Improving Your Application’s Design by Using Real Objects

Chris Richardson
Author of *POJOs in Action*
Chris Richardson Consulting, Inc

http://www.chrisrichardson.net
About Chris

- Grew up in England
- Live in Oakland, CA
- Over twenty years of software development experience
  - Building object-oriented software since 1986
  - Using Java since 1996
  - Using J2EE since 1999
- Author of POJOs in Action
- Speaker at JavaOne, JavaPolis, NFJS, JUGs, ....
- Chair of the eBIG Java SIG in Oakland ([www.ebig.org](http://www.ebig.org))
- Run a consulting and training company that helps organizations build better software faster
Overall presentation goal

Learn how to improve application design with truly object-oriented business logic
Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- Domain model building blocks
- Role of frameworks
- Common code smells
- Refactoring existing code
- Obstacles to OO
Designing business logic

- Some enterprise frameworks (e.g. Spring) promote good design practices:
  - Dependency injection for loose coupling
  - AOP for handling cross cutting concerns
- But **you** must decide how to structure your business logic:
  - Domain Model pattern – object-oriented
  - Transaction Script pattern – procedural
- Choice of pattern impacts ease of:
  - Development, testing, maintainability, ...
Lots of procedural Java code

Java is an object-oriented language

AND

Object-oriented design is a better way to tackle complexity

YET

Many complex enterprise Java applications are written in a procedural style
Example banking application

Transfer Money

Transfer Between Your Accounts

<table>
<thead>
<tr>
<th>Transfer From Account</th>
<th>SAVINGS (Avail. balance = $1,155.98)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transfer To Account</td>
<td>CHECKING (Avail. balance = $140.90)</td>
</tr>
</tbody>
</table>

Amount

Transfer Description (optional)

Descriptions appear for checking, savings, money market or market rate accounts only.
Example procedural design

Web Tier

Behavior

An anemic domain model

State

Transaction script

Missing classes for key concepts

MoneyTransferService

BankingTransaction transfer (fromAccountId, toAccountId, amount)

AccountRepository

Account findAccount (accountId)

BankingTransactionRepository

addTransaction ()

Account

accountId balance overdrawType

BankingTransaction

build date
Example procedural code

```java
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

    public BankingTransaction transfer(String fromAccountId, String toAccountId,
            double amount) throws MoneyTransferException {
        Account fromAccount = accountDAO.findAccount(fromAccountId);
        Account toAccount = accountDAO.findAccount(toAccountId);
        assert amount > 0;
        double newBalance = fromAccount.getBalance() - amount;
        switch (fromAccount.getOverdraftPolicy()) {
            case Account.NEVER:
                if (newBalance < 0)        throw new MoneyTransferException("In sufficient funds");
                break;
            case Account.ALLOWED:
                Calendar then = Calendar.getInstance();
                then.setTime(fromAccount.getDateOpened());
                Calendar now = Calendar.getInstance();
                double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
                int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);
                if (monthsOpened < 0) {
                    yearsOpened--;
                    monthsOpened += 12;
                }
                yearsOpened = yearsOpened + (monthsOpened / 12.0);
                if (yearsOpened < fromAccount.getRequiredYearsOpen() || newBalance < fromAccount.getLimit())
                    throw new MoneyTransferException("Limit exceeded");
                break;
            default:
                throw new MoneyTransferException("Unknown overdraft type: 
                        + fromAccount.getOverdraftPolicy()");
        }
        fromAccount.setBalance(newBalance);
        toAccount.setBalance(toAccount.getBalance() + amount);
        TransferTransaction txn = new TransferTransaction(fromAccount, toAccount,
                amount, new Date());
        bankingTransactionDAO.addTransaction(txn);
        return txn;
    }

    public class Account {
        public static final int NEVER = 1;
        public static final int ALLOWED = 2;
        private int id;
        private double balance;
        private int overdraftPolicy;
        private String accountId;
        private Date dateOpened;
        private double requiredYearsOpen;
        private double limit;
        Account() {} 
        public Account(String accountId, double balance, int 
                overdraftPolicy, 
                Date dateOpened, double requiredYearsOpen, 
                double limit) {
            .....
        }
        public int getId()  {return id;}public String getAccountId() {return accountId;}public void setBalance(double balance) {
        ...
    }

    public Account(String accountId, double balance, int 
            overdraftPolicy, 
            Date dateOpened, double requiredYearsOpen, 
            double limit) {
        .....
    }

    public int getOverdraftPolicy() { return overdraftPolicy; }
    public int getRequiredYearsOpen() {
        return requiredYearsOpen;
    }
    public int getBalance() { return balance; }
    public String getAccountId() { return accountId; }
    public void setBalance(int balance) {
        .....
    }
    public int getBalance() { return balance; }
    public void setBalance(double balance) {
        .....
    }
    public double getOverdraftPolicy() { return overdraftPolicy; }
    public double getBalance() { return balance; }
    public int getRequiredYearsOpen() {
        return requiredYearsOpen;
    }
    public double getBalance() { return balance; }
    public double getLimit() {return limit; }
}
```
A seductive programming style

- Implementing new functionality is easy
  - Add a new transaction script
  - Add code to a new transaction script
- No need to do any real design, e.g.
  - Create new classes
  - Determine responsibilities
Unable to handle complexity

- Works well for simple business logic
  - e.g. the example wasn’t that bad
- But with complex business logic:
  - Large transaction scripts: 100s/1000s LOC
  - Difficult/impossible to understand, test, and maintain
- What’s worse: business logic has a habit of growing
  - New requirements ⇒ Add a few more lines to the transaction script
  - Many new requirements ⇒ big mess
  - Soon or later you end up with unmaintainable code
Today – rich domain models are growing in popularity

- **POJOs**
  - Plain Old Java Objects
  - Leverage OO features of Java

- **O/R mapping frameworks for persisting POJOs:**
  - Hibernate
  - Java Persistence API
  - ...

- **Spring AOP and AspectJ for handling cross-cutting concerns:**
  - Transaction management
  - Security
  - Logging
  - Auditing
  - ...

© Copyright 2007, Chris Richardson
Agenda

- Where are the real objects?
- **Overview of the Domain Model pattern**
- Domain model building blocks
- Role of frameworks
- Common code smells
- Refactoring existing code
- Obstacles to OO
Using the Domain Model Pattern

- Business logic spread amongst a collection of classes
- Many classes correspond to real world concepts: Order, Customer, ...
- Many classes are true objects having both:
  - State – fields
  - Behavior – methods that act on the state
Procedural versus OO

Presentation Tier

Business Tier

Transaction Scripts (Session Beans)

Data Objects

Data Access Tier

Presentation Tier

Business Tier

Facade

Domain Model

Data Access Tier

Behavior

State
An example domain model

Web Tier

Business Tier

Behavior

MoneyTransferService
BankingTransaction transfer(fromId, toId, amount)

BankingTransaction Repository
addTransaction(…)

Banking Transaction
amount date

Account
balance
debit(amount)
credit(amount)

Account Repository
findAccount(id)

<<interface>>
OverdraftPolicy

NoOverdraft Policy

Limited Overdraft
limit

State + Behavior

Explicit Representation of key concepts

Chris Richardson: Improving Your Application’s Design by Using Real Objects
DEMO

Code Walkthrough
Benefits of the Domain Model Pattern

- Improved maintainability
  - The design reflects reality
  - Key domain classes are represented by classes
  - The design is more modular

- Improved testability
  - Small classes that can be tested in isolation

- Improved reusability
  - Classes can be used in other applications

- Building a domain model
  - Creates shared understanding
  - Develops an ubiquitous language
Quantifiably simpler code

**Procedural** – few, longer, more complex methods

**Object-oriented** – more, simpler, shorter methods

<table>
<thead>
<tr>
<th>Metric</th>
<th>Total Lines of Code</th>
<th>Method Lines of Code (avg/max per method)</th>
<th>McCabe Cyclomatic Complexity (avg/max per method)</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
</tbody>
</table>
Drawbacks of the Domain Model pattern

- Requires object-oriented design skills
- Requires domain model to be transparently “mappable” to the data
  - e.g. nice database schema
  - Ugly schemas and data stored in other applications is a challenge
When to use it

- The business logic is reasonably complex or you anticipate that it will be
- You have the skills to design one
- You can use an ORM framework
Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- **Domain model building blocks**
- Role of frameworks
- Common code smells
- Refactoring existing code
- Obstacles to OO
Domain model building blocks

- Roles *aka* stereotypes
- Benefits of roles:
  - Guide design
  - Help name objects
  - Aid understanding
- Roles (from Domain-Driven Design)
Entity

- Objects with a distinct identity
- Typically correspond to real world concepts
- Almost always persistent

```java
public class Account {
    private int id;
    private double balance;
    private OverdraftPolicy overdraftPolicy;
    private String accountId;
    private CalendarDate dateOpened;
    Account() {
    }

    public void debit(double amount) throws MoneyTransferException {
        assert amount > 0;
        double originalBalance = balance;
        double newBalance = balance - amount;
        overdraftPolicy.beforeDebitCheck(this, originalBalance, newBalance);
        balance = newBalance;
        overdraftPolicy.afterDebitAction(this, originalBalance, newBalance);
    }

    public void credit(double amount) {
        assert amount > 0;
        balance += amount;
    }
}
```
Value Objects

- Objects that are defined by the values of their attributes
- Two instances with identical values can be used interchangeably
- Part of an entity
- Usually persistent
- Ideally immutable
- Often missing from procedural code

```java
public class CalendarDate {
    private Date date;

    CalendarDate() {
    }

    public CalendarDate(Date date) {
        this.date = date;
    }

    public Date getDate() {
        return date;
    }

    public double getYearsOpen() {
        Calendar then = Calendar.getInstance();
        then.setTime(date);
        Calendar now = Calendar.getInstance();

        int yearsOpened = now.get(Calendar.YEAR) -
                         then.get(Calendar.YEAR);
        int monthsOpened = now.get(Calendar.MONTH) -
                           then.get(Calendar.MONTH);

        if (monthsOpened < 0) {
            yearsOpened--;
            monthsOpened += 12;
        }
        return yearsOpened + (monthsOpened/12.0);
    }
}
```
Aggregates

- A cluster of related entities and values
- Behaves as a unit
- Has a root
- Has a boundary
- Objects outside the aggregate can only reference the root
- Deleting the root removes everything
Repositories

- Manages a collection of objects
- Provides methods for:
  - Adding an object
  - Finding object or objects
  - Deleting objects
- Consists of an interface and an implementation class
- Encapsulates database access mechanism
- Keeps the ORM framework out of the domain model
- Similar to a DAO

```java
public interface AccountRepository {
    Account findAccount(String accountId);
    void addAccount(Account account);
}
```

```java
public class HibernateAccountRepository implements AccountRepository {
    private HibernateTemplate hibernateTemplate;

    public HibernateAccountRepository(HibernateTemplate template) {
        hibernateTemplate = template;
    }

    public void addAccount(Account account) {
        hibernateTemplate.save(account);
    }

    public Account findAccount(final String accountId) {
        return (Account) DataAccessUtils.uniqueResult(hibernateTemplate
            .findByNamedQueryAndNamedParam("Account.findAccountByAccountId", "accountId",
            accountId));
    }
}
```
Services

- Implements logic that cannot be put in a single entity
- Not persistent
- Consists of an interface and an implementation class
  - Service method usually:
    - Invoked (indirectly) by presentation tier
    - Invokes one or more repositories
    - Invokes one or more entities
  - **Keep them thin**

```java
public interface MoneyTransferService {
    BankingTransaction transfer(String fromAccountId, String toAccountId, double amount)
        throws MoneyTransferException;
}

public class MoneyTransferServiceImpl implements MoneyTransferService {
    private final AccountRepository accountRepository;
    private final BankingTransactionRepository bankingTransactionRepository;
    public MoneyTransferServiceImpl(AccountRepository accountRepository, BankingTransactionRepository bankingTransactionRepository) {
        this.accountRepository = accountRepository;
        this.bankingTransactionRepository = bankingTransactionRepository;
    }

    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
        ...
    }
}
```
Factories

- Use when a constructor is insufficient
  - Encapsulates complex object creation logic
  - Handles varying products
- Different kinds of factories
  - Factory classes
  - Factory methods
- Example: OrderFactory
  - Creates Order from a shopping cart
  - Adds line items
Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- Domain model building blocks
- **Role of frameworks**
- Common code smells
- Refactoring existing code
- Obstacles to OO
Use the POJO programming model

- Your domain model might outlive infrastructure frameworks ⇒ Minimize dependencies on them
- POJO = Plain Old Java Object
- Don't implement any infrastructure interfaces
- Don't call infrastructure APIs
- No infrastructure framework annotations?
Use dependency injection

- Spring instantiates and wires together
  - Services, factories and repositories
- Dependency injection into entities
  - One option is @Configurable but it’s not POJO
  - Hibernate Interceptor+Manual injection is preferable

Benefits:
- Decouples components from one another and the infrastructure
- Improves testability

```java
public AccountServiceImpl(AccountDao accountDao,
BankingTransactionDao bankingTransactionDao) {
this.accountDAO = accountDao;
this.bankingTransactionDAO = bankingTransactionDao;
}
```

```xml
<beans>
  <bean id="AccountService"
    class="net.chris...domain.AccountServiceImpl">
    <constructor-arg ref="accountDao"/>
    <constructor-arg ref="bankingTransactionDao"/>
  </bean>
  ...
</beans>
```
Use Aspect-Oriented Programming

- Spring AOP for service-level crosscutting concerns:
  - e.g. transaction management, security, logging, etc.
- AspectJ for entity and value object crosscutting concerns
  - e.g. tracking changes to fields
  - But AJC/Load-time weaving has a cost
- Benefits
  - Decouples code from infrastructure
  - Improves modularity
Use object/relational mapping

- Persisting objects with JDBC is too much work
- Implement DAOs with Spring ORM
- Benefits
  - Less code
  - Simpler code
  - Improved testability

```
<class name="Account"
  table="BANK_ACCOUNT">
  <id name="id" column="ACCOUNT_ID">
    <generator class="native"/>
  </id>
  <property name="balance"/>
  <property name="accountId"/>
  <property name="dateOpened"/>
  <many-to-one name="overdraftPolicy"/>
</class>
```

```
public class HibernateAccountDao
  implements AccountDao {
  private HibernateTemplate hibernateTemplate;

  public HibernateAccountDao(HibernateTemplate template) {
    this.hibernateTemplate = template;
  }

  public void addAccount(Account account) {
    hibernateTemplate.save(account);
  }
  ...
}
```
Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- Domain model building blocks
- Role of frameworks
- **Common code smells**
- Refactoring existing code
- Obstacles to OO
Overview of code smells

- Code smell = something about the code that does not seem right
- Impacts ease of development and testing
- Some are non-OOD
- Some are the consequences of non-OOD
Long method

- Methods should be short
- But business logic is concentrated in the services ⇒ long methods
- Long methods are difficult to:
  - Read and understand
  - Maintain
  - Test
- Fix:
  - Splitting into smaller methods

```java
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {
    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) throws MoneyTransferException {
        Account fromAccount = accountDAO.findAccount(fromAccountId);
        Account toAccount = accountDAO.findAccount(toAccountId);
        assert amount > 0;
        double newBalance = fromAccount.getBalance() - amount;
        switch (fromAccount.getOverdraftPolicy()) {
            case Account.NEVER:
                if (newBalance < 0)
                    throw new MoneyTransferException("Insufficient funds");
                break;
            case Account.ALLOWED:
                Calendar then = Calendar.getInstance();
                then.setTime(fromAccount.getDateOpened());
                Calendar now = Calendar.getInstance();
                double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
                int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);
                if (monthsOpened < 0) {
                    yearsOpened--;
                    monthsOpened += 12;
                }
                yearsOpened = yearsOpened + (monthsOpened / 12.0);
                if (yearsOpened < fromAccount.getRequiredYearsOpen() || newBalance < fromAccount.getLimit())
                    throw new MoneyTransferException("Limit exceeded");
                break;
            default:
                throw new MoneyTransferException("Unknown overdraft type: " + fromAccount.getOverdraftPolicy());
        }
        fromAccount.setBalance(newBalance);
        toAccount.setBalance(toAccount.getBalance() + amount);
        TransferTransaction txn = new TransferTransaction(fromAccount, toAccount, amount, new Date());
        bankingTransactionDAO.addTransaction(txn);
        return txn;
    }
```
Feature Envy

- Methods that are far too interested in data belonging to other classes
- Results in:
  - Poor encapsulation
  - Long methods
- Fix by moving methods to the class that has the data

```java
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {
    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) throws MoneyTransferException {
        Account fromAccount = accountDAO.findAccount(fromAccountId);
        Account toAccount = accountDAO.findAccount(toAccountId);
        assert amount > 0;
        double newBalance = fromAccount.getBalance() - amount;
        switch (fromAccount.getOverdraftPolicy()) {
            case Account.NEVER:
                if (newBalance < 0)
                    throw new MoneyTransferException("In sufficient funds");
                break;
            case Account.ALLOWED:
                Calendar then = Calendar.getInstance();
                then.setTime(fromAccount.getDateOpened());
                Calendar now = Calendar.getInstance();
                double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
                int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);
                if (monthsOpened < 0) {
                    yearsOpened--;
                    monthsOpened += 12;
                }
                yearsOpened = yearsOpened + (monthsOpened / 12.0);
                if (yearsOpened < fromAccount.getRequiredYearsOpen() || newBalance < fromAccount.getLimit())
                    throw new MoneyTransferException("Limit exceeded");
                break;
            default:
                throw new MoneyTransferException("Unknown overdraft type: " + fromAccount.getOverdraftPolicy());
        }
        fromAccount.setBalance(newBalance);
        toAccount.setBalance(toAccount.getBalance() + amount);
        TransferTransaction txn = new TransferTransaction(fromAccount, toAccount, amount, new Date());
        bankingTransactionDAO.addTransaction(txn);
        return txn;
    }
}
```
Data class

- Classes that are just getters and setters
- No business logic - it’s in the service
- Leads to: ➢ Feature envy
- Fix by moving methods that act on data into class

```java
public class Account {
    public static final int NEVER = 1;
    public static final int ALLOWED = 2;

    private int id;
    private double balance;
    private int overdraftPolicy;
    private String accountId;
    private Date dateOpened;
    private double requiredYearsOpen;
    private double limit;

    Account() {}  
    public Account(String accountId, double balance, int overdraftPolicy,
                   Date dateOpened, double requiredYearsOpen, double limit)   
                   {..... }  
    public int getId()  {return id;}  
    public String getAccountId() {return accountId;}  
    public void setBalance(double balance) { this.balance = balance; }
    public double getBalance() { return balance; }
    public int getOverdraftPolicy() { return overdraftPolicy; }
    public Date getDateOpened() { return dateOpened; }
    public double getRequiredYearsOpen() { return requiredYearsOpen; }
    public double getLimit() {return limit; }
}
```
## Primitive Obsession

- Code uses built-in types instead of application classes
- **Consequences:**
  - Reduces understandability
  - Long methods
  - Code duplication
  - Added complexity
- Fix by moving data and code into new class

```java
public class Account {
    private Date dateOpened;
}
```

```java
public class Account {
    private Date dateOpened;
}
```

```java
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) throws MoneyTransferException {
        Account fromAccount = accountDAO.findAccount(fromAccountId);
        Account toAccount = accountDAO.findAccount(toAccountId);

        Calendar then = Calendar.getInstance();
        then.setTime(fromAccount.getDateOpened());
        Calendar now = Calendar.getInstance();

        double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
        int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);

        if (monthsOpened < 0) {
            yearsOpened--;
            monthsOpened += 12;
        }
        yearsOpened = yearsOpened + (monthsOpened / 12.0);

        if (yearsOpened < fromAccount.getRequiredYearsOpen() || newBalance < fromAccount.getLimit()) {
            ...}
```
Switch Statements

- Use of type codes and switch statements instead of polymorphism
- Key concepts are not represented by classes
- Consequences:
  - Longer methods
  - Poor maintainability caused by code duplication
  - Increased code complexity
- Fix by introducing class hierarchy and moving each part of switch statement into a overriding method

```java
public class Account {
    public static final int NEVER = 1;
    public static final int ALLOWED = 2;
    ...
}

public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {
    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) throws MoneyTransferException {
        ...
        switch (fromAccount.getOverdraftPolicy()) {
            case Account.NEVER:
                ...
                break;
            case Account.ALLOWED:
                ...
                default:
                    ...
        }
    }
```
Data clumps

- Multiple fields or method parameters that belong together
- Consequences:
  - Long methods
  - Duplication
- Fix by:
  - Moving fields into their own class
  - Eliminate resulting Feature Envy

```java
public class Account {
    public static final int NEVER = 1;
    public static final int ALLOWED = 2;

    private int id;
    private double balance;
    private String accountId;
    private Date dateOpened;
    private int overdraftPolicy;
    private double requiredYearsOpen;
    private double limit;

    Account() {}
}
```
Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- Domain model building blocks
- Role of frameworks
- Common code smells
- **Refactoring existing code**
- Obstacles to OO
Transforming procedural code

- Inside every procedural design is a domain model just trying to get out
- Incrementally transform a procedural design into an OO design
  - Small, localized changes
  - Something to do on Monday morning!
Refactoring to an OO design

- Transform a procedural design to an OO design by applying refactorings
- Refactoring:
  - Restructure the code
  - Without changing behavior
- Essential cleanups for decaying code
Basic refactorings

- Extract Method
  - Eliminates long methods
- Move Method
  - Move a method to a different class (field or parameter)
  - Moves method to where the data is
- Push Down
  - Move a method into subclasses
  - Optionally leave an abstract method behind
  - Part of eliminating conditional logic
- ...
Compound refactorings

- A sequence of simpler refactorings
- Compose method
  - Apply Extract Method repeatedly
  - Use to replace long method with more readable shorter methods
- Replace Type Code With Strategy
  - Define GOF Strategy class for each type code
- Replace Conditional With Polymorphism
  - Turn into part of a switch statement into an overriding method in a subclass
- Replace Data Value with Object
  - Move field into its own class
  - Eliminates Primitive Obsession
DEMO

Refactoring procedural code
Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- Domain model building blocks
- Role of frameworks
- Common code smells
- Refactoring existing code
- **Obstacles to good OO design**
Performance tuning changes the design

- Different web requests display different object graphs
  - Display project details
  - Display project for approval or rejection
- Improve performance by loading object graphs using optimized queries (e.g. fetch join)
- Therefore we need multiple Dao (and Service) methods
Untangling what to load from the code

- Java Data Objects (JDO) has fetch groups
  - Declarative specification the object graph that needs to be loaded

- Flow:
  - Web tier configures fetch groups
  - Calls ProjectCoordinator.get()
  - O/RM loads object graph specified by active fetch groups

- Look for this feature in JPA implementations that evolved from JDO
Web frameworks need JavaBeans

- Web frameworks can bind request parameters directly to domain objects
- Easy creation of domain objects
- Easy editing of detached domain objects
  1. Load object graph from database
  2. Store object graph in HttpSession
  3. Bind Http parameters to object's properties
  4. Reattach object graph and update the database
- A lot less code
- But domain objects must be JavaBeans:
  - Public default constructor
  - JavaBean-style setters
The trouble with setters

- Poor encapsulation
  - Objects are now exposing their internal state
- No immutable objects
  - Increases code complexity
  - Makes it more error-prone
- Developers might bypass business method ⇒ Bugs, poor design

```java
public class Address {
    private String street1;
    private String street2;
    public String getStreet1() {...};
    public void setStreet1(String street1) {...}
}

public class Order {
    private Address deliveryAddress;
    public void updateDeliveryAddress(Address deliveryAddress) {
        // validate before changing
        ...
    }
    public Address getAddress() {...}
    ...
}

order.getAddress().setStreet1("Bad!");
```
How to cope

- Live with it:
  - It's often not that bad
  - Be vigilant for code bypassing business methods
- Use DTOs
  - Hides domain objects from web tier
  - But you have to write more code
- Encourage the development of better frameworks
More on web binding

- Web frameworks can bind to properties of nested objects, e.g.
  order.deliveryAddress.street1
- But some frameworks require that intermediate objects to be non-null
- This can impact the domain model
Using ORM frameworks

- ORM frameworks can access private fields
  - No setters (or getters) required 🤓
- But require:
  - Default constructor
  - Non-final fields
  - id fields
- Subtle changes:
  - Collection fields: null ⇒ empty
  - Embedded objects with null fields ⇒ null reference
- Proxy-based lazy loading:
  - Don't access fields outside of instance ⇒ affects equals() and hashCode()
  - Can affect object identity: this != proxyToSameObjectAsThis
Summary

A rich domain model:
- Organizes the business logic as classes with state AND behavior
- Improves maintainability and testability
- Enabled by POJOs and non-invasive frameworks (mostly)
- Emerges from procedural code by incremental refactoring

Use It!
For more information

- Buy my book 😊
- Send email:
  chris@chrisrichardson.net
- Visit my website:
  http://www.chrisrichardson.net
- Talk to me about consulting and training