Design Patterns

Overview
- What is a Design Pattern?
- Creational Patterns
- Structural Patterns
- Behavioral Patterns

References
- Erich Gamma et al., “Design Patterns – Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995
- Frank Buschmann et al., “Pattern-Oriented Software Architecture - A System of Patterns”, Wiley, 1996
- Steven John Metsker, “Design Patterns Java™ Workbook”, Addison-Wesley, 2002
What Is a Design Pattern?

Christopher Alexander says:

“Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing the same thing twice.”
A pattern has four essential elements:

- The pattern name that we use to describe a design problem,
- The problem that describes when to apply the pattern,
- The solution that describes the elements that make up the design, and
- The consequences that are the results and trade-offs of applying the pattern.
Design Patterns Are Not About Design

- Design patterns are not about designs such as linked lists and hash tables that can be encoded in classes and reused as is.

- Design patterns are not complex, domain-specific designs for an entire application or subsystem.

- Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.
### Describing Design Patterns

A common way to describe a design pattern is the use of the following template:

- **Pattern Name and Classification**
- **Intent**
- **Also Known As**
- **Motivation (Problem, Context)**
- **Applicability (Solution)**
- **Structure (a detailed specification of structural aspects)**
- **Participants, Collaborations (Dynamics)**
- **Implementation**
- **Example**
- **Known Uses**
- **Consequences**
- **Known Uses**
Creational Patterns

Creational patterns abstract the instantiation process. They help to make a system independent of how its objects are created, composed, and represented.

- Creational patterns for classes use inheritance to vary the class that is instantiated.
- Creational patterns for objects delegate instantiation to another object.
Why Creational Patterns?

- Creational patterns encapsulate knowledge about which concrete classes the system uses.

- Creational patterns hide how instances of these classes are created and put together.

- Creational patterns help to shift away from hard-coding a fixed set of behaviors towards defining a smaller set of fundamental behaviors that can be composed into any number of a more complex one.
Singleton

**Intent:**
- Ensure a class has only one instance, and provide a global point of access to it.

**Collaborations:**
- Clients access a singleton instance solely through singleton’s instance operation.
Structure of Singleton

Singleton

- static Instance()
- SingletonOperation()
- GetSingletonData()
- static uniqueInstances
- singletonData

return uniqueInstance
Implementation

class Singleton
{
    // private static instance of a Singleton
    private static Singleton fInstance;

    protected Singleton() { ... } // protected constructor

    // public static getter to retrieve an instance of a singleton
    // Gamma et al. recommend the use of a public method
    public static Singleton Instance
    { get { if ( fInstance == null )
        fInstance = new Singleton();
        return fInstance; } } }

Test Singleton

Singleton aObj1 = Singleton.Instance; // Get first Singleton
Singleton aObj2 = Singleton.Instance; // Get second Singleton

// test that aObj1 and aObj2 are indeed identical
if ( aObj1.Equals( aObj2 ) )
    Console.WriteLine( "The objects are identical copies!" );
else
    Console.WriteLine( "OOPS! The objects are not identical copies!" );

// test that aObj1 and aObj2 are the same
if ( aObj1 == aObj2 )
    Console.WriteLine( "The objects are the same!" );
else
    Console.WriteLine( "OOPS! The objects are not the same!" );
Output

The objects are identical copies!
The objects are the same!
Prototype

**Intent:**
- Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype.

**Collaborations:**
- A client asks a prototype to clone itself.
Structure of Prototype

Client
  Operation()

Prototype
  Clone()

p = prototype.Clone()

ConcretePrototype1
  Clone()
  return copy of self

ConcretePrototype2
  Clone()
  return copy of self

Com S 430
ICloneable Interface

Supports cloning, which creates a new instance of a class with the same value as an existing instance.

Definition:

```csharp
public interface ICloneable
{
    object Clone();
}
```
Implementation

class Prototype : ICloneable
{
    private string fValue; // some instance variable

    // Creates a new object that is a copy of the current instance.
    public object Clone() { return new Prototype( fValue );; }

    // Access is limited to the current project or derived types
    protected internal Prototype( string aValue ) { fValue = aValue; }

    public override bool Equals( Object aObject ) { … }
    public override int GetHashCode() { … }
}

Equals & GetHashCode

class Prototype : ICloneable {
    
    public override bool Equals( Object aObject )
    {
        // Check for null and compare run-time types.
        if ( aObject == null || GetType() != aObject.GetType() )
            return false;
        Prototype p = (Prototype)aObject;
        return fValue.Equals( p.fValue );
    }

    // If we override Equals, we need to override GetHashCode() as well.
    // We simply use the GetHashCode() method of String (see PS1).
    public override int GetHashCode() { return fValue.GetHashCode(); }
}
Prototype aObj1 = new Prototype( "A value" ); // Get first instance

// Get second instance using ICloneable interface
Prototype aObj2 = (Prototype)aObj1.Clone();

// test that aObj1 and aObj2 are identical copies
if ( aObj1.Equals( aObj2 ) )
    Console.WriteLine( "The objects are identical copies!" );
else
    Console.WriteLine( "OOPS! The objects are not identical copies!" );

// test that aObj1 and aObj2 are not identical
if ( aObj1 == aObj2 )
    Console.WriteLine( "OOPS! The objects are the same!" );
else
    Console.WriteLine( "The objects are not the same!" );
The objects are identical copies!
The objects are not the same!
**Factory Method**

**Intent:**
- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

**Collaborations:**
- Creator relies on its subclasses to define the factory method so that it returns an instance of the appropriate `ConcreteProduct`. 
Structure of Factory Method

Product

ConcreteProduct

Creator

FactoryMethod()
AnOperation()

ConcreteCreator

FactoryMethod()

product = FactoryMethod()

return new ConcreteProduct
A classical example of factory method is that of iterators.

An iterator provides sequentially access to elements of a collection. A concrete iterator() method isolates its caller from knowing which class to instantiate.
IEnumerable Interface

- Exposes the enumerator, which supports a simple iteration over a collection.

- Definition:

```csharp
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}
```
IEnumerator Interface

- Supports a simple iteration over a collection.

- Definition:

```csharp
public interface IEnumerator
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}
```
class Bag1 : IEnumerable
{
    private ArrayList fElements;

    public Bag1() { ... }

    public void Add( object aObject ) { ... }

    public void Remove( object aObject ) { ... }

    // Factory method: IEnumerator GetEnumerator()
    // We use the ArrayList's method
    public IEnumerator GetEnumerator() {
        return fElements.GetEnumerator();
    }
}
class Bag2 : IEnumerable
{
    // private fields to store elements (array semantics)
    private int fIndex;
    private object[] fElements;
...
    // Factory method: IEnumerator GetEnumerator()
    // We implement a new Enumerator
    public IEnumerator GetEnumerator()
    {
        return new Bag2Enumerator( fElements, fIndex );
    }

    // inner class Bag2Enumerator (ignores changes to the collection)
    class Bag2Enumerator : IEnumerator { ... }
}
**Class Bag2Enumerator**

class Bag2Enumerator : IEnumerator
{
    // private fields to store elements
    private int fNElements;
    private object[] fElements;
    private int fIndex;

    public Bag2Enumerator( object[] aElements, int aNElements ) { … } 

    public object Current { get { return fElements[fIndex]; } } 

    public bool MoveNext() { return ++fIndex < fNElements; } 

    public void Reset() { fIndex = 0; } 
}
Test Bag2

Bag2 lBag = new Bag2();

lBag.Add( 1 );
lBag.Add( "A string object" );
lBag.Add( 's' );
lBag.Add( "another string object" );

IEnumerator lEnum = lBag.GetEnumerator();
while( lEnum.MoveNext() )
    Console.WriteLine( "Found object "{0}".", lEnum.Current );
Output

Found object "1".
Found object "A string object".
Found object "s".
Found object "another string object".
Structural Patterns

- Structural patterns are concerned with how classes and objects are composed to form larger structures.
  - Structural *class* patterns use inheritance to compose interfaces or implementations.
  - Structural *object* patterns describe ways to compose objects to realize new functionality. The added flexibility of object composition comes from the ability to change the composition at runtime, which is impossible with static class composition.
Adapter

**Intent:**
- Convert the interface of a class into another interface clients expect. Adapter lets classes work together that could not otherwise because of incompatible interfaces.

**Collaborations:**
- Clients call operations on an Adapter instance. In turn, the adapter calls Adaptee operations that carry out the request.
Structure of a Class Adapter

Client → Target
Request()

Adapter
Request() → SpecificRequest()

Adaptee
SpecificRequest()
Structure of an Object Adapter

Client → Target

Target → Adapter

Adapter → Adaptee

Adaptee

SpecificRequest()
Class Adapter: Target

// IComparable defines a generalized comparison method that
// a value type or class implements to create a type-specific
// comparison method.
public interface ITower
{
    string Name { get; }
    void Add(IComparable aObject);
    IComparable Remove();
}

Problem:
We want to use System.Collections.Stack to implement the class Tower to solve the problem “Towers of Hanoi”.

Com S 430
public class Tower : Stack, ITower
{
    private string fName;

    public Tower( string aName ) : base() { fName = aName; }

    public string Name { get { return fName; } } // ITower.Name

    public void Add( IComparable aObject ) // ITower.Add
    {
        if ( (Count > 0) && (aObject.CompareTo( Peek() ) > 0) )
            throw new IllegalSizeException( ... );
        else
            Push( aObject ); } // SpecificRequest()

    public IComparable Remove() // ITower.Remove
    { return (IComparable)Pop(); } // SpecificRequest()
}
IComparable Example

```csharp
public class Disk : IComparable
{
    private int fSize;

    public Disk(int aSize) { fSize = aSize; }

    public int Size { get { return fSize; } }

    public int CompareTo(object aObject) // IComparable.CompareTo
    {
        if (aObject is Disk) // aObject must have the same type as this.
            return Size - ((Disk)aObject).Size;
        else
            throw new ArgumentException("aObject is not a Disk.");
    }
}
```
Object Adapter: Target

// IComparable defines a generalized comparison method that
// a value type or class implements to create a type-specific
// comparison method.
public interface ITower2
{
    string Name{ get; }
    int Count{ get; }
    void Add( IComparable aObject );
    IComparable Remove();
}

We will use a Stack instance variable and forward calls of Add and Remove to methods of Stack.
object Adapter: Adapter

```java
public class Tower : ITower2
{
    private string fName;
    private Stack fStack;  // delegate instance

    public Tower( string aName ) { fName = aName;
        fStack = new Stack(); }

    public int Count { get { return fStack.Count; } }  // ITower2.Count
    public string Name { get { return fName; } }  // ITower2.Name

    public void Add( IComparable aObject )  // ITower2.Add
    {   if ( (Count > 0) && (aObject.CompareTo( fStack.Peek() ) > 0) )
            throw new IllegalDiskSizeException( … );
        else fStack.Push( aObject ); }

    public IComparable Remove()  // ITower2.Remove
    { return (IComparable)fStack.Pop(); } }
```
Proxy

◆ Intent:
  - Provide a surrogate or placeholder for another object to control access to it.

◆ Collaborations:
  - Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.
Structure of Proxy

Client → Subject

Subject

Request()
...

RealSubject

Request()
...

Proxy

Request()

realSubject.Request()
{
    public SimpleService() { InitializeComponent(); }
    private void InitializeComponent() {} 

    public int Add( int x, int y ) { return x + y; } // [WebMethod]
    public int Sub( int x, int y ) { return x - y; } // [WebMethod]
    public int Mul( int x, int y ) { return x * y; } // [WebMethod]
    public int Div( int x, int y ) // [WebMethod]
    {
        if ( y == 0 ) throw new DivideByZeroException( ... );
        else return x / y;
    }
}
Running the Web Service

SimpleService

Simple Calculator Service

The following operations are supported. For a formal definition, please review the Service Description.

- **Mul**
  Multiplies $x$ and $y$

- **Sub**
  Subtracts $y$ from $x$

- **Div**
  Divides $x$ by $y$

- **Add**
  Adds $x$ and $y$
../Service1.asmx/Sub?x=3&y=4

SimpleService

Click here for a complete list of operations.

Sub
Subtracts y from x

Test
To test the operation using the HTTP GET protocol, click the 'Invoke' button.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>x:</td>
<td>3</td>
</tr>
<tr>
<td>y:</td>
<td>4</td>
</tr>
</tbody>
</table>

<?xml version="1.0" encoding="utf-8" ?>
<int xmlns="http://localhost/">-1</int>
Building a Proxy Using \texttt{wsdl.exe}

- The \texttt{wsdl.exe} command line tool generates a code file that represents the proxy to a remote Web Service.

Example:

```
wsdl /out:CalculatorProxy.cs
http://localhost/CalculatorService/Service1.asmx?WSDL
```

- “http://localhost/CalculatorService/Service1.asmx?WSDL” returns an XML file that contains a specification of the Web Service “CalculatorService” for the protocols SOAP, \texttt{HttpGet}, and \texttt{HttpPost}.
SoapHttpClientProtocol

<table>
<thead>
<tr>
<th>Inherited members</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>BeginInvoke()</td>
<td>Starts an asynchronous invocation of a method of a SOAP Web Service.</td>
</tr>
<tr>
<td>EndInvoke()</td>
<td>Ends an asynchronous invocation of a method of a SOAP Web Service.</td>
</tr>
<tr>
<td>Invoke()</td>
<td>Synchronously invokes a method of a SOAP Web Service.</td>
</tr>
<tr>
<td>Proxy</td>
<td>Gets or sets proxy information for making a Web Service request through a firewall.</td>
</tr>
<tr>
<td>Timeout</td>
<td>Gets or sets the timeout used for synchronous calls.</td>
</tr>
<tr>
<td>Url</td>
<td>Gets or sets the base URL to the server to use for requests.</td>
</tr>
<tr>
<td>UserAgent</td>
<td>Gets or sets the value for the user agent header sent with each request.</td>
</tr>
</tbody>
</table>

Com S 430
public class SimpleService :
{
    public SimpleService() { this.Url = "http://.../Service1.asmx"; } 

    public int Add(int x, int y) // synchronous call of Add
    {
        object[] results = this.Invoke("Add", new object[] { x, y});
        return ((int)(results[0]));
    }

    ...
}
Creating the Client

// Create a proxy
// Do not forget to add reference to System.Web.Services.dll!
SimpleService ws = new SimpleService();

Console.WriteLine("100 + 100 is \{0\}", ws.Add( 100, 100 ) );
try {
    ws.Div( 0, 0 );
} catch(SoapException e) {
    // DivideByZeroException will be the inner exception of SoapException
    Console.WriteLine( e.Message );
}
Output

Com S 430
Behavioral Patterns

Behavioral patterns are concerned with algorithms and the assignment of responsibilities between objects.
- Behavioral class patterns use inheritance to distribute behavior between classes.
- Behavioral object patterns use object composition rather than inheritance. Some describe how a group of peer objects cooperate to perform a task that no single object can carry out by itself.

The classic example of a behavioral pattern is Model-View-Controller (Observer), where all views of the model are notified whenever the model’s state changes.
Iterator

**Intent:**
- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

**Collaborations:**
- A ConcreteIterator keeps track of the current object in the aggregate and can compute the succeeding object in the traversal.
Structure of Iterator

Client

Aggregate

ConcreteAggregate

ConcreteIterator

Iterator

return new ConcreteIterator(this)
Observer

◆ Intent:
  ■ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

◆ Collaborations:
  ■ ConcreteSubject notifies its observers whenever a change occurs that could make its observer’s state inconsistent with its own.
Structure of Observer

Subject
- Attach(Observer)
- Detach(Observer)
- Notify()

ConcreteSubject
- GetState()
- SetState()
- subjectState

ConcreteObserver
- observerState
- observerState = subject.GetState()

return subjectState

For all o in observers
  o.Update()
IEnumerator Reviewed

- An enumerator remains valid as long as the collection remains unchanged.

- If changes are made to the collection, such as adding, modifying or deleting elements, the enumerator is irrecoverably invalidated and a call to MoveNext() or Reset() must throw an InvalidOperationException.
class Bag3 : IEnumerable {
    delegate void Notify();               // event type
    private event Notify fChanged;       // collection of observers
    ...
    public void Add( object aObject ) {
        Extend();
        fElements[fIndex++] = aObject;
        if ( fChanged != null )            // any observers registered
            fChanged();
    }
}

class Bag3Enumerator : IEnumerator { ... }
Class Bag3Enumerator

class Bag3Enumerator : IEnumerator
{
    ... 
    private bool fCollectionChanged; // new private variable

    public void CollectionChanged() // Update()
    {
        fCollectionChanged = true;
    }

    public object Current {
        get {
            if ( !fCollectionChanged )
                return fElements[fIndex];
            else
                throw new InvalidOperationException( ... );
        }
    }

    ... 
}
Bag3 IBag = new Bag3();

IBag.Add( 1 );
IBag.Add( "A string object" );
IBag.Add( 's' );
IBag.Add( "another string object" );

IEnumerator IEnum = IBag.GetEnumerator();
while( IEnum.MoveNext() )
    Console.WriteLine( "Found object "{0}".", IEnum.Current );
IBag.Add( "Change the collection" );
try { // now Reset() throws InvalidOperationException
    IEnum.Reset(); }
catch (InvalidOperationException e) {
    Console.WriteLine( "Enumerator exception raised ({0})!", e.Message ); }
Output

D:\Courses\Fall2002\Com5430\Patterns\Creational\Enumerator\bin\Debug\Enumerator.exe

- Found object "1".
- Found object "A string object".
- Found object "s".
- Found object "another string object".
- Enumerator exception raised (Reset()): Collection modified?!