Padrões de Projeto

Hoje:

1 - Correção do Exercício
(apresentações em grupos)
2 - Padrão Façade
3 - Padrão Bridge
4 - Padrão Composite
(divisão em grupos)
5 - Bate papo sobre Arquitetura de Software

Exercício Sistema de Elevadores

Requisito Funcional:

Um novo empreendimento imobiliário está com intenção de desenvolver um sistema inteligente de ***monitoramento*** *de elevadores. Você foi incumbido de projetar uma solução computacional que gerencia o controle de movimentação dos vários elevadores do edifício. Os elevadores são convencionais, podendo* ***subir****,* ***descer*** *e* ***parar*** *nos andares solicitados.*

A inteligência do sistema está no fato de que o edifício deve ***monitorar*** *o funcionamento dos elevadores para proporcionar mais segurança e também uma experiência de uso mais agradável para os moradores. As ações que o sistema pode tomar variam desde ações de segurança até mesmo a ações de entretenimento, como reprodução de músicas dentro do elevador.*

Exemplos:

- Se um determinado elevador ***emperrar*** *durante seu funcionamento, o sistema deve perceber e automaticamente invocar a equipe de manutenção e segurança. Enquanto isso uma música agradável pode começar a ser reproduzida naquele elevador específico para acalmar o usuário.*

- Se um determinado elevador ***estiver em manutenção****, o sistema pode, entre outras ações, interferir na velocidade de deslocamento dos outros elevadores.*

- Se um determinado elevador estiver em movimentação (subindo ou descendo), dependendo do peso interno a velocidade do elevador pode ser alterada

Requisitos Não-Funcionais:

- O código cliente que manipula os elevadores deve ser o mais inconsciente possível dos objetos concretos existentes

- Nesta primeira versão do sistema, um elevador nunca estará em dois estados ao mesmo tempo.

- Embora não haja limitação de memória na infraestrutura computacional usada, deve-se prezar para que não haja ***criação desnecessária de objetos em memória****.*

Padrões de Projeto

Façade

PADRÃO FAÇADE (FACHADA)

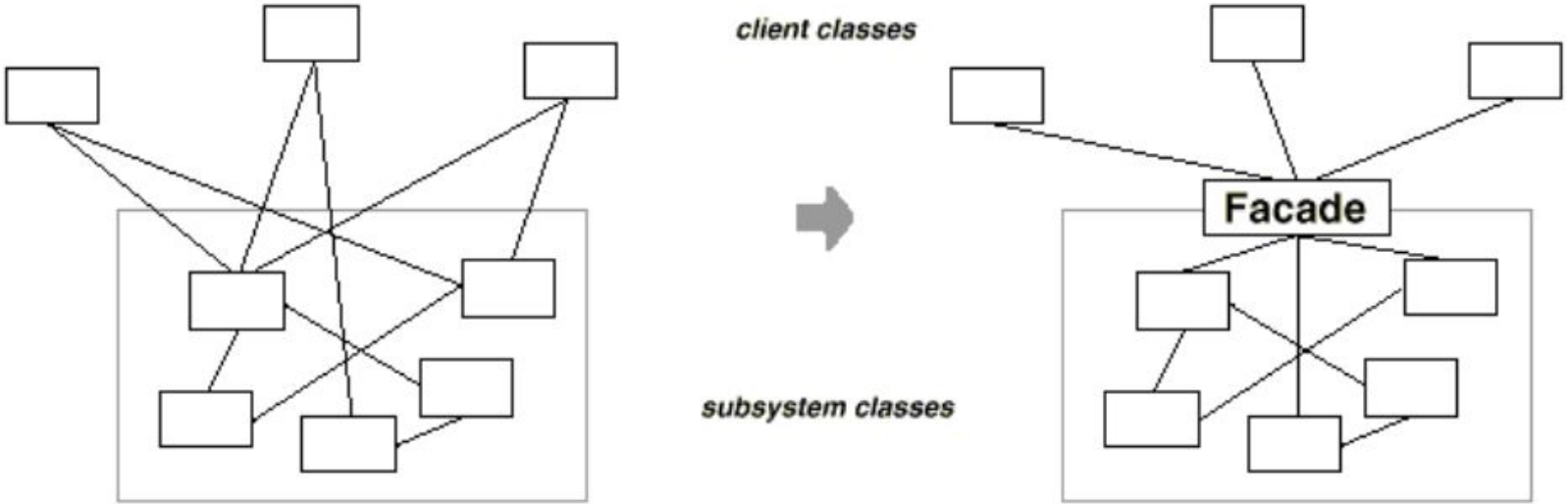
Intention

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

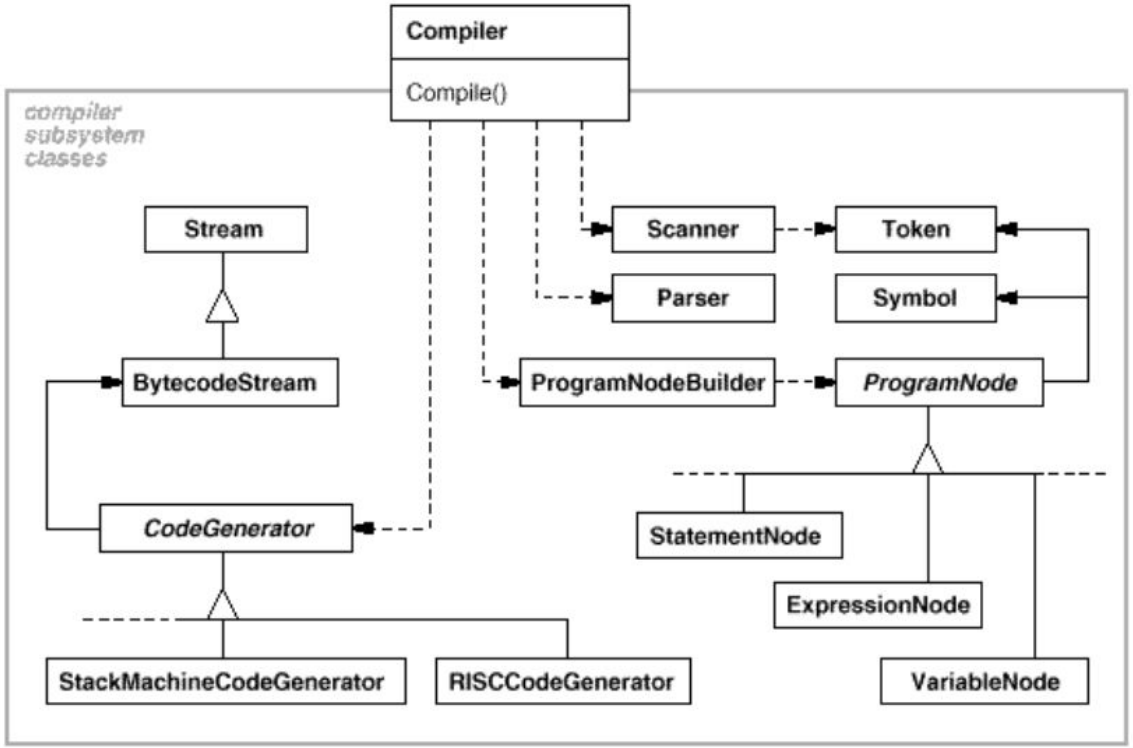
Motivation

- Structuring a system into subsystems helps reduce complexity. A common design goal is to minimize the communication and dependencies between subsystems. One way to achieve this goal is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem

PADRÃO FACHADA



PADRÃO FACHADA - EXEMPLO



PADRÃO FACHADA

Participants

- **Facade** (Compiler)
  - knows which subsystem classes are responsible for a request.
  - delegates client requests to appropriate subsystem objects.
- **subsystem classes** (Scanner, Parser, ProgramNode, etc.)
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade; that is, they keep no references to it.

PADRÃO FACHADA - EXEMPLO

Contexto: Há dois subsistemas. O subsistema 1 precisa invocar as funcionalidades básicas de um computador e o subsistema 2 tem a responsabilidade de representar e fornecer funcionalidades básicas de um computador. Entretanto, é aconselhável diminuir o acoplamento entre esses subsistemas.
O subsistema que representa o computador deve:
1 – processar dados
2 – carregar dados
3 – ler dados

```
package com.devmediaexemplo.facade;
class DiscoRigido {
    public void leDado() { }
}
```

```
package com.devmediaexemplo.facade;
class Memoria {
    public void carregaDado() { }
}
```

```
package com.devmediaexemplo.facade;
  class UnidadeCentralDeProcessamento {
      public void processaDado() { }
  }
```

PADRÃO FACHADA - EX

```
class ComputadorFacade {
    private UnidadeCentralDeProcessamento unidadeCentralDeProcessamento;
    private Memoria memoria;
    private DiscoRigido discoRigido;

    public ComputadorFacade() {
        this.unidadeCentralDeProcessamento = new UnidadeCentralDeProcessamen
        this.memoria = new Memoria();
        this.discoRigido = new DiscoRigido();
    }

    public void run() {
        unidadeCentralDeProcessamento.processaDado();
        memoria.carregaDado();
        discoRigido.LeDado();
    }
}
```

```
package com.devmediaexemplo.facade;

class UsuarioExemplo {
    public static void main(String[] args) {
        ComputadorFacade facade = new ComputadorFacade();
        facade.run();
    }
}
```

Neste exemplo, mudanças no nome dos métodos não impactam o cliente. Entretanto, se os métodos utilizados precisarem de parâmetros isso impactará também a classe consumidora.

PADRÃO FACHADA

39. (True / False) The Façade design pattern introduces a new interface..

39. (True / False) The Façade design pattern introduces new functionality.

39. (True / False) When a Façade object is used with a subsystem, the subsystem is aware of the façade object.

PADRÃO FACHADA

39. (True / False) The Façade design pattern introduces a new interface..

Ans. T

39. (True / False) The Façade design pattern introduces new functionality.

Ans. F

39. (True / False) When a Façade object is used with a subsystem, the subsystem is aware of the façade object.

Ans. False. There is no reason for the subsystem to know about the facade object. The facade object simply delegates requests to the subsystem. It is a one-way relationship.

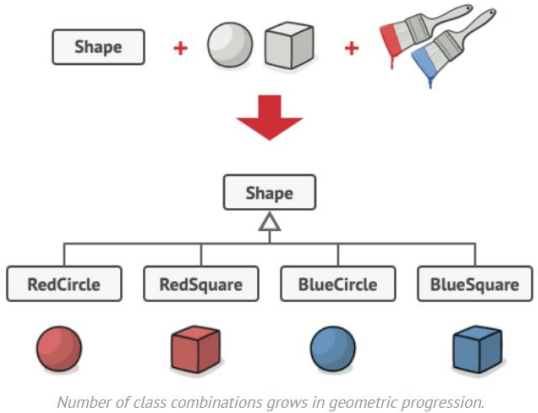
Padrão Bridge

https://refactoring.guru/design-patterns/bridge

BRIDGE

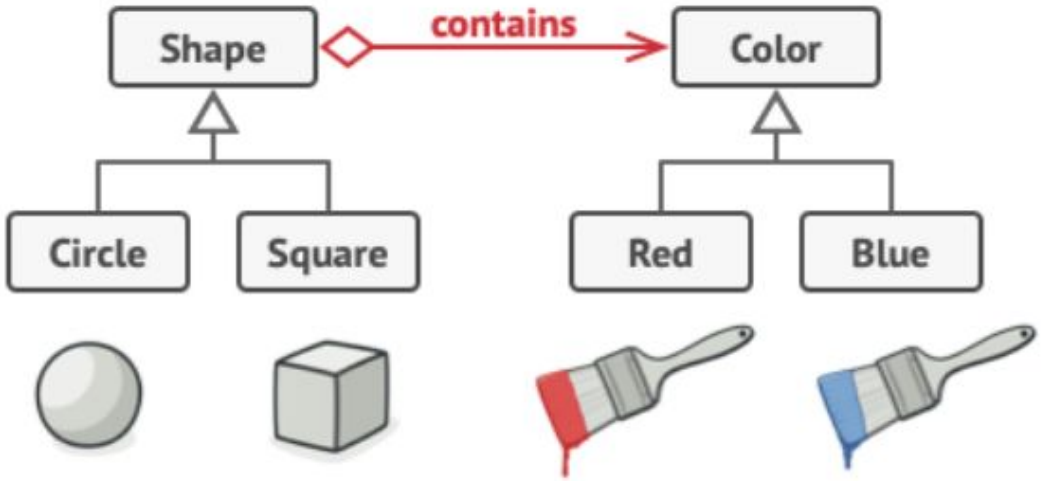
When to use?

Use the Bridge Pattern when you noticed that your hierarchy is exploding exponentially.
This happens when you are trying to extend the classes in different **dimensions**



BRIDGE

Solução: Trocar herança por composição de objetos



BRIDGE

Exemplo:

- Cliente Físico e Juridico
  - **Todos** os clientes agora também serão divididos em Cliente Silver and Gold
  - Além dos Clientes Silver e Gold, agora também todos os clientes serão classificados em Regionais e Locais…

BRIDGE

Gold e Silver e Regional e Local são dimensões diferentes ….

Como ficaria com a aplicação do Bridge ?

Suponha a seguinte alteração nos requisitos do sistema de elevadores. Como ficaria ?

Requisito Funcional:

Um novo empreendimento imobiliário está com intenção de desenvolver um sistema inteligente de ***monitoramento*** *de elevadores. Você foi incumbido de projetar uma solução computacional que gerencia o controle de movimentação dos vários elevadores do edifício. Os elevadores são convencionais, podendo* ***subir****,* ***descer*** *e* ***parar*** *nos andares solicitados.*

Os elevadores possuem as classificações normais de social e de serviço, mas também se classificam de acordo com a qualidade dos mesmos, podem ser Ouro, Prata e Bronze. Isto é, pode existir um elevador de social do tipo Ouro, mas também um de serviço do tipo Ouro.

A inteligência do sistema está no fato de que o edifício deve ***monitorar*** *o funcionamento dos elevadores para proporcionar mais segurança e também uma experiência de uso mais agradável para os moradores. As ações que o sistema pode tomar variam desde ações de segurança até mesmo a ações de entretenimento, como reprodução de músicas dentro do elevador.*

Exemplos:

- Se um determinado elevador ***emperrar*** *durante seu funcionamento, o sistema deve perceber e automaticamente invocar a equipe de manutenção e segurança. Enquanto isso uma música agradável pode começar a ser reproduzida naquele elevador específico para acalmar o usuário.*

- Se um determinado elevador ***estiver em manutenção****, o sistema pode, entre outras ações, interferir na velocidade de deslocamento dos outros elevadores.*

- Se um determinado elevador estiver em movimentação (subindo ou descendo), dependendo do peso interno a velocidade do elevador pode ser alterada

Requisitos Não-Funcionais:

- O código cliente que manipula os elevadores deve ser o mais inconsciente possível dos objetos concretos existentes

- Nesta primeira versão do sistema, um elevador nunca estará em dois estados ao mesmo tempo.

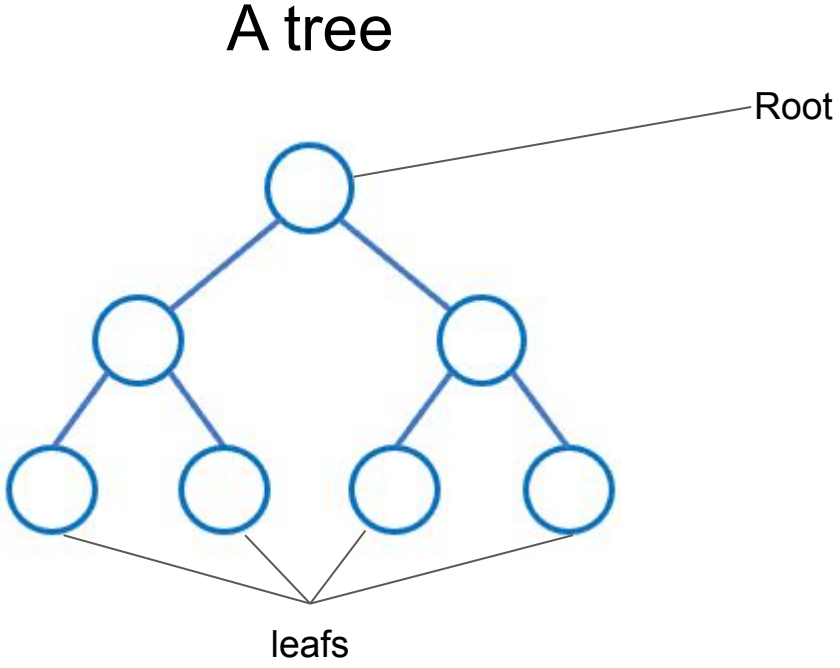
- Embora não haja limitação de memória na infraestrutura computacional usada, deve-se prezar para que não haja ***criação desnecessária de objetos em memória****.*

Padrões de Projeto

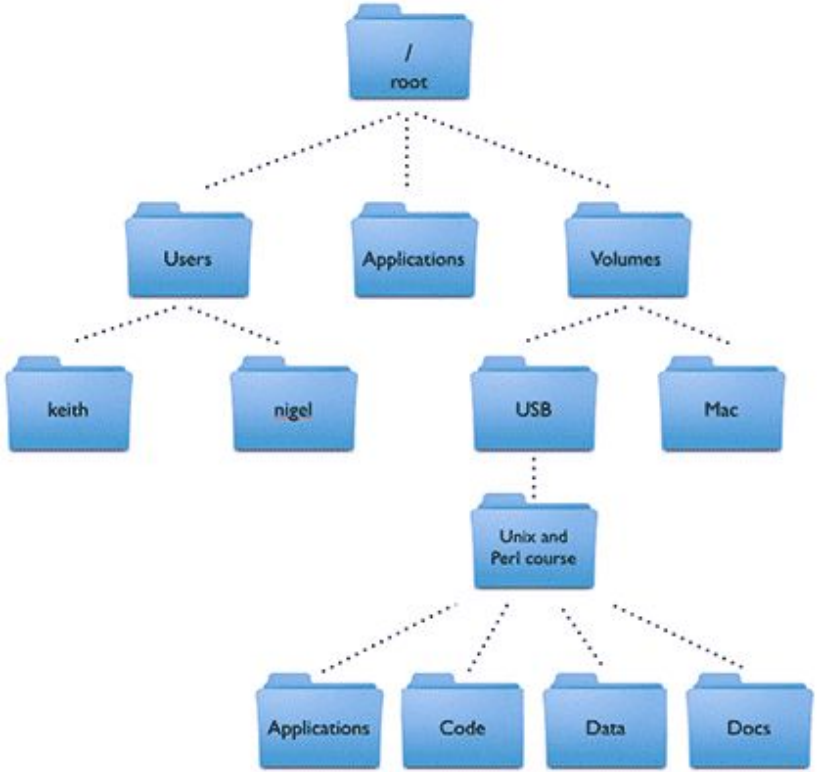
Padrão Composite

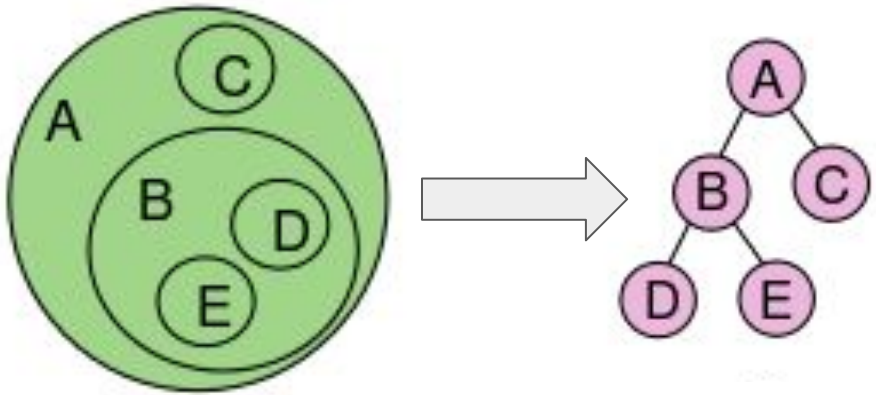
Valter Camargo

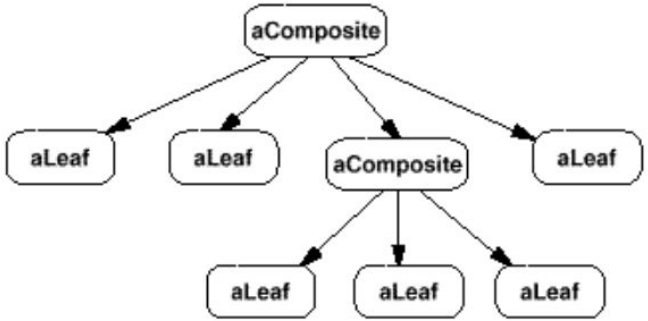
CONCEITO DE “ÁRVORE” EM PROGRAMAÇÃO



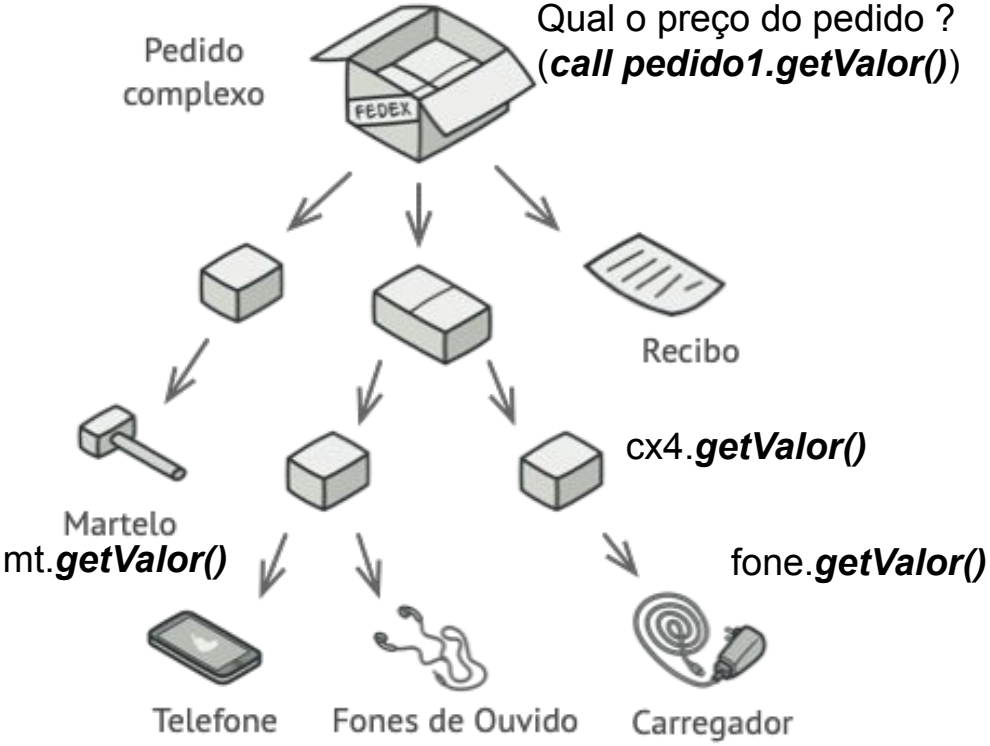
EXEMPLOS DE AGRUPAMENTOS ÁRVORE







EXEMPLO PEDIDO (CAIXA COM OUTRAS DENTRO)



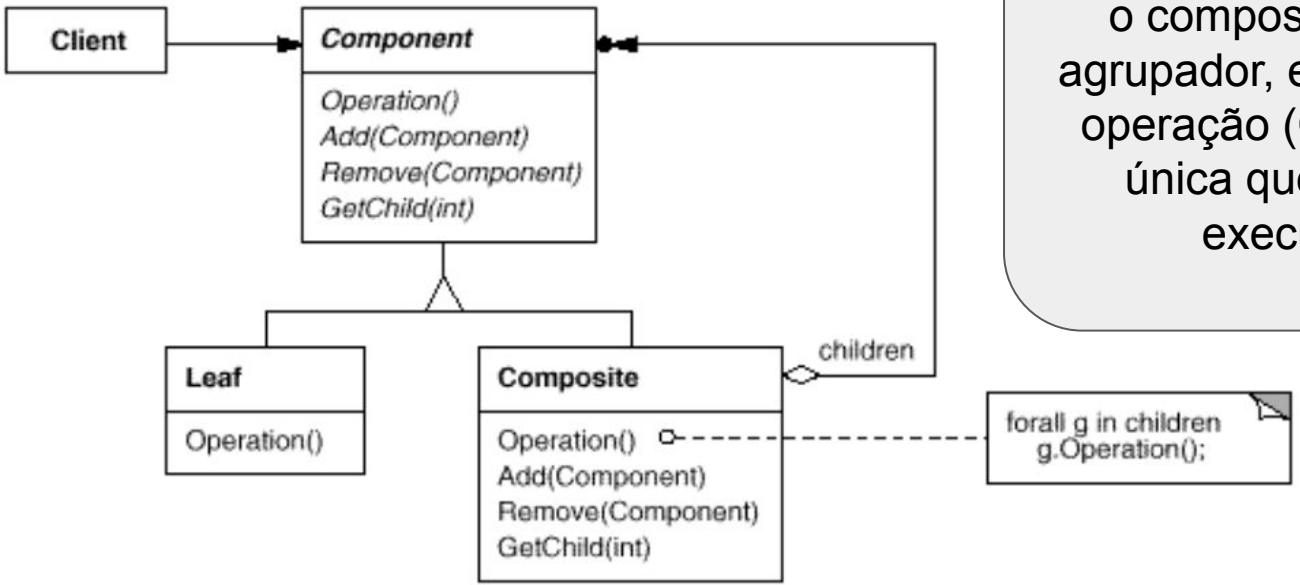
COMPOSITE

Nada mais é do que poder chamar um método para um objeto, sem se preocupar se o objeto é parte ou todo. Isto é, independentemente se o objeto é um composite ou individual

When to use?

When you need to represent composition of objects… and let clients treat (*chamar métodos*) them (individual objects and composition of objects) at the same way.

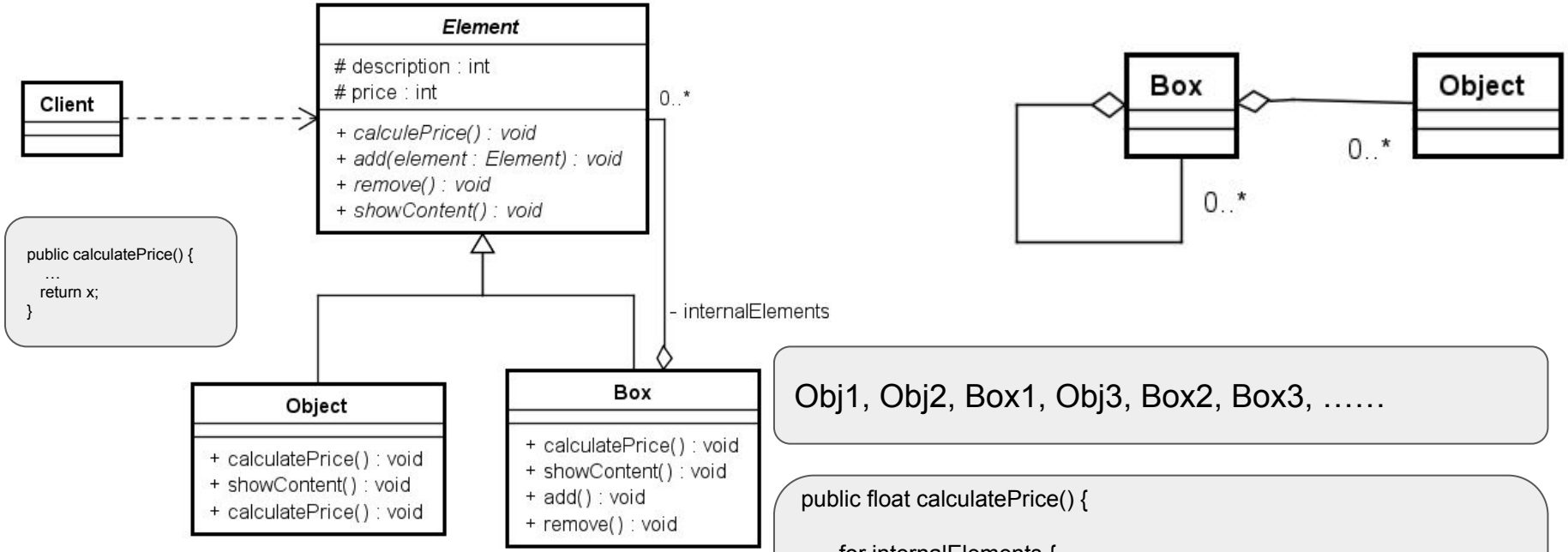
COMPOSITE



O ponto mais importante é notar que o composite só funciona como agrupador, ele não sabe executar a operação (Operation). A folha é a única que sabe efetivamente executar a operação.

COMPOSITE

Conceptual Modeling




```
public calculatePrice() {
   …
   return x;
}
```

Obj1, Obj2, Box1, Obj3, Box2, Box3, ……

```
public float calculatePrice() {

      for internalElements {
            e = internalElements.getElement();
            price = price + e.calculatePrice();
      }

}
```

Concrete Modeling (Impementation)

PRÁTICA +- RÁPIDA

- Modele apenas com diagramas de classe as situações a seguir com o padrão Composite
  - Coloque apenas as classes envolvidas (com atributos e métodos)
  - Coloque os relacionamentos envolvidos
  - Não se preocupe com o corpo dos métodos

Examples:

1) Menus contain items, each of which could be a **functionality** or another **menu**....
2) 0 Directories contain items, each of which could be a **file** or another **directory**...
3) Companies have departments and the departments can be composed of other departments or sections

E se houvesse essa alteração nos requisitos do sistema de elevadores ?

Requisito Funcional:

Um novo empreendimento imobiliário está com intenção de desenvolver um sistema inteligente de ***monitoramento*** *de elevadores. Você foi incumbido de projetar uma solução computacional que gerencia o controle de movimentação dos vários elevadores do edifício. Os elevadores são convencionais, podendo* ***subir***, ***descer*** *e* ***parar*** *nos andares solicitados.*

A inteligência do sistema está no fato de que o edifício deve ***monitorar*** *o funcionamento dos elevadores para proporcionar mais segurança e também uma experiência de uso mais agradável para os moradores. As ações que o sistema pode tomar variam desde ações de segurança até mesmo a ações de entretenimento, como reprodução de músicas dentro do elevador.*

Exemplos:

- Se um determinado elevador ***emperrar*** *durante seu funcionamento, o sistema deve perceber e automaticamente invocar a equipe de manutenção e segurança. Enquanto isso uma música agradável pode começar a ser reproduzida naquele elevador específico para acalmar o usuário.*

- Se um determinado elevador ***estiver em manutenção****, o sistema pode, entre outras ações, interferir na velocidade de deslocamento dos outros elevadores.*

- Se um determinado elevador estiver em movimentação (subindo ou descendo), dependendo do peso interno a velocidade do elevador pode ser alterada

Outro requisito funcional é que o sistema todo deve permitir o cadastro dos condomínios da incorporadora. A incorporadora possui condomínios que envolvem edifícios e casas. Uma particularidade é que, em alguns casos, um condomínio pode ter outros condomínios menores dentro dele. A principal funcionalidade disponível nessa parte é calcular o valor do imóvel, seja ele um casa, edifício ou do condomínio todo.

Requisitos Não-Funcionais:

- O código cliente que manipula os elevadores deve ser o mais inconsciente possível dos objetos concretos existentes

- Nesta primeira versão do sistema, um elevador nunca estará em dois estados ao mesmo tempo.

- Embora não haja limitação de memória na infraestrutura computacional usada, deve-se prezar para que não haja ***criação desnecessária de objetos em memória****.*

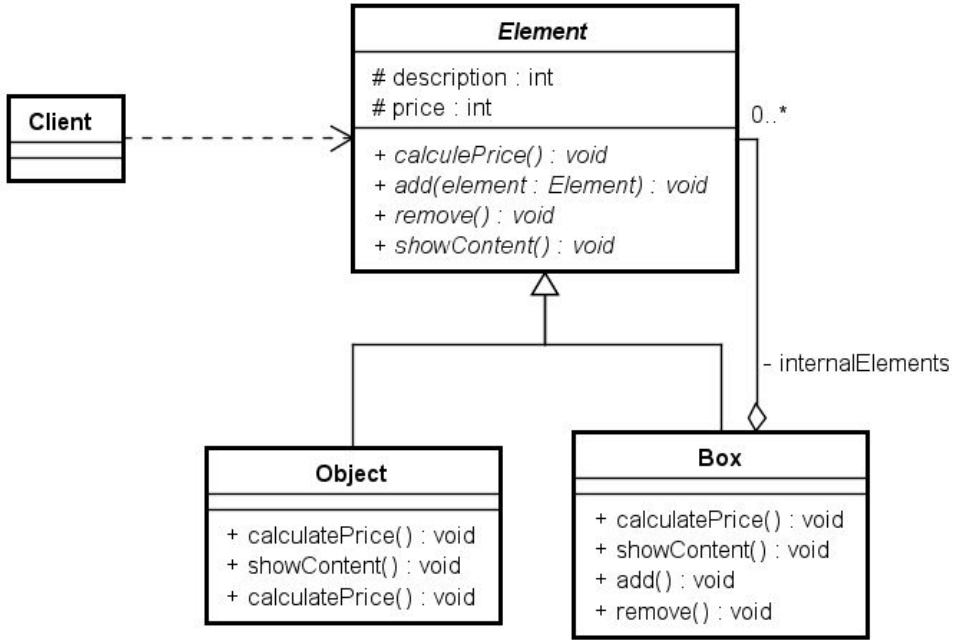
COMPOSITE - (EXER)SAMPLE !

- Considere o exemplo de Caixas e Objetos mostrado
- Queremos aplicar duas funcionalidades de forma a tratar os diferentes objetos (*caixas* e *objetos*) da mesma forma:
  - *Calcular o preço* deve funcionar tanto para uma caixa quanto para um objeto
  - *Listar infos básicas* também deve funcionar tanto para a caixa quanto para um objeto

- A saída esperada é esta ao lado →
- Faça:
  - O método add() // para adicionar objetos na caixa
  - O método calculatePrice() do Box()
  - O método showContent() do Box()
  - Implemente o Main da seguinte forma:
    - Criar três caixas
    - Criar 6 objetos com preços
    - Colocar caixas e objetos dentro de suas caixas
    - Chamar o método showContent para a Caixa 1
    - Chamar o método calculatePrice() para as 3 caixas

```
This is the Box 1
 --> Element: Object 1
 --> Element: Object 2
 --> Element: Box 2
This is the Box 2
 --> Element: Object 3
 --> Element: Object 4
 --> Element: Box 4
This is the Box 3
 --> Element: Object 5
 --> Element: Object 6

----  PRICES -----
The price of box1 is 40
The price of box2 is 20
The price of box3 is 20
```

COMPOSITE - EXERSAMPLE !



Concrete Modeling (Implementation)

COMPOSITE - EXERSAMPLE !

```
public class Client {

    public static void main(String args[]) {

        Element object1 = new Object("Object 1");
        Element object2 = new Object("Object 2");
        Element object3 = new Object("Object 3");
        Element object4 = new Object("Object 4");
        Element object5 = new Object("Object 5");
        Element object6 = new Object("Object 6");

        Element box1 = new Box("Box 1");
        Element box2 = new Box("Box 2");
        Element box3 = new Box("Box 3");
        Element box4 = new Box("Box 4");
```

Folhas !

Todos são elementos !

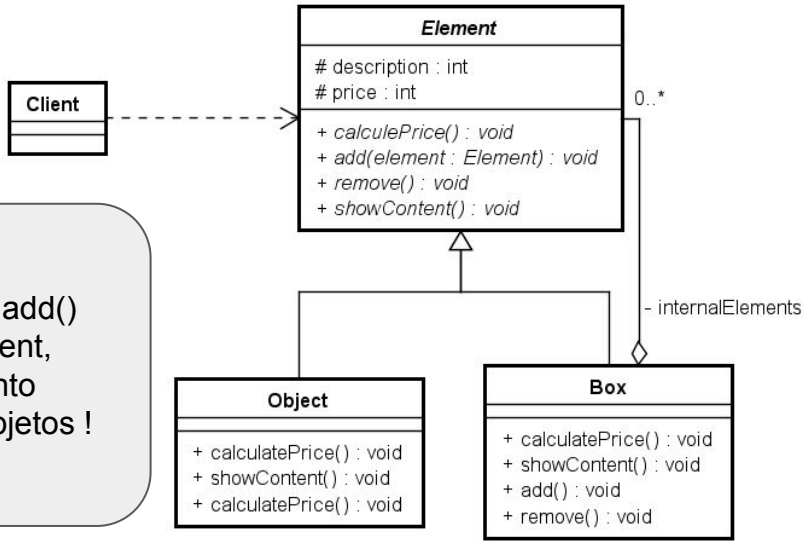
Composite !

This is the client code

COMPOSITE - EXERSAMPLE !

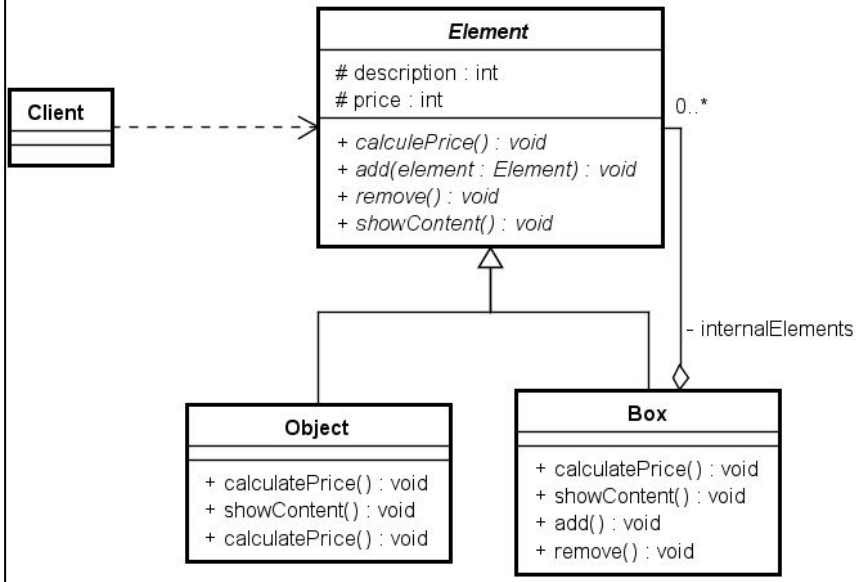
```
box1.add(object1);
box1.add(object2);
box1.add(box2);

box2.add(object3);
box2.add(object4);
box2.add(box4);

box3.add(object5);
box3.add(object6);

box1.showContent();
box2.showContent();
box3.showContent();

System.out.println("----  PRICES -----");
System.out.println("The price of box1 is " + box1.calculatePrice());
System.out.println("The price of box2 is " + box2.calculatePrice());
System.out.println("The price of box3 is " + box3.calculatePrice());
    }
}
```

Como o método add() recebe um Element, posso passar tanto caixas quanto objetos !



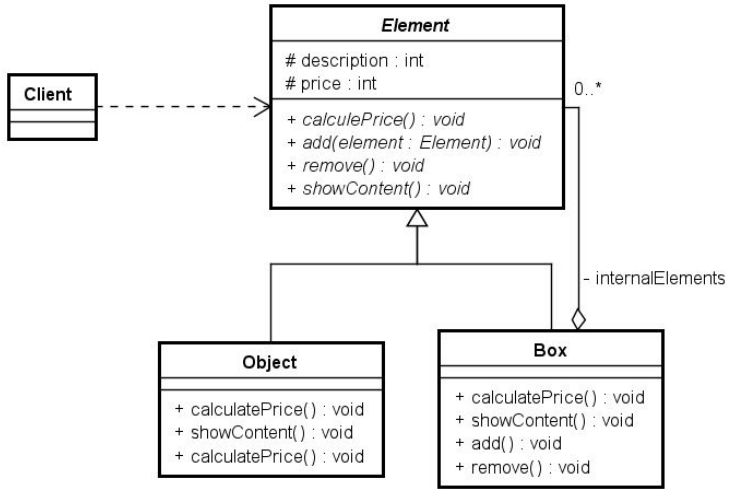
COMPOSITE - EXERSAMPLE !

```
public abstract class Element {

     protected String description;
     protected int price;

     public Element(String description) {
          this.description = description;
     }

     abstract public void add(Element newElement);

     abstract public void remove(int i);

     public abstract void showContent();

     public abstract int calculatePrice();

}
```



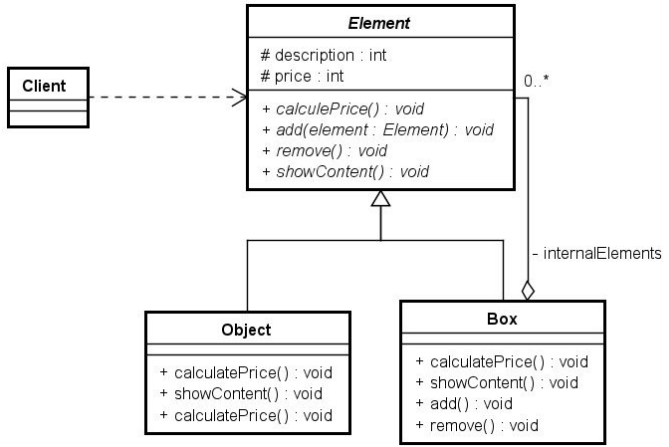
COMPOSITE - EXERSAMPLE !

```
public class Box extends ??? {

     ??????
     List internalElements = Elements[]

     public Box(String description) {
          super(description);
     }

     public void add(Element newElement) {
          if (!newElement.equals(this)) {
               ????.add(newElement);
          }
          else {
               System.out.println("It is not possible to add the same element !");
          }
     }
     public void remove(int i) {
          ????.remove(i);
     }
```



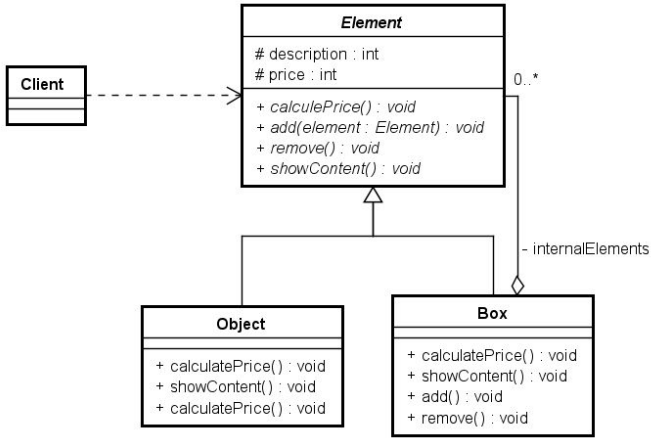
COMPOSITE - EXERSAMPLE !

```
public void showContent() {

      System.out.println("This is the " + this.description);

      for (int i=0; i < ?????.size(); i++) {

            ???? = ????.get(i);  // Element e = internalElements.get(i)
            System.out.println(" --> Element: " + ???.description);
      }
}
public int calculatePrice() {
      int sum = 0;
      ???;

      for (int i=0; i < ?????.size(); i++) {

            ??? = internalElements.get(i);
            sum = sum + element.calculatePrice();
      }
      return sum;
}
```



COMPOSITE - EXERSAMPLE !

```
public class Object extends ???? {

	public Object(String description) {
		super(description);
	}
	public void add(Element newElement) {
		// TODO Auto-generated method stub
	}
	public void remove(int i) {
		// TODO Auto-generated method stub
	}
	public void showContent() {
		System.out.println("This is the " + this.description);
	}
	public int calculatePrice() {
		  return 10;
	}

}
```



COMPOSITE - PHYTON EXERSAMPLE !

```
if __name__ == "__main__":

    object1: Element = Object("Object1")
    object2: Element = Object("Object2")
    object3: Element = Object("Object3")
    object4: Element = Object("Object4")
    object5: Element = Object("Object5")
    object6: Element = Object("Object6")

    box1: Element = Box("Box1")
    box2: Element = Box("Box2")
    box3: Element = Box("Box3")
    box4: Element = Box("Box4")

    box1.add(object1)
    box1.add(object2)
    box1.add(box2)

    box2.add(object3)
    box2.add(object4)
    box2.add(box4)

    box3.add(object5)
    box3.add(object6)

    box1.showcontent()
    box2.showContent()
    box3.showContent()

    print("-----PRICES-----")
    print("The price of box1 is %s", box1.calculatePrice())
    print("The price of box2 is %s", box2.calculatePrice())
    print("The price of box3 is %s", box3.calculatePrice())
```

This is the client code

COMPOSITE - PHYTON EXERSAMPLE !

```
class Element():
    _description: str = None
    _price: int = None

    def __init__(self, description: str) ->None:
        self._description = description

    def add(self, element: Element) ->None:
        raise NotImplementedError

    def remove(self, i: int) -> None:
        raise NotImplementedError

    def showContent(self) -> None:
        raise NotImplementedError

    def calculatePrice(self) -> int:
        raise NotImplementedError
```

COMPOSITE - PHYTON EXERSAMPLE !

```
class Box(???):
    ??????

    def __init__(self, descrption: str) ->None:
        super().__init__(descrption)

    def add(self, element: Element) ->None:
       if isinstance(element, self) :
            print("It is not possible to add the same
element !")
        else:
            ???.attach(element)

    def remove(self, element: Element) ->None:
        ????.remove(element)
```

```
def showContent(self) -> None:
    print("This is the %s", self._description)

    for e in ???:
        print("--> Element: %s", e._description())
```

```
def calculatePrice(self) -> int:
    sum: int = 0
    ???

    for e in ???:
        sum = sum + e.calculatePrice()

    return sum
```

COMPOSITE - PHYTON EXERSAMPLE !

```
class Object(???):

    def __init__(self, description: str) ->None:
        super().__init__(description)

    def add(self, element: Element) ->None:
        pass

    def remove(self, i: int) -> None:
        pass

    def showContent(self) -> None:
        print("This is the %s", self._description())

    def calculatePrice(self) -> int:
        return 10
```

Exercício Proposto

→ para quem quer se aprofundar

Exercício Proposto

A empresa Delta precisa desenvolver um sistema para gerenciar suas centenas de lojas. Uma das funcionalidades é um sistema dinâmico e adaptativo de abastecimento dos estoques. Um das premissas principais é nunca deixar com que determinada loja fique sem estoque e, por consequência, não consiga efetuar a venda de um determinado produto. Há um contrato com os fornecedores que procuram garantir que as prateleiras sempre estejam abastecidas. Entretanto, como a frequência de venda dos produtos varia, podendo acabar repentinamente, o tipo de entrega de novos produtos por fornecedores também varia. Isto é, os fornecedores podem efetuar as entregas de três formas: padrão, rápida ou muito rápida. Obviamente, dependendo do tipo, o valor do produto entregue também varia. Para que haja um equilíbrio nos preços, mas que ao mesmo tempo os estoques sempre fiquem abastecidos, o tipo de entrega pode variar mais de uma vez dentro de uma semana. Por exemplo, na segunda-feira o tipo de entrega foi padrão, já na terça foi necessário mudar para o tipo rápido. Na quarta de manhã voltou a ser padrão mas quarta à tarde já precisou ser alterado para o muito rápido. Dessa forma, dependendo da situação, o tipo de entrega pode mudar várias vezes, inclusive no mesmo dia.

Outra funcionalidade necessária para esse funcionamento é que o sistema do fornecedor seja notificado sempre que determinados produtos atinjam o nível de estoque mínimo. Entretanto, há uma lógica interna que determina o tipo de entrega que deve ser adotado quando determinados níveis de estoque são atingidos. Os termos usados pela empresa são: a) estoque baixo; b) estoque muito baixo c) sem estoque.

Um detalhe também importante é que os produtos da empresa são também padronizados e são sempre vendidos na forma de pacotes, podendo, obviamente existirem pacotes mais complexos que contém outros pacotes internos.

Também é importante ressaltar que a empresa, apesar de trabalhar com vários produtos, classifica eles atualmente em três tipos (Perecíveis, Não Perecíveis e Genéricos). Independentemente do tipo, há algumas operações que devem poder ser executadas para qualquer tipo, por exemplo: validar código, obter data de fabricação, gerar QR Code, etc. Entretanto, a empresa pretende expandir suas atuações para outros tipos de produtos, mas não sabe ainda quais seriam os outros tipos que ela poderia começar a vender. Com base nisso, um requisito não-funcional esperado para o sistema é que o Controlador responsável por manipular objetos Pacote/Produto seja independente dos tipos de produtos existentes.