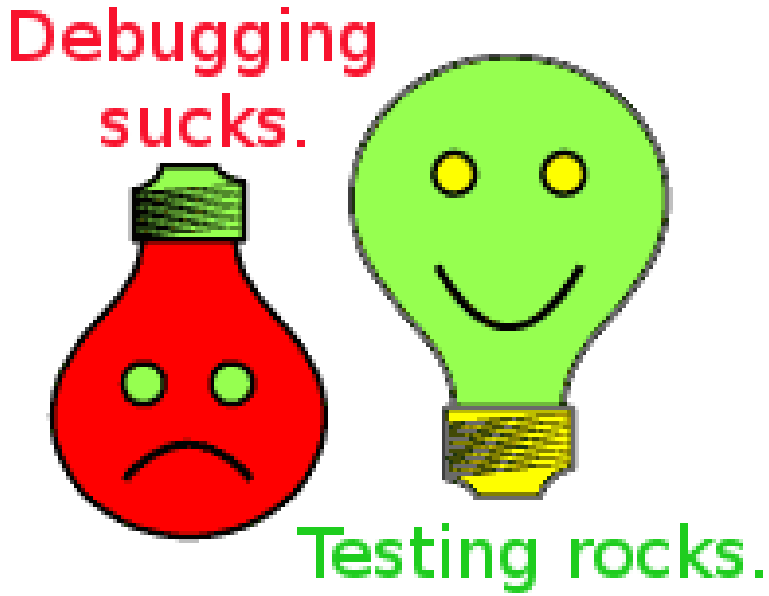


Testing y Mocking: Probando Aplicaciones (En C++)

Intraway
Corporation

IWay Tech Talks

May 28, 2009

Outline

Imágenes del capítulo anterior

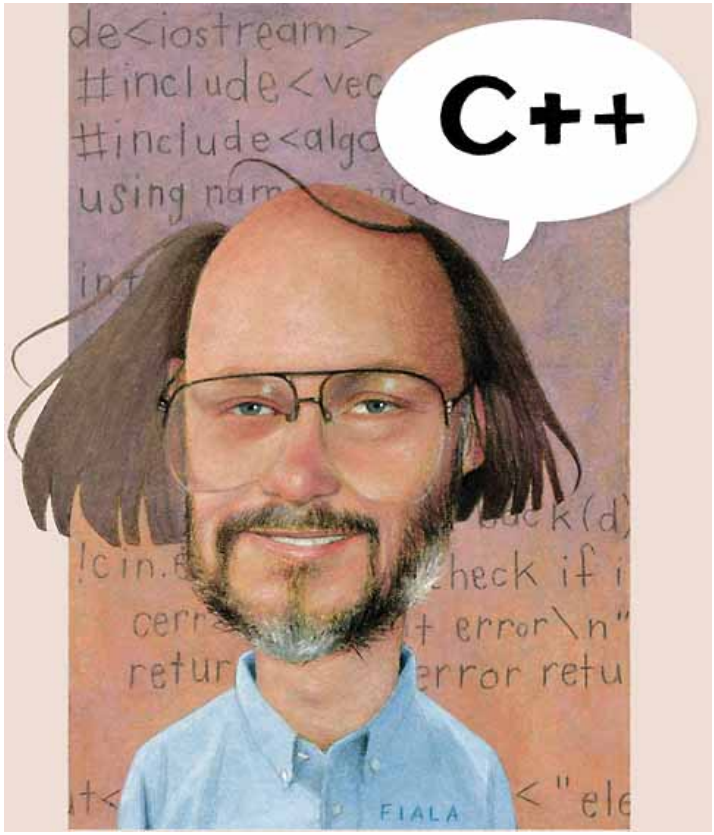
La última vez aprendimos que...

- testear es bueno
- los bugs son malos
- existen muchos tipos de tests
- tests unitarios $\Longrightarrow$ módulo aislado
- DI [a] ayuda a testear
- "mocking" es una técnica de testing

[a] inyección de dependencias

Imágenes del capítulo anterior

Pero no dijimos que en C++. . .



- se puede testar
- se puede mockear
- no es díficil!
- hay frameworks copados
- gTest sirve para unit testing
- gMock sirve para mocking
- gTest y gMock son de Google
- hay técnicas de testing

Imágenes del capítulo anterior

Tampoco dijimos que las avestruces son simpáticas

Outline

¿Qué hacemos en C++ cuando hay un bug?

Luego de llorar por los rincones

- Aislamos el problema
- Arreglamos el problema
- Commiteamos un fix
- goto 10

O creamos un test. . .

Un ejemplo

```
1  class Handshake {
2    bool saludar() {
3      Command cmd; ACE_SOCK_Stream sock;
4      uint32_t bt; ACE_Time_Value timeout(20);
5
6      ssize_t res = sock.send_n("Hello",
7                                sizeof("Hello"), &timeout, &bytes);
8
9      if (res <= 0 || bytes < msg.size())
10       DbSingleton->getInstance()->save_error("send");
11
12     res = sock.recv_n(cmd.buffer(),
13                       cmd.sizeof_buffer(), &timeout, &bytes);
14
15     if (res <= 0 || bytes < msg.size())
16       DbSingleton->getInstance()->save_princess("recv");
17
18     return true;
19   }
20 };
```

Un ejemplo y su test

```
1  class Foobar : public ACE_Svc_Handler <ACE_SOCK_Stream, ACE_MT_SYNCH> {
2    public:
3    int open(void *msg){
4      assert(0 == strcmp((char*)msg, MSG_1));
5      peer().send_n(MSG_2, strlen(MSG_2));
6      peer().close();
7      return 0;
8    }
9  };
10
11 void test_handshake() {
12   DbSingleton = new Singleton;
13   DbSingleton->setInstance(new ExcelDB);
14   DbSingleton->getInstance()->truncate_errors("recv");
15   DbSingleton->getInstance()->truncate_errors("send");
16
17   ACE_Acceptor<Foobar, ACE_SOCK_ACCEPTOR> acceptor(PORT);
18   Handshake saludador;
19   saludador.saludar();
20
21   int errs = DbSingleton->getInstance()->get_errors_cnt("recv");
22   assert(errs == 0);
23 }
24
25 int main(){
26         test_handshake();
27         return 0;
28 }
```

Un ejemplo y su test

```
1  class Foobar : public ACE_Svc_Handler <ACE_SOCK_Stream, ACE_MT_SYNCH> {
2    public:
3    int open(void *msg){
4      assert(0 == strcmp((char*)msg, MSG_1));
5      peer().send_n("Garbage", strlen("Garbage"));
6      peer().close();
7      return 0;
8    }
9  };
10
11 void test_handshake_fail() {
12   DbSingleton = new Singleton;
13   DbSingleton->setInstance(new ExcelDB);
14   DbSingleton->getInstance()->truncate_errors("recv");
15   DbSingleton->getInstance()->truncate_errors("send");
16
17   ACE_Acceptor<Foobar, ACE_SOCK_ACCEPTOR> acceptor(PORT);
18   Handshake saludador;
19   saludador.saludar();
20
21   int errs = DbSingleton->getInstance()->get_errors_cnt("recv");
22   assert(errs == 1);
23 }
24
25 int main(){
26         test_handshake_fail();
27         return 0;
28 }
```

Un ejemplo y su test: Resultados

Descubrimos un bug

```
a.out: test2.cpp:5: int main():
      Assertion `errs == 1' failed.
Cancelado
```

El mensaje no es muy descriptivo pero como el código es poco *cambiemos*

DbSingleton->getInstance()-> save_princess ("recv");

por

DbSingleton->getInstance()-> save_error ("recv");

Queda una pregunta...

Un ejemplo y su test: Resultados

¿Todo eso para un test?

Un ejemplo y su test

El test del ejemplo tiene varios problemas

- El test es fragil (¿y si PORT está ocupado?)
- La intención del test es poco clara
- Hace falta mucho "scaffolding"
- Los asserts no son herramientas de test

El código del ejemplo también...

- Handshake depende de una implementación
- Hay dependencias ocultas
- El código es poco flexible
- ¿Alguien encuentra más?

Por suerte arreglando el test arreglamos el código!

Outline

No más asserts

Google code **googletest**
Google C++ Testing Framework

¿Por qué un framework de testing?

- Nos olvidamos del scaffolding
- Ayuda a crear tests independientes y repetibles
- Imprime mas info de debug que un assert
- Provee herramientas para distintos tipos de tests

RTFM @ `http://code.google.com/p/googletest`

GoogleTest: Conceptos Básicos

RTFM @ GoogleTestAdvancedGuide

Estructura

- Cada test por separado, con la macro TEST
- El test tiene un nombre y un grupo
- Hay "expectations" en distintos puntos
- Si requiere estado $\Longrightarrow$ TEST_F (fixture)

Expectations / Assertions: Lo mas simple

- assert() $\Longrightarrow$ ASSERT_* y EXPECT_*
- {ASSERT, EXPECT}_{TRUE, FALSE, EQ, LT, STREQ}
- EXPECT_* $\Longrightarrow$ no fatal
- ASSERT_* $\Longrightarrow$ fatal (interrumpe el test)

Un ejemplo y su test

```
1  class Contestador : public ACE_Svc_Handler <ACE_SOCK_Stream, ACE_MT_SYNCH>
2    public:
3    int open(void *msg){
4      EXPECT_STREQ((char*)msg, MSG_1);
5      peer().send_n(MSG_2, strlen(MSG_2));
6      peer().close();
7      return 0;
8    }
9  };
10
11 TEST(HandshakeTest, TxOK) {
12   DbSingleton = new Singleton;
13   DbSingleton->setInstance(new ExcelDB);
14
15   ACE_Acceptor<Foobar, ACE_SOCK_ACCEPTOR> acceptor(PORT);
16   Handshake saludador;
17   saludador.saludar();
18
19   int errs = DbSingleton->getInstance()->get_errors_cnt("recv");
20   EXPECT_EQ(0, errs);
21 }
```

Un ejemplo y su test

```
1  class Contestador : public ACE_Svc_Handler <ACE_SOCK_Stream, ACE_MT_SYNCH>
2    public:
3    int open(void *msg){
4      EXPECT_STREQ((char*)msg, MSG_1);
5      peer().send_n("Garbage", strlen("Garbage"));
6      peer().close();
7      return 0;
8    }
9  };
10
11 TEST(HandshakeTest, RecvFail) {
12   DbSingleton = new Singleton;
13   DbSingleton->setInstance(new ExcelDB);
14
15   ACE_Acceptor<Foobar, ACE_SOCK_ACCEPTOR> acceptor(PORT);
16   Handshake saludador;
17   saludador.saludar();
18
19   int errs = DbSingleton->getInstance()->get_errors_cnt("recv");
20   EXPECT_EQ(1, errs);
21 }
```

Un ejemplo y su test

Está un poco mejor, pero todavía...

- Hay mucho scaffolding
- Es frágil
- Expectations en todos lados $\Longrightarrow$ test poco claro

¿Qué ganamos?

- Menos código (Linkear con -lgtest_main)
- Mejores mensajes de error

Veamos como se ve en funcionamiento...

Test run del test: Todo OK

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Handshake
[ RUN      ] Handshake.TxOK
[       OK ] Handshake.TxOK
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 1 test.
```

Test run del test: Una falla

```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[----------] Global test environment set-up.
[----------] 1 test from Handshake
[ RUN      ] Handshake.TxOK
test1.cpp:4: Failure
Value of: "Adios mundo cruel"
Expected: (char*)msg
Which is: "Hola mundo"
test1.cpp:20: Failure
Value of: errs
  Actual: 1
Expected: 0
[  FAILED  ] Handshake.TxOK
[----------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] Handshake.TxOK

 1 FAILED TEST
```

Unit Test: The Revenge

Sabiendo de que se trata gTest... *Mejoremos un poco el test*

- ¿No molesta el código repetido?
- ¿No es feo crear a mano el estado previo al test?

Reduzca el nivel de stress, use Fixtures!

Así queda el Fixture

```
1  class HandshakeTest : public Test { public:
2    // Global a todo el test suite
3    static void TearDownTestCase(){}
4    static void SetUpTestCase(){ DbSingleton.setInstance(&ExcelDB); }
5
6    // Por cada test case
7    void TearDown(){}
8    void SetUp(){
9      db.truncate_errors("recv");
10     db.truncate_errors("send");
11   }
12
13   // Puedo instanciar variables "globales" a mis tests
14   Handshake saludador; Singleton DbSingleton; ExcelDB db;
15 };
16
17 class Contestador : public ACE_Svc_Handler <ACE_SOCK_Stream, ACE_MT_SYNCH> {
18   public:
19   virtual const char* get_msg() = 0;
20   int open(void *msg){
21     EXPECT_STREQ((char*)msg, MSG_1);
22     peer().send_n(get_msg(), strlen(get_msg()));
23     peer().close();
24     return 0;
25   }
26 };
```

Así quedan los tests

```
1  TEST_F(HandshakeTest, TxOK) {
2    class Contestador_OK : public Contestador {
3      public: const char* get_msg(){ return MSG_2; }
4    };
5
6    ACE_Acceptor<Contestador_OK, ACE_SOCK_ACCEPTOR> acceptor(PORT);
7    saludador.saludar();
8    EXPECT_EQ(0, db.get_errors_cnt("recv"));
9  }
10
11 TEST_F(HandshakeTest, RecvFail) {
12   class Contestador_Fail: public Contestador {
13     public: const char* get_msg(){ return "Garbage"; }
14   };
15
16   ACE_Acceptor<Contestador_Fail, ACE_SOCK_ACCEPTOR> acceptor(PORT);
17   saludador.saludar();
18   EXPECT_EQ(1, db.get_errors_cnt("recv"));
19 }
```

Google Test: algunos tips

- En EXPECT_* usar primero el valor y luego la variable
  Los mensajes de error serán mas claros
- Usar EXPECT_THROW({ . . . }, ExcpetionType)
  EXPECT_THROW es muy útil pero no olvidar las llaves
- Los death tests pueden servir para algo
  Un death test forkea el proceso y espera que salga el hijo
- No usar especificadores de excepciones
  Hacen desastres junto con los templates. Son feas anyway.
- No usar como nombre del test el de la clase
  Pueden ocurrir cosas bizarras y es mas difícil usar GDB

Todavía no terminamos...

- El test sigue siendo frágil
- Crear un socket puede ser lento
- El código es feo y quedan partes sin testear
  - ¿Cómo testeamos que pasa si falla el socket?
  - ¿Cómo testeamos que el comando se envíe bien?

Por suerte todavía nos queda ver Google Mock!

Outline

Mocking

Google code **googlemock**
Google C++ Mocking Framework

¿Para qué usar mocking?

- Permite testear comportamiento
- Permite mejorar la cobertura
- Ayuda a aislar componentes eliminando dependencias
- Permite acelerar tests lentos (p. ej. al eliminar una DB)
- Permite prototipar un módulo no implementado

RTFM @ `http://code.google.com/p/googlemock`

Mocking

Google code **googlemock**
Google C++ Mocking Framework

¿Google Mock?

- Mocking $\Longrightarrow$ lenguaje de tipado dinámico
- Hay pocos frameworks de mocking en C++
- Google Mock es una herramienta nueva
- No hace falta mantener los mocks a mano
- Es necesario entender DI, herencia y polimorfismo

RTFM @ `http://code.google.com/p/googlemock`

Un ejemplo y su test

Recordemos el código original: Click aquí para verlo

- Los singletons dificultan el testing
- La dependencia de la DB está oculta
- Hace falta conocer la implementación
- Depencia en la implementación, no en la interfaz

Empezemos por algo fácil, refactorizemos la clase para que

- no dependa de la implementación de la BD
- podamos verificar las condiciones de error

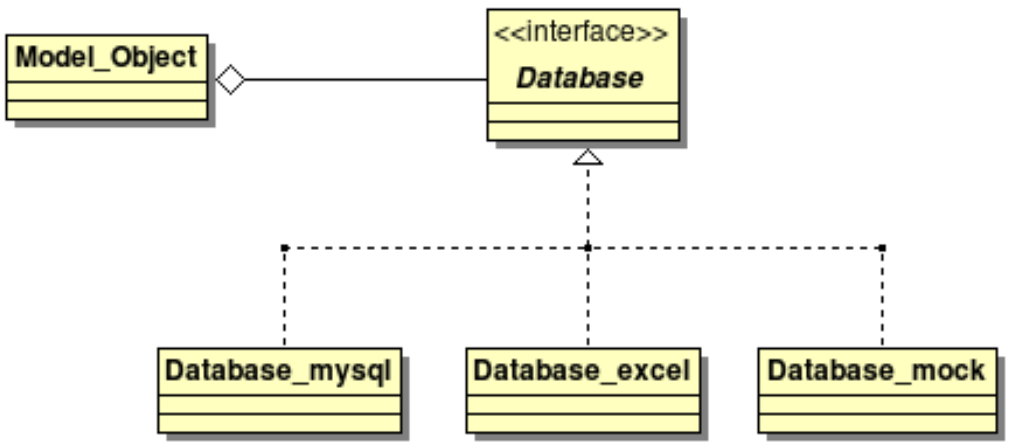
Un ejemplo, con 40% mas DI

```
1  class Handshake {
2    Db *db;
3    Handshake(Db *db) : db(db) {}
4
5    bool saludar() {
6      Command cmd; ACE_SOCK_Stream sock;
7      uint32_t bt; ACE_Time_Value timeout(20);
8
9      ssize_t res = sock.send_n("Hello",
10                               sizeof("Hello"), &timeout, &bytes);
11
12     if (res <= 0 || bytes < msg.size())
13       db->save_error("send");
14
15     res = sock.recv_n(cmd.buffer(),
16                       cmd.sizeof_buffer(), &timeout, &bytes);
17
18     if (res <= 0 || bytes < msg.size())
19       db->save_error("recv");
20
21     return true;
22   }
23 };
```

Mocking Basics

Eso fue fácil. Ahora, ¿cómo modificamos el test? Recordemos que . . .

- El objeto testeado depende de una interfaz
- El mock implementa la interfaz
- Al objeto testeado no le "importa" cual usa
- Es similar a tener mas de una implementación



Un ejemplo, con 40% mas DI y su test

```
1  class MockDb : public Db {
2    void save_error(const char*) { std::cout << "Saving error\n"; }
3  };
4
5  TEST_F(HandshakeTest, TxOK) {
6    class Contestador_OK : public Contestador {
7      public: const char* get_msg(){ return MSG_2; }
8    };
9
10   ACE_Acceptor<Contestador_OK, ACE_SOCK_ACCEPTOR> acceptor(PORT);
11
12   MockDb db;
13   Handshake saludador(&db);
14   saludador.saludar();
15 }
16
17 TEST_F(HandshakeTest, RecvFail) {
18   class Contestador_Fail: public Contestador {
19     public: const char* get_msg(){ return "Garbage"; }
20   };
21
22   MockDb db;
23   ACE_Acceptor<Contestador_Fail, ACE_SOCK_ACCEPTOR> acceptor(PORT);
24   Handshake saludador(&db);
25   saludador.saludar();
26 }
```

Un ejemplo, con 40% mas DI y su test: Análisis

Eso funciona pero

- No está bueno tener que analizar los printfs
- No es automatizable
- Hay que mantener la implementación del mock

Entonces nos conviene usar gMock

- MOCK_METHODN(nombre, firma) $\Longrightarrow$ declara un método
- N es el número de argumentos
- El método debe ser virtual

Un ejemplo, con 40% mas DI y su test

```
1  class MockDb : public Db {
2    MOCK_METHOD1(save_error, void(const char*));
3  };
4
5  TEST_F(HandshakeTest, TxOK) {
6    class Contestador_OK : public Contestador {
7      public: const char* get_msg(){ return MSG_2; }
8    };
9
10   ACE_Acceptor<Contestador_OK, ACE_SOCK_ACCEPTOR> acceptor(PORT);
11
12   MockDb db;
13   Handshake saludador(&db);
14   saludador.saludar();
15 }
16
17 TEST_F(HandshakeTest, RecvFail) {
18   class Contestador_Fail: public Contestador {
19     public: const char* get_msg(){ return "Garbage"; }
20   };
21
22   MockDb db;
23   ACE_Acceptor<Contestador_Fail, ACE_SOCK_ACCEPTOR> acceptor(PORT);
24   Handshake saludador(&db);
25   saludador.saludar();
26 }
```

Definiendo comportamiento

Compila pero no hace nada, nos falta definir las expectations

- EXPECT_CALL es la forma mas básica
- Se escribe EXPECT_CALL(objeto , método)
- El método es un "matcher"
- El "matcher" matchea los argumentos
- Luego de EXPECT_CALL se define el comportamiento
- El comportamiento es lo que hará el mock al ser llamado

Definiendo comportamiento

Compila pero no hace nada, nos falta definir las expectations

- EXPECT_CALL es la forma mas básica
- Se escribe EXPECT_CALL(objeto , método)
- El método es un "matcher"
- El "matcher" matchea los argumentos
- Luego de EXPECT_CALL se define el comportamiento
- El comportamiento es lo que hará el mock al ser llamado

EXPECT_CALL(db, save_error(MATCHER))

Definiendo comportamiento: Matchers

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un MATCHER es una descripción del argumento esperado
- Existen matchers para enteros, cadenas, etc.
- Misma función y 2 matchers $\Longrightarrow$ expectations distintas

Definiendo comportamiento: Matchers

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un MATCHER es una descripción del argumento esperado
- Existen matchers para enteros, cadenas, etc.
  - Para matchear un escalar podríamos hacer

    EXPECT_CALL(foo, bar(3))
  - Para matchear un rango podríamos hacer

    EXPECT_CALL(foo, bar(Gt(3)))
  - Para matchear una cadena podríamos hacer

    EXPECT_CALL(foo, bar(StrEq("foobar")))
- Misma función y 2 matchers $\Longrightarrow$ expectations distintas

Definiendo comportamiento: Matchers

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un MATCHER es una descripción del argumento esperado
- Existen matchers para enteros, cadenas, etc.
  - Para matchear mas de una condición

    EXPECT_CALL(foo, bar(AllOf(Gt(3), Lt(10))))
  - Para matchear cualquier cosa

    EXPECT_CALL(foo, bar(_))
- Misma función y 2 matchers $\Longrightarrow$ expectations distintas

Definiendo comportamiento: Matchers

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un MATCHER es una descripción del argumento esperado
- Existen matchers para enteros, cadenas, etc.
- Misma función y 2 matchers $\Longrightarrow$ expectations distintas
  - Acá hay dos expectations

```
EXPECT_CALL( foo, bar(2) )
EXPECT_CALL( foo, bar(3) )
```

Para que esté OK hay que hacer

```
foo.bar(2); foo.bar(3);
```

  - Aunque esto también anda . . .

```
foo.bar(2); foo.bar(3); foo.bar(4);
```

Definiendo comportamiento: Matchers

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un MATCHER es una descripción del argumento esperado
- Existen matchers para enteros, cadenas, etc.
- Misma función y 2 matchers $\Longrightarrow$ expectations distintas
- Ya podemos escribir

```
EXPECT_CALL( db, save_error("recv") )
```

- Todavía nos falta definir las acciones!

Definiendo comportamiento: Acciones

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un ACTION define lo que hará un mock al ser llamado
- Por cada matcher hay al menos una acción
- Por cada acción hay al menos una cardinalidad

Definiendo comportamiento: Acciones

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un ACTION define lo que hará un mock al ser llamado
  - Por ejemplo:

    EXPECT_CALL(foo, bar(_)).WillOnce(Return(42));

    Es un matcher que retornará 42 la primer llamada.
    ¿Qué hará en llamadas subsiguientes?
- Por cada matcher hay al menos una acción
- Por cada acción hay al menos una cardinalidad

Definiendo comportamiento: Acciones

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un ACTION define lo que hará un mock al ser llamado
- Por cada matcher hay al menos una acción
  - Si no está definida es una acción por defecto, como la segunda llamada del caso anterior.
  - Para valores numéricos es 0
  - Para punteros es NULL
  - Para objectos dará un error y fallará el test
  - La acción por defecto puede ser cambiada
- Por cada acción hay al menos una cardinalidad

Definiendo comportamiento: Acciones

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un ACTION define lo que hará un mock al ser llamado
- Por cada matcher hay al menos una acción
- Por cada acción hay al menos una cardinalidad
  - Por defecto es 1
  - La macro Times define cardinalidad explícita

    EXPECT_CALL(foo, bar(_)).Times(42);
  - La cardinalidad queda implicita al definir más de una acción

    .WillOnce(Return(42)).WillOnce(Return(42));
  - Podemos definir una acción por defecto así:

    .WillRepeatedly(Return(42));

Definiendo comportamiento: Acciones

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Un ACTION define lo que hará un mock al ser llamado
- Por cada matcher hay al menos una acción
- Por cada acción hay al menos una cardinalidad
- Ya podemos escribir

```
EXPECT_CALL( db, save_error("recv") )
    .WillOnce( ACTION );
```

- Todavía nos falta definir la acción!

Definiendo comportamiento: Acciones II

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Lo más fácil es devolver un valor
- Aunque no es difícil hacer otra cosa
- O llamar a alguien para que haga algo mas

Definiendo comportamiento: Acciones II

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Lo más fácil es devolver un valor
  - También se puede devolver "nada"

```
EXPECT_CALL( db, save_error("recv") )
    .WillOnce(Return());
```

  - Las referencias son ezpecialez

```
EXPECT_CALL( foo, bar(_) )
    .WillOnce(ReturnRef(obj));
```

- Aunque no es difícil hacer otra cosa
- O llamar a alguien para que haga algo mas

Definiendo comportamiento: Acciones II

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Lo más fácil es devolver un valor
- Aunque no es difícil hacer otra cosa
  - Por ejemplo, asignar un valor

    .WillOnce(Assign(&var, VALUE))
  - O lanzar una excepción

    .WillOnce(Throw(Exception))
- O llamar a alguien para que haga algo mas

Definiendo comportamiento: Acciones II

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Lo más fácil es devolver un valor
- Aunque no es difícil hacer otra cosa
- O llamar a alguien para que haga algo mas
  - Por ejemplo, una función [1]

    .WillOnce(Invoke(f))
  - O a un objeto

    .WillOnce(Invoke(object_ptr, &class::method)

[1] Que debe tener la misma firma que el método

Definiendo comportamiento: Acciones II

RTFM @ Google C++ Mocking Framework Cheat Sheet

- Lo más fácil es devolver un valor
- Aunque no es difícil hacer otra cosa
- O llamar a alguien para que haga algo mas
- Ahora si, estamos listos, nuestra expectation va a ser

```
EXPECT_CALL( db, save_error("recv") )
.WillOnce( Return() );
```

Un ejemplo, con 40% mas DI y su test

```
1  class MockDb : public Db {
2    MOCK_METHOD1(save_error, void(const char*));
3  };
4
5  TEST_F(HandshakeTest, TxOK) {
6    class Contestador_OK : public Contestador {
7      public: const char* get_msg(){ return MSG_2; }
8    };
9
10   ACE_Acceptor<Contestador_OK, ACE_SOCK_ACCEPTOR> acceptor(PORT);
11
12   MockDb db;
13   EXPECT_CALL(db, error("recv")).Times(0);
14   Handshake saludador(&db);
15   saludador.saludar();
16 }
17
18 TEST_F(HandshakeTest, RecvFail) {
19   class Contestador_Fail: public Contestador {
20     public: const char* get_msg(){ return "Garbage"; }
21   };
22
23   MockDb db;
24   EXPECT_CALL(db, error("recv")).WillOnce(Return());
25   ACE_Acceptor<Contestador_Fail, ACE_SOCK_ACCEPTOR> acceptor(PORT);
26   Handshake saludador(&db);
27   saludador.saludar();
28 }
```

Outline

Un ejemplo, con 40% mas DI y su test

Está quedando un poco mejor pero todavía

- No llegamos a testear todas las ramas
- Dependemos de la implementación de socket
- El fixture ya no es necesario!

Arreglemos estos problemas y veamos como queda el test!

Un ejemplo, con 80% mas DI

```
1  class Handshake {
2    Db *db; ACE_SOCK_Stream sock;
3    Handshake(Db *db, ACE_SOCK_Stream *sock) : db(db), sock(sock) {}
4
5    bool saludar() {
6      Command cmd; uint32_t bt; ACE_Time_Value timeout(20);
7
8      ssize_t res = sock->send_n("Hello",
9                                 sizeof("Hello"), &timeout, &bytes);
10
11     if (res <= 0 || bytes < msg.size())
12       db->save_error("send");
13
14     res = sock->recv_n(cmd.buffer(),
15                        cmd.sizeof_buffer(), &timeout, &bytes);
16
17     if (res <= 0 || bytes < msg.size())
18       db->save_error("recv");
19
20     return true;
21   }
22 };
```

Un ejemplo, con 80% mas DI y su test

- Nuestro fixture desaparece
- Lo reemplazamos por un header con mocks

```
1  // Como el comportamiento se define por test
2  // estos mocks van a ser siempre iguales.
3  class MockSock : public ACE_SOCK_Stream {
4    public:
5    MOCK_METHOD4(send_n, ssize_t(void*, int, int, int*));
6    MOCK_METHOD4(recv_n, ssize_t(void*, int, int, int*));
7  };
8
9  class MockDb : public Db {
10   public:
11   MOCK_METHOD1(save_error, void(const char*));
12 };
13
14 // Digamos que estos son los mensajes
15 #define CMD1 "Hello"
16 #define CMD2 "Bye"
17
18 // Y que esto hace magia
19 ACTION_P(CopyCmd, cmd){ strcpy(arg0, cmd); }
```

Un ejemplo, con 80% mas DI y su test

```
1  TEST(HandshakeTest, TxOK) {
2    MockDb db; MockSock sock;
3
4    // Expect no errors
5    EXPECT_CALL(db, error("recv")).Times(0);
6    EXPECT_CALL(db, error("send")).Times(0);
7
8    // Do send and recv
9    EXPECT_CALL(sock, send_n(MSG, sizeof(MSG), _, _))
10     .WillOnce(DoAll(
11                 SetArgumentPointee<3>(sizeof(MSG)),
12                 Return(1)
13               ));
14
15   EXPECT_CALL(sock, recv_n(_, _, _, _))
16     .WillOnce(DoAll(
17                 SetArgumentPointee<3>(sizeof(MSG2)),
18                 WithArgs<0, 1>(CopyCmd(MSG2)),
19                 Return(1)
20               ));
21
22   Handshake saludador(&db, &sock);
23   EXPECT_TRUE(saludador.saludar());
24 }
```

Un ejemplo, con 80% mas DI y su test

```
1  TEST(Handshake, RecvFail) {
2    MockDb db; MockSock sock;
3
4    // Expect no errors
5    EXPECT_CALL(db, error("recv")).Times(1);
6    EXPECT_CALL(db, error("send")).Times(0);
7
8    // Do send and recv
9    EXPECT_CALL(sock, send_n(MSG, sizeof(MSG), _, _))
10     .WillOnce(DoAll(
11                 SetArgumentPointee<3>(sizeof(MSG)),
12                 Return(1)
13               ));
14
15   EXPECT_CALL(sock, recv_n(_, _, _, _))
16     .WillOnce(DoAll(
17                 SetArgumentPointee<3>(0),
18                 Return(0)
19               ));
20
21   Handshake saludador(&db, &sock);
22   EXPECT_FALSE(saludador.saludar());
23 }
```

Un ejemplo, con 80% mas DI y su test

Ahora podemos agregar este test:

```
1  TEST(Handshake, SendFail) {
2    MockDb db; MockSock sock;
3
4    // Expect no errors
5    EXPECT_CALL(db, error("recv")).Times(0);
6    EXPECT_CALL(db, error("send")).Times(1);
7
8    // Do send and recv
9    EXPECT_CALL(sock, send_n(MSG, sizeof(MSG), _, _))
10     .WillOnce(DoAll(
11                 SetArgumentPointee<3>(0),
12                 Return(0)
13               ));
14
15   EXPECT_CALL(sock, recv_n(_, _, _, _))
16     .Times(0);
17
18   Handshake saludador(&db, &sock);
19   EXPECT_FALSE(saludador.saludar());
20 }
```

Dos bugs nuevos

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from Handshake
[ RUN      ] Handshake.TxOK
[       OK ] Handshake.TxOK
[ RUN      ] Handshake.RecvFail
test5.cpp:10: Failure
Value of: saludador.saludar()
  Actual: true
Expected: false
[  FAILED  ] Handshake.RecvFail
[ RUN      ] Handshake.SendFail
test5.cpp:13: Failure
Mock function called more times than expected - returning default value.
    Function call: recv_n(0xbff360f8, 0, 0, 0xbff360ec)
          Returns: 0
         Expected: to be never called
           Actual: called once - over-saturated and active
test5.cpp:16: Failure
Value of: saludador.saludar()
  Actual: true
Expected: false
[  FAILED  ] Handshake.SendFail
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 1 test.
[  FAILED  ] 2 tests, listed below:
[  FAILED  ] Handshake.RecvFail
[  FAILED  ] Handshake.SendFail

 2 FAILED TESTS
```

Ejemplo arreglado

```
1  class Handshake {
2    Db *db; ACE_SOCK_Stream *sock;
3    Handshake(Db *db, ACE_SOCK_Stream *sock) : db(db), sock(sock) {}
4
5    bool saludar() {
6      Command cmd; uint32_t bt; ACE_Time_Value timeout(20);
7
8      ssize_t res = sock->send_n("Hello",
9                           sizeof("Hello"), &timeout, &bytes);
10
11     if (res <= 0 || bytes < msg.size()) {
12       db->save_error("send");
13       return false;
14     }
15
16     res = sock->recv_n(cmd.buffer(),
17                  cmd.sizeof_buffer(), &timeout, &bytes);
18
19     if (res <= 0 || bytes < msg.size()) {
20       db->save_error("recv");
21       return false;
22     }
23
24     return true;
25   }
26 };
```

Ejemplo arreglado: resultados

```
Running main() from gtest_main.cc
[==========] Running 3 tests from 1 test case.
[----------] Global test environment set-up.
[----------] 3 tests from Handshake
[ RUN      ] Handshake.TxOK
[       OK ] Handshake.TxOK
[ RUN      ] Handshake.RecvFail
[       OK ] Handshake.RecvFail
[ RUN      ] Handshake.SendFail
[       OK ] Handshake.SendFail
[----------] Global test environment tear-down
[==========] 3 tests from 1 test case ran.
[  PASSED  ] 3 test.
```

Outline

Algo extraño...

Supongamos un código perfectamente inyectable como éste:

```
1  class Diccionario { public:
2    const char* saludo(){ return "Hola mundo"; }
3  };
4
5  class Saludador { public:
6    Diccionario *dict;
7    Saludador(Diccionario *dict): dict(dict) {}
8
9    const char* saludar(){ return dict->saludo(); }
10 };
```

Algo extraño...

Y un test simple para el código perfectamente inyectable:

```
1  class MockDict : public Diccionario {
2    public:
3    MOCK_METHOD_0(saludo, const char*());
4  };
5
6  TEST(SaludadorTest, saludar) {
7    MockDict dict;
8
9    EXPECT_CALL(dict, saludo())
10     .WillOnce(Return("ciao mondo"));
11
12   Saludador s(&dict);
13   s.saludar();
14 }
```

¿Por qué falló?

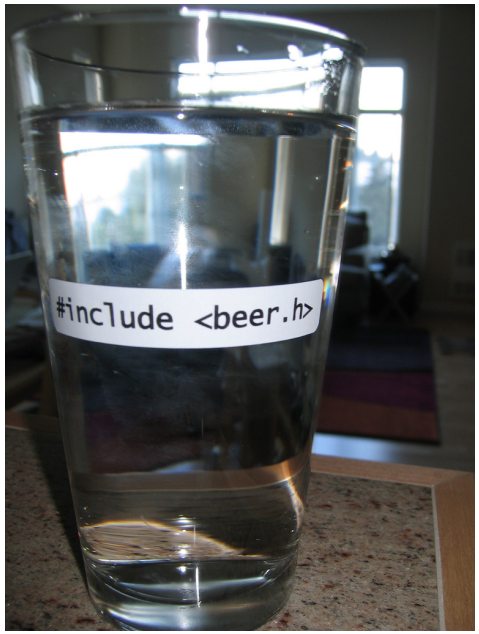
```
Running main() from gtest_main.cc
[==========] Running 1 test from 1 test case.
[-------] Global test environment set-up.
[-------] 1 test from SaludadorTest
[ RUN      ] SaludadorTest.saludar
test6.cpp:24: Failure
Actual function call count doesn't match this expectation.
         Expected: to be called once
           Actual: never called - unsatisfied and active
[  FAILED  ] SaludadorTest.saludar
[-------] Global test environment tear-down
[==========] 1 test from 1 test case ran.
[  PASSED  ] 0 tests.
[  FAILED  ] 1 test, listed below:
[  FAILED  ] SaludadorTest.saludar

 1 FAILED TEST
```

¿Cómo lo arreglamos?

No llamó a nuestro mock por no ser virtual.

En C++ tenemos distintas formas de solucionar este problema

- Virtual method
- Pseudo virtual method
- Adapter / Proxy
- Template



Virtual method

Lo más fácil es cambiar

```
const char* saludo(){ return "Hola mundo"; }
```

por

```
virtual const char* saludo(){ return "Hola mundo"; }
```

Pero no siempre se puede

- A veces el código no es nuestro
- Los femtosegundos que introduce la vtable son muchos
- No se quiere brindar un punto de extensión en la clase

Pseudo virtual method

También se puede cambiar

```
const char* saludo(){ return "Hola mundo"; }
```

por

```
VIRTUAL_TEST const char* saludo(){ return "Hola mundo"; }
```

Donde VIRTUAL_TEST es un flag de compilación.

Pero no siempre se puede

- A veces el código no es nuestro
- No queremos tocar el makefile

Adapter / Proxy

Recordar s/Diccionario/DictProxy/g en el test

```
1  // Codigo externo
2  class Diccionario { public:
3    const char* saludo(){ return "Hola mundo"; }
4  };
5
6  class DictProxy { public:
7          virtual const char* saludo() {
8                  Diccionario d;
9                  return d.saludo();
10         }
11 };
12
13 class Saludador { public:
14   DictProxy *dict;
15   Saludador(DictProxy *dict): dict(dict) {}
16
17   const char* saludar(){ return dict->saludo(); }
18 };
```

Template

```
1  class Diccionario { public:
2    const char* saludo(){ return "Hola mundo"; }
3  };
4
5  template <class T>
6  class Saludador { public:
7    T *dict;
8    Saludador(T *dict): dict(dict) {}
9
10   const char* saludar(){ return dict->saludo(); }
11 };
```

Template

```
1  class MockDict {
2    public:
3    MOCK_METHOD_0(saludo, const char*());
4  };
5
6  TEST(SaludadorTest, saludar) {
7    MockDict dict;
8
9    EXPECT_CALL(dict, saludo())
10     .WillOnce(Return("ciao mondo"));
11
12   Saludador<MockDict> s(&dict);
13   s.saludar();
14 }
```

¿Preguntas?

¿Seguro?

Está imagen me gusta pero no quedó lugar para meterla