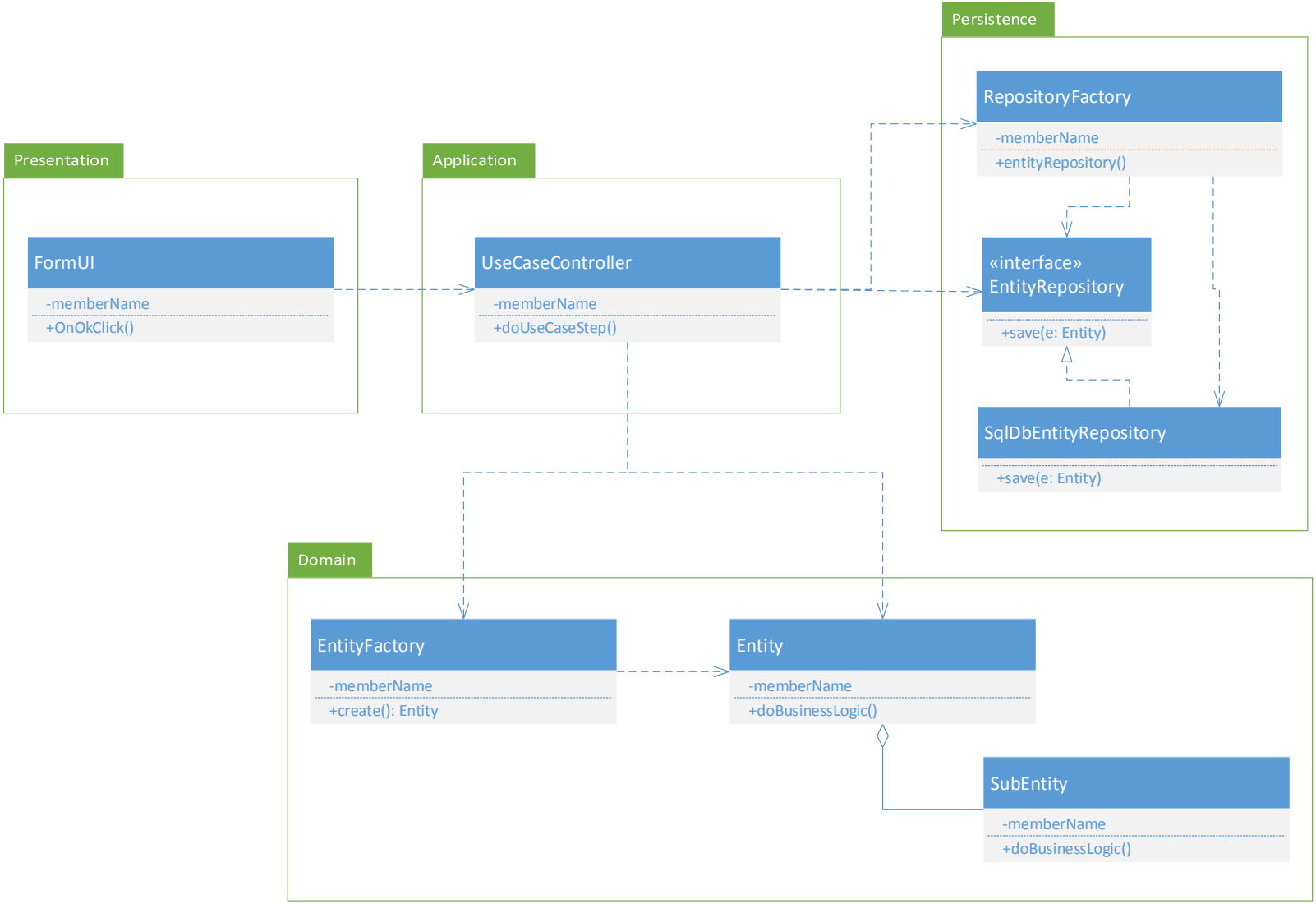
EAPLI

Princípios de Design OO: Extensão & modificação

Paulo Gandra de Sousa
pag@isep.ipp.pt

Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	Persistence Ignorance Entity Value Object Domain Service Aggregate Domain Event Observer
How to handle an object's lifecycle?	Factories - Factory method - Simple factory - Abstract factory Repositories
How to prepare the code for modification?	

Sterotypical architecture



Como preparar o código para modificação?

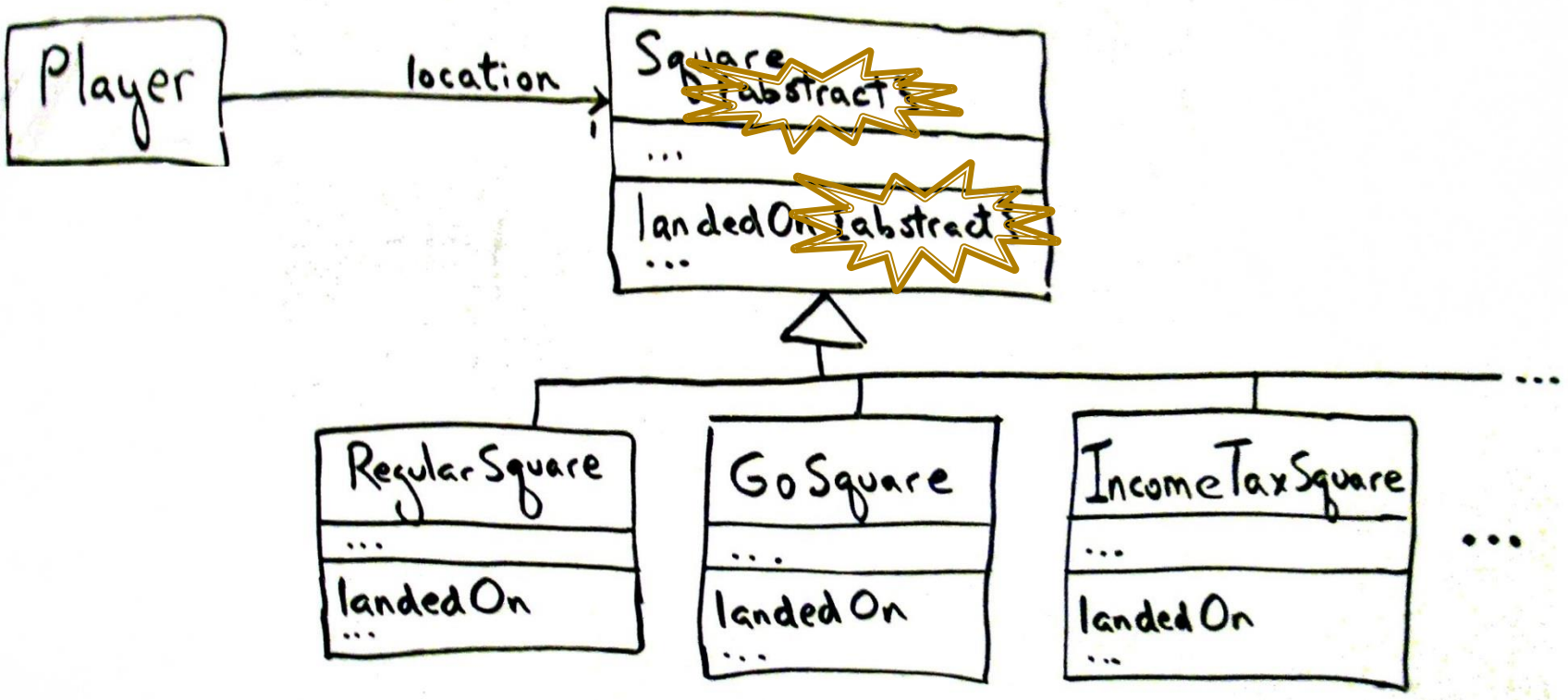
Polymorphism

GRASP

- **Problema:**
  - Como tratar alternativas baseadas em tipos (classes)? Como construir componentes de software substituíveis?
- **Solução:**
  - quando alternativas ou comportamento relacionados variam em função do tipo, deve-se utilizar operações polimórficas.

Polymorphism (exemplo 1)

- Onde aplicar o polimorfismo no Monopólio?



Liskov Substitution Principle

SOLID

- Qualquer método que espere um objeto de um tipo A deve poder trabalhar com qualquer objeto derivado de A

Corolário do Liskov Substitutio Principle

SOLID

- Classes derivadas devem cumprir o mesmo contrato (**sem alterações semânticas**) da classe base

Dependency Inversion

SOLID

- Deve depender-se de abstrações e não de concretizações
  - Ex., List vs. ArrayList
  - Programar para uma interface
- Criar uma camada de abstração que diminuirá o acoplamento entre módulos

Protected Variation

GRASP

- **Problema**:
  - como desenhar objectos, componentes e sistemas de modo a que variações nesses elementos não tenham impacto indesejável noutros elementos?
- **Solução:**
  - Identificar previsíveis pontos de variação. Atribuir responsabilidades de modo a criar uma interface estável à sua volta.

Open/Close

SOLID

- Uma classe deve ser aberta (*open*) para extensão mas fechada (*close*) para modificação

- Novos requisitos e comportamentos devem ser obtidos através da extensão da classe e não da sua modificação
- Ao criar a classe identificar (possíveis) pontos de futura variabilidade e desenhar a classe para poder ser estendida nesses pontos

Template Method

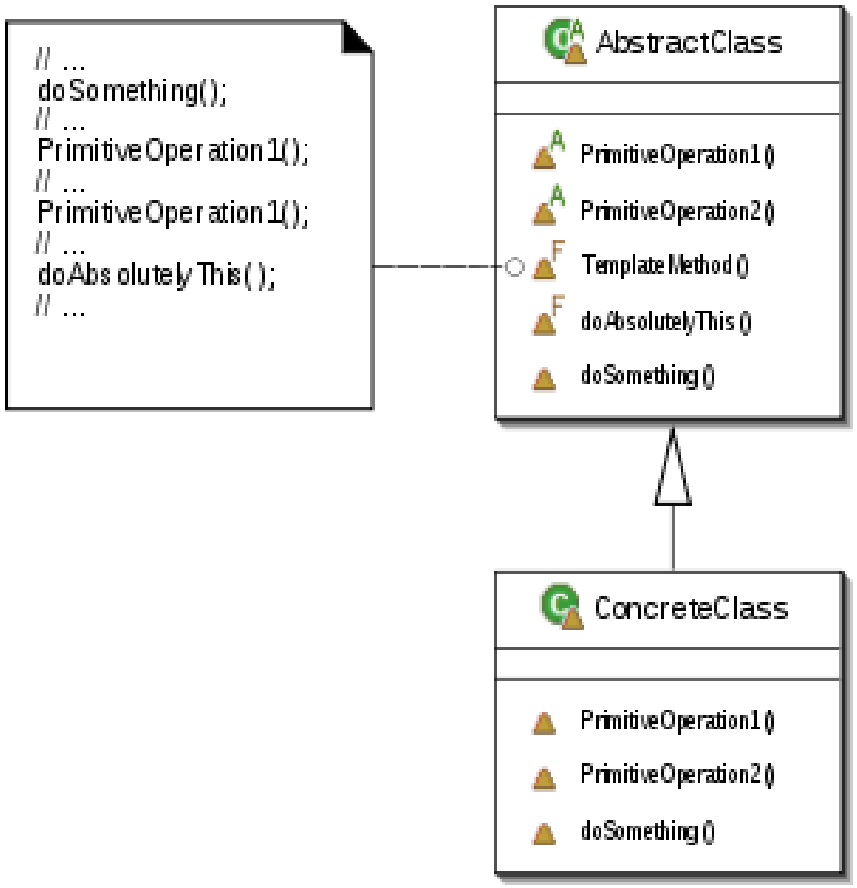
GoF

- Problem
  - How to define a general algorithm allowing for specific steps to be defined later on?
- Solution
  - Define the algorithm in the a base class with abstract methods for the steps you want to be overridden
  - Define subclasses that implement those steps

Template Method

GoF

- Template method garantes the opverall algorithm structure while allowing for certain operations (steps) to be tailored for concrete cases



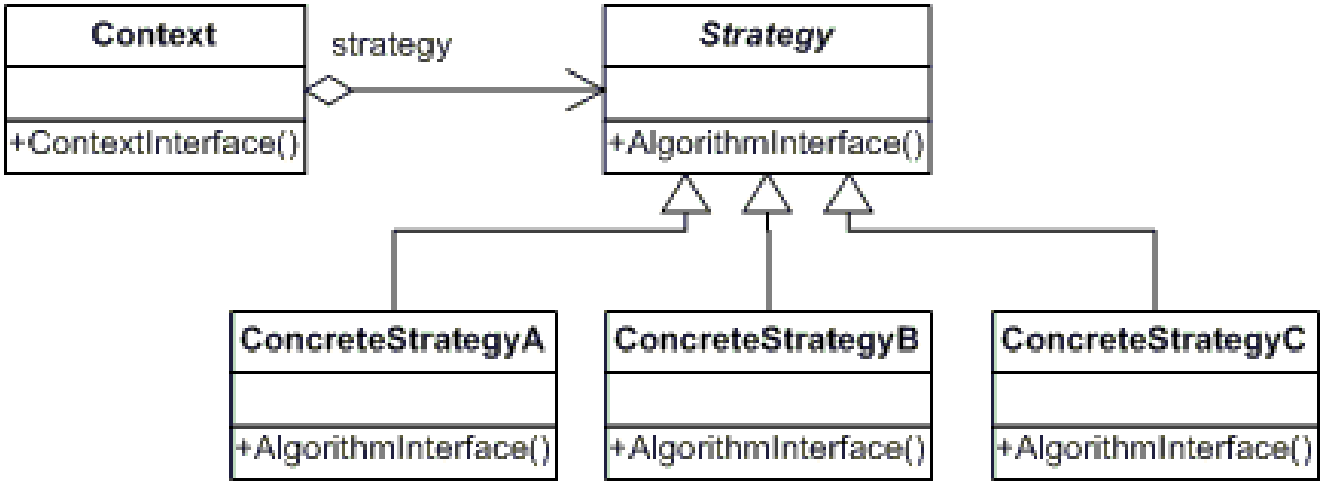
Strategy

GoF

- Problema:
  - Permitir que o cliente escolha de muitas alternativas, complexas, sendo que não se deseja incluir código para todas.
- Solução:
  - Fazer muitas implementações da mesma interface, e permitir que o cliente seleccione uma e a devolva.

Strategy

- Definir uma família de algoritmos, encapsular cada um deles, e torná-los permutáveis.
- Strategy permite que o algoritmo varie independentemente dos clientes que o usam.



fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Exemplo (C#)

- Uma colecção de elementos pode implementar diversos algoritmos (estratégias) de ordenação

```
interface SortStrategy {
   void Sort(Coleccao obj);
}

class QuickSort : SortStrategy {
   public void Sort(Coleccao obj) { ... }
}

class MergeSort : SortStrategy {
   public void Sort(Coleccao obj) { ... }
}

class ShellSort : SortStrategy {
   public void Sort(Coleccao obj) { ... }
}
```

Exemplo (C#)

- Implementar cada uma das estratégias

```
class Coleccao {
   ...
   private SortStrategy theStrategy;
   public Coleccao(SortStrategy aStrategy) {
      ...
      theStrategy = aStrategy;
   }
   public void Sort() {
      theStrategy.Sort(this);
   }
}
```

Exemplo (C#)

- Ao criar instâncias da coleção indicar qual a estratégia a utilizar

```
class TesteColeccao
{
   ...
   public void teste() {
      Coleccao c1 = new Coleccao(new QuickSort());
      C1.Sort();
      ...
      Coleccao c2 = new Coleccao(new MergeSort());
      C2.sort();
      ...
   }
}
```

Múltiplas classes SalePricingStrategy com método getTotal polimórfico

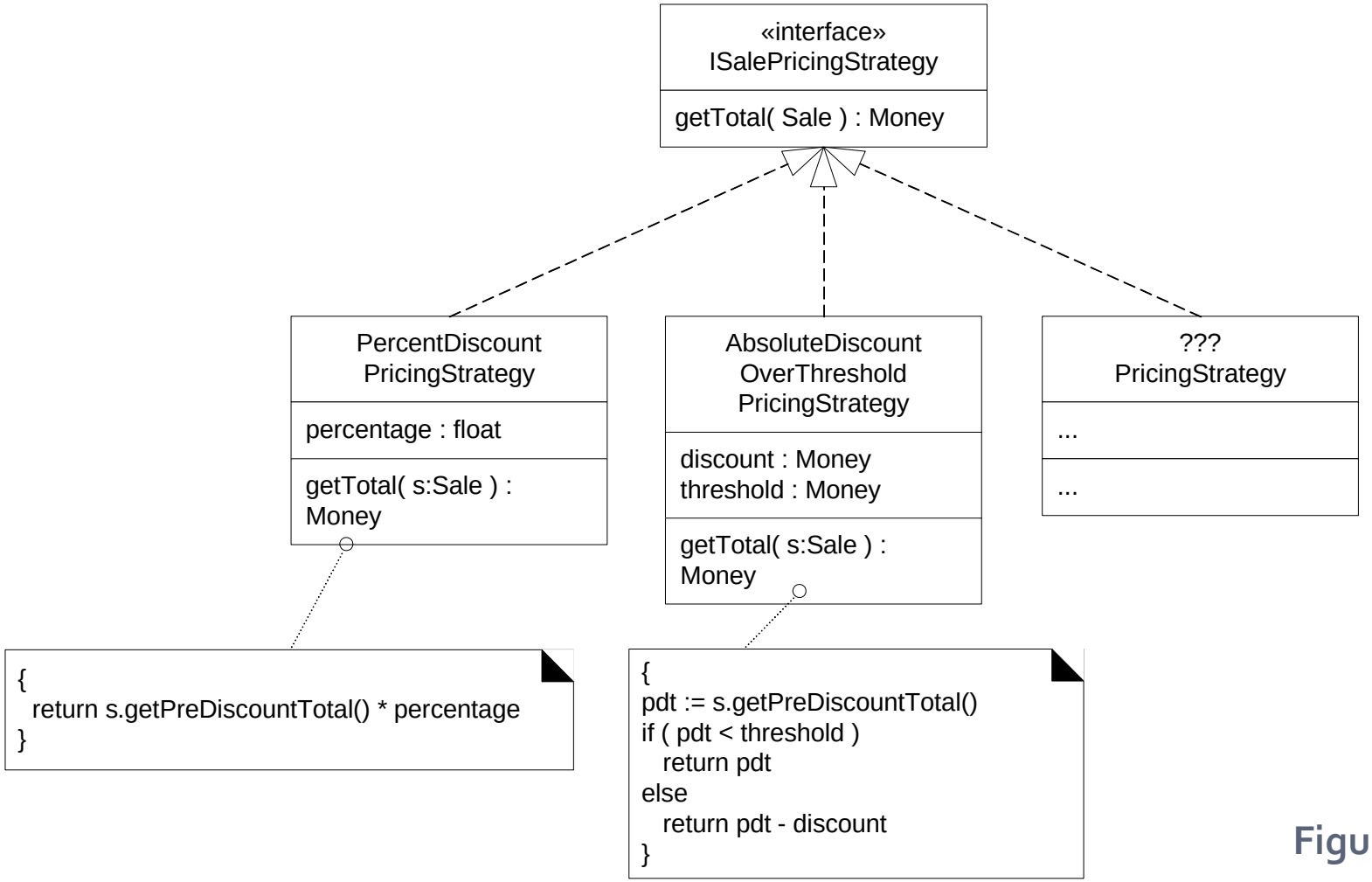


Figure 26.9

Decorator

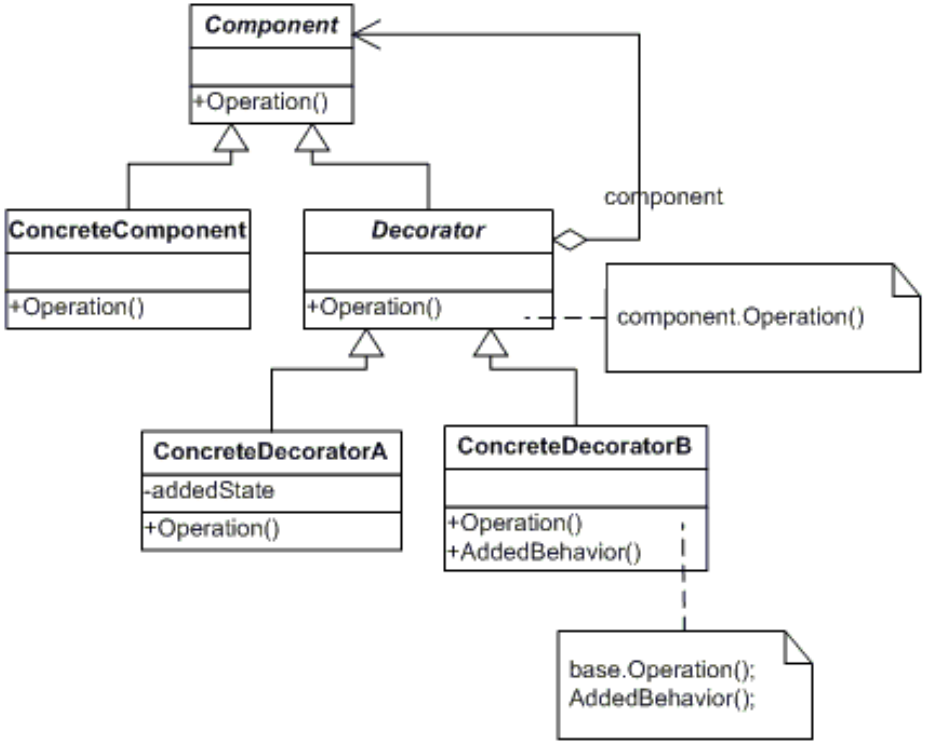
GoF

- **Problem:**
  - Allow functionally to be layered around an abstraction, but still dynamically changeable.
- **Solution:**
  - Combine inheritance and composition. By making an object that both subclasses from anther class and holds an instance of the class, can add new behavior while referring all other behavior to the original class.

Decorator

GoF

- Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.



fonte: Design Patterns: Elements of Reusable Object-Oriented Software

Exemplo (C#)

- Acrescentar capacidades de *logging* a uma classe de acesso a dados já existente

```
public interface IAcessoDados
{
    public bool Insert(object r);
    public bool Delete(object r);
    public bool Update(object r);
    public object Load(object id);
}
```

Exemplo (C#)

```
public class PessoaAcessoDados : IAcessoDados
{
    public PessoaAcessoDados() { ... }

    public bool Insert(object r) { ... }

    public bool Delete(object r) { ... }

    public bool Update(object r) { ... }

    public object Load(object id) { ... }
}
```

- Tipicamente, criariam subclasse com comportamento de *logging*.

Exemplo (C#) : solução

```
public class LoggingDecorator : IAcessoDados
{
    IAcessoDados componente;
    public LoggingDecorator(IAcessoDados componente) {
            this.componente = componente;
    }

    public bool Insert(object r) {
            LogOperation("Insert", r);
            return componente.Insert(r);
    }

    public bool Delete(object r) { ... }
    public bool Update(object r) { ... }
    public object Load(object id) { ... }

    private void LogOperation(string op, object parms)
    { ... }
}
```

Exemplo (C#)

```
public class TesteDecorator
{
    public void Teste()
    {
            IAcessoDados da = new PessoaAcessoDados();
            IAcessoDados dec = new LoggingDecorator(da);

            ...

            // use
            dec.Insert(...);
            ...
    }
}
```

Hide behind a Factory

Decorator

- Como a classe **`Decorator`** implementa a mesma interface do **`Component`**, pode ser usada em qualquer lugar do programa que necessite de um objecto **`Component`**

- Se usássemos herança não conseguiríamos resolver cenários em que necessitássemos apenas de *Logging* ou apenas de contagem ou de ambos
  - Mas é possível encadear **`Decorator`**s!

Exemplo (C#) : solução

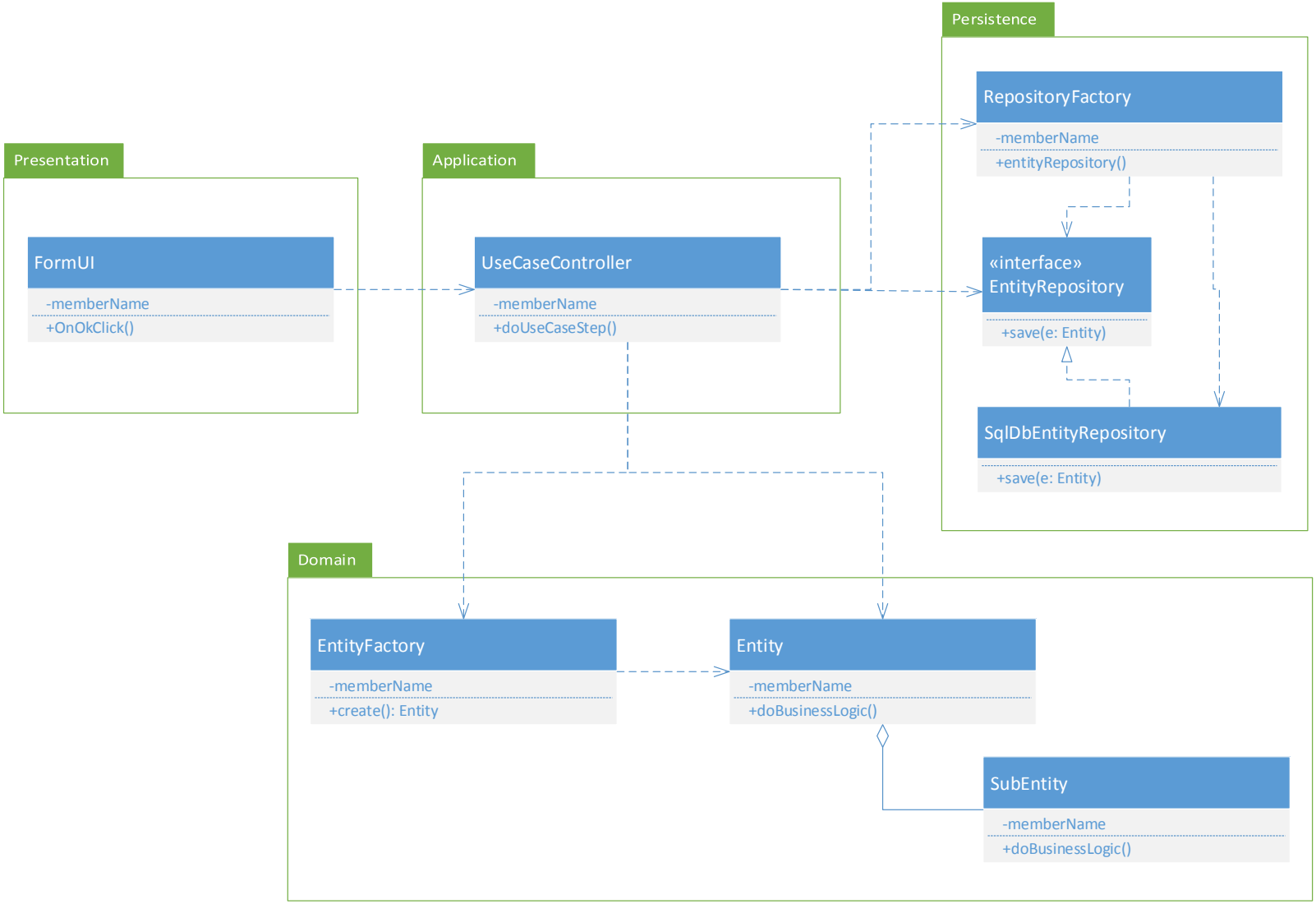
```
public class CounterDecorator : IAcessoDados
{
    int nAcessos = 0;

    IAcessoDados componente;
    public CounterDecorator(IAcessoDados componente) {
            this.componente = componente;
    }
    public bool Insert(object r) {
            nAcessos++;
            return componente.Insert(r);
    }

    public bool Delete(object r) { ... }
    public bool Update(object r) { ... }
    public object Load(object id) { ... }

    public int NumAcessos { get { return nAcessos; } }
}
```

Exemplo (C#)

```
public class BillingDAL
{
    public void Teste()
    {
            IAcessoDados da = new PessoaAcessoDados();
            IAcessoDados dec = new LoggingDecorator(da);
            IAcessoDados cd = new CounterDecorator(dec);

            ...

            cd.Insert(...);

            ...

            CounterDecorator bil = (CounterDecorator)cd;
            float custo = bil.NumAcessos * PRICE_PER_OP;
            ...
    }
}
```

Topic	Principles and patterns
Which class should a responsibility be assigned to?	Information Expert Tell, don't ask Single Responsibility Principle Interface Segregation Principle Intention Revealing Interfaces
How to organize the system's responsibilities?	
How to model the domain?	Persistence Ignorance Entity Value Object Domain Service Aggregate Domain Event Observer
How to handle an object's lifecycle?	Factories Repositories
How to prepare the code for modification?	**Protected Variation** **Open/Close Principle** **Dependency Inversion Principle** **Liskov Substitution Principle** **Template Method** **Strategy** **Decorator**

Sterotypical architecture



Bibliografia

- Domain Driven Design. Eric Evans
- Applying UML and Patterns; Craig Larman; (2nd ed.); 2002.
- Why getters and setters are Evil. Allan Holub. http://www.javaworld.com/article/2073723/core-java/why-getter-and-setter-methods-are-evil.html
- Design Principles and Design Patterns. Robert Martin. http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf
- Design Patterns-Elements of Reusable Object-oriented Software, Gamma et al. (Gang of Four)