e

Introduction to JavaEE

Using Hibernate 3.2.x and Spring 2.0.x

by Bill Six

Table of Contents

1 Introduction 1
- 1.1 Road Map 1
- 1.2 Layered Architecture 1

2 The User Stories 3
- 2.1 Business Domain 3
- 2.2 Roles 3
- 2.3 User Stories 3

3 The Domain Model 5

4 The Proxy Pattern 7
- 4.1 Dynamic Proxies 7
  - 4.1.1 Proxy Example - Smalltalk-80 7
  - 4.1.2 Dynamic Proxy Example - Spring 8

5 Data Access Layer 10
- 5.1 Structural Mismatch 10
  - 5.1.1 Object structure 10
  - 5.1.2 Relational structure 11
  - 5.1.3 Overcoming the Structural Mismatch 12
- 5.2 Behavioral Mismatch 14
  - 5.2.1 Overcoming the Behavioral Mismatch 14
- 5.3 Unit of Work 14
- 5.4 N+1 Selects Problem 14

6 Service Layer 16
- 6.1 Service Unit Test 16
- 6.2 Service Implementation 17
- 6.3 Transaction Management 18

7 Conclusion 19

1 Introduction

The purpose of this paper is to show how to use Hibernate and Spring to structure an enterprise application. To facilitate the readers understanding, I have created a simple application which almost everybody uses - an ATM.

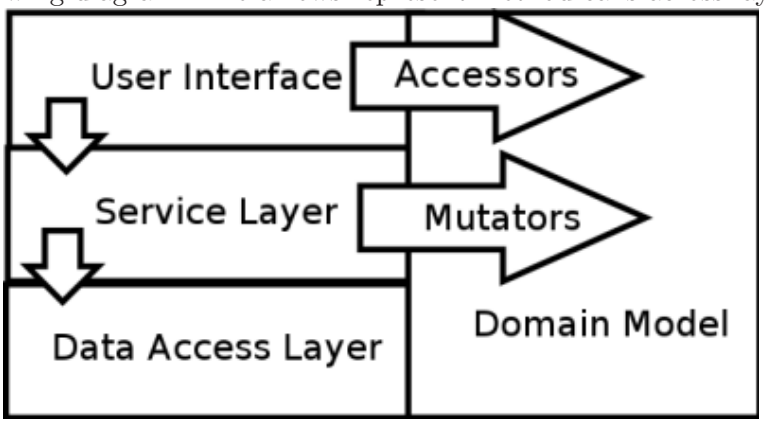
I wrote this paper because it's the paper I would have liked to have read when I first started work after college.

1.1 Road Map

- Chapter 2 - The User Stories. All examples in this paper will be driven by User Stories.
- Chapter 3 - The Domain Model. This section shows how to create a Domain Model to implement all of the User Stories which involve the business logic.
- Chapter 4 - The Proxy Pattern, introduces a core technology that Hibernate and Spring use internally.
- Chapter 5 - Data Access Layer, introduces Hibernate, an object-relational mapper used to save a domain model to a relational database. It will discuss the basics of the impedance mismatch between objects and relational databases.
- Chapter 6 - Service Layer, is a short introduction to the Spring Framework. It will show how to use Spring to simplify transaction management in the service layer.

1.2 Layered Architecture

Ideally, organizations structure their enterprise applications as depicted in the following diagram. The arrows represent method calls across layers.



The Domain Model is the core of an enterprise application. A solid Domain Model encapsulates the data and behavior of the business processes. As my

feeble diagram attempts to convey, the Domain Model should be the focus of the entire application.

The responsibility of the User Interface is to display data to the user. To achieve this, the User Interface calls the Service Layer to retrieve a sub-graph of the persistent Domain Model object graph. Upon receiving this sub-graph, the User Interface calls accessor methods, and displays the result. After rendering this data, the end user of the application needs to manipulate it. The User Interface calls upon the Service Layer to mutate the data.

The Service Layer is a very thin layer which provides application logic such as management of transactions, authentication and authorization, and logging. As such, the Service Layer mainly delegates method calls to the Domain Model and Data Access Layer. Because of its limited responsibilities, the Service Layer should fairly small.

The responsibility of the Data Access Layer is to retrieve and persist sub-graphs of the persistent Domain Model object graph.

2 The User Stories

This section describes the User Stories that will drive the application development.[1]

2.1 Business Domain

The Business Domain is how end-users refer to the entities involved. The Business Domain is developed as part of object-oriented analysis; as such, it is programming language-agnostic.

In the ATM application, the following is the Business Domain:

- Account - An account is the base unit in which a person can transfer funds. Each account has an associated collection of fund transfers.
- Fund Transfer - Any action which involves increasing or decreasing the current balance of an account. A Fund Transfer needs to record the balance before the fund transfer, after the fund transfer, and the time of the fund transfer.
- Deposit - A fund transfer which adds money to the account.
- Withdraw - A fund transfer which removes money from the account.

2.2 Roles

I find it helpful to not only define the roles for people using the application, but also to define named users with sample data. I find this helps to drive unit tests.

- Bill - a person who has an account at our ATM.
  - Username - bill
  - Password - password
  - Initial balance = $100

2.3 User Stories

The user stories are defined in the following format

- Brief description of the user story
  - Acceptance criteria

- Bill provides correct username and password before he can access any of the ATM's services
  - Bill does not provide the correct username and password, and is unable to access any services

[1] You should read "User Stories Applied" by Mike Cohn

  - Bill does provide the correct username and password, and is able to access all services
- Bill checks his current balance
  - Bill's sees that he has $100.
- Bill withdraws money from his account
  - Bill's withdraws $50, and his current balance is now $50. A Fund Transfer is created to reflect this transaction, which correctly records the before and after balance.
- Bill deposits money to his account
  - Bill's deposits $50, and his current balance is now $150. A Fund Transfer is created to reflect this transaction, which correctly records the before and after balance.
- Bill withdraws $50, deposits $25, is able to see a history of all of his fund transfers.
  - Bill is able to see all of the fund transfers.

3 The Domain Model

Rather than creating a class diagram to describe the Domain Model, I find that a good suite of unit tests not only drive the creation of a good Domain Model, they also provide adequate documentation.

```
public class AccountTest extends TestCase {
    public AccountTest( String testName ) {
        super( testName );
    }
    public static Test suite() {
        return new TestSuite( AccountTest.class );
    }
    public void setUp() {
        account = new Account("bill", "password", 100.00);
    }
    public void testPasswordIsValid() {
        assertTrue(account.passwordIsValid("password"));
        assertFalse(account.passwordIsValid("passw"));
    }
```

- Bill checks his current balance
  - Bill's sees that he has $100.

```
public void testCurrentBalance() {
    assertTrue(account.getBalance() == 100.0);
}
```

- Bill deposits money to his account
  - Bill's deposits $50, and his current balance is now $150. A Fund Transfer is created to reflect this transaction, which correctly records the before and after balance.

```
public void testDeposit() {
    account.deposit(50.00);
    assertTrue(account.getBalance() == 150.0);
    Set<FundTransfer> fundTransfers = account.getFundTransferHistory();
    assertTrue(fundTransfers.size()==1);
    FundTransfer fundTransfer = fundTransfers.iterator().next();
    assertTrue(fundTransfer.getBalanceAfterFundTransfer() == 150.0);
    assertTrue(fundTransfer.getBalanceBeforeFundTransfer() == 100.0);
}
```

- Bill withdraws money from his account
  - Bill's withdraws $50, and his current balance is now $50. A Fund Transfer is created to reflect this transaction, which correctly records the before and after balance.

```
public void testWithdraw() {
    account.withdraw(50.00);
    assertTrue(account.getBalance() == 50.0);
    Set<FundTransfer> fundTransfers = account.getFundTransferHistory();
```

```
        assertTrue(fundTransfers.size()==1);
        FundTransfer fundTransfer = fundTransfers.iterator().next();
        assertTrue(fundTransfer.getBalanceAfterFundTransfer() == 50.0);
        assertTrue(fundTransfer.getBalanceBeforeFundTransfer() == 100.0);
    }
```

- Bill withdraws $50, deposits $25, is able to see a history of all of his fund transfers.
  - Bill is able to see all of the fund transfers.

```
    public void testFundTransferHistory() {
        account.withdraw(50.00);
        account.deposit(25.0);
        for(FundTransfer fundTransfer : account.getFundTransferHistory()) {
            if(fundTransfer.getBalanceAfterFundTransfer() != 50.0 &&
                    fundTransfer.getBalanceAfterFundTransfer() != 75.0 ) {
                fail("Incorrect set of fund transfers");
            }
        }
    }
    private Account account;
}
```

4 The Proxy Pattern

This section has nothing to do with the User Stories, but before we get into Hibernate and Spring, it's important to understand the Proxy pattern. Modern Java frameworks such as Spring and Hibernate make heavy use of dynamic proxies; specifically Hibernate uses them to model unresolved references in the Domain Model sub-graph, and Spring uses them to simplify transaction management.

4.1 Dynamic Proxies

Dynamic, message-passing object-oriented languages such as Smalltalk-80, Objective C, Ruby, and Groovy allow an object to inspect messages which it receives should it not have a method which correspond to the message's selector[1]. The object then decides to execute whatever action(s) it sees fit based on the selector and arguments of the message. This feature allows a developer to create proxy objects which forward these unknown messages to another "target" object.[2][3]

While dynamic languages support the Proxy pattern via the language's semantics, the statically-typed nature of Java mandates that an object cannot receive a message, or MethodInvocation in Spring's vernacular, for which it does not have a corresponding Method. As such, a purely statically-typed system cannot allow for proxy objects without manual delegation to the proxy. In order to get around this limitation, Sun introduced the notion of a dynamic proxy in JDK 1.3[4] , which is a Java class that can be configured as a proxy to any number of Java interfaces at runtime: thus allowing safe type-casting of the proxy object to a set of target interfaces. Sun's implementation of this uses reflection. Around the same time, an open source project called CGLIB appeared, allowing similar proxying behavior via byte-code enhancement instead of reflection. The additional benefit of CGLIB is that it removed the restriction that a dynamic proxy can only proxy instances of interfaces, thus providing the ability to proxy an instance of a class.

4.1.1 Proxy Example - Smalltalk-80

The following example shows how proxy objects are implemented in Smalltalk. Since the proxy object intercepts the method call to the target object, the proxy object can perform logic before and after the actual method call is forwarded to the target.

```
ProtoObject subclass: #CustomProxyClass
```

[1] The method signature and in Java parlance

[2] http://en.wikipedia.org/wiki/Proxy_pattern

[3] In languages like Lisp where code is data, macros are a more elegant solution

[4] I'm not quite sure why the name "dynamic" proxy is used instead of just "proxy". I'm guessing it's because you can statically type a proxy class which implements a "dynamic" set of interfaces

```
   instanceVariableNames: 'target'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples'

  initializeTarget: t
   target := t

  doesNotUnderstand: aMessage
   | answer |
   Transcript show: 'Here I would open a transaction'; cr.
   answer := target perform: aMessage selector
                  withArguments: aMessage arguments.
   Transcript show: 'Here commit the transaction unless I caught an unre-
coverable exception!';cr.
   ^ answer .

 Object subclass: #CalculatorService
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples

  add: first and: second
   ^ first + second .

 ClassTestCase subclass: #CustomProxyClassTest
   instanceVariableNames: 'calculatorService'
   classVariableNames: ''
   poolDictionaries: ''
   category: 'Examples

  setUp
   calculatorService := CustomProxyClass target: (CalculatorService new).

  testCalculatorService
   self assert: ( (calculatorService add: 1 and: 2) = 3)
```

4.1.2 Dynamic Proxy Example - Spring

The following shows the same example implemented using the Java's dynamic proxies, initialized through Spring.

```
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
public class TransactionInterceptor implements MethodInterceptor {
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Here I would open a transaction");
        Object answer = invocation.proceed();
        System.out.println("Here commit the transaction unless I \
caught an unrecoverable exception!");
        return answer;
```

```
    }
}
public interface CalculatorService {
    public Integer add(Integer first, Integer second);
}
public class CalculatorServiceImplementation
       implements CalculatorService {
    public Integer add(Integer first, Integer second) {
        return first + second;
    }
}
import org.springframework.aop.framework.ProxyFactoryBean;

public class CalculatorServiceTest extends TestCase {
    public CalculatorServiceTest( String testName ) {
        super( testName );
    }
    public static Test suite() {
        return new TestSuite( CalculatorServiceTest.class );
    }
    public void setUp() throws Exception{
        CalculatorServiceImplementation target =
                        new CalculatorServiceImplementation();
        ProxyFactoryBean factory = new ProxyFactoryBean();
        factory.addAdvice(new TransactionInterceptor());
        factory.setTarget(target);
        factory.setProxyInterfaces(new Class[]{CalculatorService.class});
        calculatorService = (CalculatorService) factory.getObject();
    }
    public void testValidAccess() {
        assertTrue(calculatorService.add(4,5) == 9);
    }
    CalculatorService calculatorService ;
}
```

5 Data Access Layer

The object-oriented model and the relational model have structural and behavioral differences, called the impedance mismatch.

5.1 Structural Mismatch

Granularity The object model in the object-oriented application may be more fine-grained than the relational model.

Subtypes

In the object-oriented model, objects can have superclasses and subclasses. Relational databases do not have this notion.

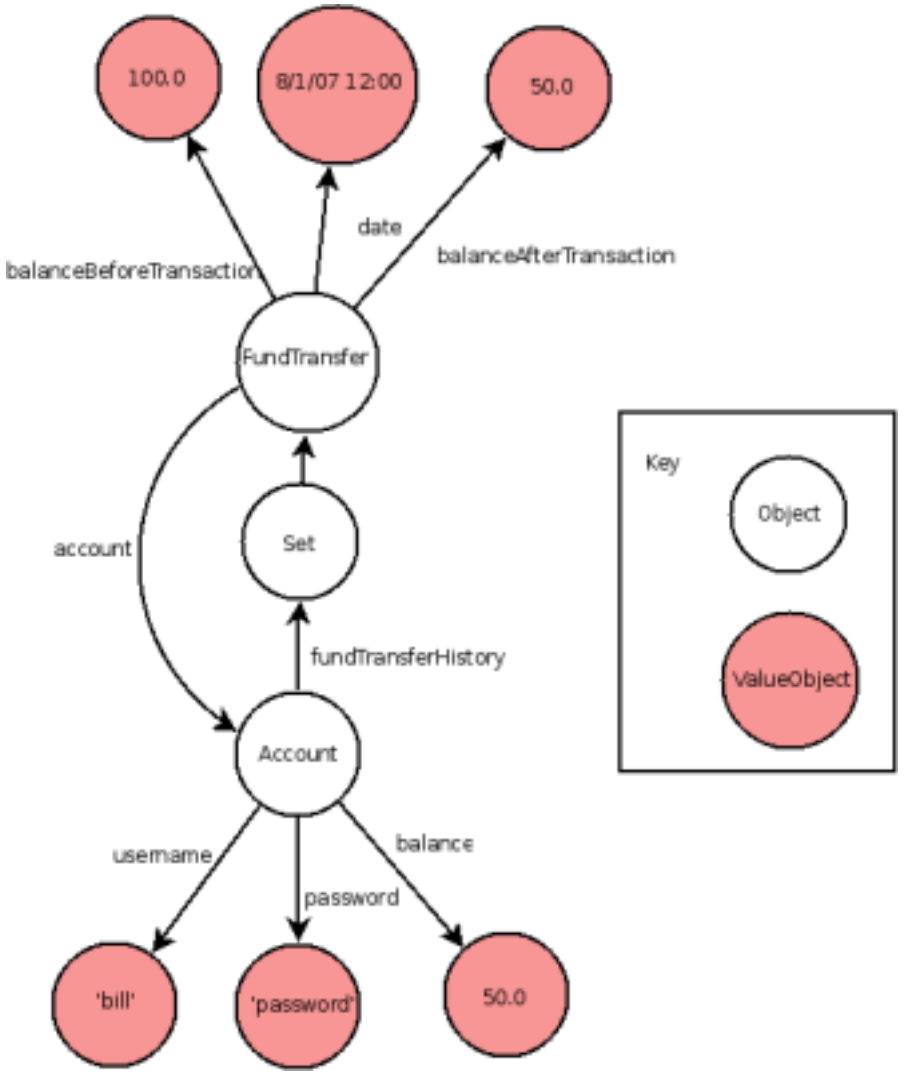
Associations

In the object-oriented model, one-to-one, one-to-many, many-to-one, and many-to-many associations are implemented natively using the object graph. In the relational model, references are represented using joins on foreign keys

5.1.1 Object structure

Object oriented languages model data as an object graph, where an object is represented by a node, and its collaborators are edges.

```
public class Account extends BaseDomainObject{
    ......
    private String username;
    private String password;
    private Double balance;
    private Set<FundTransfer> fundTransferHistory;
}
public class FundTransfer extends BaseDomainObject{
    ......
    private Account account;
    private Double balanceBeforeFundTransfer;
    private Double balanceAfterFundTransfer;
    private Calendar date;
}
```



[1]

5.1.2 Relational structure

Relational databases model data as tables. Data is stored in columns. Rows are uniquely identified by a primary key. Collections are represented by foreign keys which map to another tables primary key.

```
CREATE SEQUENCE hibernate_sequence
  INCREMENT 1
  MINVALUE 1
```

[1] from Fowler's PoEAA, a Value object is "a small simple object, like money or a date range, whose equality isn't based on identity."

```
  MAXVALUE 9223372036854775807
  START 3
  CACHE 1;

CREATE TABLE account
(
  id int8 NOT NULL,
  username varchar(255),
  password varchar(255),
  balance float8,
  version int8,
  CONSTRAINT account_pkey PRIMARY KEY (id)
)
WITHOUT OIDS;

CREATE TABLE fundtransfer
(
  id int8 NOT NULL,
  balancebeforefundtransfer float8,
  balanceafterfundtransfer float8,
  date timestamp,
  version int8,
  account_id int8,
  CONSTRAINT fundtransfer_pkey PRIMARY KEY (id),
  CONSTRAINT fk5995fdf0980ea509 FOREIGN KEY (account_id)
      REFERENCES account (id) MATCH SIMPLE
      ON UPDATE NO ACTION ON DELETE NO ACTION
)
WITHOUT OIDS;
```

Table account

id	username	password	balance	version
1	bill	password	50.0	1

Table fundtransfer

id	before	after	date	version	account_id
1	100.0	50.0	8/1/07 12:00	1	1

5.1.3 Overcoming the Structural Mismatch

To overcome these structural differences, Hibernate uses Java 5's annotations as meta-data.

```
@Entity()
@AccessType("field")
public class Account extends BaseDomainObject{
    ......
    private String username;
    private String password;
```

```
    private Double balance;
    @OneToMany(cascade = CascadeType.ALL)
    @JoinColumn(name="ACCOUNT_ID")
    private Set<FundTransfer> fundTransferHistory;
    @Id @GeneratedValue private Long id;
    @Version private Long version;
}

@Entity()
@AccessType("field")
public class FundTransfer extends BaseDomainObject{
    ......
    @ManyToOne( cascade = CascadeType.ALL )
    @JoinColumn(name="ACCOUNT_ID")
    private Account account;
    private Double balanceBeforeFundTransfer;
    private Double balanceAfterFundTransfer;
    @Temporal(TemporalType.TIMESTAMP) private Calendar date;
    @Id @GeneratedValue private Long id;
    @Version private Long version;
}
```

- The @Entity() annotation lets Hibernate know that this object corresponds to a table, which by default has the same name as the class. By default, all Value objects map to columns of the same name.
- The @AccessType("field") annotation tells Hibernate to populate the fields directly, instead of the default method of calling the accessor methods.
- The @OneToMany and @ManyToOne (along with a @JoinColumn) annotations tell Hibernate that the collection is represented in the relational database though foreign keys.
- The @Id @GeneratedValue annotations tell Hibernate that this is the primary key, and that Hibernate should generate a value upon insertion into the database.

Granularity

No user stories in the ATM application have a granularity mismatch. For more information, search for "Embedded objects" in the online Hibernate documentation.

Subtypes

No user stories in the ATM application have have the inheritance mismatch. If you need to learn more, consult Hibernate's online documentation, and Martin Fowler's Patterns of Enterprise Application Architecture.

Associations

Using the meta-data, Hibernate performs a join on the tables, and returns an instance of the Set interface.

5.2 Behavioral Mismatch

Object Identity

In the object-oriented model, two objects are equal if they are located in the same place in memory. In a relational model, two rows are equal if they have the same primary key.

5.2.1 Overcoming the Behavioral Mismatch

Object Identity

When using CRUD operations in Hibernate, Hibernate doesn't immediately execute your calls, it operates through Hibernate's Level 1 cache. This thread-scoped[2] cache is a local copy of the user accessed sub-graph of the persistent object graph. As such, the Level 1 cache acts as a buffer which ensures that each unique entry in the database corresponds to the same Java object in memory.[3]

5.3 Unit of Work

When the Service Layer calls mutator methods on the Domain model, the Domain Model needs to flush the modified sub-graph to the database upon the completion of the transaction.

Hibernate could

- Flush everything in the Level 1 cache to the database. However, not all objects in the Level 1 cache have been modified, which results in unnecessary SQL updates.
- Force the developer to manually register which objects were modified, flushing only "dirty" objects.
- Transparently register modified objects, flushing only "dirty" objects.

Hibernate uses the third method.

5.4 N+1 Selects Problem

When Bill first logs into the ATM, he may want to check his balance, withdraw, deposit, or view the entire history of his fund transfers. These operations require different sub-graphs of the object graph to be presented to the UI, and as such, it would be nice to be able to specify the exact sub-graph that we would like to retrieve.

When Bill first logs in, Hibernate will pull back his Account object into the Level 1 cache, along with all of the Value objects that the Account object references. Should Hibernate automatically load the entire object graph into

[2] Because the cache is thread-scoped, a developer only needs to set correct transaction isolation levels to deal with concurrency

[3] This is described as the "Identity Map" in Martin Fowler's Patterns of Enterprise Application Architecture

the Level 1 cache? Assuming he just wants to check his current balance, he has no need to pull back all of his fund transfers.

Hibernate could

- Load every object into the Level 1 cache that is referenced by any object in the Level 1 cache
- Lazy-load all non-Value objects referenced by any objects in the Level 1 cache. The Set of FundTransfers will not be a null reference[4], it will be a reference to a dynamic proxy[5] which can pull back the FundTransfers only when it is required.
- Load a user-specified sub-graph of the persistent object graph into the Level 1 cache.

The first option is by far the worst. There can be gigabytes of data stored in the database, causing a simple read to pull back a large portion of the object graph when only a subset is required.

The second option is more sensible, and this was the model for EJB's before EJB 3.0. The problem with this model is that you may know that you want a specified sub-graph of the full object graph. Lazy-loading will force multiple database roundtrips to fetch the sub-graph (hence the name "N+1 selects").

Hibernate (and now EJB 3.0) allows the third model, enabling loading a sub-graph in one database roundtrip.

[4] as that would violate consistency of the database

[5] In this case, the dynamic proxy is an implementation of the "Lazy Load" pattern decribed in PoEAA

6 Service Layer

The Service Layer is a thin layer on top of a Domain Model which provides application logic such as management of transactions, authentication and authorization, and logging.

There is only one User Story which deals with authorization.

- Bill provides correct username and password before he can access any of the ATM's services
  - Bill does not provide the correct username and password, and is unable to access any services
  - Bill does provide the correct username and password, and is able to access all services

To implement this, I create a base interface for the Service Layer called Authenticatable. This class enables Bill to log in, and any Service Layer interface which implements Authenticatable has access to the current account via the getAuthenticatedAccount() method.

```
@Transactional(isolation=Isolation.DEFAULT,
                propagation=Propagation.REQUIRED)
public interface Authenticatable {
    public boolean authenticate(String username, String password);
    public Account getAuthenticatedAccount();
}
```

The Service Layer is also supposed to act as a thin layer over the Domain Model, exposing methods that the User Interface needs to call.

```
@Transactional(isolation=Isolation.DEFAULT,
                propagation=Propagation.REQUIRED)
public interface ATMService extends Authenticatable{
    public Double getBalance();
    public void deposit(Double amountToDeposit);
    public void withdraw(Double amountToWithdraw);
    public Account fetchFundTransferHistory();
}
```

The `@Transactional` annotation are Spring's way of setting isolation and propagation levels for the Service.

6.1 Service Unit Test

The unit test for the Service Layer could then be implemented as follows.

```
public class ATMServiceTest extends TestCase {
    public ATMServiceTest( String testName ) {
        super( testName );
    }
    public static Test suite() {
        return new TestSuite( ATMServiceTest.class );
    }
    .......
    public void testIncorrectLogin() {
```

```
        if(atmService.authenticate("ll","password")) {
            fail("Invalid user should not be able to log in");
        }
        try{
            atmService.withdraw(100.0);
            fail("Invalid user should not be able to use the ATM Service");
        } catch(IllegalStateException ise) {
        }
    }
    public void testCurrentBalance() {
        if( !atmService.authenticate("bill","password")) {
            fail("Valid user should be able to log in");
        }
        assertTrue(atmService.getBalance() == 100.0);
    }
    .......
}
```

6.2 Service Implementation

```
public class ATMServiceImplementation implements ATMService{
  public ATMServiceImplementation(SessionFactory sessionFactory) {
      this.sessionFactory = sessionFactory;
  }
```

- Calling sessionFactory.getCurrrentSession() gets Hibernate's Session, gets the thread-local Session. The Query uses Hibernate's HQL[1].

```
  public boolean authenticate(String username, String password) {
      Account account = (Account) sessionFactory.getCurrentSession()
      .createQuery("from Account as account where account.username like :username")
      .setParameter("username", username, Hibernate.STRING)
      .uniqueResult();
      if(account == null) {
          return false;
      }

      if(account.passwordIsValid(password)) {
          this.account = account;
          return true;
      }
      return false;
  }
```

- Calling saveOrUpdate(object) on the session reattaches the "detached" instance into the current transaction. Although, since the account object will not have changed in the UI, calling load(Account.class, this.account.getId()) would perform better.

[1] HQL queries specify the sub-graph you want to load into the Level 1 cache

```
    public Double getBalance() {
        sessionFactory.getCurrentSession().saveOrUpdate(this.account);
        return this.account.getBalance();
    }
    public void deposit(Double amountToDeposit) {
        sessionFactory.getCurrentSession().saveOrUpdate(this.account);
        this.account.deposit(amountToDeposit);
    }
    public void withdraw(Double amountToWithdraw) {
        sessionFactory.getCurrentSession().saveOrUpdate(this.account);
        this.account.withdraw(amountToWithdraw);
    }
```

- Since the FundTransfer set is lazily loaded, calling Hibernate.initialize(objects) forces the fundHistory dynamic proxy to actually fetch the collection into the Level 1 cache.

```
    public Account fetchFundTransferHistory() {
        sessionFactory.getCurrentSession().saveOrUpdate(this.account);
        Hibernate.initialize(account.getFundTransferHistory());
        return account;
    }
    public Account getAuthenticatedAccount() {
        return this.account;
    }
    private SessionFactory sessionFactory;
    private Account account;
}
```

6.3 Transaction Management

Spring defines an interface called the PlatformTransactionManager, for which there are multiple implementations; allowing you to develop your Service Layer code completely independently of a specific transaction technology. These include the HibernateTransactionManager, JTATransactionManager, and others. By having the PlatformTransactionManager interface, transaction management code in the Service Layer can be developed regardless of the underlying transaction management scheme.

7 Conclusion

Hibernate solves the object-relational impedance mismatch while performing efficiently. Spring simplifies transaction management by offering a uniform transaction management scheme. JavaServer Faces faciliates exposing the Domain Model in web forms almost as easily as with rich clients.

This document, and all sample code can be downloaded from anonymously via subversion.

```
svn co https://atmexample.svn.sourceforge.net/svnroot/atmexample
```