Improving Application Design with a Rich Domain Model

Chris Richardson

Author of POJOs in Action

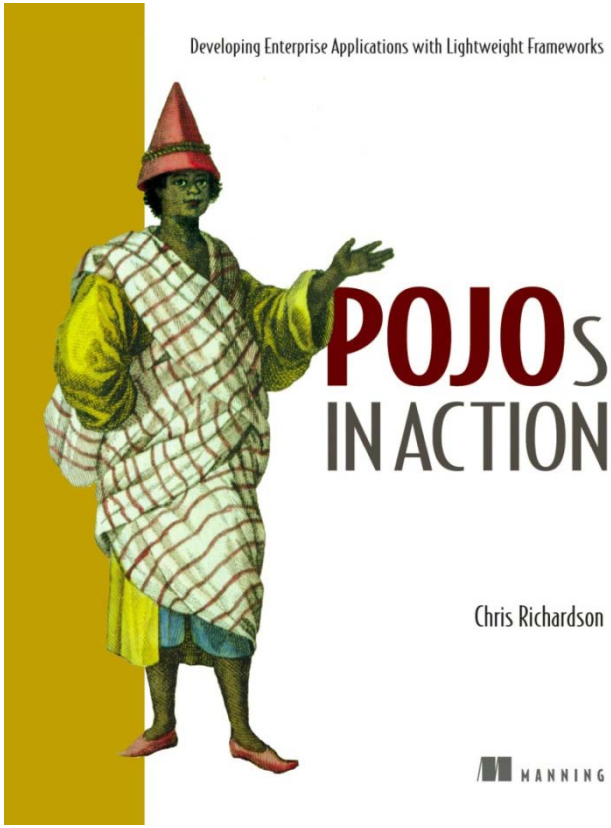
Chris Richardson Consulting, Inc

http://www.chrisrichardson.net

Overall presentation goal

Learn how to improve application design by using truly object-oriented business logic

About Chris



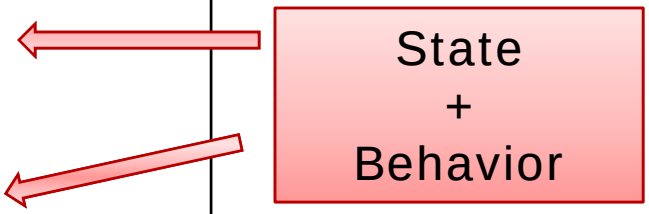
- Grew up in England
- Live in Oakland, CA
- Over twenty years of software development experience
  - Building object-oriented software since 1986
  - Using Java since 1996
  - Using J2EE since 1999
- Author of POJOs in Action
- Speaker at JavaOne, JavaPolis, NFJS, JUGs, ….
- Chair of the eBIG Java SIG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

Agenda

- **Where are the real objects?**
- Overview of the Domain Model pattern
- Domain model building blocks
- Role of frameworks
- Eliminating common code smells

Objects in LISP (1987-1993)

```
(defclass Account ()
  ((account-id :accessor account-id :initarg :account-id)
   (balance :accessor account-balance :initarg :balance))
)

(defmethod debit ((Account account) amount)
  (decf (account-balance account) amount))

(defmethod credit ((Account account) amount)
  (incf (account-balance account) amount))
```




```
CL-USER 5 > (setq a (make-instance 'account :account-id "abc123" :balance 10.0))
#<ACCOUNT 200C05AF>

CL-USER 6 > (describe a)

#<ACCOUNT 200C05AF> is an ACCOUNT
ACCOUNT-ID      "abc123"
BALANCE         10.0

CL-USER 7 > (debit a 5)
5.0

CL-USER 8 > (describe a)

#<ACCOUNT 200C05AF> is an ACCOUNT
ACCOUNT-ID      "abc123"
BALANCE         5.0
```

Objects in C++ (1993-1996)

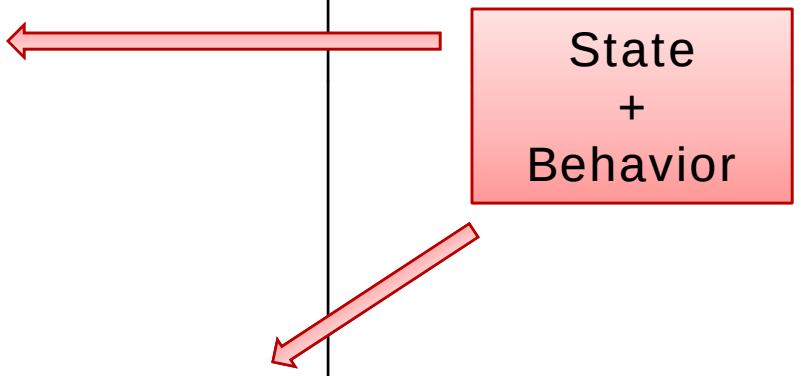
```
#ifndef ACCOUNT_H_
#define ACCOUNT_H_

class Account {
public:
        Account(char* account_id, double balance);
        void debit(double amount);
        void credit(double amount);
        double getBalance();
private:
        char* account_id;
        double balance;
};
#endif /*ACCOUNT_H_*/
```

State + Behavior

```
#include "Account.h"

Account::Account(char* account_id, double balance)  {
   ...
}
void Account::debit (double amount) {
        balance -= amount;
}

void Account::credit(double amount) {
        balance += amount;
}

double Account::getBalance() { return balance; }
```

Objects in Java (1996-1999)

```
public class Account {

  private int id;

  private double balance;

  private OverdraftPolicy overdraftPolicy;

  private String accountId;

  private CalendarDate dateOpened;

  Account() {
  }

  public void debit(double amount)  {
    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;
  }
```

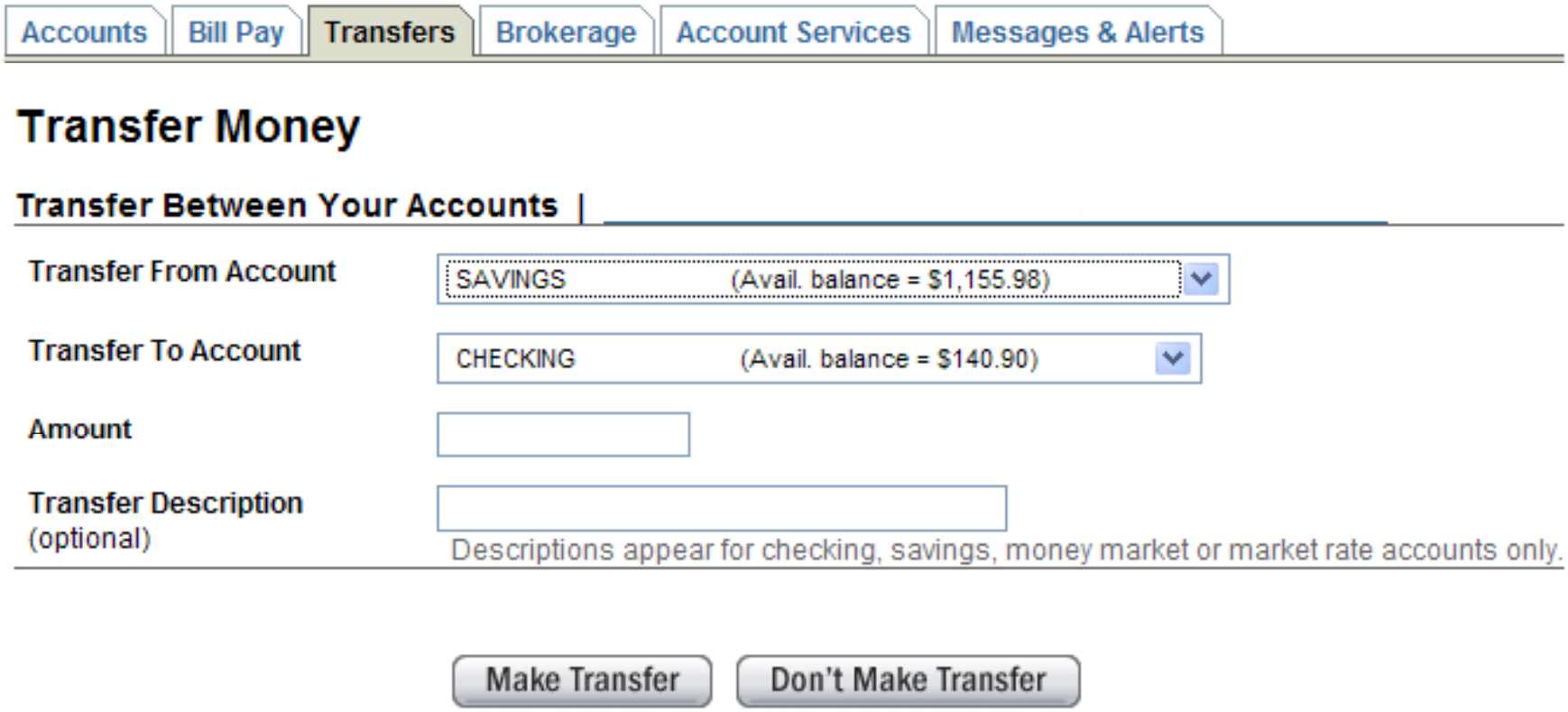


EJB objects (1999- ?)

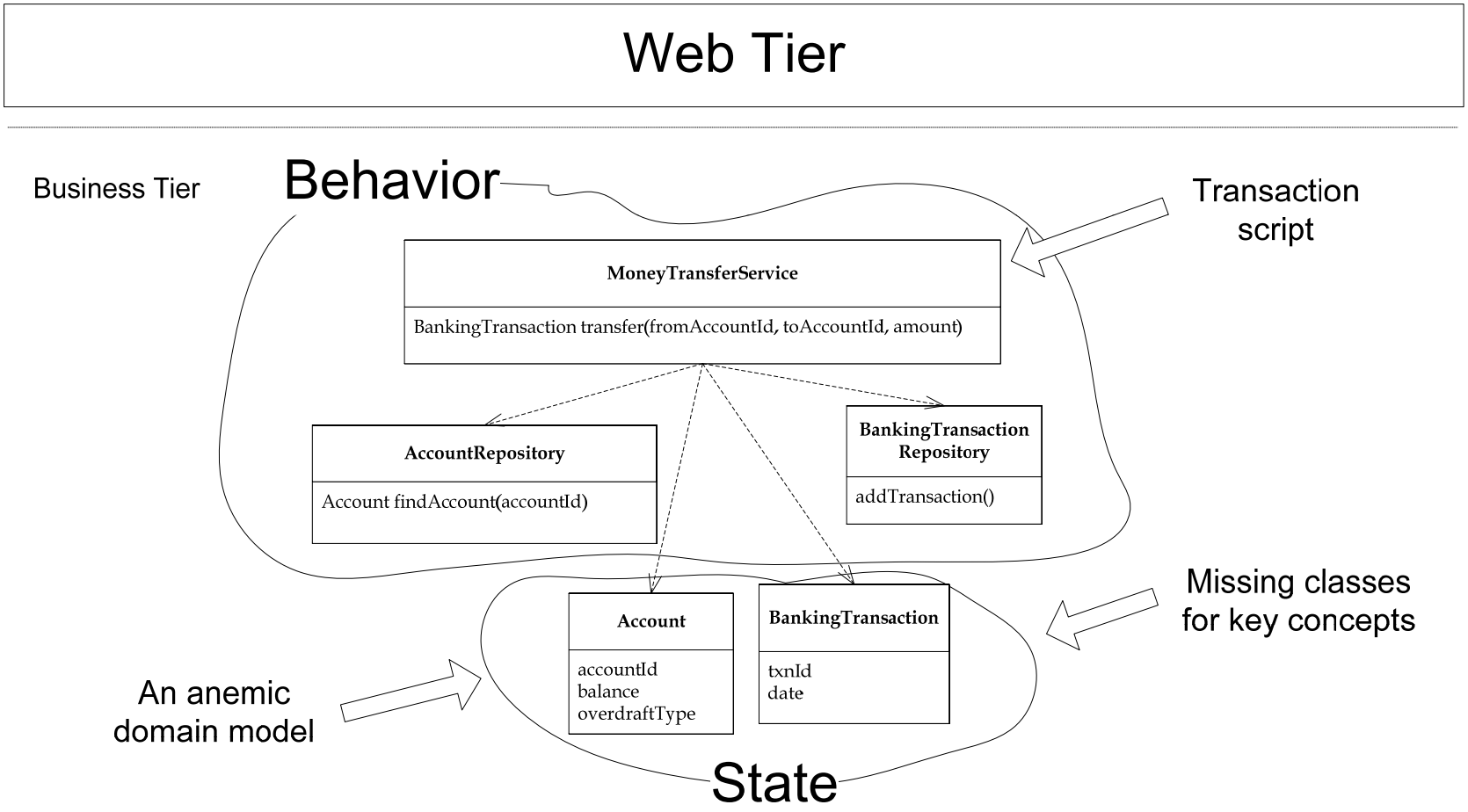
Applications were still built from objects

But those objects were very different …

Example Banking UI



Example procedural design



Example procedural code

```
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

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

Behavior

```
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;  }
}
```

State

Objects in Java EE (2008)

Java is an object-oriented language

AND

We have known for many years that OOD is a better way to tackle complexity

YET

Many complex enterprise Java applications are written in a procedural style

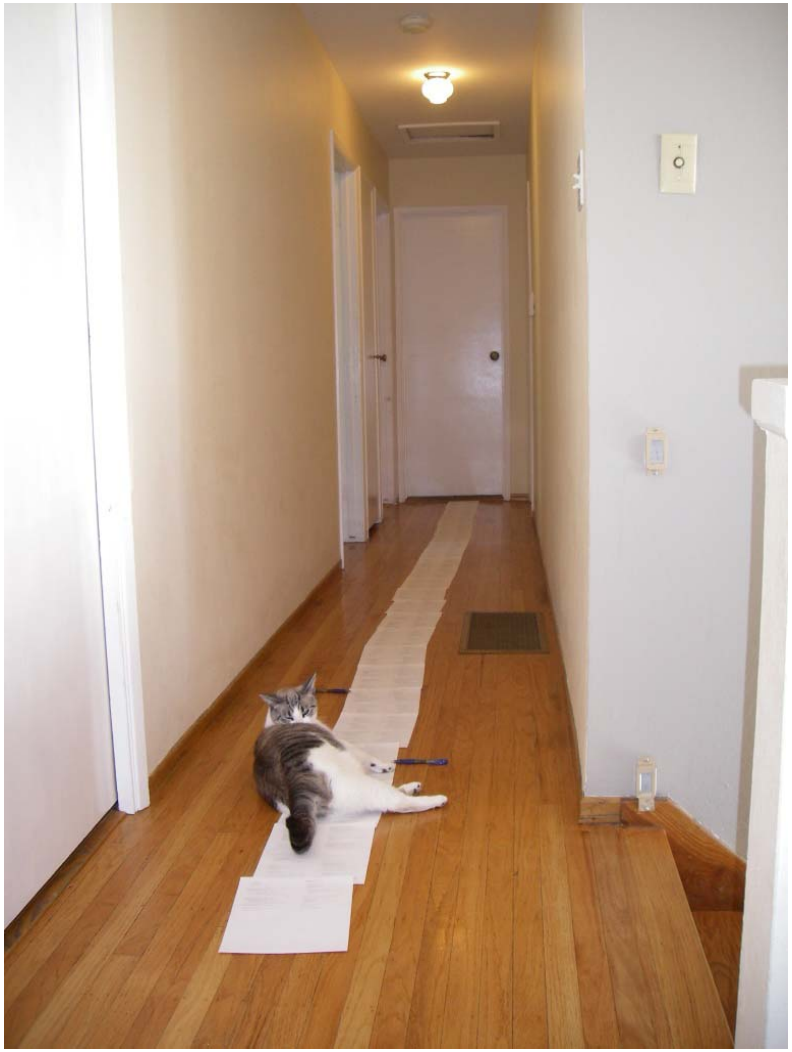
Why use leeches instead of penicillin?

- Legacy of EJB, which made writing object-oriented code difficult/impossible
- It is easy to implement new functionality
  - Add a new transaction script
  - Add code to a new transaction script
- Manipulating relational data is easier
- Distribution is easier
- No need to do any real design, e.g.
  - Create new classes
  - Determine responsibilities

So what? My code works!

- Procedural design 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

Any volunteers?



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
  - ...

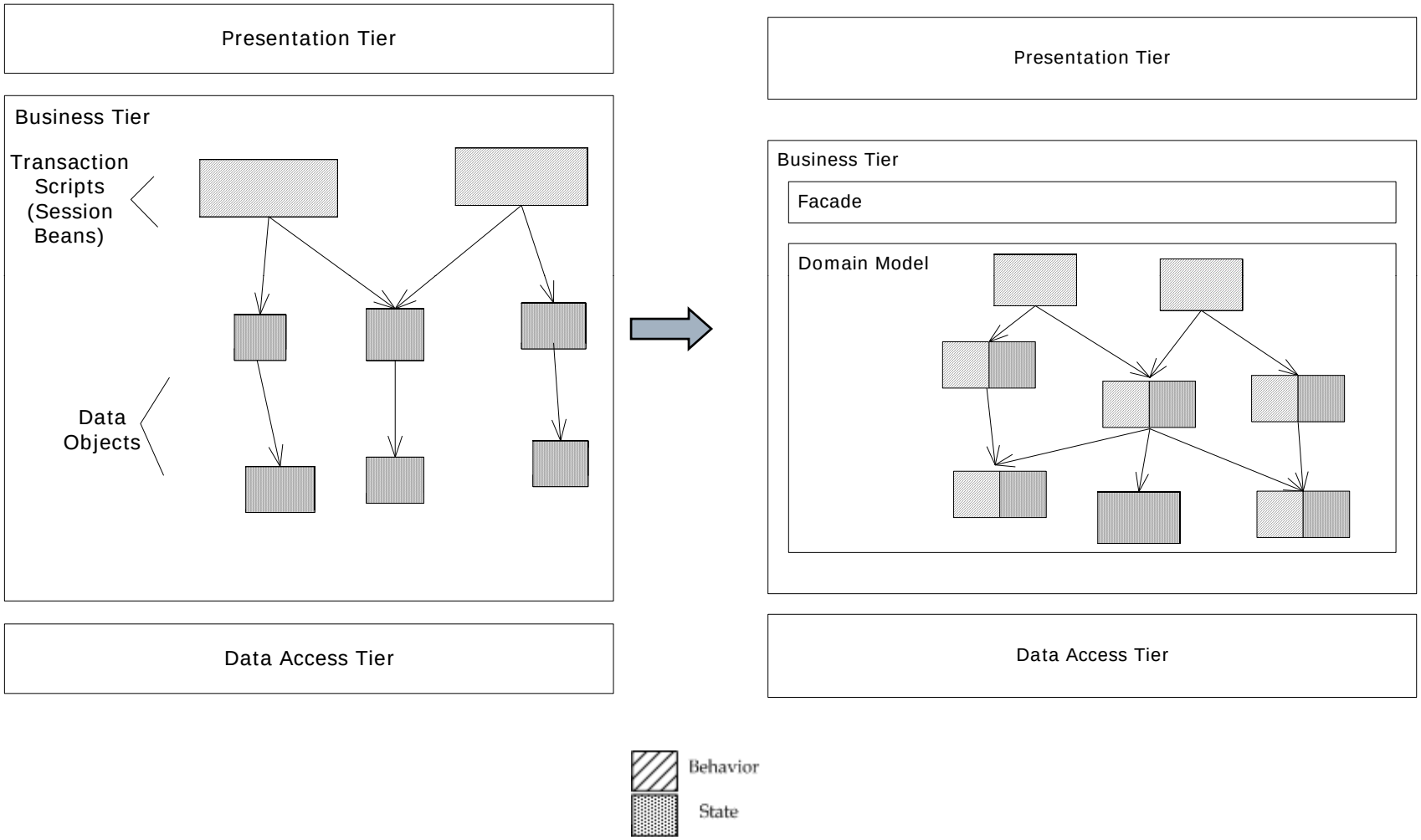
Agenda

- Where are the real objects?
- **Overview of the Domain Model pattern**
- Domain model building blocks
- Role of frameworks
- Eliminating common code smells

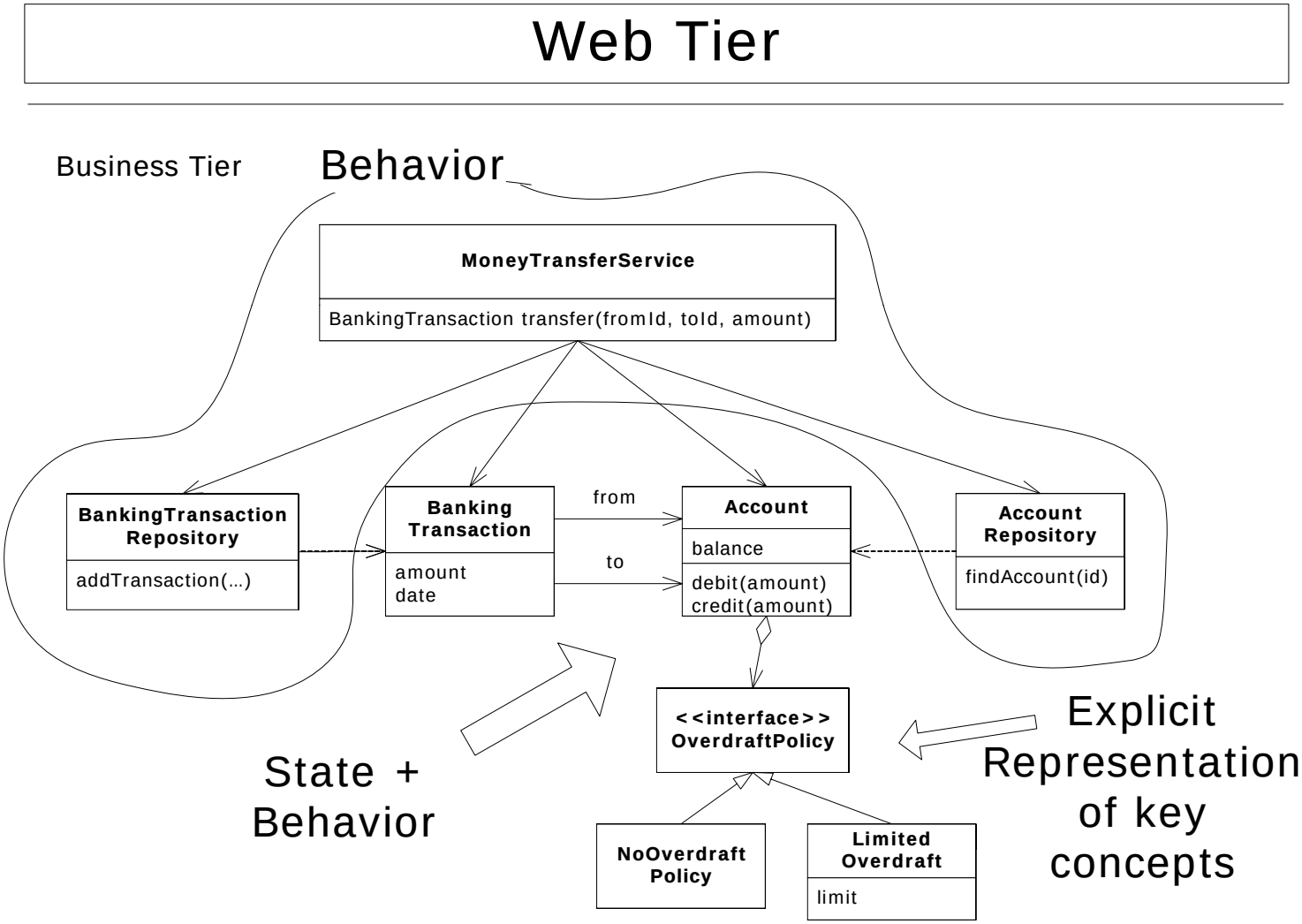
Using the Domain Model Pattern

- 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



An example domain model



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

Metric	Total	Mean	Std. D...	Maximum
Total Lines of Code	284			
Method Lines of Code (avg/max per method)	130	3.94	6.4	34
net.chrisrichardson.bankingExample.domain	116	4.14	6.8	34
net.chrisrichardson.bankingExample.domain.hibernate	14	2.8	3.12	9
McCabe Cyclomatic Complexity (avg/max per method)		1.33	1.15	7
net.chrisrichardson.bankingExample.domain		1.39	1.23	7
MoneyTransferServiceProceduralImpl.java		4	3	7
ExampleOfTransactionScriptSprawl.java		1.71	1.03	4
Account.java		1	0	1
TransferTransaction.java		1	0	1
MoneyTransferException.java		1	0	1

Object-oriented – more, simpler, shorter methods

Metric	Total	Mean	Std. D...	Maximum
Total Lines of Code	239			
Method Lines of Code (avg/max per method)	66	1.83	2.22	10
net.chrisrichardson.bankingExample.domain	57	1.9	2.37	10
net.chrisrichardson.bankingExample.domain.hibernate	9	1.5	1.12	4
McCabe Cyclomatic Complexity (avg/max per method)		1.14	0.54	4
net.chrisrichardson.bankingExample.domain		1.17	0.58	4
LimitedOverdraft.java		1.75	1.3	4
NoOverdraftAllowed.java		1.5	0.5	2
CalendarDate.java		1.25	0.43	2
TransferTransaction.java		1	0	1
MoneyTransferException.java		1	0	1
Account.java		1	0	1
MoneyTransferServiceImpl.java		1	0	1

Drawbacks of the Domain Model pattern

- Requires object-oriented design skills
- Works best if domain model is transparently “mappable” to the data
  - E.g. nice database schema
  - Ugly schemas and data stored in other applications is a challenge

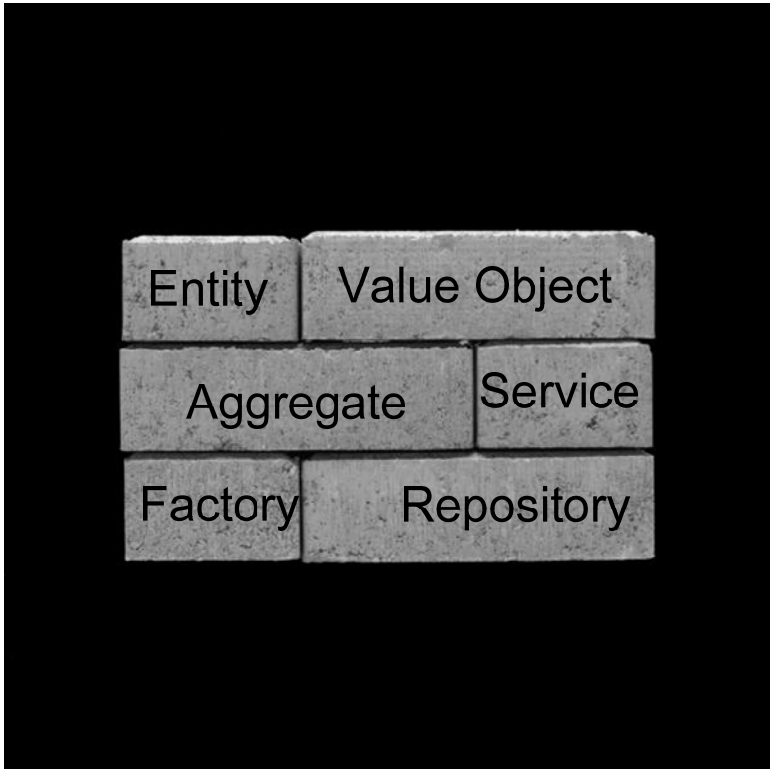
When to use it

- The business logic is (or will be) complex
- You have the skills to design one
- You can either:
  - Use an ORM framework
  - Invest in writing a data access framework

Agenda

- Where are the real objects?
- Overview of the Domain Model pattern
- **Domain model building blocks**
- Role of frameworks
- Eliminating common code smells

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
- Encapsulate state and behavior
- Often modal ⇒ call methods in a particular order

```
public class Account {

  private int id;

  private double balance;

  private OverdraftPolicy overdraftPolicy;

  private String accountId;

  private CalendarDate dateOpened;

  Account() {
  }

  public void debit(double amount)  {
    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
- Two flavors
  - Persistent – parts of entities
  - Transient – intermediate values
- Ideally immutable
- **Often missing from procedural code – Primitive Obsession code smell**

```
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);

  }

}
```

More examples of Value Objects

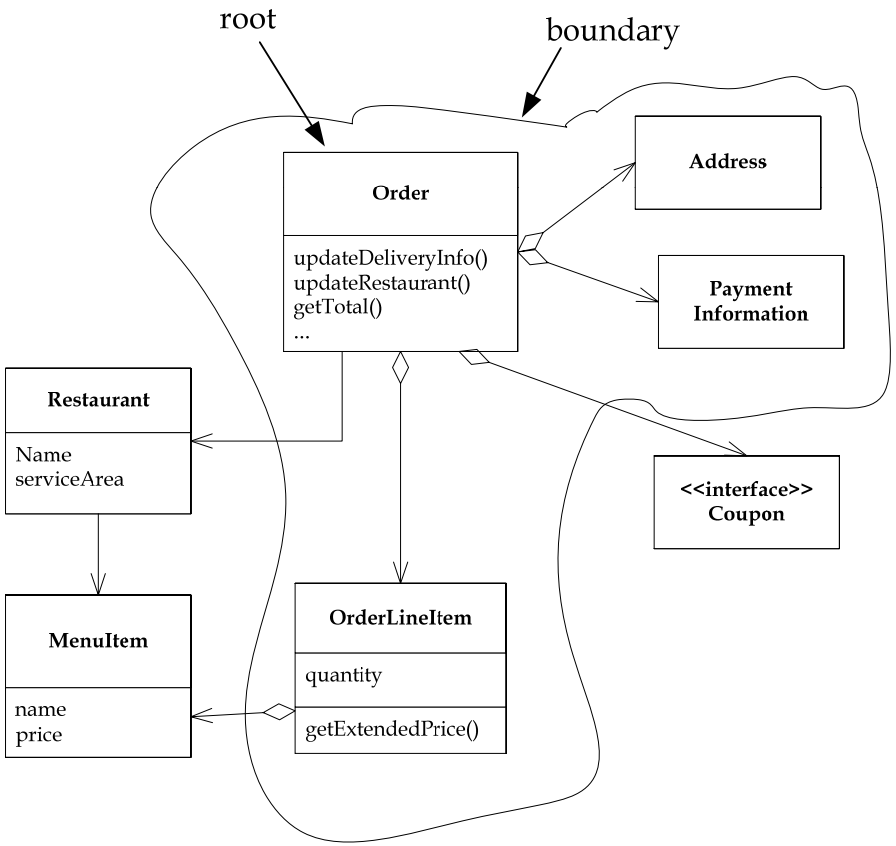
```
public class User {
  private String firstName;
  private String lastName;
  private String login;
  private String password;
  ...

}
```

```
public class User {
  private int id;
  private PersonName name;
  private UserId login;
  private Password password;
  ...
}
```

```
public class PersonName {
  private String firstName;
  private String lastName;

  PersonName() {
  }

  public PersonName(String firstName,
           String lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
```

```
public class Password implements Serializable {

 private String passwordString;

public Password(String passwordString) {
    this.passwordString = passwordString == null ? null : passwordString.trim();
  }

  ...
 @Override
  public String toString() {
    return new ToStringBuilder(this).append("password", "*****").toString();
  }
```

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

```
public interface AccountRepository {

   Account findAccount(String accountId);

   void addAccount(Account account);

}
```

```
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**

```
public interface MoneyTransferService {

    BankingTransaction transfer(String fromAccountId,
          String toAccountId, double amount);

}
```

```
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**
- Eliminating common code smells

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

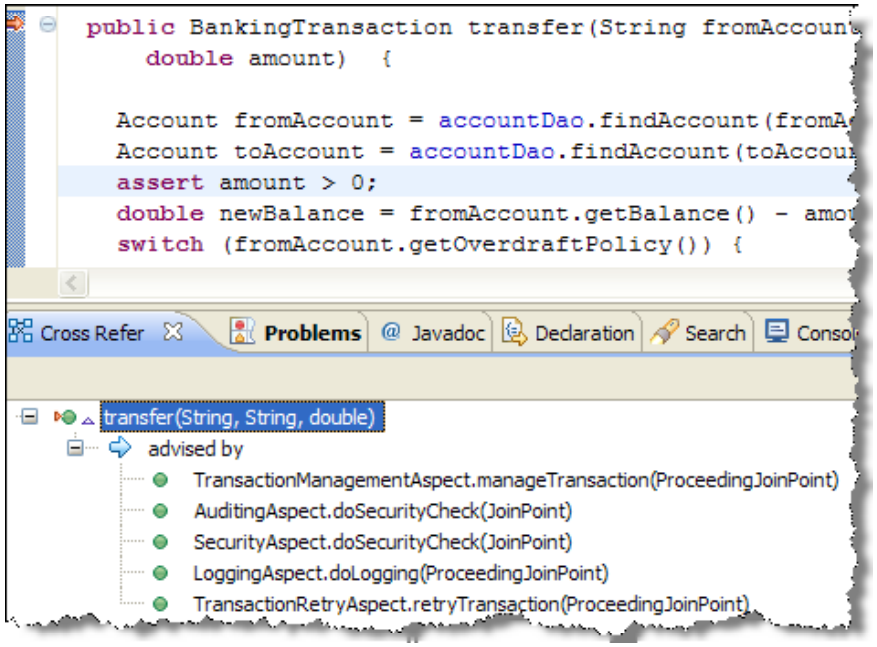
- Spring instantiates and wires together components:
  - Services, factories and repositories
- Dependency injection into entities
  - One option is @Configurable but it's not POJO
  - Use Hibernate Interceptor + manual injection instead?
- Benefits:
  - Decouples components from one another and the infrastructure
  - Improves testability

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

```
<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 usually 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

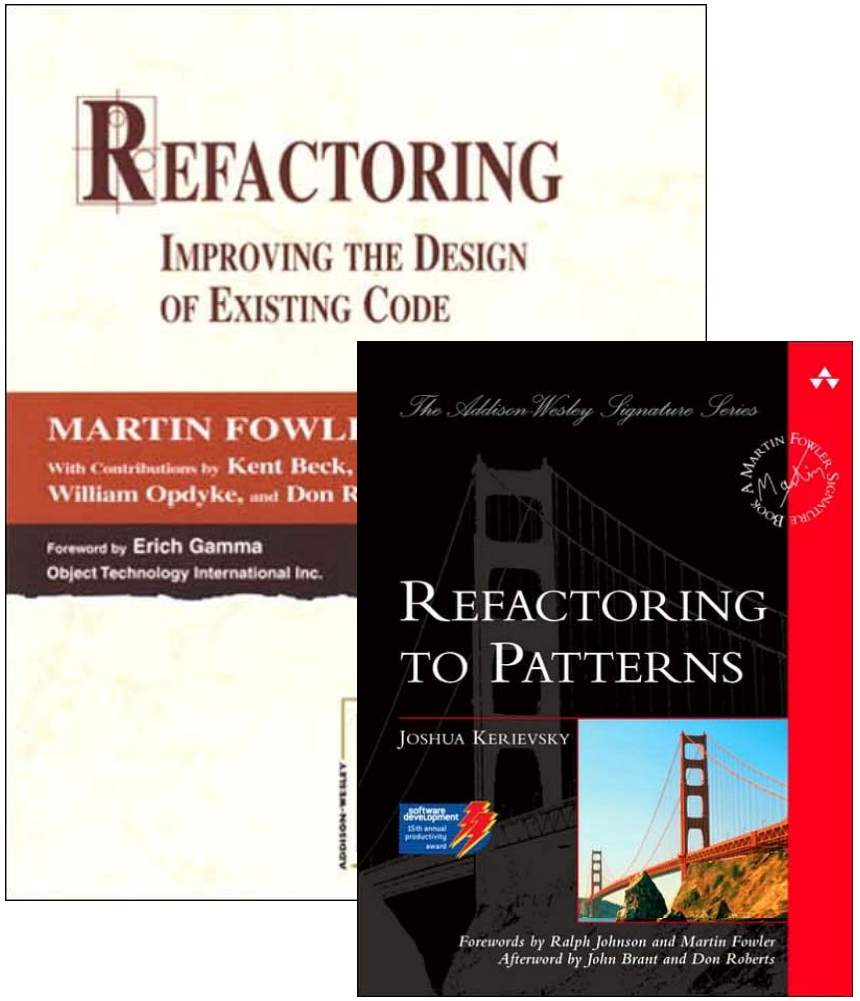
- Where are the real objects?
- Overview of the Domain Model pattern
- Domain model building blocks
- Role of frameworks
- **Eliminating common code smells**

Overview of code smells

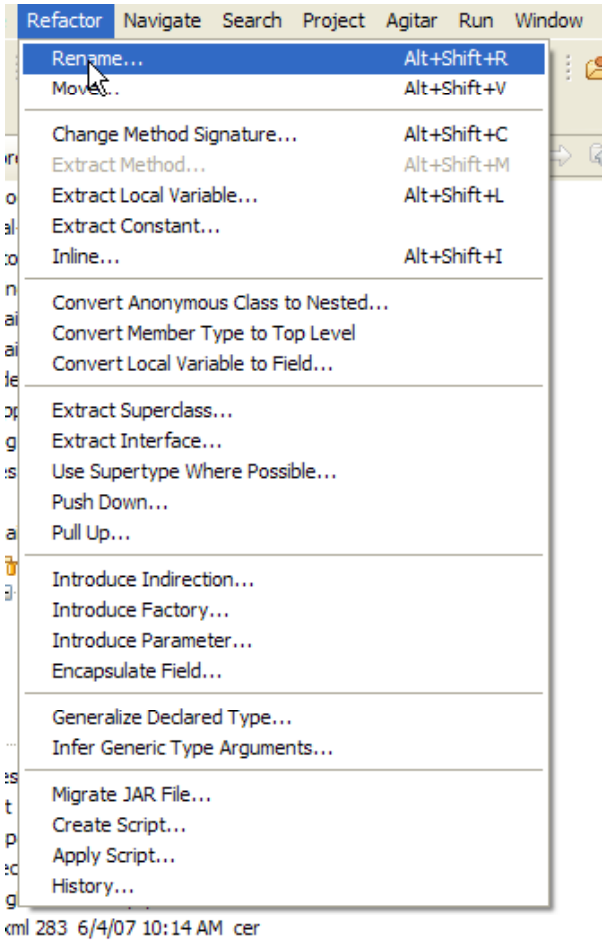
- Code smell = something about the code that does not seem right
- Impacts ease of development and testing
- Some are non-OOD
- Some are the consequences of non-OOD

Refactoring – the cure for stinky code

- Refactoring:
  - Systematic way to 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 it's own class
  - Eliminates Primitive Obsession

Long method

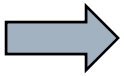
- Methods should be short
- But business logic is concentrated in the services ⇒ long methods
- Long methods are difficult to:
  - Read and understand
  - Maintain
  - Test
- Fix:
  - Splitting into smaller methods

```
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

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

Compose Method Refactoring

```
public class MoneyTransferServiceProceduralImpl
implements MoneyTransferService {

public BankingTransaction transfer(String fromAccountId,
   String toAccountId, double amount)  {
    Account fromAccount =
              accountDAO.findAccount(fromAccountId);
    Account toAccount =
               accountDAO.findAccount(toAccountId);
    double newBalance = fromAccount.getBalance() –
                                amount;
    switch (fromAccount.getOverdraftPolicy()) {
    case Account.NEVER:
       ...
      break;
    default:
       ...
   }
    fromAccount.setBalance(newBalance);
    toAccount.setBalance(toAccount.getBalance() +
                               amount);
   TransferTransaction txn = new
          TransferTransaction(fromAccount, toAccount,
                                amount, new Date());
  bankingTransactionDAO.addTransaction(txn);
   return txn;
}
```

Extract Method



```
public class MoneyTransferServiceProceduralImpl implements
MoneyTransferService {

public BankingTransaction transfer(String fromAccountId, String
          toAccountId,  double amount)  {
   Account fromAccount =
            accountDAO.findAccount(fromAccountId);
   Account toAccount = accountDAO.findAccount(toAccountId);
   assert amount > 0;
   debit(fromAccount, amount);
   credit(toAccount, amount);
   TransferTransaction txn = new
                TransferTransaction(fromAccount, toAccount,
      amount, new Date());
   bankingTransactionDAO.addTransaction(txn);
   return txn;
  }

  public void debit(Account fromAccount, double amount) {
   double newBalance = fromAccount.getBalance() –
                              amount;
   switch (fromAccount.getOverdraftPolicy()) {
   case Account.NEVER:
    ...
     break;
   default:
      ...
    }
    fromAccount.setBalance(newBalance);
}

public void credit(Account toAccount, double amount) {
    toAccount.setBalance(toAccount.getBalance() +
                               amount);
}
```

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

```
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

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

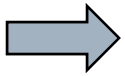
Data class

- 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

```
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;  }
}
```

Move Method refactoring

```
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

public BankingTransaction transfer(String fromAccountId, String toAccountId, double
amount)  {
   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 MoneyTransferServiceProceduralImpl implements
          MoneyTransferService {

public BankingTransaction transfer(String fromAccountId, String
          toAccountId,  double amount)  {
   Account fromAccount =
            accountDAO.findAccount(fromAccountId);
   Account toAccount = accountDAO.findAccount(toAccountId);
   assert amount > 0;
   fromAccount.debit(amount);
   toAccount.credit(amount);
   TransferTransaction txn = new
                TransferTransaction(fromAccount, toAccount,
      amount, new Date());
   bankingTransactionDAO.addTransaction(txn);
   return txn;
  }
```

```
public class Account {
  public void debit(fAccount fromAccount, double amount) {
    double newBalance = getBalance() – amount;
    switch (getOverdraftPolicy()) {
     ....
    }
   setBalance(newBalance);
}

public void credit(Account toAccount, double amount) {
    setBalance(getBalance() +  amount);
}
```

Extract and move feature envy code into data class

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

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

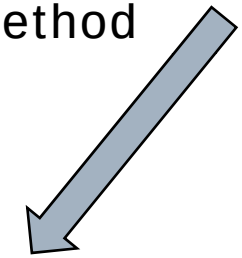
```
public class Account {
  private Date dateOpened;
}

public class MoneyTransferServiceProceduralImpl
          implements MoneyTransferService {

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

Replace Data Value with Object

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

```
public class CalendateDate {
  private Date dateOpened;
  Date getDateOpened( return dateOpened; }
}
```

Move Method



```
public class Account {
  private CalendateDate dateOpened;

  public double getYearsOpen() {
    return dateOpened.getYearsOpened();
  }
```

```
public class CalendateDate {
  private Date dateOpened;

  public double getYearsOpen() {

    Calendar then = Calendar.getInstance();
    yearsOpened = yearsOpened +
                  (monthsOpened / 12.0);
    return yearsOpened;
  }

}
```

```
public class Account {
 private CalendateDate dateOpened;

 public double getYearsOpen() {

    Calendar then = Calendar.getInstance();
    then.setTime(dateOpened.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);
    return yearsOpened;
}
```

Switch Statements

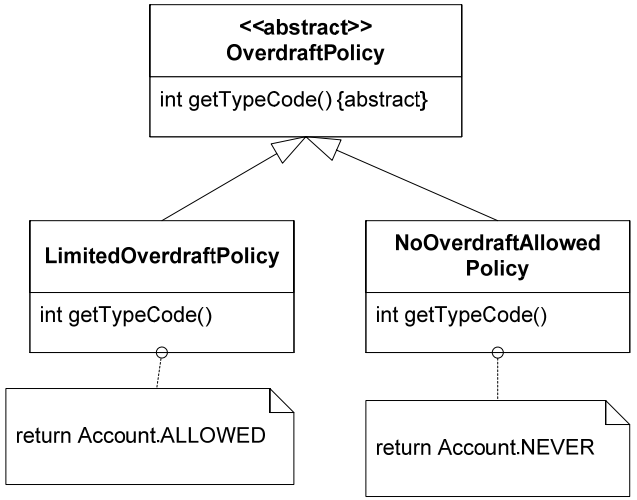
- Use of type codes and switch statements instead of polymorphism
- Key concepts are represented by type codes instead of classes
- Consequences:
  - Longer methods
  - Poor maintainability caused by code duplication
  - Increased code complexity
- Fix by introducing class hierarchy and moving each part of switch statement into a overriding method

```
public class Account {

 public static final int NEVER = 1;
 public static final int ALLOWED = 2;
 private int overdraftPolicy ;
...
```

```
public class MoneyTransferServiceProceduralImpl
implements MoneyTransferService {

public BankingTransaction transfer(String
                    fromAccountId, String toAccountId,
                    double amount) {
...
   switch (fromAccount.getOverdraftPolicy()) {
   case Account.NEVER:
     ...
     break;
   case Account.ALLOWED:
     ...
   default:
...
   }
  ...
}
```

Replace Type Code with Strategy

```
public class Account {

 public static final int NEVER = 1;
 public static final int ALLOWED = 2;

 private OverdraftPolicy overdraftPolicy;
...
```

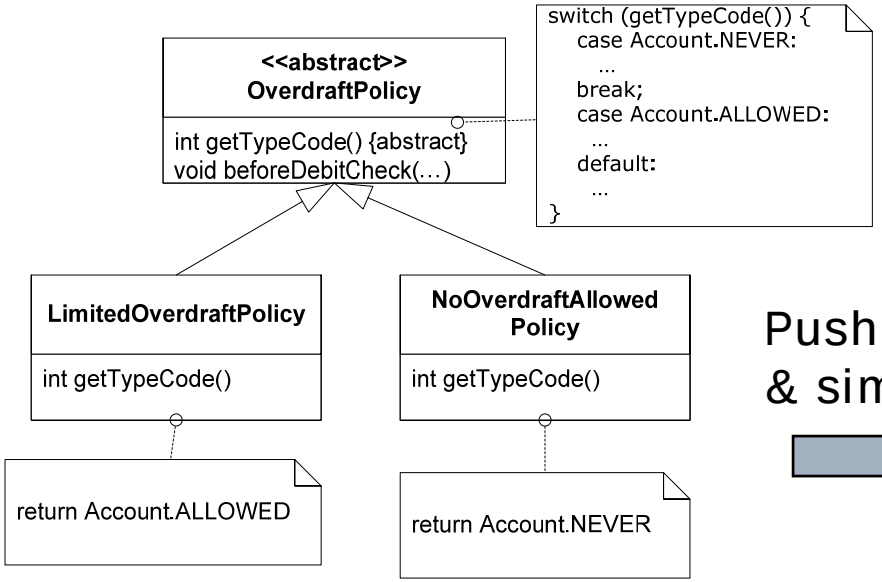



```
public class MoneyTransferServiceProceduralImpl … {

public BankingTransaction transfer(String fromAccountId,
                                   String toAccountId,
                                   double amount) {
    ...
   switch (fromAccount.getOverdraftPolicy().getTypeCode())
   {
    case Account.NEVER:
       ...
      break;
    case Account.ALLOWED:
      ...
    default:
...
    }
  ...
}
```

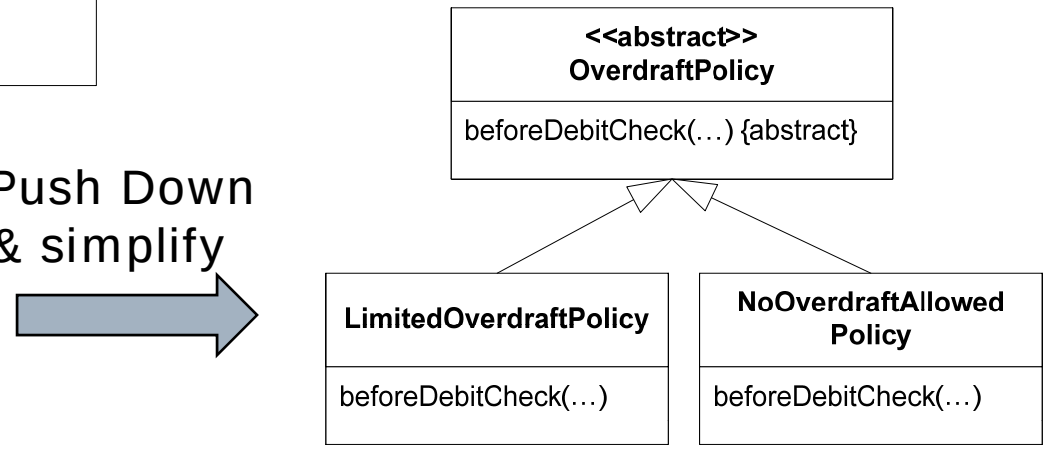
Replace Conditional with Polymorphism

Extract/Move Method

```
public class MoneyTransferServiceProceduralImpl … {

public BankingTransaction transfer(…) {
    …
    fromAccount.getOverdraftPolicy().beforeDebitCheck(…);
    …
}
```



Push Down & simplify



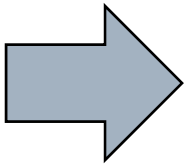
Data clumps

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

```
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() {}

}
```

Extract Class

Move Field



```
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 OverdraftPolicy overdraftPolicy;

Account() {}

}
```

```
public class OverdraftPolicy {

private int overdraftPolicy;
private double requiredYearsOpen;
private double limit;
...
```

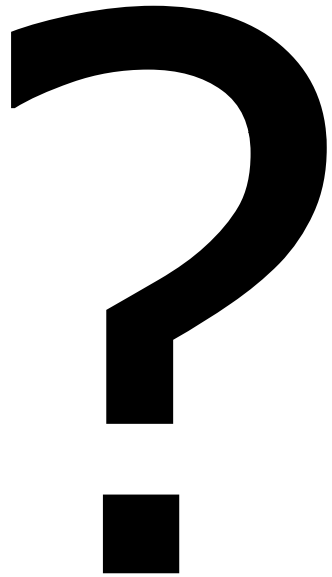
Summary

A rich domain model:

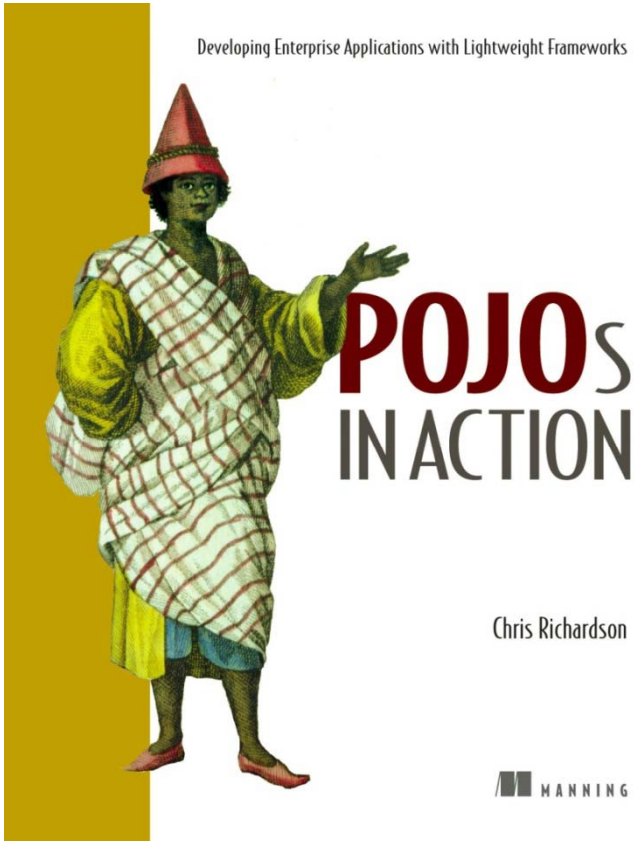
- Organizes the business logic as classes with state AND behavior
- Improves maintainability and testability
- Enabled by POJOs and non-invasive frameworks (mostly)
- Emerges from procedural code by incremental refactoring

Use it – starting Friday!

Q & A



Final thoughts



- Refactor your code on Friday
- Buy my book ☺
- Send email:

chris@chrisrichardson.net

- Visit my website:

http://www.chrisrichardson.net

- Talk to me about consulting and training