# Automating Patterns with Aspect-Oriented Programming and PostSharp

**Gaël Fraiteur**

PostSharp Technologies
Founder & Principal Engineer

my twitter

@gfraiteur

# POSTSHARP


U·S AIRWAYS

Bank of America

ORACLE

mitchell

Moody's
ANALYTICS

CITADEL

SIEMENS

HealthcareSource

ebay

THALES

intel

FULLARMOR.

EADS

BOSCH

WELLS
FARGO

CGI

JPMorgan Chase & Co.

SOCIETE
GENERALE

PHILIPS

rovi

NetApp

AON

Deutsche Bank

Scotiabank

SolidWorks

COMMERZBANK

RGA

McKESSON
Canada

RelayHealth

SKANSKA

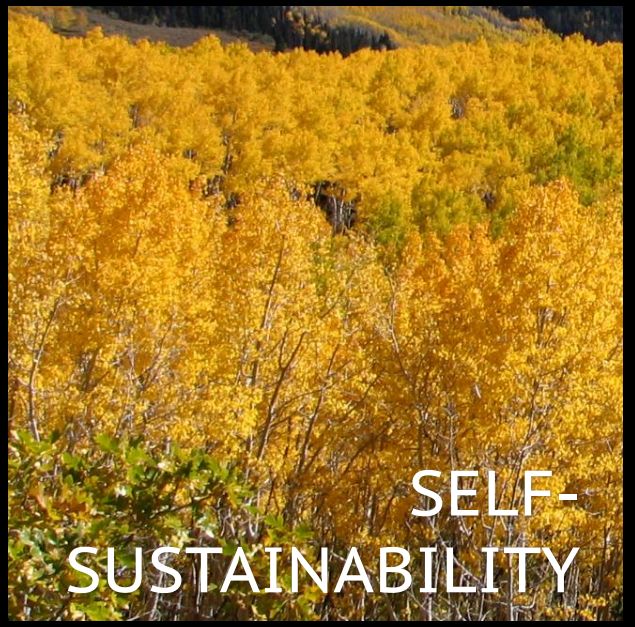
Get a FREE license of PostSharp Ultimate

**http://bit.ly/TODO-postsharp**

The vision:
Code at the right level of abstraction,
with compiler-supported design patterns

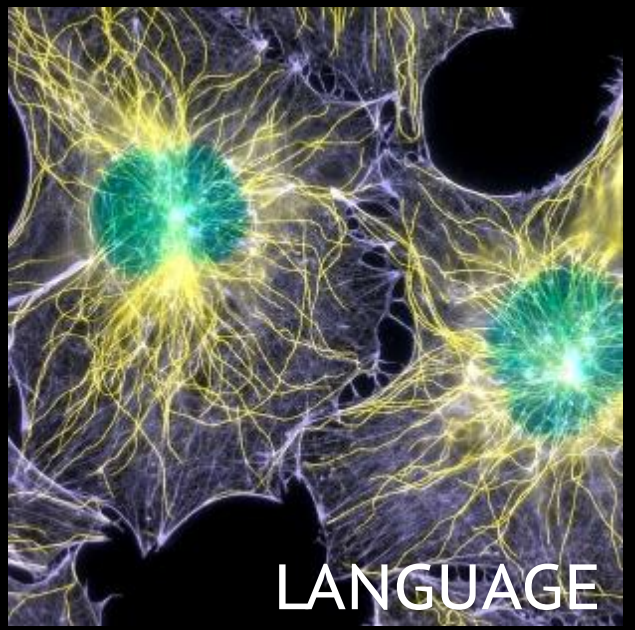

REGULARITY

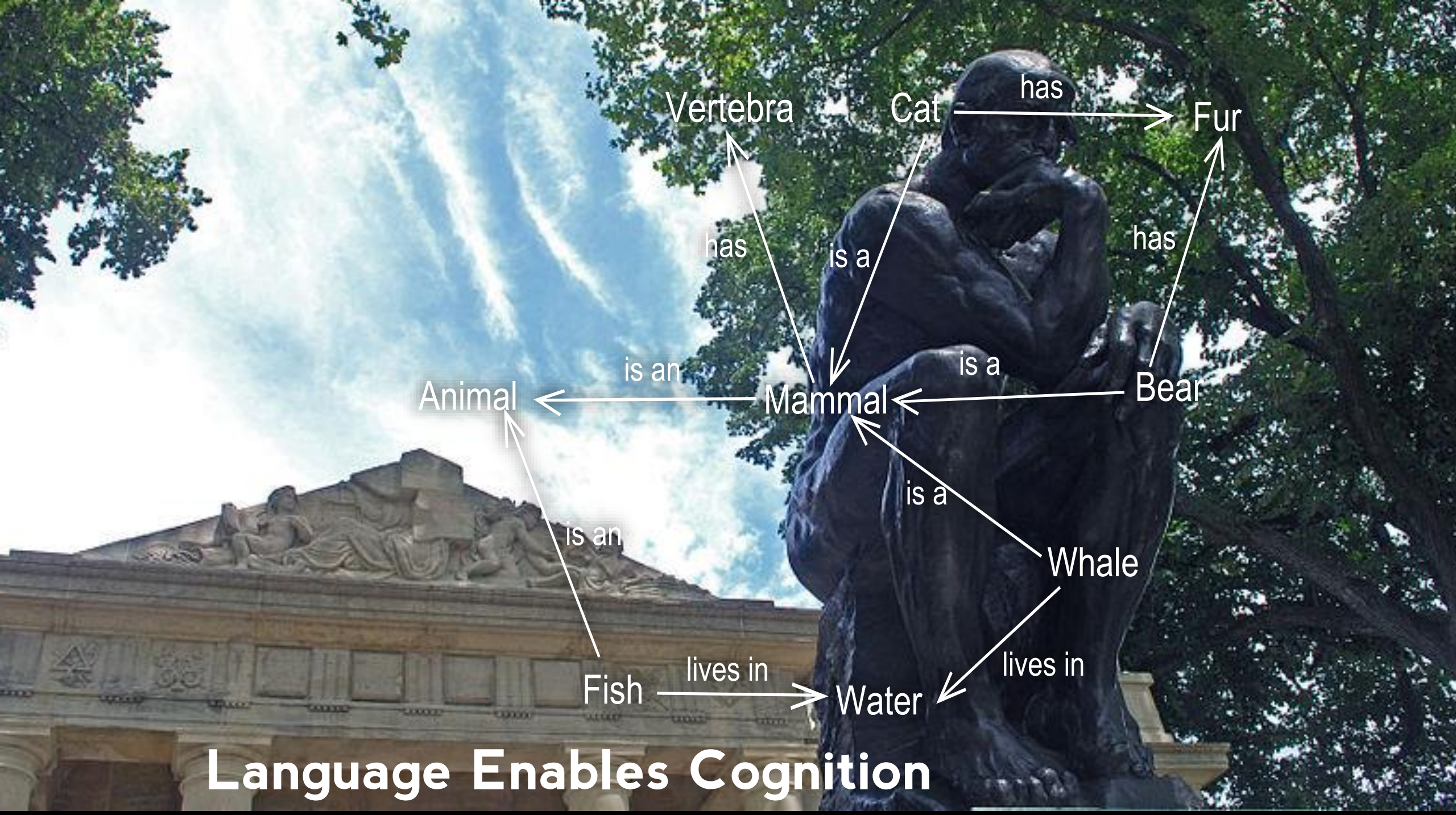

RESOLUTION
OF FORCES


SELF-
SUSTAINABILITY


VARIETY


UBIQUITY


LANGUAGE




# Language Enables Cognition

# Thinking with Patterns

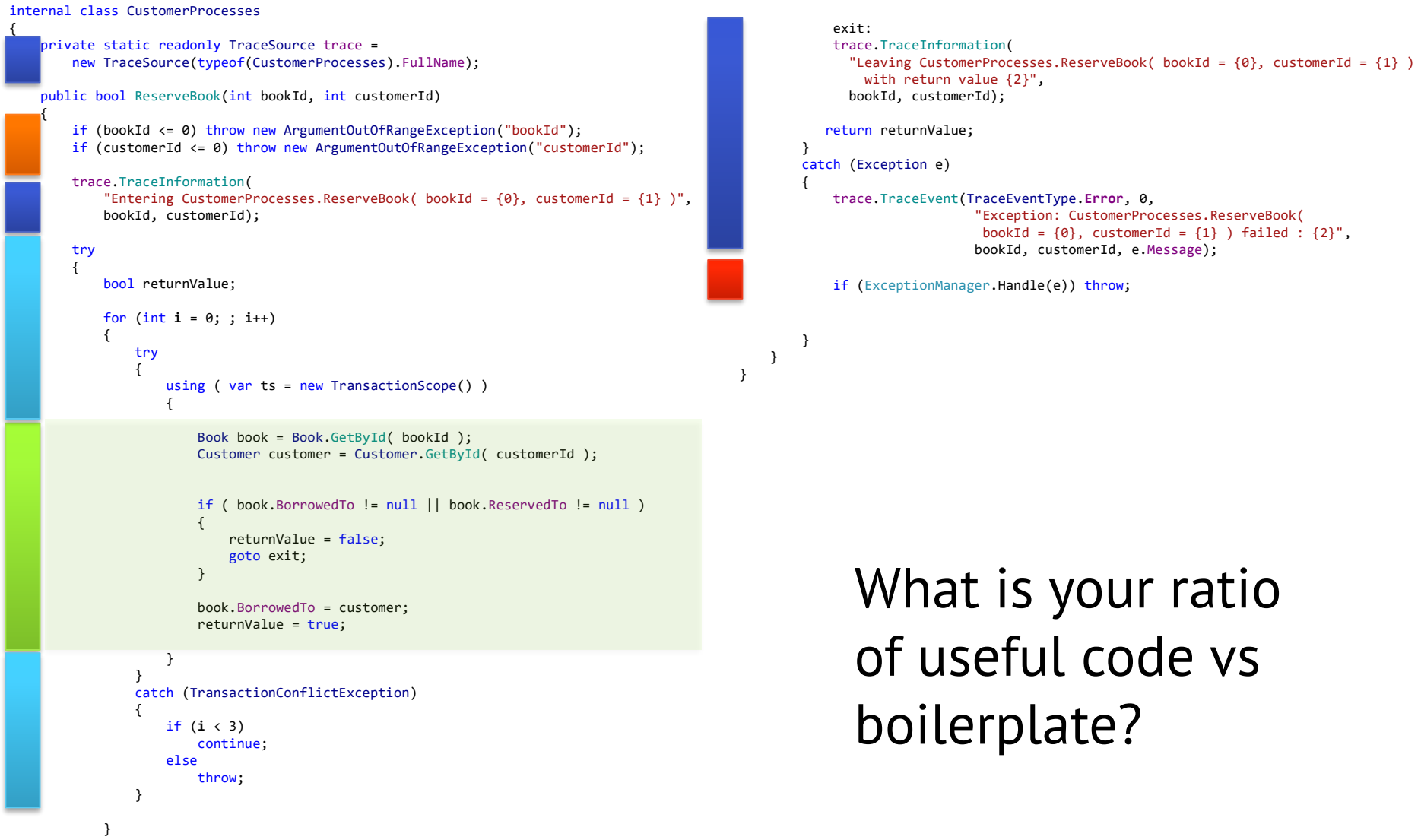
- Developers **think** at a high level of abstraction, using design patterns.

# Conventional Compilers Are Limited

- However, conventional compilers don't have a concept of patterns.
- Therefore developers write repeating code: boilerplate code.

# Typical Boilerplate

- INotifyPropertyChanged
- Undo/redo
- Code contracts (preconditions)
- Logging
- Transaction handling
- Exception handling
- Thread dispatching
- Thread synchronization
- Immutable
- Authorization
- Audit
- Caching

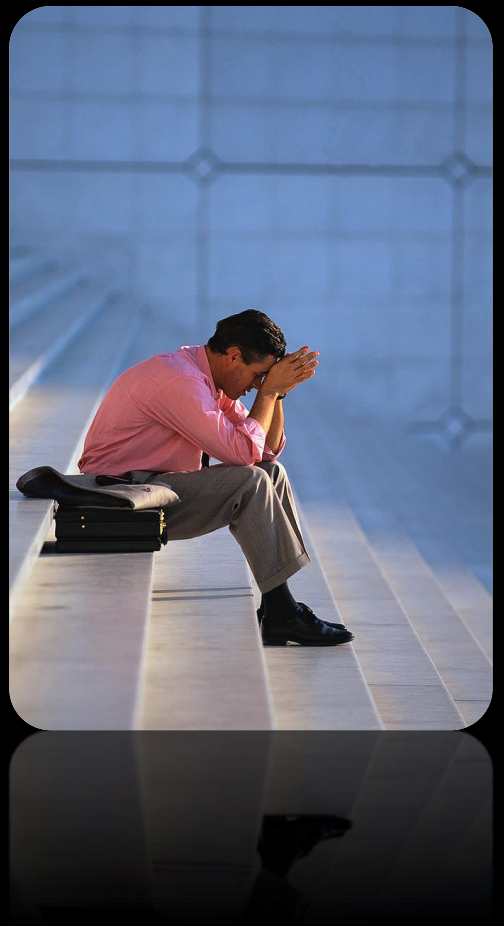


```
internal class CustomerProcesses
{
    private static readonly TraceSource trace =
        new TraceSource(typeof(CustomerProcesses).FullName);

    public bool ReserveBook(int bookId, int customerId)
    {
        if (bookId <= 0) throw new ArgumentOutOfRangeException("bookId");
        if (customerId <= 0) throw new ArgumentOutOfRangeException("customerId");

        trace.TraceInformation(
            "Entering CustomerProcesses.ReserveBook( bookId = {0}, customerId = {1} )",
            bookId, customerId);

        try
        {
            bool returnValue;

            for (int i = 0; ; i++)
            {
                try
                {
                    using ( var ts = new TransactionScope() )
                    {

                        Book book = Book.GetById( bookId );
                        Customer customer = Customer.GetById( customerId );


                        if ( book.BorrowedTo != null || book.ReservedTo != null )
                        {
                            returnValue = false;
                            goto exit;
                        }

                        book.BorrowedTo = customer;
                        returnValue = true;

                    }
                }
                catch (TransactionConflictException)
                {
                    if (i < 3)
                        continue;
                    else
                        throw;
                }

            }
            exit:
            trace.TraceInformation(
              "Leaving CustomerProcesses.ReserveBook( bookId = {0}, customerId = {1} )
                with return value {2}",
              bookId, customerId);

            return returnValue;
        }
        catch (Exception e)
        {
            trace.TraceEvent(TraceEventType.Error, 0,
                             "Exception: CustomerProcesses.ReserveBook(
                              bookId = {0}, customerId = {1} ) failed : {2}",
                             bookId, customerId, e.Message);

            if (ExceptionManager.Handle(e)) throw;


        }
    }
}
```

# What is your ratio of useful code vs boilerplate?

# Consequences of Boilerplate

- High development effort
- Poor quality of produced software
- Difficulty to add/modify functionality after first release
- Slow ramp-up of new team members



# The Big Question

- How can we produce high-quality software with less development effort… without having to replace your existing compiler?

- You may want to consider…

# Pattern-Aware Compiler Extensions

- Add support for patterns
- No more pattern hand-coding resulting in boilerplate!

# Pattern-Aware Compiler Extensions

- Four main reasons to consider pattern-aware compiler extensions:
  1. Stop writing boilerplate code and deliver faster
  2. Build more reliable software
  3. Easier to add/modify functionality after first release
  4. New members contribute quicker

# Reason 1. Deliver Cheaper and Faster

- Fewer lines of code means fewer hours of work
  - Outsource repetitive work to compiler and save time and costs immediately

# Reason 2. Build More Reliable Software

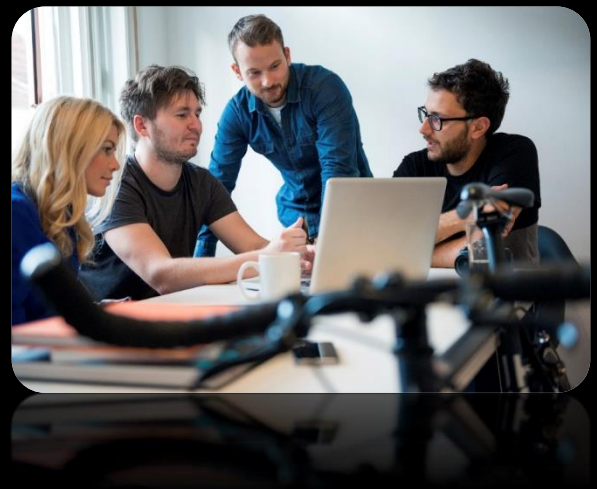
- Fewer lines of code means fewer defects
- Reliability becomes much more affordable
  - Cheaper to implement reliability features: logging, exception handling, caching, security,…
  - The “right” tool

# Reason 3. Easier to Modify Functionality

- Cleaner and shorter code is easier to understand
  - Focus on business logic and save time trying to understand the code
- Better architecture is future-proof
  - Define features such as logging, exception handling or transactions in one place and make their modification easy and fast

# Reason 4.
# New Team Members Contribute Quicker

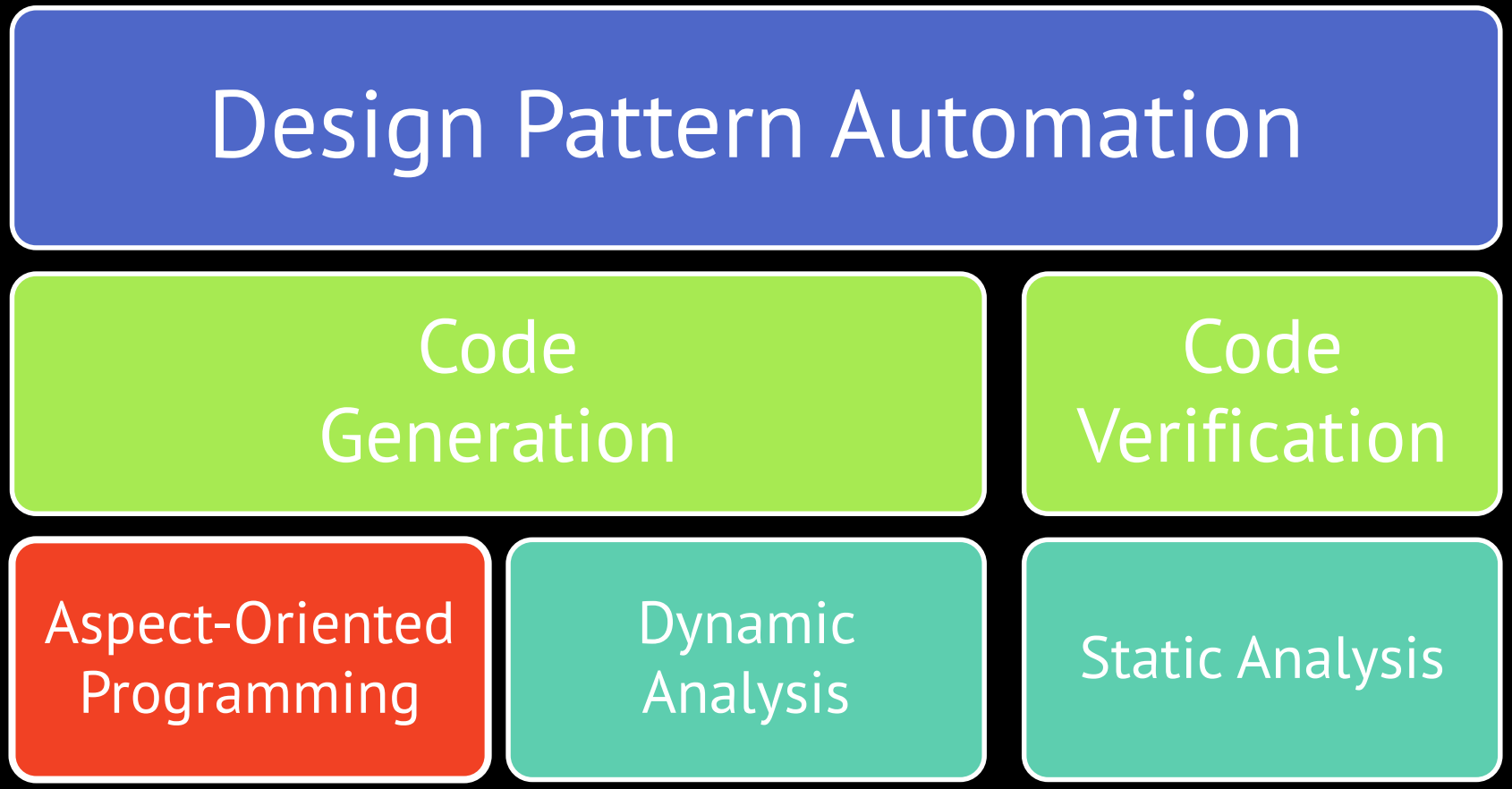
- Better division of labor
  - New team members can focus on business logic without worrying about complex architectural structures.
  - Encapsulate knowledge of senior developers into patterns and leverage them to the whole team.
- Implement a tighter feedback loop
  - Validate hand-written code against your defined rules at build time and detect bugs as quickly



# Lab 1: imagine a world with pattern-aware compilers

*Life is Good!*

# Automating Design Patterns



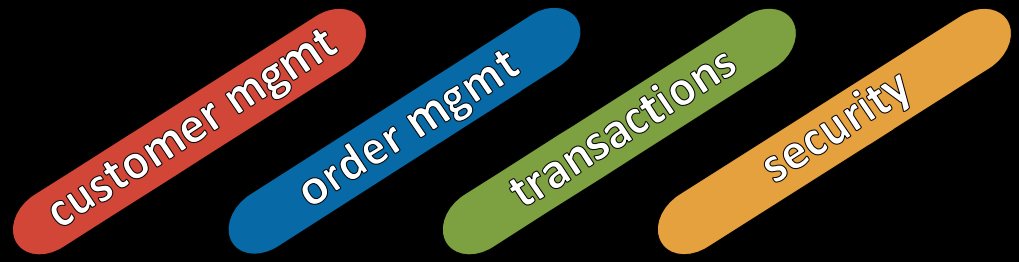
# Aspect-Oriented Programming

Problem Domain

## Cross-Cutting Concerns

Solution Domain

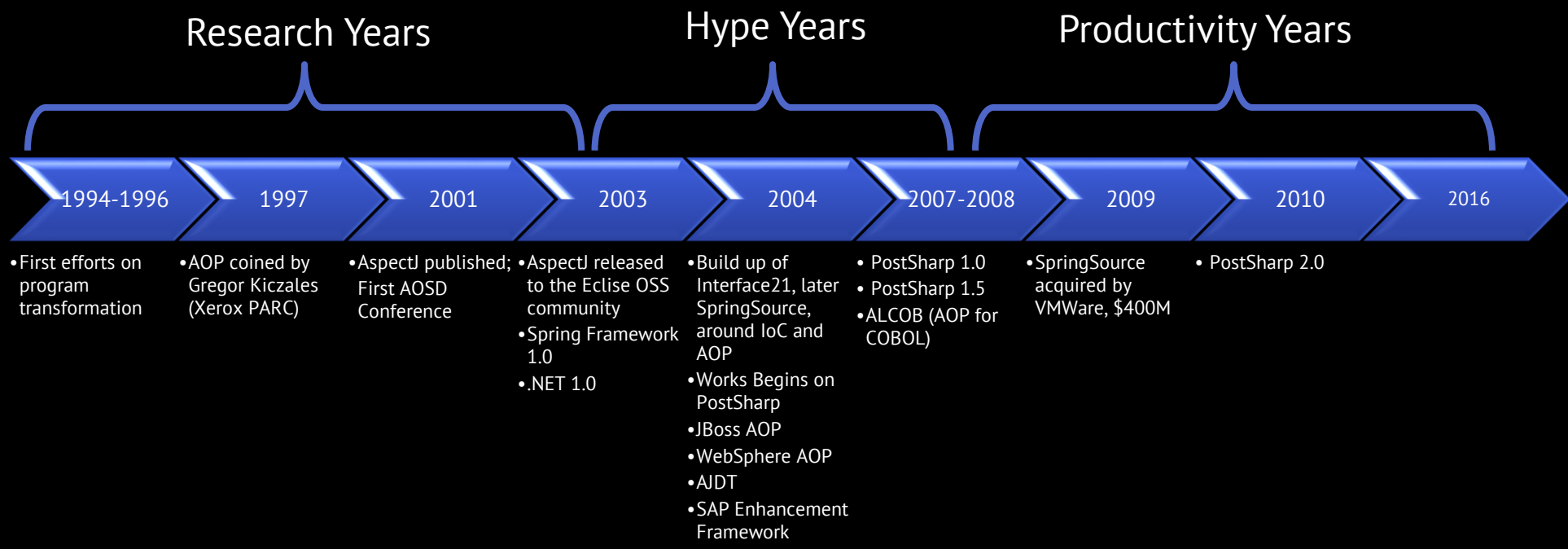
## Separation of Concerns



# What is AOP?

- An extension of (not an alternative to) OOP that addresses the issue of cross-cutting concerns by providing a mean to:
  - Encapsulate cross-cutting concerns
    into Aspects = collection of transformations of code
  - Apply aspects to elements of code

# 20 Years of AOP History



# AOP with PostSharp

# PostSharp Components

## Diagnostics

- Logging

## Model

- INotifyPropertyChanged
- Contracts

## Threading

- Threading Models
- Thread Dispatching
- Deadlock Detection

## Aspect Framework

- Aspect Primitives
- Composite Aspects
- Multicasting
- Aspect Providers

## Architecture Framework

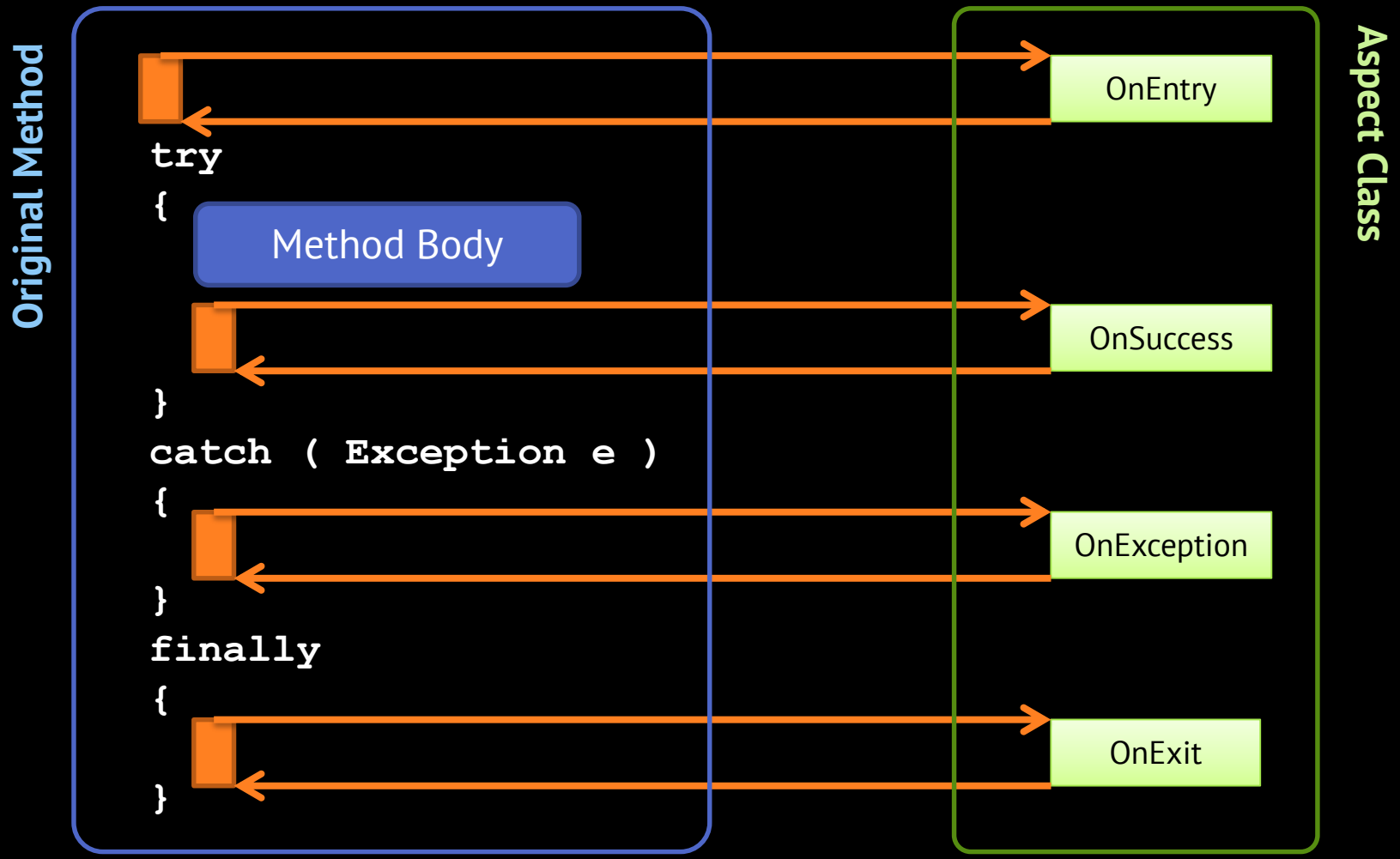
- Relationship browsing
- Syntax Tree
- Constraints

## SDK

# Aspect Types Listed

- MethodBoundaryAspect
  - OnEntry
  - OnSuccess
  - OnException
  - OnExit
- OnExceptionAspect
  - OnException
  - *GetExceptionType*
- MethodInterceptionAspect
  - OnInvoke
- LocationValidationAspect
  - ValidateValue
- **LocationInterceptionAspect**
  - OnGetValue
  - OnSetValue
- EventInterceptionAspect
  - OnAddHandler
  - OnRemoveHandler
  - OnInvokeHandler
- Composite advices
  - Introduce members
  - Introduce interfaces
  - Import member

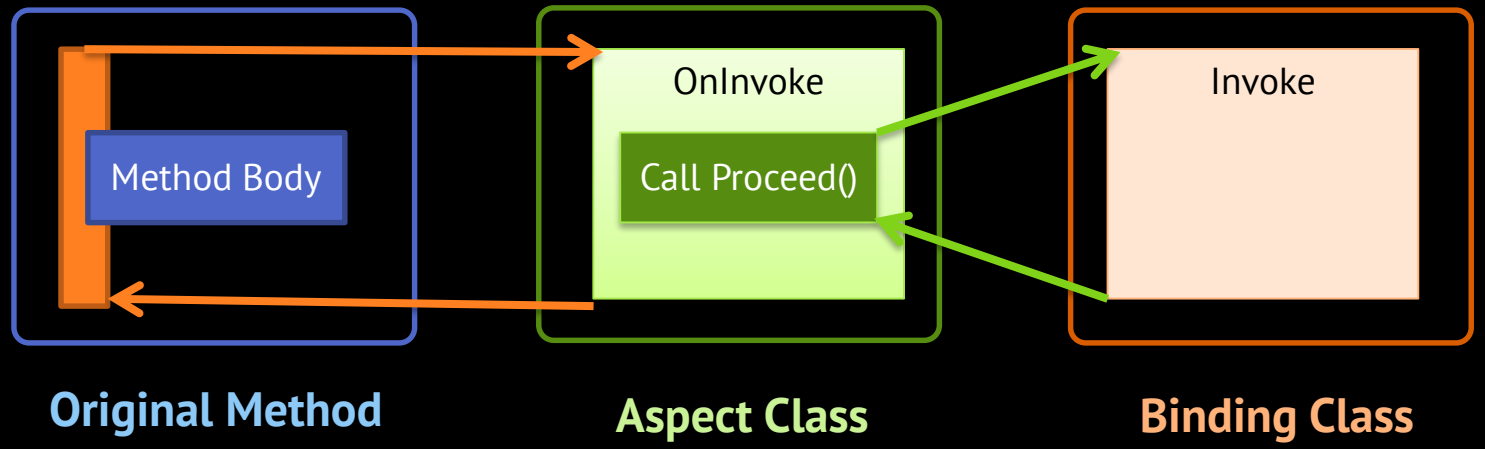
# OnMethodBoundaryAspect



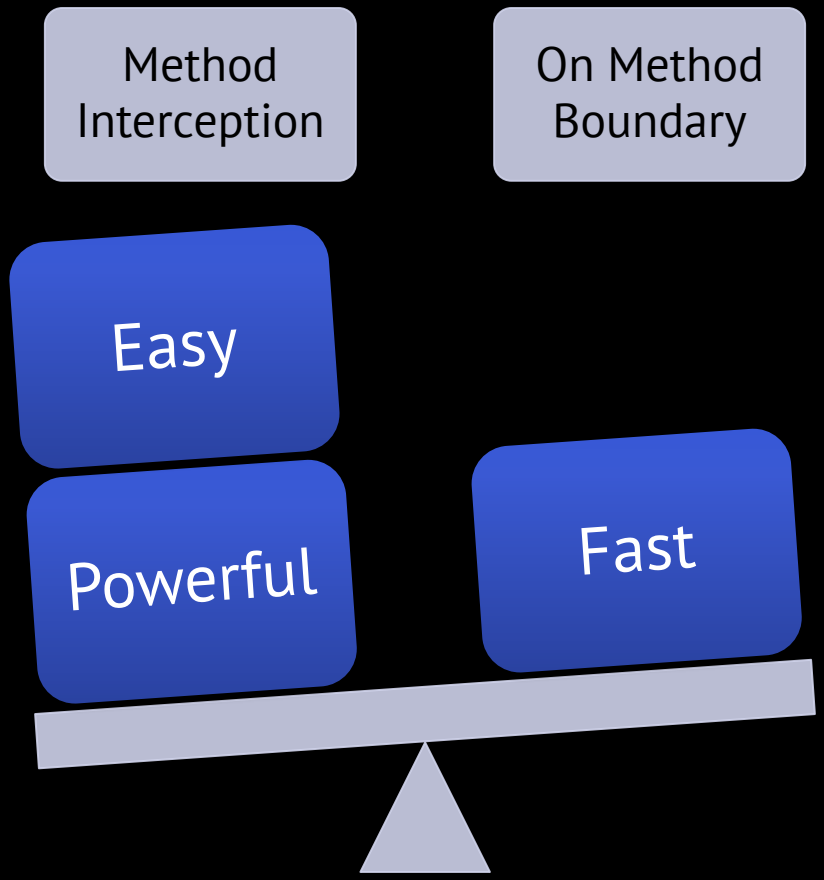
LAB 1

# Transaction Scope

# MethodInterceptionAspect



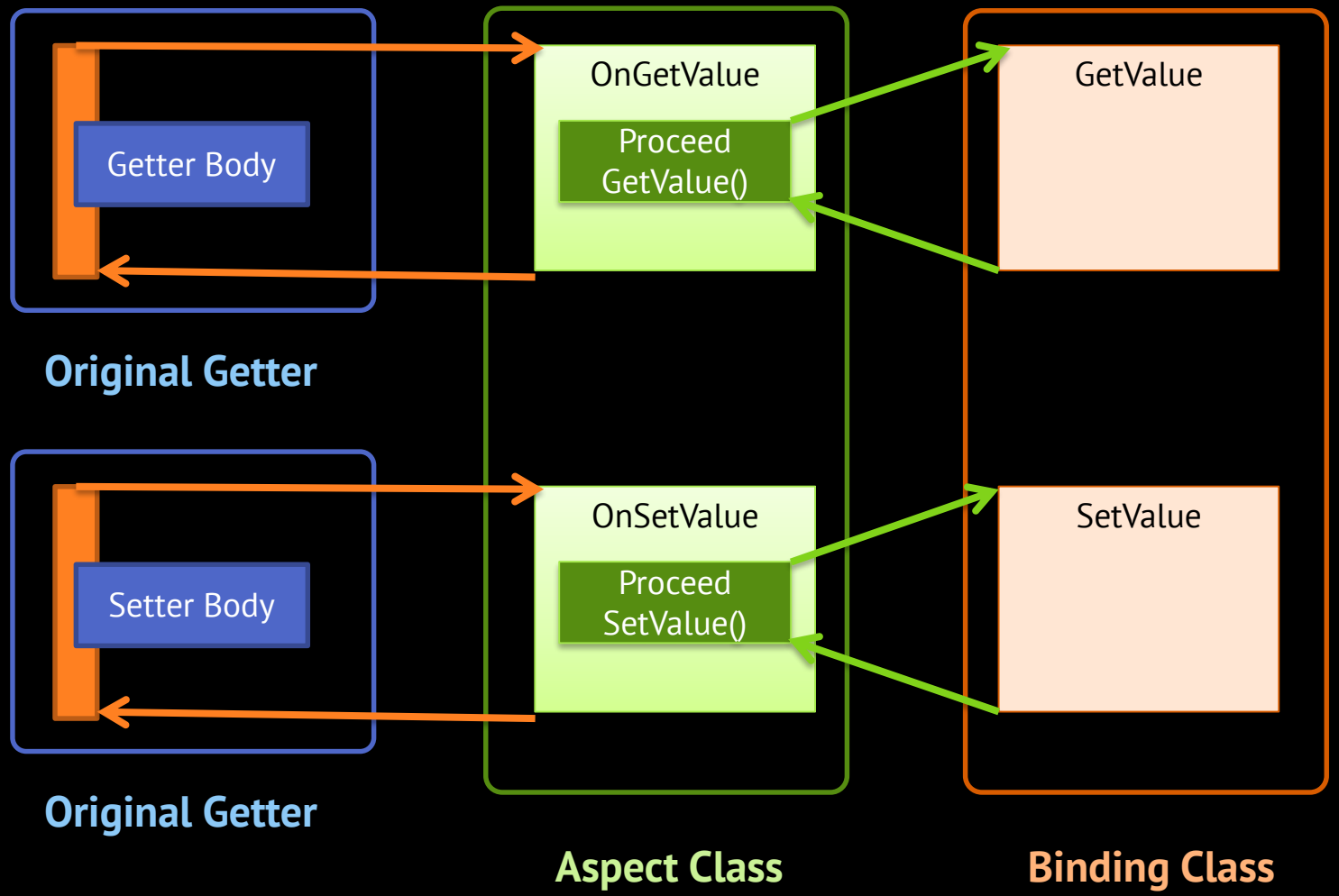
# OnMethodBoundary vs MethodInterception



LAB 2

# Auto Retry

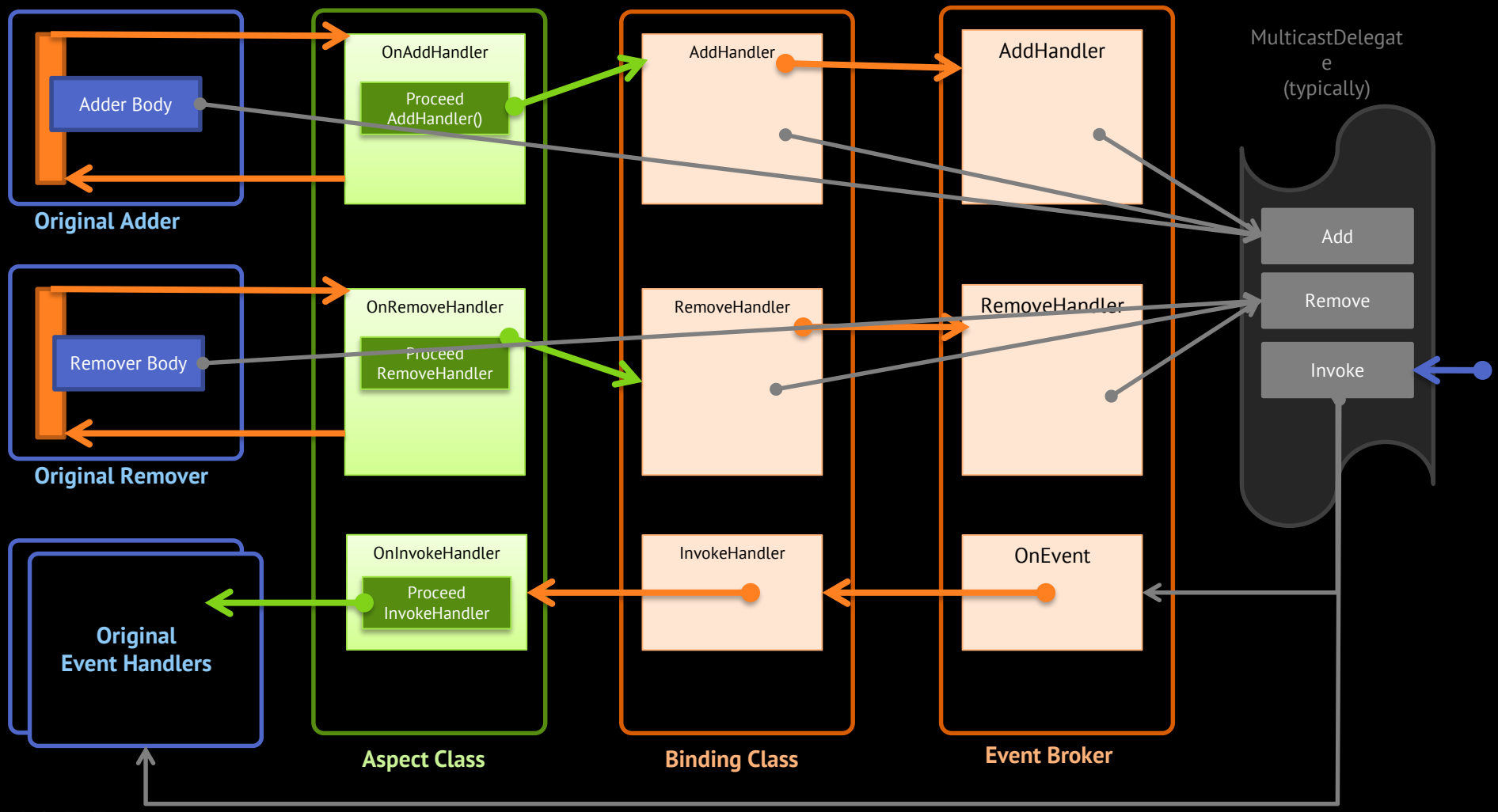
# LocationInterceptionAspect



LAB 3

# Storing fields in session state

# EventInterceptionAspect

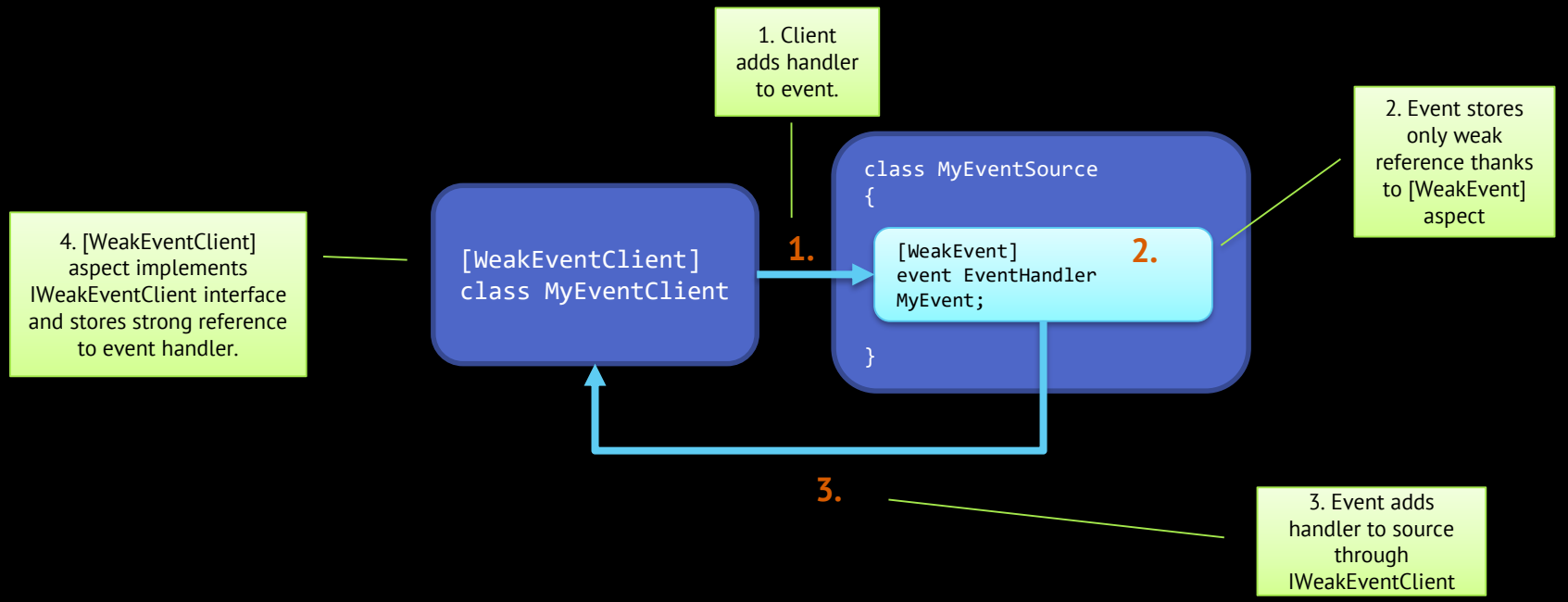


LAB 4

# Weak Event Pattern

# Example: Weak Event Pattern

- **Challenge**: Lifetime of event subscriptions controlled by the event *client*, not the event *source*.
- **Solution**: 2 aspects, 1 validation



# Applying Aspects to Code

## Applying Aspects to Code

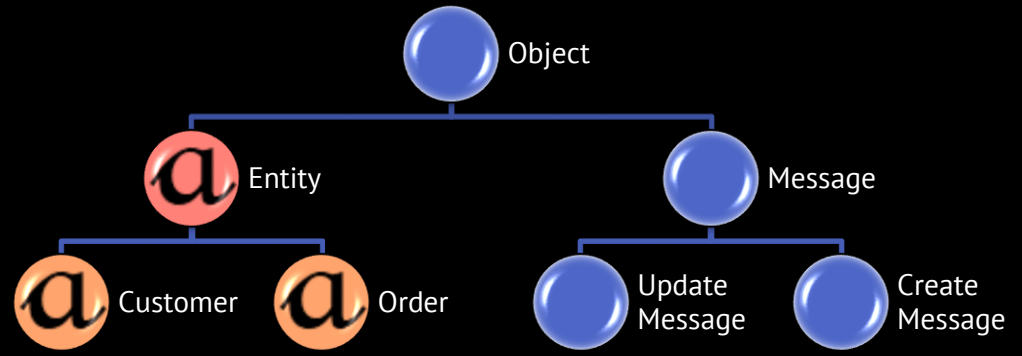
# Attribute Multicasting

- Class: MulticaseAttribute
- Default: Apply to all. Filter by:
  - Name (wildcard/regular expression)
  - Modifier (public/private/protected, virtual, …)
  - Type of elements of code

```
[ExceptionDialog(AttributeTargetMembers = "On*Click")]
public partial class ContactControl : UserControl
{
```

# Applying Aspects to Code
# Attribute Inheritance

- Interfaces
- Classes
- Virtual Methods
- Assemblies (!)




```
[Serializable]
[AspectRoleDependency(AspectDependencyAction.Order, AspectDependencyPosition.After, StandardRoles.DataBinding)]
[MulticastAttributeUsage(MulticastTargets.Class,Inheritance = MulticastInheritance.Strict)]
public sealed class UndoableAttribute : InstanceLevelAspect
{
}
```

- or -

```
[Undoable(AttributeInheritance = MulticastInheritance.Strict)]
public abstract class Entity
{
}
```

LAB 5

# Multicasting

# Applying Aspects to Code
# Aspect Provider

- Add aspects dynamically
- Arbitrary complexity (read external files, …)
- Create object graphs of aspect instances

```
class CustomAspect : TypeLevelAspect, IAspectProvider
{
    public IEnumerable<AspectInstance> ProvideAspects(object targetElement)
    {
        Type targetType = (Type)targetElement;

        return from method in targetType.GetMethods(BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public)
               where method.IsDefined(typeof(ObfuscationAttribute), true)
               select new AspectInstance(method, new CheckLicenseAspect());

    }
}
```

LAB 7

# Automatic WCF Data Contracts

# There is *much* more…

- Aspect Framework
  - Robust Aspect Composition
  - Composite Aspects
- Architecture Framework
- Patterns Libraries
  - Model
  - Threading
  - Diagnostics

# Let's Get Started!

- Go to www.postsharp.net/download.
  There's a free trial and a free Express edition.
- Read case studies and testimonials
  on www.postsharp.net/customers.
- Contact us
  at www.postsharp.net/support
  with any questions.

Get a FREE license of PostSharp Ultimate

**http://bit.ly/TODO-postsharp**

# Summary



- Compilers must do more to raise abstraction level
- PostSharp is showing the way.