



# Disegno architettuale: Gli idiomi e le linee guida di design per il .NET Framework

**msdn**<sup>®</sup>

Microsoft<sup>®</sup> Developer Network



**GUISA**

Gruppo Utenti Italiani  
Solution Architect

# About me:



**Giancarlo Sudano** (*alias janky*)

○ sono Software Architect in Objectway

(area Microsoft ma non solo)

○ Socio fondatore di GUISA ([www.guisa.org](http://www.guisa.org))

○ Passione per i framework Open Source  
(NHibernate, Spring.NET, CastleProject)

○ Blog su Ugidotnet: <http://blogs.ugidotnet.org/janky>

○ Email: [giancarlo.sudano@gmail.com](mailto:giancarlo.sudano@gmail.com)



# Webcast serie: "Aspire Architect"

- 6/11/2006 – Una introduzione
- 17/11/2006 – Dai requisiti ai casi d'uso
- 22/11/2006 – Dai casi d'uso al modello
- 27/11/2006 – Un approccio agile
- 30/11/2006 – Una introduzione ai Design Pattern
- 06/12/2006 – Pattern by example
- 13/12/2006 – Disegno architettuale, gli idiomi e le linee guida design il .NET Framework
- 20/12/2006 - .NET e gli strumenti

# Agenda

- Astrazione e **Design Idiomatico**
- Design dei **Tipi**
- Design dell'**Estendibilità**
- Design delle **Exception**
- **Design Patterns** con soluzioni idiomatiche
- Guidelines sull'**Uso** di Tipi comuni del .NET Framework

# Architetti e Astrazione

- Architetto è quella persona che ha una particolare sensibilità nell'operazione di astrarre 😊
- In generale è una buona abitudine ma attenzione:  
“**...Abstraction isn't Free...**” (Eric Gunnerson)
- Bisogna tenere conto anche della piattaforma in cui andiamo a sviluppare.

# Design Idiomatico

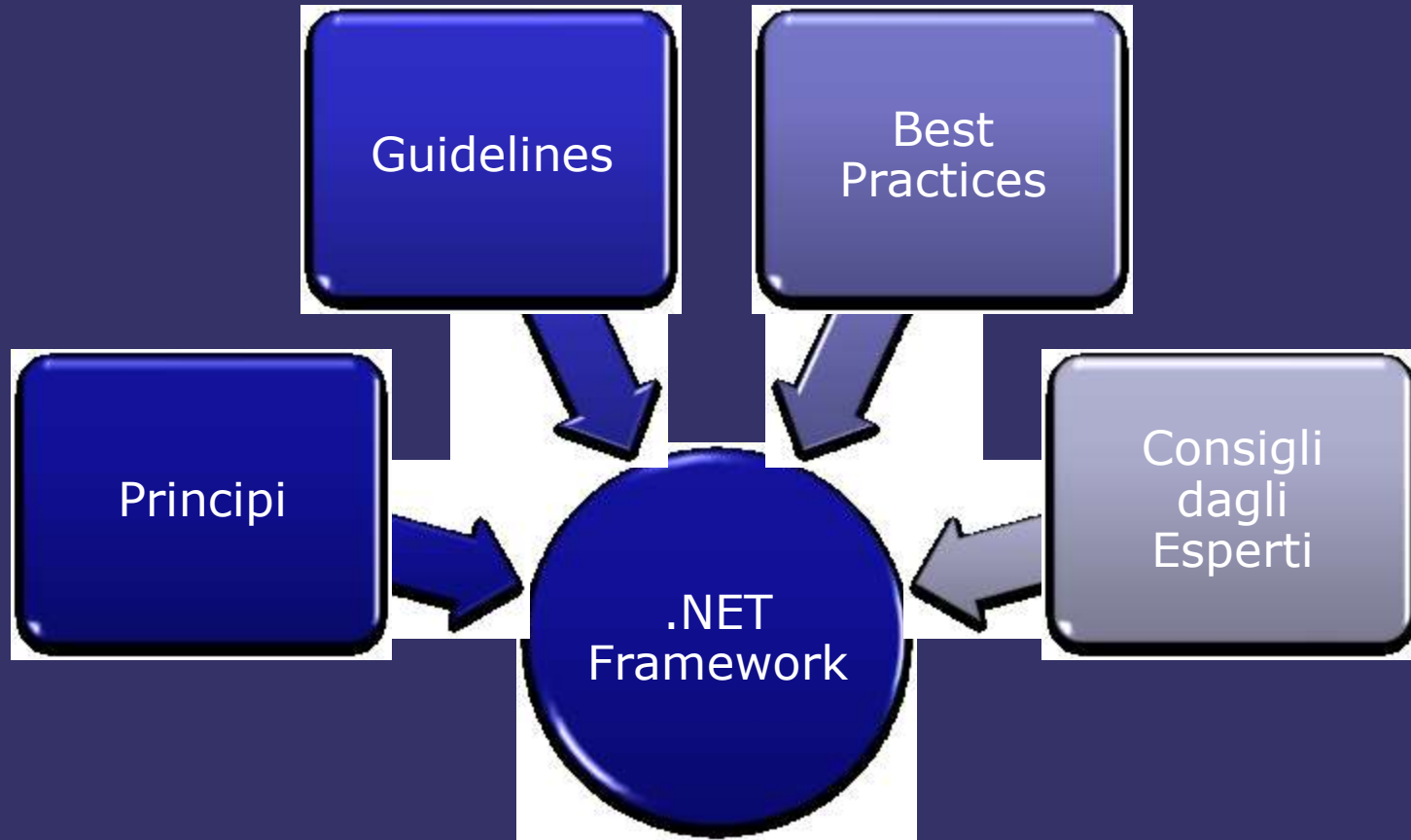
- Definizione data da Koenig [1996]
- E' il Design orientato esplicitamente alla tecnologia/piattaforma utilizzata
- La tecnologia può impattare quindi anche sulle scelte architettoniche

# Design Idiomatiko

- **NUnit** is an excellent example of idiomatic design. Most folks who port xUnit just transliterate the Smalltalk or Java version...
- ...It is written entirely in C# and has been completely redesigned to take advantage of many .NET language features, for example custom attributes and other reflection related capabilities.

[Kent Beck, Homepage di NUnit]

# Design Idiomatico



# Type Design

# Type Design: Categorie di Tipi

CLR

Reference  
Type

Value  
Type

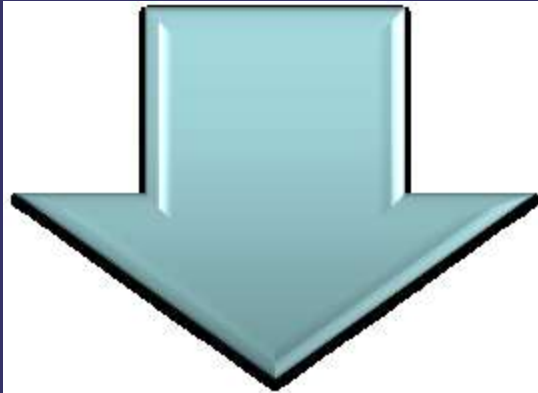
Heap

Stack

# Type Design: Come rappresentare un Tipo?



# Type Design: Class vs Struct



## Class

- Alloc. Dealloc. Costosa
- Gestito dal GC
- Copia per Riferimento
- Overhead cost (richiede +byte)



## Struct

- Alloc./Dealloc. (meno costosa)
- Copia per Valore (costosa)
- Boxing/Unboxing (molto Costoso)



# Type Design: Class vs Struct

Come comportamento di default scegliere **class** per la definizione di un vostro tipo.

Scegliete **struct** solo nei casi in cui:

1. Il vostro tipo ha una rappresentazione in memoria molto piccola (Richter consiglia al di sotto dei 16 byte o leggermente più alta se non si condidera di passarlo attraverso metodi e collezioni)
2. Il vostro tipo è rappresentato logicamente da un singolo valore, come fosse un tipo primitivo
3. Il vostro tipo è **Immutable**

# Type Design: Immutable Types

*Sono dei tipi che non hanno dei membri pubblici che possono modificare lo stato di una istanza.*

**System.String** è immutable.

Ogni operazione effettuata su un tipo string crea una nuova istanza. (es: *miaStringa.ToUpper()* )

Creare tipi Immutable può essere un buon antidoto contro il Side-Effect.

# Type Design: Value Object pattern

*A small simple object, like a money or date range, whose equality is not based on **identity**.*

[Fowler, Pattern of Enterprise Application Architecture, 2002]

# Type Design: Value Object pattern

The key difference between a reference and value object lies in how they deal with equality.

A Reference object uses identity as the basis for equality. This may be the identity within the programming system, such as the built in identity of OO programming languages; or it may be some kind of ID number, such as the primary key in a relational database.

A Value Object bases it's notion of equality on field values within the class. So *two* date objects may be the same if their day, month, and year values are the same.

.NET has a first class treatment of *Value Object*.

*In C# objects are marked as Value Object by declaring them as a struct instead as a class.*

[Fowler, Pattern of Enterprise Application Architecture, 2002]

# Type Design

## Properties vs Methods

# Type Design: props/methods style

- Property-Heavy è normalmente preferibile, soprattutto con un alto numero di parametri, in quando con Method-Heavy si tenderebbe ad avere troppi overload del metodo.
- Method-Heavy è senz'altro più performante (soprattutto in scenari di remoting)
- Method-Heavy lavora meglio in un contesto multi-thread

## // Method-Heavy API style

```
public class MiaClasse
{
    public string Metodo(int param1, int param2, int param3)
}
```

## // Property-Heavy API style

```
public class MiaClasse
{
    public int Param1 { get; set; }
    public int Param2 { get; set; }
    public int Param3 { get; set; }
    public string Metodo()
}
```

# Type Design: Quando usare una Property

- Quando il valore della property è conservato in memoria, e la property serve da mezzo per accedere al valore (esporre lo stato interno della classe)

```
public class Customer
{
    private string companyName;

    public string CompanyName
    {
        get { return this.companyName; }
        set { this.companyName = value; }
    }
}
```

- Le property sono delle smart function, lavorano come funzioni ma hanno la sintassi di un Field.

# Type Design: Quando usare una Property

- Usare property con solo Getter quando il caller non può modificare il valore della property
- Evitare l'uso di property con diversi livelli di accesso tra il getter e il setter.
- Usare dei valori di default per ogni property
- Preservare il valore di un field se la corrispondente property setter ha scatenato un'eccezione.
- Evitare l'uso di eccezioni sollevate dai getter.

# Type Design: Usare un Method quando...

- L'operazione da effettuare è notevolmente più lenta rispetto ad un accesso ad un field
- L'operazione da effettuare è una conversione
- L'operazione non restituisce dei dati in modo deterministico
- L'operazione da eseguire ha un “notevole” side-effect
- L'operazione restituisce una collezione (\*)

# Type Design

Costruttori



# Type Design: Costruttori di Istanza

- Usare i parametri di un costruttore per assegnare corrispondenti properties. In tal caso usare lo stesso nome parametro/property (facendo attenzione al casing).
- Non sovraccaricare troppo il lavoro del costruttore.
- Solleverare eccezioni nel costruttore se il caso lo prevede
- Evitare l'uso di membri virtuali dal codice del costruttore

# Type Design: Costruttori Statici

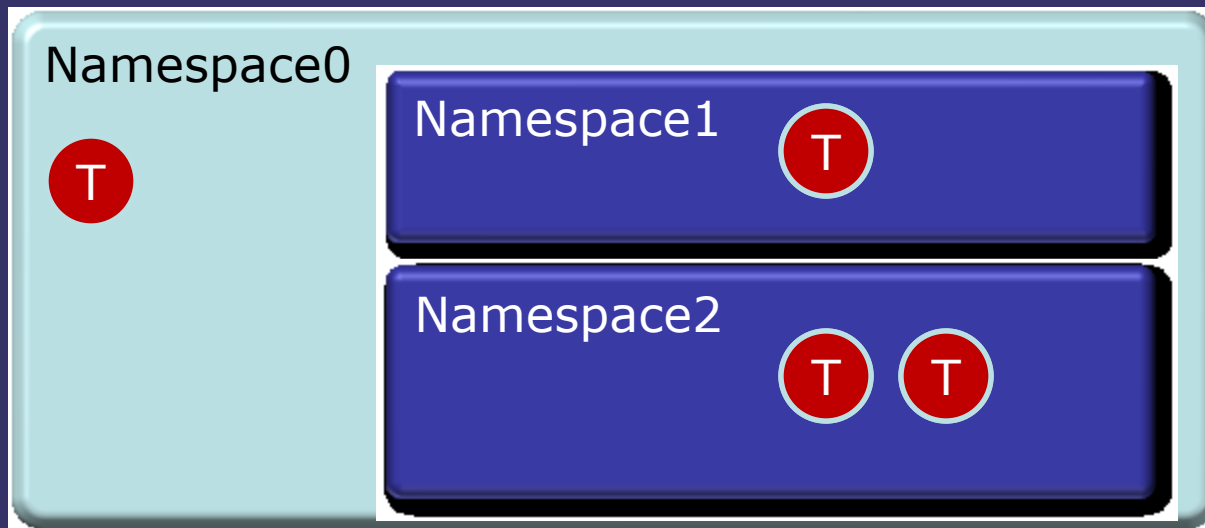
- Usare costruttori statici come privati.
- Non sollevare eccezioni da un costruttore statico.

# Type Design

## Namespaces

# Type Design: Namespace

Il design dei namespace è un processo iterativo. Ci permette di avere degli insiemi coerenti di Tipi collegati funzionalmente.



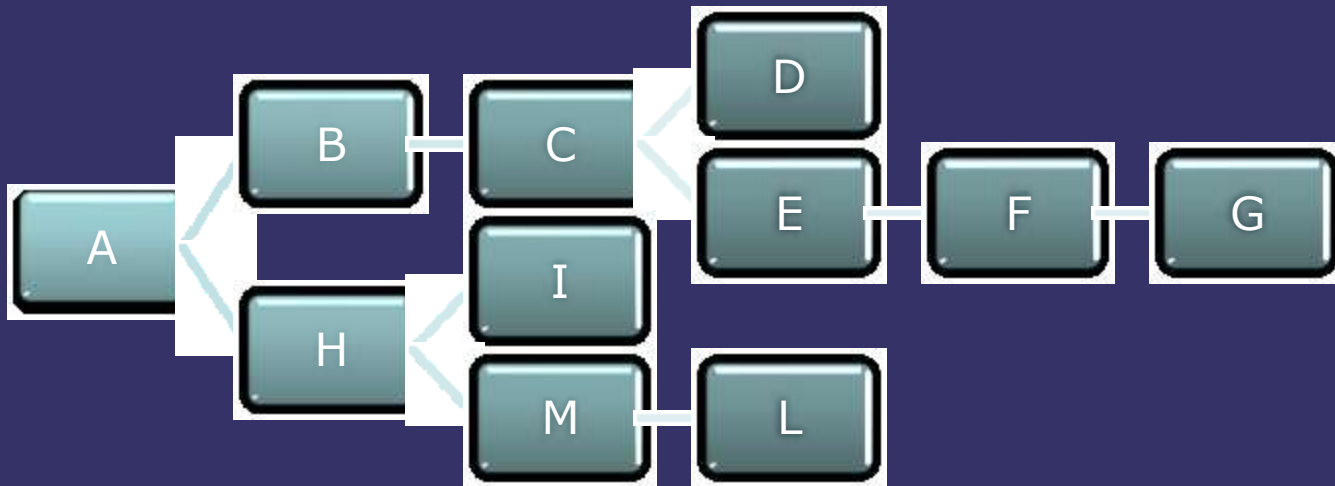
# Type Design: Namespace

“...Al contrario di quello che si crede, i namespace non servono principalmente a risolvere conflitti di nomi, ma ad organizzare i **Tipi** in **gerarchie** coerenti, navigabili, e facili da comprendere...”

[Cwalina]

# Type Design: Namespace

- Evitare gerarchie di namespace troppo profonde.
- Evitare l'uso di “troppi” namespace.



# Type Design: Namespace

Classi specializzate dovrebbero appartenere a namespace più profondi rispetto ai namespace delle classi base.

[Richter]

Esempio:

Nel Framework:

`System.Collection` ( `ArrayList` )

`System.Collection.Specialized` ( `StringCollection` )

# Type Design: Namespace

Controesempio !!! 😊

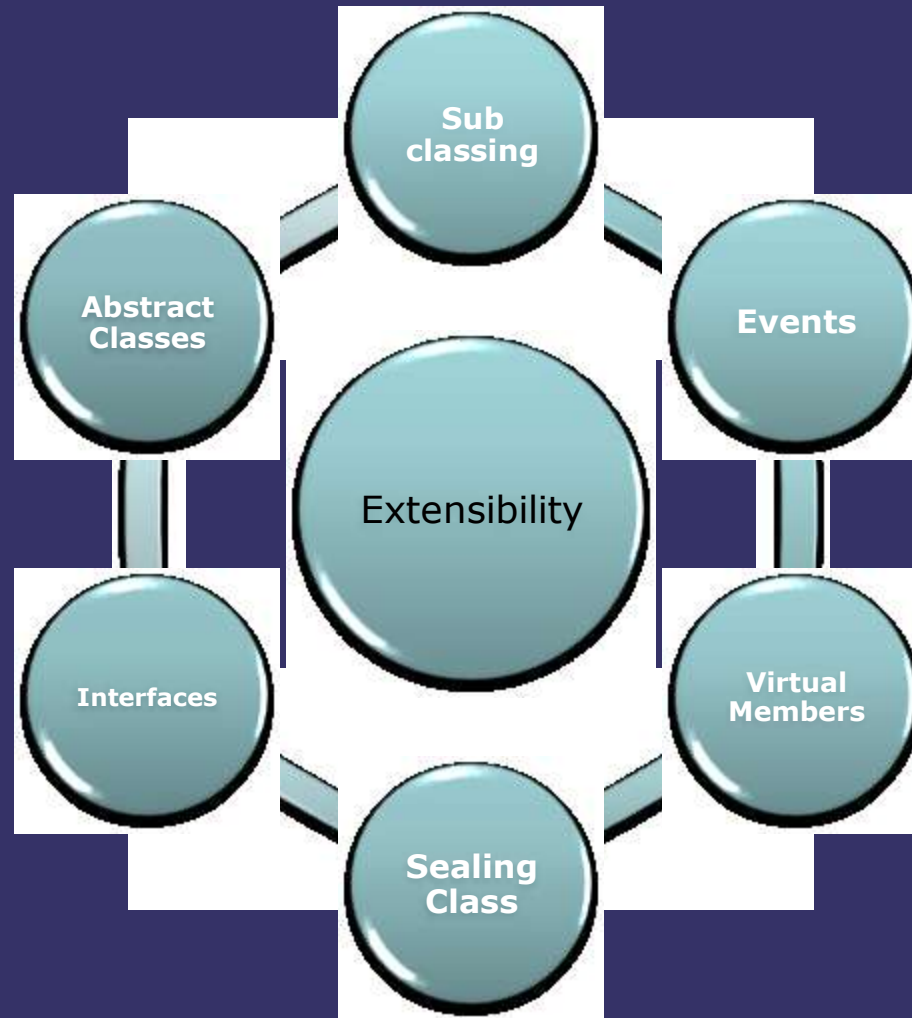
Nel Framework:

**System.Runtime.Serialization** ( Tipi per la serializzazione a Runtime )  
**System** ( Serializable, NonSerializable )

# Design for Extensibility



# Design for Extensibility



# Design for Extensibility

Una considerazione generale:

E' vero che si può sempre estendere il proprio codice, ma a volte questo può causare delle **breaking changes**...

# Design for Extensibility: Unsealed Class

Il meccanismo più efficiente per permettere una estendibilità è sempre

una classe non **sealed**  
senza metodo **virtual**  
senza metodi **protected**

Dipende tutto dal contesto in cui vi trovate.  
Sviluppare un framework è una cosa ben diversa  
da sviluppare un gestionale.

# Design for Extensibility: Unsealed Class

L'uso di metodi **virtual** e property **protected** può comportare uno sforzo maggiore di implementazione per mantenere funzionalità, testabilità e manutenibilità anche nelle classi derivate.

Attenzione!

Una property **protected** classe unsealed è a tutti gli effetti da considerare pubblica.

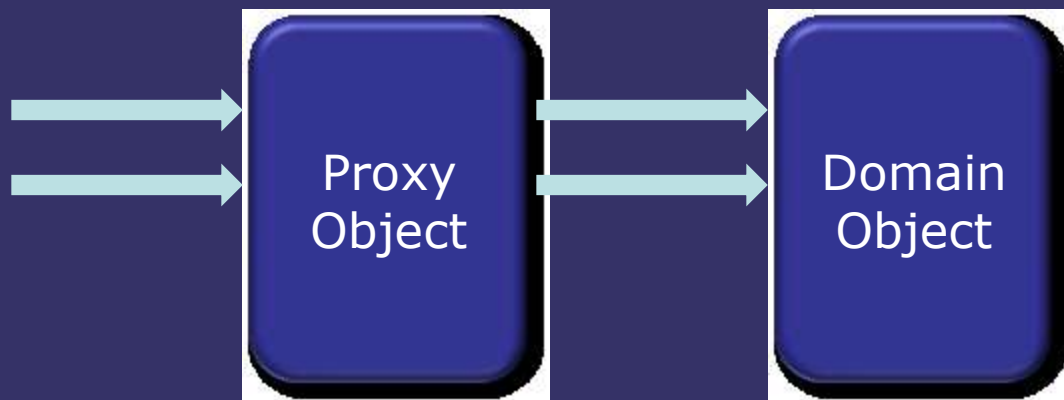
Property **protected** vanno usate per customizzazioni avanzate...ma occhio ai dati sensibili.

# Design for Extensibility: Virtual Member

Non usare metodi virtuali fino a quando non si ha un buon motivo per farlo. Lo scenario di estendibilità deve essere chiaro.

Esempio di uso:

Proxy:

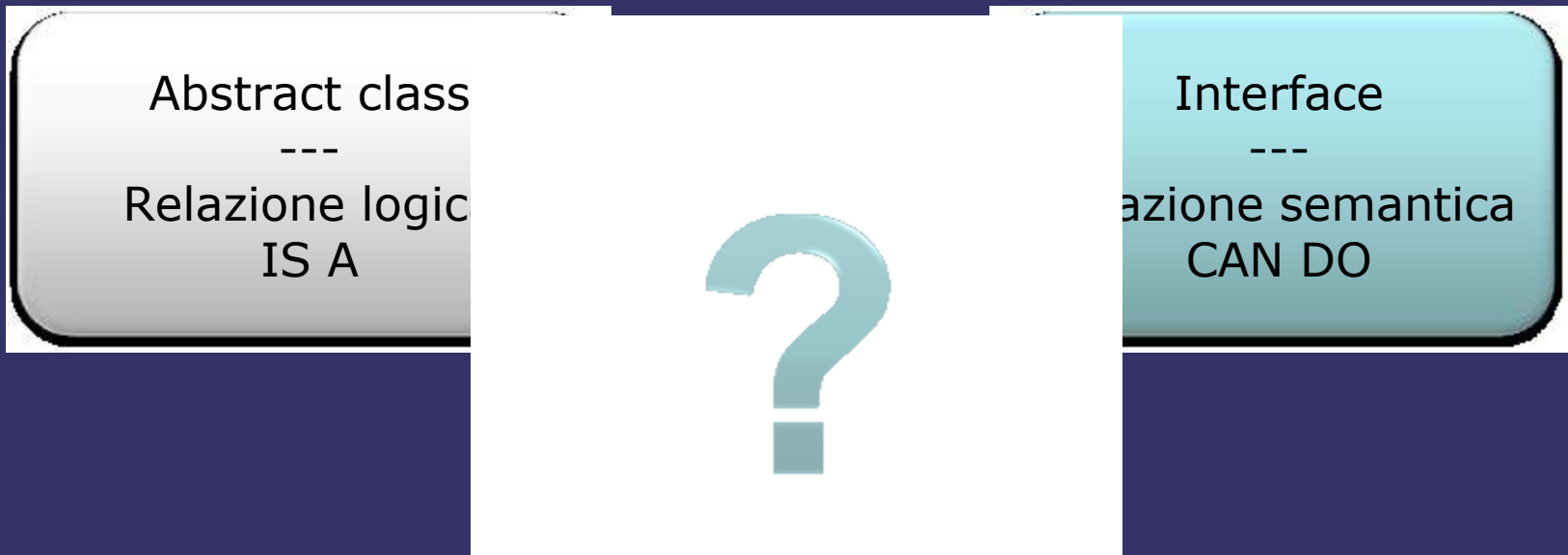


# Design for Extensibility

Abstraction...

# Design for Extensibility: Abstraction

Un astrazione è: un "Tipo" che descrive un contratto, ma non fornisce la sua implementazione



# Design for Extensibility: Abstraction

Favorire l'uso di **abstract class** al posto di **interface** per disaccoppiare contratto e implementazione.

Usare l'implementazione di **interface** per simulare l'ereditarietà multipla.

Non fornire **abstract class** senza distribuire almeno una implementazione concreta

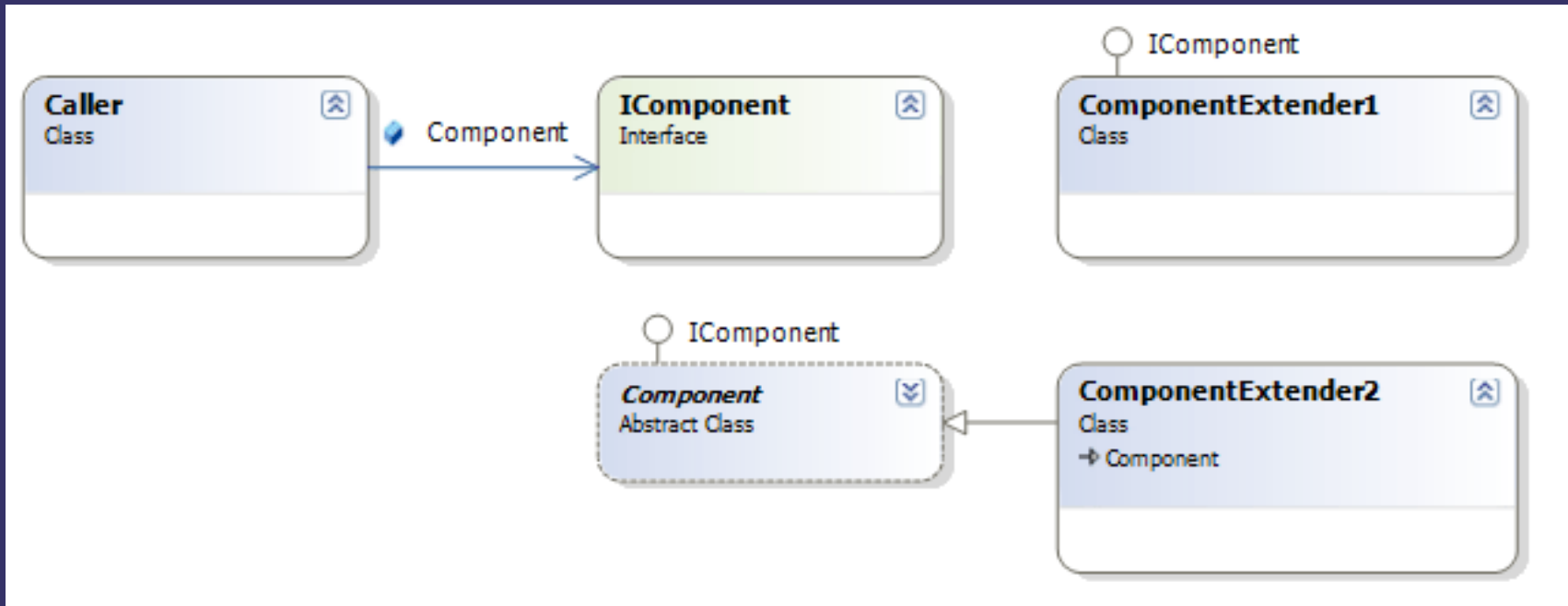
Esempio di nel framework:

Classe astratta: Stream

Classi concrete: FileStream, BinaryStream, etc...

# Design for Extensibility: Abstraction

Meglio ancora, usare tutte e due! 😊



# Exception Design

# Exceptions: quando e come usarle

Non usare Error code!

Non usare le exception per gestire il normale flusso

Semplice regola: Se un metodo non riesce a fare quello per cui gli è stato dato un nome, allora probabilmente si potrebbe sollevare una exception

Cercare di evitare di scatenare una exception da un blocco *finally* {}.

Fare il catch di una eccezione solo quando si può recuperare in modo appropriato dallo stato di errore

Per il codice di cleanup usare **try..finally** e non **try..catch**

# Exceptions: quali usare...

Cercare di usare le exception esistenti sul framework, per quanto possibile (es: ArgumentNullException)

Creare una custom exception solo quando si ha del codice che riesce a gestire in modo differente rispetto alle altre condizioni l'evento specifico

# Design Pattern e soluzioni Idiomatiche



# Design Pattern con soluzioni Idiomatiche

## Dispose Pattern

```
public class DisposableType : IDisposable {  
    private Handle resource;  
  
    public void Dispose() {  
        Dispose(true);  
        GC.SuppressFinalize(this);  
    }  
  
    protected virtual void Dispose(bool disposing) {  
        if (disposing) {  
            if (resource != null) resource.Dispose();  
        }  
    }  
}
```

# Usage Guideline for .NET Framework



# Usage Guideline: Collections

Non usare `ArrayList` o `List<T>` nelle API pubbliche.

Non usare `Hashtable` o `Dictionary<Tkey,Tvalue>` nelle API pubbliche

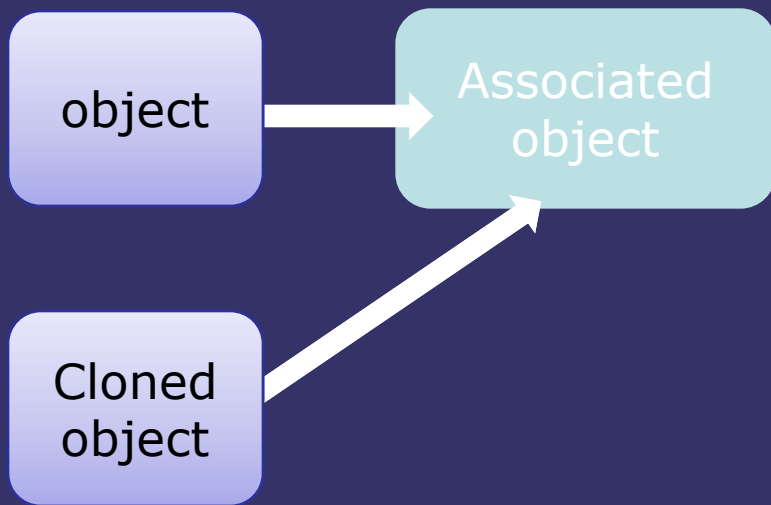
Usare i tipi meno specializzati (`IEnumerable<T>`), eventualmente verificare dinamicamente se la collezione implementa interfacce più specializzate (`ICollection<T>`, `Ilist<T>`)

Se una property è una collection, non fornire il setter

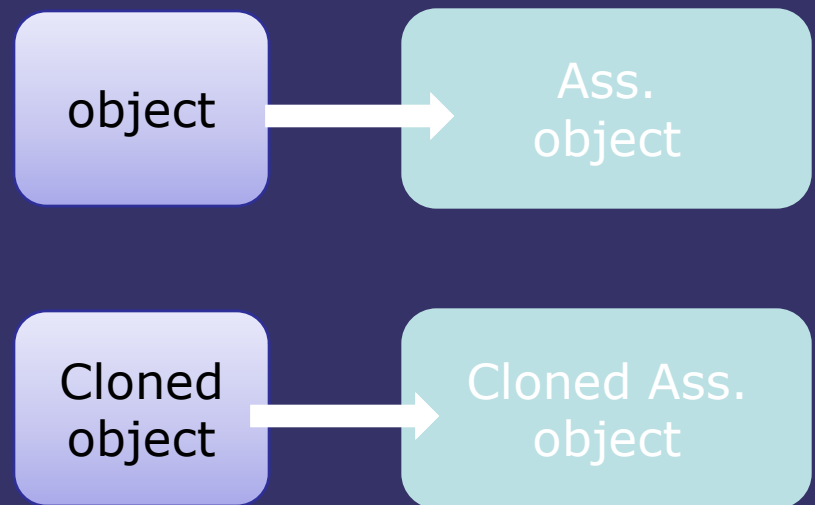
# Usage Guideline: ICloneable

Non implementare mai ICloneable

Non usare ICloneable nelle API pubbliche



Shallow Copy



Deep copy

# Usage Guideline: Equals

Fare l'override di Equals sui ref Type, quando il ref type può rappresentare un Value Type

Fare l'override di GetHashCode assieme a Equals

# Bibliografia

Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries [Cwalina, Abrams]

Il blog di Cwalina: <http://blogs.msdn.com/kcwalina/>

Il blog di Abrams: <http://blogs.msdn.com/brada/>

MSDN, linee guida per la progettazione:

[http://msdn2.microsoft.com/it-it/library/ms229042\(VS.80\).aspx](http://msdn2.microsoft.com/it-it/library/ms229042(VS.80).aspx)



# Visual Studio 2005 è disponibile: scegli il prodotto più giusto per te

[www.microsoft.it/msdn/vs2005/](http://www.microsoft.it/msdn/vs2005/)

## – Visual Studio 2005 Team Edition

- Visual Studio Team Edition (for Architects, Developers o Testers) con MSDN Premium
- Visual Studio Team Suite con MSDN Premium

## – Strumenti professionali

- Visual Studio 2005 Professional
- Visual Studio 2005 Professional con MSDN Professional
- Visual Studio 2005 Professional con MSDN Premium
- Visual Studio 2005 Tools for Microsoft Office System

## – Strumenti di base

- Visual Studio 2005 Standard
- Visual Studio 2005 Express Edition

## – Altri strumenti

- Visual SourceSafe 2005
- VisualFox Pro 9.0

Licenze individuali:  
1 sviluppatore = 1 licenza

Dove acquistare:

[www.microsoft.it/msdn/rivenditori/](http://www.microsoft.it/msdn/rivenditori/)

Per informazioni:

[itamsdn@microsoft.com](mailto:itamsdn@microsoft.com)



Domande?  
(facili facili...)

