Testability and API Design

Bloomberg

Engineering

CppNorth 2024
July 22, 2024

John Pavan, Team Lead
Lukas Zhao, Senior Software Engineer

TechAtBloomberg.com

Testability and API Design

Bloomberg

Engineering

CppNorth 2024
July 22, 2024

Aram Chung, Software Engineer
John Pavan, Team Lead
Lukas Zhao, Senior Software Engineer

TechAtBloomberg.com

Contents

- Background & Basics
- Motivation
- Techniques
- Generalized Mocking & Testing

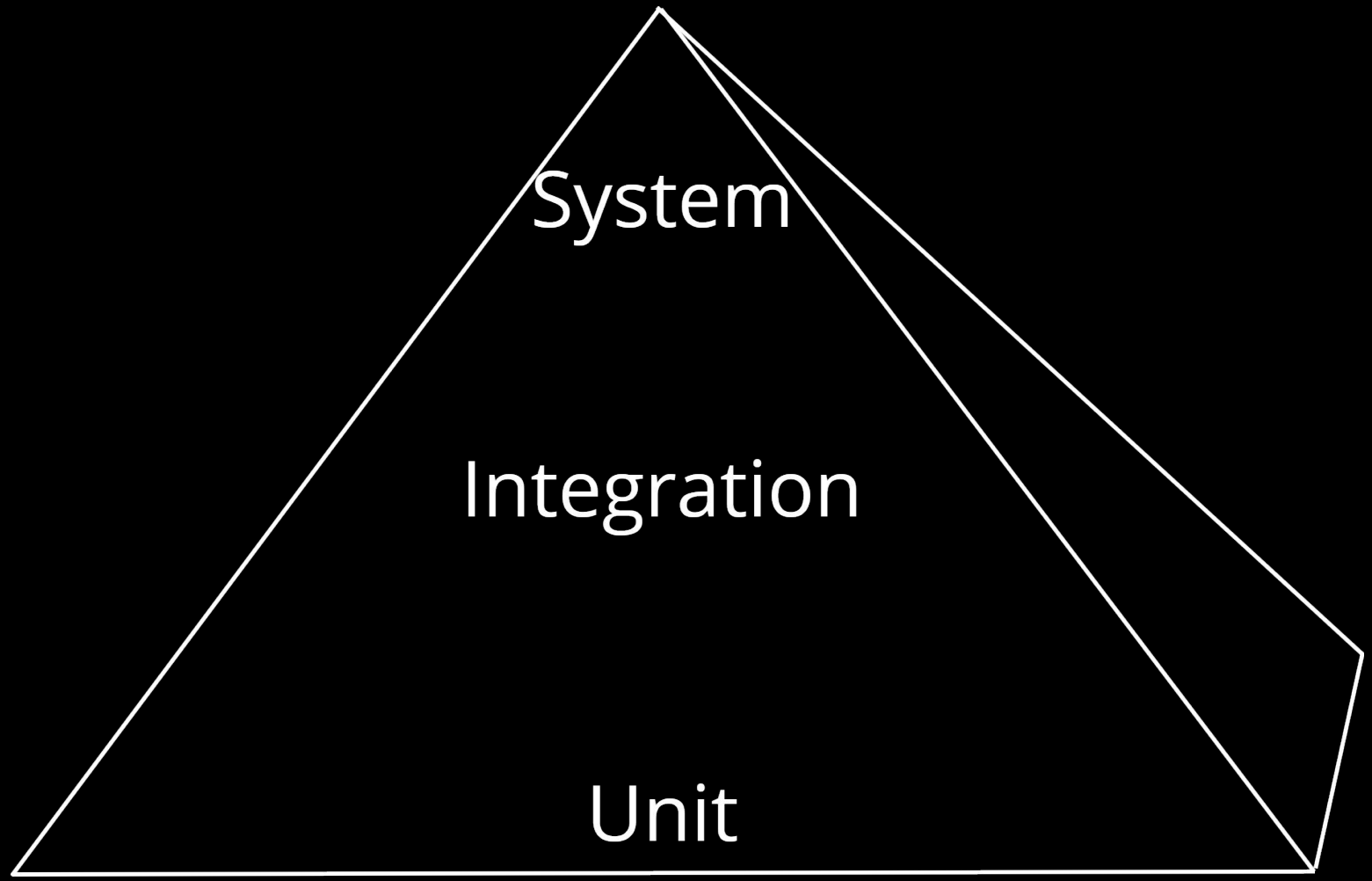
Background & Basics

- Brief testing overview
- Obligatory testing pyramid slide
- Some definitions

The Importance of testing

- Testing can verify both the happy-path and unhappy-path behaviors of your code
- Automated tests are run regularly, protect against regressions, and add value over time
- Unit, integration, and system tests have different roles and provide different benefits

Obligatory Testing Pyramid Slide



TechAtBloomberg.com

Bloomberg

Engineering

Unit Tests

- What component is tested? A unit test tests a component in isolation, with external dependencies stubbed out
- What behaviors are tested? A unit test verifies both happy-path and unhappy-path behaviors
- What is the goal? A unit test verifies that the component/class does what is expected
- What is mocked out? In a unit test, other components and classes need to be mocked out

(Did we mention that a unit test can test unhappy-path behaviors?)

Bloomberg

Engineering

Integration Tests

- What component is tested? An integration test tests a component that is running in the expected environment. This could be the full executable running in a contrived environment
- What behaviors are tested? An integration test verifies happy-path behaviors. But in an integration test it can be very difficult to test unhappy-path behaviors (e.g., force an I/O error)
- What is the goal? An integration test verifies that the component behaves as expected for its expected use case
- What is mocked out? In an integration test, other executables or system-level dependencies are often mocked out

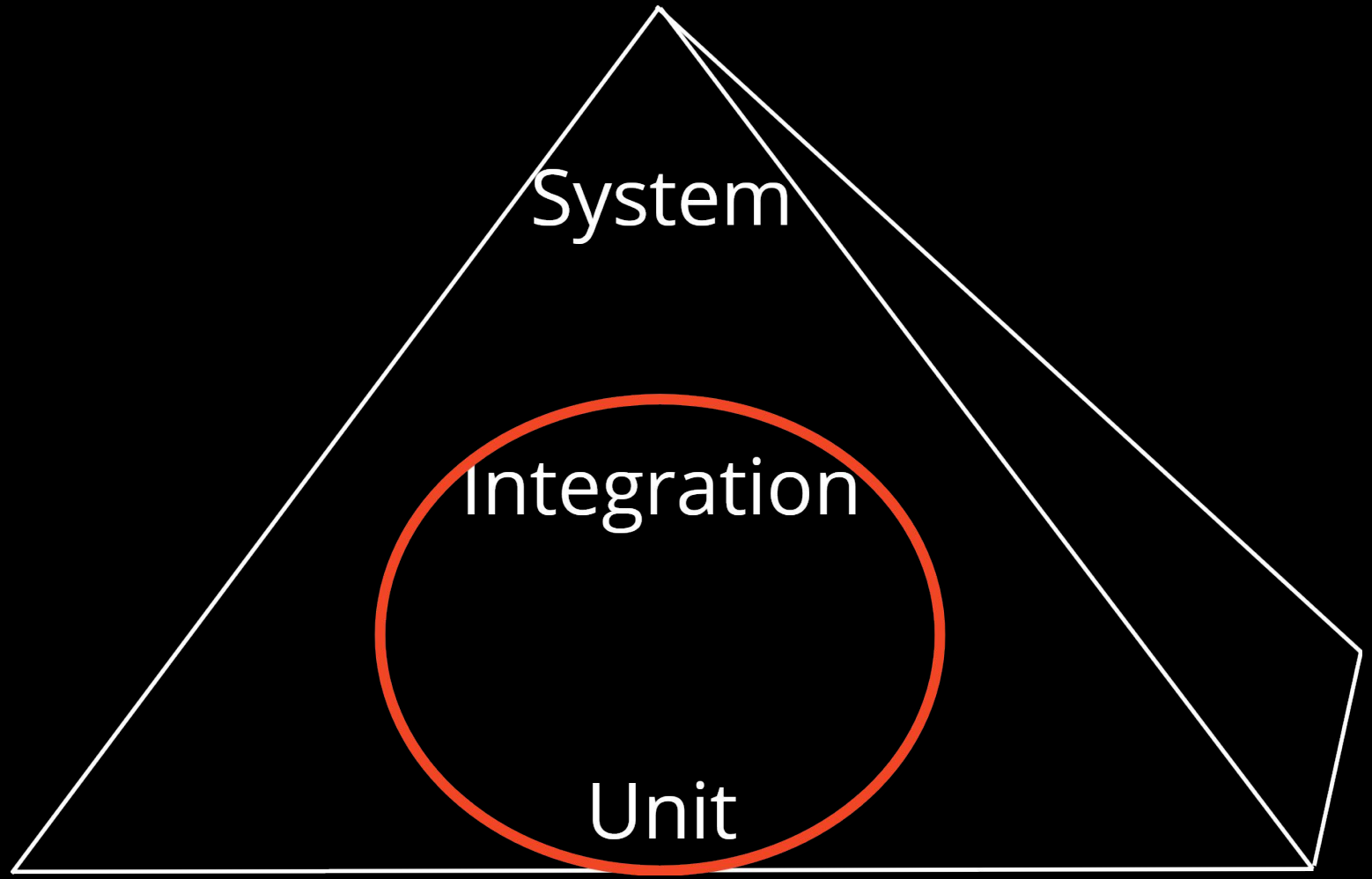
Bloomberg

Engineering

System Tests

- What component is tested? A system test tests an executable deployed to a test environment
- What behaviors are tested? A system test verifies happy-path behaviors
- What is the goal? A system test verifies functionality when integrated into a production-like system
- What is mocked out? In a system test, dependencies are not mocked out

Obligatory Testing Pyramid Slide



Motivation

- Designing for testability
- Designing for usage in test drivers
- Synergy between these & SOLID design principles

TechAtBloomberg.com

Bloomberg

Engineering

Designing for Testability

- How do I, the implementer, test my code?
- There is a deep synergy between testability and SOLID design principles

Designing for usage in test drivers

- How do the users of my library test their code?
- The API must be usable in test drivers and integration tests.
- If mocking is their chosen approach, I, the library implementer, must provide mock objects.

SOLID Design Principles

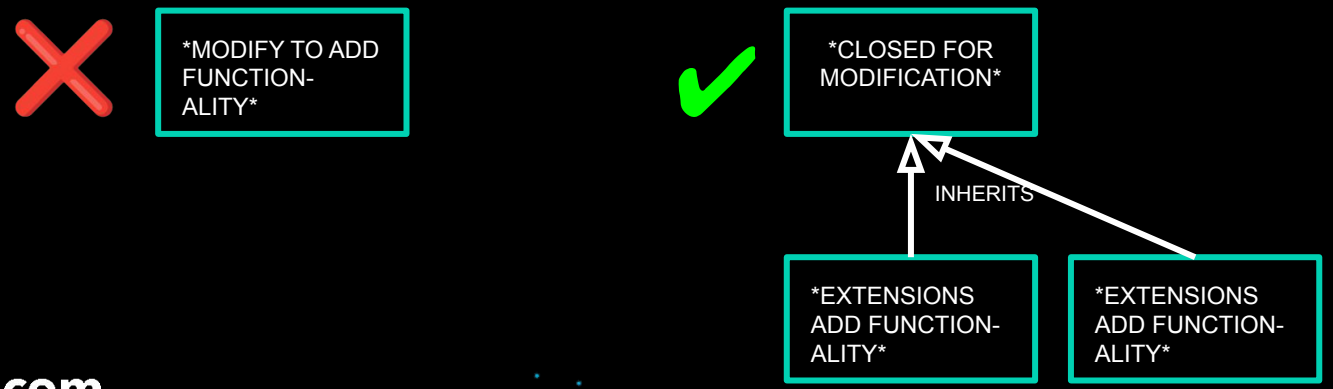
- Single Responsibility principle
- Open-closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency Inversion principle

Single Responsibility Principle

- Each class should only do one thing
- The single responsibility and interface segregation principles help not only with testing, but also with overall API design

Open-closed principle

- “Software entities should be open for extension, but closed for modification.”
- We should “be able to add new functionality without changing existing code.”
- Polymorphic: An abstract interface with multiple implementations



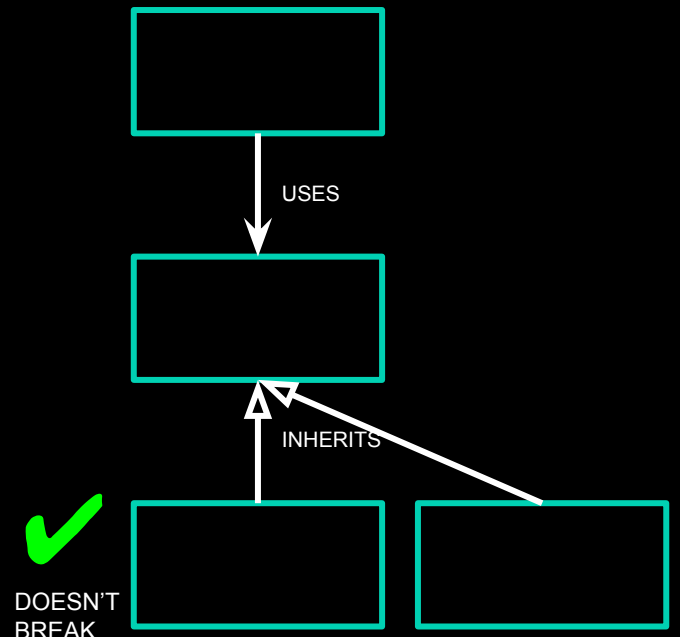
TechAtBloomberg.com

Bloomberg

Engineering

Liskov Substitution Principle

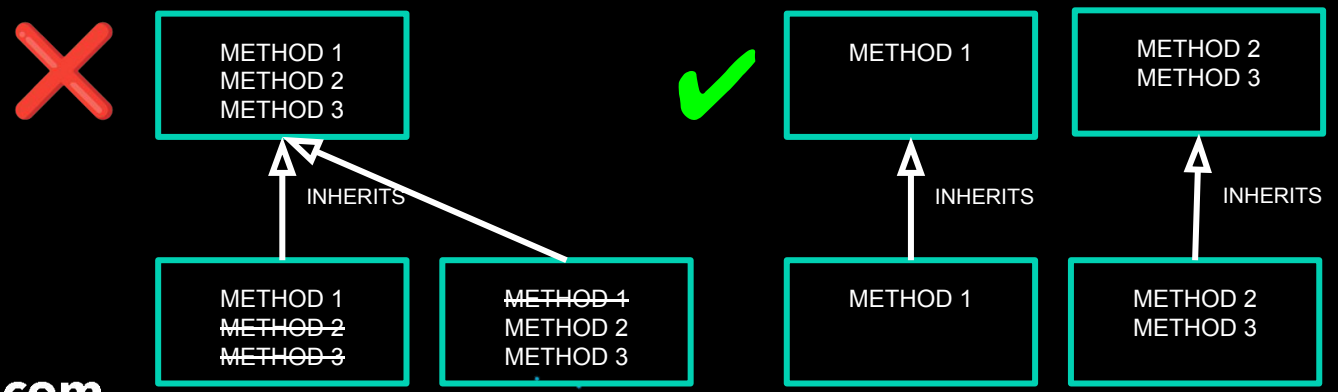
- a.k.a. Strong Behavioral Subtyping
- We should be able to replace a class with any class derived from it without breaking what's using it



❌ BREAKS

Interface Segregation Principle

- “No code should be forced to depend on methods that it does not use.”
- Split large interfaces into smaller ones



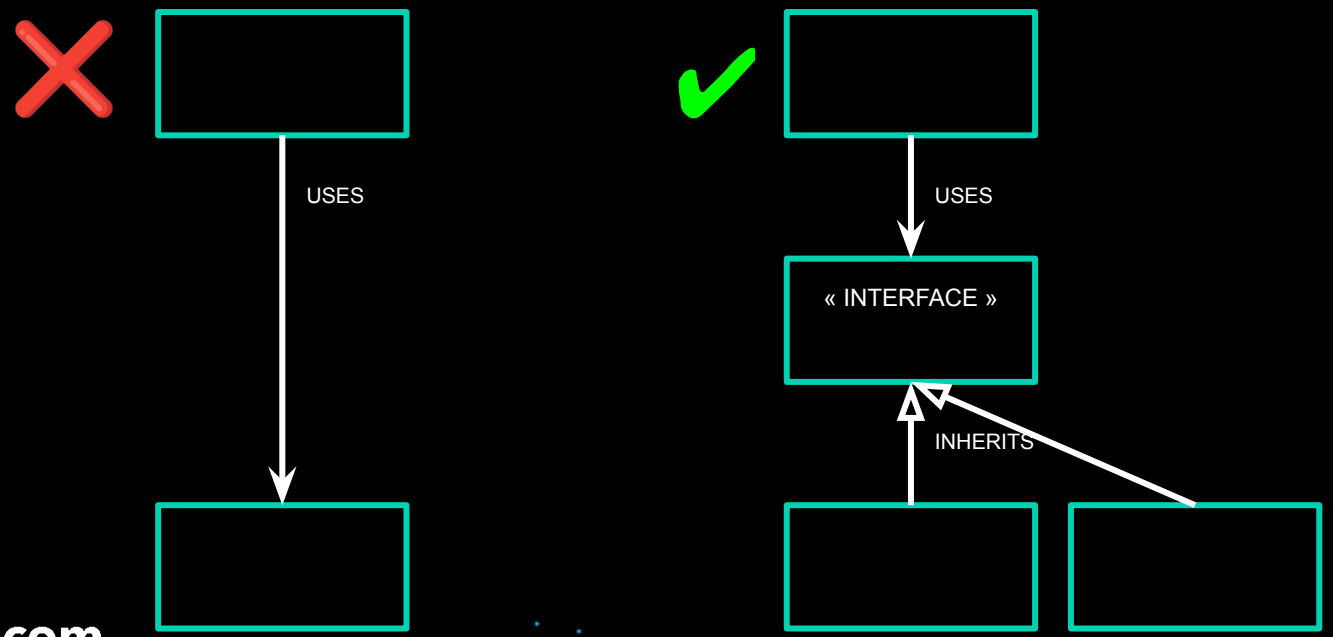
TechAtBloomberg.com

Bloomberg

Engineering

Dependency inversion principle

- “High-level modules should not import anything from low-level modules. Both should depend on abstractions (interfaces).”
- “Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions.”



Bloomberg
Engineering

Additional Considerations

- The API shouldn’t have features that exist only for testing
- API choices shouldn’t prevent testing
- We may need a way to stub out system dependencies

Techniques

- PImpl Idiom
- Depend only on interfaces
- Adding test APIs via ‘guards’
- Abstract Factory Methods

Interfaces and the PImpl idiom (1 of 2)

- “Pointer to implementation”
- Implementation details are stored in a separate class
- Allows different implementations for different systems and tests
- Implementations can be tested independently of the API
- Unit tests can have full access to the internals of the implementation

Interfaces and the Pimpl idiom (2 of 2)

```
// Without PImpl:
class Foo {
  // ...
  public:
    void func1()
    { // ... }
    // ...
};
```

```
struct FooImpl {
    void func1()
    {  //...  }
    // ...
};

class Foo {
  std::unique_ptr<FooImpl> d_impl;
  public:
    void func1() { d_impl->func1(); }
    // ...
};
```

Example (Good): BlazingMQ's CRC32 checksums (1 of 2)

```
// public API:
struct Crc32c {
    static unsigned int calculate(
                const void* data,
                unsigned int length,
                unsigned int crc = k_NULL_CRC32C);
    //...
};
```

Example (Good): BlazingMQ's CRC32 checksums (2 of 2)

```
struct Crc32c_Impl {
    static unsigned int calculateSoftware(
      const void* data,
      unsigned int length,
      unsigned int crc = Crc32c::k_NULL_CRC32C);

    static unsigned int calculateHardwareSerial(
      const void* data,
      unsigned int length,
      unsigned int crc = Crc32c::k_NULL_CRC32C);
     // ...
};
```

Depend only on interfaces (1 of 6)

- Calls back to the (polymorphic) open-closed, Liskov substitution, and dependency-inversion principles
- Makes mocks and stubs easier to implement
- Create interfaces for anything needed to write tests
  - Including system calls

Depend only on interfaces (2 of 6)

Pass interfaces into the Ctor

```
class Foo {
    Interface1* d_interface1_p;
    Interface2* d_interface2_p;
  public:
    Foo(Interface1* intrf1, Interface2* intrf2);
    void func1();
    // ...
};
```

Depend only on interfaces (3 of 6)

What if you depend on something that isn’t an interface?

```
class Foo {
  // ...
  public:
    using bsl::function<void()> FooFunctor;
  private:
    FooFunctor d_fooFunctor;
  public:
    Foo(FooFunctor fooFunctor = ProdBehavior());
  // ...
};
```

Depend only on interfaces (4 of 6)

Provide interfaces for those using your library

```
class FooIntrf {
    virtual void func1() = 0;
    // ...
};

class Foo : public FooIntrf {
    // ...
  public:
    virtual void func1();
    // ...
};
```

Depend only on interfaces (5 of 6)

Templates as a workaround to avoid heap allocation

```
template <typename BAR>
class Foo {
    BAR* d_bar_p;
  public:
    Foo(BAR* bar);
    void func1();
    // ...
};

using Foo<RealBar> RealFoo;
using Foo<TestBar> TestFoo;
```

```
class FooIntrf {
    virtual void func1() = 0;
    // ...
};

template<typename BAR>
class Foo : public FooIntrf {
 // ...
};

using Foo<RealBar> RealFoo;
using Foo<TestBar> TestFoo;
```

Depend only on interfaces (6 of 6)

Example (Good): things that use bslma::Allocator

```
bdlbb::PooledBlobBufferFactory(
            int bufferSize,
            bslma::Allocator *basicAllocator = 0);
```

How to add test APIs (1 of 4)

- Single responsibility and interface segregation principles mean we shouldn't have test-specific functions/methods in the public API

- Resource-Acquisition-Is-Initialization (RAII) style guard that adds the test methods and functions

How to add test APIs (2 of 4)

Good Example: bdlmt::EventScheduler and EventSchedulerTestTimeSource

```
bdlmt::EventScheduler schdlr;
bdlmt::EventSchedulerTestTimeSource tstTmSrc(&schdlr);
// ...
schdlr.start()
// ...
tstTmSrc.advanceTime(bsls::TimeInterval(40));
// ...
```

How to add test APIs (3 of 4)

```
class Foo {
  // ...
    ThingFunctor d_thingFunctor;
    friend class FooTestGrd;
  public:
    Foo();
    void func1();
    void func2();
  // ...
};
```

```
class FooTestGrd {
  // ...
  public:
    FooTestGrd(Foo* foo);
    void testAPI1();
    void testAPI2();
    // ...
};
```

How to add test APIs (4 of 4)

```
class OwnsFoo {
    Foo d_foo; // no way to install FooTestGrd
  public:
    OwnsFoo();
};
```

```
class UsesFoo {
    Foo* d_foo_p;
  public:
    UsesFoo(Foo* foo); // pass in w/ FooTestGrd
};
```

Abstract Factory Pattern (1 of 2)

```
class FactoryIntf {
  public:
    FactoryIntf();
    virtual ~FactoryIntf();

    virtual FooIntf* makeFoo() const;
    virtual BarIntf* makeBar() const;
    virtual BazIntf* makeBaz() const;
    // ...
};
```

Abstract Factory Pattern (2 of 2)

```
class RealFactory : public FactoryIntf
{
  //...
    FooImpl* makeFoo() const;
    BarImpl* makeBar() const;
    BazImpl* makeBaz() const;
    // ...
};
```

```
class TestFactory : public FactoryIntf
{
  //...
    FooImpl* makeFoo() const;
    BarImpl* makeBar() const;
    BazImpl* makeBaz() const;
    // ...
};
```

Example (good) - mwcio::ChannelFactory (1 of 2)

```
ChannelFactory {
  // ...
    virtual void listen (
            Status* status,
            bslma::ManagedPtr<OpHandle>* handle,
            const ListenOptions& options,
            const ResultCallback& cb) = 0;
  // ...
};
```

Example (good) - mwcio::ChannelFactory (2 of 2)

```
class TestChannelFactory : public ChannelFactory {
  // ...
    void reset();
    void setListenStatus();
    void setConnectStatus();
    mwct::PropertyBag& newHandleProperties();
    bsl::deque<ListenCall>& listenCalls();
    bsl::deque<ConnectCall>& connectCalls();
    bsl::deque<HandleCancelCall>& handleCancelCalls();
  // ...
};
```

Generalized Mocking & Testing

- Techniques
- Challenges
- A proposal for life quality improvement

TechAtBloomberg.com

Bloomberg

Engineering

Technique

- Mocking with DI via an abstract interface
- Mocking with DI via a template type
- Who is responsible for DI?
  - PImpl idiom
  - Poor man’s DI

Technique - Mocking via an abstract interface

```
class DogI {
  public:
    virtual ~DogI() = default;
    virtual int bark() = 0;
};
```

```
class Dog : public DogI {
  public:
    int bark() override;
};
```

```
class DogMock : public DogI {
  public:
    MOCK_METHOD(int, bark, (),
                        (override));
};
```

```
class MyBusiness {
  private:
    shared_ptr<DogI> d_dogI;

  public:
    MyBusiness(shared_ptr<DogI> dogI)
    : d_dogI(move(dogI)) {}

    bool doSomething() {
        auto rc = d_dogI->bark();
        // ...
    }
};
```

Technique - Mocking via an abstract interface

```
class TestMyBusiness : public Test {
  protected:
    // mocks
    shared_ptr<DogMock> d_dogMock;

    // Subject Under Test (SUT)
    unique_ptr<MyBusiness> d_myBusiness;

    void SetUp() {
        // instantiate mocks
        d_dogMock = make_shared<DogMock>();

        // instantiate SUT, inject mocks
        d_myBusiness =
           make_unique<MyBusiness>(d_dogMock);
    }
};
```

```
TEST_F(TestMyBusiness, test_doSomthing)
{
    // Given
    EXPECT_CALL(*d_dogMock, bark())
        .Times(1).WillOnce(Return(0));

    // When
    bool success =
        d_myBusiness->doSomething();

    // Then
    EXPECT_TRUE(success);
}
```

Technique - Mocking via a template type

```
class Dog {
  public:
    int bark();
};
```

```
class DogMock {
  public:
    MOCK_METHOD(int, bark, ());
};
```

```
template<typename DOG_TYPE>
class MyBusiness {
  private:
    DOG_TYPE d_dog;

  public:
    MyBusiness(const DOG_TYPE& dog)
    : d_dog(dog) {}

    bool doSomething() {
        d_dog.bark();
        // ...
    }
};
```

```
TEST(TestMyBusiness, test_doSomething)
{
    // Given SUT
    MyBusiness<DogMock> myBusiness(dogMock);
    // ...
}
```

TechAtBloomberg.com

Bloomberg

Engineering

Technique - Who is responsible for injecting dependencies?


```
class AwesomeLib {
  private:
    int d_uuid;
    Dog d_dog;
    Cat d_cat;

  public:
    AwesomeLib(int uuid,
               shared_ptr<AsvcI> asvcI,
               shared_ptr<BsvcI> bsvcI,
               shared_ptr<CsvcI> csvcI)
    : d_uuid(uuid)
    , d_dog(uuid, asvcI)
    , d_cat(bsvcI, csvcI)
    {}

    bool talk() {
        // ...
    }
};
```

AwesomeLib user: “Do I need to inject Asvc, Bsvc, Csvc?!”

Technique - Who is responsible for DI? Pimpl idiom

TechAtBloomberg.com

Bloomberg

Engineering

Technique - Who is responsible for DI? Pimpl idiom

```
class AwesomeLib : public AwesomeLibI {
  private:
    unique_ptr<AwesomeLibImp> d_awesomeLibImp;

  public:
    AwesomeLib(int uuid)
    {
        // construct real dependencies
        auto asvc = make_shared<Asvc>();
        auto bsvc = make_shared<Bsvc>();
        auto csvc = make_shared<Csvc>();

        // inject dependencies into Imp
        d_awesomeLibImp = make_unique<AwesomeLibImp>(
            uuid, asvc, bsvc, csvc);
    }

    bool talk()
    {
        return d_awesomeLibImp->talk();
    }
};
```

```
class AwesomeLibImp {
  private:
    int d_uuid;
    Dog d_dog;
    Cat d_cat;

  public:
    AwesomeLibImp(int uuid,
                  shared_ptr<AsvcI> asvcI,
                  shared_ptr<BsvcI> bsvcI,
                  shared_ptr<CsvcI> csvcI)
    : d_uuid(uuid)
    , d_dog(uuid, asvcI)
    , d_cat(bsvcI, csvcI) {}
        // AwesomeLibImp ctor opens up for
        // dependency injection, so
        // AwesomeLibImp can be unit-tested

    bool talk() {
        // real logic of talk() function
        // ...
    }
};
```

Technique - Who is responsible for DI? Poor man's DI

```
class AwesomeLib : public AwesomeLibI {
  private:
    int d_uuid;
    shared_ptr<Dog> d_dog;
    shared_ptr<Cat> d_cat;

  public:
    // ctor, instantiate all deps as real impls
    AwesomeLib(int uuid) : d_uuid(uuid)
    {
        auto asvc = make_shared<Asvc>();
        auto bsvc = make_shared<Bsvc>();
        auto csvc = make_shared<Csvc>();

        d_dog = make_shared<Dog>(uuid, asvc);
        d_cat = make_shared<Cat>(bsvc, csvc);
    }
```

```
    // ctor, opens up for dep injection
    AwesomeLib(int uuid,
               shared_ptr<AsvcI> asvcI,
               shared_ptr<BsvcI> bsvcI,
               shared_ptr<CsvcI> csvcI) :
        d_uuid(uuid),
        d_dog(make_shared<Dog>(uuid, asvcI)),
        d_cat(make_shared<Cat>(bsvcI, csvcI))
    {}

    bool talk()
    {
        // ...
    }
};
```

Challenges

- Upfront investment is unpleasant; proper testable code requires more boilerplate code and more files
- New C++ engineers may not be familiar with various patterns and techniques
- Isn’t it just much easier ***not*** to do the right things?

  → Make it easier to do the right things
  → “One-click” design patterns

Propose a new tool

Use a script to write boilerplate code for creating interfaces, mocks, unit tests, pimpl idiom, dependency injection, etc.

+ Asks you a few questions

+ Generates all boilerplate code with tests and specified patterns

Propose a new tool

Faster to do it right and better

+ No need to write boilerplate → encourage small classes
+ No need to set up unit tests → write tests as you write new functions
+ Create interface and mock
+ Create PImpl to isolate dependencies
+ Hints
+ Modern C++

Thank you!

https://techatbloomberg.com/cplusplus

https://www.bloomberg.com/careers

Contact Us!
Aram - achung118@bloomberg.net
John - jpavan@bloomberg.net
Lukas - zzhao106@bloomberg.net

Bloomberg

Engineering

TechAtBloomberg.com

Extra Slides

TechAtBloomberg.com

Bloomberg

Engineering

Definitions

- Mock
  - Implements the interface
  - Provides a way to set return values and/or expectations
  - Provides a way to determine success or failure based on if it was called
- Stub
  - Implements the interface
  - Provides a way to set return values and/or expectations
- Fake
  - Satisfies the interface, but contains minimal logic and fixed data
- Happy Path
  - How the component behaves for the intended use case
- Unhappy Path
  - How the component behaves under unexpected conditions: I/O failures, time going backwards, etc.