جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila
University of Mohamed Boudiaf - M'sila

1985

Object Oriented Analysis and Design

Chapter 03: Design Patterns

Prof. Hichem Debbi

hichem.debbi@gmail.com

November 2, 2024

Reuse is a key of success

When we want to build a software, we want it to be built in reasonable time with less effort. Therefore, in software design, we always try to ***reuse*** such perfect solutions for common problems we face. As much as we reuse, we gain time and reduce effort.

Design Patterns

definition

Design patterns is a general reusable solution to a commonly occurring problem within a given context in software design.

- It is not a finished design that can be transformed directly into source or machine code
- It is a description or template for how to solve a problem that can be used in many different situations
- Design patterns are formalized best practices that the programmer can use to solve common problems when designing an application or system

Object-oriented design patterns

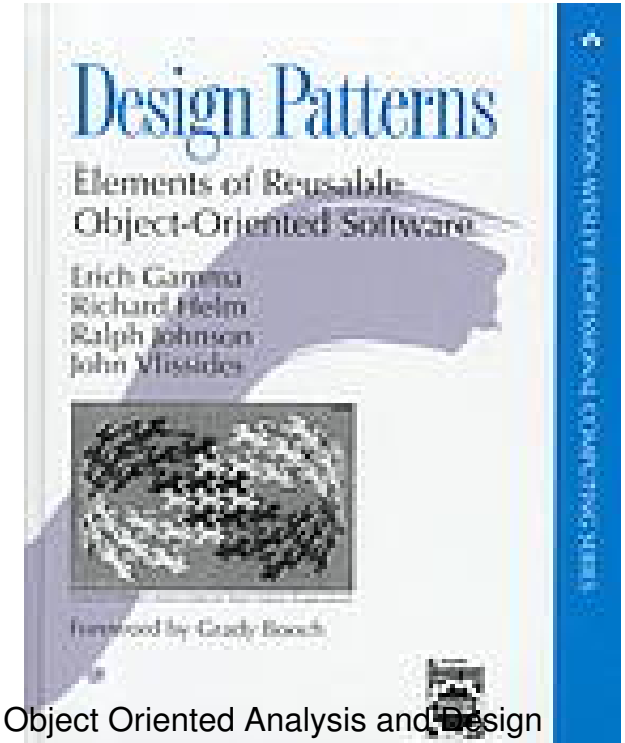
Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Object-oriented patterns are not necessarily suitable for non object oriented languages.

History of Design Patterns

In the late 1980s, when object-oriented programming was becoming more and more popular among researchers and practitioners, experts start to think about how patterns could be applied to software. The concentration was much more on Object-Oriented technology.

History of Design Patterns

In 1994, a book was published describing the first collection of software design patterns (23 patterns) known today as Gang of Four (GoF) patterns, with respect to the four authors of the book : Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Today, These 23 patterns are still considered fundamental. The code examples given in this book were introduced in C++ and Smaltalk.



Design Patterns

- Patterns are independent of programming language and application domain
- Patterns involve the use of UML diagrams such as class and sequence diagrams
- Patterns are also used widely to document a solution
- Experts should not try to describe a new pattern until they have demonstrated its effectiveness in several different applications
- Such patterns are applicable to specific areas more than other areas

A Pattern Template

- **Pattern Name:** A short name for the pattern
- **Classification:** *Creational*, *Structural* or *Behavioral*
- **Intent** : A short description summing up what the pattern is for
- **Uses:** Where the pattern has been applied in the real world
- **Structure:** One or more class diagrams and sequence diagrams that illustrate how the pattern works
- **Participants/Players:** Short descriptions of the objects involved and their responsibilities
- **Sample Code:** Full implementations of the pattern

There are seven patterns that make up the structural group, each with the role of building flexibility, longevity, and security into computer software. Structural patterns can be employed while a system is being designed, or later on during maintenance and extension. The names of these patterns are

- Decorator
- Proxy
- Bridge
- Composite
- Flyweight
- Adapter
- Façade

Decorator Pattern Role

The role of the Decorator pattern is to provide a way of attaching new state and behavior to an object dynamically. The object does not know it is being “decorated,” which makes this a useful pattern for evolving systems. A key implementation point in the Decorator pattern is that decorators both inherit the original class and contain an instantiation of it.

Decorator Pattern Illustration

Figure: Decorator pattern illustration—(a) plain photograph and (b) photograph with tags

Decorator Pattern Illustration

In the image B, we have Four objects:

- **Object 01:** The original photo as shown to the left
- **Object 02:** The image with a border
- **Object 03:** The image with "Food" tag
- **Object 04:** The image with "Yellow" tag

Decorator Pattern Illustration

- The original object is unaware of any decorations
- There is no one big feature-laden class with all the options in it
- The decorations are independent of each other

Decorator Pattern Design

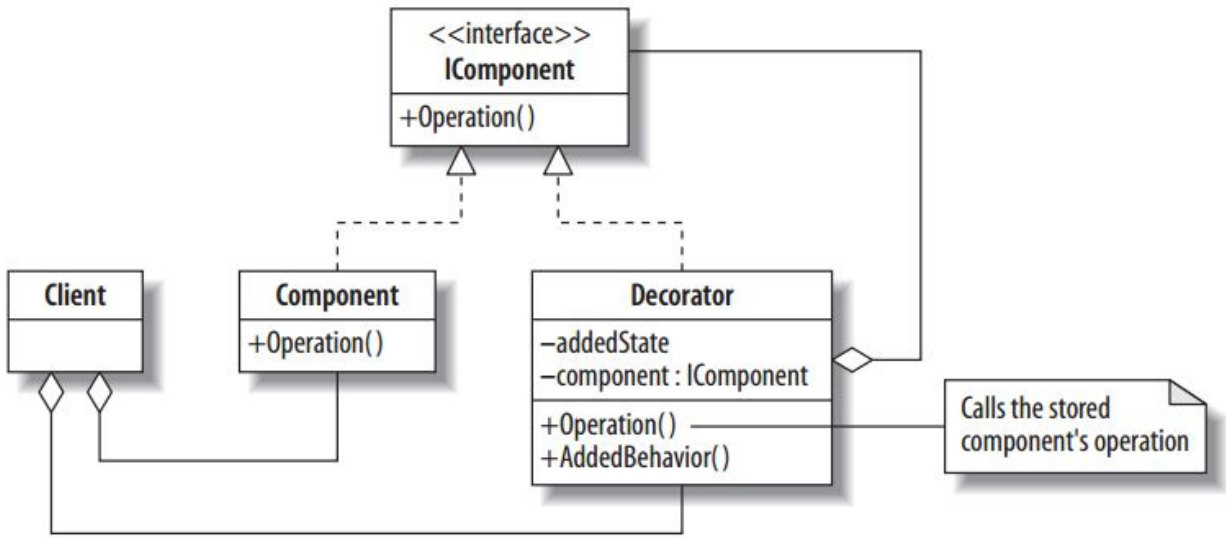


Figure: Decorator pattern UML diagram

Decorator Pattern Design: Participants

- **Component:** An original class of objects that can have operations added or modified
- **Operation:** An operation in IComponent objects that can be replaced
- **IComponent:** The interface that identifies the classes of objects that can be decorated
- **Decorator:** A class that conforms to the IComponent interface and adds state and/or behavior

Decorator Pattern Design: Relationships

Is a relationship

- Decorator realizes the IComponent interface (dotted arrow), so that Decorator objects can be used wherever IComponent objects are expected
- Is a relationship between Component and IComponent

Has a relationship

- Between Decorator and IComponent (open diamond on the Decorator) indicates that the Decorator instantiates one or more IComponent objects
- The Decorator uses the component attribute (of type IComponent) to invoke any replacement Operation

Decorator Pattern Design: Multiplicity

- **Multiple components:** Different components that conform to the interface can also be decorated
- **Multiple decorators:** Each decorator contains a component object, which might itself be a decorator
- **Multiple operations:** Decorator could perform many operations

Decorator Pattern Implementation


```
interface IComponent
{
        string Operation( );
}
class Component : IComponent
{
    public string Operation ( ) {
    return "I am walking ";
    }
}
class DecoratorA : IComponent
{
        IComponent component;
        public DecoratorA (IComponent c) {
        component = c;
        }
        public string Operation( ) {
        string s = component.Operation( );
        s += "and listening to Classic FM ";
        return s;
        }
}
```

Decorator Pattern Implementation


```
class DecoratorB : IComponent
{
        IComponent component;
        public string addedState = "past the Coffee Shop ";
        public DecoratorB (IComponent c) {
        component = c;
        }
        public string Operation ( ) {
        string s = component.Operation ( );
        s += "to school ";
        return s;
        }
        public string AddedBehavior( ) {
        return "and I bought a cappuccino ";
        }
}
```

Decorator Pattern Implementation


```
class Client
{
        static void Display(string s, IComponent c) {
            Console.WriteLine(s + c.Operation());
        }
        static void Main()
        {
            Console.WriteLine("Decorator Pattern\n");
            IComponent component = new Component();
            Display("1. Basic component: ", component);
            Display("2. A-decorated : ", new DecoratorA(component));
            Display("3. B-decorated : ", new DecoratorB(component));
            Display("4. B-A-decorated : ", new DecoratorB(
            new DecoratorA(component)));
            // Explicit DecoratorB
            DecoratorB b = new DecoratorB(new Component());
            Display("5. A-B-decorated : ", new DecoratorA(b));
            // Invoking its added state and added behavior
            Console.WriteLine("\t\t\t" + b.addedState + b.AddedBehavior());
            Console.ReadKey();
        }
}
```

Decorator Pattern Implementation


```
1. Basic component: I am walking
2. A-decorated : I am walking and listening to Classic FM
3. B-decorated : I am walking to school
4. B-A-decorated : I am walking and listening to Classic FM to school
5. A-B-decorated : I am walking to school and listening to Classic FM
                   past the Coffee Shop and I bought a cappuccino
```

Decorator Pattern USE

- Cab be very effective in graphical applications
- In web and mobile applications

Decorator Pattern USE

We can use Decorator pattern when we want to :

- Attach additional state or behavior to an object dynamically
- Make changes to some objects in a class without affecting others
- Avoid subclassing, because too many classes could result

Proxy Pattern Role

The Proxy pattern supports objects that control the creation of and access to other objects. The proxy is often a small (public) object that stands in for a more complex (private) object that is activated once certain circumstances are clear.

Proxy Pattern Illustration

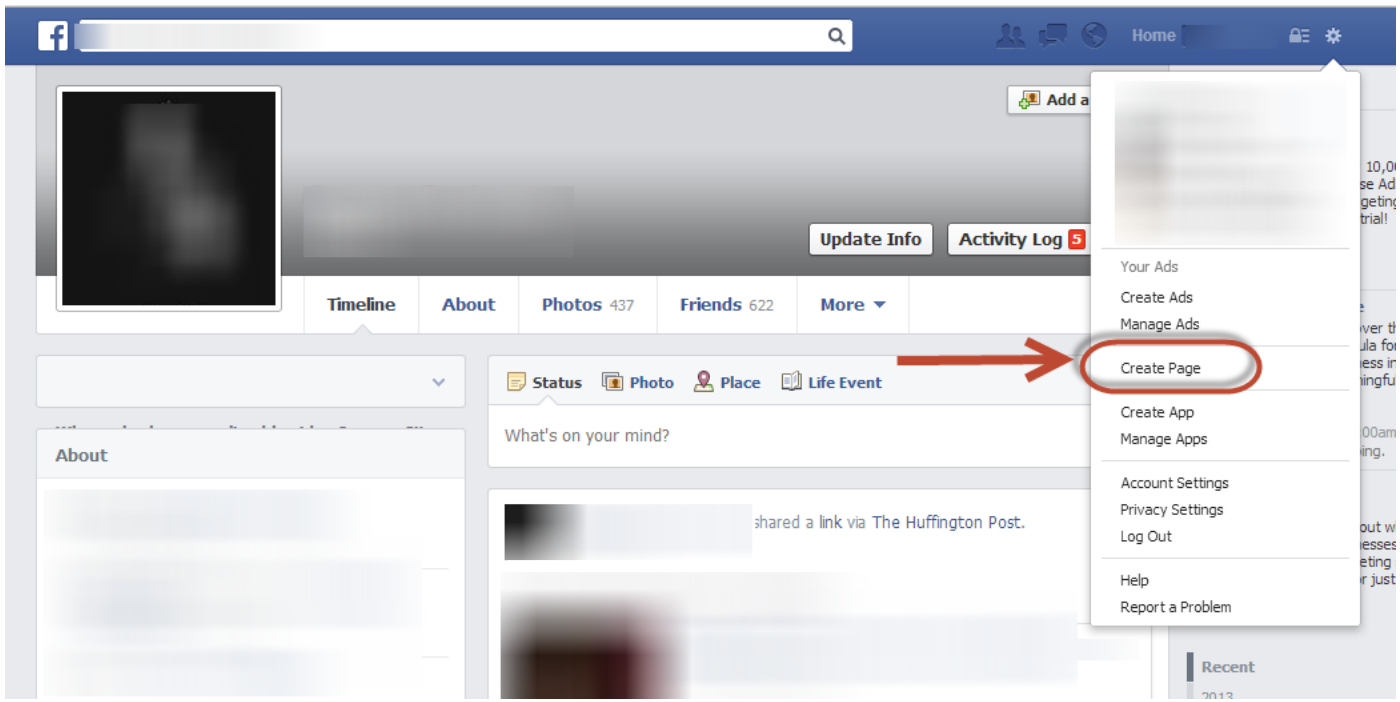


Figure: Proxy pattern illustration—a page in a social networking system

Proxy Pattern Design

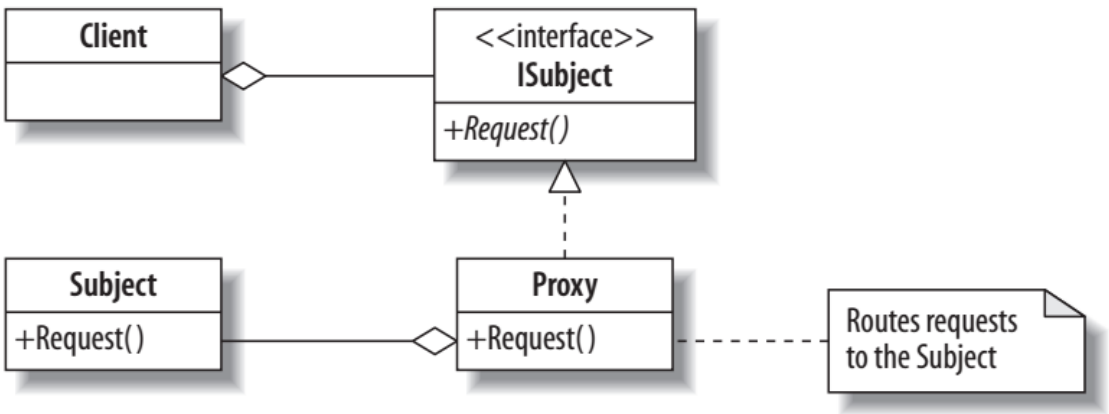


Figure: Proxy pattern UML diagram

Proxy Pattern Design: Players

- **ISubject:** A common interface for subjects and proxies that enables them to be used interchangeably
- **Subject:** The class that a proxy represents
- **Proxy:** A class that creates, controls, enhances, and authenticates access to a Subject
- **Request:** An operation on the Subject that is routed via a proxy

Proxy Pattern Design: Relationships

- Proxy, implements the ISubject interface
- Each Proxy object maintains a reference to a Subject, which is where the action really takes place

Proxy Pattern Illustration

- **ISubject:** A list of actions that can be done on the pages
- **Subject:** Actual pages belonging to one person
- **Subject.Request:** Actually changing the pages
- **Proxy:** Frontend to the actual pages
- **Proxy.Request:** Performs some action before routing the Request onward
- **Client:** A user visiting his/her pages

Proxy Pattern : Proxy Types

- **Virtual proxies:** Hands over the creation of an object to another object
- **Authentication proxies:** Checks that the access permissions for a request are correct
- **Remote Request:** Encodes requests and send them across a network
- **Smart proxies:** Adds to or change requests before sending them on

Virtual Proxy Example

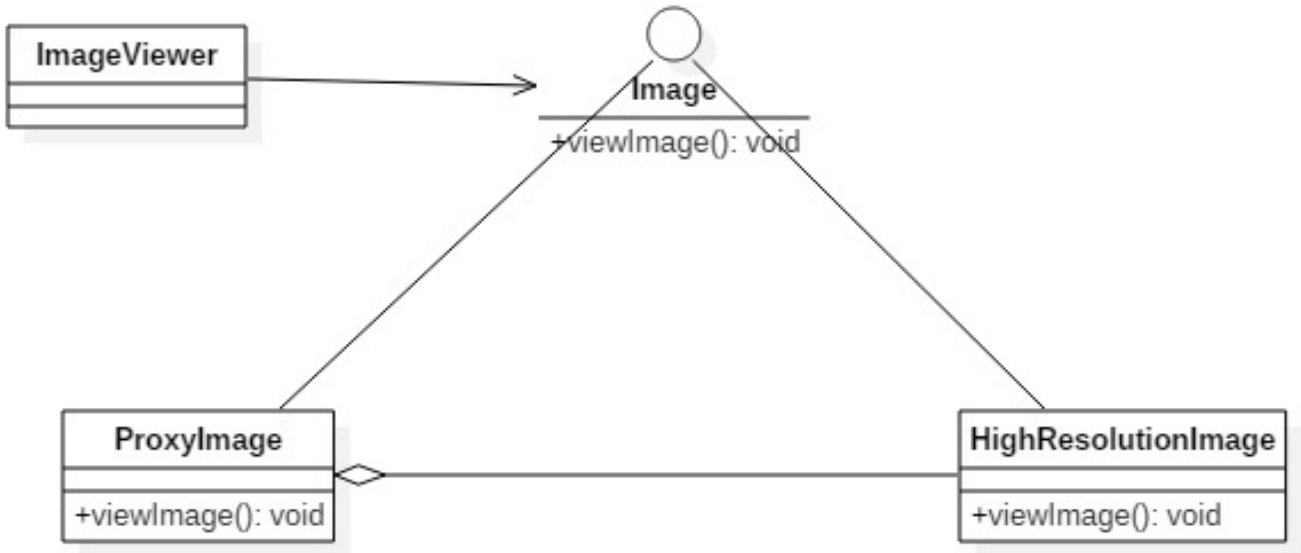


Figure: Virtual proxy pattern

Remote Proxy Example

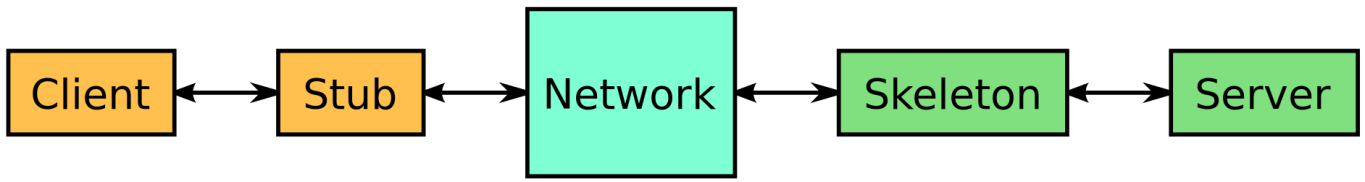


Figure: Stub object proxy in RMI

Proxy Pattern Implementation


```
class SubjectAccessor {
    public interface ISubject {
        string Request ();
    }

    private class Subject {
        public string Request() {
            return "Subject Request " + "Choose left door\n";
        }
    }

    public class Proxy : ISubject {
        Subject subject;

        public string Request() {
            // A Virtual Proxy creates the object only on its first method call
            if (subject == null) {
                Console.WriteLine("Subject inactive");
                subject = new Subject();
            }
            Console.WriteLine("Subject active");
            return "Proxy: Call to " + subject.Request();
        }
    }
```

Proxy Pattern Implementation


```
public class ProtectionProxy : ISubject {
    // An Authentication Proxy first asks for a password
    Subject subject;
    string password = "Abracadabra";

    public string Authenticate (string supplied) {
        if (supplied!=password)
            return "Protection Proxy: No access";
        else
            subject = new Subject();
        return "Protection Proxy: Authenticated";
    }

    public string Request() {
        if (subject==null)
            return "Protection Proxy: Authenticate first";
        else
            return "Protection Proxy: Call to "+
                   subject.Request();
    }
}
```

Proxy Pattern Implementation


```
class Client  : SubjectAccessor {
    static void Main() {
        Console.WriteLine("Proxy Pattern\n");
        ISubject subject = new Proxy();
        Console.WriteLine(subject.Request());
        Console.WriteLine(subject.Request());

        subject = new ProtectionProxy();
        Console.WriteLine(subject.Request());
        Console.WriteLine((subject as ProtectionProxy).Authenticate("Secret"));
        Console.WriteLine((subject as ProtectionProxy).Authenticate("Abracadabra"));
        Console.WriteLine(subject.Request());
        Console.ReadKey();
    }
}
```

Proxy Pattern Implementation


```
Proxy Pattern

Subject inactive
Subject active
Proxy: Call to Subject Request Choose left door

Subject active
Proxy: Call to Subject Request Choose left door

Protection Proxy: Authenticate first
Protection Proxy: No access
Protection Proxy: Authenticated
Protection Proxy: Call to Subject Request Choose left door
```

Proxy Pattern USE

- Proxies are front-ends to classes that have sensitive data or slow operations
- They are often found in image-drawing systems, as well as video streaming
- Proxies, like decorators, forward requests on to another object

Bridge Pattern Role

The Bridge pattern decouples an abstraction from its implementation, enabling them to vary independently. The Bridge pattern is useful when a new version of software is brought out that will replace an existing version, but the older version must still run for its existing client base. The bridge uses encapsulation, aggregation, and can use inheritance to separate responsibilities into different classes

Bridge Pattern Illustration

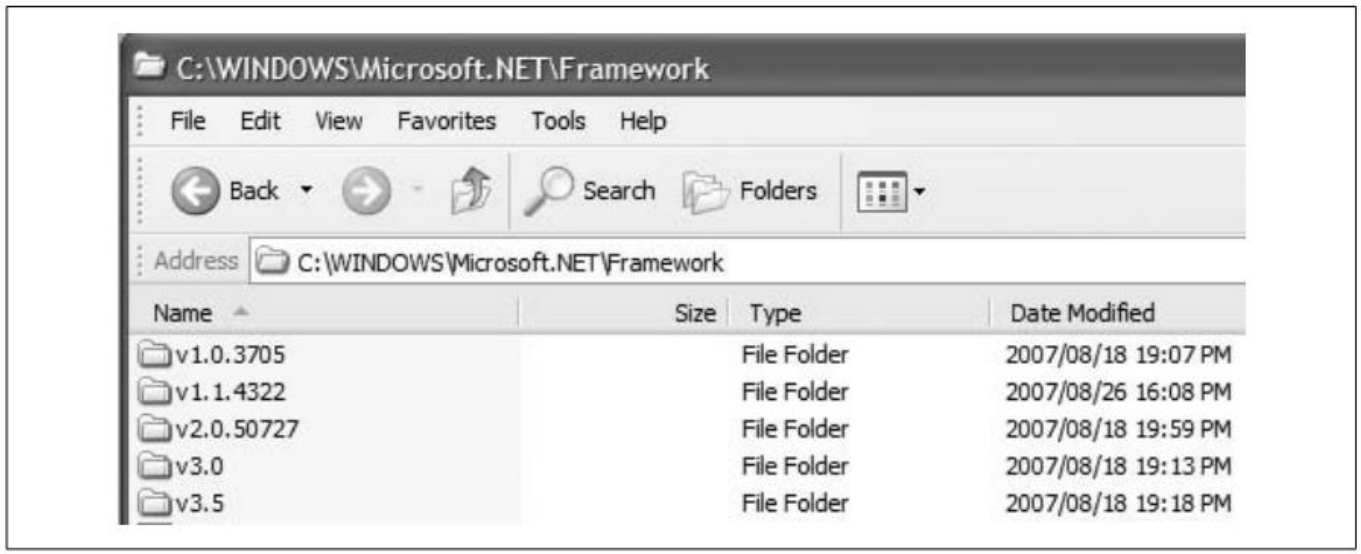


Figure: five versions of the .NET Framework loaded

Bridge Pattern Illustration

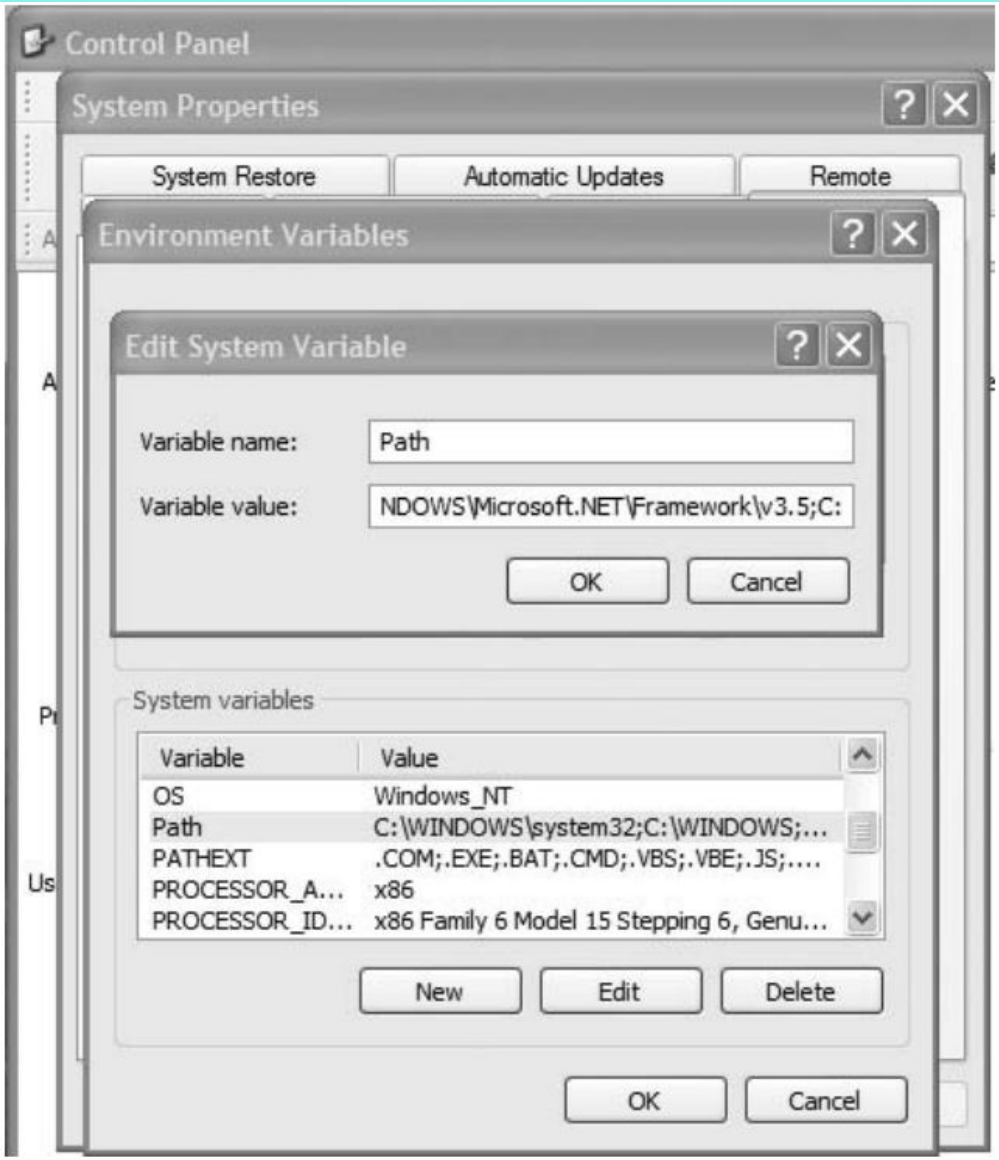
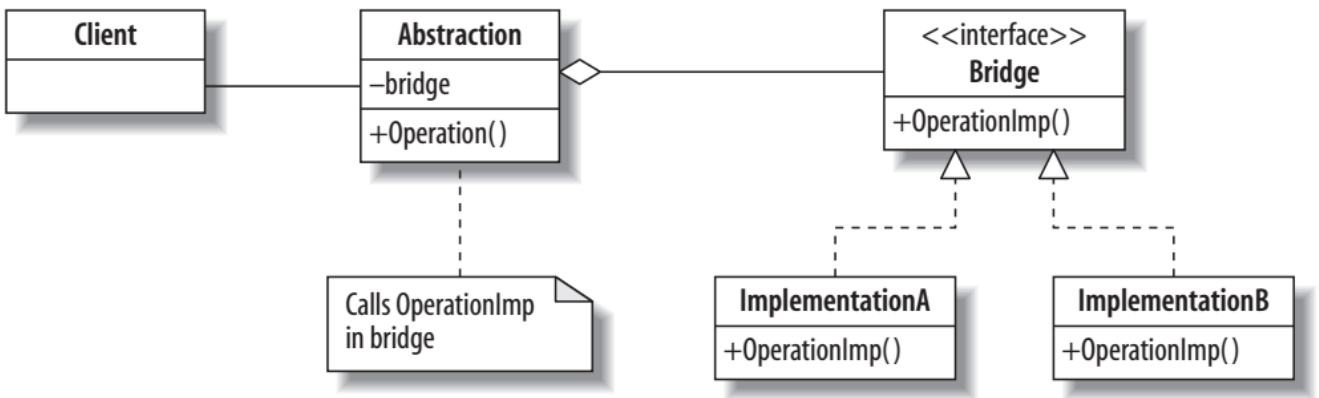


Figure: Environment Path variable set to Version 3.5

Bridge Pattern Design



Bridge Pattern Design: Players

- **Abstraction:** The interface that the client sees
- **Operation:** A method that is called by the client
- **Bridge:** An interface defining those parts of the Abstraction that might vary
- **ImplementationA and ImplementationB:** Implementations of the Bridge interface
- **OperationImp:** A method in the Bridge that is called from the Operation in the Abstraction

Bridge Pattern Design: Relationships

- Abstraction aggregates Bridge
- ImplementationA and ImplimentationB implement Bridge
- The Bridge pattern provides an alternative to inheritance when there is more than one version of an abstraction

Bridge Pattern Illustration

- **Abstraction:** NET Framework
- **Operation:** C# compiler
- **Bridge:** Environment Path variable
- **ImplementationA:** 3.5 version of the Framework
- **ImplementationB:** 2.0.50377 version and build of the Framework
- **A's OperationImp:** 3.0 compiler
- **B's OperationImp:** 2.0 compiler

Bridge Pattern Implementation


```
public interface IVideoQuality
{
    string PlayVideo();
}

class SdQuality : IVideoQuality
{
    private const string QualityName = "Video video in SD quality";

    public string PlayVideo()
    {
        return string.Format("Playing - {0}", QualityName);
    }
}
```

Bridge Pattern Implementation


```
class HdQuality : IVideoQuality
{
    private const string QualityName = "Video in Hd quality";

    public string PlayVideo()
    {
        return string.Format("Playing - {0}", QualityName);
    }
}

    class FhdQuality : IVideoQuality
{
    private const string QualityName = "Video in full Hd quality";

    public string PlayVideo()
    {
        return string.Format("Playing - {0}", QualityName);
    }
}
```

Bridge Pattern Implementation


```
class VideoPlayer
{
    IVideoQuality _currentVideoQuality;

    public IVideoQuality VideoQuality
    {
        get
        {
            return _currentVideoQuality;
        }
        set
        {
            _currentVideoQuality = value;
        }
    }

    public void PlayTv()
    {
        if (_currentVideoQuality != null)
        {
            Console.WriteLine(_currentVideoQuality.PlayVideo());
        }
        else
        {
            Console.WriteLine("Please select a Video Quality to play");
        }
    }
}
```

Bridge Pattern Implementation


```
static void Main(string[] args)
{
    VideoPlayer myVideoPlayer = new VideoPlayer();
    Console.WriteLine("Select A Quality to Play");
    Console.WriteLine("1. SD \n2. HD \n3. FHD");
    ConsoleKeyInfo input = Console.ReadKey();
    // Let us see what user has selected and select the video Quality accrodingly
    switch (input.KeyChar)
    {
        case '1':
            myVideoPlayer.VideoQuality = new SdQuality();
            break;
        case '2':
            myVideoPlayer.VideoQuality = new HdQuality();
            break;
      case '3':
            myVideoPlayer.VideoQuality = new FhdQuality();
            break;

    }
    Console.WriteLine();
    //Let us now play the selected Quality.
    myVideoPlayer.PlayTv();
    Console.WriteLine();
    Console.ReadKey();
}
```

Bridge Pattern USE

A well-quoted use of the Bridge pattern is in graphics, where different displays have different capabilities and drivers.
We can use it when we want to :

- Completely hide implementations from clients
- Avoid binding an implementation to an abstraction directly
- Change an implementation without even recompiling an abstraction

Creational Patterns

The creational patterns aim to separate a system from how its objects are created, composed, and represented. Creational patterns encapsulate the knowledge about which classes a system uses, but they hide the details of how the instances of these classes are created and put together.

Creational Patterns

The creational patterns aim to separate a system from how its objects are created, composed, and represented. Creational patterns encapsulate the knowledge about which classes a system uses, but they hide the details of how the instances of these classes are created and put together.
Programmers have come to realize that composing systems with inheritance makes those systems too rigid. The creational patterns are designed to break this close coupling. The names of these patterns are

Creational Patterns

The creational patterns aim to separate a system from how its objects are created, composed, and represented. Creational patterns encapsulate the knowledge about which classes a system uses, but they hide the details of how the instances of these classes are created and put together.
Programmers have come to realize that composing systems with inheritance makes those systems too rigid. The creational patterns are designed to break this close coupling. The names of these patterns are

- Prototype
- Factory Method
- Singleton
- Abstract Factory
- Builder

Factory Pattern Role

The Factory Method pattern is a way of creating objects, but letting subclasses decide exactly which class to instantiate. Various subclasses might implement the interface; the Factory Method instantiates the appropriate subclass based on information supplied by the client or extracted from the current state.

Factory Pattern Illustration

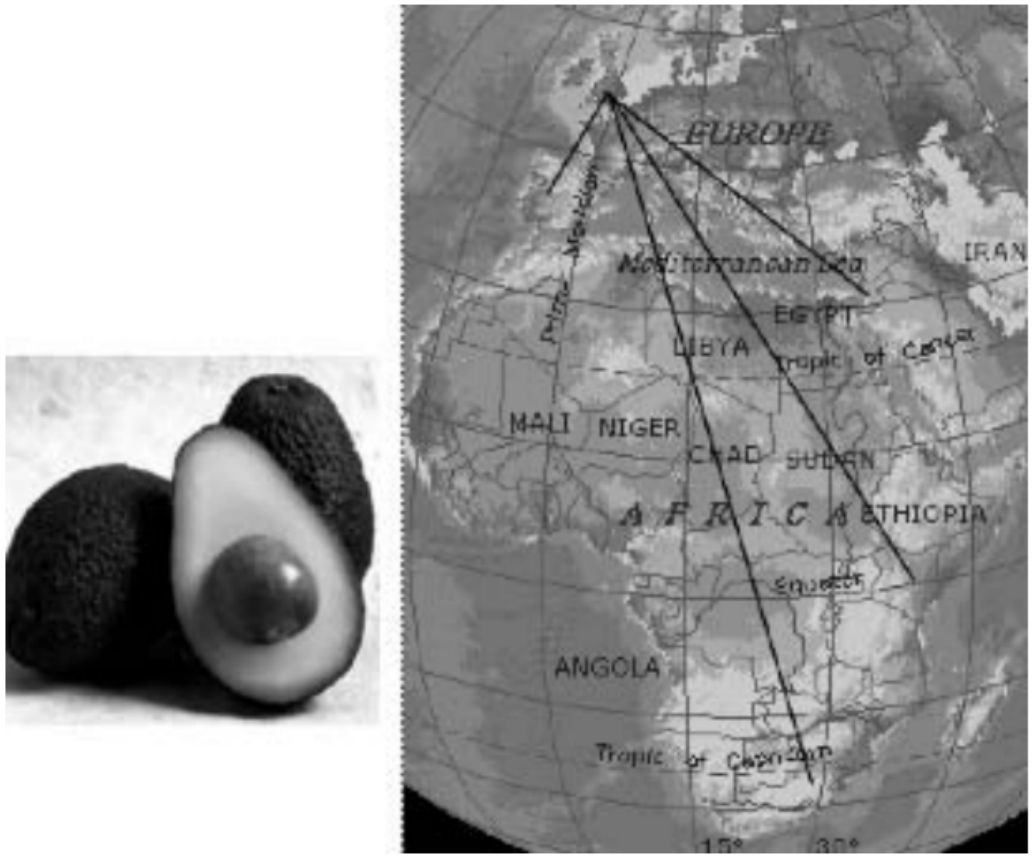
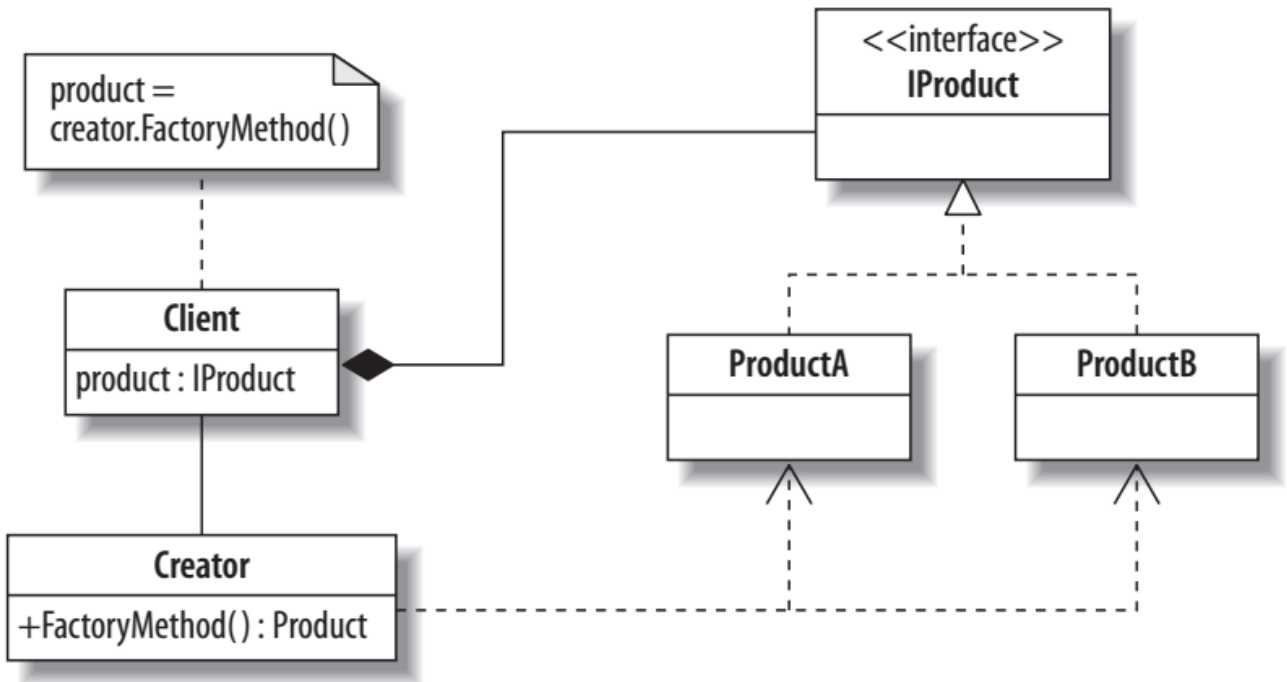


Figure: Factory Method pattern illustration—avocado sourcing

Factory Pattern Design



Factory Pattern Design: Players

- **IProduct:** The interface that the client sees
- **ProductA and ProductB:** Classes that implement IProduct
- **Creator:** Provides the FactoryMethod
- **FactoryMethod:** Decides which class to instantiate

Factory Pattern Design: Relationships

- The client could request many products without knowing their subclasses
- The client can invoke multiple creators for different types of products

Factory Pattern Illustration

- **Client:** Storekeeper
- **Creator:** Buyer
- **ProductA:** Supplier of avocados from Spain
- **ProductB:** Supplier of avocados from South Africa
- **IProduct:** Supplying avocados

Factory Pattern Implementation


```
interface IProduct {
  string ShipFrom();
}

class ProductA : IProduct {
  public String ShipFrom () {
    return " from South Africa";
  }
}

class ProductB : IProduct {
  public String ShipFrom () {
          return "from Spain";
  }
}
```

Factory Pattern Implementation


```
class Creator {
  public  IProduct FactoryMethod(int month) {
    if (month >= 4 && month <=11)
      return new ProductA();
    else
    if (month == 1 || month == 2 || month == 12)
      return new ProductB();
    else
      return new DefaultProduct();
  }
}
```

Factory Pattern Implementation


```
    static void Main() {
      Creator c = new Creator();
      IProduct product;

      for (int i=1; i<=12; i++) {
        product = c.FactoryMethod(i);
        Console.WriteLine("Avocados "+product.ShipFrom());
      }
      Console.ReadKey();
    }
}
```

Factory Pattern Implementation


```
/* Output
Avocados from Spain
Avocados from Spain
Avocados not available
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados  from South Africa
Avocados from Spain
*/
```

Singleton Pattern Role

The purpose of the Singleton pattern is to ensure that there is only one instance of a class, and that there is a global access point to that object.

- The pattern ensures that the class is instantiated only once and that all requests are directed to that one and only object
- It is the class itself that is responsible for ensuring this constraint, not the clients of the class
- The object should not be created until it is actually needed

Singleton Pattern Illustration

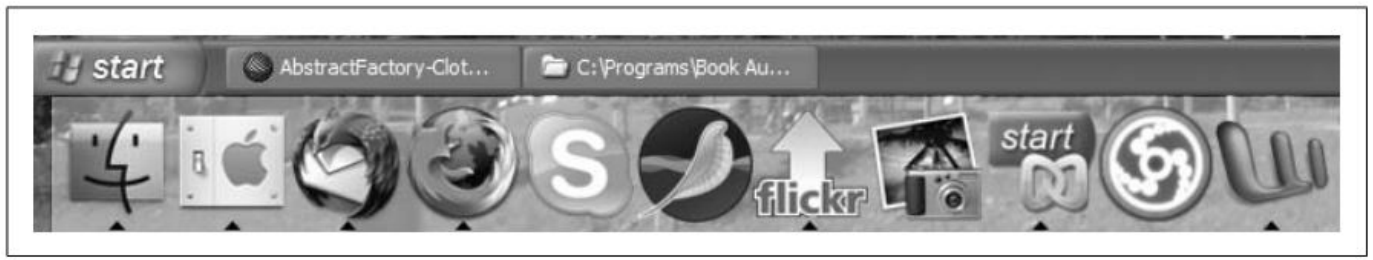
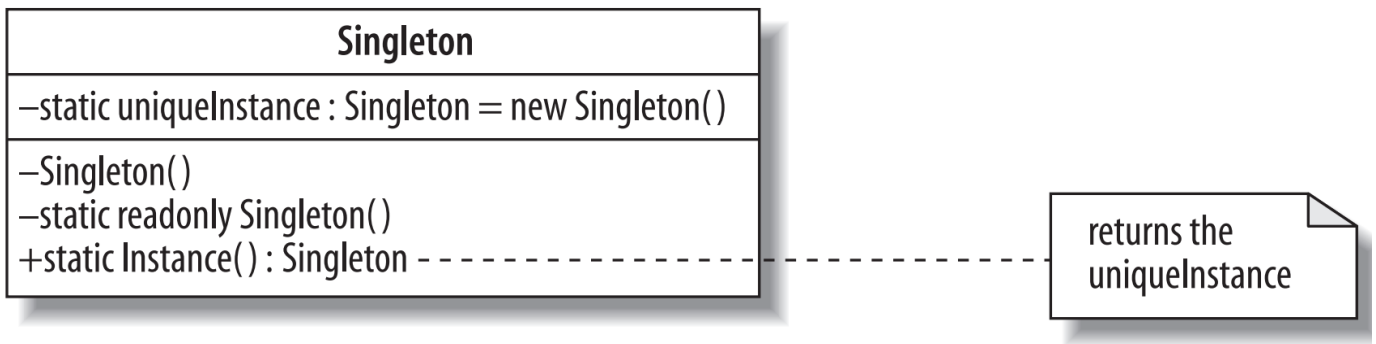


Figure: Singleton pattern illustration—the Mac Dock

Singleton Pattern Design



Singleton Pattern Design: Players

- **Singleton:** The class containing the mechanism for a unique instance of itself
- **Instance:** A property for making and accessing an instance of the Singleton class

Singleton Pattern Design: Modifications

- Make the constructor private and add a private static constructor as well
- Add a private static read-only object that is internally instantiated using the private constructor
- Add a public static property that accesses the private object

Singleton Pattern Illustration

- **Singleton:** An application icon in the Dock
- **Instance:** An application

Singleton Pattern Implementation


```
public sealed class Singleton {
  // Private Constructor
  Singleton() { }

  // Private object instantiated with private constructor
  static readonly Singleton Instance = new Singleton();

  // Public static property to get the object
  public static Singleton UniqueInstance {
    get { return Instance;}
  }
}
```

Singleton Pattern Implementation


```
static void Main() {
  Singleton s1 = Singleton.UniqueInstance;
  Singleton s2 = Singleton.UniqueInstance;

  if (s1 == s2) {
    Console.WriteLine("Objects are the same instance");
  }
  Console.ReadKey();
}
```

Singleton Pattern Implementation


```
/* OutPut
Objects are the same instance
*/
```

Singleton Pattern USE

Many other patterns—for example, the Builder, Prototype, and Abstract Factory patterns can make use of the Singleton pattern to ensure that only one copy of a class is created. In Facade pattern, Facade objects are often singletons.
We can use it when we want to :

- Ensure that there is only one instance of a class
- Control access to that instance is essential

Prototype, Factory and Singleton patterns comparison

Abstract Factory Pattern Role

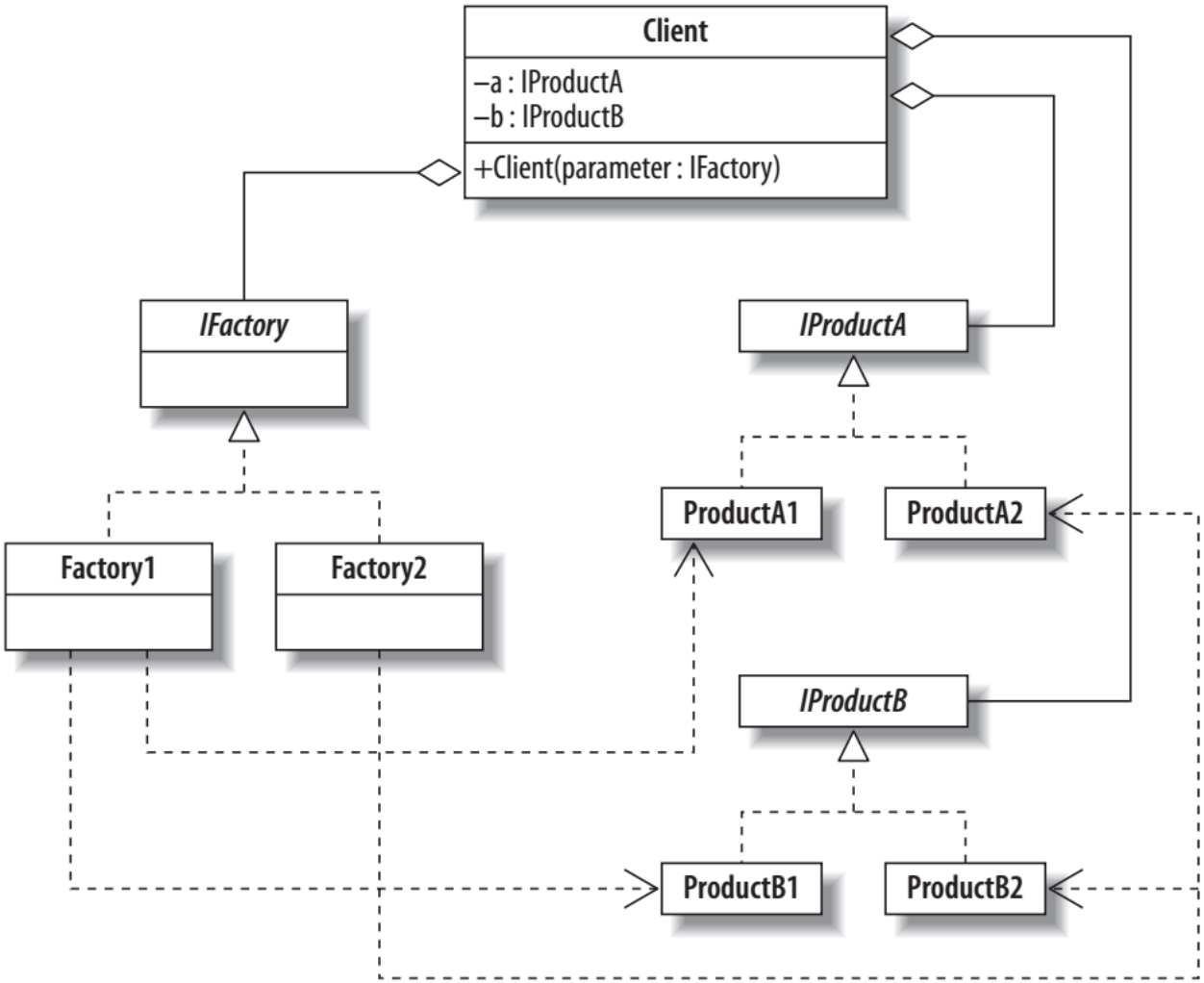
This pattern supports the creation of products that exist in families and are designed to be produced together. The abstract factory can be refined to concrete factories, each of which can create different products of different types and in different combinations

- The pattern isolates the product definitions and their class names from the client
- Product families can easily be interchanged or updated

Abstract Factory Pattern Illustration

Figure: Abstract Factory pattern illustration—replica Gucci handbags (Poochies)

Abstract Factory Design



Abstract Factory Pattern Design: Players

- **AbstractFactory:** An interface with Create operations for each of the abstract products
- **Factory1, Factory2:** Implementations of all the AbstractFactory creation operations
- **AbstractProduct:** An interface for a kind of product with its own operations
- **ProductA1, ProductA2, ProductB1, ProductB2 :** Classes that implement the AbstractProduct interface and define product objects to be created by the corresponding factories
- **Client:** A class that accesses only the AbstractFactory and AbstractProduct interfaces

Abstract Factory Pattern Illustration

- **Client:** Customer buying a handbag
- **AbstractFactory:** Returns a handbag
- **FactoryA:** Gucci
- **FactoryB:** Poochy
- **AbstractProduct:** Handbags
- **ProductA1:** Genuine bags
- **ProductA2:** Replica bags

Generic Classes

- Generic classes are meant to define type-safe data structures, without committing to actual data types.
- using generic classes results in re-usability, type safety and efficiency and good code quality.
- Programmer can reuse data processing algorithms without duplicating type-specific code.
- They are widely used in collections like linked lists, stacks, trees ...

Generic Classes - Syntax

- Generic classes are meant to define type-safe data structures, without committing to actual data types.
- using generic classes results in re-usability, type safety and efficiency and good code quality.
- Programmer can reuse data processing algorithms without duplicating type-specific code.
- They are widely used in collections like linked lists, stacks, trees ...

Generic classes- Syntax


```
public class Stack<T>
{
    T[] m_Items;

    public void Push(T item)
    {

    }

    public T Pop()
    {

    }
}
class Program
{
    static void Main(string[] args)
    {

    Stack<int> stack = new Stack<int>();
    stack.Push(1);
    stack.Push(2);
    int number = stack.Pop();
    }
}
```

Generic classes - Syntax


```
public class Stack<T> where T : new()
{
    T[] m_Items;

    public void Push(T item)
    {

    }

    public T Pop()
    {
        return new T();
    }
}
```

Generic classes - Example


```
public class MyGenericArray<T>
{
    private T[] array;

    public MyGenericArray(int size)
    {
        array = new T[size + 1];
    }

    public T getItem(int index)
    {
        return array[index];
    }

    public void setItem(int index, T value)
    {
        array[index] = value;
    }
}
```

Generic classes - Example


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

         //declaring an int array
         MyGenericArray<int> intArray = new MyGenericArray<int>(5);

         //setting values
         for (int c = 0; c < 5; c++)
         {
             intArray.setItem(c, c * 5);
         }

         //retrieving the values
         for (int c = 0; c < 5; c++)
         {
             Console.Write(intArray.getItem(c) + " ");
         }

         intArray.setItem(2,4);
         Console.WriteLine();
```

Generic classes - Example


```
//declaring a character array
MyGenericArray<char> charArray = new MyGenericArray<char>(5);

//setting values
for (int c = 0; c < 5; c++)
{
    charArray.setItem(c, (char)(c + 97));
}

//retrieving the values
for (int c = 0; c < 5; c++)
{
    Console.Write(charArray.getItem(c) + " ");
}
Console.WriteLine();
intArray.setItem(2,'d');
Console.ReadKey();
```

Generic classes - Example


```
/* Output */
// 0   5   10   15   20
// a   b   c    d    e
```

Abstract Factory Pattern Implementation


```
interface IFactory<Brand>
  where Brand : IBrand {
  IBag CreateBag();
  IShoes CreateShoes();
}

// Concrete Factories (both in the same one)
class Factory<Brand> : IFactory<Brand>
  where Brand : IBrand, new() {
  public IBag CreateBag() {
    return new Bag<Brand>();
  }

  public IShoes CreateShoes() {
    return new Shoes<Brand>();
  }
}
```

Abstract Factory Pattern Implementation


```
// Product 1
interface IBag {
    string Material { get; }
}

// Product 2
interface IShoes {
    int Price { get; }
}
```

Abstract Factory Pattern Implementation


```
// Concrete Product 1
class Bag<Brand> : IBag
  where Brand : IBrand, new() {
  private Brand myBrand;
  public Bag() {
    myBrand = new Brand();
  }

  public string Material { get { return myBrand.Material; } }
}

// Concrete Product 2
class Shoes<Brand> : IShoes
  where Brand : IBrand, new() {

  private Brand myBrand;

  public Shoes() {
    myBrand = new Brand();
  }

  public int Price { get { return myBrand.Price; } }
}
```

Abstract Factory Pattern Implementation


```
interface IBrand {
  int Price { get; }
  string Material { get; }
}

class Gucci : IBrand {
  public int Price { get { return 1000; } }
  public string Material { get { return "Crocodile skin"; } }
}

class Poochy : IBrand {
  public int Price { get { return new Gucci().Price / 3; } }
  public string Material { get { return "Plastic"; } }
}

class Groundcover : IBrand {
  public int Price { get { return 2000; } }
  public string Material { get { return "South african leather"; } }
}
```

Abstract Factory Pattern Implementation


```
class Client<Brand>
  where Brand : IBrand, new() {
  public void ClientMain() { //IFactory<Brand> factory)
    IFactory<Brand> factory = new Factory<Brand>();

    IBag bag = factory.CreateBag();
    IShoes shoes = factory.CreateShoes();

    Console.WriteLine("I bought a Bag which is made from " + bag.Material);
    Console.WriteLine("I bought some shoes which cost " + shoes.Price);
    Console.ReadKey();
  }
}

static class Program {
  static void Main() {
    // Call Client twice
    new Client<Poochy>().ClientMain();
    new Client<Gucci>().ClientMain();
    new Client<Groundcover>().ClientMain();
  }
}
```

Abstract Factory Pattern Implementation


```
/* Output
I bought a Bag which is made from Plastic
I bought some shoes which cost 333
I bought a Bag which is made from Crocodile skin
I bought some shoes which cost 1000
I bought a Bag which is made from South african leather
I bought some shoes which cost 2000
*/
```

Abstract Factory Pattern USE

- It is ideal for generating different layouts and multiple-look-and-feel user interfaces
- It can also be successfully applied for portability across operating systems

Limitation: It is not easy to add new kinds of products. The number and names of the abstract products are coded directly into the abstract factory

Iterator Pattern Role

The Iterator pattern provides a way of accessing elements of a collection sequentially, without knowing how the collection is structured. As an extension, the pattern allows for filtering elements in a variety of ways as they are generated.

Iterator Pattern Illustration

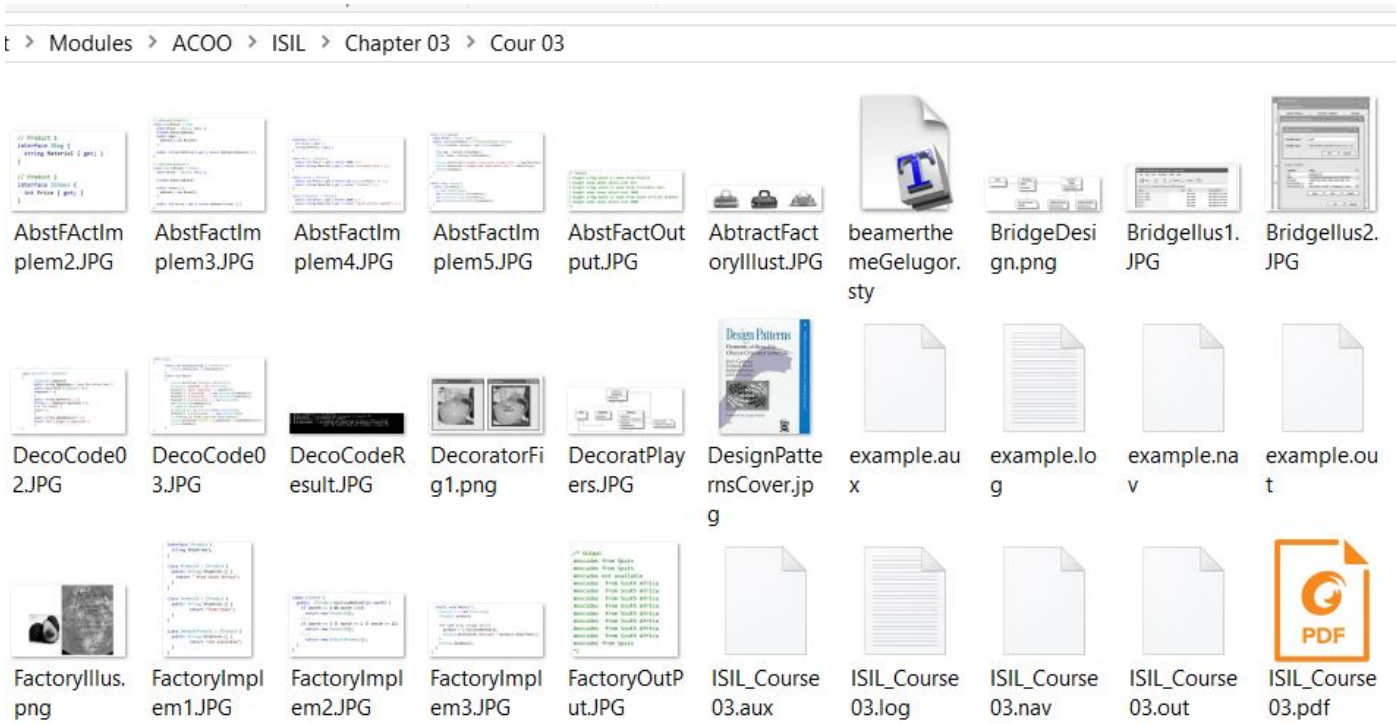


Figure: Iterator pattern illustration (a)—files

Iterator Pattern Illustration

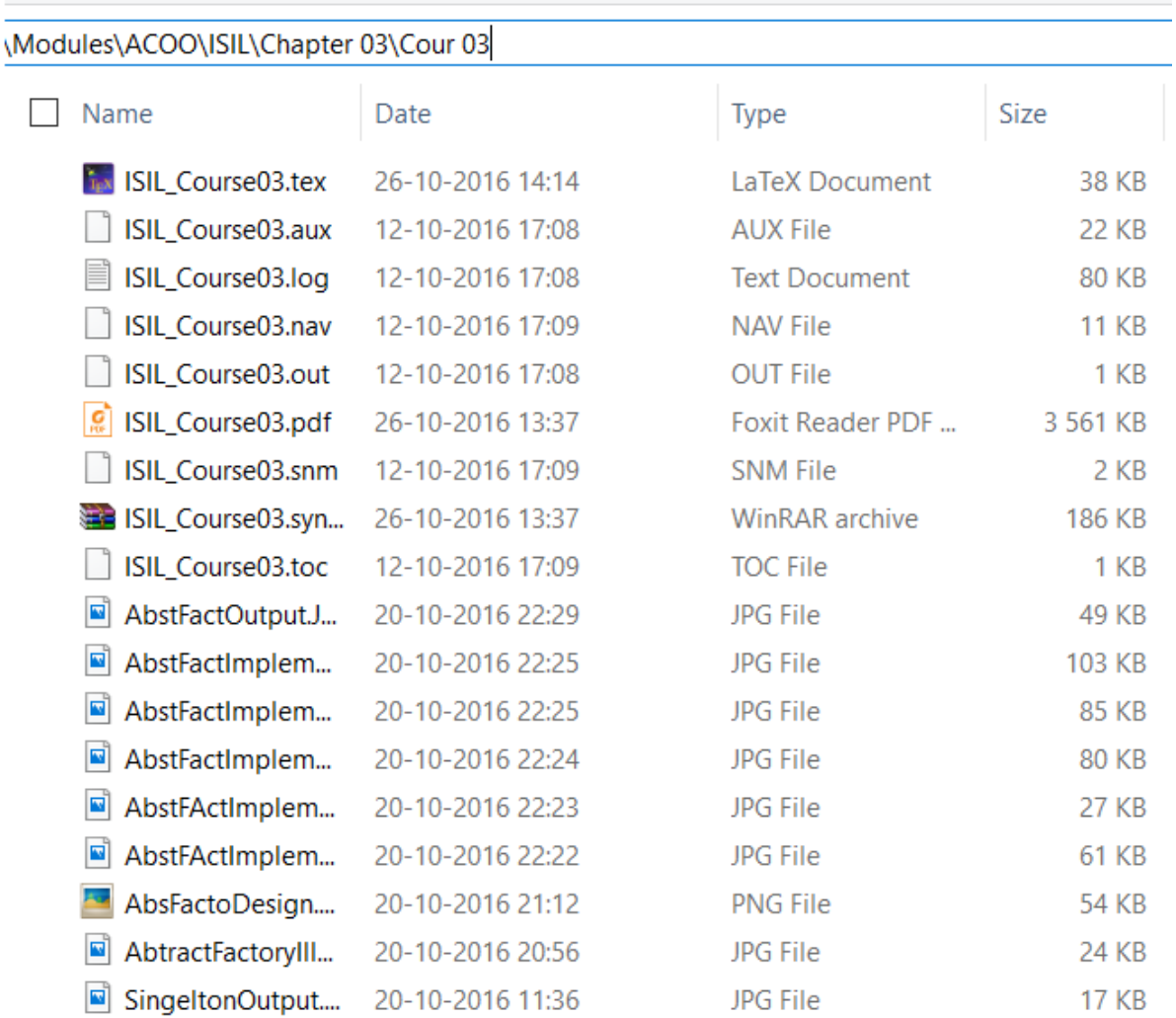


Figure: Iterator pattern illustration (a)—files ordered by date

Iterator Pattern Illustration

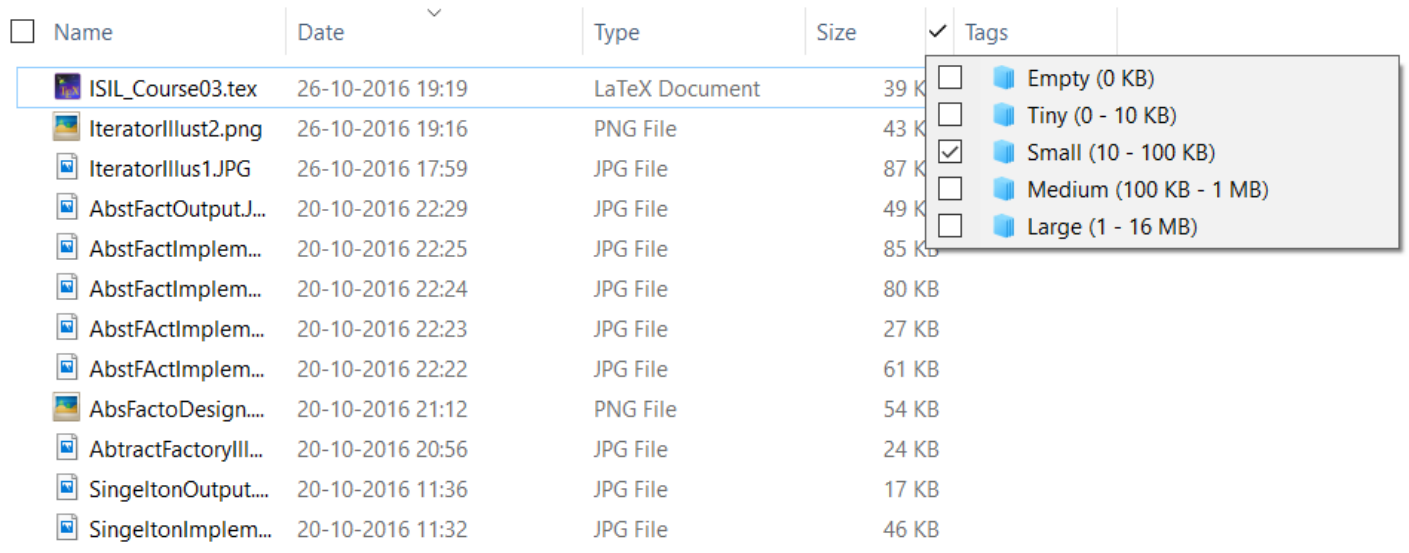
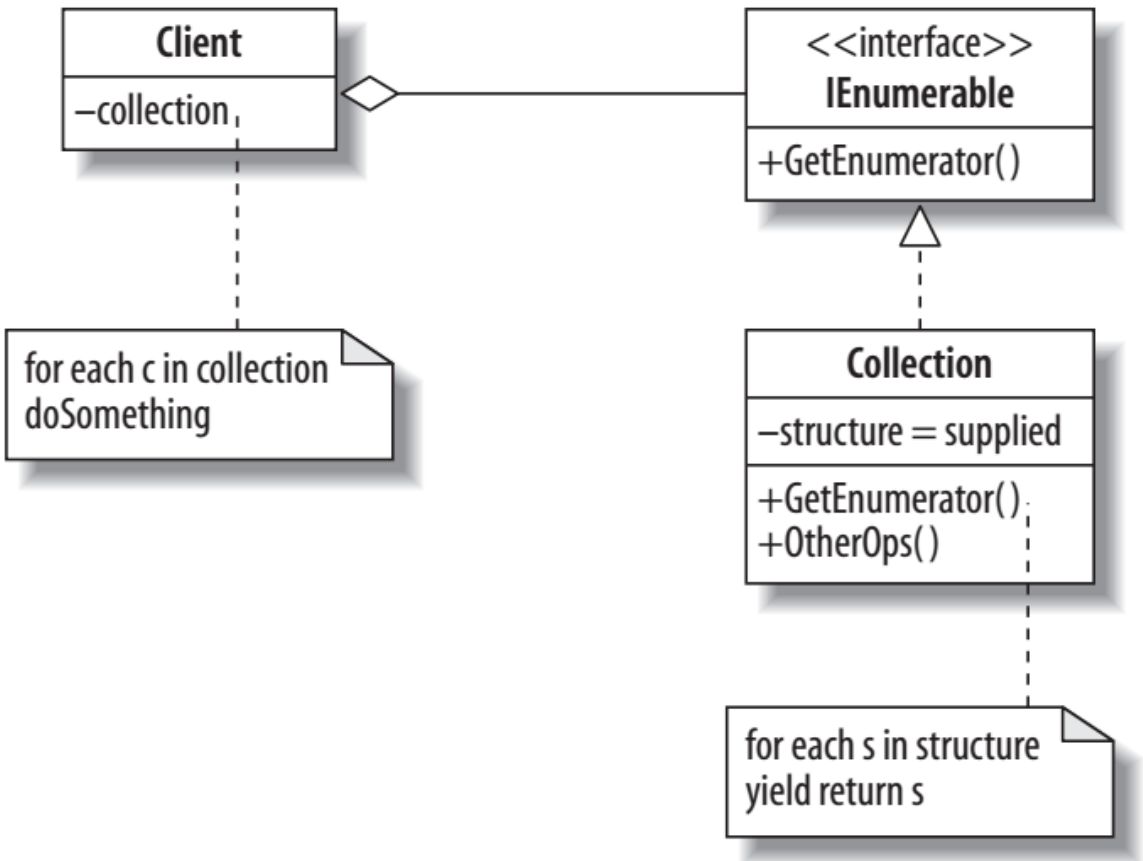


Figure: Iterator pattern illustration (b)—files ordered by size

Iterator Pattern Design



Iterator Pattern Design: Players

- **Client:** Keeps a Collection object and uses a foreach statement to iterate over it
- **IEnumerable:** A defined interface in C# for the operation GetEnumerator
- **Collection:** A data type containing or having the ability to generate a collection of values
- **GetEnumerator :** A method that supplies the values in sequence
- **OtherOps:** Other methods that supply Collection values in different sequences with different filters and transformations

Iterator Pattern Illustration

- **Client:** The user who is iterating over file directories
- **Collection:** A structure of directories and files
- **GetEnumerator:** Returns a single file or directory item
- **OtherOps:** Returns a single file or directory item according to other criteria (such as date or size)

Iterator Pattern Implementation


```
class MonthCollection : IEnumerable {

  string [] months = {"January", "February", "March", "April", "May", "June",
  "July", "August", "September", "October", "November", "December"};

  public IEnumerator GetEnumerator () {
    // Generates values from the collection
      foreach (string element in months)
      yield return element;
    }
  }
```

Iterator Pattern Implementation


```
  static void Main() {
    MonthCollection collection = new MonthCollection();
    // Consumes values generated from collection's GetEnumeror method
    foreach (string n in collection)
      Console.Write(n+"   ");
      Console.WriteLine("\n");
      Console.ReadKey();
    }
}
```

Iterator Pattern Implementation


```
/*Output
January   February   March   April   May   June   July   August   September   October
November   December
*/
```

Enumerators vs Iterators

- **Enumerators:** Provided by all the collections in the library, generic and nongeneric. They are supported by the IEnumerable interface and the yield return and yield break statements.
- **Iterators:** Written by clients and supported by the foreach statement and ***query expressions***.

Query expressions - LINQ

Query

A query is a set of instructions that describes what data to retrieve from a given data source

Query Expression

A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery

LINQ

Language-Integrated Query (LINQ) is the name for a set of technologies based on the integration of query capabilities directly into the C# language (also in Visual Basic and potentially any other .NET language). With LINQ, a query is now a first-class language construct, just like classes, methods, events and so on

Query expressions - LINQ

Three main steps :

- Obtain the data source
- Create the query
- Execute the query

Iterator Pattern - LINQ


```
static void Main() {

  Dictionary <string, int> daysInMonths = new Dictionary <string, int> {
      {"January", 31},    {"February", 28},
      {"March", 31},      {"April", 30},
      {"May", 31},         {"June", 30},
      {"July", 31},          {"August", 31},
      {"September", 30}, {"October", 31},
      {"November", 30},  {"December", 31}};

  var selection = from n in daysInMonths
          where n.Key.Length> 5
          select n;

  selection = from n in selection
          where n.Value ==31
          orderby n.Key
          select n;
```

Alterator Pattern - LINQ


```
        foreach (KeyValuePair<string,int> n in selection)
              Console.Write(n+"    ");
        Console.WriteLine("\n");
        Console.ReadKey();
    }
}
```

Iterator Pattern - LINQ


```
/*Output

[January, 31]   [August, 31]   [October, 31]   [December, 31]

*/
```

Query expressions - Lambda Expression

Lambda expressions are particularly helpful for writing LINQ query expressions.
(input parameters) => expression

- specify input parameters (if any) on the left side of the lambda operator $=>$
- put the expression or statement block on the other side

Query expressions - Lambda Expression


```
var selection = daysInMonths.Where
    (pair => (pair.Key.Length > 5 && pair.Value == 31));
```

Iterator Pattern USE

The Iterator pattern is useful when :

- The structure is complex
- Different iterations are required over it, potentially at the same time
- The same structure and iterations can be applied to different data

Mediator Pattern Role

The Mediator pattern is there to enable objects to communicate without knowing each other's identities. It also encapsulates a protocol that objects can follow.

Mediator Pattern Illustration

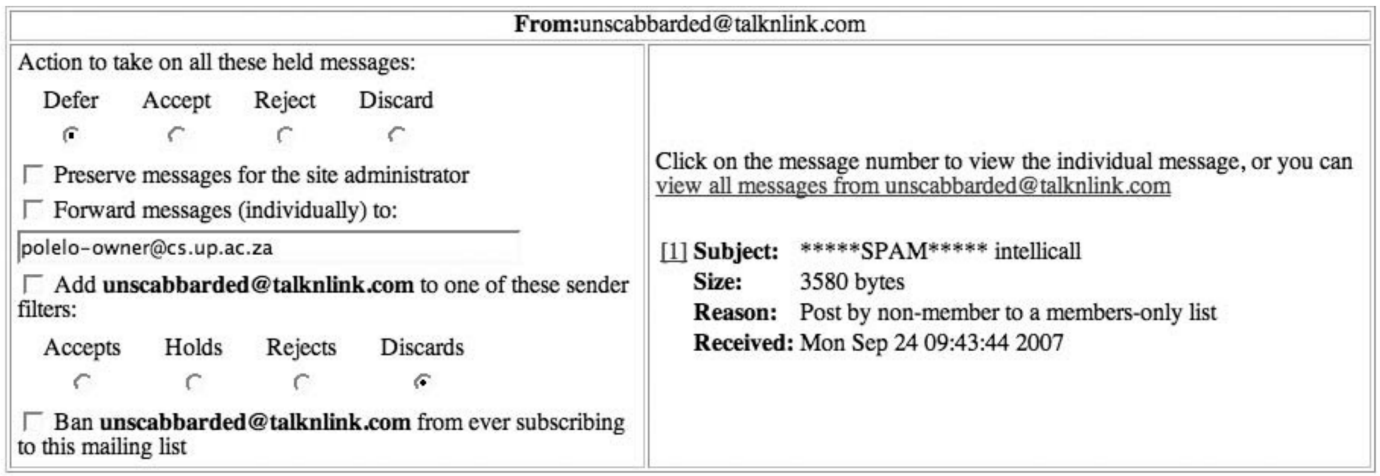
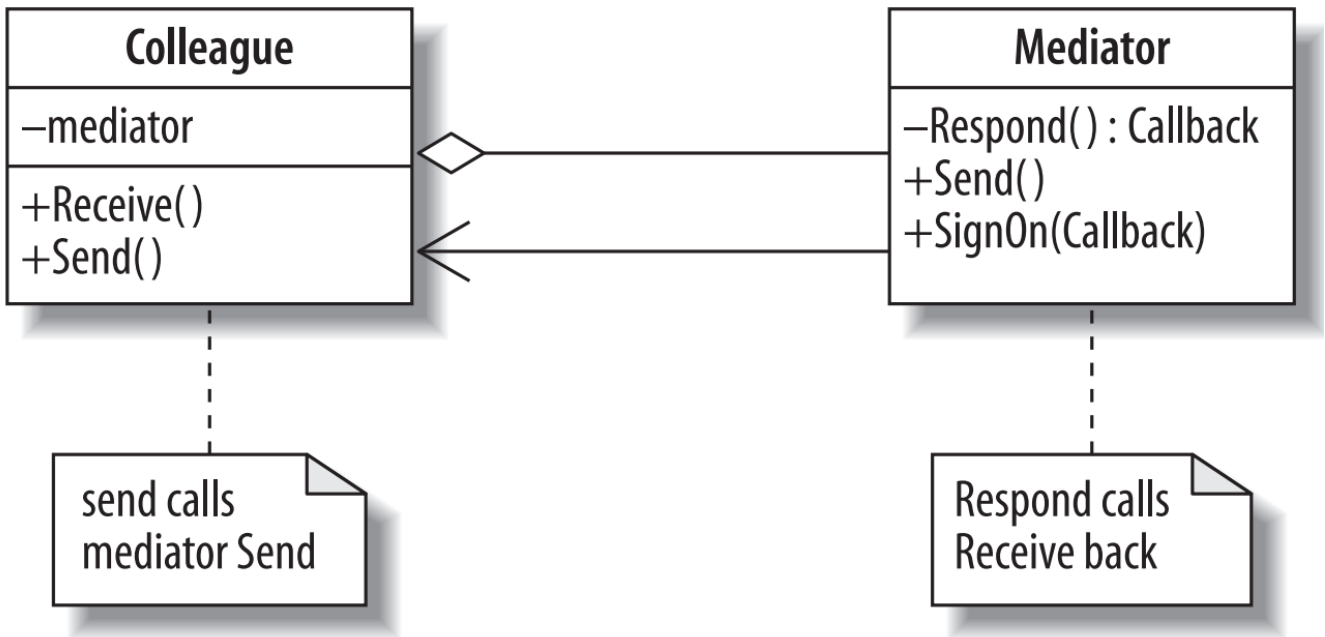


Figure: Mediator pattern illustration—mailing list moderation

Mediator Design



Mediator Pattern Design: Players

- **Colleague:** Registers with a Mediator by supplying a Receive method; issues Send requests to other colleagues via the Mediator
- **Mediator:** Broadcasts sent messages to all signed-on Colleagues using the Respond delegate

Mediator Pattern Illustration

- **Colleague:** A user subscribed to a mailing list
- **Mediator:** The mailing list moderator
- **Respond:** Calls back the signed-on users
- **Callback:** A means of contacting all users on the list
- **Send:** Mailing a message to the list
- **Receive:** Receiving a message from the list

Delegates

- Used for passing a function itself as a parameter
- A delegate is like a pointer to a function in C++. It is a reference type data type and it holds the reference of a method
- Delegates are used to define callback methods and implement event handling
- It has the form delegate result-type identifier ([*parameters*])
  public delegate void MyDelegate(string text);
- A delegate should always be instantiated before being used

Example Delegate


```
public delegate int MyDelegate(int x, int y);

class Program

{

    static int Sum(int x, int y)

    {

        return x + y;

    }

    static void Main()

    {

        MyDelegate d = new MyDelegate(Sum);
        int result = d.Invoke(12, 15);
        Console.WriteLine(result);
    }
}
```

Multicast delegate

he "+" operator adds a function to the delegate object and the "-" operator removes an existing function from a delegate object

Multicast delegate


```
  public delegate void Print(int value);
static void Main(string[] args)
  {
      Print printDel = PrintNumber;
      printDel += PrintHexadecimal;
      printDel += PrintMoney;
      printDel(1000);
      printDel -= PrintHexadecimal;
      printDel(2000);
  }
public static void PrintNumber(int num)
  {
      Console.WriteLine("Number: {0,-12:N0}", num);
  }
public static void PrintMoney(int money)
  {
      Console.WriteLine("Money: {0:C}", money);
  }
public static void PrintHexadecimal(int dec)
  {
      Console.WriteLine("Hexadecimal: {0:X}", dec);
  }
```

Multi-Cast Output

Number: 1,000
Hexadecimal: 3EB
Money: $ 1,000.00
Number: 2,000
Money: $2,000.00

Mediator Pattern Implementation

- Each Colleague is passed a Mediator at instantiation and keeps it as a private reference
- Each Mediator keeps a list of signed-on Colleagues as a private reference

Mediator Pattern Implementation


```
class Mediator {

  public delegate void Callback (string message, string from);
  Callback respond;

  public void SignOn (Callback method) {
    respond += method;
  }
  public void Block (Callback method) {
    respond -=method;
  }
  public void Unblock(Callback method) {
    respond +=method;
  }

  // Send is implemented as a broadcast
  public void Send(string message, string from) {
    respond(message, from);
    Console.WriteLine();
  }
}
```

Mediator Pattern Implementation


```
class Colleague {
  Mediator mediator;
  protected string name;

  public Colleague(Mediator mediator,string name) {
    this.mediator = mediator;
    mediator.SignOn(Receive);
    this.name = name;
  }

  public virtual void Receive(string message, string from) {
    Console.WriteLine(name +" received from "+from+": " + message);
  }

  public void Send(string message) {
    Console.WriteLine("Send (From "+name+ "): "+message);
    mediator.Send(message, name);
  }
}
```

Mediator Pattern Implementation


```
class ColleagueB :  Colleague {
  public ColleagueB(Mediator mediator,string name)
      : base (mediator, name) {
}

// Does not get copies of own messages
public override void Receive(string message, string from) {
  if (!String.Equals(from,name))
    Console.WriteLine(name +" received from "+from+": " + message);
  }
}
```

Mediator Pattern Implementation


```
static void Main () {
  Mediator m = new Mediator();
  // Two from head office and one from a branch office
    Colleague head1 = new Colleague(m,"John");
    ColleagueB branch1 = new ColleagueB(m,"David");
    Colleague head2 = new Colleague(m,"Lucy");

    head1.Send("Meeting on Tuesday, please all ack");
    branch1.Send("Ack"); // by design does not get a copy
    m.Block(branch1.Receive); // temporarily gets no messages
    head1.Send("Still awaiting some Acks");
    head2.Send("Ack");
    m.Unblock(branch1.Receive); //open again
    head1.Send("Thanks all");
    Console.ReadKey();
  }
}
```

Mediator Pattern Implementation


```
/* Output
Send (From John): Meeting on Tuesday, please all ack
John received from John: Meeting on Tuesday, please all ack
David received from John: Meeting on Tuesday, please all ack
Lucy received from John: Meeting on Tuesday, please all ack

Send (From David): Ack
John received from David: Ack
Lucy received from David: Ack

Send (From John): Still awaiting some Acks
John received from John: Still awaiting some Acks
Lucy received from John: Still awaiting some Acks

Send (From Lucy): Ack
John received from Lucy: Ack
Lucy received from Lucy: Ack

Send (From John): Thanks all
John received from John: Thanks all
Lucy received from John: Thanks all
David received from John: Thanks all
*/
```

Mediator Pattern USE

- It is used in many modern systems that reflect a send/receive protocol, such as list servers and chat rooms
- It has an important role to play as a means of centralizing interconnections between objects

Observer Pattern Role

The Observer pattern defines a relationship between objects so that when one changes its state, all the others are notified accordingly. There is usually an identifiable single publisher of new state, and many subscribers who wish to receive it.

A good example of Oberver pattern is blog site. Each blog site is maintained by a single blogger. People who come to read about the information put out by a blogger and find it interesting can subscribe to be informed each time a new blog entry is posted.

Observer Pattern Illustration

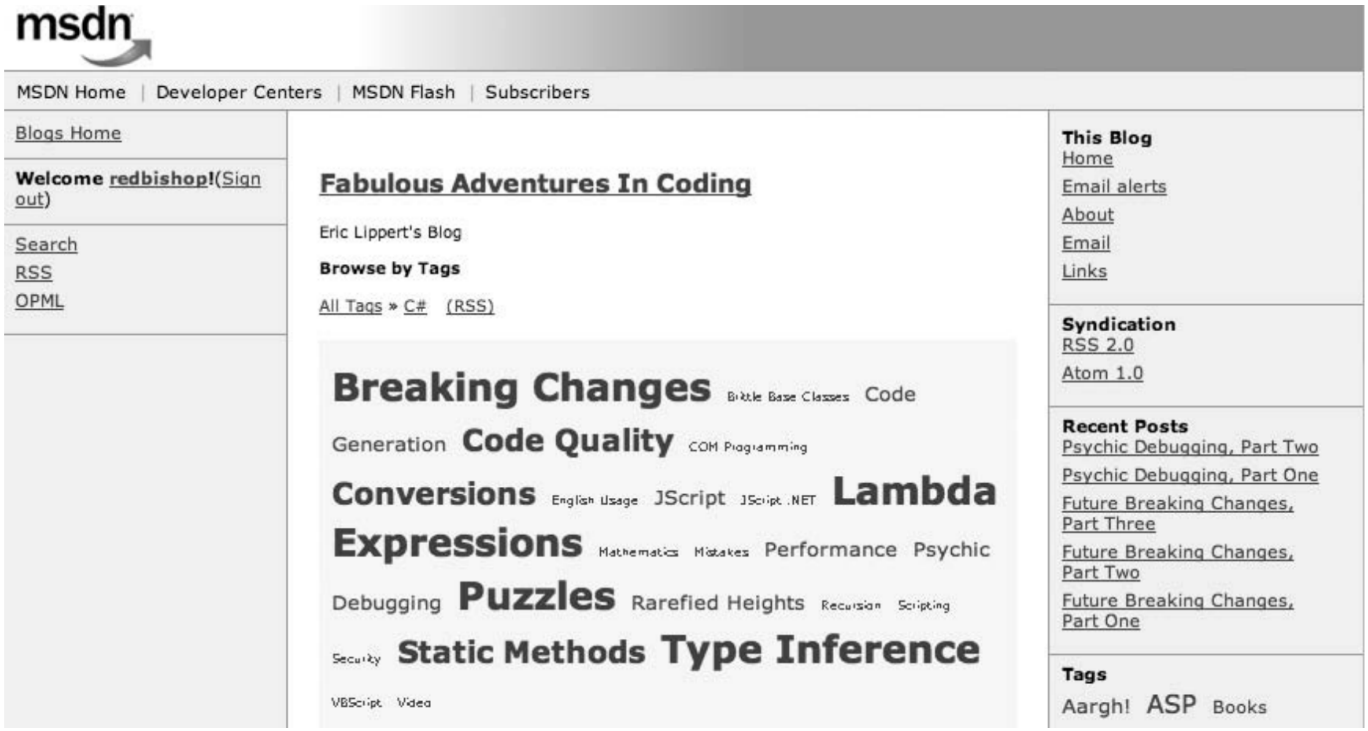
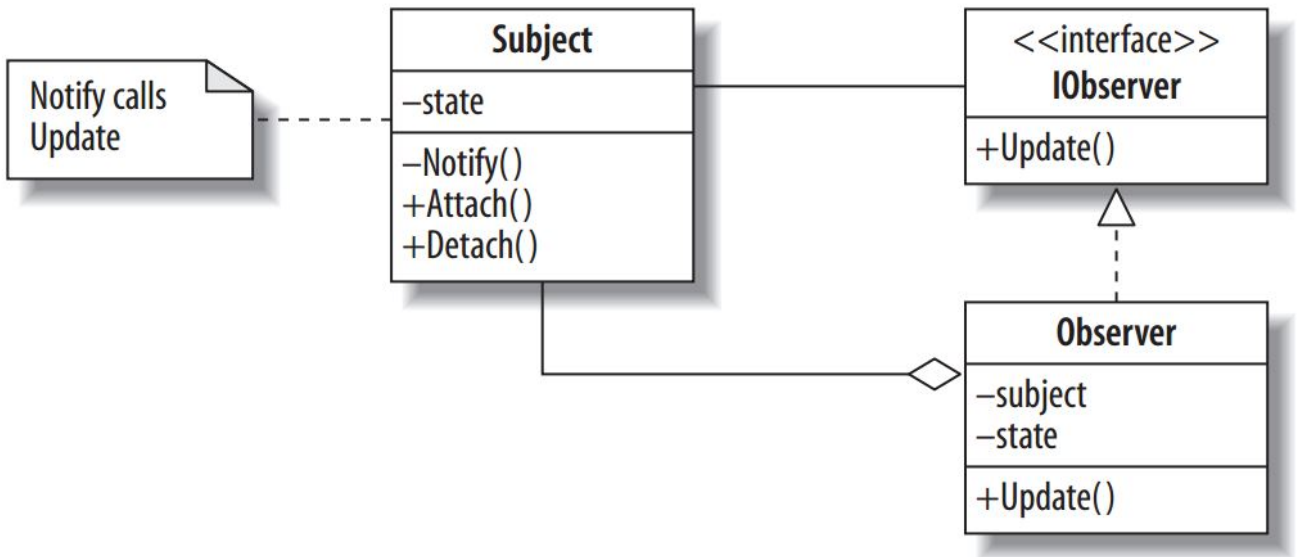


Figure: Observer pattern illustration—a blog site

Observer Design

Like mediator, the design consists of two classes: **Subject** whose objects change their state at an independent rate. **Observers** may indicate that they wish to be informed of these changes, in which case, the Subject will send them notifications.

Observer Design



Observer Pattern Design: Players

- **Subject:** The class whose instances independently change their state and notify Observers
- **IObserver:** An interface for Observers specifying how they should be updated
- **Observer:** A class that provides an Update method to enable its instance's state to stay consistent with the Subject's
- **Update:** The operation that forms the interface between the Subject and the Observers
- **Notify:** The event mechanism for calling the Update operation on all Observers

Observer Pattern Illustration

- **Subject:** A blogger who writes about C#
- **IObserver:** The “Email alert”
- **Observer:** A programmer interested in C#
- **Update:** Receiving an alert by email
- **Notify:** An email from the blog site to all those signed up
- **State:** A blog

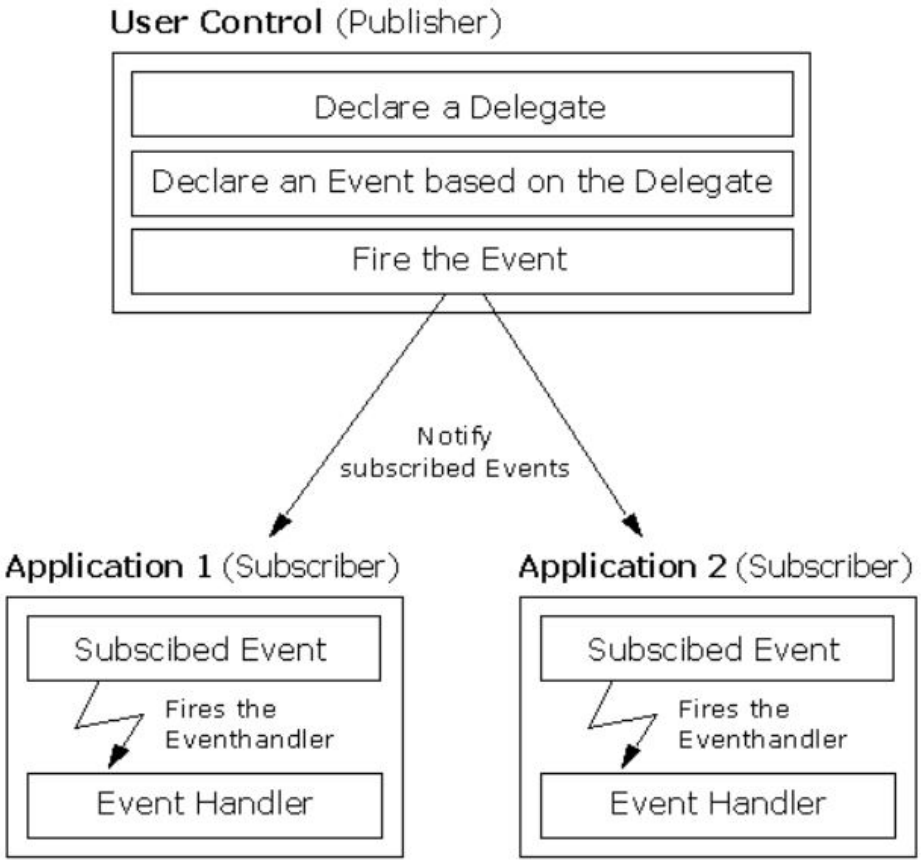
Delegates and Events

- The Event model is used in asynchronous programming
- Publishers publish an event, then send out their event only to subscribers who have subscribed to receive the specific event
- When the publishing class raises an event, all the subscribed applications are notified.
- To declare an event inside a class, first a delegate type for the event must be declared
- The event then is declared, using the *event* keyword
- An **Event** declaration adds a layer of abstraction and protection on the **delegate** instance

Delegates and Events Handlers

- A delegate can be passed as a method parameter.
- An event is raised but cannot be passed as a method parameter.
- EventHandler and EventHandler<TEventArgs> are delegates
- EventHandler delegates are used all events that do not include event data
- EventHandler<TEventArgs> delegate are used for events that include data about the event
- EventHandler<TEventArgs> have no return type value and take two parameters: an object for the source of the event and an object for event data

Event Delegate



Observer Pattern Implementation


```
class Simulator : IEnumerable
{

    string[] moves = { "5", "3", "1", "6", "7" };

    public IEnumerator GetEnumerator()
    {
        foreach (string element in moves)
            yield return element;
    }
}
```

Observer Pattern Implementation


```
class Subject {
  public delegate void Callback (string s);

  public event Callback Notify;

  Simulator simulator = new Simulator();
  const int speed = 200;
  public string SubjectState {get; set;}

  public void Go() {
    new Thread(new ThreadStart(Run)).Start();
  }

  void Run () {
    foreach (string s in simulator) {
      Console.WriteLine("Subject: "+s);
      SubjectState = s;
      Notify(s);
      Thread.Sleep(speed); //millisconds
    }
  }
}
```

Observer Pattern Implementation


```
interface IObserver {
 void Update(string state);
}

class Observer : IObserver {
  string name;
  Subject subject;
  string state;
  string gap;

  public Observer (Subject subject, string name,string gap) {
    this.subject = subject;
    this.name = name;
    this.gap = gap;
    subject.Notify += Update;
  }

  public void Update(string subjectState) {
    state = subjectState;
    Console.WriteLine(gap+name+": "+state);
  }
}
```

Observer Pattern Implementation


```
    static void Main () {
      Subject subject = new Subject();
      Observer Observer =  new Observer(subject,"Center","\t\t");
      Observer observer2 = new Observer(subject,"Right","\t\t\t\t");
      subject.Go();
      Console.ReadKey();
    }
}
```

Observer Pattern Implementation


```
Subject: 5
                Center: 5
                                Right: 5
Subject: 3
                Center: 3
                                Right: 3
Subject: 1
                Center: 1
                                Right: 1
Subject: 6
                Center: 6
                                Right: 6
Subject: 7
                Center: 7
                                Right: 7
```

Observer Pattern USE

- The Observer pattern is widely used in blog sites.
- It has a formal face as the “publish/subscribe” model and the “Model-View-Controller” (MVC) architectural pattern

Iterator, Mediator and Observer Comparison

- Mediator and Observer are very similar patterns. The difference is that the Mediator centralizes the communication between colleagues, whereas the Observer distributes control of information by differentiating between subjects (publishers) and observers (subscribers)
- In the Observer, the coupling is one step removed, in that observers do not send any information; they wait to receive it from subjects.
- Both Mediator and Observer could use Iterator

References

1 C# Design Patterns, Judith Bishop, O'Reilly, 2008