCUMBER

RUN

```
./cucumber test/features/file_viewer
```

```
Feature: File Viewer

Scenario 1: Value from a file is displayed # test/.../file_viewer:12
 Given I have a file with a 42 value in it # file_viewer.cpp:22
   And the App is created                  # file_viewer.cpp:33
 When The App runs                         # file_viewer.cpp:34
 Then The 42 should be printed             # file_viewer.cpp:48
```

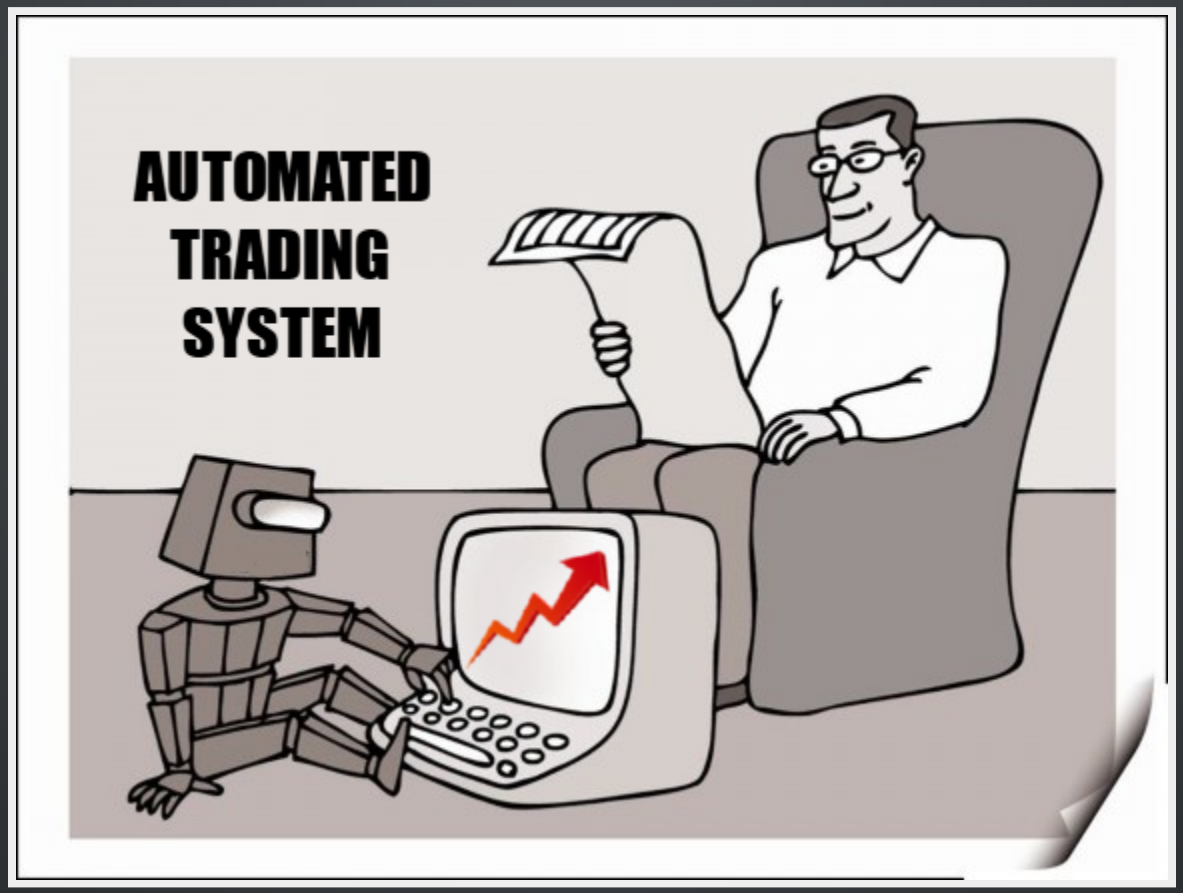
OUTPUT

```
1 scenario (1 passed)
4 steps (4 passed)
0m0.015s
```

BEHAVIOUR DRIVEN DEVELOPMENT

- Complementary with TDD!
- Focus on customer needs (behaviours)

SHOWCASE (USER STORY -> ... -> MERGE REQUEST)



TOOLS

Methodology	Agile (Scrum)
Modeling	UML 2.5
Design	Concepts Driven
Development	BDD/TDD/eXtreme Programming
Delivery	Continuous Integration

PRODUCT BACKLOG REFINEMENT

Priority	Story	Description
1.	Automated Trading System	A trading system should trade stocks automatically
2.	...	...

PRODUCT BACKLOG REFINEMENT - ACCEPTANCE CRITERIA (BDD)

```
Feature: Automated Trading System
```

```
Scenario 1: Trading System requests to buy shares
  Given The Trading System is up and running

  When GOOGL stock is rising (last 1000 transactions)

  Then A buy order for 100 shares of APPL should have been executed
   And The TS should own 100 shares of APPL
```

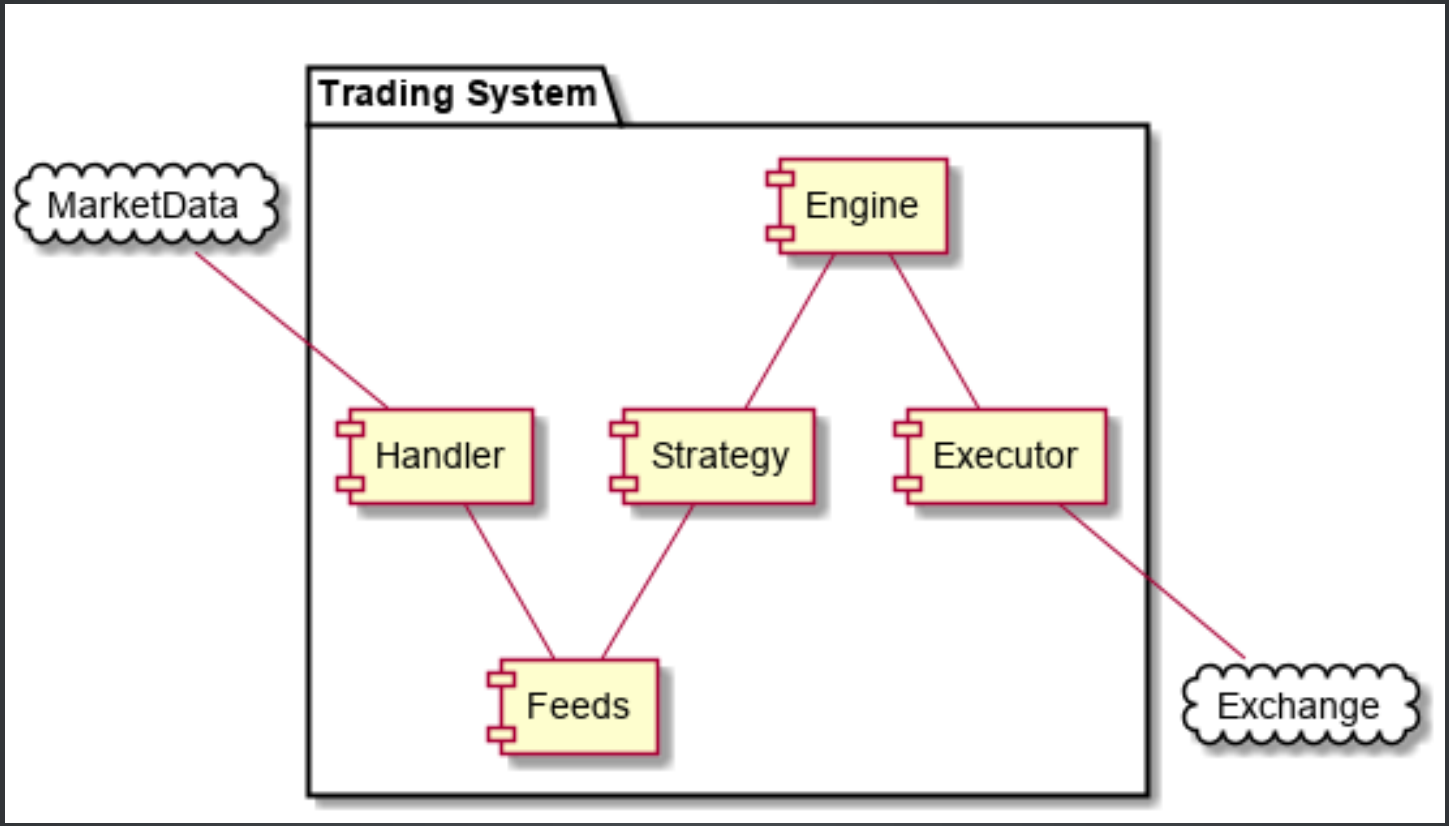
```
Scenario 2: Trading System requests to sell shares
  Given The Trading System is up and running
    And The TS owns 50 shares of APPL stock

  When GOOGL stock is falling (last 1000 transactions)

  Then A sell order for 25 shares of APPL stock should
       have been executed
   And The TS should own 25 shares of APPL stock
```

```
Scenario 3: Trading system requests to hold shares
Scenario 4: Trading system disconnects from the exchange
...
Scenario N: ...
```

IF WE USE CUCUMBER WE ALREADY HAVE ACCEPTANCE TESTS!

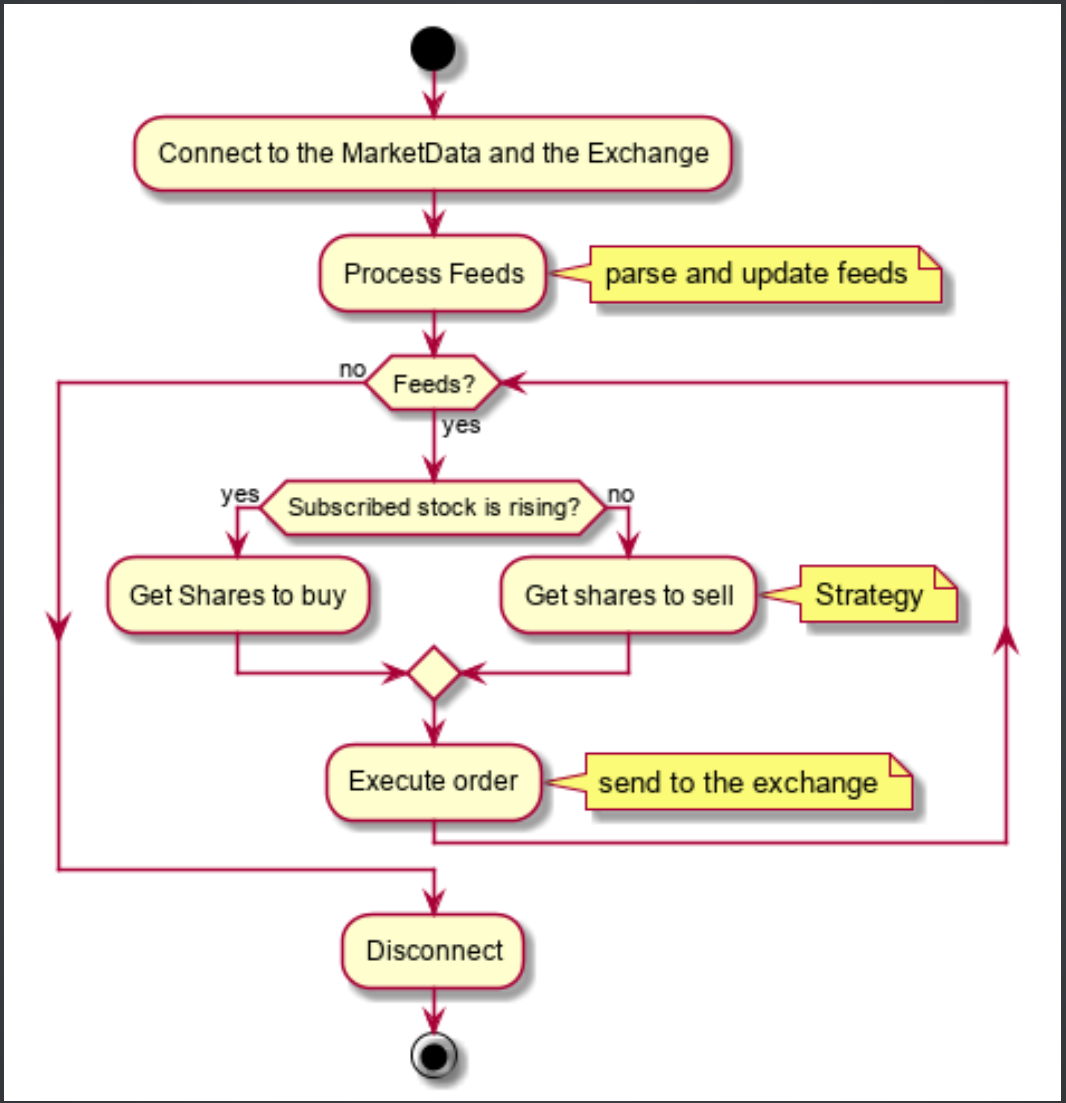
Acceptance Tests are written by the business for the purpose of ensuring that the production code does what the business expects it to do

DESIGN/WORKSHOP - DOMAINS/COMPONENTS

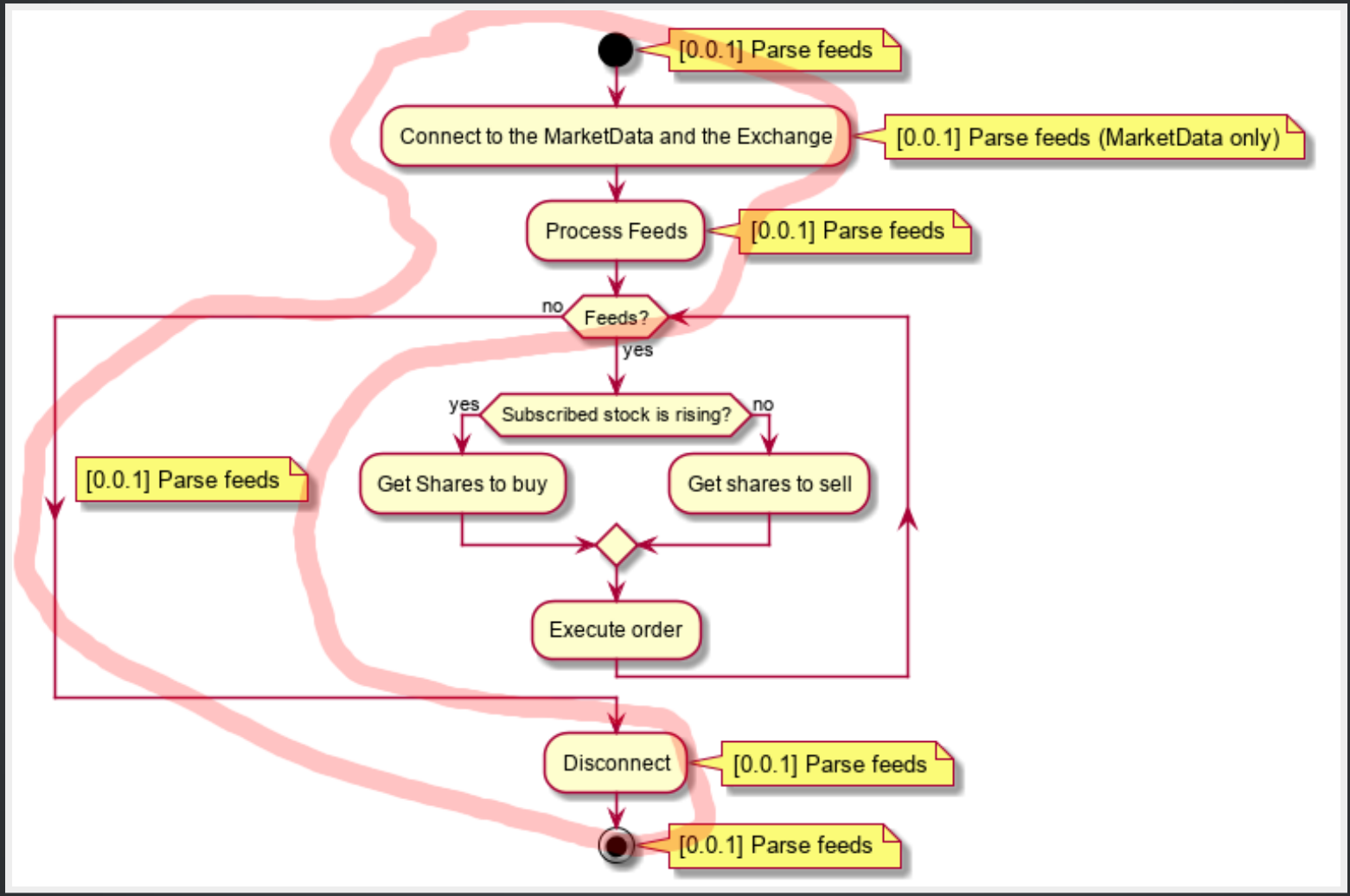


LOOSELY COUPLED COMPONENTS (THE VISION)

DESIGN/WORKSHOP - ACTIVITIES



STORIES -> TASKS (CROSS FUNCTIONAL)



STORIES -> TASKS

PRIORITY ORDER

[0.0.1] Parse feeds (deliverable)

- Connect to the MarketData
- Process feeds
- Feeds? -> no
- Disconnect

[0.0.2] Handle buy (deliverable)

[0.0.3] Handle sell (deliverable)

PLANNING (NO DESIGN/WORKSHOP)

ESTIMATES

- [0.0.1] Parse feeds (3SP)
- [0.0.2] Handle buy (8SP) (~3x more complex than [0.0.1])
- [0.0.3] Handle sell (2SP)

COMMITMENT (TEAM VELOCITY ~12 STORY POINTS)

- [0.0.1] Parse feeds <- COMMIT (3SP)
- [0.0.2] Handle buy <- COMMIT (3SP + 8SP)

Story Point - a number that tells the team how hard the story is

SPRINT - LET'S DO IT!

TODO	WIP	DONE
	[0.0.1] Parse Feeds	
[0.0.2] Handle Buy		
[0.0.3] Handle Sell		

TDD AND EXTREME PROGRAMMING / PAIRING

1. *One dev is writing a test and the other is making it pass (the simplest way)*
2. *Switch the roles!*

TDD/RED - WRITE A BIT OF TEST (EXPECTATIONS/INTENTIONS FIRST)

```
"should connect to the market data on startup"_test = [] {
  auto [ts, mocks] = testing::make<trading_system>();

  EXPECT_CALL(mocks<MarketData>(), connect);

  ts.start();
};
```

TDD/GREEN - MAKE IT COMPILE/PASS (THE SIMPLEST WAY)

```
// Concept and Mock! (It's not our API, yet!)
using MarketData = decltype( Callable<void()>($(connect)) );
```

```
template<class TMarketData = MarketData>
struct trading_system {
  TMarketData& marketData;
  void start() { marketData.connect(); }
};
```

TDD/RED - WRITE ANOTHER TEST (SWITCH ROLES!)

```
"should not disconnect on stop if it wasn't connected"_test = [] {
  auto [ts, mocks] = testing::make<trading_system>();

  EXPECT_CALL(mocks<MarketData>(), disconnect).Times(0);

  ts.stop();
};
```

TDD/GREEN - MAKE ALL TESTS PASS

```
// Disconnect needed to verify that it wasn't called
using MarketData = decltype( Callable<void()>($(connect)) &&
                             Callable<void()>($(disconnect)) );
```

```
template<class TMarketData = MarketData>
struct trading_system {
  TMarketData& marketData;

  void start() { marketData.connect(); }
  void stop() { } // The simplest implementation!
};
```

TDD - REFACTOR (IN BOTH, CODE AND TESTS)

- Remove duplicates
- Cleanup the code
- Extract files
- ...

TDD/REFACTOR - REMOVE DUPLICATION OF CREATING A TRADING_SYSTEM AND MOCKS

```
"Trading System [Parse Feeds]"_test = [] {
  auto [ts, mocks] = testing::make<trading_system>(); // DRY, setup

  SHOULD("connect to the market data on startup") {
    EXPECT_CALL(mocks<MarketData>(), connect);
    ts.start();
  }

  SHOULD("not disconnect on stop if it wasn't connected") {
    EXPECT_CALL(mocks<MarketData>(), disconnect).Times(0);
    ts.stop();
  };
};
```

TDD/REFACTOR - CLEANUP THE CODE

```
template<class TMarketData = MarketData>
class trading_system {
public:
  explicit trading_system(TMarketData& marketData)
    : marketData(marketData)
  { }

  void start() {
    marketData.connect();
  }

private:
  TMarketData& marketData;
};
```

TDD/RED - WRITE ANOTHER TEST (SWITCH ROLES AGAIN!)

```
"Trading System [Parse Feeds]"_test = [] {
  auto [ts, mocks] = testing::make<trading_system>(); // DRY, setup

  SHOULD("connect to the market data on startup") { ... }
  SHOULD("not disconnect on stop if it wasn't connected") { ... }

  SHOULD("disconnect on stop if it was connected") {
    InSequence sequence{};
    {
      EXPECT_CALL(mocks<MarketData>(), connect);
      EXPECT_CALL(mocks<MarketData>(), disconnect);
    }

    ts.start();
    ts.stop();
  }
};
```

TDD/GREEN - MAKE IT PASS!

```
template<class TMarketData = MarketData>
class trading_system {
public:
  explicit trading_system(TMarketData& marketData)
    : marketData(marketData)
  { }

  void start() { marketData.connect(); connected = true; }
  void stop() {
    if (connected) {
      marketData.disconnect();
    }
  }
private:
  TMarketData& marketData;
  bool connected{};
};
```

TDD/REFACTOR - EXTRACT TO SEPARATE FILES IF NEEDED

```
#ifndef CONCEPTS_MARKET_DATA_HPP
#define CONCEPTS_MARKET_DATA_HPP // concepts/market_data.hpp

namespace trading_system::concepts {
inline namespace v1 {

using MarketData = decltype( Callable<void()>($(connect)) &&
                             Callable<void()>($(disconnect)) );
} // v1
}} // trading_system::concepts

#endif
```

AND SO ON...

WHEN ALL TASKS FOR THE STORY "[0.0.1] PARSE FEEDS" ARE IMPLEMENTED (IMPLIES THAT ALL TESTS ARE PASSING!) THEN...

COMMIT (REBASE IF NEEDED)

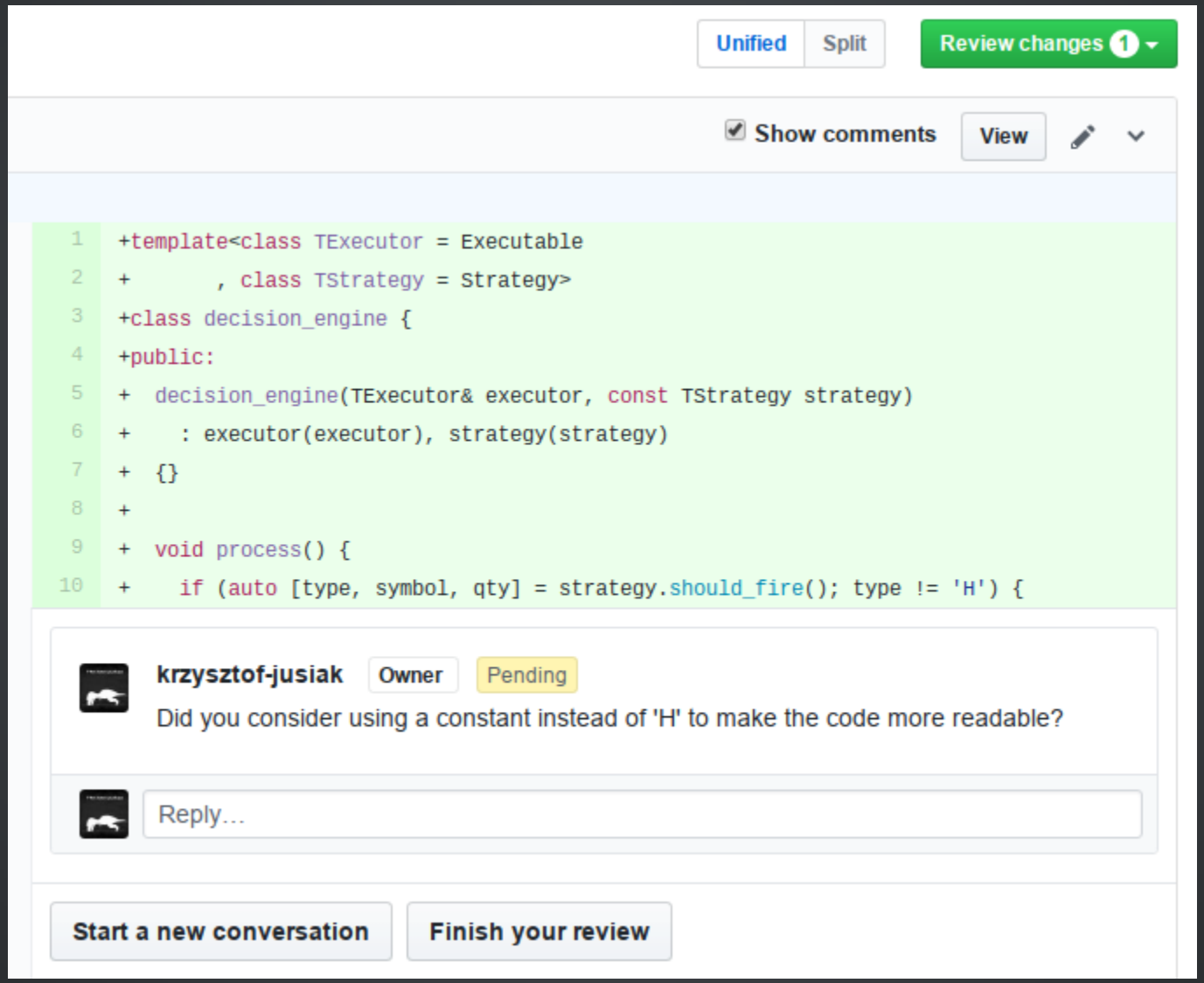
MESSAGE

```
[0.0.1] :new: Parse feeds - Ability to parse feeds from MarketData

Problem:
- Trading system has to connect to the market data
- Trading system is required to subscribe for market
  data feeds in order to receive data to make decisions
- Trading system has to disconnect from the market data

Solution:
- Connect to the market data on startup of the trading system
- Implement feed handler which can parse feeds from the market data
- Disconnect from the market data when there is no more feeds
```

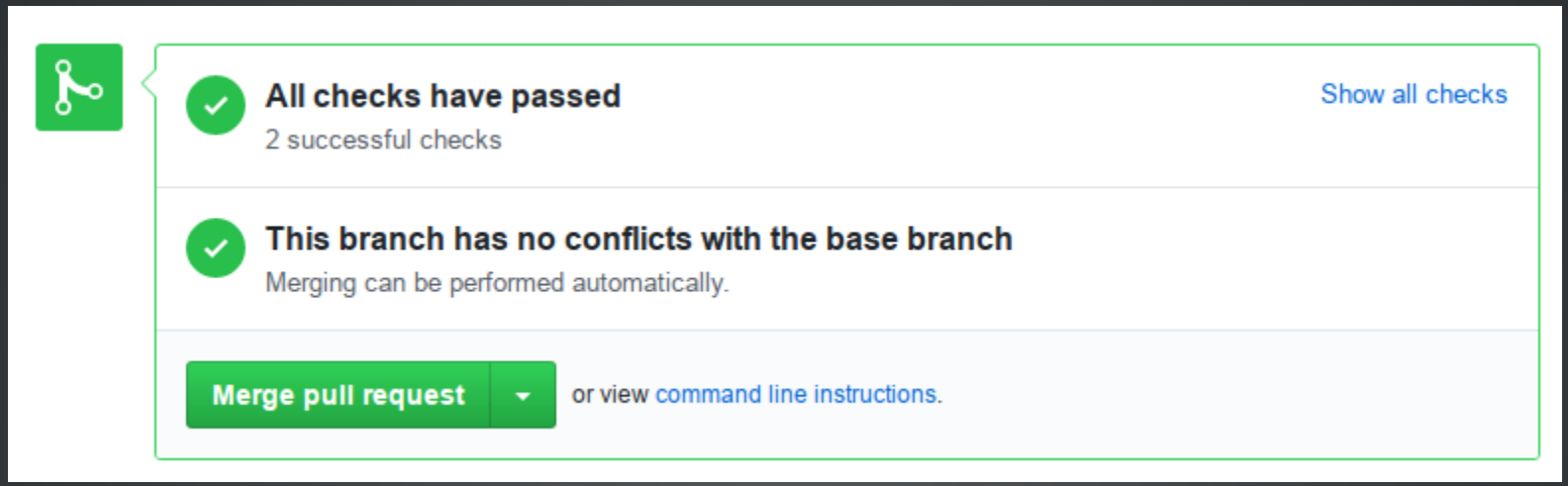
CONTENT (CROSS FUNCTIONAL DELIVERABLE WITH ALL TESTS)

```
- include/concepts/market_data.hpp
- include/trading_system.hpp
- test/unit/trading_system.cpp
- test/integration/trading_system.cpp // plumbing tests
- test/acceptance_tests/parse_feeds.cpp // cucumber scenarios
```

CODE REVIEW (DID YOU CONSIDER...?)



MERGE/PULL REQUEST (DEFINITION OF DONE)



MERGED BY A TEAM MEMBER, ONLY IF DOD IS SATISFIED:

- All code review dicussions were resolved
- All checks are passing
  - All tests/static,dynamic analysis, etc...

TODO	WIP	DONE
		~~[0.0.1] Parse Feeds~~
	[0.0.2] Handle Buy	<- Take the next story... (in priority order)

PAIR WITH SOMEONE ELSE OF THE TEAM! (KNOWLEDGE SHARING)

TESTING AND C++ 2X

AUTOMATIC TEST REGISTRATION AND STD::TESTING ASSERTIONS

```
"should add 2 numbers"_test = [] {
  std::testing::assert(4 == add(2, 2));
};
```

TEST SUITES/FIXTURES

```
"Calculator"_test_fixture = [] {
  // set-up
  auto calc = calcualtor{};

  "should add 2 numbers"_test = [&] {
    std::testing::assert(4 == calc.add(2, 2));
  };

  "should sub 2 numbers"_test = [&] {
    std::testing::assert(0 == calc.sub(2, 2));
  };

  // tear-down
};
```

STATIC REFLECTION - MOCKS GENERATION

```
template<class T>
$class GMock {
public:
    virtual ~GMock() noexcept { }
    constexpr {
      for... (auto f: $GMock.functions()) {
        f.make([] { fs...; });
      }
    }

private:
 flat_map<string, unique_ptr<UntypedFunctionMockerBase>> fs{};
};
```

PROPOSALS

(SG7) Static reflection	Mocks Generation
(SG8) Concepts lite	Type constraints
Virtual Concepts	Concepts based type erasure
?	Testing/Assertions

SUMMARY

GOOD PRACTISES ARE GOOD PRACTICES FOR A REASON!

- Consider using a good* testing framework
- Consider using a good* mocking framework
- Consider writing `SOLID` instead of `STUPID` code
- Consider using dependency injection framework to avoid the wiring mess and inject mocks automatically
- Consider writing tests before the implementation

LAST BUT NOT LEAST...

"IF YOU LIKED IT THEN YOU SHOULD HAVE PUT A TEST ON IT"
BEYONCE RULE

QUESTIONS?

Dependency Injection	[Boost].DI	https://github.com/boost-experimental/di
Virtual Concepts	VC	https://github.com/boost-experimental/vc
Testing/Mocking	GUnit	https://github.com/cpp-testing/GUnit

kris@jusiak.net | @krisjusiak | linkedin.com/in/kris-jusiak