MOCK OBJECTS
MOCKITO

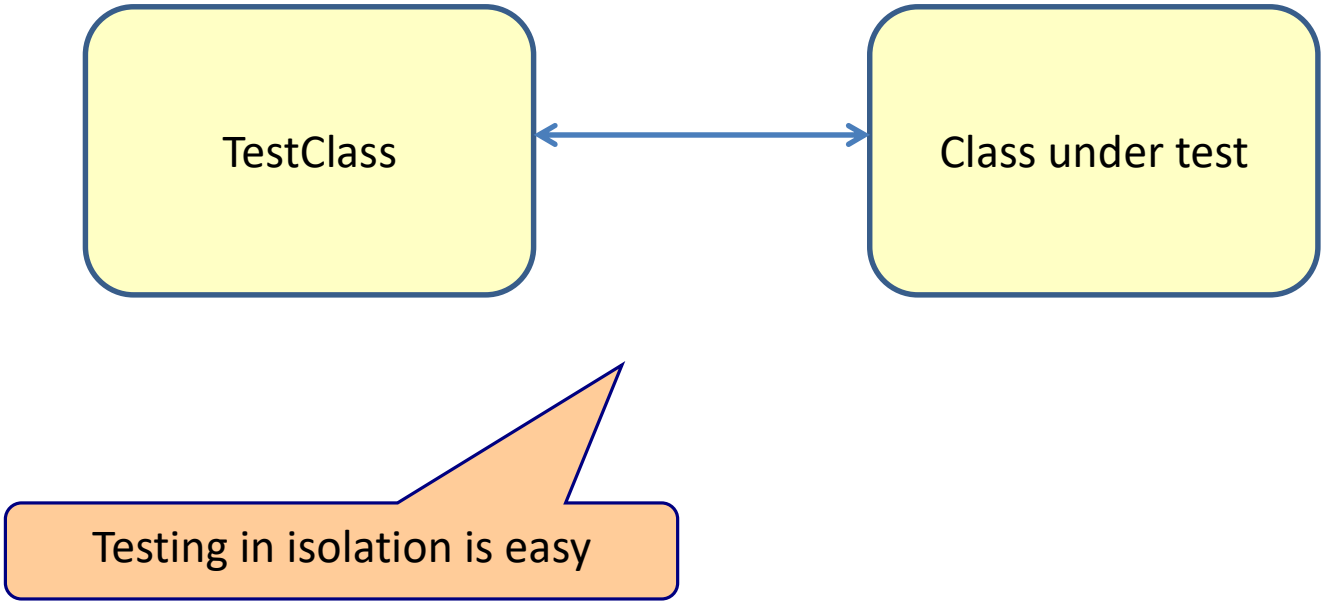
Mock objects

- Dummy objects
- Allow to test a class in isolation

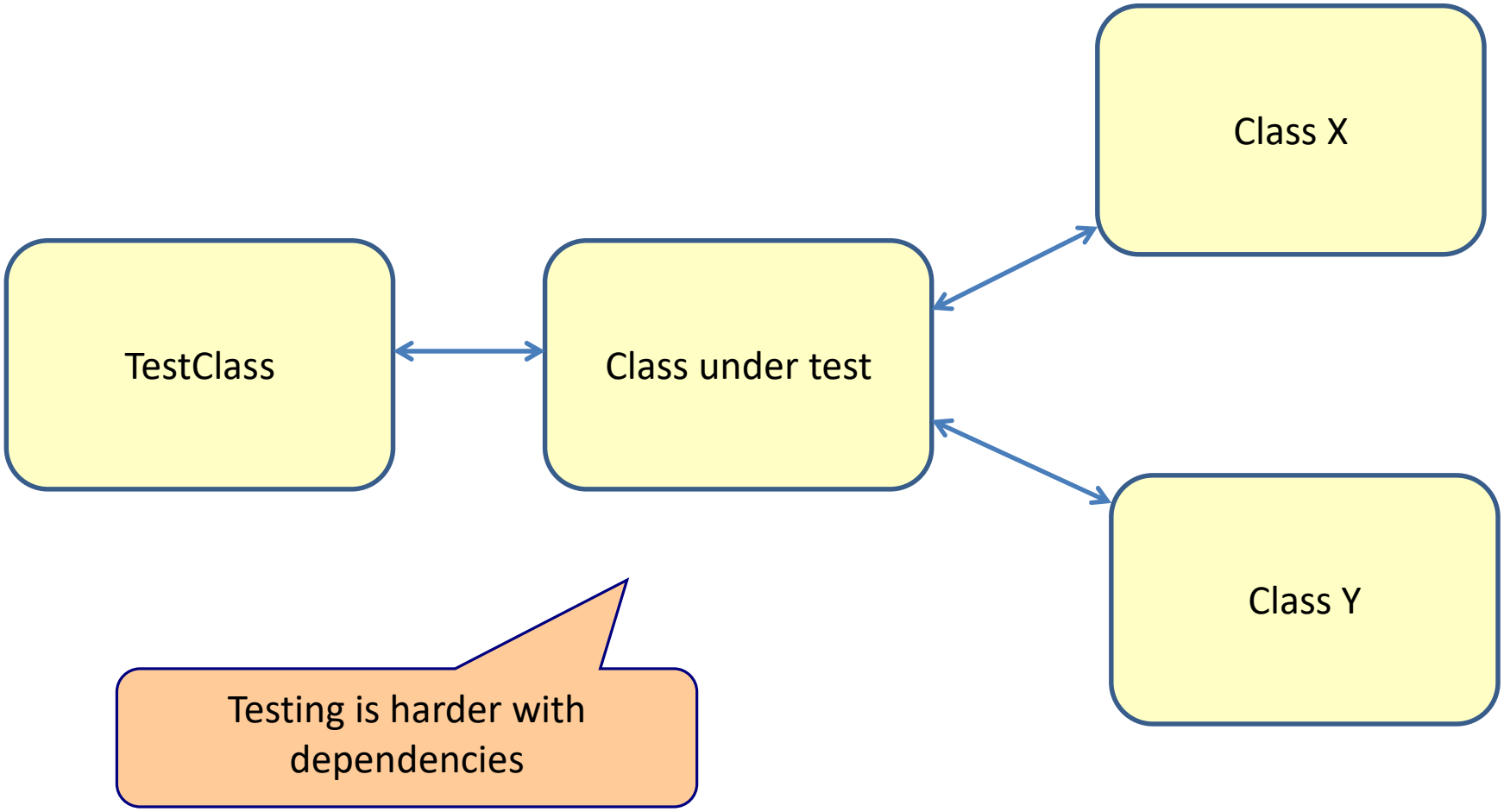
Mock vs. stub

- Mock is al about interaction
  - Did the service class correctly call the DAO class?
- Stub is all about state
  - Return some dummy data
  - assertEquals(4, item.getCount());

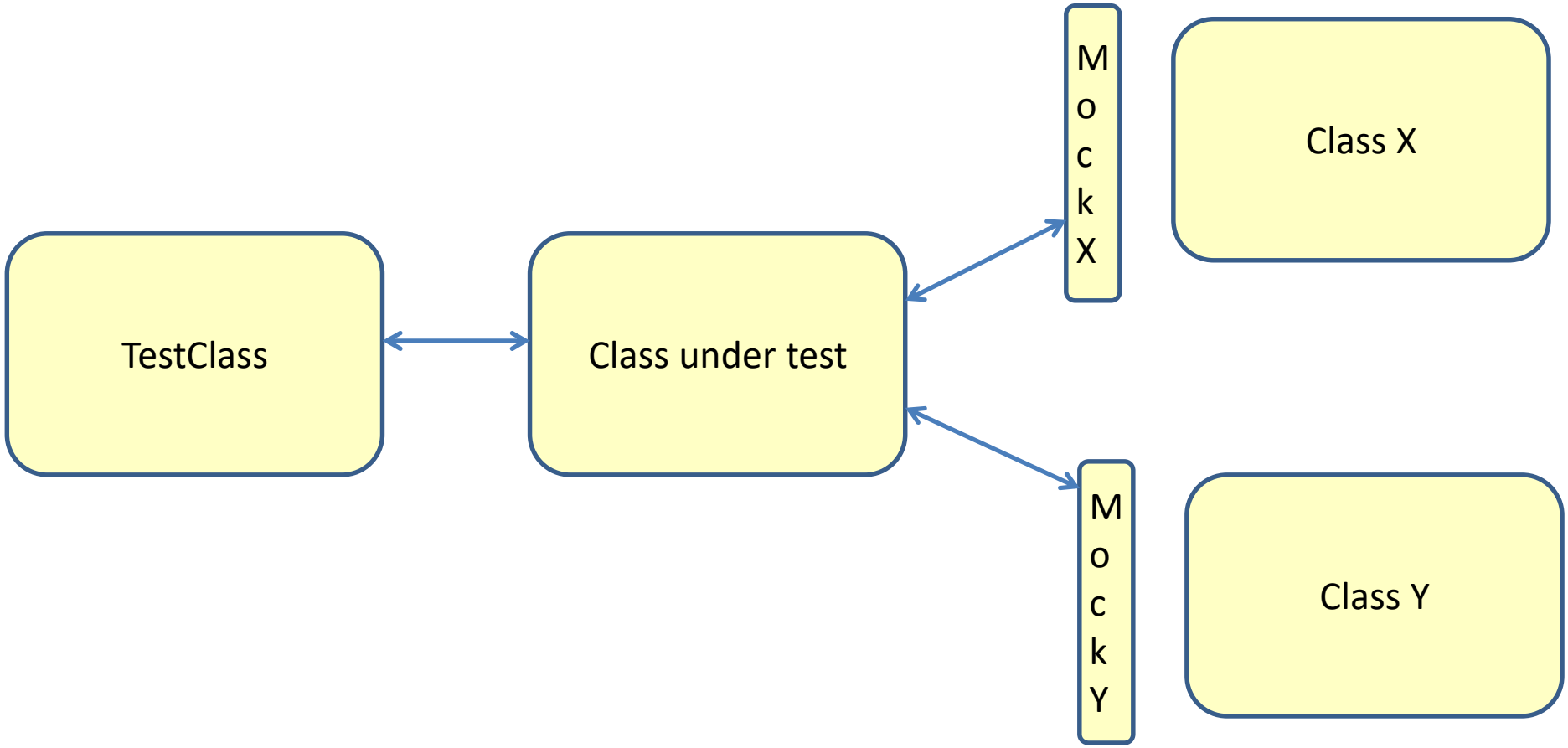
Why mock objects?



Why mock objects?



Mock objects



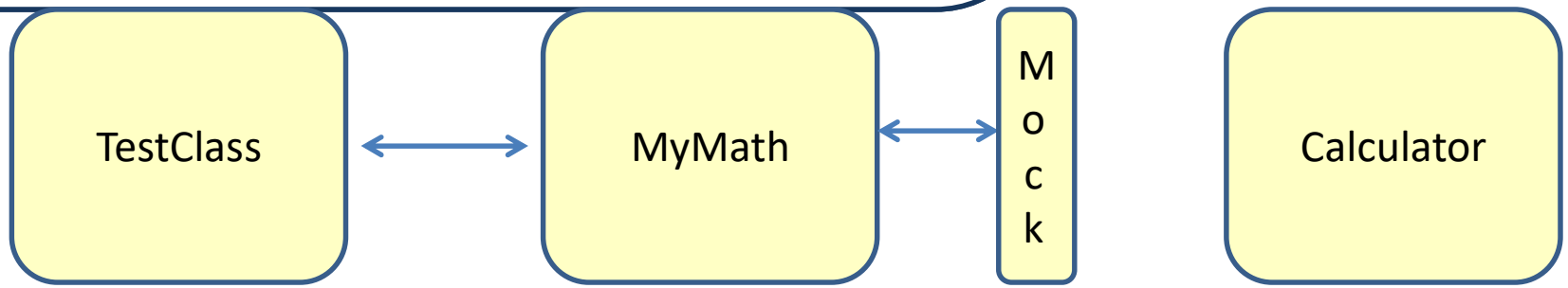
How does this work

- First tell the Mock how to behave
  - When methodX() is called return true
- Call the method on the class under test
- Verify that the mock is called correctly
  - Is methodX() called?
  - Verify the order of which the methods on the Mock are called.

Simple example

```
public interface ICalculator {
    double add(double x, double y);
    double subtract(double x, double y);
    double multiply(double x, double y);
    double divide(double x, double y);
    int add(int x, int y);
    int subtract(int x, int y);
    int multiply(int x, int y);
    double divide(int x, int y);
}
```

```
public class MyMath {
  ICalculator calc;

  public MyMath(ICalculator calc) {
    this.calc = calc;
  }

   public double doComplexCalculation(int x, int y){
     int k=calc.add(x, y);
     int l=calc.multiply(k, k);
     double result = calc.divide(l, x);
     return result;
   }
}
```



TestClass

```
@RunWith(value=MockitoJUnitRunner.class)
public class MyMathTest {

  @Mock
  ICalculator calc;

  @Test
  public void testComplexCalculation() {
    MyMath math = new MyMath(calc);
    //tell the calc stub how to behave
    when(calc.add(3,4)).thenReturn(7);
    when(calc.multiply(7,7)).thenReturn(49);
    when(calc.divide(49,3)).thenReturn(16.33333);
    //perform the method call
    double result=math.doComplexCalculation(3, 4);
    //verify the result
    assertEquals(16.33333, result, 0.001);
  }

}
```

Use the MockitoJUnitRunner

Mock the calculator

Extend the TestClass

```
@Test
public void testComplexCalculation() {
  MyMath math = new MyMath(calc);
  //tell the calc stub how to behave
  when(calc.add(3,4)).thenReturn(7);
  when(calc.multiply(7,7)).thenReturn(49);
  when(calc.divide(49,3)).thenReturn(16.33333);
  //perform the method call
  double result=math.doComplexCalculation(3, 4);
  //verify the result
  assertEquals(16.33333, result, 0.001);
  //verify that both the add, the multiply and the divide is called on the
    Calculator mock
  verify(calc, times(1)).add(3,4);
  verify(calc, times(1)).multiply(7,7);
  verify(calc, times(1)).divide(49,3);
  verifyNoMoreInteractions(calc);
  //verify that first add, then multiply and then divide is called on the
    Calculator mock
  InOrder inOrder = inOrder(calc);
  inOrder.verify(calc).add(3,4);
  inOrder.verify(calc).multiply(7,7);
  inOrder.verify(calc).divide(49,3);
}
```

Testing invocations

```
//exact number of invocations verification
verify(calc, times(2)).add(3,4);

//verification using never(). never() is an alias to times(0)
verify(calc, never()).add(3,4);

//verification using atLeast()/atMost()
verify(calc, atLeastOnce()).add(3,4);
verify(calc, atLeast(2)).add(3,4);
verify(calc, atMost(5)).add(3,4);
```

Exceptions

```
doThrow(new RuntimeException()).when(calc).divide(0,0);

//following throws RuntimeException:
calc.divide(0,0);
```

Test for no interaction

```
//verify that a mock is never called
verifyZeroInteractions(calc);
```

Iteration style stubbing

```
when(calc.getNextNumber())
   .thenReturn(1, 2, 3, 4, 5);

when(calc.add(anyInt(), anyInt()))
   .thenReturn(1, 2, 3, 4, 5);
```

Dependency Injection

```
public class MyMath {
  ICalculator calc;

  public MyMath(ICalculator calc) {
    this.calc = calc;
  }

}
```

Dependency Injection leads to flexible, easy to test code

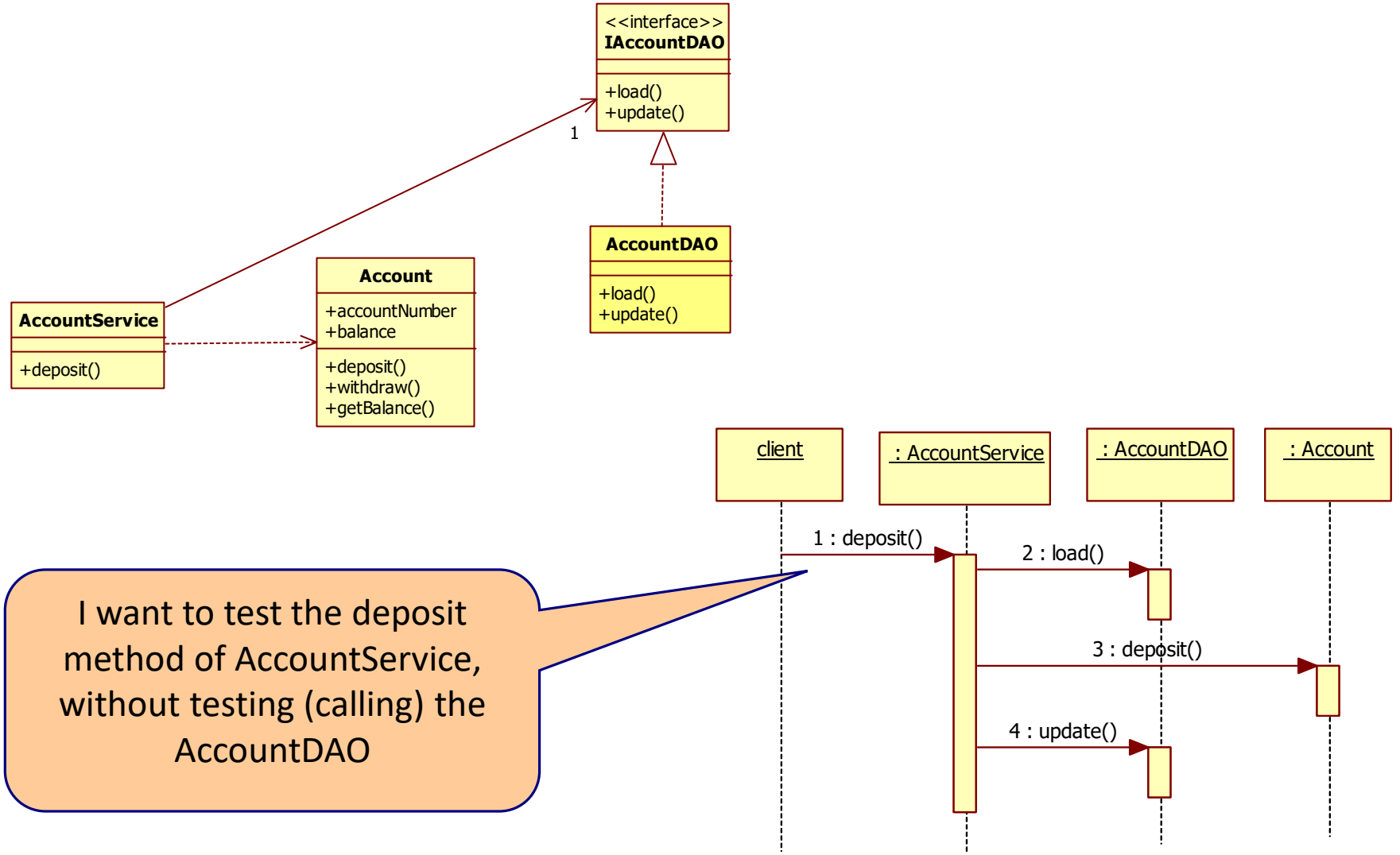
```
public interface ICalculator {
    double add(double x, double y);
    double subtract(double x, double y);
    double multiply(double x, double y);
    double divide(double x, double y);
    int add(int x, int y);
    int subtract(int x, int y);
    int multiply(int x, int y);
    double divide(int x, int y);
}
```

```
public class MyMath {
  ICalculator calc;

  public MyMath() {
    this.calc = new Calculator();
  }

}
```

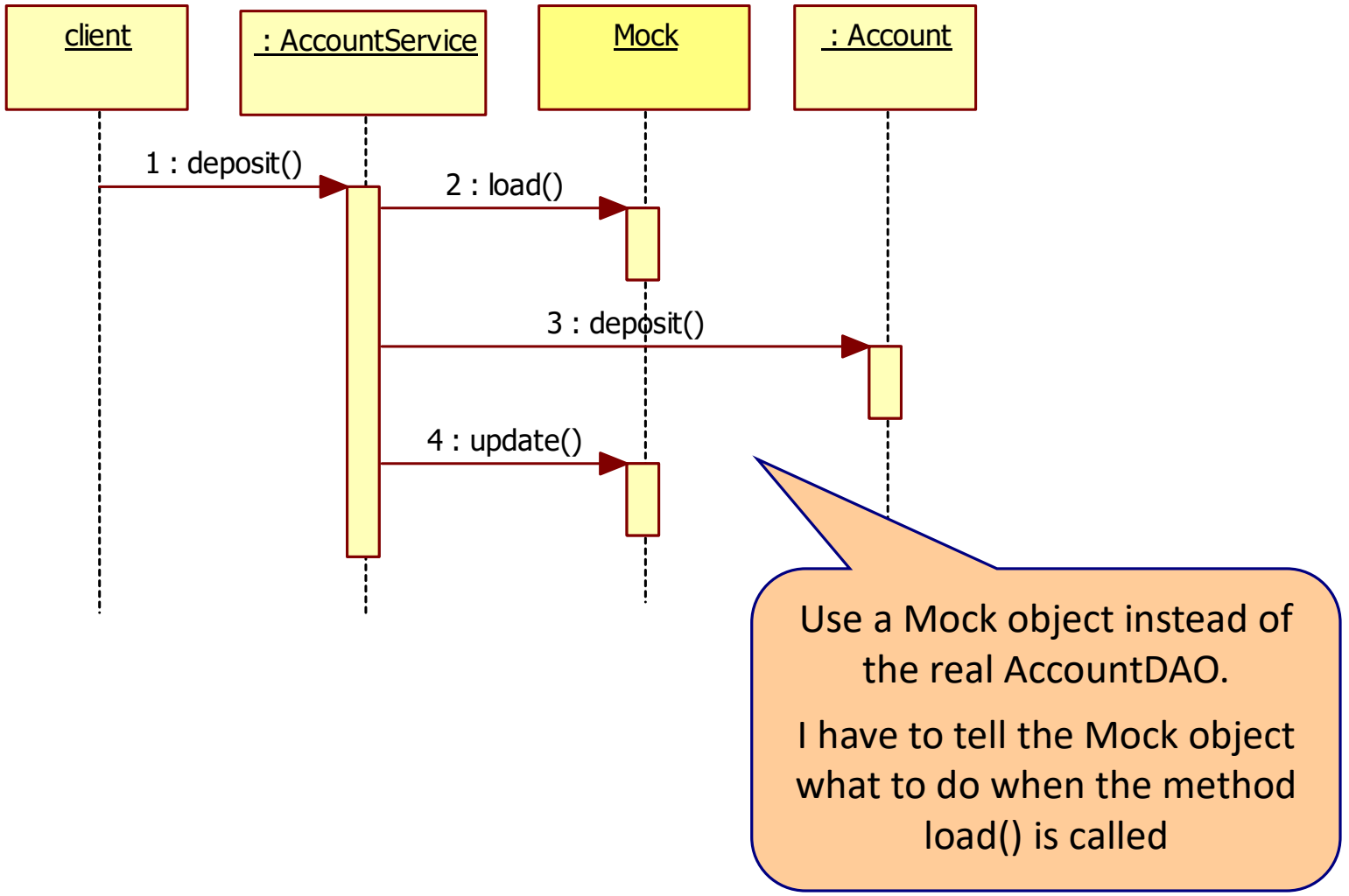
No Dependency Injection.

It is not easy to test the MyMath class without using the Calculator

Mock example



Mock solution



AccountService

```
public class AccountService {

  IAccountDAO accountDao;

  public AccountService(IAccountDAO accountDao) {
    this.accountDao = accountDao;
  }

  public double deposit(int accountnumber, double amount){
    Account account = accountDao.load(accountnumber);
    account.deposit(amount);
    accountDao.update(account);
    return account.getBalance();
  }
}
```

```
public interface IAccountDAO {
  Account load(int accountnumber);
  void update(Account account);
}
```

Account

```
public class Account {
  private long accountnumber;
  private double balance;

  public Account (long accountnr){
    this.accountnumber = accountnr;
  }
  public long getAccountnumber() {
    return accountnumber;
  }
  public void setAccountnumber(long accountnumber) {
    this.accountnumber = accountnumber;
  }
  public void deposit(double amount){
    balance=balance+amount;
  }
  public void withdraw(double amount){
    balance=balance-amount;
  }
  public double getBalance() {
    return balance;
  }
}
```

MockitoJUnitRunner

```
@RunWith(value=MockitoJUnitRunner.class)
public class AccountServiceTest {

  @Mock
  IAccountDAO accountDao;


  @Test
  public void testDeposit() {
    AccountService accService = new AccountService(accountDao);
    when(accountDao.load(101)).thenReturn(new Account(101));
    double accBalance=accService.deposit(101, 50.45);
    assertEquals(accBalance, 50.45, 0.001);
  }
}
```

Use the MockitoJUnitRunner

Perform mocking

Tell the mock what to do

Deposit will call the mock object

Expand the Test

```
@Test
public void testDeposit() {
  Account account = new Account(101);

  AccountService accService = new AccountService(accountDao);
  when(accountDao.load(101)).thenReturn(account);
  double accBalance=accService.deposit(101, 50.45);
  assertEquals(accBalance, 50.45, 0.001);

   //verify that both the load and the update method are called on AccountDAO
   verify(accountDao, times(1)).load(101);
   verify(accountDao, times(1)).update(account);

   //verify that first the load and then the update method is called on AccountDAO
   InOrder inOrder = inOrder(accountDao);
   inOrder.verify(accountDao).load(101);
   inOrder.verify(accountDao).update(account);
}
```