



Clean Systems

Clean Code at the Architecture Level

Dean Wampler

dean@objectmentor.com

IM's, Twitter: deanwampler

Robert C. Martin Series

PRENTICE
HALL

Clean Code

A Handbook of Agile Software Craftsmanship

Robert C. Martin

*How would you
build a city?*





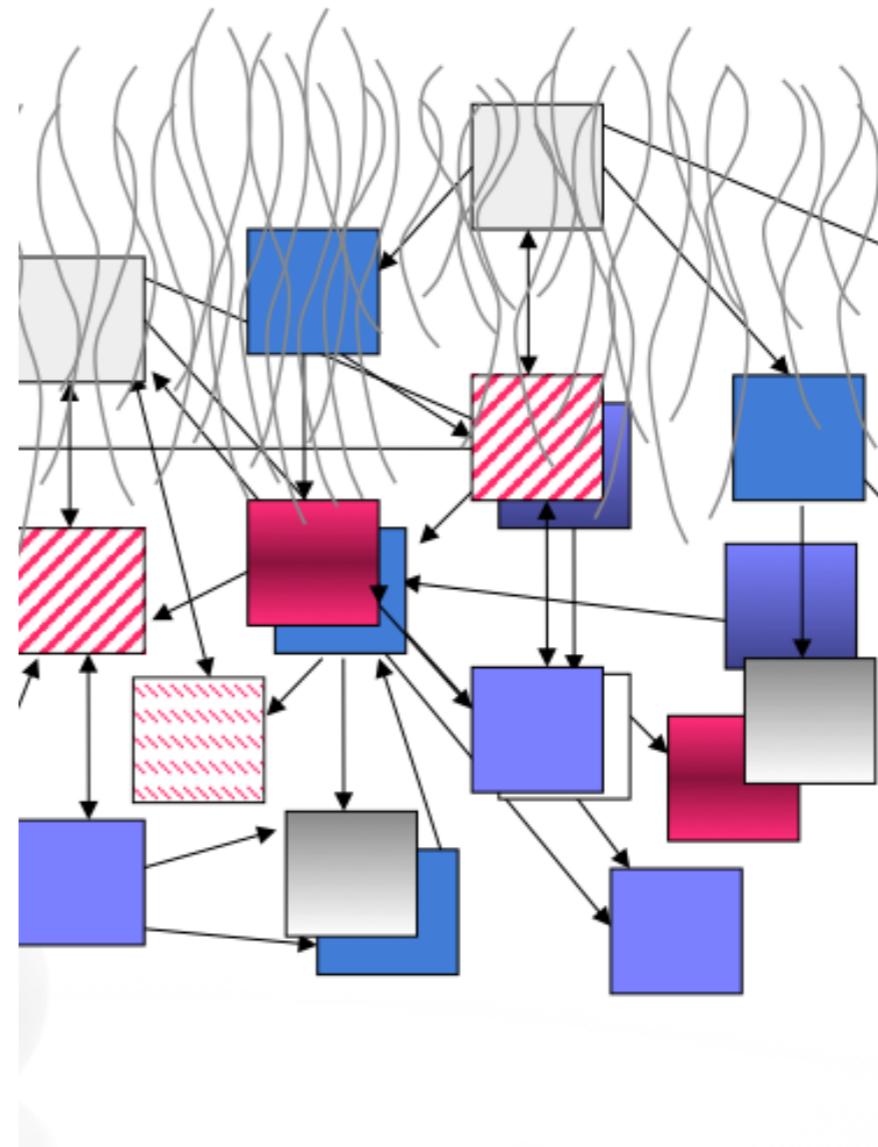
Cities are modular

- They have *appropriate* levels of abstraction.
- They separate *concerns*.



Your software systems?

- *Appropriate* abstractions?
- *Clear* separation of concerns?



*Clean systems
are built on
clean code*

Leave now if code makes you squeamish...

#1

Separate
construction
from
use

During *construction*

- People in hard hats.
- Lots of heavy lifting.
- ...



During *use*

- People in nicer clothes.
- Business tasks.
- ...



Software *construction* vs. *use*

- *Startup* is one task.
 - Component *wiring*.
- *Running* involves different tasks.

An example

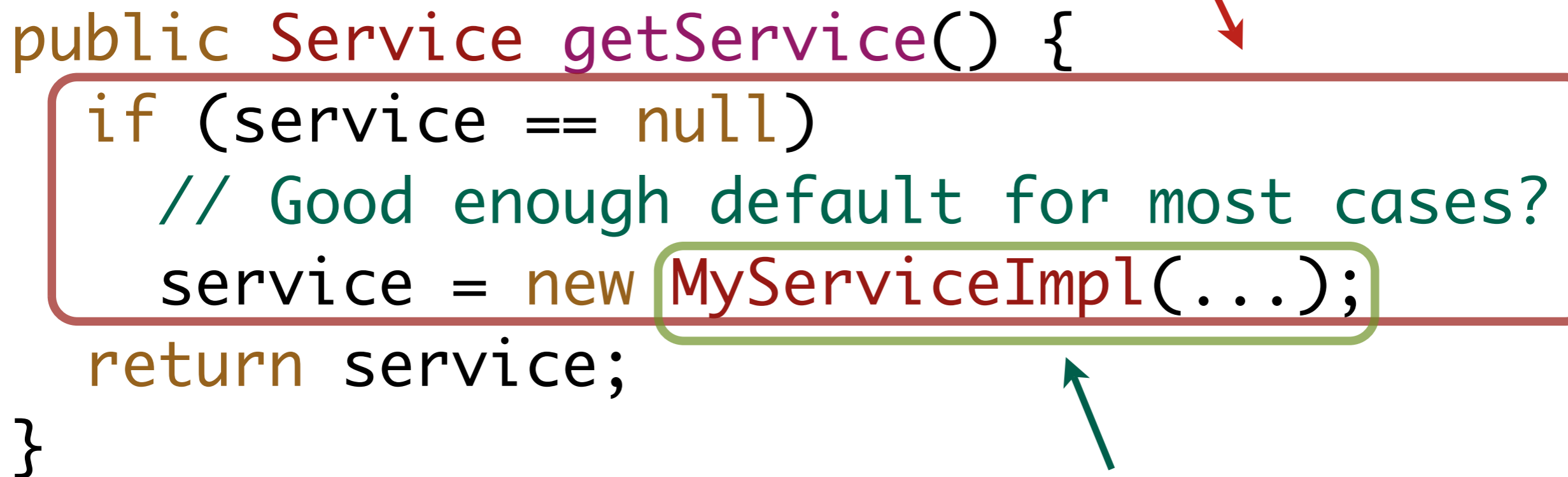
```
public Service getService() {  
    if (service == null)  
        // Good enough default for most cases?  
        service = new MyServiceImpl(...);  
    return service;  
}
```

Lazy Initialization Pattern

What's Wrong with LI?

“Wiring strategy” *scattered* and *tangled*.

```
public Service getService() {  
    if (service == null)  
        // Good enough default for most cases?  
        service = new MyServiceImpl(...);  
    return service;  
}
```



Specific *decisions* hard coded.

Other problems...

- Testing
 - Must somehow set a *mock* for *service* before *getService* called.
 - Must still compile with *MyServiceImpl*.
 - Is *MyServiceImpl* really the best *default*?
 - Breaks the *Single Responsibility Principle*

Setup *concern*

- Requires a *global strategy*.
- *Consistent* approach.
- *Modularized* decisions.

Solution

Dependency Injection

Dependency Injection

- Special type of *Inversion of Control*.
- Objects are *given* their dependencies
 - Passive vs. Active
- *Authoritative* mechanism makes *wiring* decisions.

Options

- Attribute “setters”.
- Constructor arguments.
 - Object leaves constructor fully formed.
 - Slightly better.

Spring Framework

```
<beans>
  <bean id="service"
    class="org.example.services.MyServiceImpl"/>

  <bean id="clientOfService" class="org.example.app.ClientImpl"
    p:service-name= "service"/>

  ...
</beans>
```

```
XmlBeanFactory bf = new XmlBeanFactory(
    new ClassPathResource("app.xml", getClass()));
Client client = (Client) bf.getBean("client");
```

Dependency Injection

- Separates *construction* from *use*.
- Decouples *abstractions* from *implementations*.

#2

Scale up
systems
on demand

Small vs. *Large* Systems



Common Myth:

*Why didn't you get the design
right the first time?*

Design Dilemma

- The *perfect* design for *today's* system.
- vs.
- The *perfect* design for *tomorrow's* system.

Agile methods

- Taught us to *evolve* the design to meet today's needs,
- But keep it *adaptable* for tomorrow's needs
 - Without *anticipatory design*.

Software is *unique*

- Unlike *physical* structures,
- We can *change everything* in software, even the *architecture*.

Software is *unique*

- ... but only if we keep it *agile*!

How *not* to keep it agile

EJB's versions 1 and 2

Enterprise Java Beans

- Forced *tangling* of *concerns*:
 - *Application* logic mixed with
 - *Container* life-cycle, etc.
 - *Persistence*,
 - etc.

Example: Bank EJB

```
public interface BankLocal  
    extends javax.ejb.EJBLocalObject {  
    ...
```

- Forced to subclass an EJB class.
- Can't use application *domain* hierarchy.
- Tight *coupling* to *container* details.

Example: Bank EJB

...

```
String getCity()
```

```
    throws java.ejb.EJBException;
```

```
void addAccount(AccountDTO dto)
```

```
    throws java.ejb.EJBException;
```

...

- Tight *coupling* for methods, too.

EJB implementation

```
public abstract class Bank  
    implements javax.ejb.EntityBean {  
    ...
```

- Forced to implement EJB interface

EJB implementation

...

```
public void addAccount(AccountDTO dto)
{
    InitialContext ctx =
        new InitialContext();
    AccountHomeLocal accountHome =
        ctx.lookup("AccountHomeLocal");
    AccountLocal account =
        accountHome.create(dto);
    Collection accounts = getAccounts();
    accounts.add(account);
}
```

EJB Implementation

...

// Required container methods...

```
public void ejbActivate() {}
```

```
public void ejbPassivate() {}
```

```
public void ejbLoad() {}
```

```
public void ejbStore() {}
```

```
public void ejbRemove() {}
```

...

But wait, there's more!

- Several *more classes* and *interfaces*.
- Many *more methods*.
- ***X**ML* to define
 - Transactions,
 - Persistence mapping,
 - Security, ...

Forget about:

- *Reuse.*
- *Object orientation* of your domain model.
 - => Lot's of duplication between *EJB's* and *POJO* domain objects.
- Easy *TDD.*
- High *productivity...*

But, not *all* was *wrong*

- Using *XML* to specify transactional, persistence, and security behaviors,
 - Separated these concerns from *code*.
- EJBs anticipated *Aspect-Oriented Programming*.

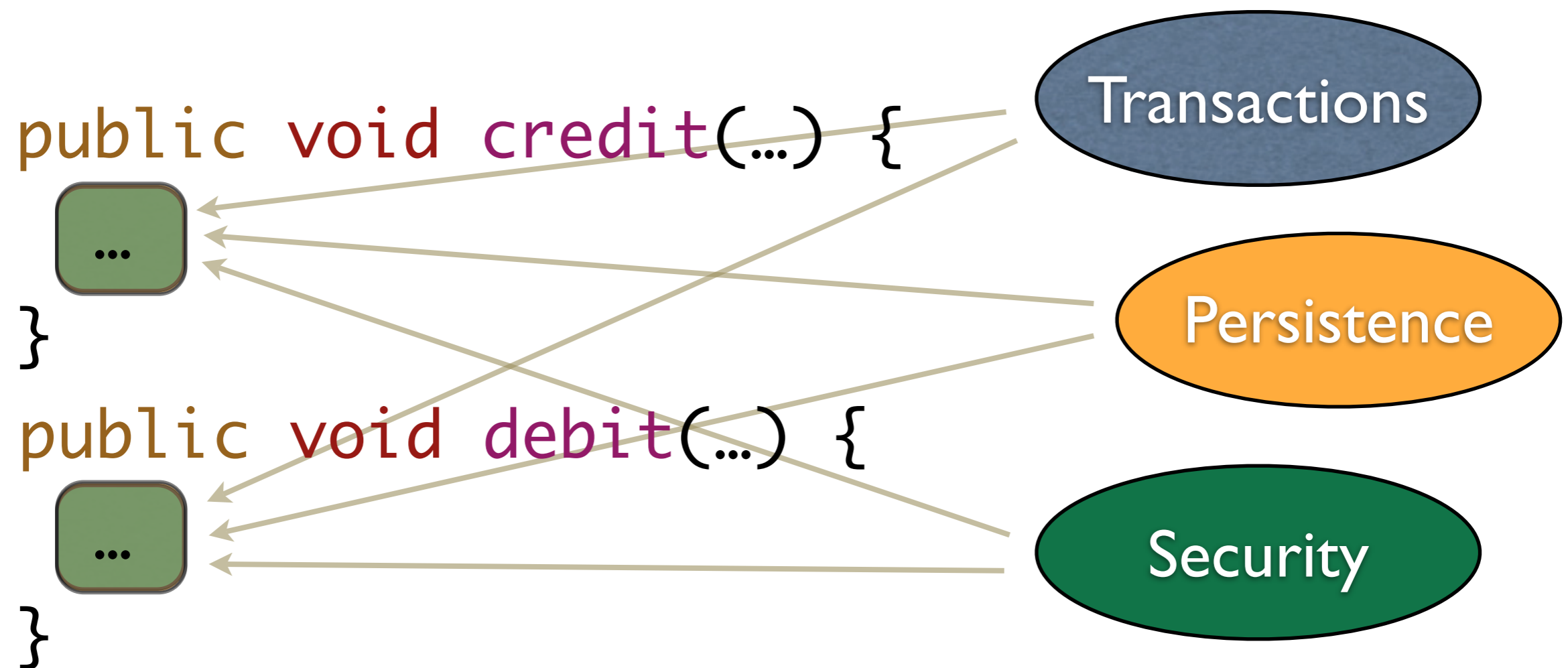
Solution

Aspect-Oriented Programming

```
public class BankAccount {  
    private Money balance;  
    public Money getBalance () {...}  
  
    public void credit(Money amount) {  
        balance += amount;  
    }  
    public void debit(Money amount) {  
        balance -= amount;  
    }  
    ...  
}
```

Clean Code

However, real applications need:

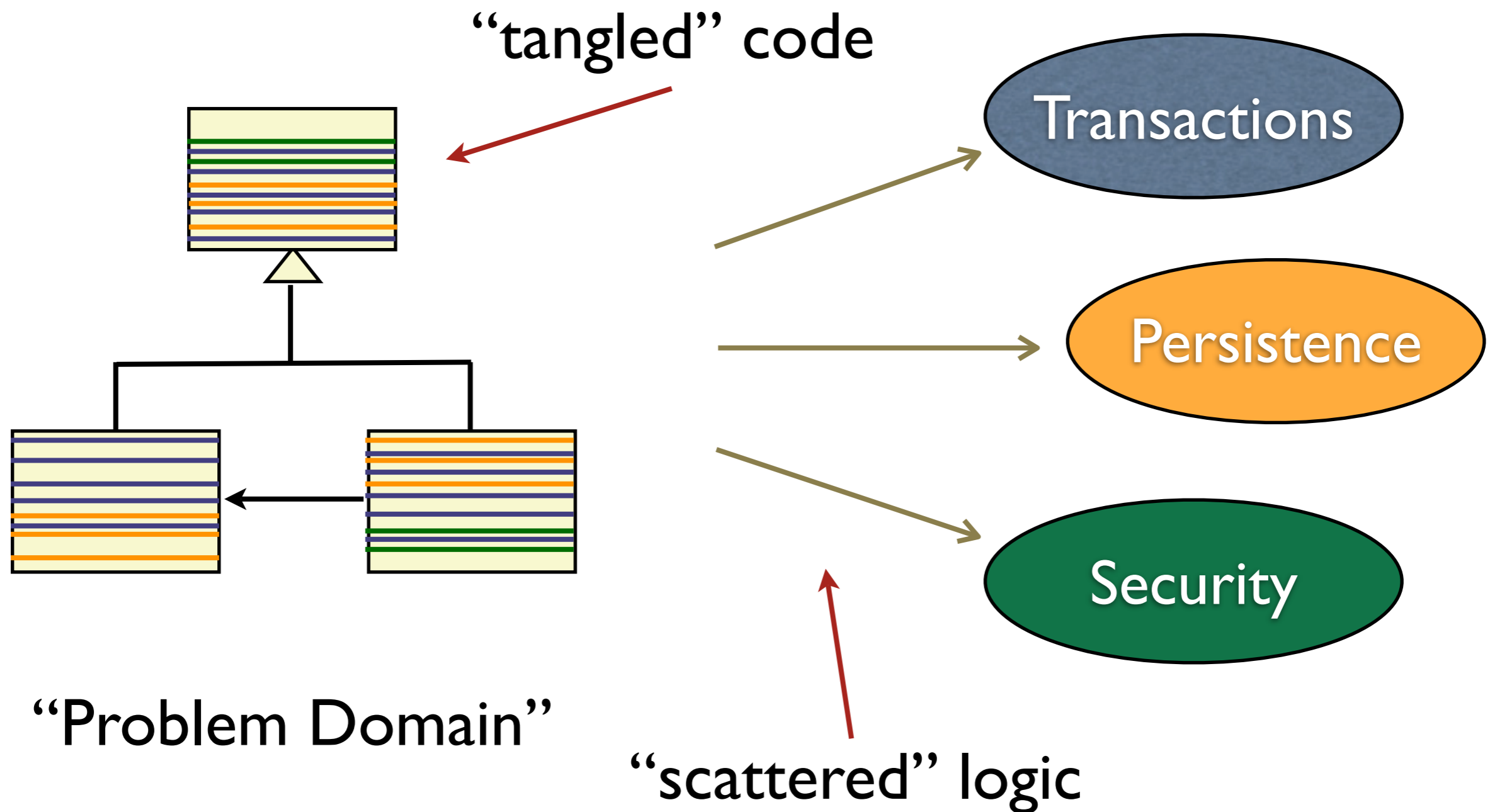


So **credit** becomes...

```
public void credit(Money amount)
    throws ApplicationException {
    try {
        Money oldBalance = balance;
        beginTransaction();
        balance += amount;
        persistChange(this);
        ...
    }
}
```

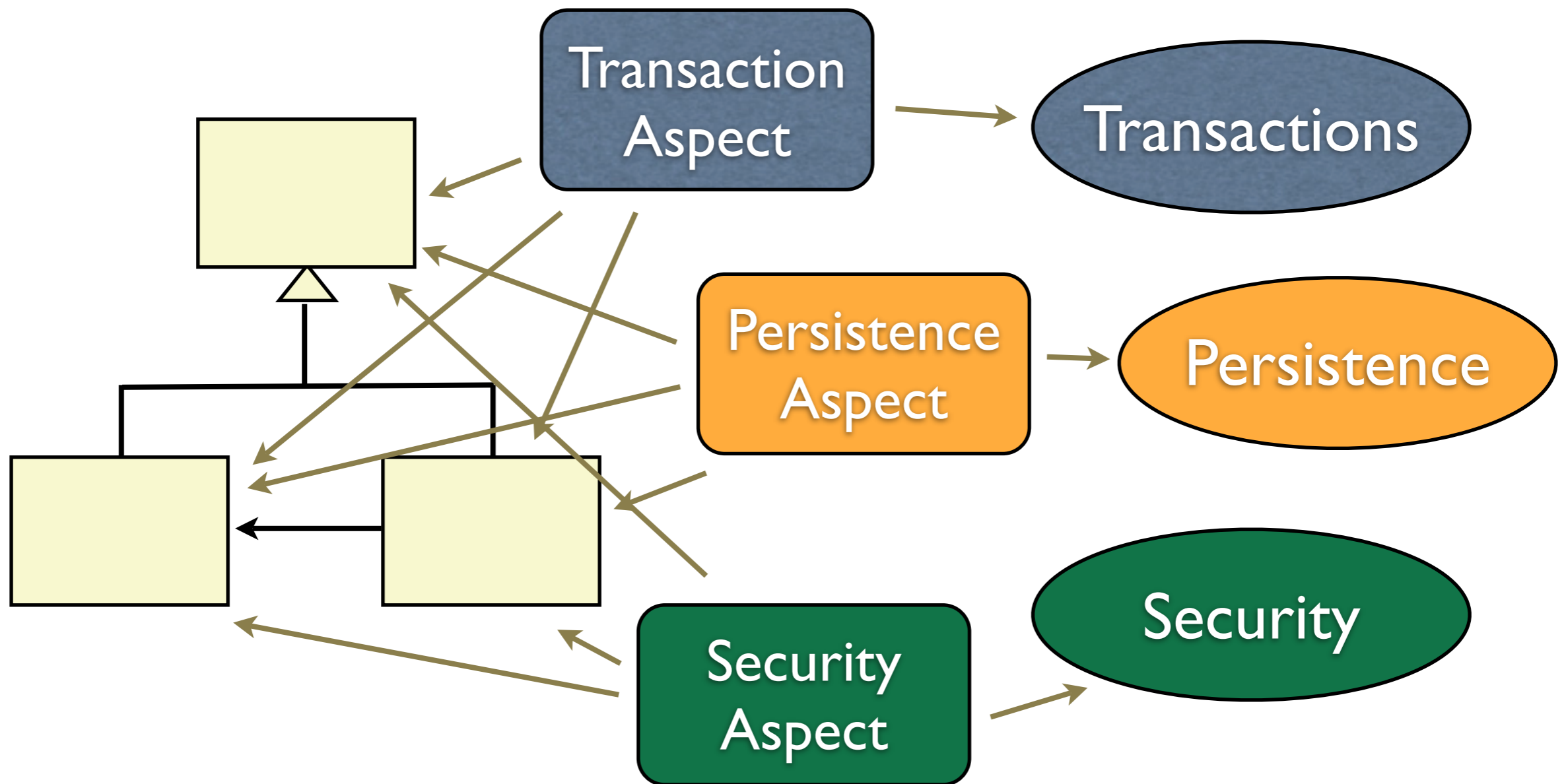
```
catch (Throwable th) {  
    logError(th);  
    balance = oldBalance;  
    throw new ApplicationException(th);  
} finally {  
    endTransaction();  
}  
}
```

We're mixing *multiple domains*, with fine-grained *intersections*.



Objects alone *don't*
prevent *tangling*.

Aspects restore *modularity* by encapsulating the *intersections*.



AOP tool options

- *AspectJ*
- “Pure Java” *Spring AOP* or *JBoss AOP*

AspectJ

```
public aspect PersistentBankAccount {  
    pointcut stateChange(BankAccount ba):  
        (call(* BankAccount.debit(..)) ||  
         call(* BankAccount.credit(..))) &&  
         this(ba);  
  
    after(BankAccount ba): stateChange(ba) {  
        persistChange(ba);  
    }  
}
```

AspectJ

“Class-like” construct



```
public aspect PersistentBankAccount {
```

```
pointcut stateChange(BankAccount ba):
```

```
    (call(* BankAccount.debit(..)) ||
```

```
     call(* BankAccount.credit(..))) &&
```

```
    this(ba);
```



When state changes occur.

```
after(BankAccount ba): stateChange(ba) {
```

```
    persistChange(ba);
```

```
}
```



When and how to persist changes.

```
}
```

Spring AOP

```
<beans>
```

```
<bean id="bankDataSource"
```

```
class="org.apache.commons.dbcp.BasicDataSource"
```

```
destroy-method="close"
```

```
p:driverClassName="com.mysql.jdbc.Driver"
```

```
p:url="jdbc:mysql://localhost:3306/mydb"
```

```
p:username="me" />
```

```
<bean id="bankDataAccessObject"
```

```
class="com.banking.persistence.BankDataAccessObject"
```

```
p:dataSource-ref="bankDataSource"/>
```

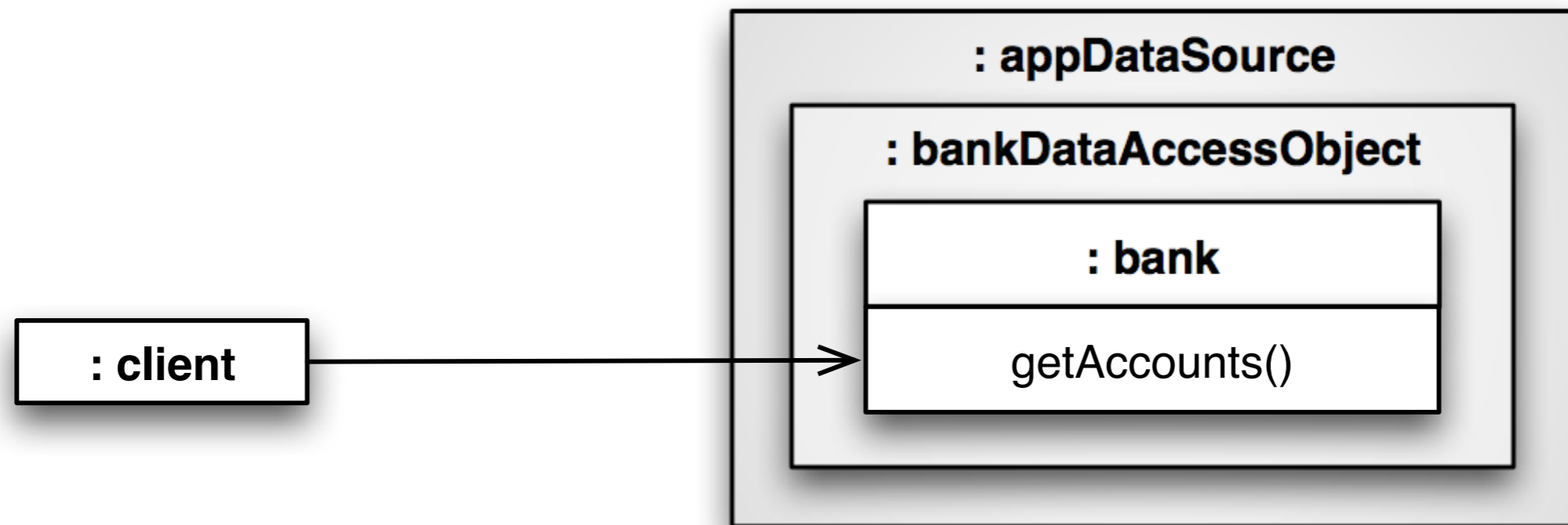
```
<bean id="bank"
```

```
class="com.banking.model.Bank"
```

```
p:dataAccessObject-ref="bankDataAccessObject"/>
```

```
</beans>
```

*“Matryoshka” doll**



* *i.e.*, Russian nested doll

EJB version 3

- Largely adopted the *POJO* model of *Spring AOP*.

Aspect-Oriented Programming

- Separates *concerns* with *fine-grained* coupling.
- Allows *concerns* to *evolve* and *scale independently*.
- Allows *architectures* to *evolve* and *scale*.

#3

Test drive
the
system architecture

Cities are *modular*

- Discrete components.
- Minimal coupling.
- Concurrent modifications.
- Concurrent execution.



They *grow* from *villages*

- Dirt roads are replaced by paved roads.
- Highways are added.
- Small buildings are replaced with towers.

The transition can be *painful* at times.

Big Design Up Front?

- *Architecture evolution* is possible if,
 - The *components* that implement *concerns* are *decoupled* from one another and
 - The *components* are *wired* together using *aspect-like* mechanisms.

Hazards of *BDUF*

- You're *thinking* in a *vacuum*, without *feedback* from a running system.
- It's hard to *throw* the design *away* when you've *invested* so much into it.

Solution

Test-Driven Development

In TDD,
tests are *proxies* for
requirements

Therefore, *grow* the
system in response to
“*test pressure*”

#4

Optimize *decision making*

The *best* decisions:

- are made at the *last responsible moment*,
- when you have the *most recent information*.

Solution

Incremental Evolution

With a *test-driven*
architecture,
you can
optimize decision making

Timing decisions

- You can make
 - many *small* decisions,
 - rather than *big, risky* decisions.

This is only possible
with an *agile architecture*

#5

Use *standards*
wisely,
when they add
demonstrable value

Benefits of Standards

- *Reuse* and *encapsulation* of
 - *ideas.*
 - *components.*
- Shared *expertise.*

Drawbacks of Standards

- *Slow to emerge.*
- *Design by committee.*
- *Bloat.*

Does the standard
meet the *needs* it was
intended to *serve*?

#6

Minimize the
mental gap
between
requirements and *code*

Does your *code*
read like the
problem domain?

Solution

Domain-Specific Languages

Every Domain has a *Language*

- Rich vocabulary.
- Idioms and patterns.
- Clear and concise communications.



Code should *read* like the *domain*

- *DSL's*
 - Introduce appropriate levels of *abstraction*.
 - Minimize *mental gap* between *domain concepts* and *code*.
 - *Optimize communication*.

Recap

- Separate *construction* from *use*.
- Use *dependency injection*.
- Scale up *on demand*.
- Decouple concerns with *Aspects*.

Recap

- *Test-drive* the system *architecture*.
- Requires *modular concerns*.
- *Optimize* decision making.
- An *agile architecture* lets you make *decisions* at the most *appropriate times*.

Recap

- Use *standards* wisely.
 - Only if they *demonstrate value*.
- Use *domain-specific languages*.
 - Map the *domain* to *code*.
 - Introduce appropriate *levels* of *abstraction*.

Final Thought:

Complexity kills. It sucks the life out of developers, it makes products difficult to plan, build and test. ... Each of us should ... explore and embrace techniques to reduce complexity.

Ray Ozzie, Chief Technology Officer, Microsoft Corporation

Thank You!

- dean@objectmentor.com
- <http://blog.objectmentor.com>
- <http://aspectprogramming.com/papers>