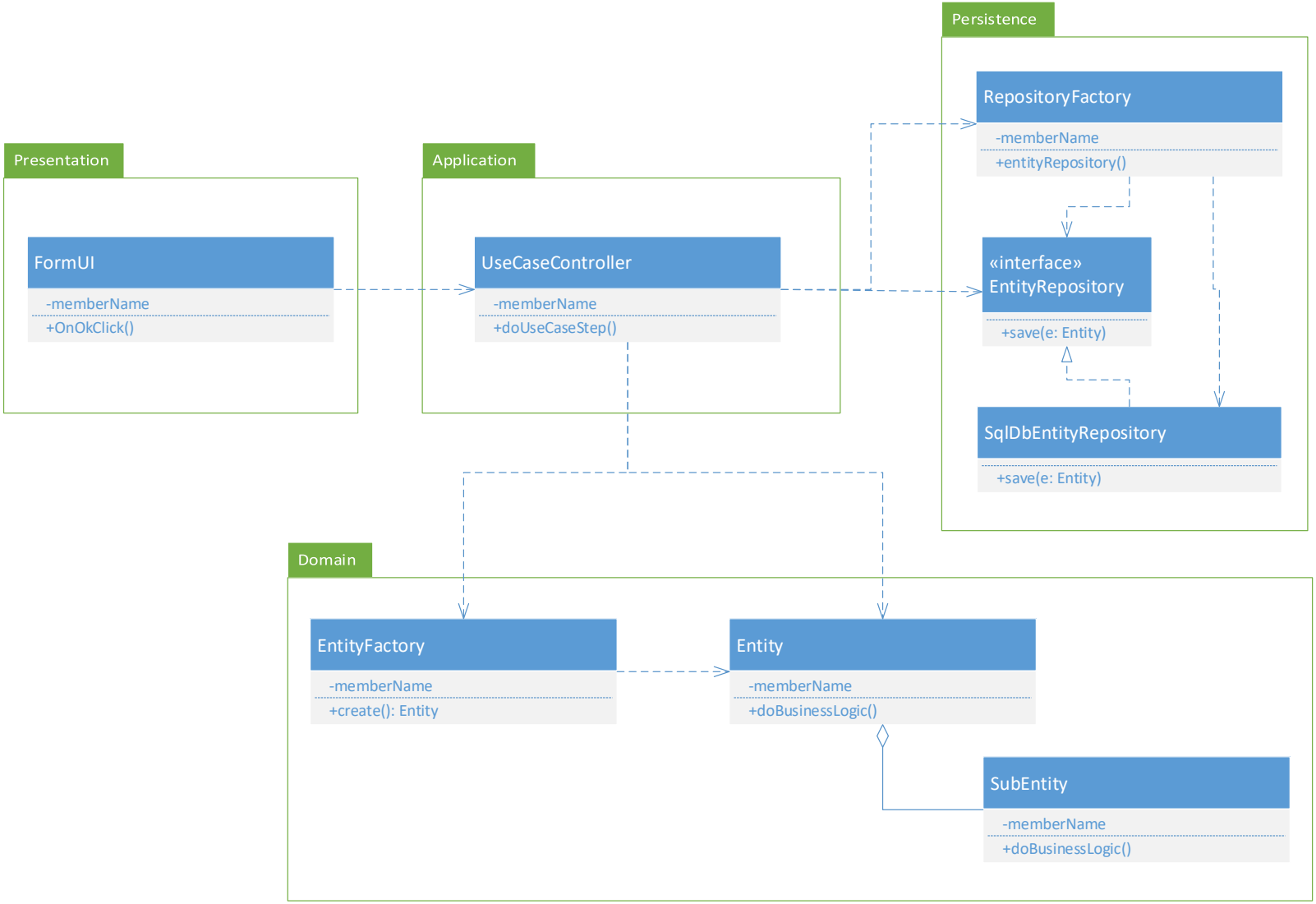
EAPLI

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?	Layers Módulos/packages Information Expert High cohesion/low coupling
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	

Sterotypical architecture



Como preparar o código para modificação?

We are in maintenance mode from the minute we do the first commit

Maintenance = bug fixes, change requests, new features

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

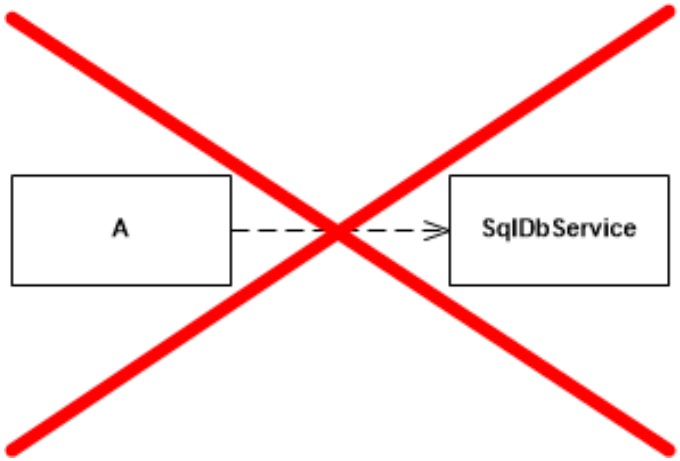
- Uma classe deve ser aberta (*open*) para extensão mas fechada (*close*) para modificação

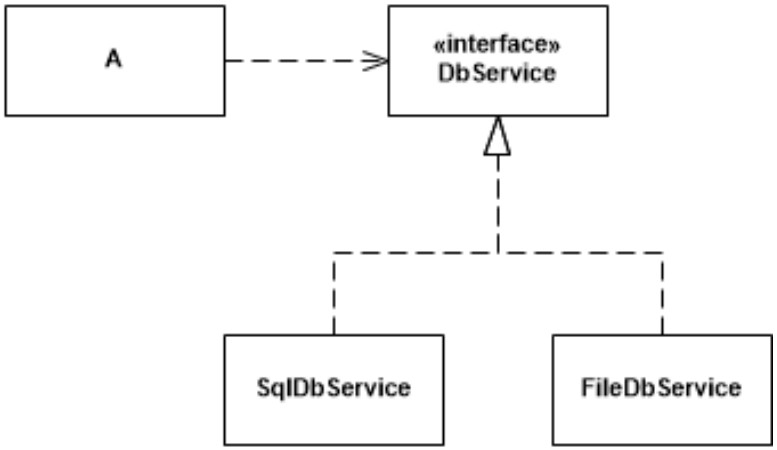
- Novos requisitos e comportamentos devem ser obtidos através da extensão da classe e não da sua modificação
- Ao criar a classe identificar (possíveis) pontos de futura variabilidade e desenhar a classe para poder ser estendida nesses pontos

Dependency Inversion Principle

SOLID

Clients should depend on abstractions, not concretions. I.e., program to an interface not a realization.



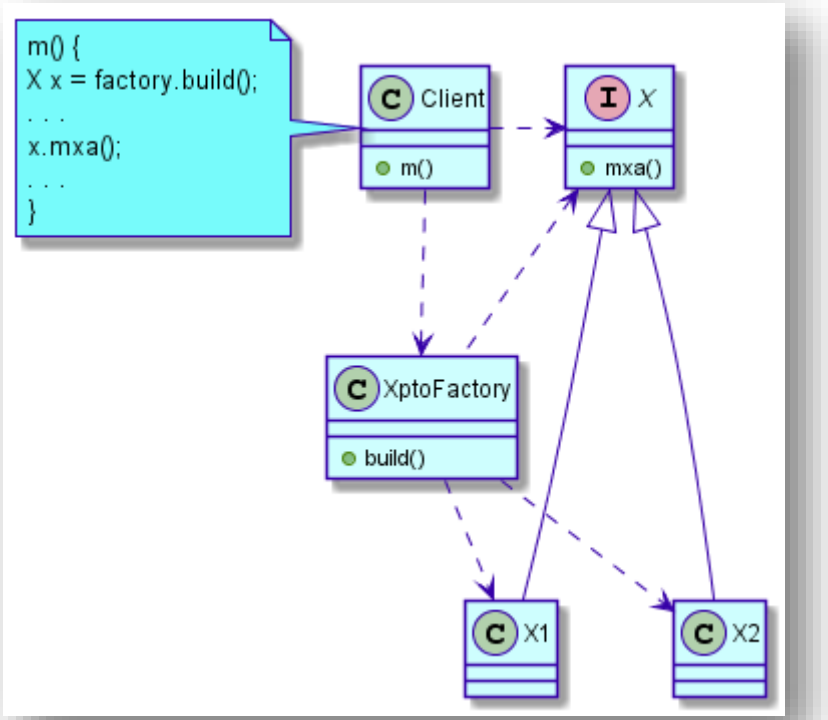


Dependency Inversion

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

As Fábricas são nossas amigas



- Se
  - queremos modificar/substituir comportamento
  - Sem impacto no restante código
- Então
  - necessário isolar a "utilização" da "criação"

Factory Method
Simple Factory
Abstract Factory
Builder

Como preparar o código para modificação?

Substituição de comportamentos

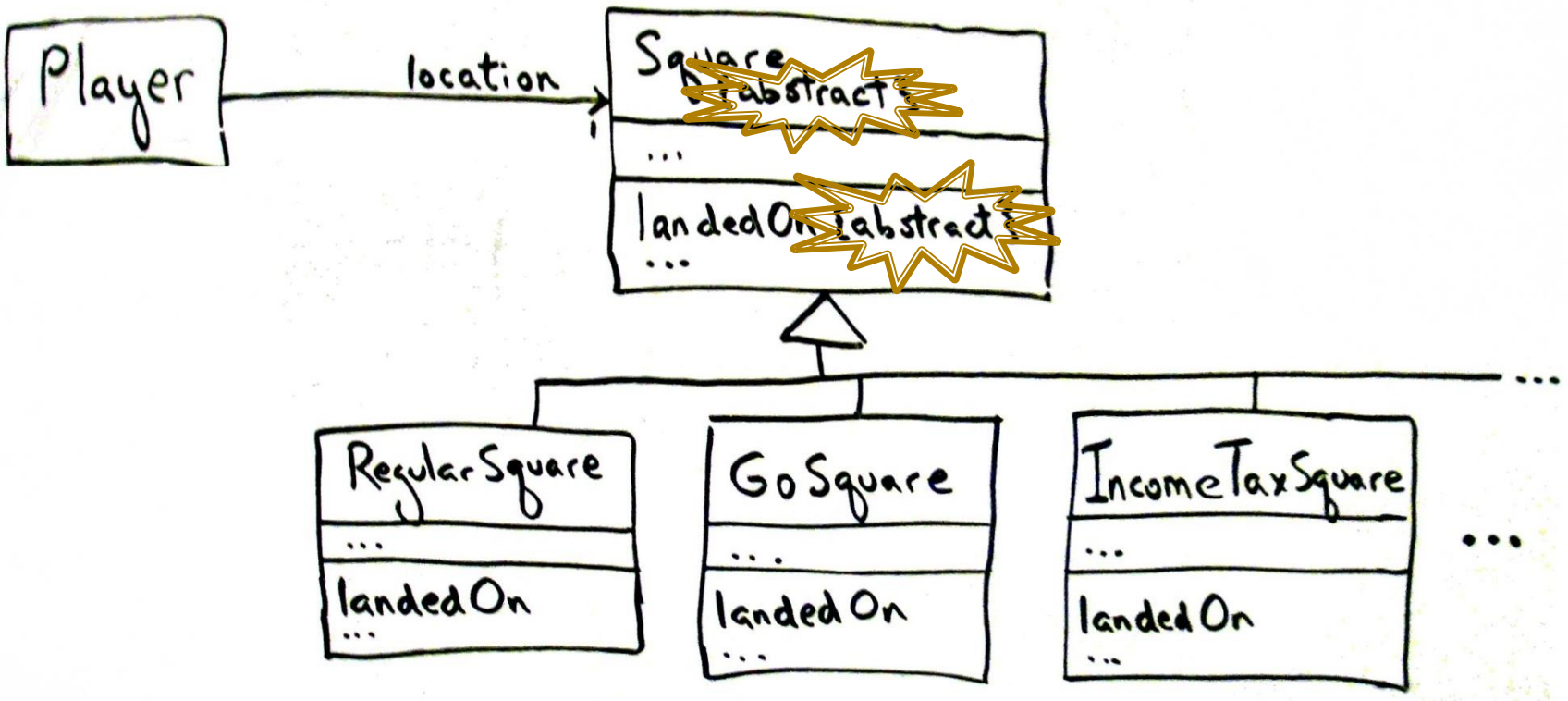
Polymorphism

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

- Onde aplicar o polimorfismo no Monopólio?



Liskov Substitution Principle

Subclasses should be substitutable for their base classes.

THE IMPLICATION IS:

Subclasses ***must*** *abide the* ***same contract and invariants*** *of the base class,* ***without semantic changes****.*

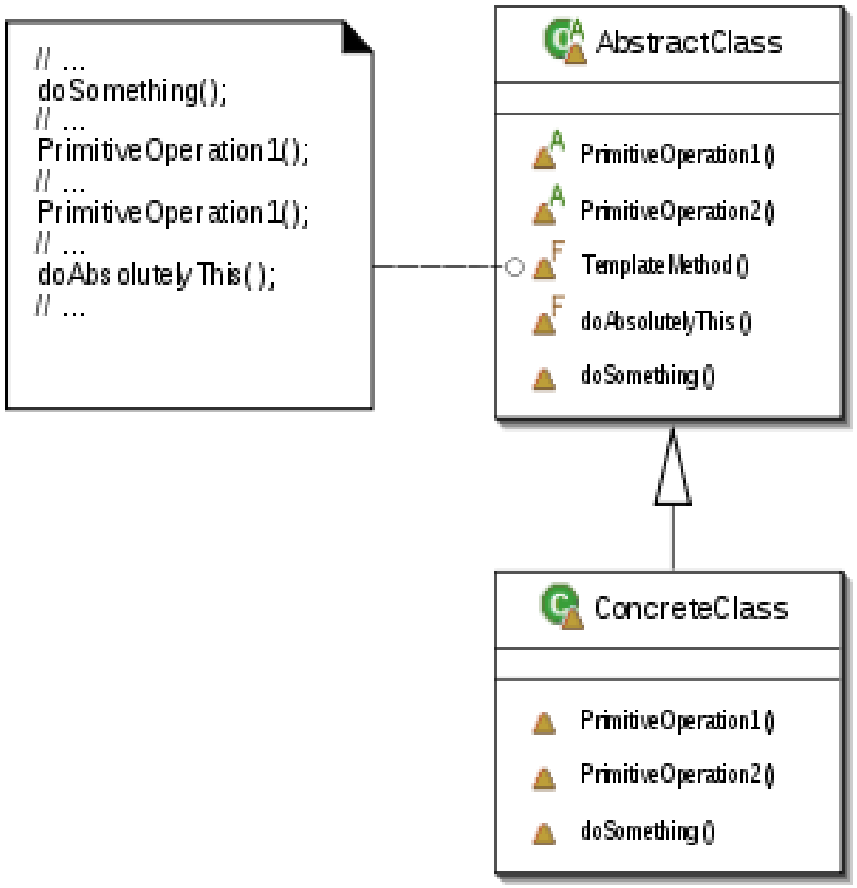
Template Method

GoF

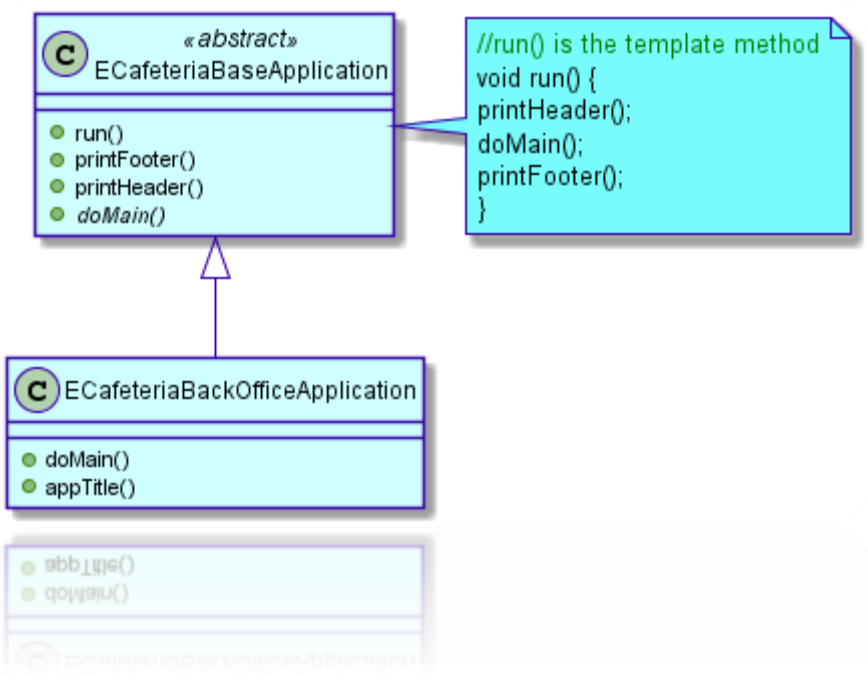
- Problem
  - How to define a general algorithm allowing for specific steps to be defined later on?
- Solution
  - Define the algorithm in the a base class with abstract methods for the steps you want to be overridden
  - Define subclasses that implement those steps

Template Method

Define an overall algorithm structure while allowing for certain operations (steps) to be tailored for concrete cases



Exemplo eCafeteria



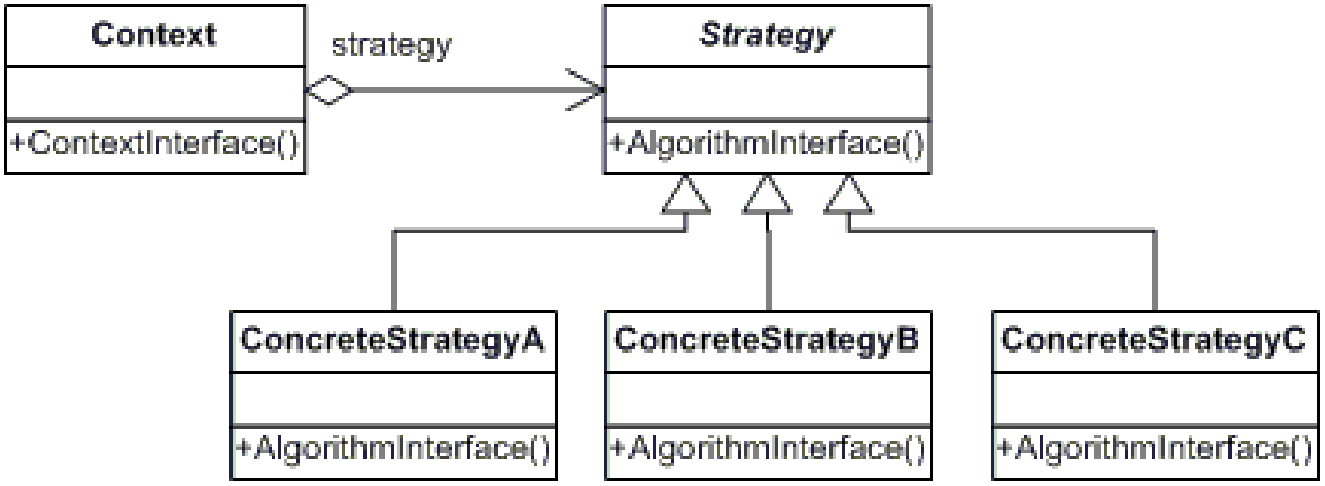
Strategy

GoF

- Problema:
  - Permitir que o cliente escolha de muitas alternativas, complexas, de um mesmo algoritmo
- Solução:
  - Fazer muitas implementações da mesma interface, e permitir que o cliente seleccione uma.

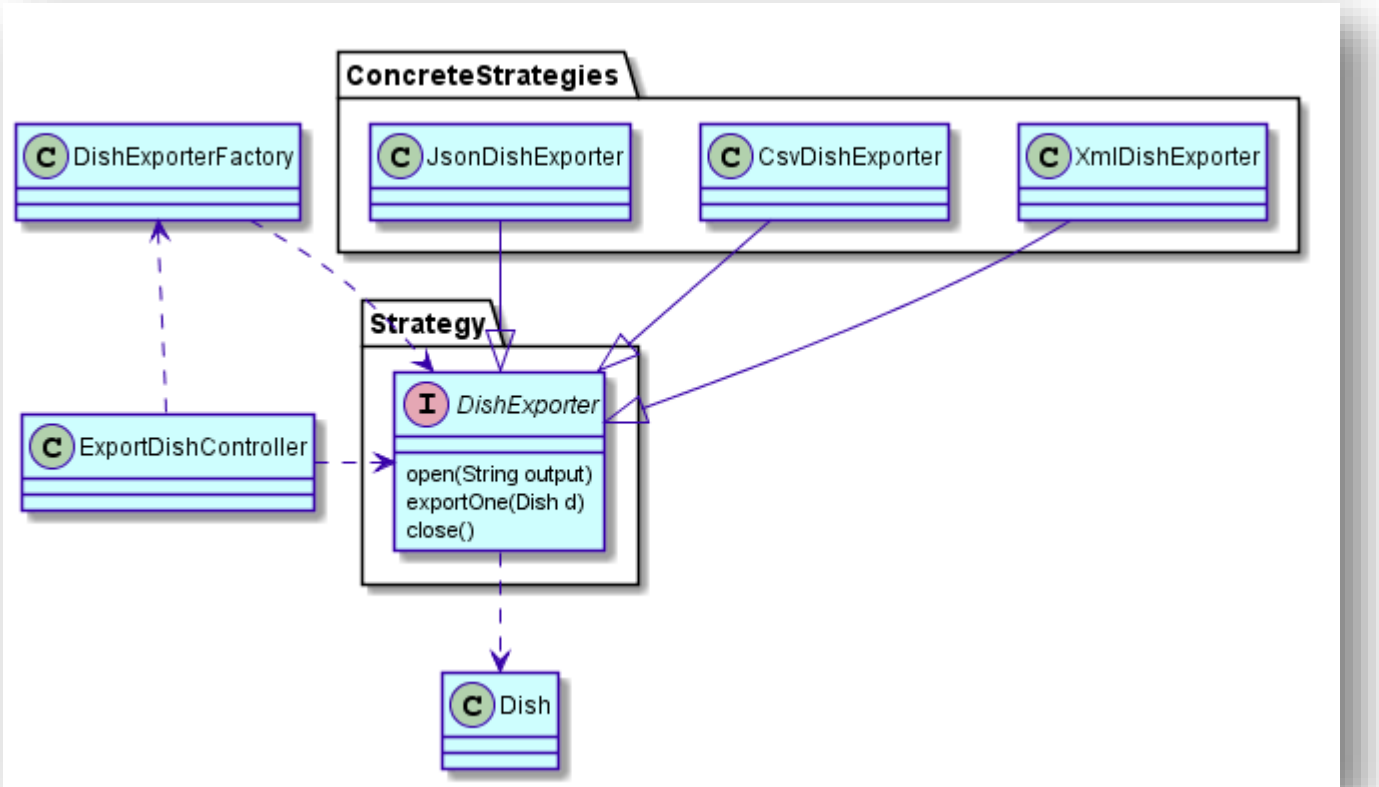
Strategy

- Strategy permite que o algoritmo varie independentemente dos clientes que o usam.



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

Exemplo


ConcreteStrategies
DishExporterFactory
JsonDishExporter
CsvDishExporter
XmlDishExporter
Strategy
DishExporter
open(String output)
exportOne(Dish d)
close()
ExportDishController
Dish

Outro Exemplo

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

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

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

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

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

Outro Exemplo (2)

- Implementar cada uma das estratégias

```
class Coleccao {
   ...
   private SortStrategy theStrategy;

   public Coleccao(SortStrategy aStrategy) {
      ...
      theStrategy = aStrategy;
   }

   public void Sort() {
      theStrategy.Sort(this);
   }
}
```

Outro Exemplo (3)

- 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();
      ...
   }
}
```

Como preparar o código para modificação?

Composição de comportamentos

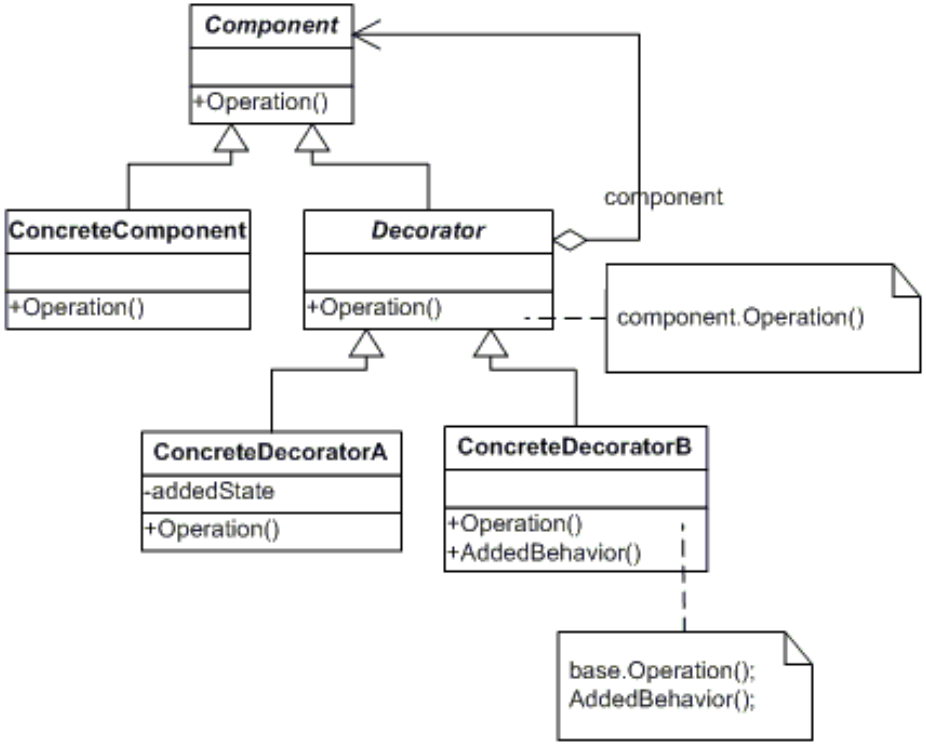
Decorator

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

Dynamically attach additional responsibilities to an object. Decorators provide a flexible alternative to subclassing for extending functionality.



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

Exemplo (1): contexto

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


public class PessoaAcessoDados implements IAcessoDados
{
    public PessoaAcessoDados() { ... }

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

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

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

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

Exemplo (2): problema

- Como acrescentar capacidades de logging a uma classe de acesso a dados já existente?

- Tipicamente:
  - Alterar classe existente
  - Criar subclasse com comportamento de logging.

Exemplo (3) : 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 (4)

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

            ...

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

Hide behind a Factory

Exemplo (5): evolução

- Também precisamos de contar o número de vezes que cada operação é executada
- Nalguns clientes temos que fazer logging e contar as operações, mas noutros é só contar e noutros é só logging
- Noutros ainda vamos querer notificar o DPO se os dados contiverem determinados conteudos
- E queremos que tudo seja possivel ☺
- …

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 (6) : solução

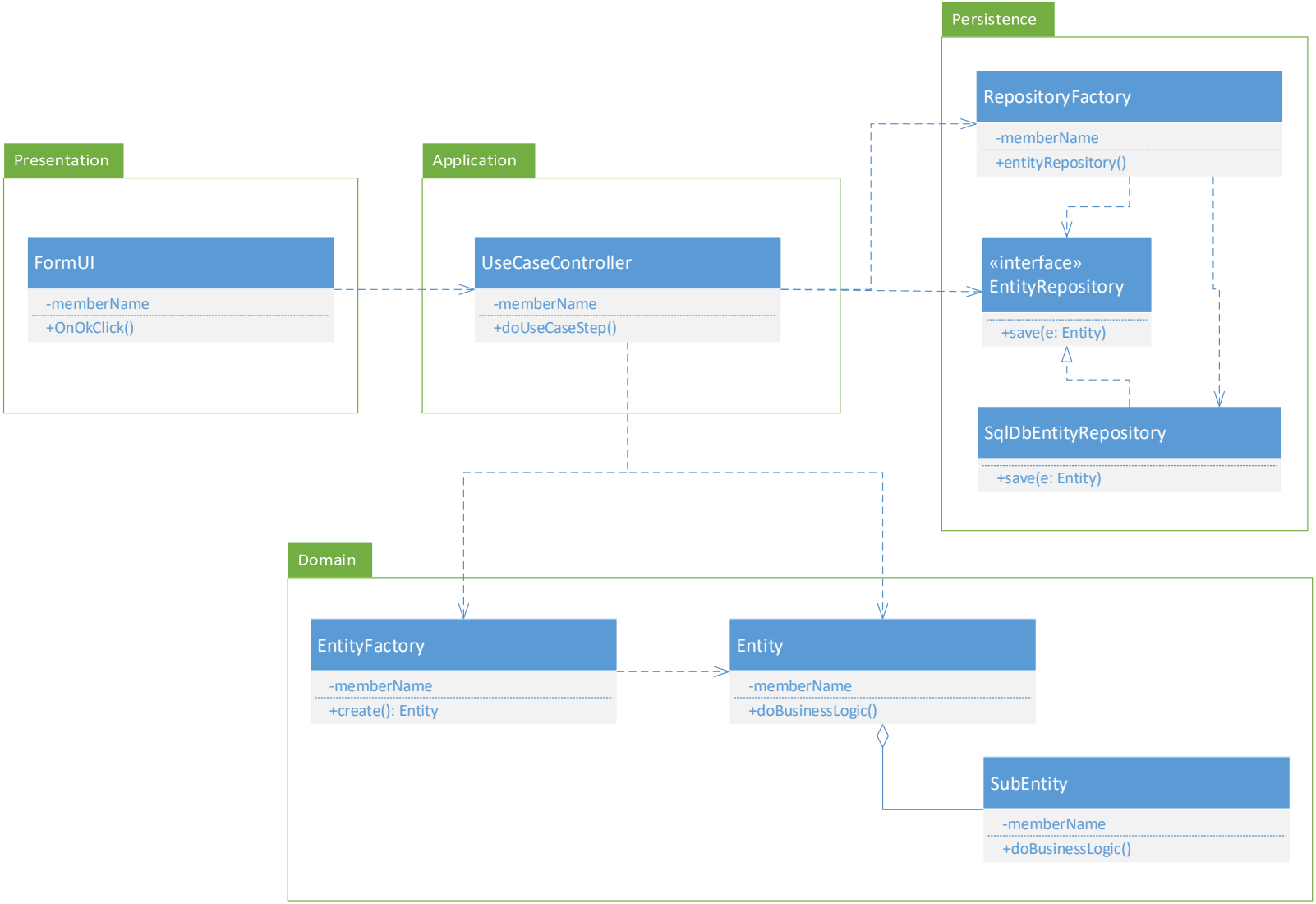
```
public class CounterDecorator implements IAcessoDados
{
    int nAcessos = 0;

    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 (7): solução

```
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?	Layers Módulos/packages Information Expert High cohesion/low coupling
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

- 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)