



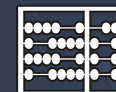
INF0992

Programação Avançada em C#

Aula 3

Hervé Yviquel
hyviquel@unicamp.br

8 de Outubro de 2022





- Introdução
- Padrões
- Classificações de Padrões
- Implementação dos Padrões

The background features a network of gray lines connecting various colored circles (orange, yellow, blue, green) and a dark blue horizontal band with a white circuit-like pattern.

Introdução



- **Projetar software para reuso é difícil**
 - Decomposição do problema e abstração correta
 - Flexibilidade, modularidade e elegância
- **Projetos emergem de um processo iterativo**
 - Tentativas e muitos erros
- **A boa notícia é que bons projetos existem**
 - Tem características recorrentes mas nunca são idênticos
- **Perspectiva de engenharia**
 - Projetos podem ser descritos, codificados ou padronizados?
 - Pode diminuir a fase de tentativa e erro
 - Bons software produzidos mais rápidos

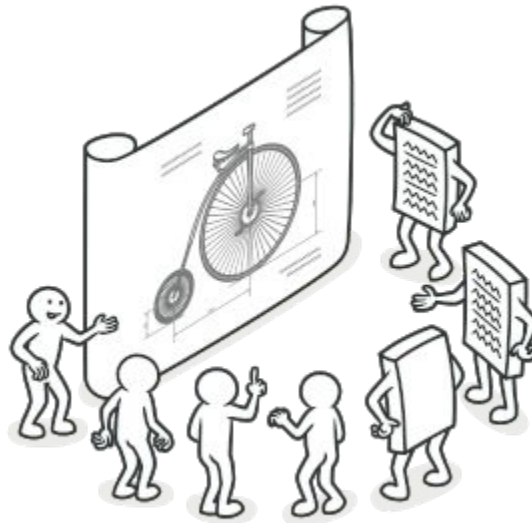


- A idéia de padrões foi apresentada por Christopher Alexander em 1977 no contexto de Arquitetura (de prédios e cidades)
 - A Pattern Language: Towns, Buildings, and Construction
 - Serviu de inspiração para os desenvolvedores de software

“Cada padrão descreve um problema que ocorre repetidamente de novo e de novo em nosso ambiente, e então descreve a parte central da solução para aquele problema de uma forma que você pode usar esta solução um milhão de vezes, sem nunca implementá-la duas vezes da mesma forma.”



- Padrões de Projeto de Software Orientado a Objetos
- Também conhecidos como
 - Padrões de Desenho de Software OO
 - ou simplesmente como Padrões





- Um padrão encerra o conhecimento de uma pessoa muito experiente em um determinado assunto de uma forma que este conhecimento pode ser transmitido para outras pessoas menos experientes.
 - Outras ciências (p.ex. química) e engenharias possuem catálogos de soluções.
- Desde 1995, o desenvolvimento de software passou a ter o seu primeiro catálogo de soluções para projeto de software
 - O livro GoF

Gang of Four (GoF)



- Passamos a ter um vocabulário comum para conversar sobre projetos de software
 - Soluções que não tinham nome passaram a ter nome
 - **Antes:** descrição dos sistemas em termos de pilhas, filas, árvores e listas ligadas,
 - **Depois:** descrição de muito mais alto nível como Fábricas, Fachadas, Observador, Estratégia, etc.
- A maioria dos autores eram entusiastas de Smalltalk
 - Principalmente o Ralph Johnson
 - Mas acabaram baseando o livro em C++ para que o impacto junto à comunidade de CC fosse maior
 - O impacto foi enorme, o livro vendeu centenas de milhares de cópias.



Padrões

O Formato de um padrão



- Todo padrão inclui
 - Nome
 - Problema
 - Solução
 - Consequências / Forças

O Formato dos Padrões no GoF



1. Nome (inclui número da página)
 - Um bom nome é essencial para que o padrão caia na boca do povo
2. Objetivo / Intenção (ou Motivação)
 - Um cenário mostrando o problema e a necessidade da solução
3. Aplicabilidade
 - Como reconhecer as situações nas quais o padrão é aplicável
4. Estrutura
 - Representação gráfica da estrutura de classes do padrão
5. Participantes
 - As classes e objetos que participam e quais são suas responsabilidades
6. Colaborações
 - Como os participantes colaboram para exercer as suas responsabilidades
7. Conseqüências
 - Vantagens e desvantagens, trade-offs
8. Implementação
 - Com quais detalhes devemos nos preocupar quando implementamos o padrão
 - Aspectos específicos de cada linguagem
9. Exemplo de Código



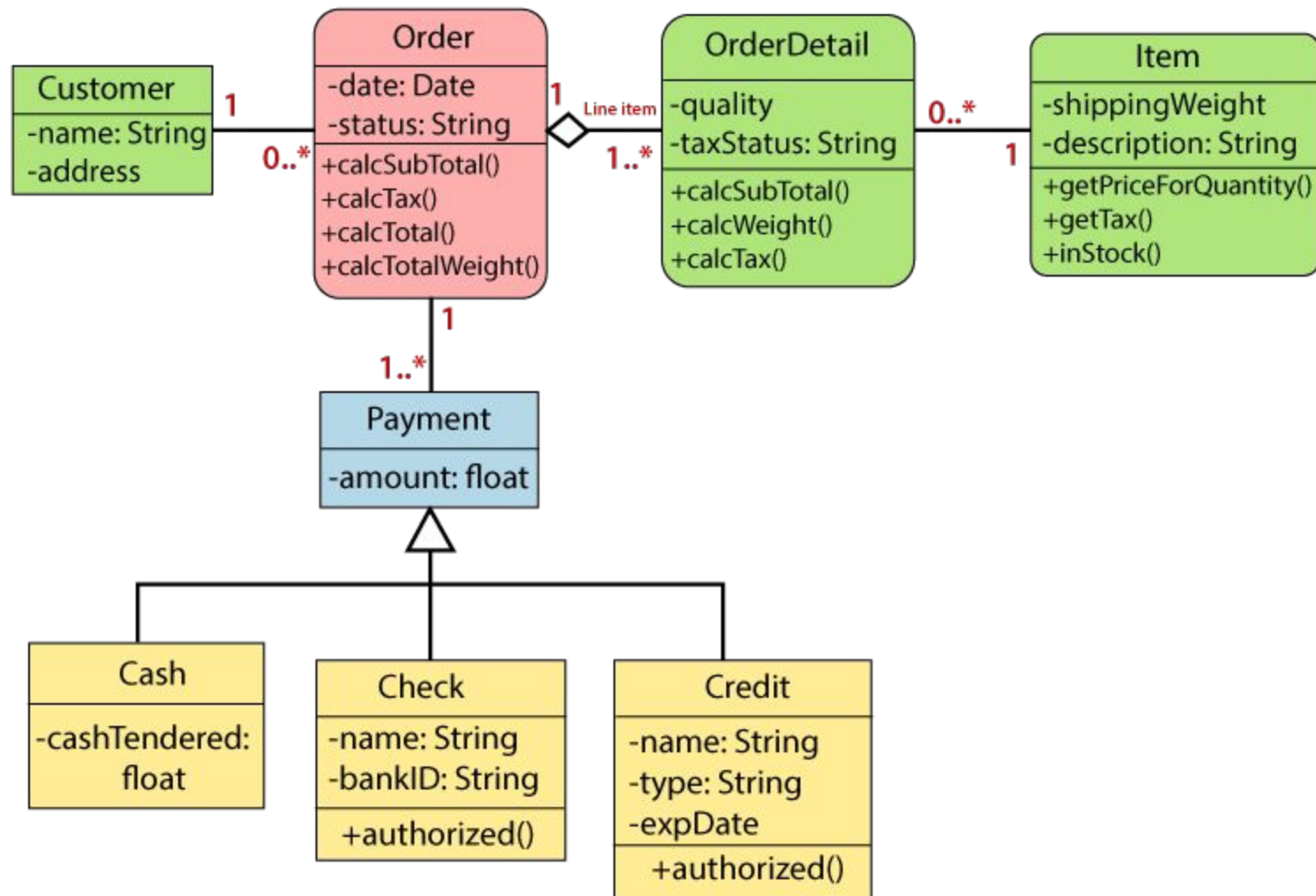
- **Classe**
 - Descrição de um conjunto de objetos
- **Objetos**
 - Instâncias de classes
- **Métodos**
 - Procedimentos que realizam as ações próprias do objeto
- **Atributos**
 - Propriedades das classes
 - Um atributo é uma variável que pertence a um objeto
 - Os dados de um objeto são armazenados nos seus atributos
- **Interface**
 - Contrato entre a classe e o mundo exterior

Uma breve introdução à UML



- **Unified Modeling Language**
 - Especificação do OMG (Object Management Group)
- Linguagem de modelagem de software orientado objetos
 - Estrutura, comportamento, arquitetura, processos de negócios, estruturas de gestão de dados, etc
- Múltiplas notações para vários diagramas
 - Diagrama de caso de uso
 - **Diagrama de classes**
 - Diagrama de sequência
 - Diagrama de Atividades
 - Diagrama de estados

Diagrama de Classe



The background features a network of gray lines connecting various colored circles (orange, yellow, blue, green) and a dark blue horizontal band with a white circuit-like pattern.

Classificação de Padrões



- Todos os padrões podem ser categorizados por sua intenção ou propósito
- **Padrões de Criação**
 - fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização do código existente
- **Padrões Estruturais**
 - explicam como montar objetos e classes em estruturas maiores, mantendo essas estruturas flexíveis e eficientes
- **Padrões Comportamentais**
 - cuidam da comunicação eficaz e da atribuição de responsabilidades entre os objetos



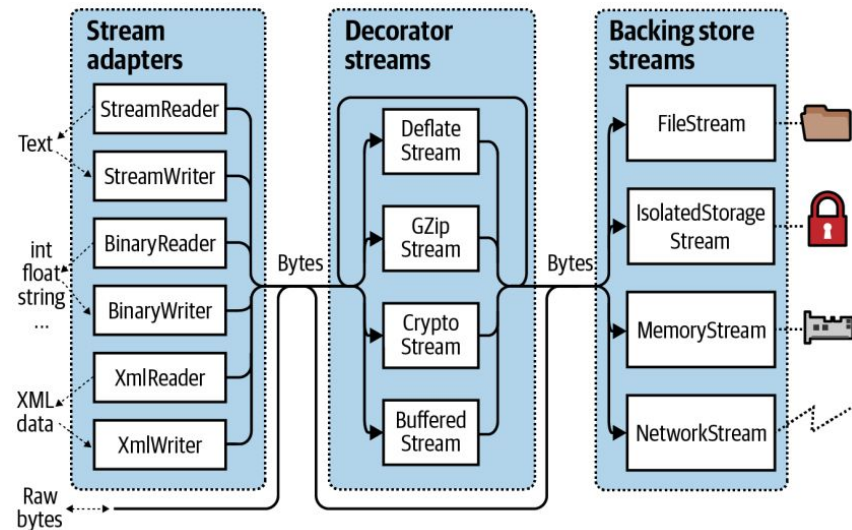
- Processo de criação de Objetos
 - Abstract Factory
 - Builder
 - Factory Method
 - Prototype
 - **Singleton**



- Composição de classes ou objetos

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

Usam implementações para Stream





- Forma na qual classes ou objetos interagem e distribuem responsabilidades
 - Chain of Responsibility
 - Command
 - Interpreter
 - Iterator
 - Mediator
 - Memento
 - **Observer**
 - State
 - **Strategy**
 - Template Method
 - **Visitor**

A stylized logo consisting of five colored circles (blue, orange, yellow, green, and grey) connected by white lines on a dark blue background. The lines form a network-like structure, with some segments highlighted in blue, orange, and green.



The background features a network of gray lines connecting various colored circles (orange, yellow, blue, green) and a dark blue horizontal band with a circuit-like pattern.

Implementação dos Padrões



- Garante que uma classe tenha apenas uma instância
 - e forneça um ponto global de acesso a ela

```
public class Singleton
{
    static Singleton instance;

    // Constructor is 'protected'

    protected Singleton()
    {
    }

    public static Singleton Instance()
    {
        // Uses lazy initialization.
        // Note: this is not thread safe.
        if (instance == null)
        {
            instance = new Singleton();
        }

        return instance;
    }
}
```

Método e atributo
estáticos

Singleton
-instance : Singleton
-Singleton()
+Instance() : Singleton

```
// Constructor is protected -- cannot use new

Singleton s1 = Singleton.Instance();
Singleton s2 = Singleton.Instance();

// Test for same instance

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


Exemplo de Singleton no Mundo Real



```
public class LoadBalancer
{
    static LoadBalancer instance;
    List<string> servers = new List<string>();
    Random random = new Random();

    // Lock synchronization object
    private static object locker = new object();

    // Constructor (protected)
    protected LoadBalancer()
    {
        // List of available servers
        servers.Add("ServerI");
        servers.Add("ServerII");
        servers.Add("ServerIII");
        servers.Add("ServerIV");
        servers.Add("ServerV");
    }

    public static LoadBalancer GetLoadBalancer()
    {
        // Support multithreaded applications
        if (instance == null)
        {
            lock (locker)
            {
                if (instance == null)
                {
                    instance = new LoadBalancer();
                }
            }
        }

        return instance;
    }

    // Simple, but effective random load balancer
    public string Server
    {
        get
        {
            int r = random.Next(servers.Count);
            return servers[r].ToString();
        }
    }
}
```

```
public class Program
{
    public static void Main(string[] args)
    {
        LoadBalancer b1 = LoadBalancer.GetLoadBalancer();
        LoadBalancer b2 = LoadBalancer.GetLoadBalancer();
        LoadBalancer b3 = LoadBalancer.GetLoadBalancer();
        LoadBalancer b4 = LoadBalancer.GetLoadBalancer();

        // Same instance?
        if (b1 == b2 && b2 == b3 && b3 == b4)
        {
            Console.WriteLine("Same instance\n");
        }

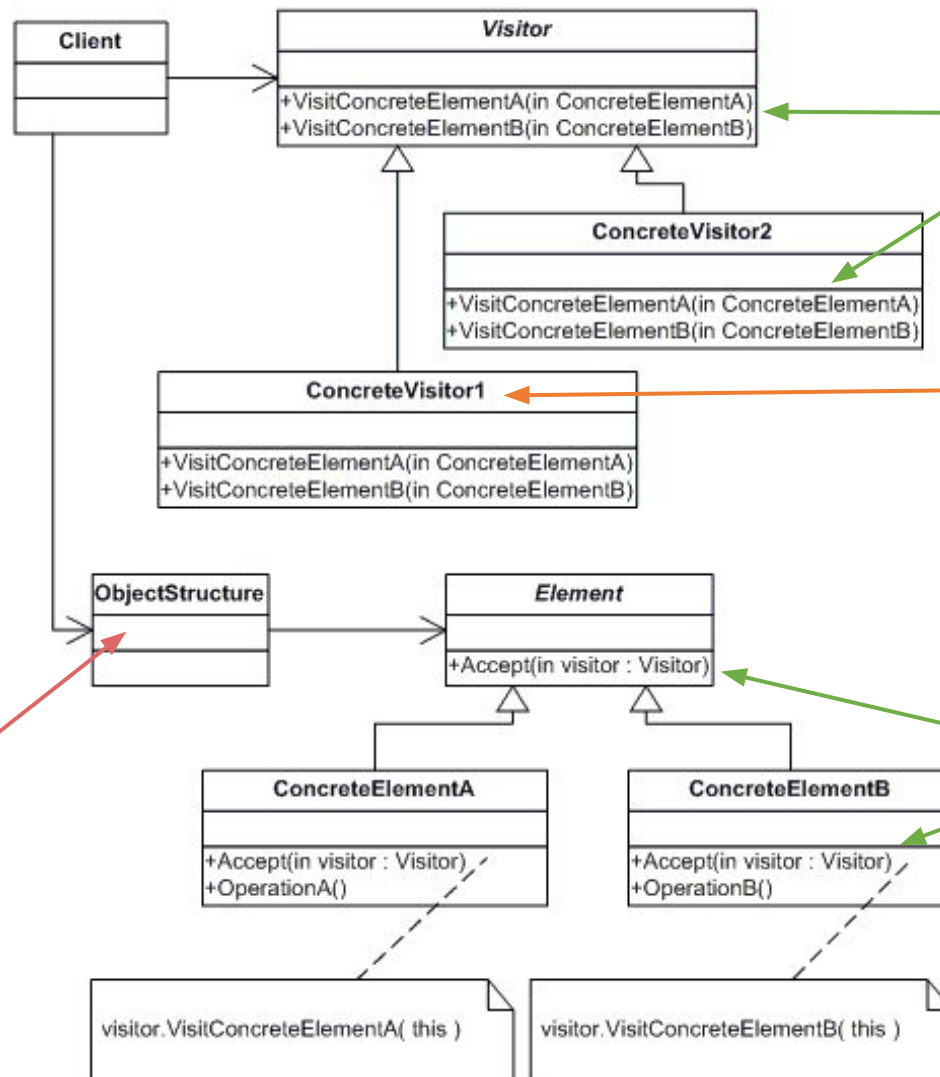
        // Load balance 15 server requests
        LoadBalancer balancer = LoadBalancer.GetLoadBalancer();
        for (int i = 0; i < 15; i++)
        {
            string server = balancer.Server;
            Console.WriteLine("Dispatch Request to: " + server);
        }

        // Wait for user
        Console.ReadKey();
    }
}
```



- Representa uma operação a ser executada nos elementos de uma estrutura de objeto
 - permite definir uma nova operação sem alterar as classes dos elementos nos quais ela opera

Diagrama de Classe do Visitor



Um método *Visit* para cada tipo de elemento

O ConcreteVisitor fornece o contexto para o algoritmo e armazena seu estado local.

A estrutura para visitar

Método *Accept* com o Visitor em argumento

Implementação do Visitor



```
public abstract class Visitor
{
    public abstract void VisitConcreteElementA(
        ConcreteElementA concreteElementA);
    public abstract void VisitConcreteElementB(
        ConcreteElementB concreteElementB);
}

public class ConcreteVisitor1 : Visitor
{
    public override void VisitConcreteElementA(
        ConcreteElementA concreteElementA)
    {
        Console.WriteLine("{0} visited by {1}",
            concreteElementA.GetType().Name, this.GetType().Name);
    }
    public override void VisitConcreteElementB(
        ConcreteElementB concreteElementB)
    {
        Console.WriteLine("{0} visited by {1}",
            concreteElementB.GetType().Name, this.GetType().Name);
    }
}
```

```
public abstract class Element
{
    public abstract void Accept(Visitor visitor);
}

public class ConcreteElementA : Element
{
    public override void Accept(Visitor visitor)
    {
        visitor.VisitConcreteElementA(this);
    }
    public void OperationA()
    {
        // ...
    }
}
```

```
public static void Main(string[] args)
{
    // Setup structure
    ObjectStructure o = new ObjectStructure();
    o.Attach(new ConcreteElementA());
    o.Attach(new ConcreteElementB());
    // Create visitor objects
    ConcreteVisitor1 v1 = new ConcreteVisitor1();
    ConcreteVisitor2 v2 = new ConcreteVisitor2();
    // Structure accepting visitors
    o.Accept(v1);
    o.Accept(v2);
    // Wait for user
    Console.ReadKey();
}
```

ConcreteElementA visited by ConcreteVisitor1
ConcreteElementB visited by ConcreteVisitor1
ConcreteElementA visited by ConcreteVisitor2
ConcreteElementB visited by ConcreteVisitor2

Podem chamar
accept de
elementos filhos
dentro dos **visit**

Visitor
hierárquico



- Define uma dependência um-para-muitos entre objetos para que
 - quando um objeto muda de estado
 - todos os seus dependentes sejam notificados e atualizados automaticamente

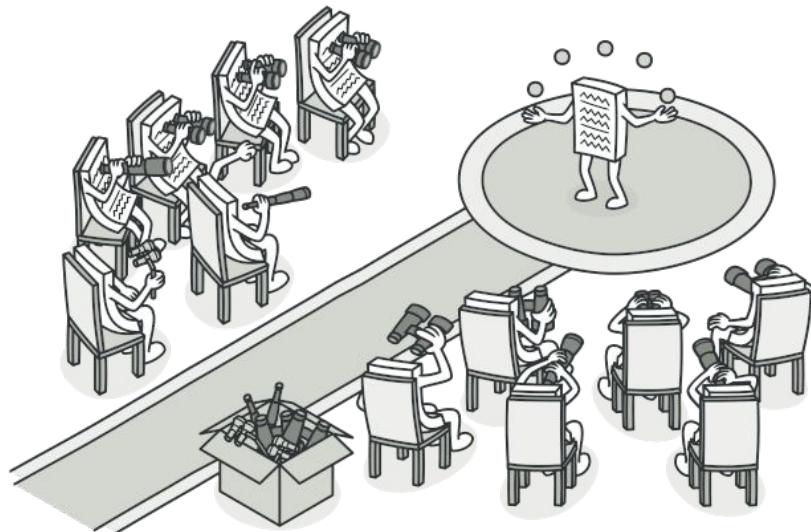
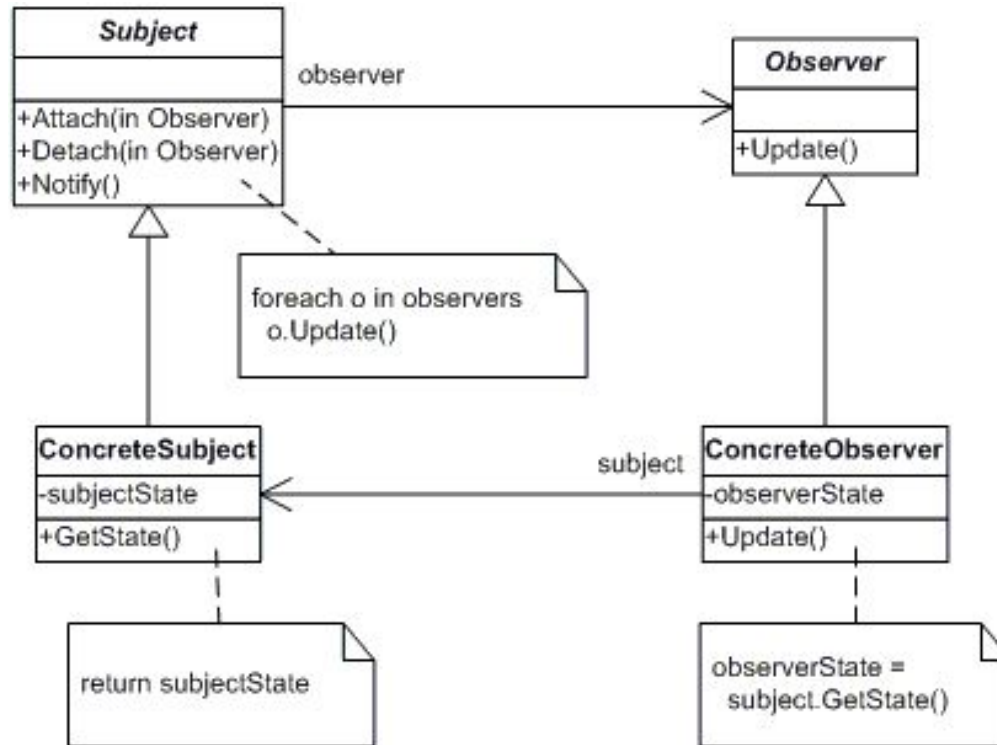


Diagrama de Classe do Observer



Implementação do Observer



```
/// The 'Subject' abstract class
public abstract class Stock
{
    private double price;
    private var investors = new List<IInvestor>();

    public Stock(string symbol, double price) {
        this.symbol = symbol;
        this.price = price;
    }

    public void Attach(IInvestor investor) {
        investors.Add(investor);
    }

    public void Detach(IInvestor investor) {
        investors.Remove(investor);
    }

    public void Notify()
    {
        foreach (var investor in investors)
            investor.Update(this);
        Console.WriteLine("");
    }

    public double Price {
        get { return price; }
        set {
            if (price != value) {
                price = value;
                Notify();
            }
        }
    }
}
```

Chama Notify quando tem mudanças

```
/// The 'Observer' interface
public interface IInvestor {
    void Update(Stock stock);
}

/// The 'ConcreteObserver' class
public class Investor : IInvestor {
    private string name;
    private Stock stock;

    public Investor(string name) {
        this.name = name;
    }

    public void Update(Stock stock) {
        Console.WriteLine("Notified {0} of {1}'s "
            + "change to {2:C}", name, stock.Symbol,
            stock.Price);
    }

    // Gets or sets the stock
    public Stock Stock {
        get { return stock; }
        set { stock = value; }
    }
}
```

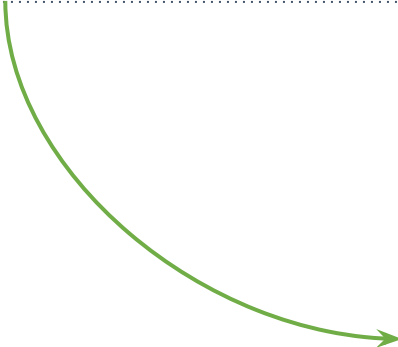
Os observers podem ser adicionados/removidos durante a execução

Notify chama o Update dos Observers

Implementação do Observer (2)



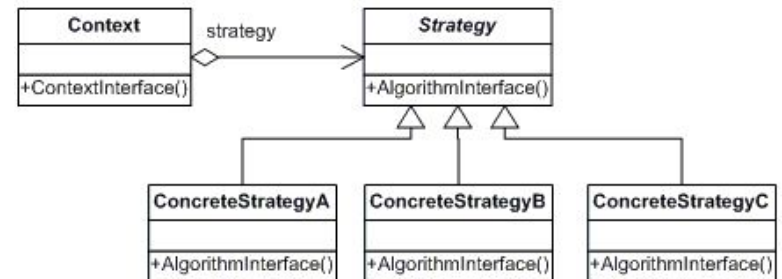
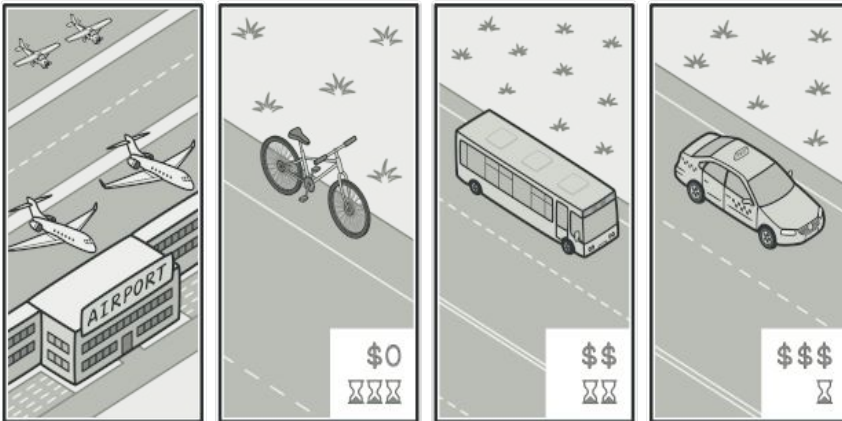
```
public static void Main(string[] args) {  
    // Create IBM stock and attach investors  
    IBM ibm = new IBM("IBM", 120.00);  
    ibm.Attach(new Investor("Sorros"));  
    ibm.Attach(new Investor("Berkshire"));  
    // Fluctuating prices will notify investors  
    ibm.Price = 120.10;  
    ibm.Price = 121.00;  
    ibm.Price = 120.50;  
    ibm.Price = 120.75;  
    // Wait for user  
    Console.ReadKey();  
}
```



```
Notified Sorros of IBM's change to $120.10  
Notified Berkshire of IBM's change to $120.10  
  
Notified Sorros of IBM's change to $121.00  
Notified Berkshire of IBM's change to $121.00  
  
Notified Sorros of IBM's change to $120.50  
Notified Berkshire of IBM's change to $120.50  
  
Notified Sorros of IBM's change to $120.75  
Notified Berkshire of IBM's change to $120.75
```



- Define uma família de algoritmos
 - encapsula cada um deles e os torna intercambiáveis
- Esse padrão permite que o algoritmo varie independentemente dos clientes que o utilizam



...e os outros pattern?



- Há muitos sites (e livros) que descrever todos os patterns
 - Refactoring Guru
 - <https://refactoring.guru/design-patterns/>
 - DoFactory – C# Design Patterns
 - <https://www.dofactory.com/net/design-patterns>
- Pode ter várias maneiras de implementar um padrão
 - em particular com a evolução das linguagens de programação
 - por exemplo, esses 5 tipos de Visitor em C#
<https://www.codeproject.com/Articles/5326263/Visitor-Pattern-in-Csharp-5-Versions>
- Há 23 padrões descritos no GoF mas outros foram criados depois



- Truques para uma linguagem de programação fraca
 - Normalmente, a necessidade de padrões surge quando escolhem uma linguagem que não possui o nível de abstração necessário
- Soluções ineficientes
 - Padrões tentam sistematizar abordagens que já são amplamente utilizadas
 - Essa unificação é vista por muitos como um dogma, e eles implementam padrões “à letra”, sem adaptá-los ao contexto de seu projeto
- Uso injustificado
 - Problema de muitos novatos que acabaram de se familiarizar com os padrões
 - Tentam aplicá-los em todos os lugares, mesmo em situações em que um código mais simples funcionaria bem



- GoF (Gang of Four) book
 - E. Gamma and R. Helm and R. Johnson and J. Vlissides. Design Patterns - Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- Curso da Profa. Thienne Johnson (USP)
 - Padrões de Projeto de Software Orientado a Objetos
- Refactoring Guru
 - <https://refactoring.guru/design-patterns/>
 - As imagens vêm deste site
- DoFactory – C# Design Patterns
 - <https://www.dofactory.com/net/design-patterns>
 - Os exemplos vêm deste site



Curso de Extensão Tecnologias Microsoft



INF0992

Obrigado !!
Merci !!

Hervé Yviquel
hyviquel@unicamp.br

