

Anti-Patterns

The Top 10 List

Common ways in which architects and developers shoot themselves in the foot

Copyright © 2005 Rob Daigneau



Overview

➤ Identify common design and coding missteps

➤ Provide insight as to how we can do better

➤ Goals

➤ Improve

➤ Ease of Maintenance

➤ Extensibility, Adaptability

➤ Performance

➤ Reduce

➤ Rework

➤ Head-aches

Assumptions on Experience

↗ Intermediate to Advanced (C# or VB.Net)

↗ Beginner to Advanced

↗ Design Patterns

↗ G04, J2EE, MS Patterns & Practices

↗ Basic understanding of UML

↗ Survey

↗ Your primary language

↗ Your experience with patterns

Introduction

Copyright © 2005 Rob Daigneau



We All Know What Design Patterns Are, Right?

➤ Quick Review

➤ Patterns Name and Describe

- Common design problems
- Common “best-practice” design solutions
- Trade-offs

➤ Goals of Design Patterns

- Reusable design solutions for frequent design problems
 - e.g. Clover-leaf, suspension bridge
- Lessons learned and codified

What's an Anti-Pattern ?

➤ Jim Coplien

“Something that looks like a good idea, but which backfires badly when applied.”

➤ Categories

➤ Design, Architecture

➤ Development

➤ Organizational, Management

➤ We're all guilty

Anti-Patterns Predominate

↗ Hard to filter list down to 10

↗ Enumeration/Codification not yet there

↗ Oftentimes many names for similar anti-pattern

↗ 7/10 of the names in this presentation are from moi

↗ Why: Wanted more concise, intuitive, easy-to-remember terms

↗ I am certainly not the first to recognize these anti-patterns

SOA vs Traditional OOD

- SOA mandates a dramatic paradigm shift
- SOA patterns will be influenced by traditional OOD
- Look for future presentations on SOA patterns and anti-patterns from yours truly

The List

Copyright © 2005 Rob Daigneau



Number 10

Patterns Fetish



Copyright © 2005 Rob Daigneau



Patterns Fetish

- Unreasonable and excessive use of design patterns
- Designer looks for places to use patterns

Patterns Fetish

Jason Tiscioni: developers.slashdot.org/comments.pl?sid=33602&cid=3636102

```
public class HelloWorld {  
  
    public static void main(String[] args) {  
  
        MessageBody mb = new MessageBody();  
  
        mb.configure("Hello World!");  
  
        AbstractStrategyFactory asf = DefaultFactory.getInstance();  
  
        MessageStrategy strategy = asf.createStrategy(mb);  
  
        mb.send(strategy);  
    }  
}
```

... but wait, there's more ...

Copyright © 2005 Rob Daigneau

Patterns Fetish

```
public interface MessageStrategy {  
    public void sendMessage();  
}  
  
public abstract class AbstractStrategyFactory {  
    public abstract MessageStrategy createStrategy(MessageBody mb);  
}  
  
public class MessageBody {  
    Object payload;  
    public Object getPayload() {  
        return payload;  
    }  
    public void configure(Object obj) {  
        payload = obj;  
    }  
    public void send(MessageStrategy ms) {  
        ms.sendMessage();  
    }  
}
```

Patterns Fetish

```
public class DefaultFactory extends AbstractStrategyFactory {
    private DefaultFactory() {};
    static DefaultFactory instance;
    public static AbstractStrategyFactory getInstance() {
        if (instance==null) instance = new DefaultFactory();
        return instance;
    }

    public MessageStrategy createStrategy(final MessageBody mb) {
        return new MessageStrategy() {
            MessageBody body = mb;
            public void sendMessage() {
                Object obj = body.getPayload();
                System.out.println((String)obj);
            }
        };
    }
}
```

Yo, Know What I'm Sayin'?

➤ Look at the design problem

➤ Select **ONLY** those patterns that address the current problem

➤ Favor simple solutions

➤ e.g. Page-Controller over Front-End Controller

Abort, Retry

```
public class HelloWorld {  
    public static void main(String[ ] args) {  
        System.out.println("Hello, world!");  
    }  
}
```

Number 9

The Swiss-Army Knife



Copyright © 2005 Rob Daigneau



The Swiss-Army Knife

Alex Papadimoulis:

<http://www.thedailywtf.com/AddPost.aspx?PostID=30308&Quote=True>

➤ Occurs when ...

- You have more interfaces than classes

- The number of interfaces on a class produce a severe case of **vertigo**

➤ A Basic Litmus-Test

- Do your classes implement more than 3 interfaces?

Implements ISwissArmyKnife

```
public class ScheduleItemDisplay Implements  
    MessageListener, ErrorListener, HelpRequestListener,  
    OpenScheduleListener, AttributeBigListListener,  
    OKListener, DeleteScheduleListener,  
    RefreshScheduleListListener,  
    ScheduleWeekChangeListener, NetworkChangeListener,  
    ScheduleItemListener, LayoutChangedListener,  
    ItemListener, ScheduleChangeValidator,  
    InternalFrameListener, RenameScheduleListener,  
    CancelListener, MirrorPolicyControlListener  
{  
  
}
```

btw: This is Java; No language is immune from sucky solutions!!!

Stop the Madness !!!

➤ More != Better

➤ Too many interfaces ...

➤ Confusion

➤ Maintenance problems

➤ Each interface requires implementation of items on that interface

➤ Do your classes actually need to share an interface?

Why Use Interfaces Anyway?

- Interfaces are Contracts
- Should communicate intent
- Will become increasingly important with SOA

Abort, Retry

- Maybe your class needs to USE another class
 - Aggregation, Composition
- Consider the “Façade” design pattern
 - Façade selects inner objects to do the work



The Façade Pattern

```
public class MutualFund
{
    private CashAccount _cashAccount;

    public MutualFund(Customer customer)
    {
        _cashAccount = new CashAccount( customer.CashAccountID );
    }

    public void BuySecurity(int CUSIP, decimal shares)
    {
        Security security = SecurityFactory.Make(CUSIP);
        security.Buy(DateTime.Now, shares, _cashAccount);
    }
}
```

- This class provides a simplified interface
- Can deal with a myriad of Securities
- Hides and manages the interactions between Securities and the Cash Account

Number 8

Apocalypse
Ready



Copyright © 2005 Rob Daigneau



Apocalypse Ready

➤ Default design is for the “worst case”

➤ Insufficient evidence to support defensive approach


➤ Don't know the probability that the worst might occur

➤ Don't know the potential loss

Apocalypse Ready – The Absurd

```
try
{
    rdr = cmd.ExecuteReader();
}
catch (Exception ex)
{
    try
    {
        throw ex;
    }
    catch (Exception ex2)
    {
        log.error(ex2); // Welcome to the "Absurd Zone"
        throw new ApplicationException(ex2.Name, ex2);
    }
}
}
```

Huh ???



This is "real code" ... no foolin' !!!

Apocalypse Ready ...

Other Examples

- Improper use of Publish/Subscribe (i.e. Observer Pattern) to sync distributed data
 - What's the probability of the "lost update" problem?
- Almost any use of pessimistic locking
 - Kills concurrency and contributes to dead-locks

Number 7

The Crystal Ball



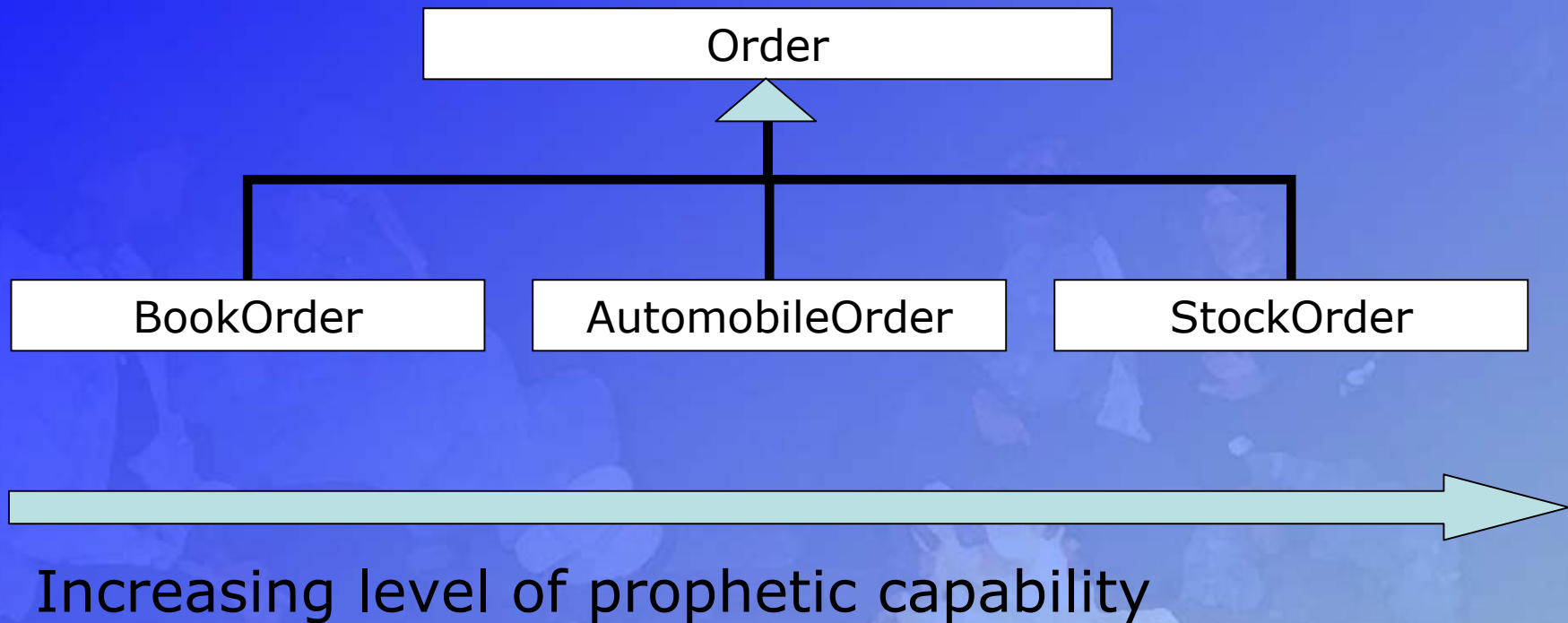
Copyright © 2005 Rob Daigneau



The Crystal Ball

- Designing for the unknown or future potentials
 - Little to no justification
- Design reaches out too far and assumes too much

The Crystal Ball - The Absurd



The Crystal Ball

- **Base classes are overly generic for the problem domain**
 - They implement methods/attributes that add little value
- **Sub-classes have little in common with their siblings**
 - **Remember the Sesame Street Rule:**
 - One of these things is not like the other



Number 6

The God Class



Copyright © 2005 Rob Daigneau



The God Class

➤ One “Do It All” class

➤ Symptoms

➤ Major tip-off = an excessive number of methods

➤ Lack of specialization, low cohesion

➤ One class manages all behaviors for what probably should be “logical subtypes”

One Conditional to Rule Them All

```
public void DoSomething( )
{
    switch( _customerType)
    {
        case (CustType.Gold)
            // do something for "gold" customers
            break;
        case (CustType.Silver)
            // do something for "silver" customers
            break;
        case (CustType.Bronze)
            // do something for "bronze" customers
            break;
    }
}
```

Abort, Retry

- Revisit the concepts of implementation inheritance and polymorphism

```
public abstract class Customer
{
    public abstract void DoSomething();
}

public class CustomerGold:Customer
{
    public override void DoSomething()
    {
        // do something here
    }
}
```

```
Customer customer = new CustomerGold( );
customer.DoSomething( );
```

I am Omnipotent

```
Customer customer = new Customer( parms);  
int orderId = customer.CreateOrder( parms);  
customer.AddOrderItem(orderId , parms);  
customer.SaveOrders( );
```

This class can apparently ...

- Create Customers
- Create Orders
- Add Items to an Order and Save the Order

Danger Will Robinson !!!

Abort, Retry

- Cohesive classes are easier to maintain
- Especially as object model changes

```
Customer customer = new Customer( parms);  
  
Order order = new Order( customer.CustomerId, other parms);  
  
order.Items.Add( new OrderItem( parms ) );  
  
order.Save();
```

Number 5

Object Orgy

**Mature
Audiences
Only**

Copyright © 2005 Rob Daigneau



Object Orgy

- Objects are a little too intimate with the members of other objects
- Objects “get around” a bit too much
 - Indiscriminate passing of objects
 - Intentional or and non-intentional

Object Adultery

➤ Why is this object messin' with another's attributes?

```
public class ProductDistributor{  
    ...  
    public void UpdateVendorProduct ( a bunch o' parms){  
        Product product = _currentVendor.GetProduct(productId);  
        product.Category = newCategory;  
        product.Cost = newCost;  
        etc.  
    }  
}
```

➤ Should the class be manipulating members of some other class ?

Passing Objects Around

What's the difference between this ...

```
public void UpdateOrder( OrderItem orderItem){  
    // do something with order  
}
```

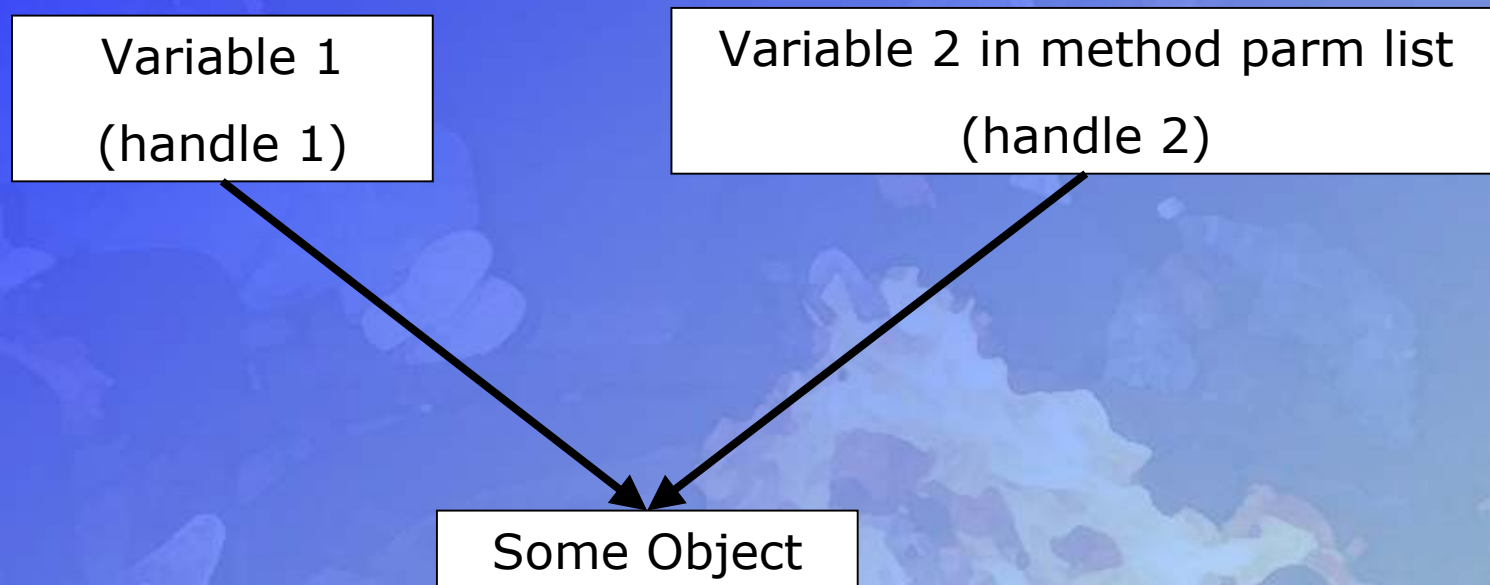
... and this ...

```
public void UpdateOrder( ref OrderItem orderItem){  
    // do something with order  
}
```

???????

Passing Objects Around

- Passing objects “By Value” only means ...
 - Make a copy of the HANDLE to a memory location
 - Pass this new handle



Implications for Child Objects

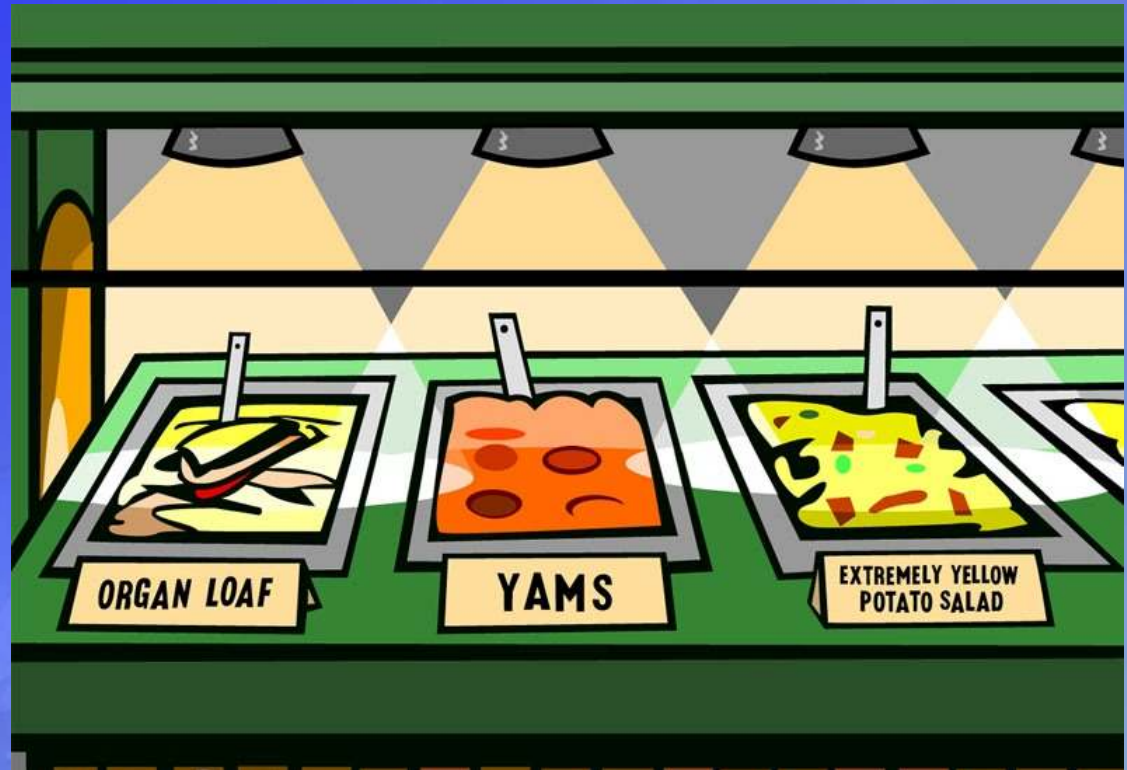
- Consider whether or not child objects should be exposed

```
public class Customer{  
    ...  
    public ArrayList Orders{  
        get{ return _orders;}  
    }  
}
```

- Use cloning when appropriate

Number 4

Data Buffet



Copyright © 2005 Rob Daigneau



Data Buffet




- ↗ The object retrieves too much data
- ↗ All data for all child objects are retrieved on instantiation of outermost object
- ↗ Will children be accessed?

Data Buffet

```
public class Vendor{
    private int _vendorId;
    private ArrayList _products=null;

    public class Vendor( int vendorId){
        _vendorId = vendorId;
        _products= this.GetProducts( _vendorId );
    }
    public ArrayList Products
    { get{ return _products;} }
}
```

 Imagine that each product retrieves data ...

-  Pricelists for different regions
-  Related products
-  Etc.

Abort, Retry ... Lazy Load

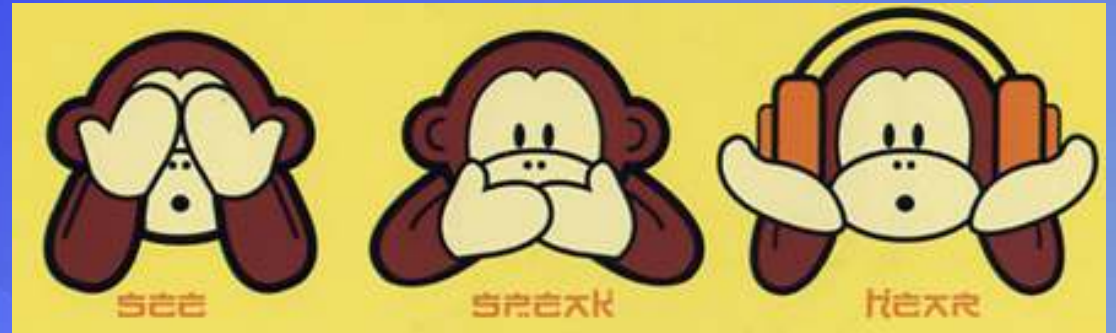
```
public class Vendor{
    private int _vendorId;
    private ArrayList _products=null;

    public class Vendor( int vendorId){
        _vendorId = vendorId;
    }

    public ArrayList Products{
        get{
            if ( _products == null)
                _products = this.GetProducts( _vendorId );
            return _products;
        }
    }
}
```

Number 3

See No Evil



Copyright © 2005 Rob Daigneau



See No Evil

- ↗ The opposite of “Apocalypse Ready”
- ↗ The assumption that nothing will go wrong
- ↗ Most common = lack of defensive programming
 - ↗ Examples
 - ↗ Not checking input parameters to methods
 - ↗ Not checking to see if a class supports a type before we cast

See No Evil

➤ We don't need no stinking transactions ...

```
foreach( OrderItem item in Order){  
    item.Save( );  
}
```

➤ What happens if an exception occurs part way through?

Abort, Retry : Case 1

➤ ADO.Net transactions orchestrated by an Order business object

```
SqlTransaction transaction=null;

try
{
    this._dbConn = DBHelper.GetInstance(); // Call a singleton
    transaction = _dbConn.BeginTransaction( );

    foreach( OrderItem item in _orderItems){
        item.Save( this._dbConn, transaction );
    }
    transaction.Commit( );
}
catch
{
    if(transaction!=null) transaction.Rollback( );
}
```

Abort, Retry : Case 2

- Submit multiple operations as one batch
 - Let the “receiver” manage the request as one transaction
- Concept is over 40 years old
 - Master-File Maintenance

Batch Updates

➤ Step 1

➤ Build an XML document that describes what to do

```
StringBuilder xmlDoc = new StringBuilder( );  
foreach( OrderItem item in _orderItems){  
    xmlDoc.Append(item.ToXML( ) );  
}
```

```
<OrderItems>  
  <Item  ID="1" Count="20" Txn="Add" />  
  <Item  ID="2" Count="1" Txn="Update" />  
  <Item  ID="3" Count="5" Txn="Add" />  
  <Item  ID="4" Txn="Delete" />  
</OrderItems>
```

Abort, Retry : Case 2 (continued)

➤ Step 2

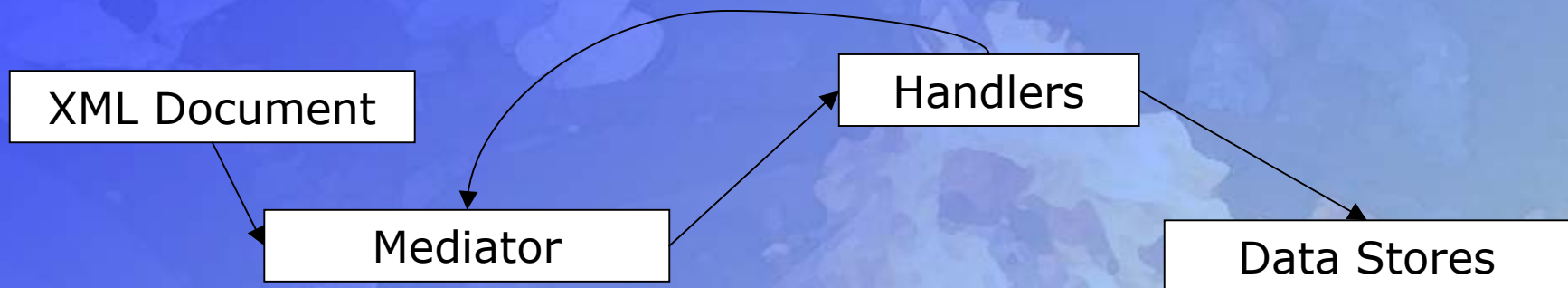
➤ Submit XML document to a “Mediator”

➤ Object or Stored Proc

➤ Step 3

➤ Mediator consumes document

➤ Identifies Logical Operations and dispatches to appropriate handlers



Abort, Retry : Case 2

```
CREATE PROCEDURE ProcessOrderItems(  
    @OrderID          INT,  
    @orderItemsXML    NVARCHAR(4000)  
)  
AS  
BEGIN  
    DECLARE @idoc INT  
  
    DECLARE @OrderItemsWork TABLE  
    (  
        ItemID          INT,  
        ItemCount       INT,  
        TransactionType VARCHAR(6)  
    )  
  
    EXEC sp_xml_preparedocument @idoc OUTPUT,  
        @orderItemsXML
```

Abort, Retry : Case 2 continued

```
INSERT
INTO @OrderItemsWork
SELECT * FROM OPENXML(@idoc, '/OrderItems/Item')
WITH (   ItemID           INT           '@ID',
        ItemCount        INT           '@Count'
        TransactionType  VARCHAR      '@Txn')
```

BEGIN TRAN

```
INSERT
INTO OrderItems
(   OrderID,           OrderItemID,
    ItemCount )
SELECT  @OrderID,      ItemID,
        ItemCount
FROM @OrderItemsWork
WHERE TransactionType = "Add"
```

```
IF @@ERROR != 0
BEGIN
```

```
        ROLLBACK
        RETURN
```

```
END
```

```
COMMIT TRAN
```

Number 2

Faux Object



Copyright © 2005 Rob Daigneau



Faux Objects

➤ Oftentimes stateless

or

➤ Objects with an abundance of static methods

➤ Exception = Helper Classes

➤ Classes look more like namespaces

➤ Methods look more like APIs

➤ Reminiscent of Procedural Languages

Faux Objects, Classes as Namespaces

(Bad, Not Good)

```
Customer customer = Customer.Insert( lots of parms );  
...  
Customer.Update ( customerId , lots of parms );  
...  
Customer.Delete ( customerId );
```

Abort, Retry (i.e. Better)

```
Customer customer = new Customer( lots of parms);  
...  
customer.Save( ); // Validates and saves new customer  
...  
// Change data on the customer object  
customer.Save( ); // Validates and Saves updated customer data  
...  
customer.Delete ( );
```

Numero Uno

The Mutant



Copyright © 2005 Rob Daigneau



The Mutant

- ↗ Objects exhibit unpredictable behaviors over time
 - ↗ A.k.a – The Blob, Big Ball of Mud
- ↗ Difficult to make changes without breaking functionality
- ↗ The following practices yield mutants ...
 - ↗ Copy/Paste Programming
 - ↗ Spaghetti-Code Madness
 - ↗ Solution-Sprawl
 - ↗ Ever-Changing Default Interfaces
- ↗ This stuff seems basic, but is the #1 problem (imho)

Copy/Paste Programming

➤ Code Duplicated in Haste

➤ Copied code might be tweaked only slightly

➤ Abort, Retry

➤ Stop yourself when you Ctrl-C, Ctrl-V

➤ Create private methods with parameters

➤ So what if the class has lots of private methods?

Spaghetti-Code Madness

➤ Long Methods (*Fowler, Beck*)

➤ Conditional Complexity (*Kerievsky*)

➤ i.e. nested ifs from here to eternity

➤ Abort, Retry

➤ Don't let routines exceed 1 printed page

➤ Break it down into smaller pieces of work

➤ Create private parameterized methods

➤ Don't go more than 3 levels deep in an "If construct"

Solution-Sprawl

➤ Solution Sprawl (*Kerievsky*)

- Responsibility for a task is spread over too many classes
 - i.e. tightly coupled classes
- Additions/Changes to system features require multiple changes in many classes

➤ Abort, Retry

- Problem is usually identified after it's too late
- Once anti-pattern is recognized, reigning it in is no simple task
 - Refactoring may employ many different approaches

Ever-Changing Interfaces

- Public methods and attributes change without thought to client or consumer
- Abort, Retry
 - Recognize that interfaces are contracts
 - You don't need to implement an interface explicitly to have an interface

Conclusion

Copyright © 2005 Rob Daigneau



Why Does This Occur ?

- Pressure from management to “just get it done”?
- Lack of training ? Our schools ?
- Tools that “dumb it down” but don’t encourage better design?

How Do We Change It?

- ↗ Continuous education
- ↗ Sharing of knowledge and lessons learned
- ↗ Show some backbone to management
 - ↗ Know when to swing ... perfection isn't required or even possible
 - ↗ It's not easy, but educate management
 - ↗ Time spent planning might seem unproductive BUT
 - ↗ Planning saves money and time

It's Like Walking a Tight-rope

We are all guilty of

Over-Engineering

AND

Under-Engineering



Final Thought

Everything should be made as simple as possible, but not simpler

Albert Einstein

... or as the “common man” might say ...
Keep It Simple Stupid (KISS)