Design Patterns

Ruben Leivas Ledo (IT-IS)
Brice Copy (IT-AIS)

CERN – Geneva (CH)

Introduction

- About Patterns
  - The idea of patterns
  - What is a Pattern?
  - Pattern Definitions
  - Why Patterns?
  - Patterns Elements and Forms
    - Canonical Pattern Form
    - GoF Pattern Form
    - Comparison

The Idea of Patterns

- *Designing Object Oriented SW is HARD but, making it reusable is even HARDER!*

Erich Gamma

- *Unfortunately we live in a world where is "basic" create reusable applications*

The Idea of Patterns

- How to become a “Master of Chess”
  - Learning the rules.
    - Name of the figures, allowed movements, geometry and table chess orientation.
  - Learning the principles
    - Value of the figures, strategic movements
  - BUT….
    - Being as good as Kasparov means studying, analyzing, memorized and constantly applied the matches of other Masters
  - There are hundreds of this matches

The Idea of Patterns

- How to become a SW Master
  - Learning the rules.
    - Algorithms, data structures, programming languages, etc.
  - Learning the principles
    - Structural programming, Modular programming, Object Oriented, etc.
  - BUT….
    - Being as good as Kasparov means studying, analyzing, memorized and constantly applied the “solutions” of other Masters
  - There are hundreds of these solutions (~patterns)

The Idea of Patterns

- *Each pattern describes a problem that happens several times in our environment, offering for it a solution in a way that it can be applied one million times without being the same twice.*
   - Christopher Alexander (1977)

Patterns

- What is a Pattern?
  - A Solution for a problem in a particular context.
  - Recurrent (applied to other situations within the same context)
  - Learning tool
  - With a Name
    - Identifies it as unique.
    - Common for the users community. (SIMBA)

Motivation of Patterns

- Capture the experience of the experts and make them accessible to the “mortals”
- Help the SW engineers and developers to understand a system when this is documented with the patters which is using
- Help for the redesign of a system even if it was not assumed originally with them
- Reusability
  - A framework can support the code reusability

So… Why Patterns?

- Do you need more hints?
- *Designing Object Oriented SW is HARD but, making it reusable is even HARDER!*
  - *Why not gather and document solutions that have worked in the past for similar problems applied in the same context?*
  - Common tool to describe, identify and solve recurrent problems that allows a designer to be more productive
  - And the resulting designs to be more flexible and reusable

Types of Software Patterns

- Riehle & Zullighoven *(Understanding and Using Patterns in SW development)*
- *Conceptual Pattern*
  - *Whose form is described by means of terms and concepts from the application domain.*
- *Design Pattern*
  - *Whose form is described by means of SW design constructs (objects, classes, inheritance, etc.)*
- *Programming Pattern*
  - *Whose form is described by means of programming language constructs*

Gang Of Four

- There are several Design Patterns Catalogue
- Most of the Designers follow the book Design Patterns: Elements of Reusable Object Oriented Software
  - E. Gamma, R. Helm, R. Johnson, J. Vlissides.

Classification of Design Patterns

- Purpose (what a pattern does)
  - Creational Patterns
    - Concern the process of Object Creation
  - Structural Patterns
    - Deal with de Composition of Classes and Objects
  - Behavioral Patterns
    - Deal with the Interaction of Classes and Objects

- Scope – what the pattern applies to
  - Class Patterns
    - Class, Subclass relationships
    - Involve Inheritance reuse
  - Object Patters
    - Objects relationships
    - Involve Composition reuse

Essential Elements of Design Pattern

- Pattern Name
  - Having a concise, meaningful name improves communication between developers
- Problem
  - Context where we would use this pattern
  - Conditions that must be met before this pattern should be used

Essential Elements of Design Pattern

- Solution
  - A description of the elements that make up the design pattern
  - Relationships, responsibilities and collaborations
  - Not a concrete design or implementation. Abstract
- Consequences
  - Pros and cons of using the pattern
  - Includes impacts of reusability, portability…

Pattern Template

- Pattern Name and Classification
- Intent
  - What the pattern does
- Also Known As
  - Other names for the pattern
- Motivation
  - A scenario that illustrates where the pattern would be useful
- Applicability
  - Situations where the pattern can be used

Pattern Template - II

- Structure
  - Graphical representation of the pattern
- Participants
  - The classes & objects participating in the pattern
- Collaborations
  - How to do the participants interact to carry out their responsibilities?
- Consequences
- Implementations
  - Hints and Techniques for implementing it

Pattern Template - III

- Sample Code
  - Code fragments for a Sample Implementation
- Known Uses
  - Examples of the pattern in real systems
- Related Patterns
  - Other patterns closely related to the patterns

Pattern Groups (GoF)

Let’s go to the kernel !!

- Taxonomy of Patterns
  - Creational Patterns
    - They abstract the process of instances creation
  - Structural Patterns
    - How objects and classes are used in order to get bigger structures
  - Behavioral Patterns
    - Characterize the ways in which classes or objects interact and distribute responsibilities

Creational Patterns

- Deal with the best way to create instances of objects

```
Listbox list = new Listbox()
```

- Our program should not depend on how the objects are created
- The exact nature of the object created could vary with the needs of the program
  - Work with a special “creator” which abstracts the creation process

Creational Patterns (II)

- Factory Method
  - Simple decision making class that returns one of several possible subclasses of an abstract base class depending on the data we provided
- Abstract Factory Method
  - Interface to create and return one of several families of related objects
- Builder Pattern
  - Separates the construction of a complex object from its representation
- Prototype Pattern
  - Clones an instantiated class to make new instances rather than creating new instances
- Singleton Pattern
  - Class of which there can be no more than one instance. It provides single global point of access to that instance

Structural Patterns

- Describe how classes & objects can be combined to form larger structures
  - *Class Patterns: How inheritance can be used to provide more useful program interfaces*
  - *Object Patterns: How objects can be composed into larger structures (objects)*

Structural Patterns II

- Adapter
  - Match interfaces of different classes
- Bridge
  - Separates an object’s interface from its implementation
- Composite
  - A tree structure of simple and composite objects
- Decorator
  - Add responsibilities to objects dynamically
- Façade
  - A single class that represents an entire subsystem
- Flyweight
  - A fine-grained instance used for efficient sharing
- Proxy
  - An object representing another object

Behavioral Patterns

- Concerned with communication between objects
- It’s easy for an unique client to use one abstraction
- Nevertheless, it’s possible that the client may need multiple abstractions
- …and may be it does not know before using them how many and what!
  - This kind of Patters (observer, blackboard, mediator) will allow this communication

Behavioral Patterns

- Chain of Responsibility
  - A way of passing a request between a chain of objects
- Command
  - Encapsulate a command request as an object
- Interpreter
  - A way to include language elements in a program
- Iterator
  - Sequentially access the elements of a collection
- Mediator
  - Defines simplified communication between classes
- Memento
  - Capture and restore an object's internal state

Behavioral Patterns III

- Observer
  - A way of notifying change to a number of classes
- State
  - Alter an object's behavior when its state changes
- Strategy
  - Encapsulates an algorithm inside a class
- Template
  - Defer the exact steps of an algorithm to a subclass
- Visitor
  - Defines a new operation to a class without change

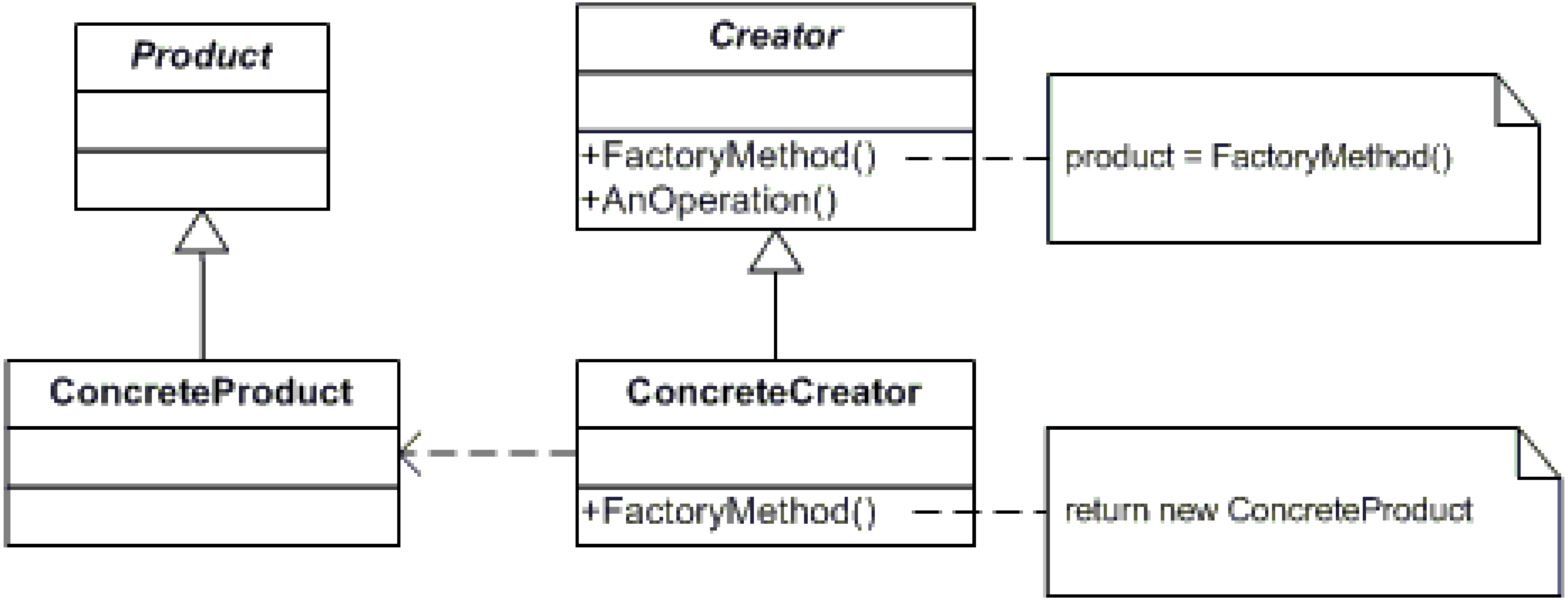
Examples applied to real life

Creational Pattern Example

- Factory
  - Define an interface for creating an object, but let subclasses decide which class to instantiate.
  - Factory Method lets a class defer instantiation to subclasses.
- Participants
  - **Product (Page)**
    - defines the interface of objects the factory method creates
  - **ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)**
    - implements the Product interface
  - **Creator (Document)**
    - declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
    - may call the factory method to create a Product object.
  - **ConcreteCreator (Report, Resume)**
    - overrides the factory method to return an instance of a ConcreteProduct.

Creational Pattern Examples

- UML Diagram



Sample Code (Factory)

-
  ```
  // Factory Method pattern –

  using System;
  using System.Collections;

  // "Product"

  abstract class Product
  {
  }

  // "ConcreteProductA"

  class ConcreteProductA :
  Product
  {
  }

  // "ConcreteProductB"

  class ConcreteProductB :
  Product
  {  }
  ```

-
  ```
  // "Creator"

  abstract class Creator
  {
    // Methods
    abstract public Product
  FactoryMethod();
  }

  // "ConcreteCreatorA"

  class ConcreteCreatorA :
  Creator
  {
    // Methods
    override public Product
  FactoryMethod()
    {
      return new
  ConcreteProductA();
    }
  }
  ```

Sample Code (Factory)

-
  ```
  // "ConcreteCreatorB"

  class ConcreteCreatorB :
  Creator
  {
    // Methods
    override public Product
  FactoryMethod()
    {
      return new
  ConcreteProductB();
    }
  }
  ```

-
  ```
  class Client
  {
    public static void Main(
  string[] args )
    {

      // FactoryMethod returns
  ProductA
      Creator c = new
  ConcreteCreatorA();
      Product p =
  c.FactoryMethod();
      Console.WriteLine(
  "Created {0}", p );

      // FactoryMethod returns
  ProductB
      c = new
  ConcreteCreatorB();
      p = c.FactoryMethod();
      Console.WriteLine(
  "Created {0}", p );
  ```

Sample Code (Factory)

-
  ```
  using System;
  using System.Collections;

  // "Product"

  abstract class Page
  {
  }

  // "ConcreteProduct"

  class SkillsPage : Page
  {
  }

  // "ConcreteProduct"

  class EducationPage : Page
  {
  }

  // "ConcreteProduct"

  class ExperiencePage : Page
  {
  }
  ```

-
  ```
  // "ConcreteProduct"

  class IntroductionPage : Page
  {
  }
  // "ConcreteProduct"

  class ResultsPage : Page
  {
  }

  // "ConcreteProduct"

  class ConclusionPage : Page
  {
  }

  // "ConcreteProduct"

  class SummaryPage : Page
  {
  }
  ```

Sample Code (Factory)

-
  ```
  // "Creator"

  abstract class Document
  {
    // Fields
    protected ArrayList pages = new ArrayList();

    // Constructor
    public Document()
    {
      this.CreatePages();
    }

    // Properties
    public ArrayList Pages
    {
      get{ return pages; }
    }

    // Factory Method
    abstract public void CreatePages();
  }
  ```

Sample Code (Factory)

-
  ```
  // "ConcreteCreator"

  class Resume : Document
  {
    // Factory Method
  ```

-
  ```
    override public void
  CreatePages()
    {
      pages.Add( new
  SkillsPage() );
      pages.Add( new
  EducationPage() );
      pages.Add( new
  ExperiencePage() );
    }
  }
  ```

-
  ```
  // "ConcreteCreator"

  class Report : Document
  {
    // Factory Method
  ```

-
  ```
    override public void
  CreatePages()
    {
      pages.Add( new
  IntroductionPage() );
      pages.Add( new ResultsPage()
  );
      pages.Add( new
  ConclusionPage() );
      pages.Add( new SummaryPage()
  );
      pages.Add( new
  BibliographyPage() );
    }
  }
  ```

Sample Code (Factory)

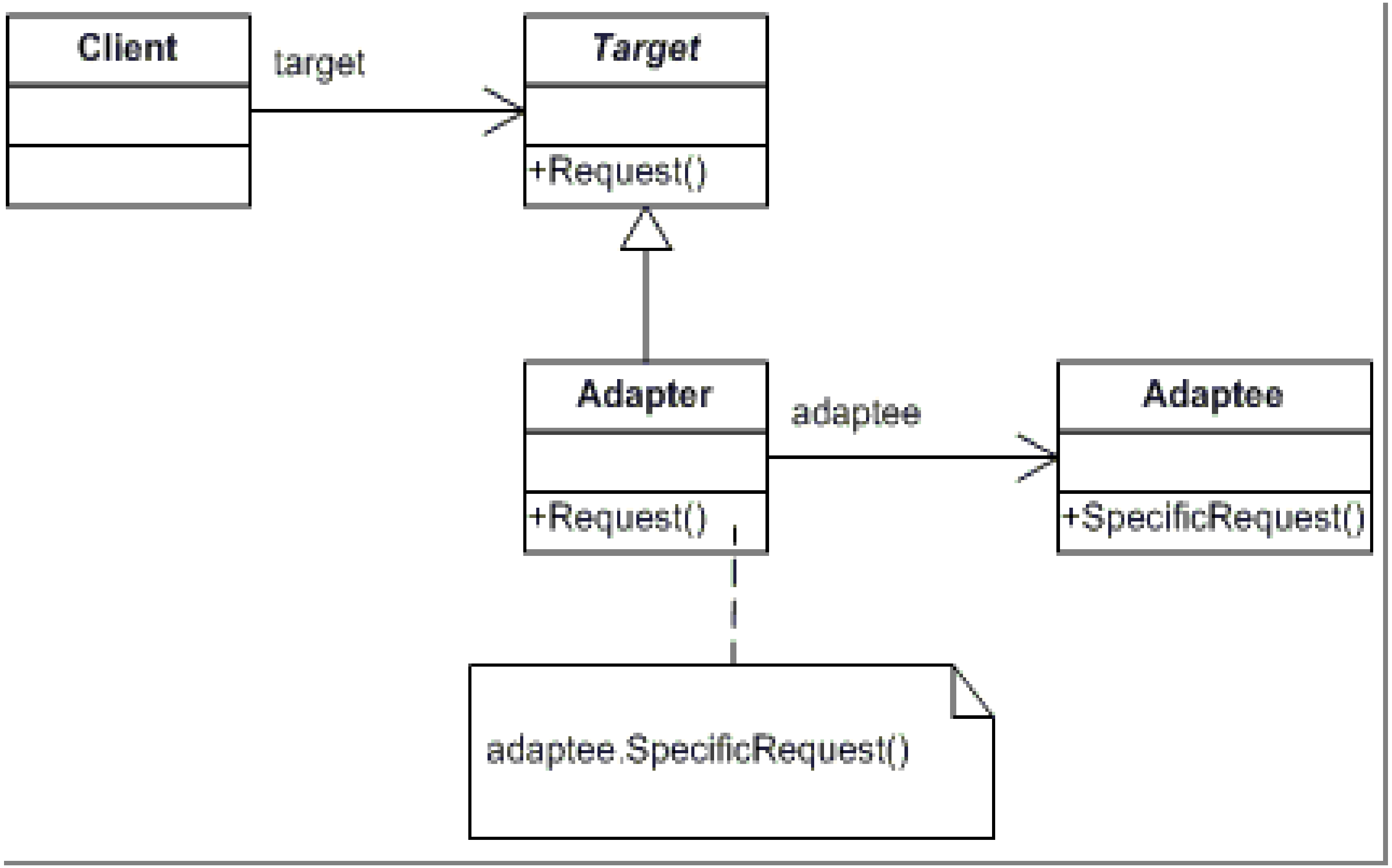
```
/// <summary>
///  FactoryMethodApp test
/// </summary>
class FactoryMethodApp
{
  public static void Main( string[] args )
  {
    Document[] docs = new Document[ 2 ];

    // Note: constructors call Factory Method
    docs[0] = new Resume();
    docs[1] = new Report();

    // Display document pages
    foreach( Document document in docs )
    {
      Console.WriteLine( "\n" + document + " ------- " );
      foreach( Page page in document.Pages )
        Console.WriteLine( " " + page );
    }
  }
}
```

Structural Pattern Example

- Adapter
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces
- **Participants**
  - **Target (ChemicalCompound)**
    - defines the domain-specific interface that Client uses.
  - **Adapter (Compound)**
    - adapts the interface Adaptee to the Target interface.
  - **Adaptee (ChemicalDatabank)**
    - defines an existing interface that needs adapting.
  - **Client (AdapterApp)**
    - collaborates with objects conforming to the Target interface.

Sample Code (Adapter)

- UML Diagram



Sample Code (Adapter)

-
  ```
  using System;

  // "Target"

  class ChemicalCompound
  {
    // Fields
    protected string name;
    protected float boilingPoint;
    protected float meltingPoint;
    protected double
       molecularWeight;
    protected string
       molecularFormula;

    // Constructor
     public ChemicalCompound
       ( string name )
     {
      this.name = name;
     }
  ```

-
  ```
    // Properties
     public float BoilingPoint
     {
       get{ return boilingPoint; }
     }

     public float MeltingPoint
     {
       get{ return meltingPoint; }
     }

     public double MolecularWeight
     {
       get{ return molecularWeight;
  }
     }

     public string MolecularFormula
     {
       get{ return
  molecularFormula; }
     }
  }
  ```

Sample Code (Adapter)

```
// "Adapter"

class Compound : ChemicalCompound
{
  // Fields
  private ChemicalDatabank bank;

  // Constructors
  public Compound( string name ) : base( name )
  {
    // Adaptee
    bank = new ChemicalDatabank();
    // Adaptee request methods
    boilingPoint = bank.GetCriticalPoint( name, "B" );
    meltingPoint = bank.GetCriticalPoint( name, "M" );
    molecularWeight = bank.GetMolecularWeight( name );
    molecularFormula = bank.GetMolecularStructure( name );
  }

  // Methods
  public void Display()
  {
    Console.WriteLine("\nCompound: {0} ------ ",name );
    Console.WriteLine(" Formula: {0}",MolecularFormula);
    Console.WriteLine(" Weight : {0}",MolecularWeight );
    Console.WriteLine(" Melting Pt: {0}",MeltingPoint );
    Console.WriteLine(" Boiling Pt: {0}",BoilingPoint );
  }
}
```

Sample Code (Adapter)

-
```
// "Adaptee"

class ChemicalDatabank
{
  // Methods -- the Databank 'legacy API'
  public float GetCriticalPoint( string
compound, string point )
  {
    float temperature = 0.0F;
    // Melting Point
    if( point == "M" )
    {
      switch( compound.ToLower() )
      {
        case "water": temperature = 0.0F;
break;
        case "benzene" : temperature =
5.5F; break;
        case "alcohol": temperature = -
114.1F; break;
      }
    }
    // Boiling Point
    else
    {
      switch( compound.ToLower() )
      {
        case "water": temperature =
100.0F;break;
        case "benzene" : temperature =
80.1F; break;
        case "alcohol": temperature =
78.3F; break;
      }
    }
```

```
public string GetMolecularStructure(
 string compound )
   {
     string structure = "";
     switch( compound.ToLower() )
     {
       case "water": structure =
 "H20"; break;
       case "benzene" : structure =
 "C6H6"; break;
       case "alcohol": structure =
 "C2H6O2"; break;
     }
     return structure;
   }

   public double GetMolecularWeight(
 string compound )
   {
     double weight = 0.0;
     switch( compound.ToLower() )
     {
       case "water": weight = 18.015;
 break;
       case "benzene" : weight =
 78.1134; break;
       case "alcohol": weight =
 46.0688; break;
     }
     return weight;
```

Sample Code (Adapter)

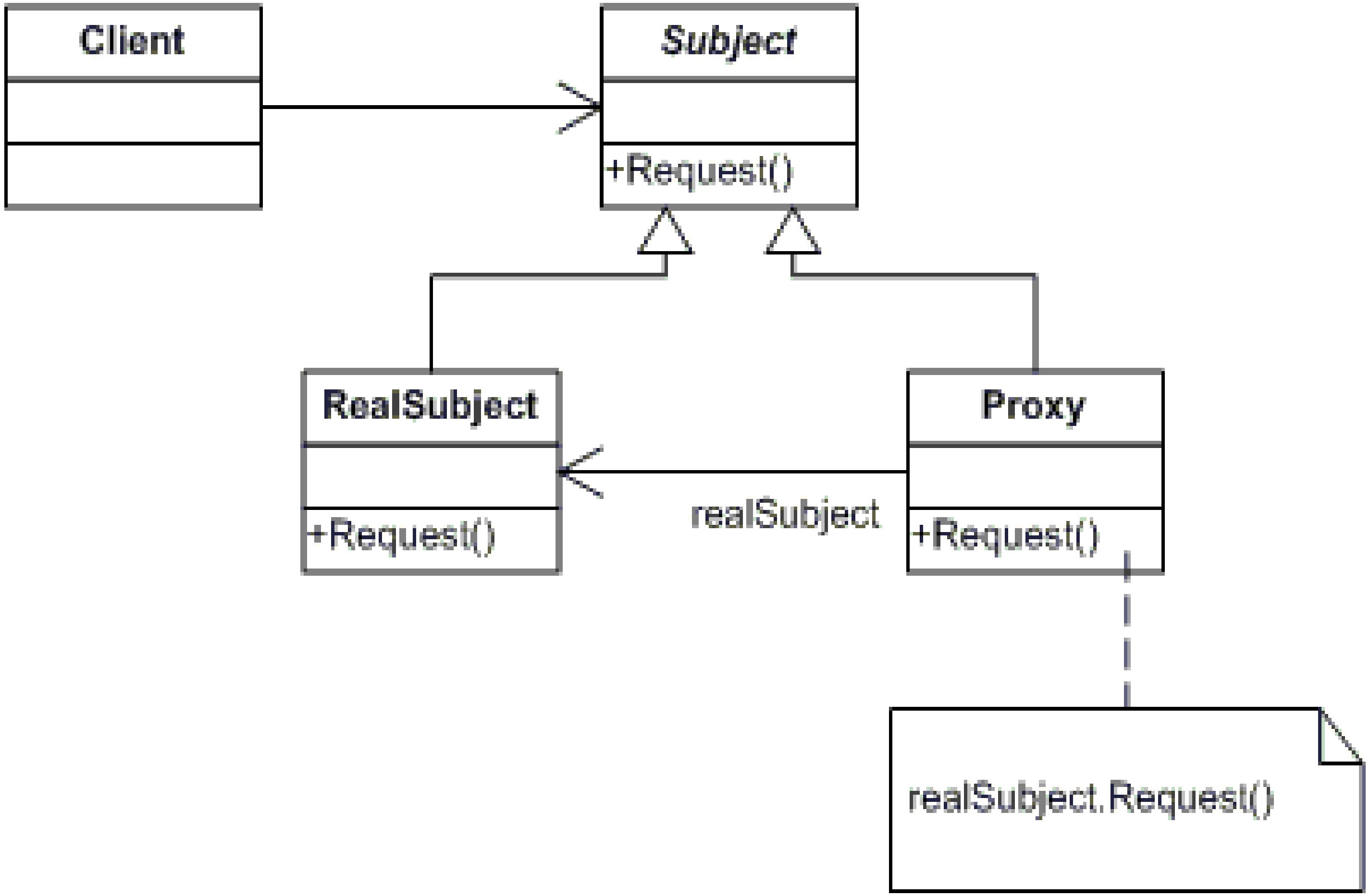
```
/// <summary>
/// AdapterApp test application
/// </summary>
public class AdapterApp
{
  public static void Main(string[] args)
  {
    // Retrieve and display water characteristics
    Compound water = new Compound( "Water" );
    water.Display();

    // Retrieve and display benzene characteristics
    Compound benzene = new Compound( "Benzene" );
    benzene.Display();

    // Retrieve and display alcohol characteristics
    Compound alcohol = new Compound( "Alcohol" );
    alcohol.Display();

  }
}
```

Behavioral Patterns Example

- Proxy
  - Provide a surrogate or placeholder for another object to control access to it.
- Participants
  - **Proxy (MathProxy)**
    - maintains a reference that lets the proxy access the real subject. Proxy may refer to a Subject if the RealSubject and Subject interfaces are the same.
    - provides an interface identical to Subject's so that a proxy can be substituted for for the real subject.
    - controls access to the real subject and may be responsible for creating and deleting it.
    - other responsibilites depend on the kind of proxy:
      - *remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
      - *virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Motivation caches the real images's extent.
      - *protection proxies* check that the caller has the access permissions required to perform a request.
  - **Subject (IMath)**
    - defines the common interface for RealSubject and Proxy so that a Proxy can be used anywhere a RealSubject is expected.
  - **RealSubject (Math)**
    - defines the real object that the proxy represents.

Sample Code (Proxy)

- UML Diagram



Sample Code (Proxy)

-
  ```
  using System;
  using System.Runtime.Remoting;

  // "Subject"

  public interface IMath
  {
    // Methods
    double Add( double x, double y );
    double Sub( double x, double y );
    double Mul( double x, double y );
    double Div( double x, double y );
  }

  // "RealSubject"

  class Math : MarshalByRefObject, IMath
  {
    // Methods
    public double Add( double x, double y )
   { return x + y; }
    public double Sub( double x, double y )
   { return x - y; }
    public double Mul( double x, double y )
   { return x * y; }
    public double Div( double x, double y )
   { return x / y; }
  }
  ```

-
  ```
  // Remote "Proxy Object"

  class MathProxy : IMath
  {
    // Fields
    Math math;
    // Constructors
    public MathProxy()
    {
      // Create Math instance in a different AppDomain
      AppDomain ad = System.AppDomain.CreateDomain(
                              "MathDomain",null, null );
      ObjectHandle o =
        ad.CreateInstance("Proxy_RealWorld", "Math", false,
        System.Reflection.BindingFlags.CreateInstance,
        null, null, null,null,null );
      math = (Math) o.Unwrap();
    }

    // Methods
    public double Add( double x, double y )
    {
      return math.Add(x,y);
    }
     public double Sub( double x, double y )
    {
      return math.Sub(x,y);
    }
    public double Mul( double x, double y )
    {
      return math.Mul(x,y);
    }
    public double Div( double x, double y )
    {
      return math.Div(x,y);
    }
  }
  ```

Sample Code (Proxy)

-
  ```
  public class ProxyApp
  {
    public static void Main( string[] args )
    {
      // Create math proxy
      MathProxy p = new MathProxy();

      // Do the math
      Console.WriteLine( "4 + 2 = {0}", p.Add( 4, 2 ) );
      Console.WriteLine( "4 - 2 = {0}", p.Sub( 4, 2 ) );
      Console.WriteLine( "4 * 2 = {0}", p.Mul( 4, 2 ) );
      Console.WriteLine( "4 / 2 = {0}", p.Div( 4, 2 ) );
    }
  }
  ```

Inversion of Control Pattern (IoC) *a.k.a. Dependency injection*

- Basically, a multi-purpose factory
- A 4GL replacement, exploits metadata from your code to provide a declarative environment
- Configuring instead of coding
  - Encapsulates complexity
  - Lets you expose only “key” parameters that you may change

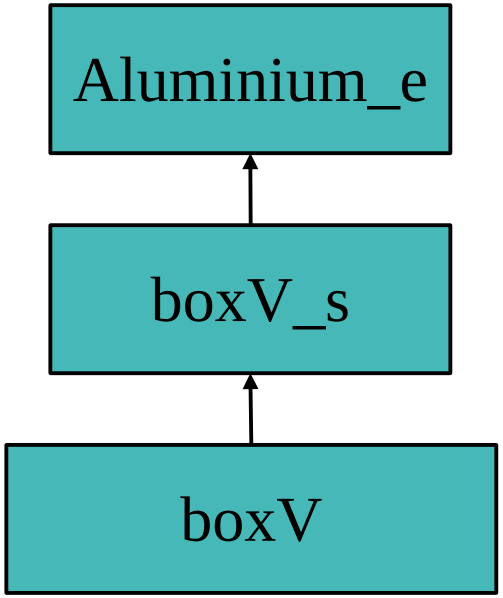
IoC : Advantages

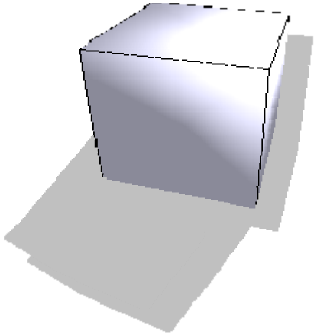
- Forces you to write clean code
  - No more complex dependencies
  - For complex objects, use factories
  - IoC will wire objects for you (matching object names to method parameters for instance)
  - Destruction of your objects is also handled
- Saves you from writing boring code
  - Calling new operators and getters/setters is both error prone and very simple anyway

IoC Configuration sample

Let us imagine a complex geometry setup :

- A material (aluminium)
- A volume (a cube)
- A physical volume (yes, that cube)





IoC configuration sample

in GDML

```
<element  name="Aluminium_e"
          Z=" 13.0000"  N=" 27" >
    <atom type="A" unit="g/mol"
          value=" 26.9815" />
</element>

<box  lunit="cm" aunit="degree"
      name="boxV_s"
      x="20.0000"  y="60.0000"
      z="50.0000" />

<volume name="boxV">
   <materialref ref="Aluminium_e"/>
   <solidref ref="boxV_s"/>
</volume>
```

IoC configuration sample

in IoC XML

```
<bean  name="Aluminium_e" class=”cern.mygdm.Material”>
 <property name=”Z” value=”13.0000”/>  /
 <property name=”N” value=”27”/>
 <property name=”A”>
   <bean class=”cern.mygdm.Atom”>
     <constructor-arg><value>A</value></constructor-arg>
     <constructor-arg><value>g/mol</value></constructor-arg>
     <constructor-arg><value>26.9815</value></constructor-arg>
   </bean>
 </property>
</bean>
<bean name="boxV_s" class=”cern.mygdm.Box”>
 <property name=”lunit” value=”cm”/>  /
 <property name=”aunit” value=”degree”/>
 <property name=”X” value=”20.0000”/>
 <property name=”Y” value=”60.0000”/>
 <property name=”Z” value=”50.0000”/>
<bean name="boxV" class=”cern.mygdm.PVolume”>
 <property name=”solidref”><bean name=”boxV_s”/></property>
 <property name=”materialref”><bean ref=”${material}”/></property>
</volume>
```

IoC configuration sample

Using your configuration

```
// Pseudo-code (only compiles in my head)
BeanFactory myFactory =
               IoCFactory.read(“myVolume.xml”);

myFactory.setProperty(“material”,”ALUMINIUM_e”);
cern.mygdm.PVolume myVolume = myFactory.get(“boxV”);

// ...or you could change it like so
// assuming you defined a “LEAD” material
myFactory.setProperty(“material”,”LEAD_e”);
cern.mygdm.PVolume myVolume = myFactory.get(“boxV”);
```

IoC configuration sample

What's in it for you ?

- It is more verbose but...
- Totally generic -> easy integration
- Replaces code by configuration
- Configurable (pre and post process)
- Can be nested with other configurations
- No specific XML format maintenance (even though they may be useful for conciseness)

IoC platforms

- Primarily Java, as it currently offers the richest reflection mechanism (including interceptors and runtime proxy generation)
- Your langage needs reflection some way or another
- .NET somewhat supports this, but development effort is slower at the moment

IoC frameworks

- Spring Framework
  - A simple yet powerful java IoC framework
  - A huge toolbox with very good default beans
  - With aspect oriented programming support
  - Comes with extensions for :
    - JDBC / ORM frameworks
    - Servlet API
    - JMS
    - Transaction management
    - Etc...
  - Spring.NET version – in the works

IoC frameworks (2)

- PICO container
  - A basic but lightweight IoC library
  - No built-in aspects support
- Apache Avalon's Fortress
- Castle for .NET (http://www.castleproject.org)

IoC Benefits

- Cleaner code, heavy usage of interfaces
- Lets you encapsulate complexity and make it configurable (mini pluggable blackbox)
- Encourages teamwork by sharing object models, not lines of code or libraries
- ... Like for all patterns, those advantages are not obvious until you try it

Conclusion

- Software Design Patterns are NOT
  - Restricted to Object Oriented designs
  - Untested ideas/theories/inventions
  - Solutions that have worked only once
  - Abstract Principles
  - Universally applicable for every context
  - A “silver bullet” or a panacea

Conclusion

- Software Design Patterns are
  - Recurring solutions to common design problems
  - Concrete solutions to real world problems
  - Context Dependants
  - A literary form for documenting best practices
  - Shared for the community
  - Excessively hyped!!!!!