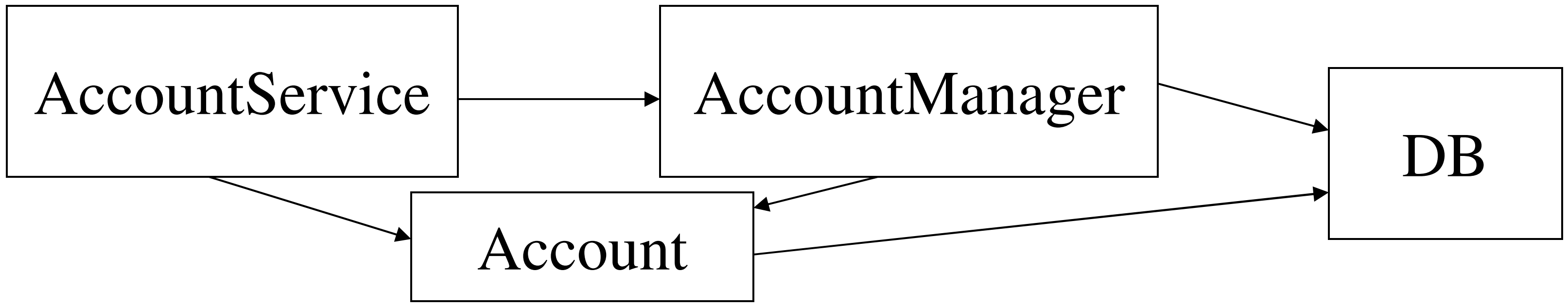
Testing with Mock Objects

- A *mock object* is an object created to stand in for an object that your code will be collaborating with. Your code can call methods on the mock object, which will deliver results as set up by your tests.
- In other words, in a fine-grained unit test, you want to test only one thing. To do this, you can create mock objects for all of the other things that your method/object under test needs to complete its job.

Testing with Mock Objects

- Example
  - Suppose we have a system that allows us to transfer money from one bank account to another.
  - We want to test just the *transfer* method which resides in the AccountService class and uses an AccountManager to retrieve Accounts, all of which are stored in a database.



Testing with Mock Objects

```
//from JUnit in Action
public class AccountService {
    private AccountManager accountManager;
    public void setAccountManager(AccountManager manager)    {
        this.accountManager = manager;
    }
    public void transfer(String senderId, String beneficiaryId, long amount)    {
        Account sender = this.accountManager.findAccountForUser(senderId);
        Account beneficiary = this.accountManager.findAccountForUser(beneficiaryId);

        sender.debit(amount);
        beneficiary.credit(amount);

        this.accountManager.updateAccount(sender);
        this.accountManager.updateAccount(beneficiary);
    }
}
```

Testing with Mock Objects

```
//from JUnit in Action
public interface AccountManager{
    Account findAccountForUser(String userId);
    void updateAccount(Account account);
}
```

- We assume there is a class that implements the AccountManager interface and interacts with the database.
- We want to create a mock object in place of the real thing.
- Account is simple enough that we will use the actual class.

Testing with Mock Objects

```
//from JUnit in Action
import java.util.Hashtable;
public class MockAccountManager implements AccountManager {
    private Hashtable accounts = new Hashtable();
    public void addAccount(String userId, Account account)
    {
        this.accounts.put(userId, account);
    }
    public Account findAccountForUser(String userId)
    {
        return (Account) this.accounts.get(userId);
    }
    public void updateAccount(Account account)
    {
        // do nothing
    }
}
```

Testing with Mock Objects

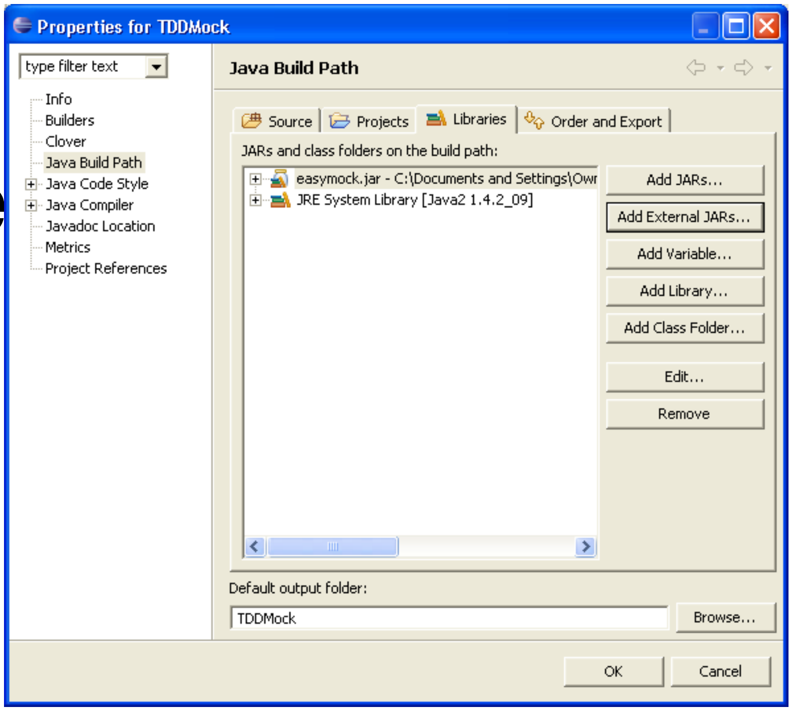
```
//from JUnit in Action
public class TestAccountService extends TestCase{
    public void testTransferOk()    {
        MockAccountManager mockAccountManager =
            new MockAccountManager();
        Account senderAccount = new Account("1", 200);
        Account beneficiaryAccount = new Account("2", 100);

        mockAccountManager.addAccount("1", senderAccount);
        mockAccountManager.addAccount("2", beneficiaryAccount);

        AccountService accountService = new AccountService();
        accountService.setAccountManager(mockAccountManager);
        accountService.transfer("1", "2", 50);
        assertEquals(150, senderAccount.getBalance());
        assertEquals(150, beneficiaryAccount.getBalance());
    }
}
```

Testing with Mock Objects

- Tests use production code.
- Sometimes a test pushes you to change your production code to make it more testable, usually making it more flexible and less coupled to other objects.
- Mock Objects are notorious for improving your production code.

EasyMock

- EasyMock is a third-party library for simplifying creating mock objects
- Download EasyMock1.2 for Java1.3 from www.easymock.org
  - EasyMock 2.0 depends on Java 1.5
- Extract download
- Add easymock.jar to proje



JARs and class folders on the build path:

easymock.jar - C:\Documents and Settings\Owr

JRE System Library [Java2 1.4.2_09]

Default output folder:

TDDMock

Testing Bank with EasyMock

```
package bank;
import java.util.Collection;
import org.easymock.MockControl;
import junit.framework.TestCase;
public class TestBank extends TestCase {
    private Bank b;
    private MockControl control;
    private Collection mock;
    protected void setUp() {
        control = MockControl.createControl(Collection.class);
        mock = (Collection) control.getMock();
        b = new Bank(mock);
    }
      public void testNumAccounts() {
              mock.size();
              control.setReturnValue(7);
              control.replay();
              assertEquals(b.getNumAccounts(),7);
              control.verify();
      }
}
```

Import MockControl

Collection is mock.
We want to test Bank,
not Collection

Recording what we we expect:
size() should be called, returning 7

Turn on mock with replay()

Check expectations with verify()

Testing getLargest() with EasyMock

```
package bank;
import java.util.SortedSet;
import org.easymock.MockControl;
import junit.framework.TestCase;
public class TestBank extends TestCase {
     …
    public void testGetLargest() {
        control = MockControl.createControl(SortedSet.class);
        SortedSet mock = (SortedSet) control.getMock();
        b = new Bank(mock);
        mock.last();
        try{
            control.setReturnValue(new Account("Richie Rich",77777,99999.99));
            control.replay();
            assertEquals(b.getLargest().getBalance(),99999.99,.01);
        } catch (Exception e) { fail("testGetLargest should not throw exception"); }
        control.verify();
    }
}
```

last() should be called on mock

When to use Mock Objects

- When the real object has non-deterministic behavior (e.g. a db that is always changing)
- When the real object is difficult to set up
- When the real object has behavior that is hard to cause (such as a network error)
- When the real object is slow
- When the real object has (or is) a UI
- When the test needs to query the object, but the queries are not available in the real object
- When the real object does not yet exist

* from http://c2.com/cgi/wiki?MockObject