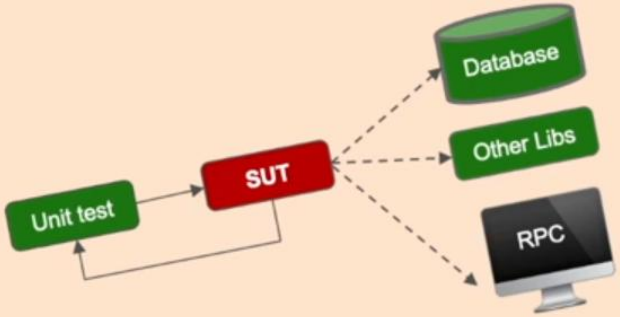



```
class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

Mock Class

```
TEST(AtmMachine, CanWithdrawForceSequenceOnSomeCalls) {

  // Expectations
  InSequence seq;
  EXPECT_CALL(mock_bankserver, Connect()).Times(1);
  EXPECT_CALL(mock_bankserver,
    GetBalance(_))
     .Times(1)
     .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);
}
```

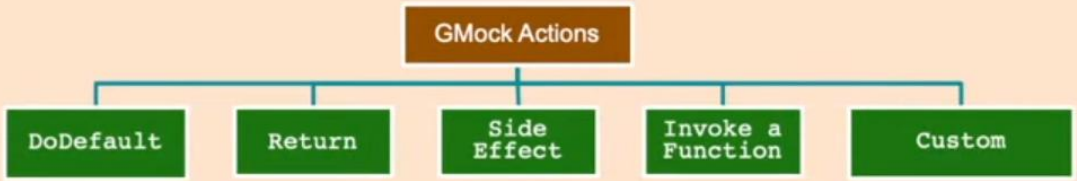
GMock Test Structure

```
ON_CALL(mock_object, method_name(matchers...))
.With(multi_argument_matcher) // Can be used at most once
.WillByDefault(action); // Required
```

ON_CALL

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

EXPECT_CALL



https://www.youtube.com/watch?v=vxCH4AuVRgo

What Are We Covering?

- Google **Test**
  - Google Test Macros
  - Test **Fixtures**
  - Manipulating Tests
  - Advanced Topics
    - Sharing Test Logic
    - Typed Test
    - Type-Parameterized Test
    - Assertion in Subroutines
    - Testing Private Code

- Google **Mock**
  - Matchers
  - **EXPECT_THAT**
  - Mocking **Classes** and **Functions**
    - **EXPECT_CALL**
    - **ON_CALL**
    - **GMock Actions**
  - Advanced Topics
    - Forcing **Sequence**
    - **Checkpoints**
    - **Delegation**
    - GMock **Custom Actions**

https://www.youtube.com/watch?v=vxCH4AuVRgo

Prerequisite: Installing Bazel

This repo uses `Bazel` for building C++ files. You can install Bazel using this link.

Cloning this repo

```
git clone https://github.com/ourarash/cpp-template.git
```

Examples:

Hello World Example:

You can run this using `bazel`:

```
bazel run src/main:main
```

https://github.com/ourarash/cpp-template

Settings

https://www.youtube.com/watch?v=vxCH4AuVRgo

Who Needs GMock?

I work in the **industry**

✓

I'm a **student**

✓

I'm studying for **job interviews**

✓

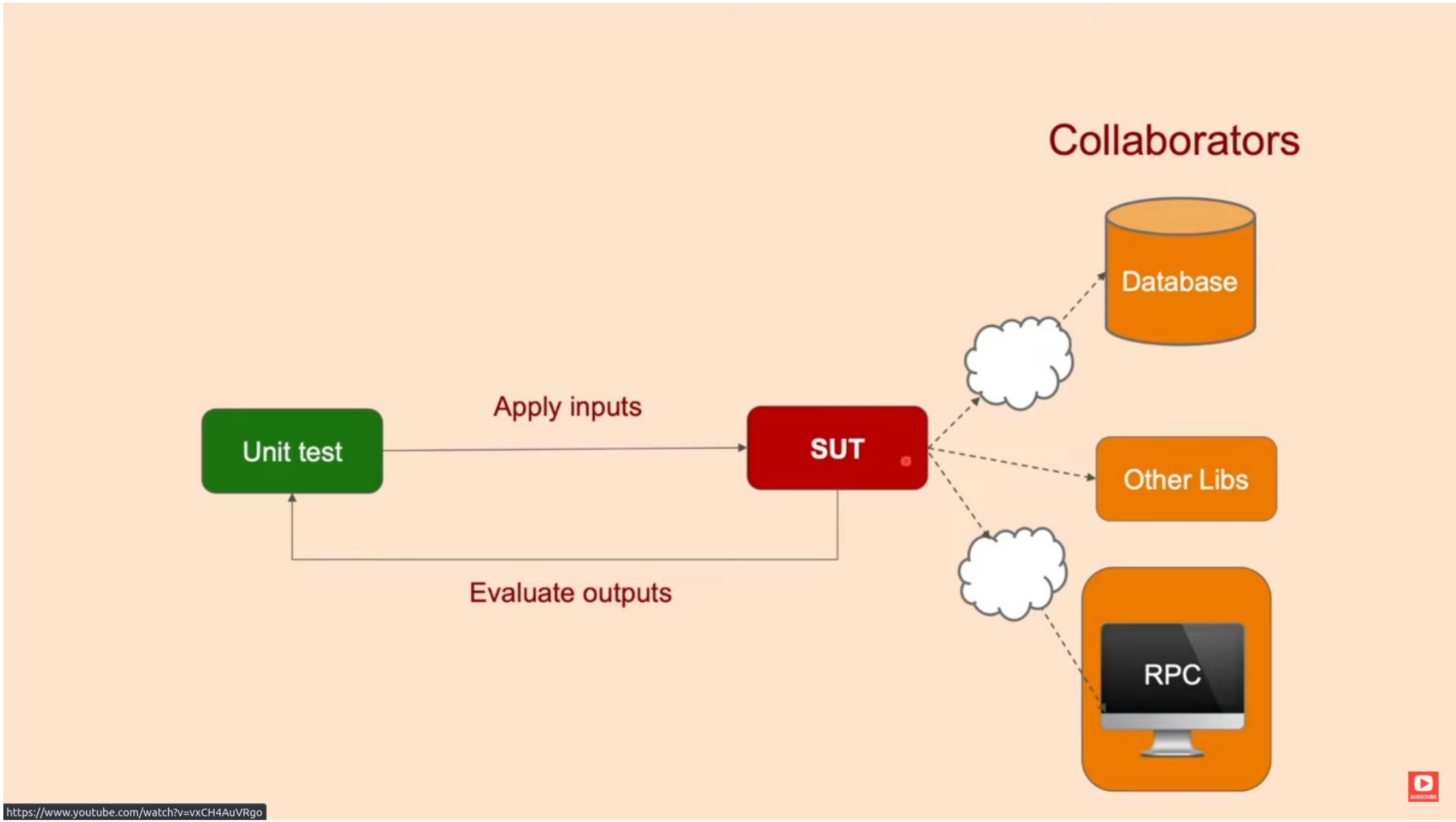
```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver,
          GetBalance(account_number))
          .WillRepeatedly(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(
          account_number, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

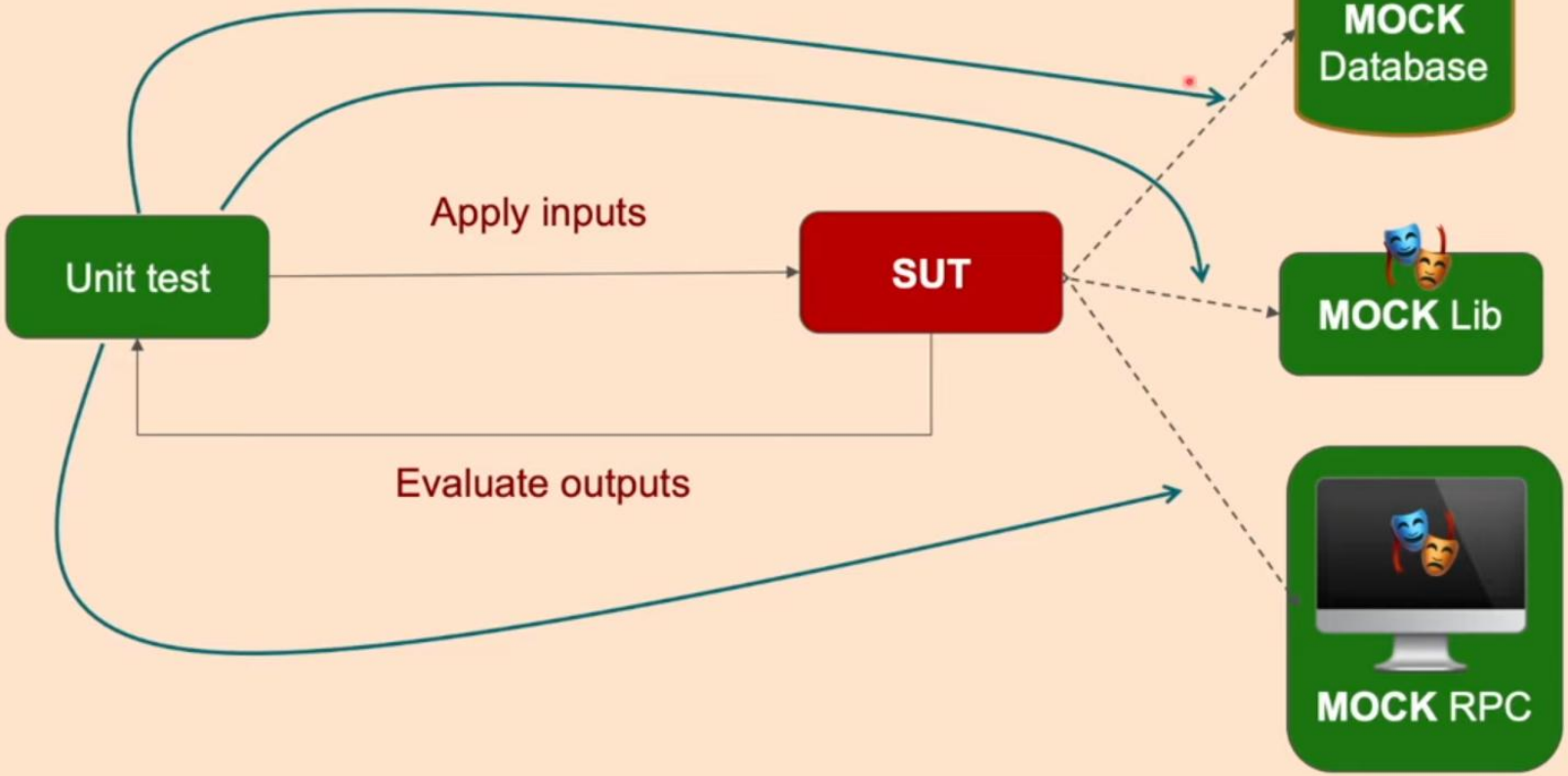
https://www.youtube.com/watch?v=vxCH4AuVRgo

Google Mock

Mocking Classes and Functions

Settings
SUBSCRIBE
https://www.youtube.com/watch?v=vxCH4AuVRgo


Collaborators
Database
Apply inputs
Unit test
SUT
Other Libs
Evaluate outputs
RPC
https://www.youtube.com/watch?v=vxCH4AuVRgo

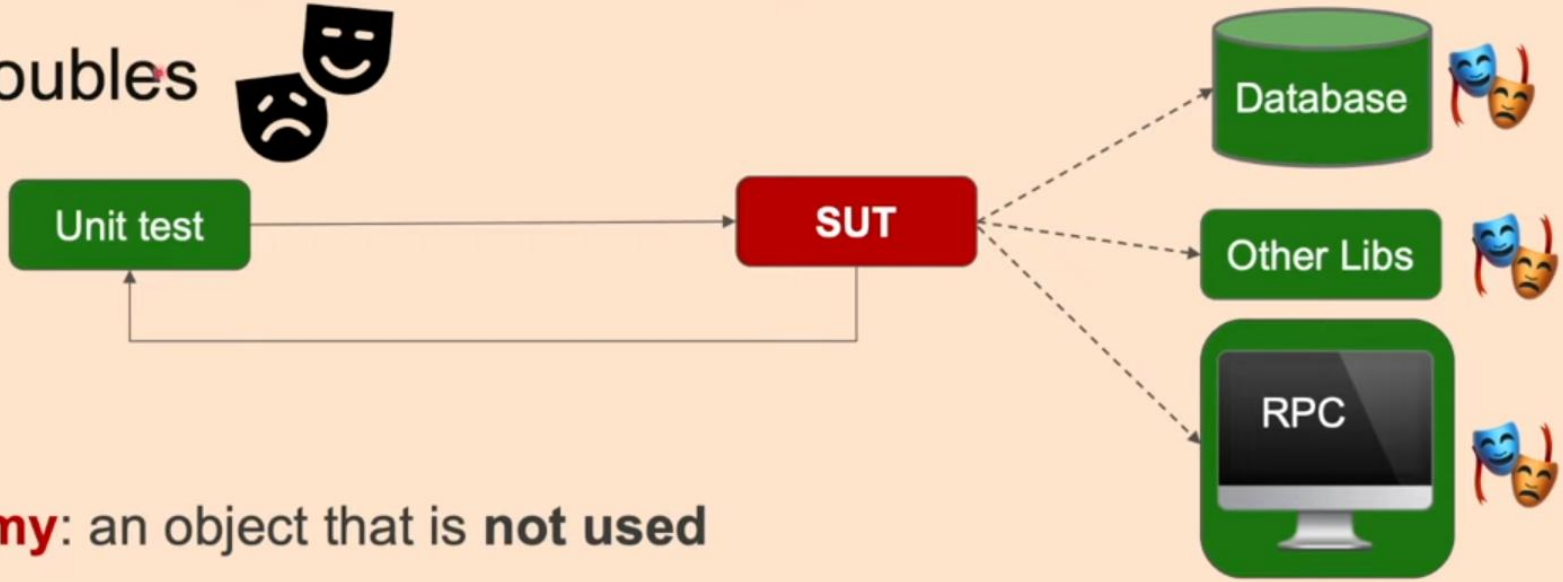


Settings
https://www.youtube.com/watch?v=vxCH4AuVRgo

Settings

https://www.youtube.com/watch?v=vxCH4AuVRgo

Test Doubles

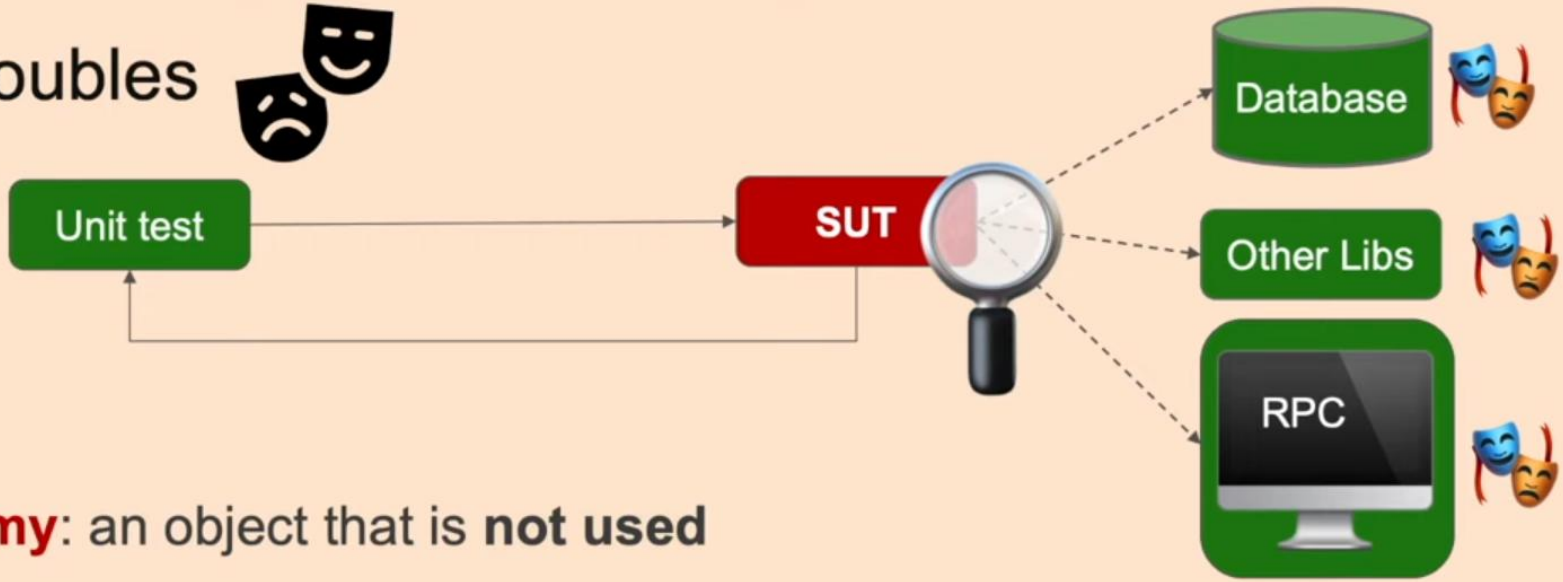


Dummy: an object that is **not used**

State Verification	Behavior Verification
Fake: a **simplified** implementation	**Mock:** An object to which we can **specify**:
Stub: returns **canned** values	- **Which** of its **methods** should be **called**
Spy: a stub that **records** info	- **How** each method should be **called**
	- **What** each method should **return**

Settings
https://www.youtube.com/watch?v=vxCH4AuVRgo

Test Doubles



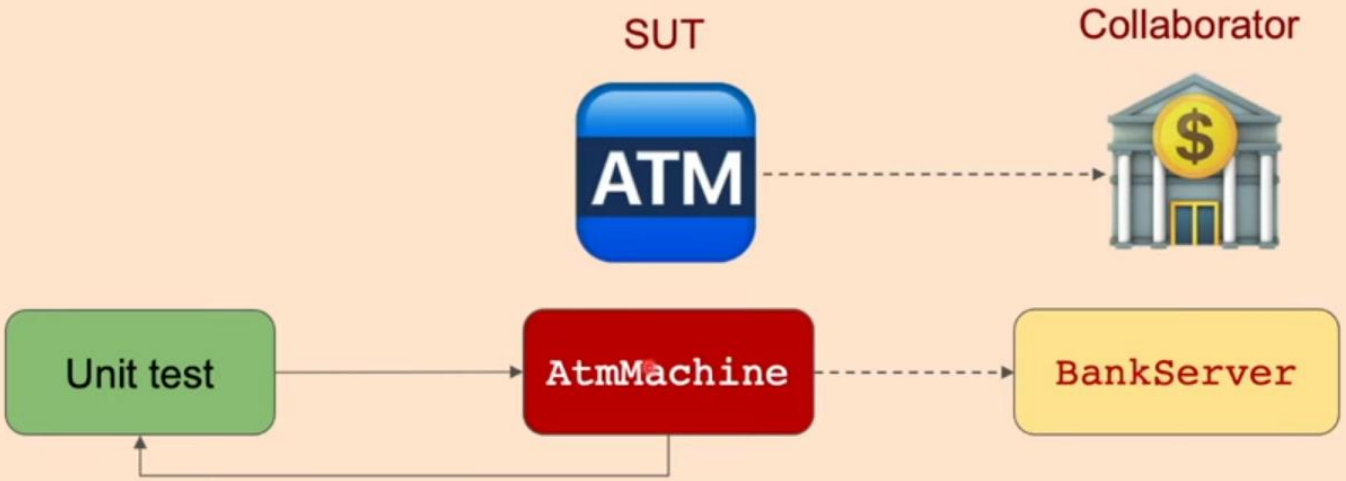
Dummy: an object that is **not used**

State Verification	Behavior Verification
Fake: a **simplified** implementation	**Mock:** An object to which we can **specify**: - **Which** of its **methods** should be **called** - **How** each method should be **called** - **What** each method should **return**
Stub: returns **canned** values	
Spy: a stub that **records** info	

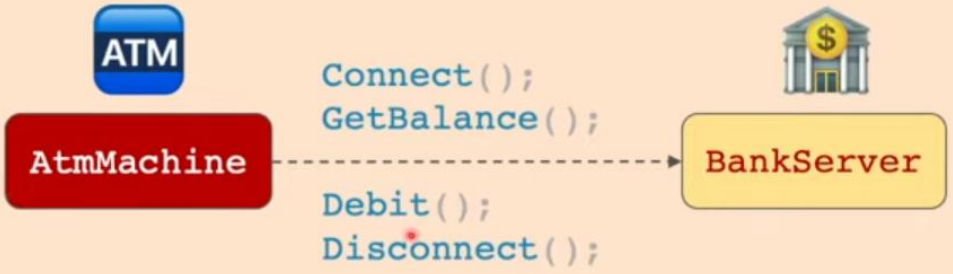
Settings
https://www.youtube.com/watch?v=vxCH4AuVRgo

Example:

Testing class `AtmMachine` which depends on class `BankServer`.



Settings
7:07 / 1:17:34 • Example Test with Doubles ›
https://www.youtube.com/watch?v=vxCH4AuVRgo




```
atm_machine.Withdraw():

1. bankServer_->Connect();
2. bankServer_->GetBalance();
3. bankServer_->Debit();
4. bankServer_->Disconnect();
```

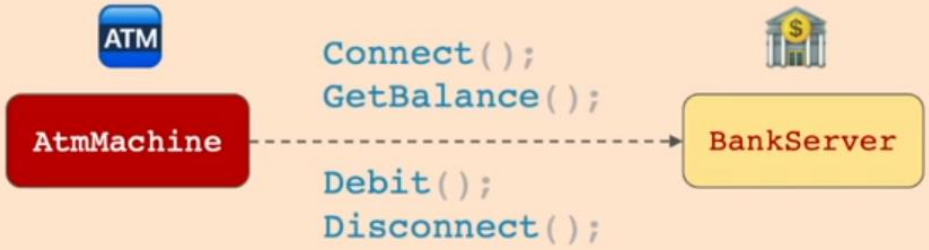
Settings
https://www.youtube.com/watch?v=vxCH4AuVRgo

```cpp
#include <bankserver.h>
class AtmMachine {
BankServer* bankServer_;

public:
  AtmMachine(BankServer* bankServer) :
                          bankServer_(bankServer) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;

    bankServer_->Connect();

    auto available_balance = bankServer_->
        GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      result = true;
    }

    bankServer_->Disconnect();
    return result;
  }
};
```

atm_machine.h

```cpp
class BankServer {
public:
 virtual ~BankServer() {}
 virtual void Connect() = 0;
 virtual void Disconnect() = 0;
 virtual void Credit(int account_number, int value) = 0;
 virtual void Debit(int account_number, int value) = 0;
 virtual bool DoubleTransaction(int account_number,
         int value1, int value2) = 0;
 virtual int GetBalance(int account_number) const = 0;
};
```

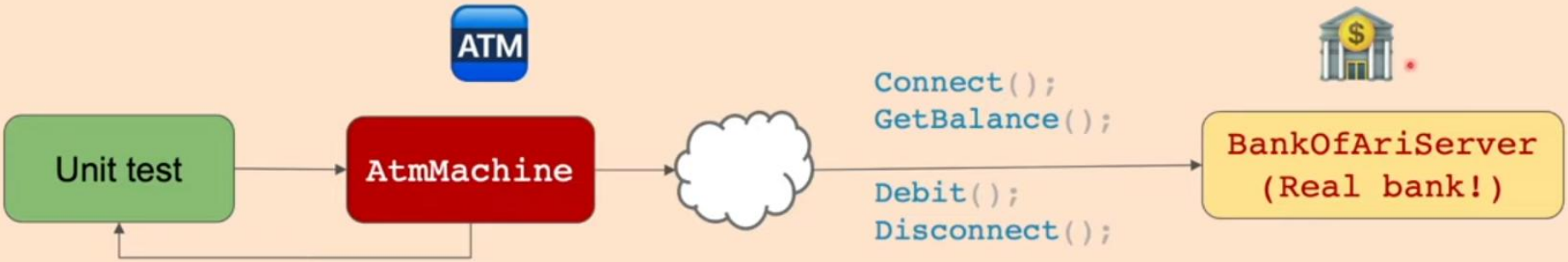
bankserver.h



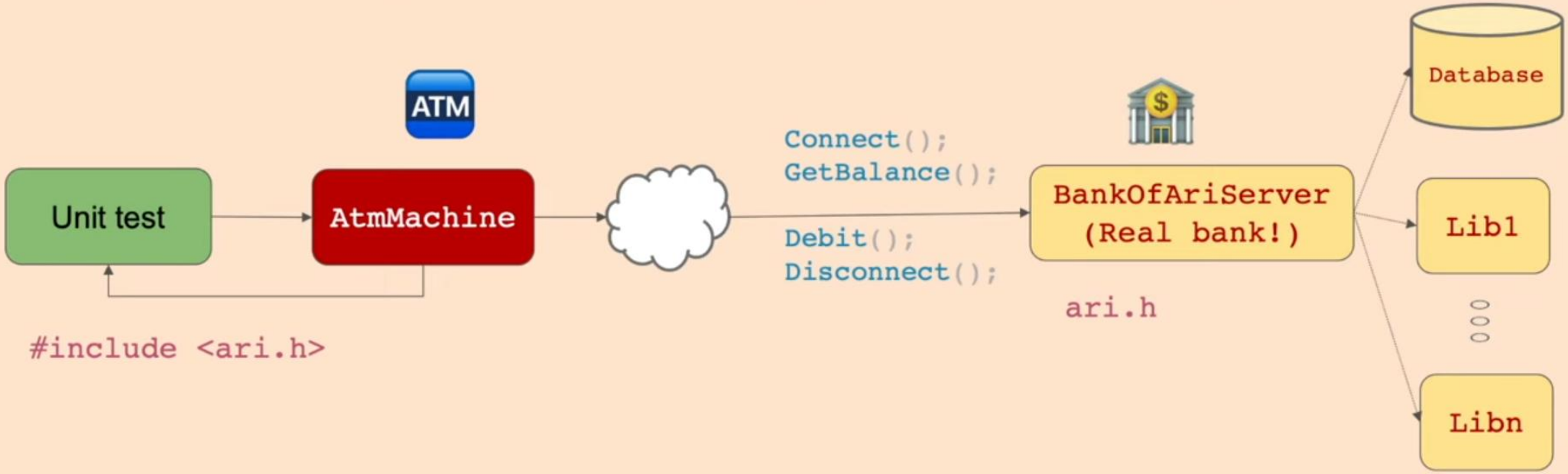

```
atm_machine.Withdraw():

1. bankServer_->Connect();
2. bankServer_->GetBalance();
3. bankServer_->Debit();
4. bankServer_->Disconnect();
```

Settings
https://www.youtube.com/watch?v=vxCH4AuVRgo

Scenario: The ATM machine **can withdraw** correctly when the **current balance is high enough**.



https://www.youtube.com/watch?v=vxCH4AuVRgo



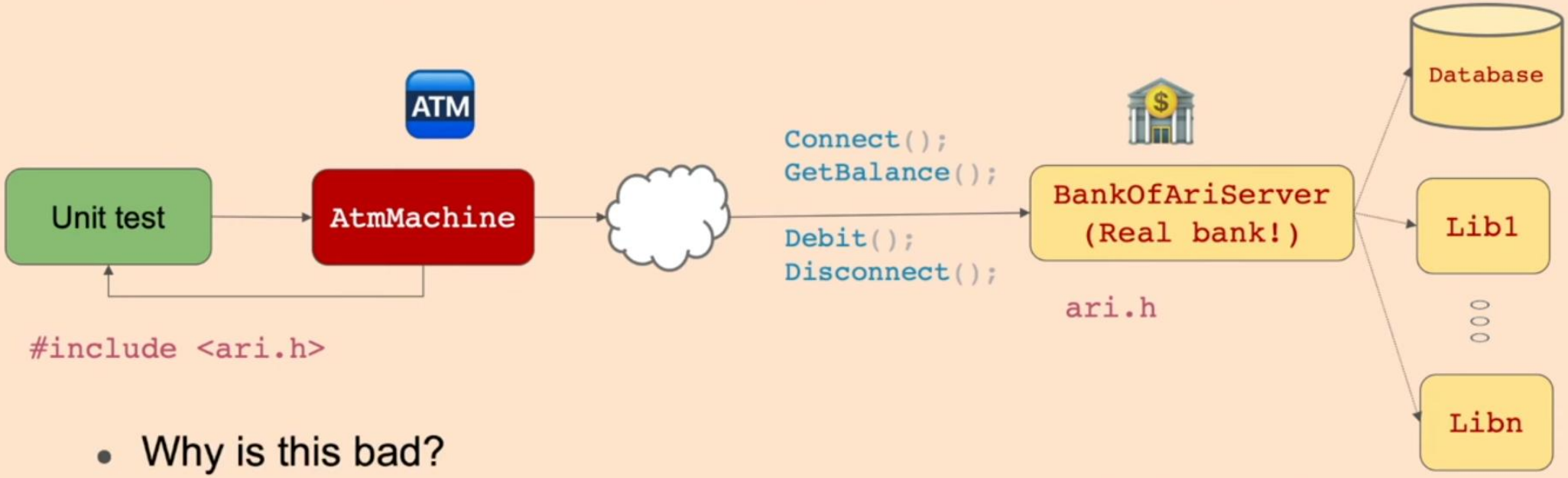

```
#include <ari.h>

TEST(AtmMachine, CanWithdraw) {

  BankOfAriServer bankserver;


  AtmMachine atm_machine(&bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  EXPECT_TRUE(withdraw_result);
}
```

BAD

```
#include <complicated_dependencies.h>

class BankOfAriServer : public BankServer {
 // Actual implementation of APIs
 // Lots of dependencies!
};
```

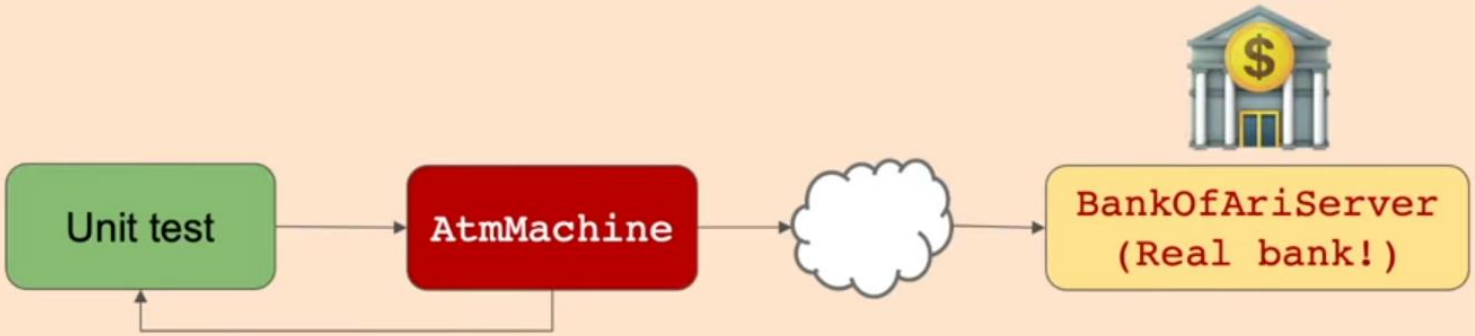
ari.h

https://www.youtube.com/watch?v=vxCH4AuVRgo



- Why is this bad?
  - We must include **`ari.h`** and link to **`ari.cc`**.
  - The **network connection** should be up and running
  - My bank account will be **manipulated** for each test.
  - The **API calls are real**. Might be expensive and calling them might have undesirable consequences.

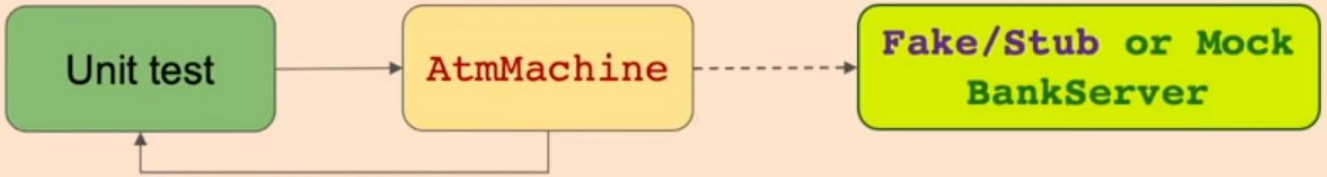
https://www.youtube.com/watch?v=vxCH4AuVRgo



https://www.youtube.com/watch?v=vxCH4AuVRgo

Solution: Use a Test Double

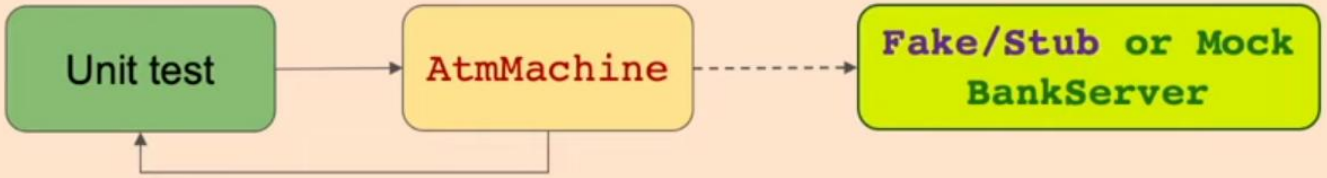
- Fake, Stub
- Mocks



https://www.youtube.com/watch?v=vxCH4AuVRgo

Solution: Use a Test Double

- Fake, Stub
- Mocks



https://www.youtube.com/watch?v=vxCH4AuVRgo

Pass a **mock** `BankServer` class to `AtmMachine` class

```
class BankServer {
public:
 virtual ~BankServer() {}
 virtual void Connect() = 0;
 virtual void Disconnect() = 0;
 virtual void Credit(int account_number, int value) = 0;
 virtual void Debit(int account_number, int value) = 0;
 virtual bool DoubleTransaction(int account_number,
             int value1, int value2) = 0;
 virtual int GetBalance(int account_number) const = 0;
};
```

```
#include <bankserver.h>
class AtmMachine {
BankServer* bankServer_;

public:
  AtmMachine(BankServer* bankServer) :
                         bankServer_(bankServer) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;

    bankServer_->Connect();

    auto available_balance = bankServer_->
          GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      result = true;
    }

    bankServer_->Disconnect();
    return result;
  }
};
```

ATM

Production

Test

https://www.youtube.com/watch?v=vxCH4AuVRgo

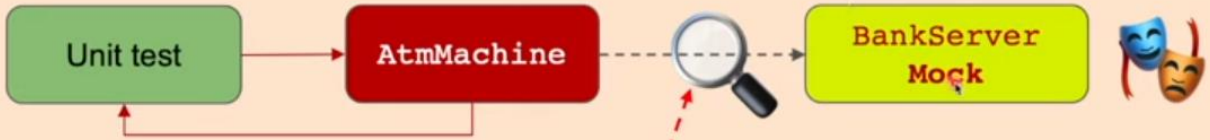
Using Fake/Stub/Spy

```
atm_machine.Withdraw(1234, 1000):

1. bankServer_->Connect();
2. bankServer_->GetBalance(1234);
3. bankServer_->Debit(1234, 1000);
4. bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdraw) {

  StubBankServer stub_bankserver;
  // Possibly set up the stub_bankserver.

  AtmMachine atm_machine(&stub_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  EXPECT_TRUE(withdraw_result);
  // Possibly check the status of stub_bankserver.
}
```

https://www.youtube.com/watch?v=vxCH4AuVRgo

Using a Mock




```
atm_machine.Withdraw(1234, 1000):

1. bankServer_->Connect();
2. bankServer_->GetBalance(1234);
3. bankServer_->Debit(1234, 1000);
4. bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdraw) {
  // Expectations:

     Connect() is called exactly once.
     GetBalance(1234) is called exactly once for which return 2000.
     Debit(1234, 1000) is called exactly once.
     Disconnect() is called exactly once.

  // Assert the following:
      The return value of atm_machine.Withdraw(1234, 1000) is true.
}
```

https://www.youtube.com/watch?v=vxCH4AuVRgo

```
TEST(AtmMachine, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect()).Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
       .Times(1)
       .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(account_number, withdraw_value))
       .Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect()).Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
TEST(AtmMachine, CanWithdraw) {
  // Expectations:

        Connect() is called exactly once.
        GetBalance(1234) is called exactly once for which return 2000.
        Debit(1234, 1000) is called exactly once.
        Disconnect() is called exactly once.

  // Assert the following:
        The return value of atm_machine.Withdraw(1234, 1000) is true.
}
```

Using Google Mock

https://www.youtube.com/watch?v=vxCH4AuVRgo

Pass a **mock** `BankServer` class to `AtmMachine` class

```
class BankServer {
public:
 virtual ~BankServer() {}
 virtual void Connect() = 0;
 virtual void Disconnect() = 0;
 virtual void Credit(int account_number, int value) = 0;
 virtual void Debit(int account_number, int value) = 0;
 virtual bool DoubleTransaction(int account_number,
             int value1, int value2) = 0;
 virtual int GetBalance(int account_number) const = 0;
};
```

```
#include <bankserver.h>
class AtmMachine {
BankServer* bankServer_;

public:
  AtmMachine(BankServer* bankServer) :
                         bankServer_(bankServer) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;

    bankServer_->Connect();

    auto available_balance = bankServer_->
          GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      result = true;
    }

    bankServer_->Disconnect();
    return result;
  }
};
```

Production

Test

https://www.youtube.com/watch?v=vxCH4AuVRgo

Our First Test With GMock

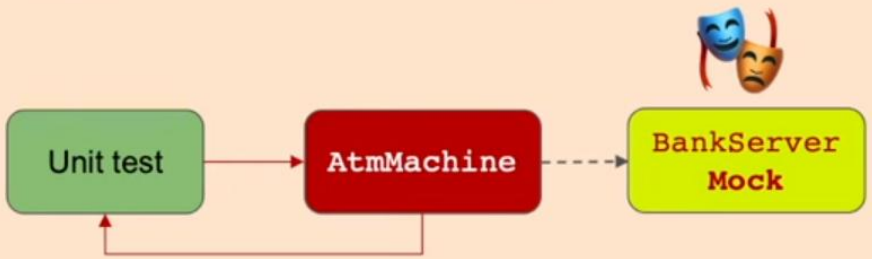
```
class BankServer {
public:
  virtual ~BankServer() {}
  virtual void Connect() = 0;
  virtual void Disconnect() = 0;
  virtual void Credit(int account_number, int value) = 0;
  virtual void Withdraw(int account_number, int value) = 0;
  virtual bool DoubleTransaction(int account_number,
          int value1, int value2) = 0;
  virtual int GetBalance(int account_number) const = 0;
};
```

bankserver.h

```
#include "bankserver.h"
#include "gmock/gmock.h"

class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(bool, DoubleTransaction, (int, int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

mock_bankserver.h

```
#include "mock_bankserver.h"

TEST(AtmMachine, CanWithdraw) {
  // Arrange
  MockBankServer bankserver;

  // Expectations
  // ...

  AtmMachine atm_machine(&bankserver);

  // Act
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```



tests/gmock_demo/first_example.cc

https://www.youtube.com/watch?v=vxCH4AuVRgo

The **MOCK_METHOD** Macro

```
MOCK_METHOD(ReturnType, MethodName, (Args...));
MOCK_METHOD(ReturnType, MethodName, (Args...), (Specs...));
```

```
#include "bankserver.h"
#include "gmock/gmock.h"

class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(bool, DoubleTransaction, (int, int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

mock_bankserver.h

- **const** – Required if overriding a const method.
- **override** – Recommended if overriding a virtual method.
- **noexcept** – Required if overriding a noexcept method.

Note: Macros are limited

```
class MyMock {
public:
  MOCK_METHOD(ReturnType, MethodName, (Args...));
  MOCK_METHOD(ReturnType, MethodName, (Args...), (Specs...));
};
```

```
class MockFoo {
public:
  MOCK_METHOD(std::pair<bool, int>, GetPair, ()); // Won't compile!
  MOCK_METHOD(bool, CheckMap, (std::map<int, double>, bool)); // Won't compile!
};
```

Solution 1: Use **parenthesis**

```
class MockFoo {
public:
  MOCK_METHOD((std::pair<bool, int>), GetPair, ());
  MOCK_METHOD(bool, CheckMap, ((std::map<int, double>), bool));
};
```

Solution 2: Use **alias**

```
class MockFoo {
public:
  using BoolAndInt = std::pair<bool, int>;
  MOCK_METHOD(BoolAndInt, GetPair, ());

  using MapIntDouble = std::map<int, double>;
  MOCK_METHOD(bool, CheckMap, (MapIntDouble, bool));
};
```

```
class BankServer {
public:
  virtual ~BankServer() {}
  virtual void Connect() = 0;
  virtual void Disconnect() = 0;
  virtual void Credit(int account_number, int value) = 0;
  virtual void Debit(int account_number, int value) = 0;
  virtual bool DoubleTransaction(int account_number,
          int value1, int value2) = 0;
  virtual int GetBalance(int account_number) const = 0;
};
```

```
#include "mock_bankserver.h"

TEST(AtmMachine, CanWithdraw) {
  // Arrange
  MockBankServer bankserver;

  // Expectations
  // ...

  AtmMachine atm_machine(&bankserver);

  // Act
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

```
#include "bankserver.h"
#include "gmock/gmock.h"

class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(bool, DoubleTransaction, (int, int, int),
          (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

Specifying What Should the Mock Do.

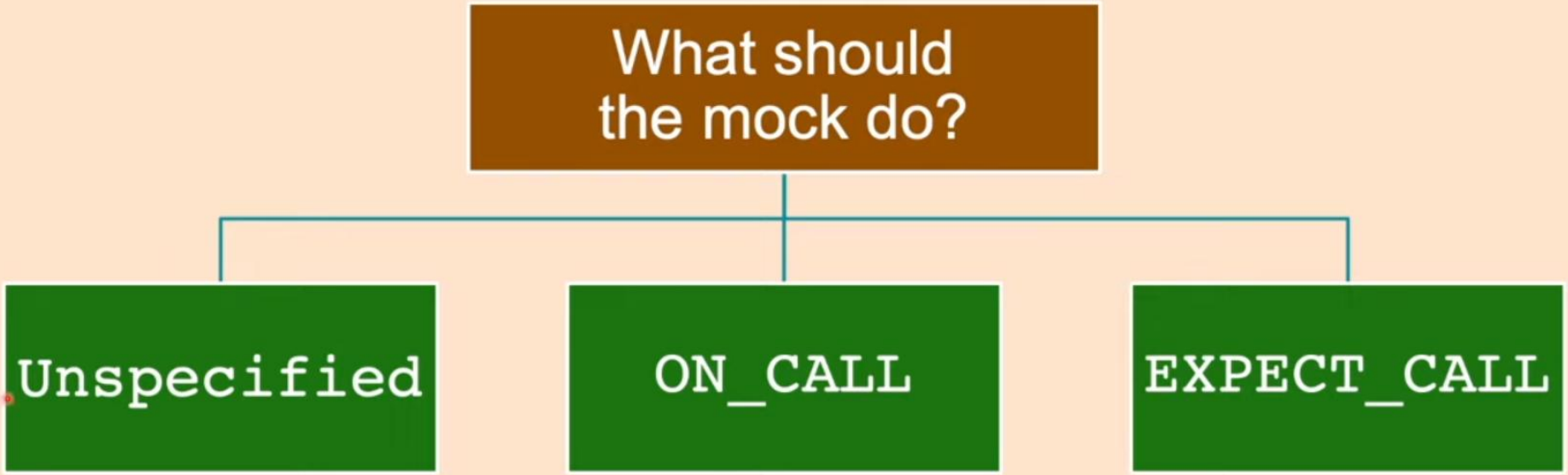
```
#include "bankserver.h"
#include "gmock/gmock.h"

class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(bool, DoubleTransaction, (int, int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

```cpp
#include <bankserver.h>
class AtmMachine {
BankServer* bankServer_;

public:
  AtmMachine(BankServer* bankServer) :
                          bankServer_(bankServer) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;

    bankServer_->Connect();

    auto available_balance = bankServer_->
          GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      result = true;
    }

    bankServer_->Disconnect();
    return result;
  }
};
```

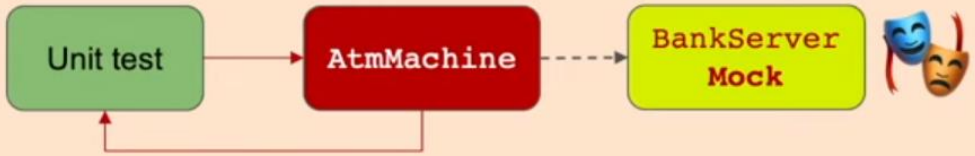
```cpp
#include "mock_bankserver.h"

TEST(AtmMachine, CanWithdraw) {
  // Arrange
  MockBankServer bankserver;

  // Expectations
  // ...

  AtmMachine atm_machine(&bankserver);

  // Act
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

What should the mock do?

Unit test
AtmMachine
Connect();
GetBalance();
Debit();
Disconnect();
BankServer
Mock


What should the mock do?
Unspecified
ON_CALL
EXPECT_CALL

GMock Test Structure




```
atm_machine.Withdraw(1234, 1000):

1. bankServer_->Connect();
2. bankServer_->GetBalance(1234);
3. bankServer_->Debit(1234, 1000);
4. bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // What should the Mock do?
  // Unspecified or ON_CALL or EXPECT_CALL

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
      atm_machine.Withdraw(account_number,
                           withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

What should the mock do?

Exercise some code

Check the result

```
atm_machine.Withdraw(1234, 1000):

1.  bankServer_->Connect();
2.  bankServer_->GetBalance(1234);
3.  bankServer_->Debit(1234, 1000);
4.  bankServer_->Disconnect();
```

What should the mock do?

Unspecified

- **Built-in** types:
  - `void: return()`
  - `bool: return(false)`
  - `all other: return (0)`
- **User defined** types:
  - the **default constructor**: `return UDT();`

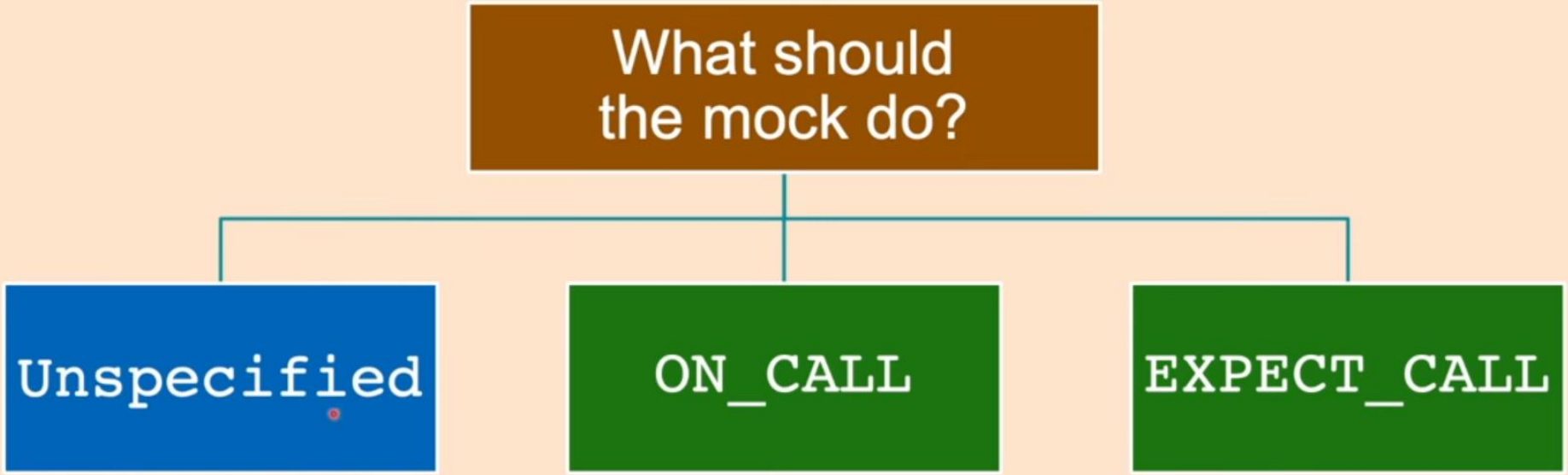
ON_CALL

- Each function call can be **specified separately**.
- **Different actions** based on the value of its **parameters**.
  - `f1(1, 2)` → `Action1`
  - `f1(2, 2)` → `Action2`
  - `f2(1)` → `Action3`
  - `f2(2)` → `Action4`
- The calls will **not be verified**!

EXPECT_CALL

- Like **ON_CALL**, but:
  - The calls **will be verified**!
- May **other things** can be specified, e.g.:
  - Order of calls
  - Number of calls

Unspecified Mocks



Unspecified Mock

```
class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(bool, DoubleTransaction, (int, int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

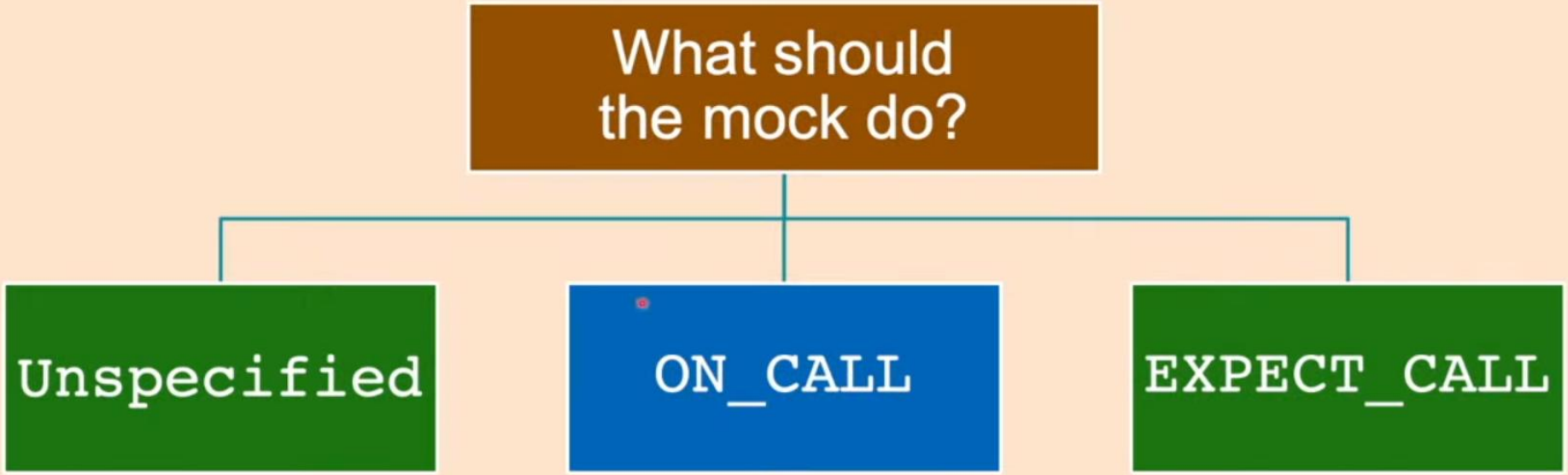
```
atm_machine.Withdraw(1234, 1000):

1. bankServer_->Connect();
2. bankServer_->GetBalance(1234);
3. bankServer_->Debit(1234, 1000);
4. bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdrawSimpleUnspecified) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // We do not specify what should happen!

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
       withdraw_value);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

return (0)

tests/gmock_demo/unspecified.cc

Specifying with the ON_CALL Macro



```
atm_machine.Withdraw(1234, 1000):

1. bankServer_->Connect();
2. bankServer_->GetBalance(1234);
3. bankServer_->Debit(1234, 1000);
4. bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdrawSimpleUnspecified) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // We do not specify what should happen!

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(
        account_number, withdraw_value);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

```
TEST(AtmMachine, OnCallWithZero) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // All default values.
  ON_CALL(mock_bankserver, Connect)
        .WillByDefault(Return());

  ON_CALL(mock_bankserver, GetBalance)
        .WillByDefault(Return(0));

  ON_CALL(mock_bankserver, Debit)
        .WillByDefault(Return());

  ON_CALL(mock_bankserver, Disconnect)
        .WillByDefault(Return());

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(
        account_number, withdraw_value);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

```
atm_machine.Withdraw(1234, 1000):

1.  bankServer_->Connect();
2.  bankServer_->GetBalance(1234);
3.  bankServer_->Debit(1234, 1000);
4.  bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // We only specified GetBalance.
  // The rest are still unspecified.
  ON_CALL(mock_bankserver, GetBalance(account_number))
    .WillByDefault(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(
         account_number, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
TEST(AtmMachine, OnCallWithNonZero) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;


  ON_CALL(mock_bankserver, Connect)
         .WillByDefault(Return());

  ON_CALL(mock_bankserver, GetBalance)
         .WillByDefault(Return(2000));

  ON_CALL(mock_bankserver, Debit)
         .WillByDefault(Return());

  ON_CALL(mock_bankserver, Disconnect)
         .WillByDefault(Return());

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(
         account_number, withdraw_value);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

```
TEST(AtmMachine, OnCallWithOneFunctionSpecified) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // All default values.
  ON_CALL(mock_bankserver, Connect).WillByDefault(Return());
  ON_CALL(mock_bankserver, Disconnect).WillByDefault(Return());
  ON_CALL(mock_bankserver, Debit).WillByDefault(Return());
  ON_CALL(mock_bankserver, Credit).WillByDefault(Return());
  ON_CALL(mock_bankserver, GetBalance(account_number))
      .WillByDefault(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result1 = atm_machine.Withdraw(account_number, withdraw_value);
  bool withdraw_result2 = atm_machine.Withdraw(account_number + 1, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result1);
  EXPECT_FALSE(withdraw_result2);
}
```

The GetBalance Call for me is **unspecified!**

ON_CALL With Test Fixtures

```
class DoDefaultTest : public ::testing::Test {
public:
  void SetUp() override {
    ON_CALL(mock_bankserver_, GetBalance)
        .WillByDefault(Return(1000));
  }

  MockBankServer mock_bankserver_;
};
```

```
TEST_F(DoDefaultTest, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Act
  AtmMachine atm_machine(&mock_bankserver_);
  bool withdraw_result = atm_machine.Withdraw(account_number, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_return.cc

What Does ON_CALL OFFER?

Remember Matchers?

A list of **matchers**

```
ON_CALL(mock_object, method_name(matchers...))
.With(multi_argument_matcher) // Can be used at most once
.WillByDefault(action); // Required
```

```
ON_CALL(mock_bankserver, GetBalance(_))
.WillByDefault(Return(2000));
```

```
ON_CALL(mock_bankserver, GetBalance(Gt(1000)))
.WillByDefault(Return(2000));
```

```
ON_CALL(mock_bankserver,
GetBalance(AllOf(Gt(1000), Lt(2000))))
.WillByDefault(Return(2000));
```

Multi-argument Matchers

Passes all arguments as **one tuple** into the matcher
`Matcher<std::tuple<A1, ..., An>>,`

```
ON_CALL(mock_object, method_name(matchers...))
.With(multi_argument_matcher) // Can be used at most once
.WillByDefault(action); // Required
```

Multi-argument Matchers

Technically, all matchers match a *single* value. A "multi-argument" matcher is just one that matches a *tuple*. The following matchers can be used to match a tuple `(x, y)`:

Matcher	Description
`Eq()`	`x == y`
`Ge()`	`x >= y`
`Gt()`	`x > y`
`Le()`	`x <= y`
`Lt()`	`x < y`
`Ne()`	`x != y`

You can use the following selectors to pick a subset of the arguments (or reorder them) to participate in the matching:

Matcher	Description
`AllArgs(m)`	Equivalent to `m`. Useful as syntactic sugar in `.With(AllArgs(m))`.
`Args<N1, N2, ..., Nk>(m)`	The tuple of the `k` selected (using 0-based indices) arguments matches `m`, e.g. `Args<1, 2>(Eq())`.

Remember Matchers?

Passes all arguments as **one tuple** into the matcher

`Matcher<std::tuple<A1, ..., An>>,`

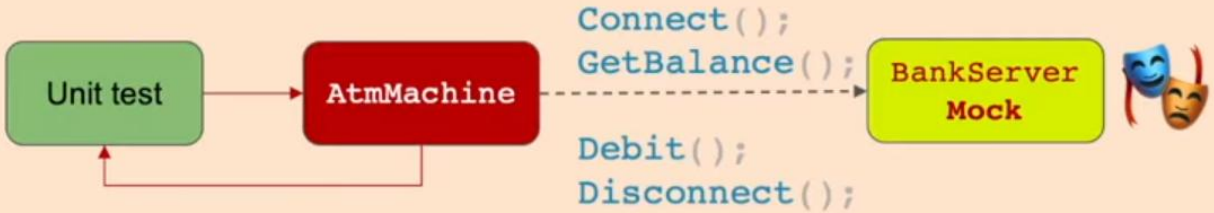
```
ON_CALL(mock_object, method_name(matchers...))
.With(multi_argument_matcher) // Can be used at most once
.WillByDefault(action); // Required
```

```
virtual bool DoubleTransaction(int account_number, int value1, int value2) = 0;

MOCK_METHOD(bool, DoubleTransaction, (int, int, int), (override));
```

```
ON_CALL(mock_bankserver, DoubleTransaction(_, _, _))
      .With(Args<1, 2>(Gt()))
      .WillByDefault(Return(true));
```

```
ON_CALL(mock_bankserver, DoubleTransaction(_, _, _))
      .With(AllOf(Args<0, 1>(Gt()), Args<1, 2>(Gt())))
      .WillByDefault(Return(true));
```

Specifying with the EXPECT_CALL Macro



What should the mock do?

- Unspecified
- ON_CALL
- EXPECT_CALL

EXPECT_CALL vs ON_CALL

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  ON_CALL(mock_bankserver, GetBalance(account_number))
    .WillByDefault(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
          withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

The call will **not be verified**!

```
TEST(AtmMachine, CanWithdrawSimpleExpectation) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
    .Times(AtLeast(1))
    .WillRepeatedly(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
          withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

The call will **be verified**!

tests/gmock_demo/expect_call.cc

Which One Should We Use?

```
EXPECT_CALL(mock_bankserver, GetBalance(account_number))
    .WillRepeatedly(Return(2000));
```

```
ON_CALL(mock_bankserver, GetBalance(account_number))
  .WillByDefault(Return(2000));
```

- Use **ON_CALL** if you just want to specify what should happen when the mocked function is called.
- Use **EXPECT_CALL** if you want to also **check if and how** the mocked function is being called.
- Don't use **EXPECT_CALL** if you don't have to.

What Else Does EXPECT_CALL Offer?

Reminder: EXPECT_CALL vs ON_CALL

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

```
ON_CALL(mock_object, method_name(matchers...))
.With(multi_argument_matcher) // Can be used at most once
.WillByDefault(action); // Required
```

```
TEST(AtmMachine, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
    .Times(1)
    .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
      atm_machine.Withdraw(account_number,
                           withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
TEST(AtmMachine, CanWithdrawSimple) {

// Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  ON_CALL(mock_bankserver, GetBalance(account_number))
    .WillByDefault(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
      atm_machine.Withdraw(account_number,
                           withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

EXPECT_CALL: Matchers

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

```
EXPECT_CALL(mock_bankserver,
      GetBalance(account_number))
      .Times(10)
      .WillOnce(Return(1000))
      .WillOnce(Return(2000));
      .WillRepeatedly(Return(3000));
```

```
EXPECT_CALL(mock_bankserver,
      GetBalance(_))
      .Times(10)
      .WillOnce(Return(1000))
      .WillOnce(Return(2000));
      .WillRepeatedly(Return(3000));
```

Wildcard Matcher: _

```
using ::testing::_; // Matches anything

TEST(AtmMachine, CanWithdraw) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect()).Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(_))
    .Times(1)
    .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(_, _))
    .Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect()).Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

_ matches anything.

tests/gmock_demo/expect_call_matchers.cc

Matchers Example:

```
TEST(AtmMachine, CanWithdrawWithMatchers2) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect()).Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(Gt(1000)))
        .Times(1)
        .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(_, Lt(10000))).Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect()).Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_matchers.cc

Composite Matchers

```
EXPECT_CALL(mock_bankserver, GetBalance(AllOf(Gt(1000), Lt(10000))))
.Times(1)
.WillOnce(Return(2000));
```

Matcher	Description
`AllOf(m1, m2, ..., mn)`	`argument` matches all of the matchers `m1` to `mn`.
`AllOfArray({m0, m1, ..., mn})`, `AllOfArray(a_container)`, `AllOfArray(begin, end)`, `AllOfArray(array)`, or `AllOfArray(array, count)`	The same as `AllOf()` except that the matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`AnyOf(m1, m2, ..., mn)`	`argument` matches at least one of the matchers `m1` to `mn`.
`AnyOfArray({m0, m1, ..., mn})`, `AnyOfArray(a_container)`, `AnyOfArray(begin, end)`, `AnyOfArray(array)`, or `AnyOfArray(array, count)`	The same as `AnyOf()` except that the matchers come from an initializer list, STL-style container, iterator range, or C-style array.
`Not(m)`	`argument` doesn't match matcher `m`.
`Conditional(cond, m1, m2)`	Matches matcher `m1` if `cond` evaluates to true, else matches `m2`.

tests/gmock_demo/expect_call_matchers.cc

Matchers Example: Ignoring Arguments

```
TEST(AtmMachine, CanWithdrawIgnoreArguments) {
// Arrange
MockBankServer mock_bankserver;

// Expectations
EXPECT_CALL(mock_bankserver, Connect).Times(1);

EXPECT_CALL(mock_bankserver, GetBalance).Times(1).WillOnce(Return(2000));

EXPECT_CALL(mock_bankserver, Debit).Times(1);

EXPECT_CALL(mock_bankserver, Disconnect).Times(1);

// Act
AtmMachine atm_machine(&mock_bankserver);
bool withdraw_result = atm_machine.Withdraw(1234, 1000);

// Assert
EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_matchers.cc

EXPECT_CALL: What should it do?

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

```
ON_CALL(mock_object, method_name(matchers...))
.With(multi_argument_matcher) // Can be used at most once
.WillByDefault(action); // Required
```

```
EXPECT_CALL(mock_bankserver, Debit(_, _))
  .With(Gt())
  .Times(1);
```

Multi-argument Matchers

Technically, all matchers match a *single* value. A "multi-argument" matcher is just one that matches a *tuple*. The following matchers can be used to match a tuple `(x, y)`:

Matcher	Description
`Eq()`	`x == y`
`Ge()`	`x >= y`
`Gt()`	`x > y`
`Le()`	`x <= y`
`Lt()`	`x < y`
`Ne()`	`x != y`

You can use the following selectors to pick a subset of the arguments (or reorder them) to participate in the matching:

Matcher	Description
`AllArgs(m)`	Equivalent to `m`. Useful as syntactic sugar in `.With(AllArgs(m))`.
`Args<N1, N2, ..., Nk>(m)`	The tuple of the `k` selected (using 0-based indices) arguments matches `m`, e.g. `Args<1, 2>(Eq())`.

Matchers Example: Multi Argument Matchers

```
TEST(AtmMachine, CanWithdrawWithMultiArgumentMatchers) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver,
        GetBalance(_)).Times(1).WillOnce(Return(200));

  EXPECT_CALL(mock_bankserver, Debit(_, _))
        .With(Gt())
        .Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_matchers.cc

Cardinality

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

```
EXPECT_CALL(mock_bankserver,
        GetBalance(account_number))
        .Times(10)
        .WillOnce(Return(1000));
```

```
EXPECT_CALL(mock_bankserver,
        GetBalance(account_number))
        .Times(AtMost(10))
        .WillOnce(Return(1000));
```

Cardinality	Meaning
`AnyNumber()`	The function can be called any number of times.
`AtLeast(n)`	The function call is expected at least *n* times.
`AtMost(n)`	The function call is expected at most *n* times.
`Between(m, n)`	The function call is expected between *m* and *n* times, inclusive.
`Exactly(n)` or `n`	The function call is expected exactly *n* times. If *n* is 0, the call should never happen.

If the `Times` clause is omitted, GoogleTest infers the cardinality as follows:

- If neither `WillOnce` nor `WillRepeatedly` are specified, the inferred cardinality is `Times(1)`.
- If there are *n* `WillOnce` clauses and no `WillRepeatedly` clause, where *n* >= 1, the inferred cardinality is `Times(n)`.
- If there are *n* `WillOnce` clauses and one `WillRepeatedly` clause, where *n* >= 0, the inferred cardinality is `Times(AtLeast(n))`.

Cardinality

```
TEST(AtmMachine, CanWithdrawSpecifyTimesReturnOnce) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
          .Times(AtMost(10))
          .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result1 = atm_machine.Withdraw(account_number,
        withdraw_value);
  bool withdraw_result2 = atm_machine.Withdraw(account_number,
        withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result1);
  EXPECT_FALSE(withdraw_result2);
}
```

- **Expect:**
  - At most 10 calls
- **Return:**
  - **First** time: 2000
  - The other times: **unspecified**
    - Return **default**, i.e. 0.

tests/gmock_demo/expect_call_cardinality.co

Cardinality

```
TEST(AtmMachine, CanWithdrawSpecifyTimesReturnTwice) {
// Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
        .Times(AtMost(10))
        .WillOnce(Return(2000))
        .WillOnce(Return(3000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result1 = atm_machine.Withdraw(account_number,
        withdraw_value);
  bool withdraw_result2 = atm_machine.Withdraw(account_number,
        withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result1);
  EXPECT_TRUE(withdraw_result2);
}
```

- **Expect:**
  - At most 10 calls
- **Return:**
  - **First** time: 2000
  - **Second** time: 3000
  - The other times: **unspecified**
    - Return **default**, i.e. 0.

tests/gmock_demo/expect_call_cardinality.co

Cardinality

```
TEST(AtmMachine, CanWithdrawSpecifyRepeatedly) {
// Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
        .Times(AtMost(10))
        .WillOnce(Return(2000))
        .WillRepeatedly(Return(3000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result1 = atm_machine.Withdraw(account_number,
        withdraw_value);
  bool withdraw_result2 = atm_machine.Withdraw(account_number,
        withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result1);
  EXPECT_TRUE(withdraw_result2);
}
```

- **Expect:**
  - At most 10 calls
- **Return**
  - First time: 2000
  - Anytime after: 3000

tests/gmock_demo/expect_call_cardinality.co

Cardinality: Times Clause Omitted, No Return

```
TEST(AtmMachine, CanWithdrawNoCardinality_noWillRepeatedly) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result1 = atm_machine.Withdraw(
        account_number, withdraw_value);
  bool withdraw_result2 = atm_machine.Withdraw(
        account_number, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result1);
  EXPECT_TRUE(withdraw_result2);
}
```

- **Expect:**
  - **1** call
- **Return**:
  - **Default:** 0
- The test will **FAIL**!

If the `Times` clause is omitted, GoogleTest infers the cardinality as follows:

- If neither `WillOnce` nor `WillRepeatedly` are specified, the inferred cardinality is `Times(1)`.
- If there are *n* `WillOnce` clauses and no `WillRepeatedly` clause, where *n* >= 1, the inferred cardinality is `Times(n)`.
- If there are *n* `WillOnce` clauses and one `WillRepeatedly` clause, where *n* >= 0, the inferred cardinality is `Times(AtLeast(n))`.

tests/gmock_demo/expect_call_cardinality.cc

Cardinality: Times Clause Omitted

```
TEST(AtmMachine, CanWithdrawNoCardinality_noWillRepeatedly_WillOnce) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
        .WillOnce(Return(1000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result1 = atm_machine.Withdraw(account_number,
        withdraw_value);
  bool withdraw_result2 = atm_machine.Withdraw(account_number,
        withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result1);
  EXPECT_FALSE(withdraw_result2);
}
```

- **Expect:**
  - **1** call
- **Return:**
  - **Always** 1000
- The test will **FAIL**!

If the `Times` clause is omitted, GoogleTest infers the cardinality as follows:

- If neither `WillOnce` nor `WillRepeatedly` are specified, the inferred cardinality is `Times(1)`.
- If there are *n* `WillOnce` clauses and no `WillRepeatedly` clause, where *n* >= 1, the inferred cardinality is `Times(n)`.
- If there are *n* `WillOnce` clauses and one `WillRepeatedly` clause, where *n* >= 0, the inferred cardinality is `Times(AtLeast(n))`.

tests/gmock_demo/expect_call_cardinality.co

Cardinality: Disallow a Call

```
TEST(AtmMachine, CanWithdrawDisallowSomeCalls) {
  // Arrange
  const int account_number = 1234;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
        .Times(1)
        .WillOnce(Return(0));

  EXPECT_CALL(mock_bankserver, Debit(_, _)).Times(0);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
         1000);

  // Assert
  EXPECT_FALSE(withdraw_result);
}
```

- **Expect:**
  - Exactly **0** times
- The test will pass since there is not enough balance and Debit will not be called

tests/gmock_demo/expect_call_cardinality.co

Uninteresting Calls

```
exec ${PAGER:-/usr/bin/less} "$0" || exit 1
Executing tests from //tests/gmock_demo:unspecified
-------------------------------------------------------------------------------
Running main() from gmock_main.cc
[==========] Running 1 test from 1 test suite.
[----------] Global test environment set-up.
[----------] 1 test from AtmMachine
[ RUN      ] AtmMachine.CanWithdrawSimpleUnspecified

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: Connect()
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding
an EXPECT_CALL() if you don't mean to enforce the call.  See https://github.com/google/googletest/blob/master/docs/
gmock_cook_book.md#knowing-when-to-expect for details.

GMOCK WARNING:
Uninteresting mock function call - returning default value.
    Function call: GetBalance(1234)
          Returns: 0
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding
an EXPECT_CALL() if you don't mean to enforce the call.  See https://github.com/google/googletest/blob/master/docs/
gmock_cook_book.md#knowing-when-to-expect for details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: Disconnect()
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding
an EXPECT_CALL() if you don't mean to enforce the call.  See https://github.com/google/googletest/blob/master/docs/
gmock_cook_book.md#knowing-when-to-expect for details.
[       OK ] AtmMachine.CanWithdrawSimpleUnspecified (0 ms)
[----------] 1 test from AtmMachine (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

Uninteresting Calls?

Uninteresting and **Interesting** Mock Functions

```
#include "bankserver.h"
class AtmMachine {
BankServer* bankServer_;

public:
  AtmMachine(BankServer* bankServer) : bankServer_(bankServer) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;
    bankServer_->Connect();
    auto available_balance = bankServer_->
          GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      result = true;
    }

    bankServer_->Disconnect();
    eturn result;
  }
};
```

```
atm_machine.Withdraw(1234, 1000):

1.  bankServer_->Connect();
2.  bankServer_->GetBalance(1234);
3.  bankServer_->Debit(1234, 1000);
4.  bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
          .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
    atm_machine.Withdraw(account_number, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

A call `x.Y(...)` is **uninteresting** if there's ***not even a single*** `EXPECT_CALL(x, Y(...))`.

Uninteresting and **Interesting** Mock Functions

```
[ RUN      ] AtmMachine.CanWithdrawSpecifyTimess

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: Connect()
NOTE: You can safely ignore the above warning unless this call should n
an EXPECT_CALL() if you don't mean to enforce the call.  See https://gi
gmock_cook_book.md#knowing-when-to-expect for details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: Debit(1234, 1000)
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding
an EXPECT_CALL() if you don't mean to enforce the call.  See https://github.com/google/googletest/blob/master/docs/
gmock_cook_book.md#knowing-when-to-expect for details.

GMOCK WARNING:
Uninteresting mock function call - returning directly.
    Function call: Disconnect()
NOTE: You can safely ignore the above warning unless this call should not happen.  Do not suppress it by blindly adding
an EXPECT_CALL() if you don't mean to enforce the call.  See https://github.com/google/googletest/blob/master/docs/
gmock_cook_book.md#knowing-when-to-expect for details.
[       OK ] AtmMachine.CanWithdrawSpecifyTimess (0 ms)
[----------] 1 test from AtmMachine (0 ms total)

[----------] Global test environment tear-down
[==========] 1 test from 1 test suite ran. (0 ms total)
[  PASSED  ] 1 test.
```

```
atm_machine.Withdraw(1234, 1000):

1.  bankServer_->Connect();
2.  bankServer_->GetBalance(1234);
3.  bankServer_->Debit(1234, 1000);
4.  bankServer_->Disconnect();
```

A call `x.Y(...)` is **uninteresting** if there's ***not even a single*** `EXPECT_CALL(x, Y(...))`.

```
TEST(AtmMachine, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect())
    .Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
    .Times(1)
    .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(account_number,
          withdraw_value))
    .Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect())
    .Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
      atm_machine.Withdraw(account_number,
                           withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

All calls are **specified** and hence **interesting**.

tests/gmock_demo/expect_call.cc

Suppressing Uninteresting Call Warnings

```
using ::testing::NiceMock;

TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  ...
}
```

NO uninteresting warnings

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  ...
}
```

Uninteresting warnings

```
using ::testing::StrictMock;

TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  StrictMock<MockBankServer> mock_bankserver;

  ...
}
```

- Makes uninteresting warnings **failures.**
- Disallow unexpected calls

NiceMock and **StrictMock** only affects *uninteresting* calls (calls of *methods* with no expectations)

They do not affect ***unexpected*** calls (calls of methods with expectations, but they don't match).

Uninteresting vs **Unexpected**

A call `x.Y(...)` is **uninteresting** if there's ***not even a single*** `EXPECT_CALL(x, Y(...))`.

A call `x.Y(...)` is **unexpected** if there are ***some*** `EXPECT_CALL(x, Y(...))` set, but none of them matches the call.

```
atm_machine.Withdraw(5678, 1000):

1.  bankServer_->Connect();
2.  bankServer_->GetBalance(5678);
3.  bankServer_->Debit(5678, 1000);
4.  bankServer_->Disconnect();
```

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
    atm_machine.Withdraw(5678, withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

Uninteresting Calls:

```
bankServer_->Connect();
bankServer_->Debit(...);
bankServer_->Disconnect();
```

Unexpected Call:

```
bankServer_->GetBalance(5678);
```

Expected Call:

```
bankServer_->GetBalance(1234);
```

A call `x.Y(...)` is **unexpected** if there are ***some*** `EXPECT_CALL(x, Y(...))` set, but none of them matches the call.

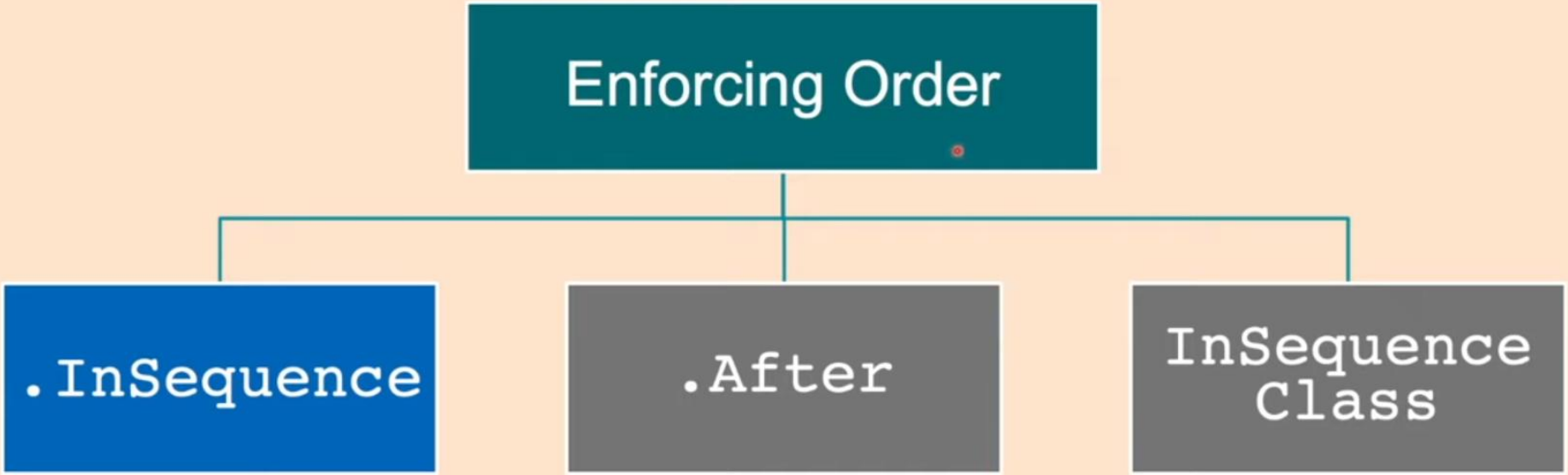
An unexpected call is always an error, as the code under test doesn't behave the way the test expects it to behave.

EXPECT_CALL:
Enforcing the Order of Calls

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

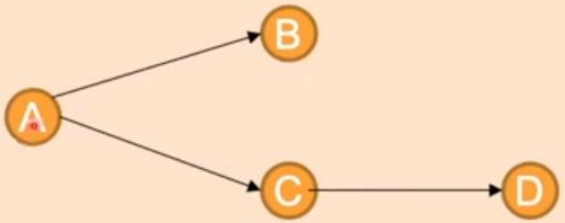
Enforcing the Order of Calls

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



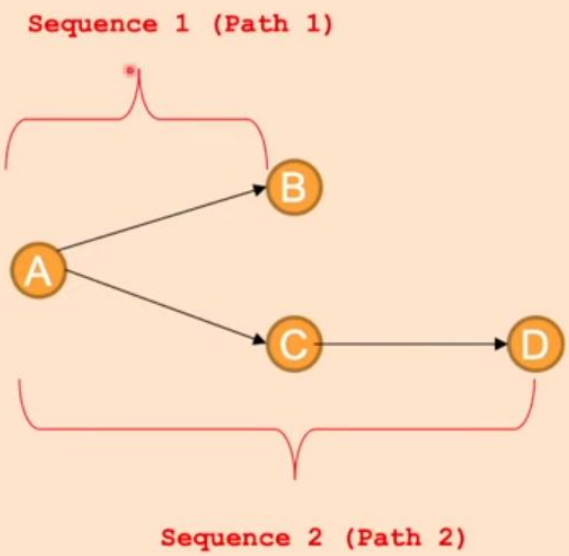
EXPECT_CALL: Forcing Partial Order

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



- A should be called before B
- A should be called before C
- C should be called before D

EXPECT_CALL: Forcing Partial Order




```
using ::testing::Sequence;
... Sequence s1, s2;

EXPECT_CALL(myMock, A()).InSequence(s1, s2);
EXPECT_CALL(myMock, B()).InSequence(s1);

EXPECT_CALL(myMock, C()).InSequence(s2);
EXPECT_CALL(myMock, D()).InSequence(s2);
```

The combination of **Sequence** and the **order** of appearance forces the partial order.

EXPECT_CALL: Forcing Partial Order

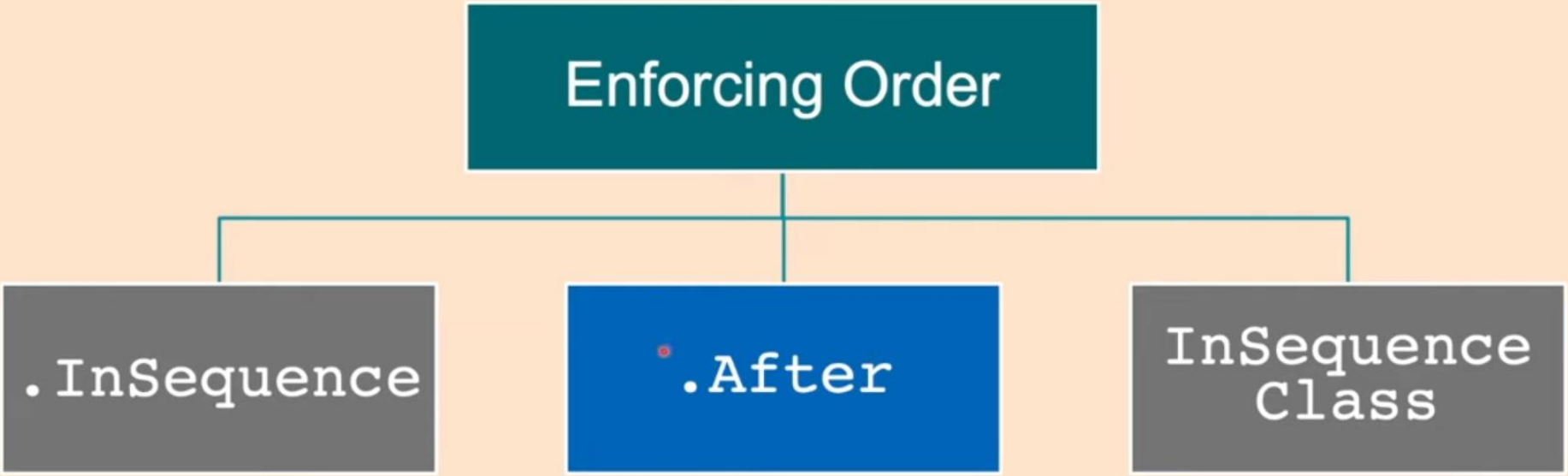
Only forcing **Connect** to be called before **Disconnect**.

```
TEST(AtmMachine, CanWithdrawExpectForcePartialOrder) {
  Sequence s1;
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect())
    .Times(1).InSequence(s1);

  EXPECT_CALL(mock_bankserver, Debit(_, _))
    .Times(1);
  EXPECT_CALL(mock_bankserver, GetBalance(_))
    .Times(1).WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Disconnect())
   .Times(1)
   .InSequence(s1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
   atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_order.cc

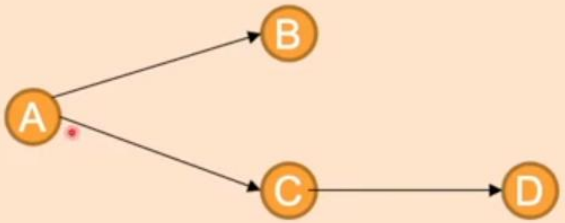
Enforcing the Order of Calls

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



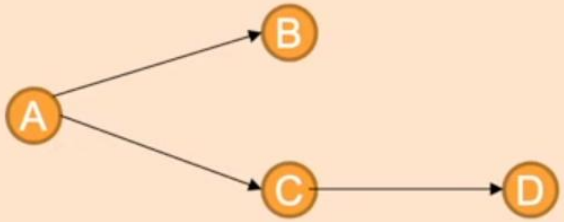
EXPECT_CALL: Forcing Partial Order with After

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



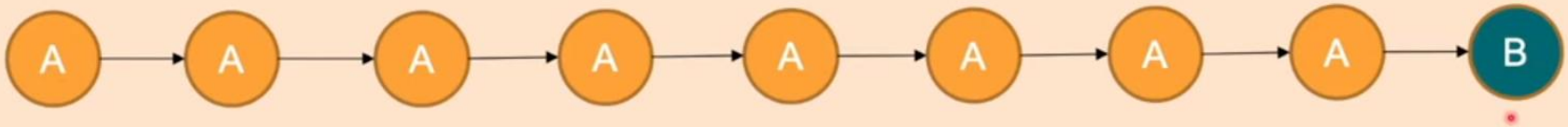
- A should be called before B
- A should be called before C
- C should be called before D

EXPECT_CALL: Forcing Partial Order with After




```
using ::testing::Expectation;
//...
Expectation a = EXPECT_CALL(myMock, A());

Expectation b = EXPECT_CALL(myMock, B()).After(a);

Expectation c = EXPECT_CALL(myMock, C()).After(a);

Expectation d = EXPECT_CALL(myMock, D()).After(a, c);
```

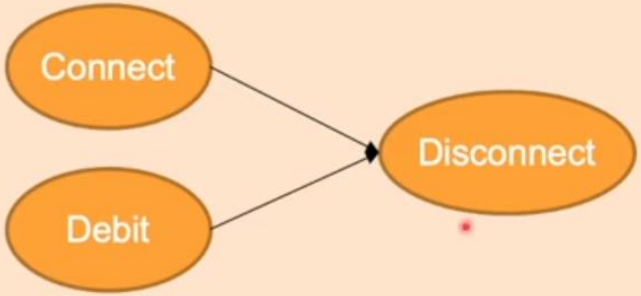
EXPECT_CALL: .After with ExpectationSet




```
using ::testing::ExpectationSet;
// ...
ExpectationSet all_a;

// Collect all expectations of A() calls
for (int i = 0; i < 8; i++) {
  all_a += EXPECT_CALL(my_mock, A(i));
}

// Expect B() call after all A() calls
EXPECT_CALL(my_mock, B())
       .After(all_a);
```

- `.After` takes up to **five** `Expectation` or `ExpectationSet`.
- If you want more, use **`ExpectationSet`**

Forcing **Disconnect** after **Connect** and **Debit** with **.After**

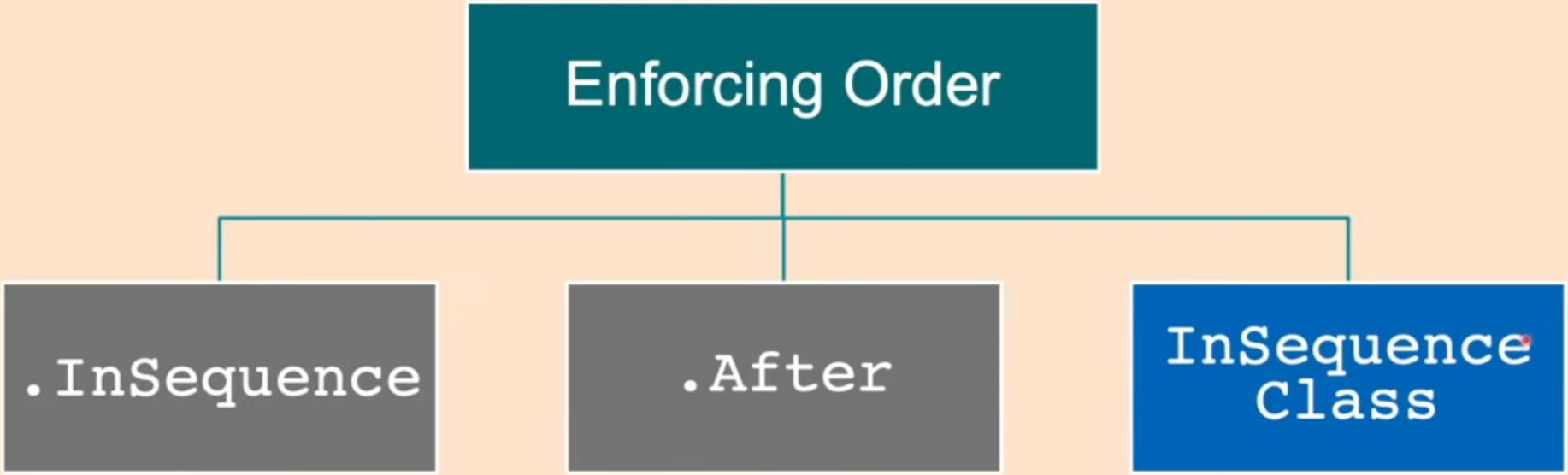



```
TEST(AtmMachine, CanWithdrawExpectForcePartialOrderWithAfter) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  Expectation e_connect = EXPECT_CALL(mock_bankserver, Connect()).Times(1);
  Expectation e_debit = EXPECT_CALL(mock_bankserver, Debit(_, _)).Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect())
    .Times(1)
    .After(e_connect, e_debit);

  EXPECT_CALL(mock_bankserver,
    GetBalance(_)).Times(1).WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_order.cc

Enforcing the Order of Calls

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



Forcing Total Order with InSequence Class

```
TEST(AtmMachine, CanWithdrawExpectInAnyOrder) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(_))
    .Times(1)
    .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Connect())
    .Times(1);

  EXPECT_CALL(mock_bankserver, Debit(_, _))
    .Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect())
    .Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
          atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
// This test will fail if the order of EXPECT_CALLS is changed!
TEST(AtmMachine, CanWithdrawForceSequence) {
  InSequence seq;

  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect()).Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(_))
   .Times(1)
   .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(_, _)).Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect()).Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_order.cc

Forcing Total Order with InSequence Class

```
TEST(AtmMachine, CanWithdrawExpectInAnyOrder) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(_))
    .Times(1)
    .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Connect())
    .Times(1);

  EXPECT_CALL(mock_bankserver, Debit(_, _))
    .Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect())
    .Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
          atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
// This test will fail if the order of EXPECT_CALLS is changed!
TEST(AtmMachine, CanWithdrawForceSequence) {
  InSequence seq;

  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect()).Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(_))
   .Times(1)
   .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(_, _)).Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect()).Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_order.cc

Forcing Partial Order with InSequence Class

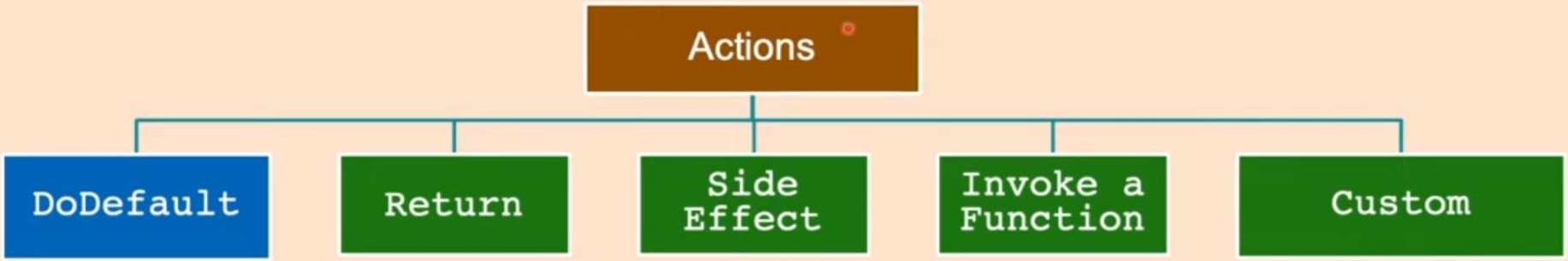
```
TEST(AtmMachine, CanWithdrawForceSequenceOnSomeCalls) {
  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  {
    InSequence seq;
    EXPECT_CALL(mock_bankserver, Debit(_, _)).Times(1);
    EXPECT_CALL(mock_bankserver, Disconnect()).Times(1);
  }

  {
    InSequence seq;
    EXPECT_CALL(mock_bankserver, Connect()).Times(1);
    EXPECT_CALL(mock_bankserver, GetBalance(_)).Times(1).WillOnce(Return(2000));
  }

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_order.cc

GMock Actions

GMock Actions

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



DoDefault

```
class DoDefaultTest : public ::testing::Test {
public:
  void SetUp() override {
    ON_CALL(mock_bankserver_, GetBalance)
          .WillByDefault(Return(1000));
  }

  MockBankServer mock_bankserver_;
};
```

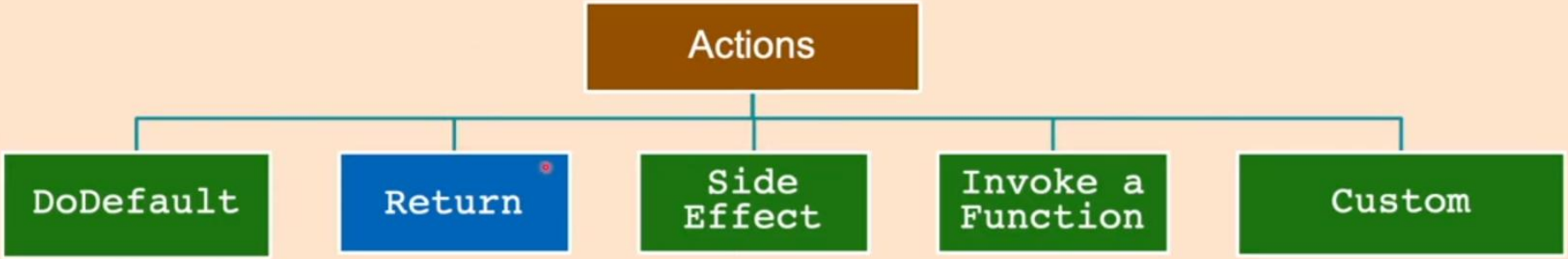
```
TEST_F(DoDefaultTest, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver_, GetBalance(account_number))
          .WillRepeatedly(DoDefault());

  // Act
  AtmMachine atm_machine(&mock_bankserver_);
  bool withdraw_result = atm_machine.Withdraw(account_number,
          withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
TEST_F(DoDefaultTest, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver_, GetBalance(account_number))
          .WillRepeatedly(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver_);
  bool withdraw_result = atm_machine.Withdraw(account_number,
          withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_return.cc

GMock Actions

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



Return a Value

```
EXPECT_CALL(mock_bankserver, GetBalance(account_number))
.Times(10)
.WillOnce(Return(1000))
.WillOnce(Return(2000));
.WillRepeatedly(Return(3000));
```

Returning a Value

Action	Description
`Return()`	Return from a `void` mock function.
`Return(value)`	Return `value`. If the type of `value` is different to the mock function's return type, `value` is converted to the latter type *at the time the expectation is set*, not when the action is executed.
`ReturnArg<N>()`	Return the `N`-th (0-based) argument.
`ReturnNew<T>(a1, ..., ak)`	Return `new T(a1, ..., ak)`; a different object is created each time.
`ReturnNull()`	Return a null pointer.
`ReturnPointee(ptr)`	Return the value pointed to by `ptr`.
`ReturnRef(variable)`	Return a reference to `variable`.
`ReturnRefOfCopy(value)`	Return a reference to a copy of `value`; the copy lives as long as the action.
`ReturnRoundRobin({a1, ..., ak})`	Each call will return the next `ai` in the list, starting at the beginning when the end of the list is reached.

Return Round Robin

```
TEST(AtmMachine, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
   .Times(3)
   .WillRepeatedly(ReturnRoundRobin({2000, 3000, 4000}));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = true;
  for (int i = 0; i < 3; i++) {
    withdraw_result =
      withdraw_result && atm_machine.Withdraw(account_number, withdraw_value);
  }

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

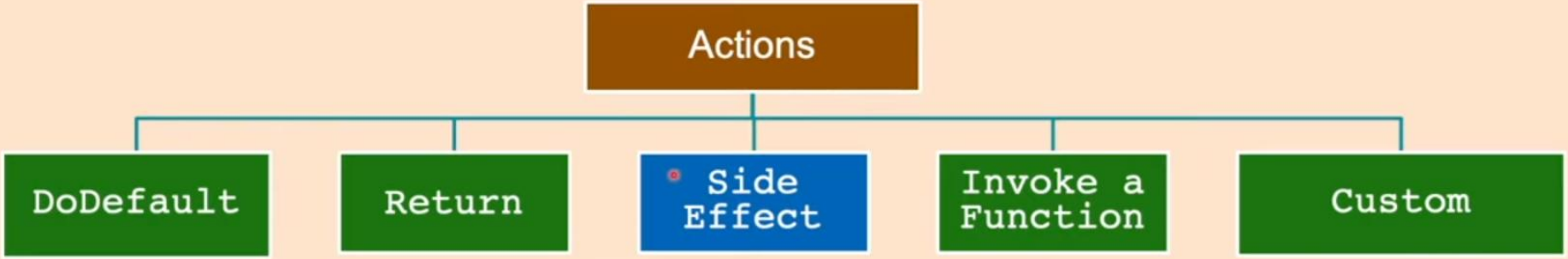
tests/gmock_demo/expect_call_return.cc

```
TEST(AtmMachine, CanWithdrawReturnMultiple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  int current_balance = 10000;
  int* ptr = &current_balance;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect()).WillRepeatedly(Return());

  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
        .WillOnce(ReturnPointee(ptr))
        .WillOnce(ReturnArg<0>())
        .WillRepeatedly(Return(current_balance));

  EXPECT_CALL(mock_bankserver, Disconnect()).WillRepeatedly(DoDefault());

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = true;
  for (int i = 0; i < 10; i++) {
    withdraw_result =
    withdraw_result && atm_machine.Withdraw(account_number, withdraw_value);
  }

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/expect_call_return.cc

GMock Actions

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



Side Effect Actions

```
bool done = false;
EXPECT_CALL(mock_bankserver, Disconnect())
.Times(1)
.WillOnce(Assign(&done, true));
```

Side Effects

Action	Description
`Assign(&variable, value)`	Assign `value` to variable.
`DeleteArg<N>()`	Delete the `N`-th (0-based) argument, which must be a pointer.
`SaveArg<N>(pointer)`	Save the `N`-th (0-based) argument to `*pointer`.
`SaveArgPointee<N>(pointer)`	Save the value pointed to by the `N`-th (0-based) argument to `*pointer`.
`SetArgReferee<N>(value)`	Assign `value` to the variable referenced by the `N`-th (0-based) argument.
`SetArgPointee<N>(value)`	Assign `value` to the variable pointed by the `N`-th (0-based) argument.
`SetArgumentPointee<N>(value)`	Same as `SetArgPointee<N>(value)`. Deprecated. Will be removed in v1.7.0.
`SetArrayArgument<N>(first, last)`	Copies the elements in source range [`first`, `last`) to the array pointed to by the `N`-th (0-based) argument, which can be either a pointer or an iterator. The action does not take ownership of the elements in the source range.
`SetErrnoAndReturn(error, value)`	Set `errno` to `error` and return `value`.
`Throw(exception)`	Throws the given exception, which can be any copyable value. Available since v1.1.0.

Side Effects

```
TEST(AtmMachine, CanWithdrawSideEffect) {
  bool done = false;
  int account_number, value;

  // Arrange
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Debit(_, _))
   .Times(1)
   .WillOnce(DoAll(SaveArg<0>(&account_number), SaveArg<1>(&value)));

  EXPECT_CALL(mock_bankserver, Disconnect())
   .Times(1)
   .WillOnce(Assign(&done, true));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);

  // Assert
  EXPECT_TRUE(withdraw_result);

  EXPECT_TRUE(done);
  EXPECT_EQ(account_number, 1234);
  EXPECT_EQ(value, 1000);
}
```

tests/gmock_demo/expect_call_side_effects.cc

Set errno

```
TEST(AtmMachine, CanWithdrawSideEffectErrno) {
  // Arrange
  MockBankServer mock_bankserver;
  errno = 0; // Defined in <errno.h>

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(4567))
        .Times(1)
        .WillOnce(SetErrnoAndReturn(EINVAL, 0));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(4567, 1000);

  // Assert
  EXPECT_FALSE(withdraw_result);
  EXPECT_EQ(errno, EINVAL);
}
```

tests/gmock_demo/expect_call_side_effects.cc

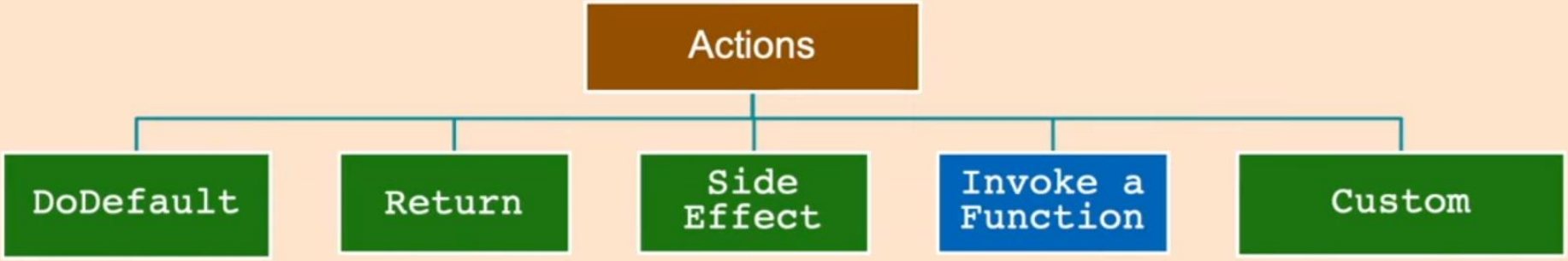
Throw an Exception as a Side Effects

```
TEST(AtmMachine, ThrowException) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect())
        .WillRepeatedly(Throw(InvalidValueException()));

  // Act and Assert
  AtmMachine atm_machine(&mock_bankserver);

  EXPECT_THROW(atm_machine.Withdraw(account_number, withdraw_value),
        InvalidValueException);

  EXPECT_THAT([&]() { atm_machine.Withdraw(account_number, withdraw_value); },
        Throws<InvalidValueException>());

  EXPECT_THAT([&]() { atm_machine.Withdraw(account_number, withdraw_value); },
        ThrowsMessage<InvalidValueException>(StrEq("Invalid value")));
}
```

tests/gmock_demo/expect_call_side_effects.cc

GMock Actions

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



Invoking a Function

```
double Square(double x) { return (x * x); }

TEST(AtmMachine, CanWithdraw) {
  EXPECT_CALL(mock_bankserver, GetBalance(_))
    .Times(1)
    .WillOnce(Invoke(Square));
  bool withdraw_result = atm_machine.Withdraw(1234, 1000);
}
```

Action	Description
`f`	Invoke `f` with the arguments passed to the mock function, where `f` is a callable.
`Invoke(f)`	Invoke `f` with the arguments passed to the mock function, where `f` can be a global/static function or a functor.
`Invoke(object_pointer, &class::method)`	Invoke the method on the object with the arguments passed to the mock function.
`InvokeWithoutArgs(f)`	Invoke `f`, which can be a global/static function or a functor. `f` must take no arguments.
`InvokeWithoutArgs(object_pointer, &class::method)`	Invoke the method on the object, which takes no arguments.
`InvokeArgument<N>(arg1, arg2, ..., argk)`	Invoke the mock function's `N`-th (0-based) argument, which must be a function or a functor, with the `k` arguments.

The return value of the invoked function is used as the return value of the action.

```
TEST(AtmMachine, CanWithdrawWithMultipleInvoke) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;
  const int number_of_calls = 10;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
          .Times(number_of_calls)
          .WillOnce(Invoke(Square))
          .WillOnce(Square)
          .WillOnce(DoAll(InvokeWithoutArgs(PrintHello), Return(1000)))
          .WillOnce([](int n) { return Square(n); })
          .WillOnce([](int n) { return Sum(n, 1000); })
          .WillOnce([]() { return Return10000(); })
          .WillOnce(Return10000)
          .WillRepeatedly(Return(Return10000()));

  EXPECT_CALL(mock_bankserver, Debit(account_number, withdraw_value))
          .WillRepeatedly(Invoke([](int a, int w) {
                    std::cout << "a: " << a << ", w: " << w << std::endl;
          }));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = true;
  for (int i = 0; i < number_of_calls; i++) {
    withdraw_result = withdraw_result &&
          atm_machine.Withdraw(account_number, withdraw_value);
  }

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

```
void PrintHello() { std::cout << "Hello!" << std::endl; }
double Square(double x) { return (x * x); }
int Sum(int x, int y) { return x + y; }
int Return10000() { return 10000; }
```

Composite Actions

```
EXPECT_CALL(mock_bankserver, Debit(_, _))
    .Times(1)
    .WillOnce(DoAll(SaveArg<0>(&account_number), SaveArg<1>(&value)));
```

Composite Actions

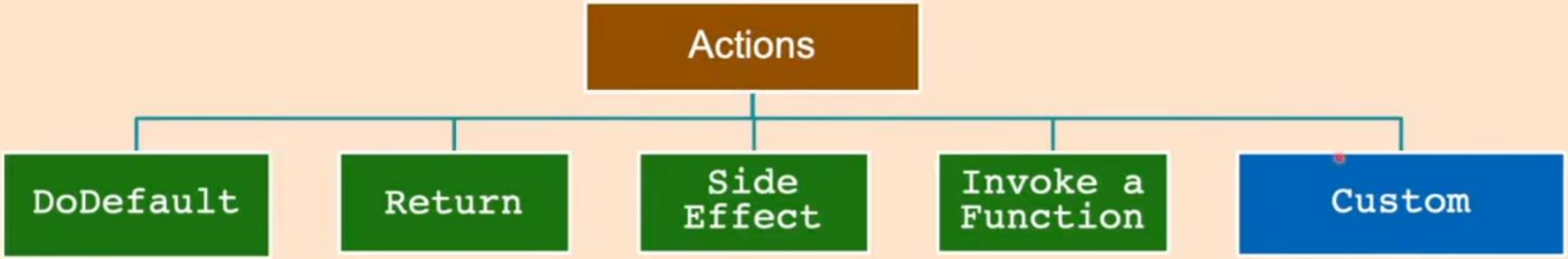
Action	Description
`DoAll(a1, a2, ..., an)`	Do all actions `a1` to `an` and return the result of `an` in each invocation. The first `n - 1` sub-actions must return void and will receive a readonly view of the arguments.
`IgnoreResult(a)`	Perform action `a` and ignore its result. `a` must not return void.
`WithArg<N>(a)`	Pass the `N`-th (0-based) argument of the mock function to action `a` and perform it.
`WithArgs<N1, N2, ..., Nk>(a)`	Pass the selected (0-based) arguments of the mock function to action `a` and perform it.
`WithoutArgs(a)`	Perform action `a` without any arguments.

WithArg Example

```
class MyMock {
public:
  MOCK_METHOD(int, method, (int, int, int, int), ());
};

TEST(MyTest, CanDouble) {
  MyMock mock;

  EXPECT_CALL(mock, method(_, _, _, _))
       .WillOnce(WithArg<1>(Invoke(
              [](int x) {
                return x * 2;
              })));

  EXPECT_EQ(mock.method(1, 5, 1, 1), 10);
}
```

GMock Actions

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```



Custom Action Example 1

Step 1: Define a functor class with a possibly templated call operator

```
struct Double {
  template <typename T>
  T operator()(T arg) {
    return 2 * (arg);
  }
};
```

Step 2: Use it!

```
TEST(MyTest, CanDoubleWithFunctor) {
  MyMock mock;

  EXPECT_CALL(mock,method(_))
       .WillRepeatedly(Double{});

  EXPECT_EQ(mock.method(5), 10);
  EXPECT_EQ(mock.method(7), 14);
}
```

Custom Action Example 2

Step 1: Define a functor class with a possibly templated call operator

```
struct SumOfAll {
  template <typename T>
  T operator()(T arg0, T arg1, T arg2, T arg3) {
    return arg0 + arg1 + arg2 + arg3;
  }
};
```

Step 2: Use it!

```
TEST(MyTest, SumOfAllWithFunctor) {
  MyMock mock;

  EXPECT_CALL(mock, method2(_, _, _, _))
       .WillOnce(SumOfAll{});

  EXPECT_EQ(mock.method2(1, 2, 3, 4), 10);
}
```

Sticky Rules

```
EXPECT_CALL(mock_object, method_name(matchers...))
  .With(multi_argument_matcher) // Can be used at most once
  .Times(cardinality) // Can be used at most once
  .InSequence(sequences...) // Can be used any number of times
  .After(expectations...) // Can be used any number of times
  .WillOnce(action) // Can be used any number of times
  .WillRepeatedly(action) // Can be used at most once
  .RetiresOnSaturation(); // Can be used at most once
```

All Expectations Are **Sticky** (Unless Said Otherwise)

They remain **active** even **after** we have reached their invocation **upper bounds**.

EXPECT_CALLs will be matched from **bottom** to **top.**

Rules are Sticky!

```
TEST(AtmMachine, CanWithdrawNonOverlapping) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(Gt(1234)))
        .Times(AnyNumber())
        .WillRepeatedly(Return(2000));

  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(1000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(5678, withdraw_value);
  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

We both stay active
but we don't overlap.

tests/gmock_demo/expect_call_sticky_rules.cc

Rules are Sticky!

```
TEST(AtmMachine, CanWithdraw_MultipleCalls) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(1000));

  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(1234, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

We both stay active
and overlap!
The test will FAIL!

tests/gmock_demo/expect_call_sticky_rules.cc

Retire the Saturated Rule

```
TEST(AtmMachine, CanWithdraw_MultipleCalls_Retire) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(1000));

  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(2000))
        .RetiresOnSaturation();

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(1234, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

I retire after the first match!
The test will PASS.

tests/gmock_demo/expect_call_sticky_rules.cc

Using a Single EXPECT_CALL

```
TEST(AtmMachine, CanWithdraw_MultipleCalls_WillOnce) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(2)
        .WillOnce(Return(1000))
        .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(1234, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

A better way of writing the same thing if they completely overlap

tests/gmock_demo/expect_call_sticky_rules.cc

EXPECT_CALLs Matched from Bottom to Top

```
TEST(AtmMachine, CanWithdraw_GenericRuleFirst_DifferentValues) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(_))
        .Times(1)
        .WillRepeatedly(Return(2000));

  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(1000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(5678, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

We both stay active!

However, I only get triggered once

The test will PASS.

tests/gmock_demo/expect_call_sticky_rules.cc

EXPECT_CALLs Matched from Bottom to Top

```
// This test is disabled since it fails due to stickiness of the rules.
TEST(AtmMachine, DISABLED_CanWithdraw_GenericRuleLast_DifferentValues) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(1000));

  EXPECT_CALL(mock_bankserver, GetBalance(_))
        .Times(1)
        .WillRepeatedly(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(5678, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

We both stay active!

I get triggered twice!
The test will **FAIL**!

tests/gmock_demo/expect_call_sticky_rules.cc

EXPECT_CALLs Matched from Bottom to Top

```
TEST(AtmMachine, CanWithdraw_GenericRuleFirst_DifferentValues) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(_))
        .Times(1)
        .WillRepeatedly(Return(2000));

  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(1)
        .WillOnce(Return(1000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(5678, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

We both stay active!

However, I only get triggered once

The test will PASS.

tests/gmock_demo/expect_call_sticky_rules.cc

Using a Single EXPECT_CALL

```
TEST(AtmMachine, CanWithdraw_MultipleCalls_WillOnce) {
  // Arrange
  const int withdraw_value = 1000;

  NiceMock<MockBankServer> mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, GetBalance(1234))
        .Times(2)
        .WillOnce(Return(1000))
        .WillOnce(Return(2000));

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  auto result1 = atm_machine.Withdraw(1234, withdraw_value);
  auto result2 = atm_machine.Withdraw(1234, withdraw_value);

  // Assert
  EXPECT_TRUE(result1);
  EXPECT_TRUE(result2);
}
```

A better way of writing the same thing if they completely overlap

tests/gmock_demo/expect_call_sticky_rules.cc

GMock
Advanced Topics

Creating CheckPoints

Checkpoints

```
mock_bank_server mock_bank_server;
//-----------------------------------------------------
// If value is product of 10000, withdraws 10000 from
account 1234.
void Withdraw10kProducts(int value) {
  if (value % 10000 == 0) {
    mock_bank_server.Debit(1234, 10000);
  }
}
//-----------------------------------------------------
TEST(Withdraw10kProductsTest, WithdrawIsCalledTwice) {
  EXPECT_CALL(mock_bank_server, Debit(1234, 10000))
   .Times(2);

  Withdraw10kProducts(20000);
  Withdraw10kProducts(2000);
  Withdraw10kProducts(3000);
  Withdraw10kProducts(4000);
  Withdraw10kProducts(30000);
}
```

```
class Checkpoint {
public:
  MOCK_METHOD(void, Check, (std::string), ());
};

// The expectation spec says that the first Withdraw(1234, 10000) call must
// happen before checkpoint 1, the second Withdraw(1234, 10000) call must happen
// after checkpoint 2, and nothing should happen between the two checkpoints.
TEST(Withdraw10kProductsTest, WithdrawIsCalledCorrectlyCheckpoint) {
  Checkpoint checkpoint;

  {
    InSequence s;

    EXPECT_CALL(mock_bank_server, Debit(1234, 10000));
    EXPECT_CALL(checkpoint, Check("1"));
    EXPECT_CALL(checkpoint, Check("2"));
    EXPECT_CALL(mock_bank_server, Debit(1234, 10000));
  }

  // Withdraw should be called once
  Withdraw10kProducts(20000);
  checkpoint.Check("1");

  // Withdraw should not be called
  Withdraw10kProducts(2000);
  Withdraw10kProducts(3000);
  Withdraw10kProducts(4000);

  // Withdraw should be called once
  checkpoint.Check("2");
  Withdraw10kProducts(30000);
}
```

tests/gmock_demo/expect_call_checkpoint.cc

Checkpoints

```
mock_bank_server mock_bank_server;
//-------------------------------------------------------
// If value is product of 10000, withdraws 10000 from
account 1234.
void Withdraw10kProducts(int value) {
  if (value % 10000 == 0) {
    mock_bank_server.Withdraw(1234, 10000);
  }
}
//-------------------------------------------------------
TEST(Withdraw10kProductsTest, WithdrawIsCalledTwice) {
  EXPECT_CALL(mock_bank_server, Debit(1234, 10000))
   .Times(2);

  Withdraw10kProducts(20000);
  Withdraw10kProducts(2000);
  Withdraw10kProducts(3000);
  Withdraw10kProducts(4000);
  Withdraw10kProducts(30000);
}
```

```
// The expectation spec says that the first Withdraw(1234, 10000) call must
// happen before checkpoint 1, the second Withdraw(1234, 10000) call must happen
// after checkpoint 2, and nothing should happen between the two checkpoints.
TEST(Withdraw10kProductsTest, WithdrawIsCalledCorrectly) {
  // Class MockFunction<F> has exactly one mock method. It is named
  // Call() and has type F.
  MockFunction<void(std::string check_point_name)> checkpoint;

  {
    InSequence s;

    EXPECT_CALL(mock_bank_server, Debit(1234, 10000));
    EXPECT_CALL(checkpoint, Call("1"));

    EXPECT_CALL(checkpoint, Call("2"));
    EXPECT_CALL(mock_bank_server, Debit(1234, 10000));
  }

  // Withdraw should be called once
  Withdraw10kProducts(20000);
  checkpoint.Call("1");

  // Withdraw should not be called
  Withdraw10kProducts(2000);
  Withdraw10kProducts(3000);
  Withdraw10kProducts(4000);

  // Withdraw should be called once
  checkpoint.Call("2");
  Withdraw10kProducts(30000);
}
```

tests/gmock_demo/expect_call_checkpoint.cc

Mocking
(More Details)

```
class BankServer {
public:
  virtual ~BankServer() {}
  virtual void Connect() = 0;
  virtual void Disconnect() = 0;
  virtual void Credit(int account_number, int value) = 0;
  virtual void Debit(int account_number, int value) = 0;
  virtual bool DoubleTransaction(int account_number,
          int value1, int value2) = 0;
  virtual int GetBalance(int account_number) const = 0;
};
```

Mocking **Private** or **Protected** Methods

```
class Foo {
public:
  virtual bool Transform(Gadget* g) = 0;

protected:
  virtual void Resume();

private:
  virtual int GetTimeOut();
};
```

```
class MockFoo : public Foo {
public:
...
  MOCK_METHOD(bool, Transform, (Gadget* g), (override));

// The following must be in the public section, even though the
// methods are protected or private in the base class.
  MOCK_METHOD(void, Resume, (), (override));
  MOCK_METHOD(int, GetTimeOut, (), (override));
};
```

Mocking **Overloaded** Methods

```
class Foo {
...

// Must be virtual as we'll inherit from Foo.
  virtual ~Foo();

// Overloaded on the types and/or
// numbers of arguments.
  virtual int Add(Element x);
  virtual int Add(int times, Element x);

// Overloaded on the const-ness of this object.
  virtual Bar& GetBar();
  virtual const Bar& GetBar() const;
};
```

```
class MockFoo : public Foo {
  ...
  MOCK_METHOD(int, Add, (Element x), (override));
  MOCK_METHOD(int, Add, (int times, Element x), (override));

  MOCK_METHOD(Bar&, GetBar, (), (override));
  MOCK_METHOD(const Bar&, GetBar, (), (const, override));
};
```

Mock overloads as usual

- No special attention needed.

Mocking **Template** classes

```
template <typename Elem>
class StackInterface {
...

  virtual ~StackInterface();

  virtual int GetSize() const = 0;
  virtual void Push(const Elem& x) = 0;
};
```

```
template <typename Elem>
class MockStack : public StackInterface<Elem> {
...
  MOCK_METHOD(int, GetSize, (), (override));
  MOCK_METHOD(void, Push, (const Elem& x), (override));
};
```

Mock them just like any class!

Mocking Non-Virtual Classes

Our First Test With GMock

```
class BankServer {
public:
  virtual ~BankServer() {}
  virtual void Connect() = 0;
  virtual void Disconnect() = 0;
  virtual void Credit(int account_number, int value) = 0;
  virtual void Withdraw(int account_number, int value) = 0;
  virtual bool DoubleTransaction(int account_number,
          int value1, int value2) = 0;
  virtual int GetBalance(int account_number) const = 0;
};
```

bankserver.h

```
#include "bankserver.h"
#include "gmock/gmock.h"

class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(bool, DoubleTransaction, (int, int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));
};
```

mock_bankserver.h

```
#include <complicated_dependencies.h>

class BankOfAriServer : public BankServer {
 // Actual implementation of APIs
 // Lots of dependencies!
};
```

ari.h

Pass a **mock** `BankServer` class to `AtmMachine` class

```
class BankServer {
public:
 virtual ~BankServer() {}
 virtual void Connect() = 0;
 virtual void Disconnect() = 0;
 virtual void Credit(int account_number, int value) = 0;
 virtual void Debit(int account_number, int value) = 0;
 virtual bool DoubleTransaction(int account_number,
              int value1, int value2) = 0;
 virtual int GetBalance(int account_number) const = 0;
};
```

```
#include <bankserver.h>
class AtmMachine {
BankServer* bankServer_;

public:
  AtmMachine(BankServer* bankServer) :
                         bankServer_(bankServer) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;

    bankServer_->Connect();

    auto available_balance = bankServer_->
          GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      result = true;
    }

    bankServer_->Disconnect();
    return result;
  }
};
```

Production

Test

Mocking **Non-Virtual** classes

```
class BankOfAriServer {
public:
  ~BankOfAriServer() {}
  void Connect();
  void Disconnect();
  void Credit(int account_number, int value);
  void Debit(int account_number, int value);
  int GetBalance(int account_number) const;
};
```

```
class MockBankServer {
public:
  MOCK_METHOD(void, Connect, (), ());
  MOCK_METHOD(void, Disconnect, (), ());
  MOCK_METHOD(void, Credit, (int, int), ());
  MOCK_METHOD(void, Debit, (int, int), ());
  MOCK_METHOD(int, GetBalance, (int), (const));
};
```

```
template <class T>
class AtmMachine {
  T* bankServer_;
  ...
};
```

Production

Test

```
AtmMachine<BankOfAriServer>
        atm_machine(&mock_bankserver);
```

```
AtmMachine<MockBankServer>
        atm_machine(&mock_bankserver);
```

Mocking **Non-Virtual** classes

```
class BankOfAriServer {
public:
  ~BankOfAriServer() {}
  void Connect();
  void Disconnect();
  void Credit(int account_number, int value);
  void Debit(int account_number, int value);
  int GetBalance(int account_number) const;
};
```

```
class MockBankServer {
public:
  MOCK_METHOD(void, Connect, (), ());
  MOCK_METHOD(void, Disconnect, (), ());
  MOCK_METHOD(void, Credit, (int, int), ());
  MOCK_METHOD(void, Debit, (int, int), ());
  MOCK_METHOD(int, GetBalance, (int), (const));
};
```

```
template <class T>
class AtmMachine {
  T* bankServer_;
  ...
};
```

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;
  ON_CALL(mock_bankserver, GetBalance(account_number))
        .WillByDefault(Return(2000));

  AtmMachine<MockBankServer> atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
        withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/mock_non_virtual_classes.cc

Mocking **a Free Function**

```
bool Log(std::string message) {
  std::cout << "message: " << message << std::endl;
  return true;
}
```

Step 1: Wrap in a class

```
class LogWrapperInterface {
public:
  virtual bool Log(std::string msg) = 0;
};
```

Step 2: Mock the class

```
class LogWrapper : public LogWrapperInterface {
public:
  bool Log(std::string msg) override {
    return ::Log(msg);
  }
};

class MockLogWrapper : public LogWrapperInterface {
public:
  MOCK_METHOD(bool, Log, (std::string), (override));
};
```

Step 3: Use and inject the wrapped class.

```
class AtmMachine {
BankServer* bankServer_;
LogWrapperInterface* logWrapper_;

public:
  AtmMachine(BankServer* bankServer,
             LogWrapperInterface* logWrapper):
             bankServer_(bankServer), logWrapper_(logWrapper) {}

  bool Withdraw(int account_number, int value) {
    bool result = false;
    bankServer_->Connect();
    logWrapper_->Log("Connected.");
    auto available_balance = bankServer_->
           GetBalance(account_number);

    if (available_balance >= value) {
      bankServer_->Debit(account_number, value);
      logWrapper_->Log("Withdraw done.");
      result = true;
    }

    bankServer_->Disconnect();
    logWrapper_->Log("Disconnected.");
    return result;
  }
};
```

tests/gmock_demo/mocking_free_function.cc

Delegate To Fake

```
class FakeBankServer : public BankServer {
public:
  void Connect() override {}
  void Disconnect() override {}
  void Credit(int account_number, int value) override {}
  void Debit(int account_number, int value) override {}
  int GetBalance(int account_number) const override {
    switch (account_number) {
      case 1234:
      case 4567:
        return 2000;

      default:
        return account_number * account_number +
            account_number / 3;
    }
  }
};
```

Fake Class

If your mock is **non-trivial** or you want to **share** it among multiple tests, you can delegate the functionality to a Fake class.

```
class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));

  // Delegates to a FakeBankServer object.
  // This must be called *before* the custom ON_CALL() statements.
  void DelegateToFake() {
    ON_CALL(*this, GetBalance)
    .WillByDefault([this](int account_number) {
      return fake_.GetBalance(account_number);
    });
}

private:
  FakeBankServer fake_; // Keeps an instance of the fake in the mock.
};
```

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  mock_bankserver.DelegateToFake(); // Enables delegation.

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
           withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

tests/gmock_demo/delegate_to_fake.cc

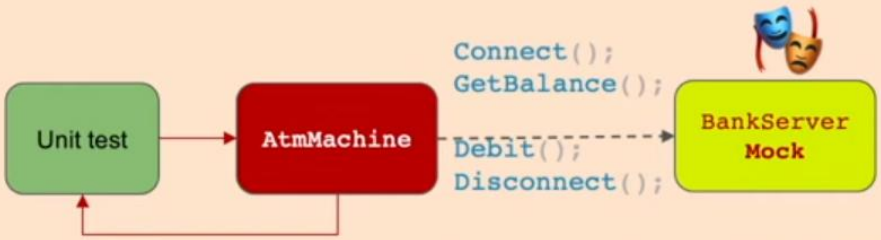
Delegate To Real

```
class MockBankServer : public BankServer {
public:
  MOCK_METHOD(void, Connect, (), (override));
  MOCK_METHOD(void, Disconnect, (), (override));
  MOCK_METHOD(void, Credit, (int, int), (override));
  MOCK_METHOD(void, Debit, (int, int), (override));
  MOCK_METHOD(int, GetBalance, (int), (const, override));

  // Delegates to a RealBankServer object.
  // This must be called *before* the custom
  // ON_CALL() statements.
  void DelegateToReal() {
    ON_CALL(*this, GetBalance)
    .WillByDefault([this](int account_number) {
      return real_.GetBalance(account_number);
    });
}

private:
  RealBankServer real_;
};
```

```
TEST(AtmMachine, CanWithdrawSimple) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  mock_bankserver.DelegateToReal(); // Enables delegation.

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result = atm_machine.Withdraw(account_number,
          withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```

Criticisms of Google Mock

- **GMock Criticisms**
  - **Macros** are bad and hard to debug.
  - Not exactly **Standard C++** (e.g. `Eq()` instead of ==)
  - Requires:
    - **Virtual** classes
    - **Templates**
    - **Wrapper** class for **free functions**
  - Can't be used with **legacy code** without modification.
  - The macros are **confusing**.
    - The choice between **ON_CALL** and **EXPECT_CALL** is unclear.
    - The stickiness of **EXPECT_CALL** is confusing.
  - GMock tests are **hard to debug**: they can fail from multiple points.
  - Mocking creates a tight coupling between the SUT and collaborators.
  - Tests become **too long** and hard to read.




```
TEST(AtmMachine, CanWithdraw) {
  // Arrange
  const int account_number = 1234;
  const int withdraw_value = 1000;

  MockBankServer mock_bankserver;

  // Expectations
  EXPECT_CALL(mock_bankserver, Connect())
    .Times(1);

  EXPECT_CALL(mock_bankserver, GetBalance(account_number))
    .Times(1)
    .WillOnce(Return(2000));

  EXPECT_CALL(mock_bankserver, Debit(account_number,
          withdraw_value))
    .Times(1);

  EXPECT_CALL(mock_bankserver, Disconnect())
    .Times(1);

  // Act
  AtmMachine atm_machine(&mock_bankserver);
  bool withdraw_result =
      atm_machine.Withdraw(account_number,
                           withdraw_value);

  // Assert
  EXPECT_TRUE(withdraw_result);
}
```