



Padrões e Desenho de Software Teóricas

Resumos

Introdução ao desenho de *software*

GRASP

Padrões de desenho: Criacionais, estruturais e comportamentais

Padrões de arquitetura de software

Gonçalo Matos, 92972

Licenciatura em Engenharia Informática

2.º Ano | 2.º Semestre | Ano letivo 2019/2020

Última atualização a 25 de maio de 2020

Este documento é baseado nos *slides* teóricos do professor José Luis Oliveira e nos conteúdos de várias páginas da *internet* e livros, devidamente referenciadas no início de cada capítulo onde são utilizados.

Índice

1. Introdução ao desenho de software.....	5
Complexidade do software.....	6
O que devemos considerar?.....	6
Características do desenvolvimento de software.....	7
Processo de desenho.....	7
Padrões.....	8
Os diferentes modelos e a sua representação.....	9
2. GRASP.....	11
Creator.....	12
Information Expert.....	13
Low Coupling.....	14
High Cohesion.....	15
Controller.....	16
Polymorphism.....	17
Pure Fabrication.....	18
Indirection.....	19
Protected Variations.....	20
Liskov Substitution Principle (LSP).....	20
Law of Demeter (Don't talk to strangers).....	20
Outros princípios.....	22
SOLID.....	22
Minimalismo.....	22
DRY (Don't Repeat Yourself).....	22
IoC (Inversion of control).....	22
3. Padrões de desenho.....	23
4. Padrões criacionais.....	25
Class: Factory Method.....	26
Object: Abstract Factory.....	28
Object: Builder.....	30
Object: Singleton.....	32

Object: Object pool.....	33
Object: Prototype.....	35
5. Padrões estruturais.....	37
Class: Adapter.....	38
Subclassing vs. Delegation.....	40
Object: Bridge.....	41
Object: Composite.....	43
Object: Decorator.....	45
Object: Façade.....	47
Object: Flyweight.....	49
Object: Proxy.....	51
6. Padrões comportamentais.....	53
Chain of responsibility.....	54
Command.....	56
Iterator.....	59
Mediator.....	61
Memento.....	63
Null object.....	65
Observer.....	66
State.....	68
Strategy.....	70
Template Method.....	71
Visitor.....	73
Considerações finais.....	75
State, Strategy e Bridge.....	75
7. Padrões de arquitetura de software.....	76
Layered Architecture.....	77
Event-Driven Architecture.....	80
Microkernel Architecture.....	83
Microservices Architecture Pattern.....	85
Space-Based Architecture.....	88

1. Introdução ao desenho de software

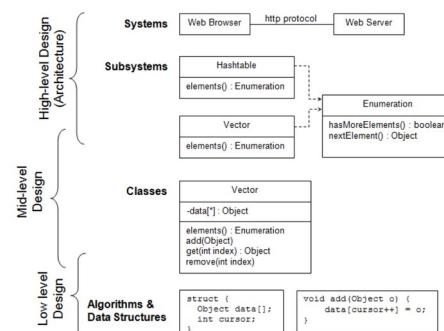
O desenho pode beneficiar a construção de qualquer elemento que seja composto por elementos mais simples.

No *software*, este desenho começa pela definição dos **requisitos**. Este processo deve ser feito pelo programador junto dos *stakeholders*, sendo fundamental este ter uma boa capacidade de comunicação e filtragem, devendo durante o mesmo ficar esclarecidas quais as **necessidades do sistema a implementar**.

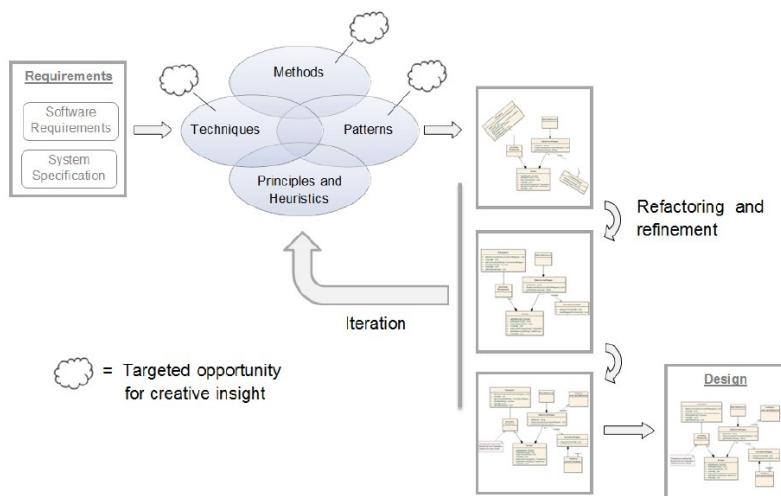
A fase seguinte é o **desenho**, onde se **definem as classes, os métodos, as relações entre elas, os fluxos de trabalho...**

Todo o processo culmina na **construção**, que segue os dois processos anteriores e já consiste na **escrita de código que implemente o sistema delineado**.

É importante que durante estes procedimentos sejam considerados diferentes níveis de detalhe do sistema.



O processo de desenho pode ser tornado mais **previsível e sistemático** através da aplicação de métodos, técnicas e padrões, de acordo com os princípios e a heurística¹.



Complexidade do software

Um programa que seja mal desenhado (demasiado complexo) é **difícil de entender e modificar**, podendo a curto prazo ser mais rápido de desenvolver, mas tornando-se **insustentável a longo prazo**.

O desenho é então uma ferramenta poderosa para o **controlo da complexidade** no desenvolvimento de *software*, gerindo a **complexidade essencial** (associada ao problema) e evitando a **complexidade accidental** (associada à solução e à forma como é implementada).

Complexidade total = complexidade essencial + complexidade accidental

O que devemos considerar?

Para evitar a complexidade no desenho de *software* devemos ter em conta os seguintes princípios:

Modularidade

Dividir o problema em problemas mais pequenos e fáceis de resolver.

#Divide and conquer

Abstração

Utilizá-la para **omitir detalhes** onde não são necessários.

Escondida de informação

Desenvolver **interfaces simples** para detalhes complexos.

Herança

Reutilização de **componentes genéricos** para **definir mais específicos**.

Composição

Reutilizar componentes para criar uma **nova solução**.

Características do desenvolvimento de software

O desenho de software caracteriza-se por ser **não determinístico**ⁱⁱ, pois a maioria dos processos de desenho para o mesmo programa atingem o mesmo resultado, **heurístico**, pois as técnicas de desenho apoiam-se na heurística e em **regras práticas**, ao invés de processos repetitivos e **emergente**ⁱⁱⁱ, pois o **resultado final evolui** de acordo com a experiência e o *feedback*.

É assim fundamental acompanhar todas as fases de desenvolvimento, pois elas estão fortemente interligadas e durante o desenvolvimento de uma pode haver necessidade de realizar alterações nas anteriores.

Com este trabalho é criado um sistema complexo, como resultado de pequenas e simples iterações.

Processo de desenho

Depois de **percebido** o problema através da **definição dos seus requisitos**, é criado um **modelo da solução**, sendo as especificações geralmente representadas com **casos de uso**. Segue-se a **procura por soluções existentes** e **desenhados protótipos**. O desenho é então **revisto** e **refactored**^{iv}. Durante estas duas últimas fases, é importante garantir que o desenho tem as seguintes características:

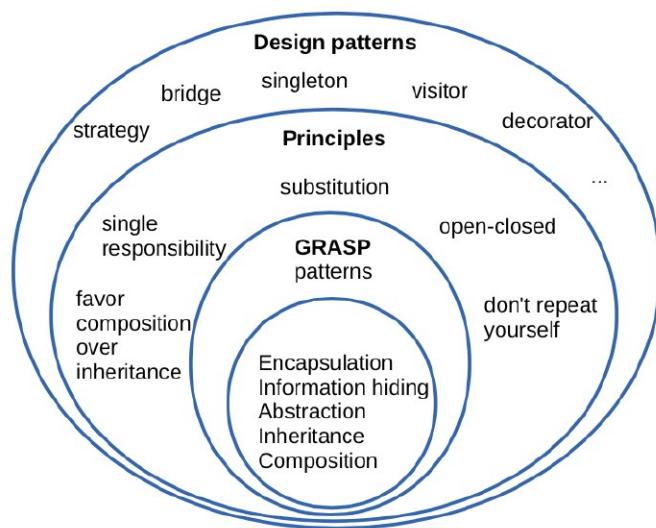
- Minimal complexity – Keep it simple. Maybe you don't need high levels of generality.
- Loose coupling – minimize dependencies between modules
- Ease of maintenance – Your code will be read more often than it is written.
- Extensibility – Design for today but with an eye toward the future. Note, this characteristic can be in conflict with "minimize complexity". Engineering is about balancing conflicting objectives.
- Reusability – reuse is a hallmark of a mature engineering discipline
- Portability – works or can easily be made to work in other environments
- High fan-in on a few utility-type modules and low-to-medium fan-out on all modules. High fan-out is typically associated with high complexity.
- Leanness – when in doubt, leave it out. The cost of adding another line of code is much more than the few minutes it takes to type.
- Stratification – Layered. Even if the whole system doesn't follow the layered architecture style, individual components can.
- Standard techniques – sometimes it's good to be a conformist! Boring is good. Production code is not the place to try out experimental techniques.

Para assegurar todo este processo são necessários os **requisitos do utilizador**, **conhecimento do domínio** em que o sistema de insere (termos técnicos, p.e.) e do **ambiente de execução** em que o programa vai ser implementado (as suas capacidades e limitações).

Padrões

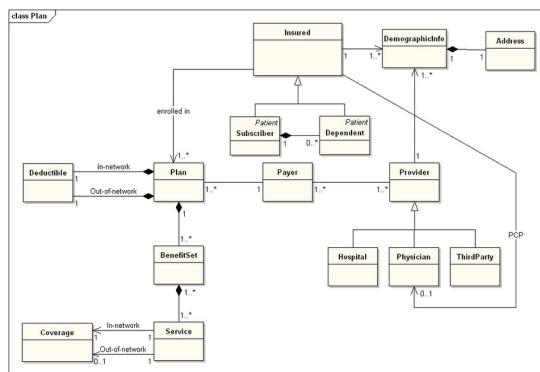
Uma **solução para um problema de desenho comum reutilizável** diz-se um **padrão de desenho**. Estes são adaptados às características do problema que resolvem e dependendo do nível de desenho a que se aplicam, podem ser divididos em **padrões arquiteturais**, **padrões de desenho** ou **idiomas de programação**.

Na falta de uma metodologia específica para definir o melhor desenho orientado a objetos, neste documento serão abordados alguns princípios, padrões e heurísticas.



Os diferentes modelos e a sua representação

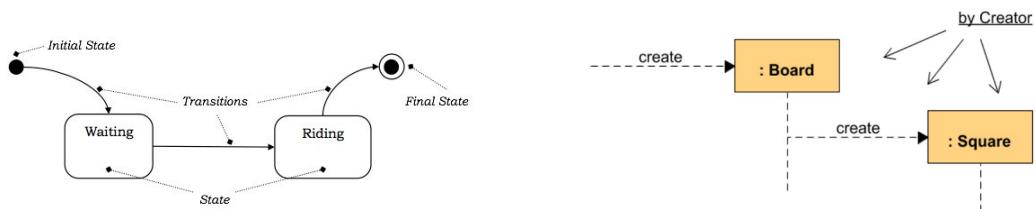
Em diferentes fases do desenho de software são utilizados diferentes modelos para a sua representação. O que **domínio de modelo** representa um modelo conceitual, incorporando tanto o **comportamento**, como a **informação** de cada entidade.



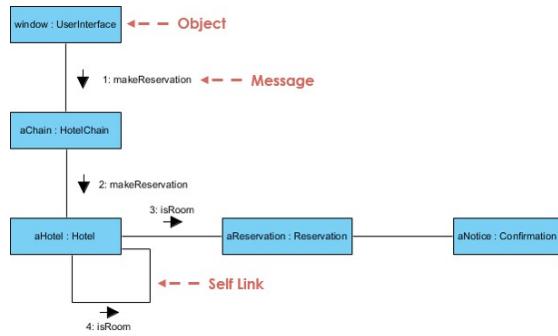
Nestes, a herança e a composição são representadas, respetivamente, por:



É também muito comum a utilização do **domínio dinâmico**, este mais focado na **evolução temporal** dos objetos.



Para modelar as interações num comportamento gerado num caso de uso particular utilizam-se **diagramas de colaboração**.



2. GRASP

Capítulo com base nos slides teóricos e no livro 'Applying UML and Patterns', de Craig Larman, 3.ª Edição, Capítulos 17 e 30

Uma das metodologias de desenvolvimento de *software* mais comuns desgina-se **responsability-driven design (RDD)** e consiste na definição dos objetos em termos de **responsabilidade, papéis e colaborações**.

As responsabilidades podem ser generalizadas em **fazer**, seja criar um objeto, fazer um cálculo, despoletar ações ou coordenar atividades noutros objetos e **saber**, por exemplo, os seus dados privados, objetos relacionados...

Estas por sua vez são implementadas através de métodos, que podem atuar sozinhos ou em colaboração com outros.

Os **General Responsability Assignment Software Patterns**, ou **GRASP**, são uma resposta à criação desta metodologia, descrevendo os princípios fundamentais do desenho e atribuição de responsabilidade aos objetos.

A tradução literal de *grasp* é compreender, sendo esta analogia proposta para destacar a importância do domínio dos princípios fundamentais para ter sucesso no desenvolvimento de *software*.

De uma forma geral, permite-nos atribuir responsabilidades, evitar ou minimizar dependências e acoplamento^v, maximizar a coesão^{vi} e encapsulamento^{vii}, potenciar a reutilização e reduzir a manutenção, exponenciar a legibilidade e facilidade de compreensão do código...

A maioria dos padrões definidos pelos GRASP são conhecimento comum. No entanto, há a necessidade de os estandardizar, dando-lhes um nome e registando o seu funcionamento, facilitando assim a comunicação entre programadores e a memorização por parte dos mesmos.

Creator

#Problema

Quem cria o objeto X? #Fazer

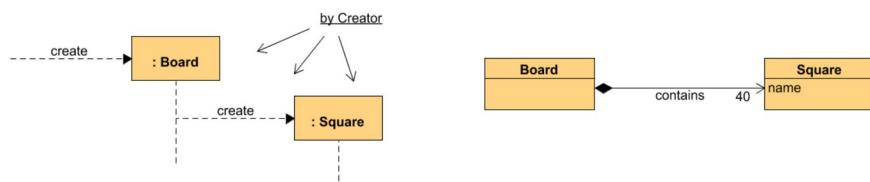
Como resposta, a maioria dos programadores pensa imediatamente no *container* criar o elemento contido. Este raciocínio tem por base o **low representational gap**^{viii} e não está errado.

Na verdade, existem mais alguns critérios, sendo as condições para B criar A listadas abaixo.

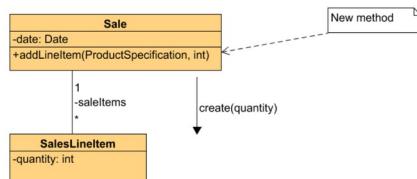
#Solução

- **B contém ou agrupa** (contém um conjunto de) A;
- **B regista A;**
- **B utiliza** com frequência A;
- **B tem os dados de inicialização** de A.

Num jogo de monopólio e segundo este princípio, o **tabuleiro** cria as casas, uma vez que um tabuleiro agrupa (tem uma coleção) de casas.



Num POS (Point of Sale), uma **LinhaDeVenda** é criada pela **Venda**, uma vez que a venda agrupa várias linhas.



#Prós

Minimiza o acoplamento, ao fazer uma instância de uma classe responsável por criar os objetos que referenciam.

#Contras

Pode levar ao **aumento da complexidade**, aquando da reciclagem de instâncias para aumentar a performance e da criação condicional de instâncias através de uma família de classes semelhantes. Nestes casos, aplicam-se outros princípios.

Information Expert

#Problema

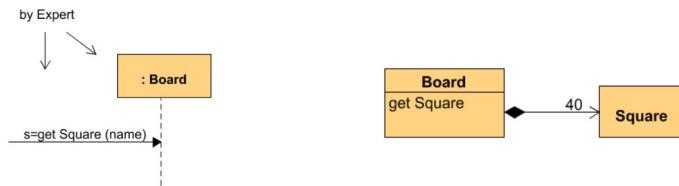
Qual o princípio para atribuir responsabilidades a objetos? #Saber #Fazer

#Solução

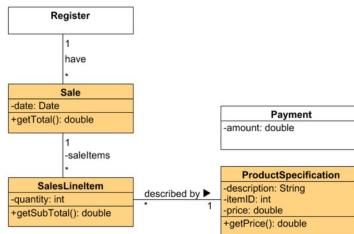
Atribuir a responsabilidade à classe que tem a informação necessária para a assumir.

Este método baseia-se na filosofia de ‘**Não faças nada que podes atribuir a outra pessoa**’.

No monopólio, a questão que está na origem deste princípio pode colocar-se na forma de **Quem tem o conhecimento das casas, dada uma chave?** Como vimos pelo princípio *Creator*, o tabuleiro vai criar as casas, sendo esta a classe que as agrupa e por isso que tem informação sobre elas. Conclui-se assim que o **tabuleiro** é a classe indicada para assumir a responsabilidade de identificar uma casa dada uma chave.



No POS, o responsável por **saber o total de uma venda** será a classe **Venda**, uma vez que é ela que instancia as LinhasDeVenda, tendo por isso a informação necessária para assumir esta responsabilidade. Por sua vez, para calcular o total, a **Venda** necessita de saber o subtotal de cada linha, sendo para esta responsabilidade, a classe **LinhaDeVenda** a *expert* para assumir, uma vez que tem a referência do produto (onde acede ao preço unitário) e a sua quantidade. Por sua vez, para saber o seu preço, cada **produto** é o *expert*.



A questão da complexidade levanta-se na dúvida de quem tem a responsabilidade de guardar a **Venda** na base de dados, sendo esta a classe *expert*, pois com a organização estipulada acima, cada classe apresenta os seus próprios serviços para se guardar na base de dados.

#Prós

Cada classe utiliza assim a informação que dispõe para cumprir tarefas, tornando-se **mais coesas e promovendo o encapsulamento e low coupling**.

#Contras

No entanto, pode tornar uma classe **excessivamente complexa**.

Low Coupling

#Problema

Como reduzir o impacto da mudança e encorajar a reutilização?

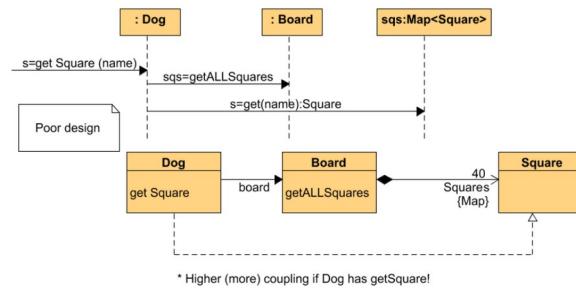
O **acoplamento** define-se como o **quão ligado, informado ou dependente** está um elemento de outro.

Uma classe X acopla a Y outra quando tem um **atributo** para uma instância da Y, quando um dos seus **métodos** instancia a Y, quando X é uma **subclasse** de Y ou Y é uma interface implementada por X.

#Solução

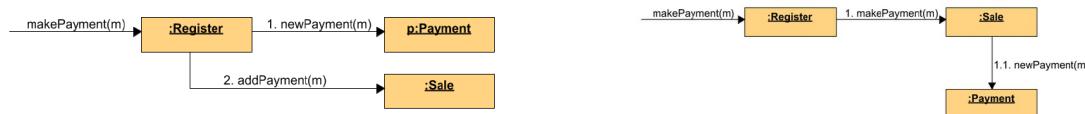
Atribuir responsabilidades de forma a que o acoplamento se mantenha reduzido, tentando que uma classe conheça o menor número de outras.

No monopólio não faz sentido atribuir a responsabilidade de obter uma **casa** a uma classe **cão**, quando este teria de consultar o **tabuleiro**, a entidade *expert* sobre as casas.



Outro exemplo é a **peça**. Normalmente esta estaria associada a uma **casa**, que por sua vez está associada ao **tabuleiro**. Caso o **tabuleiro** tivesse informações da **peça** sem consultar a **casa**, seria estariam perante um cenário de *higher coupling* do que o anteriormente descrito.

No POS, o método que cria um **Pagamento**, segundo este princípio deve estar na **Venda**, apesar de ser despoletado na sequência de outro método na **Venda** pela **CaixaRegistadora**.



#Prós

O código torna-se **mais fácil de alterar/manter** (mudanças tornam-se mais localizadas), de **entender** e de **reutilizar**, tornando-lo mais eficiente.

No entanto, em classes estáveis, um acoplamento maior não é um grande problema.

Os princípios do método *Information Expert*, visto anteriormente, servem de suporte a este princípio, pois procuramos a classe que tem mais informação para cumprir a responsabilidade.

É impossível ter um acoplamento nulo, uma vez que é também fundamental que as classes troquem mensagens entre si. Deve é ser na medida certa!

High Cohesion

#Problema

Como manter as classes focadas, percutíveis e maleáveis?

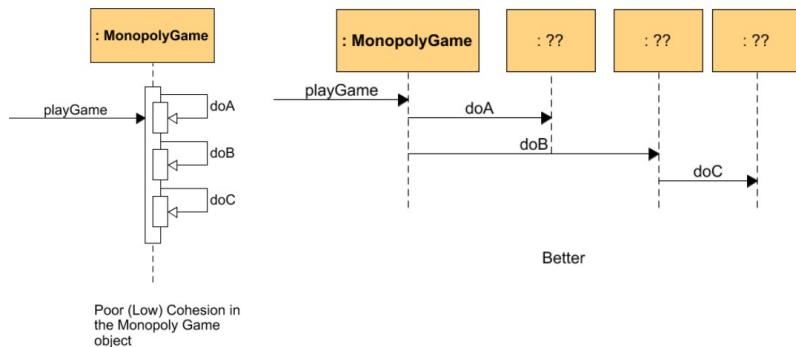
#Solução

Atribuir responsabilidades de forma a manter uma alta coesão.

A **coesão** diminui com o aumento da quantidade de código e da diversidade de funcionalidades. O objetivo é então delegar responsabilidades e coordenar o trabalho.

Geralmente uma baixa coesão está aliada a um elevado acoplamento.

No monopólio, uma implementação em que a classe **Jogo** faça todas as tarefas é pouco coesa, sendo preferível uma em que esta delega outras as várias tarefas, coordenando-las.



No exemplo do POS dado no princípio do *Low Coupling*, o ideal também cumpre os princípios de alta coesão.

#Prós

Este princípio torna o código **mais fácil de compreender, manter, complementando o baixo acoplamento**.

#Contras

No entanto, por vezes é necessário criar objetos pouco coesos, nomeadamente nas comunicações e objetos remotos, em que é necessário criar uma única interface para várias operações.

Controller

#Problema

Quem deve ser responsável por eventos UI?

#Solução

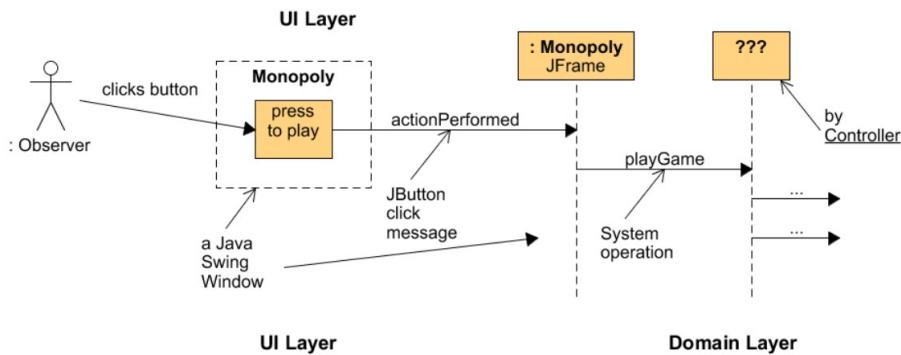
Criar uma classe que gira o evento, separando e fazendo a ponte entre as fontes do evento e os objetos que lidam com estes.

Esta classe pode ser construída de acordo com duas abordagens:

- Ser **representativa do modelo de negócio** em que se insere, denominada de **façade controller**
- Ser artificial, seguindo o **método Pure Fabrication**, denominada de **case controller**

Limita-se a encaminhar os eventos e o *output* dos métodos ativados por este.

Num monopólio com interface visual interativa, quando o utilizador pressionar o botão para iniciar um novo **Jogo**, este evento deve ser gerido por uma classe do tipo **controller**, e não diretamente pelo **Jogo**.



#Prós

Ao separar os eventos externos dos seus gestores internos, cria-se uma **abstração quanto ao tipo e comportamento das instâncias** ligados pelo **controller**, que podem ser modificadas sem interferir uma com a outra.

Permite-nos ainda **garantir que as operações são realizadas numa sequência válida**, prevenindo assim a realização de ações ilegais que podem levar à criação de erros.

Polymorphism

#Problema

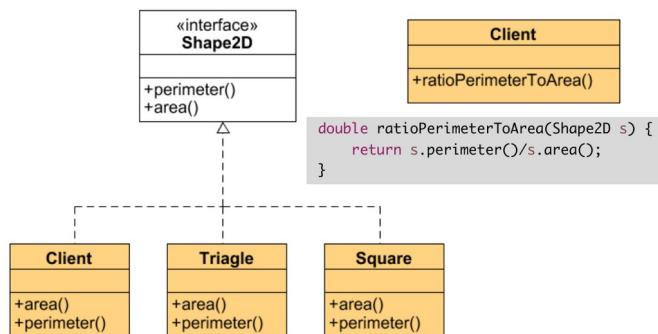
Como gerir o comportamento com base no tipo, mas sem instruções condicionais?

#Solução

Utilizar métodos polimórficos, ou seja, atribuir o mesmo nome a serviços (métodos) distintos em classes diferentes.

Numa aplicação que lide com formas geométricas 2D, sabemos que existem formas diferentes cujo perímetro e área têm fórmulas distintas. Para evitar numa função que lide com todas a necessidade de determinar o seu tipo, podemos utilizar o polimorfismo.

Fazemos assim com que todas as classes que representam uma forma geométrica implementem uma interface comum, que basicamente determina os métodos que estas vão implementar.



#Prós

Torna o código mais **robusto** e **fácil de escrever e aumentar** no futuro (com novas funcionalidades).

#Contras

Aumenta o número de classes, o que pode levar a uma maior **dificuldade na compreensão** do código.

Pure Fabrication

#Problema

Que objeto deve assumir uma responsabilidade quando nenhuma das classes do problema a pode assumir sem violar os princípios do low coupling e high cohesion?

Esta pergunta coloca-se pois nem todas as responsabilidades se enquadram no domínio das classes, como por exemplo as comunicações, interação com o utilizador...

#Solução

Atribuir um conjunto coeso de responsabilidades a uma classe artificial que não representa nada no domínio no problema.

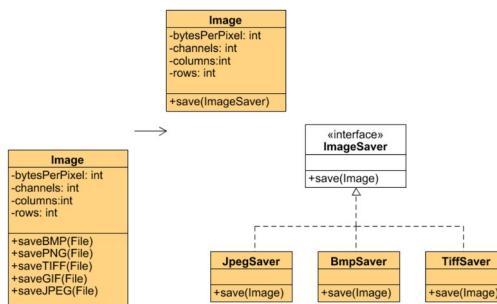
Por exemplo, no POS, se quisermos guardar o registo de cada venda numa base de dados relacional, pelo princípio *Information Expert* faria sentido atribui-la à classe **Venda**, no entanto, com esta atribuição a classe iria tornar-se pouco coesa e fortemente acoplada à base de dados.

A solução que este princípio propõe é criar uma nova classe, responsável por guardar elementos na base de dados relacional, que vai ser invocada pela classe **Venda**.



Esta classe, por sua vez, pode ser fortemente reutilizada.

Outro exemplo é numa classe **Imagen**, que segundo este princípio deve ser abstraída das operações de ser guardada em diferentes formatos.



#Prós

Aliado ao princípio da **alta coesão**, uma vez que a classe artificial tem um foco muito específico.

Potencial para a **reutilização**.

Indirection

ix

#Problema

Como evitar acoplamento direto, desacoplando sem perder o potencial de reutilização?

#Solução

Atribuir a responsabilidade a um objeto intermédio.

No princípio anterior, a criação de uma classe para gerir as interações com a base de dados é também um exemplo da aplicação deste método.

Outro cenário em que este se aplica é no POS, nomeadamente devido à necessidade de estabelecimento de um canal de comunicação com um siste de pagamento para validar uma transação. Para tal, deve ser criada uma classe responsável por esta operação.

#Prós

Permite **baixo acoplamento** e promove a **reutilização**.

Protected Variations

#Problema

Como desenhar software de forma a que as suas variações não tenham um impacto negativo noutras elementos?

#Solução

Identificar os pontos de instabilidade e atribuir responsabilidades de forma a criar uma interface estável em sua volta.

Serve de base a muitos padrões de desenho!

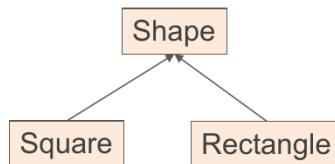
Liskov Substitution Principle (LSP)

Princípio com base no *protected variations*, defende que sendo B uma subclasse de A, os **objetos de A devem poder ser substituídos por B sem alterar a normal execução do programa**, ou seja:

- B não deve remover métodos implementados em A;
- Cada método de B que reescreva um definido em A não deve alterar o seu comportamento esperado, ou seja, **não deve surpreender cliente!**

Voltando ao exemplo das figuras geométricas, do ponto de vista matemático, um quadrado é um retângulo com os lados todos iguais. No entanto, num programa, onde o cliente espera encontrar uma instância de um retângulo, se lhe apresentarmos um quadrado, não vamos dar ao cliente o que ele espera, uma vez que ele não pode manipular as dimensões de forma a ter uma largura diferente do comprimento.

Devemos então, por forma a garantir a estabilidade da sua utilização, eliminar esta relação entre o quadrado e o retângulo, tornando-los classes distintas.



Law of Demeter (Don't talk to strangers)

Este é outro princípio que deriva do *protected variations*, estabelecendo que de forma a **evitar que objetos conheçam a estrutura de outros que não lhe estão diretamente ligados**, cada objeto só deve comunicar consigo, os seus parâmetros e atributos, ou objetos criados pelos seus métodos.

Numa **Empresa** dividida em **Departamentos**, cada um com um gestor, que é uma instância da classe **Empregado**, se a empresa quiser saber o total de dinheiro que paga aos gestores não deve invocar um método no **Empregado**, mas sim no **Departamento**, uma vez que este é seu atributo (o empregado não!).

#Prós

Diminui o encapsulamento e torna o desenho mais **rubosto**.

#Contras

Os métodos indiretos podem levar ao **aumento de overhead**^x.

Outros princípios

SOLID

Single Responsibility

Cada classe deve **assumir uma única responsabilidade**, mas assumi-la em pleno (encapsulamento máximo).

Open/closed

As classes devem estar **abertas à extensão**, mas **fechadas à alteração**.

Liskov substitution

Ver [Protected Variations, do GRASP](#)

Interface segregation

Nenhum cliente deve ser forçado a depender de métodos que não utiliza.

Ver [High Cohesion, do GRASP](#)

Dependency inversion

Módulos de alto nível não devem depender de módulos de nível inferior, devendo ambos depender de abstrações.

Minimalismo

DRY (Don't Repeat Yourself)

IoC (Inversion of control)

3. Padrões de desenho

Define-se por padrão **princípios e soluções estruturalmente codificados que descrevem uma solução** para um determinado problema, podendo ser aplicado em diversos contextos.

Não são mais do que soluções com que programadores no passado se depararam e decidiram anotar, para mais tarde recordar e partilhar com outros. Trocado por miúdos, não passam de conselhos.

Como resultado, facilitou-se a reutilização de código orientado a objetos entre programas e programadores.

Um **bom padrão** caracteriza-se por **resolver um problema cuja solução não é óbvia, descrevendo uma relação que é um conceito provado, com algum componente humano envolvido.**

Os padrões podem ser classificados em três tipos:

Arquiteturais, quando expressam uma **estrutura organizacional do sistema como um todo**;

De desenho, quando estabelece um **esquema para os subsistemas ou componentes** de um sistema, ou as suas relações;

Idiomáticos, se descrevem como implementar um **aspetto particular de um componente**, ou as relações entre eles.

Neste capítulo serão abordados os padrões definidos no livro “Design patterns: elements of reusable object oriented software”, do gang of four (4 escritores), que cataloga 23 padrões, divididos em três grupos:

Creational, relacionados com a criação de objetos;

Structural, relacionados com a composição de classes e objetos;

Behavioral, caracterizam a forma como classes e objetos interagem e distribuem responsabilidades.

By Purpose				
		Creational	Structural	Behavioral
By Scope	Class	<ul style="list-style-type: none"> Factory Method 	<ul style="list-style-type: none"> Adapter (class) 	<ul style="list-style-type: none"> Interpreter Template Method
	Object	<ul style="list-style-type: none"> Abstract Factory Builder Prototype Singleton 	<ul style="list-style-type: none"> Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy 	<ul style="list-style-type: none"> Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Estes estão fortemente interligados.



Um bom programador deve, para além de conhecer conceitos de POO como herança, encapsulamento ou notação UML, de saber princípios OO, exemplos de bons desenhos e padrões de desenho!

4. Padrões criacionais

Capítulo com base nos slides teóricos e no site sourcemaking.com e refactoring.guru

No que toca à construção de objetos, há dois grandes problemas:

O construtor de um objeto **não pode devolver um subtipo do seu tipo**;

Solução: Padrões Factory

O construtor **devolve sempre uma nova instância** daquela classe, não permitindo a sua reutilização.

Solução: Padrões "Sharing" (Singleton)



Abstract Factory

Criar uma instância a partir de várias famílias de classes.



Factory Method

Criar uma instância a partir de várias classes derivadas.



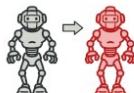
Builder

Separa a construção de um objeto da sua representação.



Singleton

Uma classe da qual só pode existir uma instância.



Prototype

Permite a clonagem de um objeto sem estar dependente da sua classe.



Object Pool

Evitar construções e eliminações que consomem muitos recursos através da reutilização de objetos que já não estão a ser utilizados.

Class: Factory Method

#Intenção

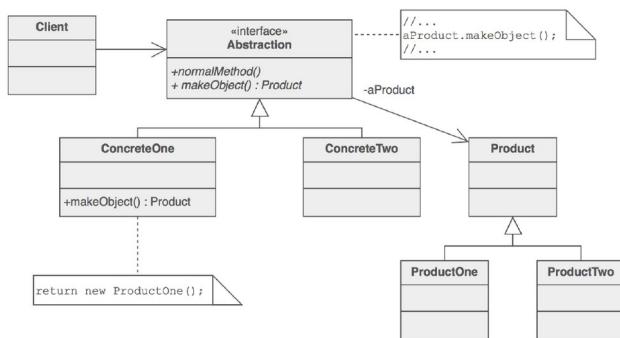
Definir uma interface para criar um objeto numa superclasse, permitindo às subclasses alterar o tipo de objeto a ser criado, através de um construtor virtual.

#Problema

É necessário estandardizar a arquitetura, continuando a permitir a aplicações individuais definir os seus domínios.

#Solução

Substituir as chamadas aos construtores de cada um dos objetos a criar, por chamadas ao **método de fábrica (estático)**, passando um argumento que defina o **tipo de objeto a construir**, sendo devolvido um objeto deste tipo, também designado por produto.



#Check list

- ✓ Se hierarquia tiver herança, considerar definir o método de fábrica na classe base;
- ✓ Os argumentos do método devem permitir distinguir com clareza a classe derivada a instanciar;
- ✓ Considerar a criação de uma reserva de objetos criados, evitando assim a sua criação duplicada;
- ✓ Considerar fazer todos os construtores *private* ou *protected*.

#Prós

Permite que devolver uma instância de uma subclasse, a reutilização de um objeto já criado. Os métodos podem ter nomes mais sugestivos do que os construtores.

É possível adicionar novos métodos de fábrica para novos produtos sem alterações para o cliente!

#Exemplos

1. Num **Viveiro**, para criar uma **Árvore**, podemos ter um método de fábrica que aceite como argumento uma *string* e devolva um objeto do tipo **Árvore**, mas de uma subclasse. Assim, o programa principal fica livreto de fazer *new*, limitando-se a invocar o método estátido no **Viveiro**.

2.

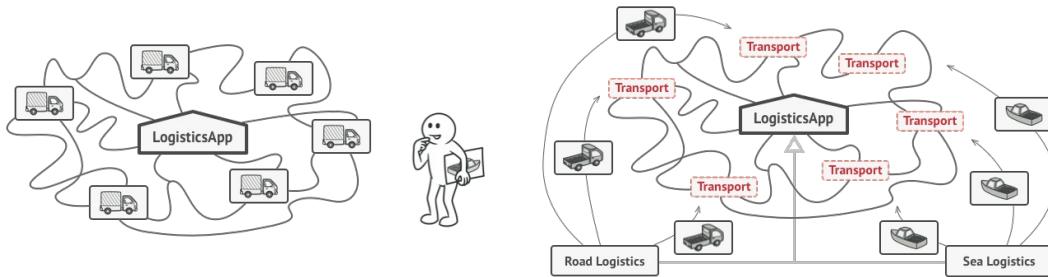
```
class Race {
    Race createRace() {
        Bicycle bike1 = new Bicycle();
        Bicycle bike2 = new Bicycle(); //...
    }
}
class TourDeFrance extends Race {
    Race createRace() {
        Bicycle bike1 = new RoadBicycle();
        Bicycle bike2 = new RoadBicycle(); //...
    }
}
```

```
class Race {
    Bicycle createBicycle() {
        return new Bicycle();
    }
    Race createRace() {
        Bicycle bike1 = createBicycle();
        Bicycle bike2 = createBicycle(); //...
    }
}
class TourDeFrance extends Race {
    Bicycle createBicycle() {
        return new RoadBicycle();
    }
}
```

Podem ser adicionadas novas corridas e bicicletas, sem modificações para o cliente.

3. Uma transporta faz as suas entregas com carrinhos. No entanto, a determinada altura com o crescimento do volume de entregas e da sua abrangência geográfica começa a operar entregas em navios, o que não é suportado pelo seu programa.

Para evitar este cenário e que o mesmo se repita cada vez que a transportadora comece a operar com um novo tipo de veículo, faz sentido utilizarmos o método de fábrica, definindo uma classe abstrata representante da **logística** e uma subclasse da mesma para cada tipo de veículo, devolvendo cada um um objeto deste tipo.



Object: Abstract Factory

#Intenção

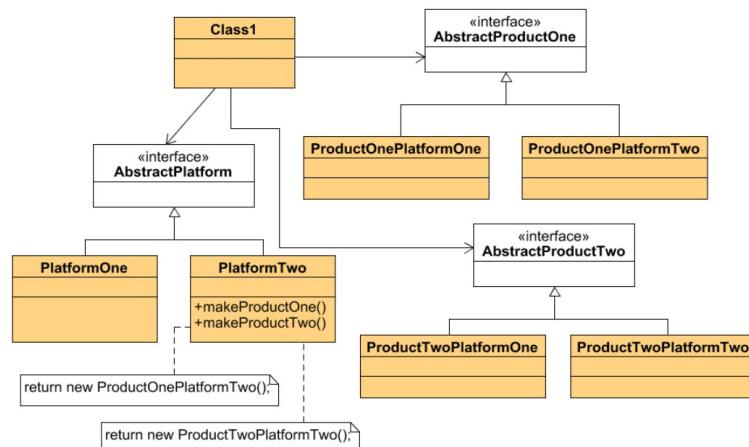
Criar uma **interface para criar famílias de objetos** relacionados sem especificar a sua classe, através de uma hierarquia que abranja **várias plataformas e a construção de vários produtos**, evitando a utilização do operador *new*.

#Problema

Devendo uma aplicação ser portátil, é necessário que encapsule as suas dependências, seja o sistema de janelas, sistema operativo, base de dados... Este **encapsulamento deve ser previsto no desenho do software**.

#Solução

Criar uma interface por produto, definindo para cada uma uma subclasse para cada plataforma e um método **abstract factory** com a interface e uma subclasse para cada plataforma, definindo um método para cada objeto, sendo o devolvido o correspondente a essa plataforma.

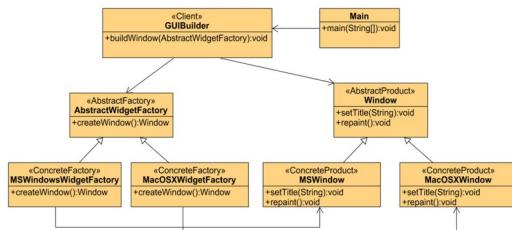


#Checklist

- ✓ Verificar se a independência da plataforma e a criação de serviços são um problema;
- ✓ Planear a relação plataforma/produto;
- ✓ Definir **uma interface para o método de fábrica por produto**;
- ✓ Definir **uma classe derivada para cada plataforma** que encapsula todas as referências ao operador *new*;
- ✓ O cliente deve deixar de fazer referência ao operador *new*, utilizando o método de fábrica para criar os produtos.

#Exemplo

1.

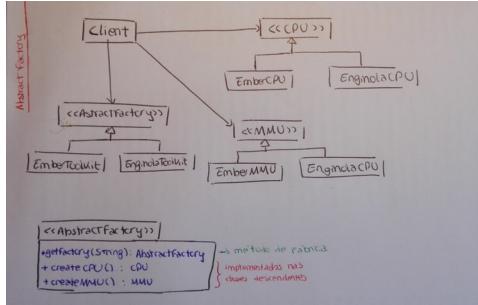


```

public class MainTest {
    public static void main(String[] args) {
        GUIBuilder builder = new GUIBuilder();
        if (Platform.currentPlatform() == "MACOSX")
            builder.buildWindow(new Mac OSX WidgetFactory());
        else if (Platform.currentPlatform() == "WIN")
            builder.buildWindow(new MsWindowsWidgetFactory());
        else //...
    }
}

public class GUIBuilder {
    public void buildWindow(AbstractWidgetFactory widgetFactory) {
        Window window = widgetFactory.createWindow();
        window.setTitle("New Window");
    }
}
  
```

2. [+](#)



```

abstract class AbstractFactory {
    private static final EmberToolkit EMBER_TOOLKIT = new EmberToolkit();
    private static final EnginolaToolkit ENGINOLA_TOOLKIT = new EnginolaToolkit();

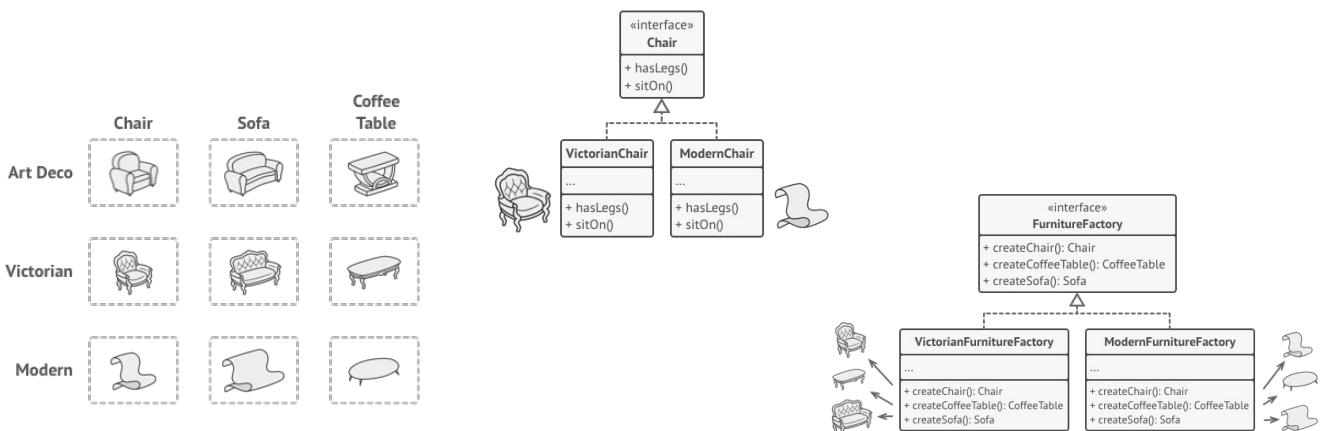
    // Returns a concrete factory object that is an instance of the
    // concrete factory class appropriate for the given architecture.
    static AbstractFactory getFactory(Architecture architecture) {
        AbstractFactory factory = null;
        switch (architecture) {
            case ENGINOLA:
                factory = ENGINOLA_TOOLKIT;
                break;
            case EMBER:
                factory = EMBER_TOOLKIT;
                break;
        }
        return factory;
    }

    public abstract CPU createCPU();

    public abstract MMU createMMU();
}

public class Client {
    public static void main(String[] args) {
        AbstractFactory factory = AbstractFactory.getFactory(Architecture.EMBER);
        CPU cpu = factory.createCPU();
    }
}
  
```

3.



Object: Builder

#Intenção

Separar a construção de um objeto complexo da sua representação, de forma a que a construção (passo a passo) possa criar diferentes representações.

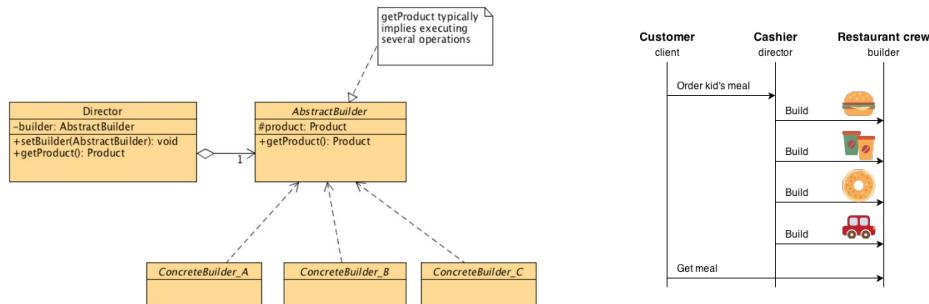
#Problema

A construção de um objeto complexo geralmente implica um construtor com imensos parâmetros, alguns dos quais nem sempre necessitamos. Uma solução seria ter uma classe base com os parâmetros principais e as variações tomarem forma em subclasses. No entanto, são tantas que iríamos exponenciar a complexidade.

#Solução

Retirar a criação do objeto da sua classe e movê-la para outra, o *builder*, um elemento secundário onde os vários elementos são armazenados para depois serem representados no primário (o objeto).

Devemos assim de criar um *abstract builder* que é responsável por fazer os *sets* e um *Director*, responsável por gerir a construção do objeto.



Ou, **caso o objeto tenha muitos atributos**, criar um *builder inner class*.

Uma classe que utiliza este padrão é a *StringBuilder*. Sendo a classe *String* do tipo *final*, esta oferece uma alternativa à criação única dos objetos desse tipo, providenciando um armazenamento secundário e manipulável (com *append()*), devolvendo o objeto *String* construído aquando da invocação do método *toString()*.

#Checklist

- ✓ Verificar se o problema consiste em ter um *input* comum e vários *outputs* (representações) possíveis;
- ✓ Encapsular o *input* numa classe *Director*;
- ✓ Desenhar um protocolo para criar todas as representações possíveis;
- ✓ Definir uma classe derivada do *builder* para cada representação;
- ✓ O cliente cria um *Director* e um *Builder*, passando o último ao primeiro, pedindo de seguida ao *Director* que construa e devolva o produto.

#Exemplos

1. Abstract builder e Director

```
class Pizza { /* "Product" */
    private String dough;
    private String sauce;
    private String topping;
    public void setDough(String dough) { this.dough = dough; }
    public void setSauce(String sauce) { this.sauce = sauce; }
    public void setTopping(String topping) { this.topping = topping; }
}

abstract class PizzaBuilder { /* "Abstract Builder" */
    protected Pizza pizza;
    public Pizza getPizza() { return pizza; }
    public void createNewPizzaProduct() { pizza = new Pizza(); }

    public abstract void buildDough();
    public abstract void buildSauce();
    public abstract void buildTopping();
}
```

```
/* "ConcreteBuilder" */
class HawaiianPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("cross"); }
    public void buildSauce() { pizza.setSauce("mild"); }
    public void buildTopping() { pizza.setTopping("ham+pineapple"); }
}

/* "ConcreteBuilder" */
class SpicyPizzaBuilder extends PizzaBuilder {
    public void buildDough() { pizza.setDough("pot baked"); }
    public void buildSauce() { pizza.setSauce("hot"); }
    public void buildTopping() { pizza.setTopping("pepperoni+salami"); }
}

class Waiter { /* "Director" */
    private PizzaBuilder pizzaBuilder;
    public void setPizzaBuilder(PizzaBuilder pb) { pizzaBuilder = pb; }
    public Pizza getPizza() { return pizzaBuilder.getPizza(); }

    public void constructPizza() {
        pizzaBuilder.createNewPizzaProduct();
        pizzaBuilder.buildDough();
        pizzaBuilder.buildSauce();
        pizzaBuilder.buildTopping();
    }
}
```

O *concrete builder* é uma aplicação do método *abstract factory*.

2. Builder inner class

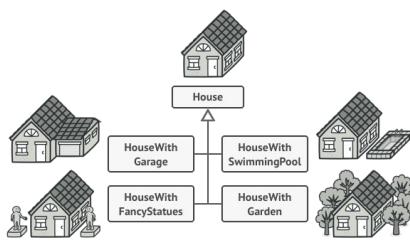
```
public class NutritionFacts { // Builder Pattern
    private final int servingSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingSize;
        private final int servings;
        // Optional parameters - initialized to default values
        private int calories = 0;
        private int fat = 0;
        private int carbohydrate = 0;
        private int sodium = 0;
        public Builder(int servingSize, int servings) {
            this.servingSize = servingSize; this.servings = servings;
        }
        public Builder calories(int val) {
            calories = val;
            return this;
        }
        public Builder fat(int val) {
            fat = val;
            return this;
        }
        public NutritionFacts build() {
            return new NutritionFacts(this);
        }
    } // end of class Builder

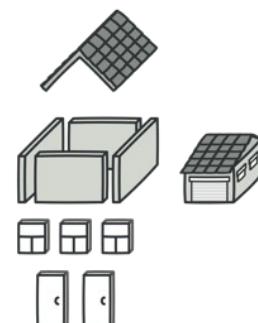
    private NutritionFacts(Builder builder) {
        servingSize = builder.servingSize;
        servings = builder.servings;
        calories = builder.calories;
        fat = builder.fat;
        sodium = builder.sodium;
        carbohydrate = builder.carbohydrate;
    }
}

We can now use this static inner class as follows:
NutritionFacts sodaDrink = new NutritionFacts.Builder(240, 8).
    calories(100).sodium(35).carbohydrate(27).build();
```

3.



```
HouseBuilder
...
+ buildWalls()
+ buildDoors()
+ buildWindows()
+ buildRoof()
+ buildGarage()
+ getResult(): House
```



Object: Singleton

#Intenção

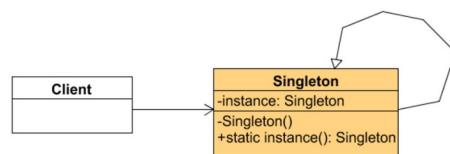
Garantir que uma classe tem uma única instância e fornecer um ponto de acesso global à mesma, com inicialização na primeira utilização (*just-in-time* ou *lazy*).

#Problema

É impossível de implementar com um construtor, uma vez que este cria uma nova instância do objeto por defeito. A criação de um ponto de acesso global também é um desafio, tal como a *lazy initialization*.

#Solução

Definir o construtor como privado, a par de um atributo estático para uma instância de si próprio e um método também estático para lhe aceder.



#Check list

- ✓ Definir o construtor como privado;
- ✓ Definir um atributo estático do seu tipo na classe;
- ✓ Definir um método público para lhe aceder
 - ✓ Com *lazy initialization* (criar se ainda não existe)
 - ✓ Este deve ser o único método utilizado pelos clientes para lhe aceder.

#Exemplo

Um país só pode ter um governo. Mesmo que os elementos que o compõe mudem, em funções só existe uma instância do governo.

Object: Object pool

#Intenção

Aumentar a eficiência na utilização de objetos com custos elevados de inicialização que são utilizados frequentemente mas cuja instância simultânea é reduzida.

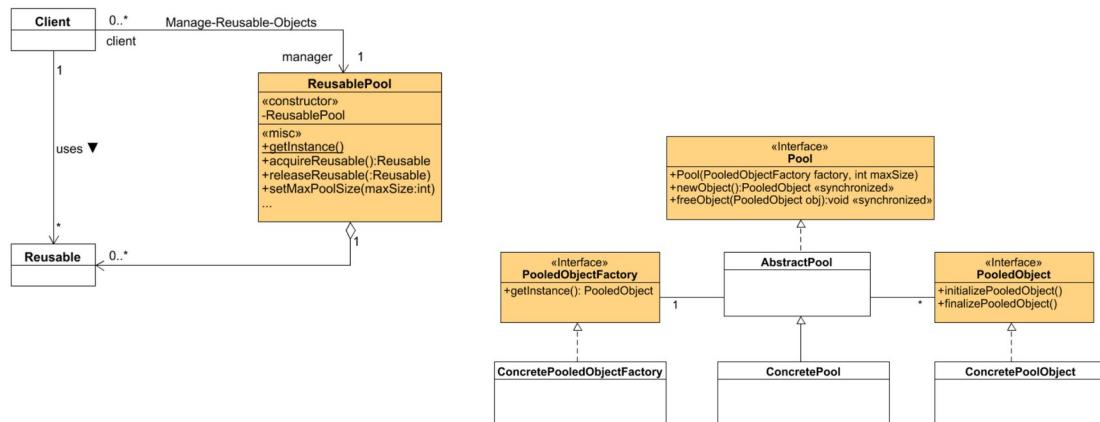
Por exemplo conexões de rede/BD, *threads*...

#Problema

É desejável manter todos os objetos disponíveis na mesma *pool*, de forma à sua reutilização ser feita de forma coerente.

#Solução

Criar uma classe responsável por gerir os objetos reutilizáveis.



#Check list

- ✓ Criar uma classe do tipo *pool* com uma coleção de *pool objects*;
- ✓ Criar métodos *acquire()* e *release()* nesta classe.

#Exemplo

```

public class AbstractPool implements Pool {
    protected final int MAX_FREE_OBJECT_INDEX;

    protected PooledObjectFactory factory;
    protected PooledObject[] freeObjects;
    protected int freeObjectIndex = -1;

    /**
     * @param factory the object pool factory instance
     * @param maxSize the maximum number of instances stored in the pool
     */
    public AbstractPool(PooledObjectFactory factory, int maxSize)
    {
        this.factory = factory;
        this.freeObjects = new PooledObject[maxSize];
        MAX_FREE_OBJECT_INDEX = maxSize - 1;
    }

    /**
     * Stores an object instance in the pool to make it available for a subsequent
     * call to newObject() (the object is considered free).
     * @param obj the object to store in the pool and that will be finalized
     */
    public synchronized void freeObject(PooledObject obj)
    {
        if (obj != null) {
            // Finalize the object
            obj.finalizePooledObject();
            // put an object in the pool only if there is still room for it
            if (freeObjectIndex < MAX_FREE_OBJECT_INDEX) {
                freeObjectIndex++;
                // Put the object in the pool
                freeObjects[freeObjectIndex] = obj;
            }
        }
    }
}

    /**
     * Creates a new object or returns a free object from the pool.
     * @return a PooledObject instance already initialized
     */
    public synchronized PooledObject newObject() {
        PooledObject obj = null;

        if (freeObjectIndex == -1) {
            // There are no free objects so I just
            // create a new object that is not in the pool.
            obj = factory.getInstance();
        } else {
            // Get an object from the pool
            obj = freeObjects[freeObjectIndex];
            freeObjectIndex--;
        }
        obj.initializePooledObject();
        return obj;
    }
}

```

Object: Prototype

#Intenção

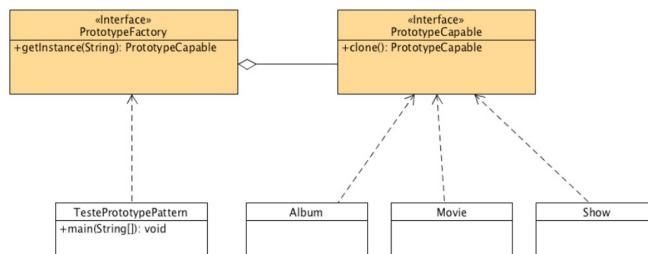
Criar objetos novos a partir de existentes, como protótipos, sem estar dependente das suas classes.

#Problema

A cópia de um objeto pressupõe a construção com os mesmos parâmetros e a definição dos restantes (não definidos no construtor) um a um. No entanto, por vezes há atributos privados a que não conseguimos aceder “de fora”.

#Solução

Delegar o processo de clonagem no objeto a ser copiado, através de um método declarado numa interface comum a todos os objetos passíveis de clonagem.



Os objetos que implementam esta interface dizem-se **prototypes**.

Os atributos privados podem ser copiados sem problema, uma vez que é possível um objeto aceder a um atributo privado de outro.

#Check list

- ✓ Adicionar um método *clone()* ao produto;
- ✓ Desenhar um registo que mantém a *cache* dos objetos prototípaveis;
 - ✓ Pode estar encapsulado numa classe do tipo *factory*, ou na classe base da hierarquia do produto;
 - ✓ Pode ou não aceitar argumentos, descobre o protótipo correto a clonar e invoca o método *clone()*, retornando o seu resultado;
- ✓ O cliente substitui todas as referências ao operador *new* e substitui-las por invocações ao método de fábrica.

#Exemplo

1.

```

public interface PrototypeCapable extends Clonable
{
    public PrototypeCapable clone() throws CloneNotSupportedException;
}

public class PrototypeFactory {
    public static enum ModelType {
        MOVIE, ALBUM, SHOW;
    }

    private static Map<ModelType, PrototypeCapable> prototypes =
        new HashMap<>();
    static {
        prototypes.put(ModelType.MOVIE, new Movie());
        prototypes.put(ModelType.ALBUM, new Album());
        prototypes.put(ModelType.SHOW, new Show());
    }

    public static PrototypeCapable getInstance(ModelType s)
        throws CloneNotSupportedException {
        return (prototypes.get(s)).clone();
    }
}

public class Album implements PrototypeCapable
{
    private String name = null;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    @Override
    public Album clone() throws CloneNotSupportedException {
        System.out.println("Cloning Album object..");
        return (Album) super.clone();
    }
    @Override
    public String toString() {
        return "Album";
    }
}

the same for Movie, Show, ...

```

```

public class TestPrototypePattern {
    public static void main(String[] args) {
        try {
            PrototypeCapable proto;
            proto = PrototypeFactory.getInstance(ModelType.MOVIE);
            System.out.println(proto);

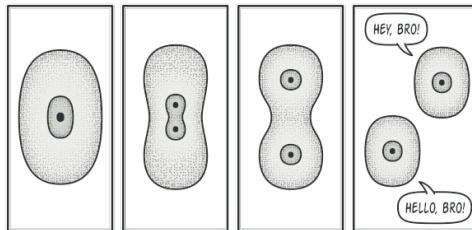
            proto = PrototypeFactory.getInstance(ModelType.ALBUM);
            System.out.println(albumPrototype);

            proto = PrototypeFactory.getInstance(ModelType.SHOW);
            System.out.println(proto);
        }
        catch (CloneNotSupportedException e)
            e.printStackTrace();
    }
}

Cloning Movie object..
Movie
Cloning Album object..
Album
Cloning Show object..
Show

```

2. Em biologia aprendemos no que consiste a mitose, o processo no qual uma célula se divide em duas, criando duas réplicas exatas da primeira. Esta, à luz deste padrão, seria um *prototype*.



5. Padrões estruturais

Capítulo com base nos slides teóricos e no site sourcemaking.com e refactoring.guru

Os padrões de desenho estruturais facilitam a construção de software através da identificação de formas simples de relacionar entidades.



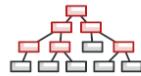
Adapter

Permite a colaboração de objetos de interfaces incompatíveis.



Bridge

Permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.



Composite

Permite que você componha objetos em estrutura de árvores e então trabalhe com essas estruturas como se fossem objetos individuais.



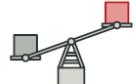
Decorator

Permite que você adicione novos comportamentos a objetos colocando eles dentro de um envoltório (wrapper) de objetos que contém os comportamentos.



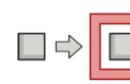
Facade

Fornecê uma interface simplificada para uma biblioteca, um framework, ou qualquer outro conjunto complexo de classes.



Flyweight

Permite que você coloque mais objetos na quantidade disponível de RAM ao compartilhar partes do estado entre múltiplos objetos ao invés de manter todos os dados em cada objeto.



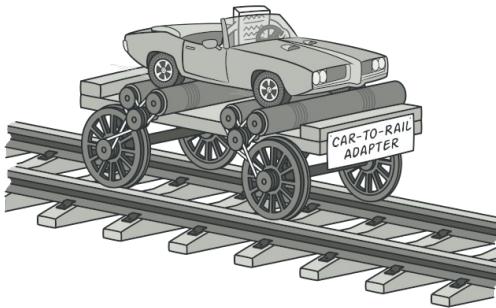
Proxy

Permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.

Class: Adapter

#Intenção

Converter a interface de uma classe, de encontro aos requisitos do cliente. **Permite assim a colaboração de objetos com interfaces incompatíveis**, fornecendo uma nova interface a uma classe já existente.



#Problema

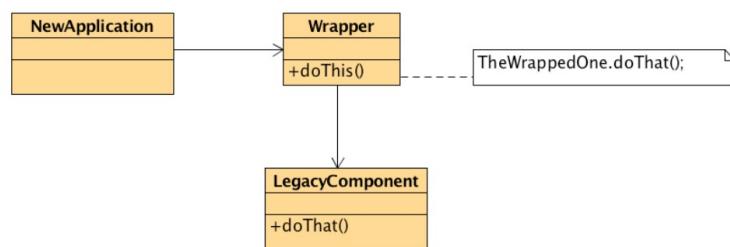
Quando surge a necessidade de **integrar um componente externo ao nosso ecossistema e este já existe**, podemos reutilizá-lo. No entanto, por vezes surgem problemas de compatibilidade.

#Solução

Nem sempre é desejável alterar código do componente que vamos reutilizar (trabalhosos e suscetível a erros e nem sempre temos a possibilidade de o manipular, pois pode ser uma ferramenta externa), sendo a solução a criação de um **adapter**, que é responsável por:

- Fornecer uma interface, compatível com um dos objetos;
- Este objeto acede aos métodos desta interface;
- A cada chamada, o *adapter* passa o pedido ao segundo objeto, mas com a formatação adequada (processo de adaptação).

O objeto “adaptado” não se apercebe da existência do *adapter*.



#Check List

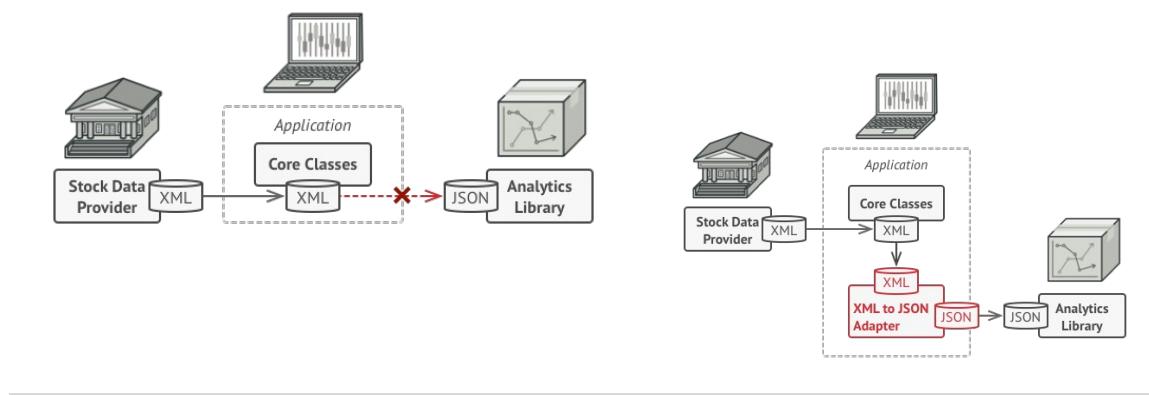
- ✓ Identificar os componentes a servir (clientes) e os que é necessário adaptar;
- ✓ Identificar a interface requerida pelo cliente;
- ✓ Desenhar uma classe *wrapper/adapter* que torne a classe a adaptar compatível com o cliente;
- ✓ O *adapter* deve ter (*has a*) instância da classe a adaptar;
- ✓ O *adapter* mapeia a interface do cliente para a interface da classe a adaptar;
- ✓ O cliente utiliza a interface do *adapter*.

#Exemplo

Uma analogia à vida real são as viagens internacionais, cenários em que muitas vezes nos deparamos com fichas diferentes das a que estamos habituados. A solução é utilizarmos um adaptador.

No mundo tecnológico podemos pensar numa aplicação de monitorização das ações do mercado, que recorre a uma fonte de dados que os fornece em XML. Em determinada altura surge a necessidade de integrar uma biblioteca de análise de dados, mas a que está disponível apenas faz o tratamento de dados JSON.

Para a conseguirmos integrar sem fazer modificações neste componente, de forma simples e sem o modificar, podemos construir um *adapter*, acedido pelo componente, que consulta os dados na fonte em XML e os converte para JSON, passando depois esta informação ao novo componente (o cliente).



Subclassing vs. Delegation

Por vezes a adaptação não se trata de alterar o funcionamento dos métodos, mas de **extender a classe a novos métodos, que fazem manipulação dos existentes.**

Nesta situação temos duas abordagens distintas:

Subclassing

Acesso automático a todos os métodos da superclasse;

Mais eficiente.

Delegation

Permite a remoção de métodos;

Wrappers podem ser adicionados e removidos de forma dinâmica;

Composição múltipla de objetos;

Mais flexível.

#Exemplo

```
interface Rectangle {
    void scale(int factor); //grow or shrink by factor
    void setWidth();
    float getWidth();
    float area(); ...
}

class Client {
    void clientMethod(Rectangle r) {
        // ...
        r.scale(2);
    }
}

class NonScalableRectangle {
    void setWidth(); ...
    // no scale method!
}
```

How to use this rectangle in Client?

❖ Class adapter adapts via subclassing

```
class ScalableRectangle1
    extends NonScalableRectangle
    implements Rectangle {

    void scale(int factor) {
        setWidth(factor*getWidth());
        setHeight(factor*getHeight());
    }
}
```

❖ Object adapter adapts via delegation:

– it forwards work to delegate

```
class ScalableRectangle2 implements Rectangle {
    NonScalableRectangle r; // delegate
    ScalableRectangle2(NonScalableRectangle r) {
        this.r = r;
    }

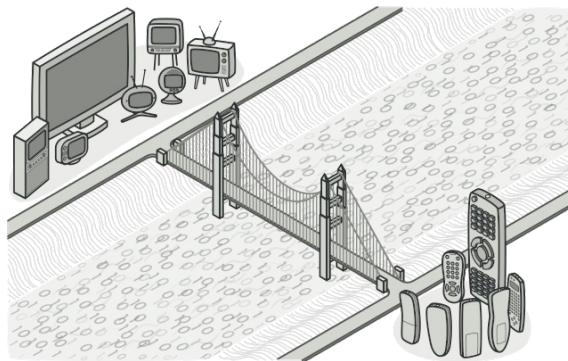
    void scale(int factor) {
        setWidth(factor * r.getWidth());
        setHeight(factor * r.getHeight());
    }

    float getWidth() { return r.getWidth(); }
    // ...
}
```

Object: Bridge

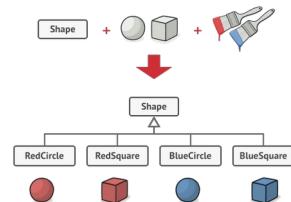
#Intenção

Permitir a divisão de uma classe ou um conjunto de classes relacionadas em duas hierarquias: abstração e implementação, para que ambas possam ser desenvolvidas de forma independente.



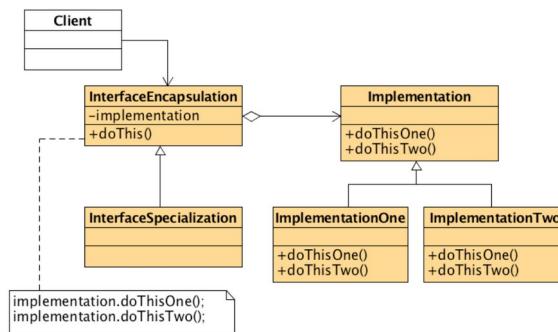
#Problema

Para criar implementações alternativas para uma classe uma hipótese é criar subclasses. No entanto, esta solução não é escalável, aumentando a sua complexidade exponencialmente com o número de variáveis envolvidas.



#Solução

O problema prende-se com a extensão das classes em duas dimensões. A proposta deste padrão é deixar a herança e apostar na **composição**, permitindo assim separar as dimensões em duas hierarquias de classes distintas.

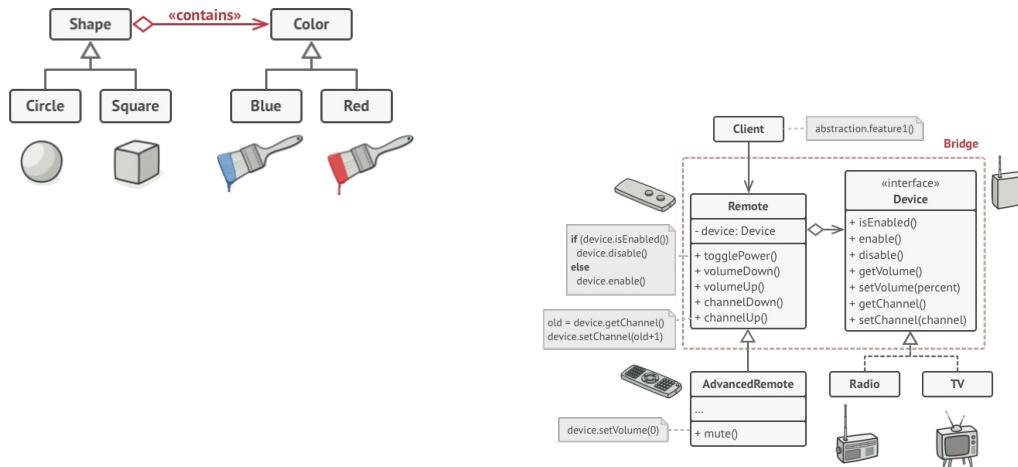


#Check List

Depois de verificar que existem duas dimensões variáveis no problema:

- ✓ Desenhar a separação das mesmas, tendo em conta o que a plataforma fornece e os requisitos do cliente;
- ✓ Desenhar uma interface mínima, necessária e suficiente;
- ✓ Definir uma classe derivada dessa interface para cada plataforma;
- ✓ Criar uma **classe abstrata base**, que tem uma instância da plataforma, a que delega as suas funcionalidades;
- ✓ Definir **especializações** da classe abstrata, se necessário.

#Exemplos



Object: Composite

#Intenção

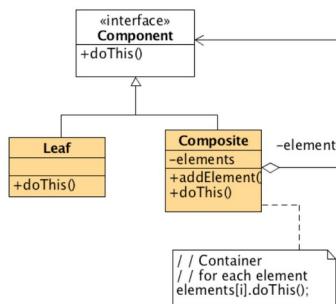
Ter componentes de produtos mas de forma a que um componente em si seja também um produto, ou seja, compor objetos em estruturas de árvore e utilizá-las como se fossem objetos (composição recursiva).

#Problema

Não é desejável que a aplicação tenha de distinguir um produto de uma composição de produtos.

#Solução

Criar uma interface comum ao produto e à composição de produtos, sendo que esta última contém um atributo que é uma coleção de elementos desta interface.



#Check list

Se o problema for a representação de relações hierárquicas parte/todo:

- ✓ Considerar a heurística “caixas que contém produtos podem ser um produto em si”;
- ✓ Criar uma interface que torne os produtos e as caixas intercambiáveis¹;
 - Tanto o produto como a caixa estabelecem a relação *is-a* com a interface;
 - A caixa estabelece a relação *has-a* com a interface *one-to-many*;
- ✓ Os métodos para adicionar e remover produtos da caixa devem ser declarados na classe da caixa (não na interface!).

¹ adj. | Que se pode intercambiar ou trocar.

#Exemplo

1. Os sistemas de ficheiros são aplicações práticas e com as quais lidamos frequentemente deste padrão. Uma pasta (caixa) pode conter vários ficheiros (produtos), mas também outras pastas (novamente caixas).

2.



Object: Decorator

#Intenção

Adicionar novas responsabilidades a um objeto de forma dinâmica, colocando-lo “dentro” de outros objetos que têm esse comportamento.

#Problema

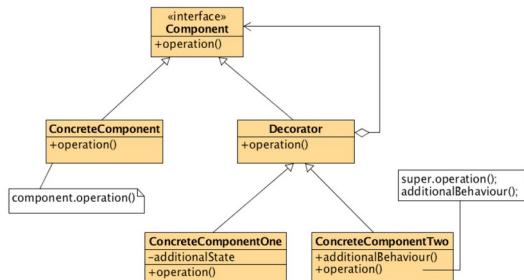
Para concretizar a intenção, uma opção é a herança. No entanto, se considerarmos várias funcionalidades que podem ser combinadas entre si (ou não), estaríamos a falar de demasiadas subclasses, pelo que esta não parece ser a solução adequada.

Para além disto, a **herança é estática**, não permitindo alterar o comportamento de um objeto em tempo de execução (a menos que substituído por um novo) e **não há herança múltipla**, pelo que cada classe pode derivar de uma e de uma só.

#Solução

A alternativa é a **agregação ou composição**, criando num objeto (**wrapper**) referência para outro, no qual delega alguma funcionalidade.

O *wrapper* implementa o mesmo conjunto de métodos do alvo, no qual delega os pedidos que recebe, podendo, no entanto, manipular os dados antes ou depois de os passar.



#Check list

Se estivermos perante um cenário com um componente central e vários opcionais associados:

- ✓ Criar uma interface que torne todas as classes intercambiáveis (estilo *composite*);
- ✓ Criar uma subclasse *decorator*, que suporte *wrappers* adicionais;
 - Este têm como atributo um elemento da interface;
 - Para cada componente opcional criar uma classe derivada;
- ✓ Tanto a classe do componente central como a *decorator* extendem a interface;

#Exemplo

1.

❖ Consider the following entities:

- Futebolista (joga, passa, remata),
- Tenista (joga, serve),
- Jogador (joga)

❖ Let's complicate:

- O Rui joga Basquete e Futebol
- A Ana joga Badminton e Basquete
- O Paulo joga Xadrez, Futebol e Basquete

❖ Solution?

```
interface JogadorInterface {
    void joga();
}

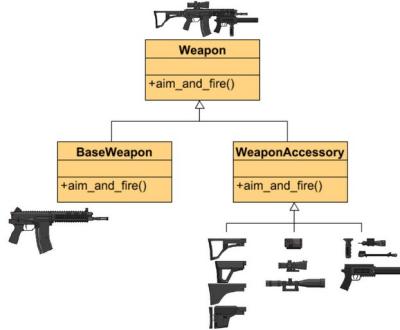
class Jogador implements JogadorInterface {
    private String name;
    Jogador(String n) { name = n; }
    @Override public void joga()
    { System.out.print("\n"+name+" joga "); }
}

abstract class JogDecorator implements JogadorInterface {
    protected JogadorInterface j;
    JogDecorator(JogadorInterface j) { this.j = j; }
    public void joga() { j.joga(); }
}

public class PlayTest{
    public static void main(String args[])
    {
        JogadorInterface j1 = new Jogador("Rui");
        Futebolista f1 = new Futebolista(new Jogador("Luis"));
        Xadrezista x1 = new Xadrezista(new Jogador("Ana"));
        Xadrezista x2 = new Xadrezista(j1);
        Xadrezista x3 = new Xadrezista(f1);
        Tenista t1 = new Tenista(j1);
        Tenista t2 = new Tenista(
            new Xadrezista(
                new Futebolista(
                    new Jogador("Bruno"))));
        JogadorInterface lista[] = { j1, f1, x1, x2, x3, t1, t2 };
        for (JogadorInterface ji: lista)
            ji.joga();
    }
}
```

Rui joga
Luis joga futebol
Ana joga xadrez
Rui joga xadrez
Luis joga futebol xadrez
Rui joga ténis
Bruno joga futebol xadrez ténis

2.



3. A classe `java.util.Scanner` aceita como argumento outro `scanner`, permitindo por exemplo ter um para ler linha a linha e para cada uma um novo para analisar o texto palavra a palavra.

4. Quando está a chover vestimos uma ou várias camisolas umas por cima das outras e ainda um casaco impermeável, de forma a que, cada uma com a sua funcionalidade, juntas consigam proteger-nos do frio e da chuva (algo que nenhuma sozinha consegue fazer por completo).

Object: Façade

#Intenção

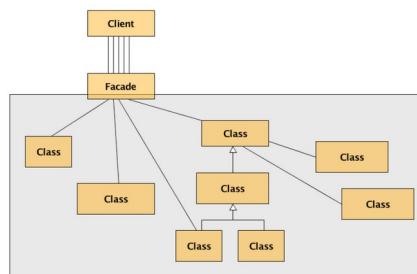
Criar uma interface comum (e mais simples) a um conjunto (complexo) de interfaces num subsistema, facilitando a utilização do sistema.

#Problema

Complexidade do sistema.

#Solução

Implementar interface que utiliza apenas os métodos que o cliente necessita.



#Check list

- ✓ Identificar uma interface mais simples e unificada para o sistema;
 - Só esta vai ser utilizada pelo cliente!
- ✓ Desenhar um *wrapper* que encapsule todos os subsistemas;
 - Responsável por delegar os métodos apropriados;
 - Considerar se dentro dos subsistemas acrescentaria valor a criação de outras classes segundo o padrão *façade*.

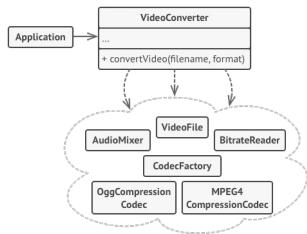
#Prós

Ao esconder a implementação do cliente podemos alterar os subsistemas sem este se aperceber, promovendo um baixo acoplamento e reduz as dependências.

Não se pode dizer no entanto que adicione alguma funcionalidade ou que previna os clientes de aceder a determinados métodos.

#Exemplo

1.



```
// Essas são algumas das classes de um framework complexo de um
// conversor de vídeo de terceiros. Nós não controlamos aquele
// código, portanto não podemos simplificá-lo.

class VideoFile
// ...

class OggCompressionCodec
// ...

class MPEG4CompressionCodec
// ...

class CodecFactory
// ...

class BitrateReader
// ...

class AudioMixer
// ...


// Nós criamos uma classe fachada para esconder a complexidade
// do framework atrás de uma interface simples. É uma troca
// entre funcionalidade e simplicidade.

class VideoConverter {
    method convert(filename, format) File is
        file = new VideoFile(filename)
        sourceCodec = new CodecFactory.extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)
}

// As classes da aplicação não dependem de um bocado de classes
// fornecidas por um framework complexo. Também, se você decidir
// trocar de frameworks, você só precisa reescrever a classe
// fachada.

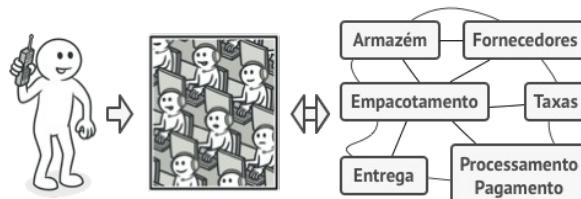
class Application {
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-cats-video.ogg", "mp4")
        mp4.save()
}
```

2.

```
// ...
class TravelFacade {
    private HotelBooker hotelBooker;
    private FlightBooker flightBooker;
    private LocalTourBooker tourBooker;
    public void getFlightsAndHotels(City dest, Date from, Date to) {
        List<Flight> flights = flightBooker.getFlightsFor(dest, from, to);
        List<Hotel> hotels = hotelBooker.getHotelsFor(dest, from, to);
        List<Tour> tours = tourBooker.getToursFor(dest, from, to);
        // process and return
    }
}

public class FacadeDemo {
    public static void main(String[] args) {
        TravelFacade facade = new TravelFacade();
        facade.getFlightsAndHotels(destination, from, to);
    }
}
```

3. Um cenário real que ilustra este padrão é um call-center de uma loja, através do qual podemos realizar várias operações através de um único canal: a chamada com o operador.



Object: Flyweight

#Intenção

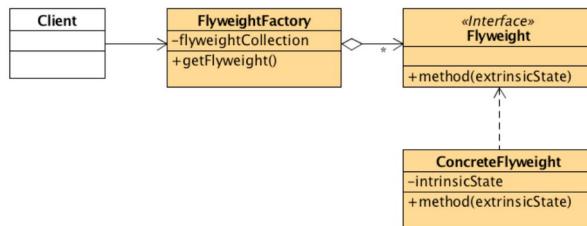
Utilizar a partilha de forma a suportar muitos objetos compactos de forma eficiente.

#Problema

O desenho de objetos com baixos níveis de granularidade aumenta a flexibilidade, mas pode ter custos elevados no que toca à performance e utilização de memória.

#Solução

Separar os atributos comuns a todos os objetos dessa classe (**estado intrínseco**) dos específicos e mutáveis em cada um (**estado extrínseco**) em duas classes, sendo a que contém o estado extrínseco derivada da que tem o intrínseco.



Podemos ainda associar-lhes o padrão object pool.

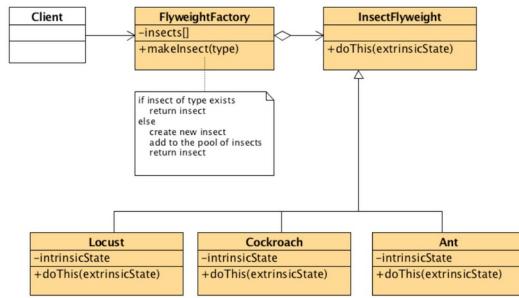
#Check list

Depois de verificar que uma grande quantidade de objetos de uma determinada classe estão a consumir demasiados recursos:

- ✓ Identificar os atributos pertencentes ao estado intrínseco e extrínseco;
 - Remover os extrínsecos e adicioná-los aos argumentos dos métodos que os utilizam;
- ✓ Criar uma fábrica segundo o padrão *object pool*;
 - O cliente deve utilizar a fábrica em vez do operador *new()*

#Exemplo

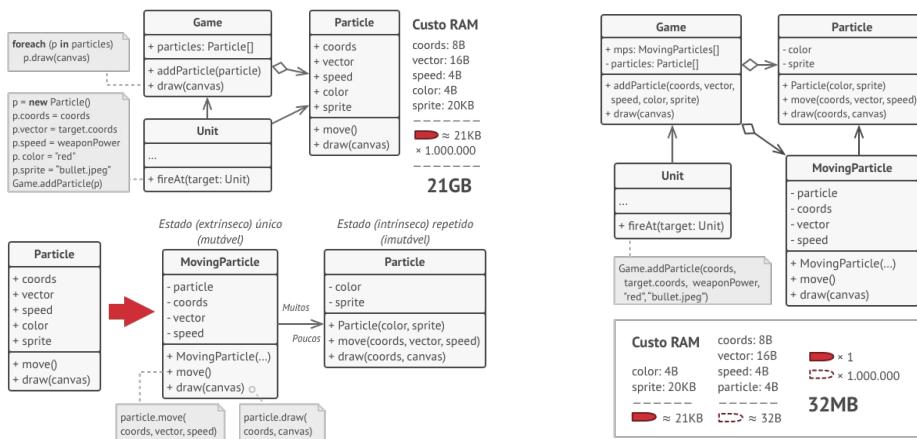
1 e 2.



```

public final class Integer extends Number implements Comparable<Integer> {
    // ...
    public static Integer valueOf(int i) {
        final int offset = 128;
        if (i >= -128 && i <= 127) { // must cache
            return IntegerCache.cache[i + offset];
        }
        return new Integer(i);
    }
    // ...
    private static class IntegerCache {
        static final Integer cache[] = new Integer[-(-128) + 127 + 1];
        static {
            for(int i = 0; i < cache.length; i++)
                cache[i] = new Integer(i - 128);
        }
    }
}
  
```

3.



4.

```

// A classe flyweight contém uma parte do estado de uma árvore
// Esses campos armazenam valores que são únicos para cada
// árvore em particular. Por exemplo, você não vai encontrar
// coordenadas da árvore aqui. Já que esses dados geralmente são
// GRANDES, você gastaria muita memória mantendo-os em cada
// objeto árvore. Ao invés disso, nós podemos extrair a textura,
// cor e outros dados repetitivos em um objeto separado os quais
// muitas árvores individuais podem referenciar.
class TreeType is
    field name
    field color
    field texture
constructor TreeType(name, color, texture) { ... }
method draw(canvas, x, y) is
    // 1. Cria um bitmap de certo tipo, cor e textura.
    // 2. Desenha o bitmap em uma tela nas coordenadas X e
    // Y.

// A fábrica flyweight decide se reutiliza o flyweight existente
// ou cria um novo objeto.
class TreeFactory is
    static field treeTypes: collection of tree types
    static method getTreeType(name, color, texture) is
        type = treeTypes.find(name, color, texture)
        if (type == null)
            type = new TreeType(name, color, texture)
            treeTypes.add(type)
        return type
  
```

```

// O objeto contextual contém a parte extrínseca do estado da
// árvore. Uma aplicação pode criar bilhões desses estados, já
// que são muito pequenos:
// apenas dois números inteiros para coordenadas e um campo de
// referência.
class Tree is
    field x, y
    field type: TreeType
constructor Tree(x, y, type) { ... }
method draw(canvas) is
    type.draw(canvas, this.x, this.y)

// As classes Tree (Árvore) e Forest (Floresta) são os clientes
// flyweight. Você pode unir-las se não planeja desenvolver mais
// a classe Tree.
class Forest is
    field trees: collection of Trees

method plantTree(x, y, name, color, texture) is
    type = TreeFactory.getTreeType(name, color, texture)
    tree = new Tree(x, y, type)
    trees.add(tree)

method draw(canvas) is
    foreach (tree in trees) do
        tree.draw(canvas)
  
```

Object: Proxy

#Intenção

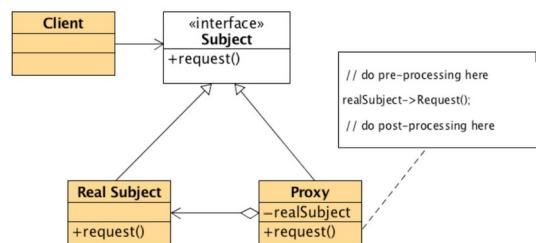
Controlar o acesso ao objeto original, permitindo que se faça algo antes ou depois de chegar ao mesmo.

#Problema

Lidar com objetos exigentes ao nível de recursos e sem a possibilidade de alterar o seu código (implementado o método *object pool*, p.e.) leva a que queiramos instanciá-los apenas quando e até serem necessários.

#Solução

Criar uma classe **proxy** com a mesma interface do objeto original, que por sua vez passa tudo ao original, permitindo agora fazer algo antes ou depois da inicialização do mesmo.



Sendo invisível para o cliente, não vai afetar o seu código, ou seja, onde era esperado um objeto real pode ser colocado um proxy.

#Check list

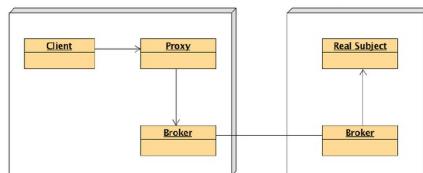
- ✓ Identificar a funcionalidade que a implementar no *wrapper*.
- ✓ Definir a interface que torne o *proxy* e o objeto real intercambiáveis;
- ✓ Considerar definir uma fábrica que decida qual dos dois é desejável;
- ✓ O *proxy* deve ter como atributo o objeto real.

#Exemplo

1, 2 e 3.

Known Uses: Distributed Objects

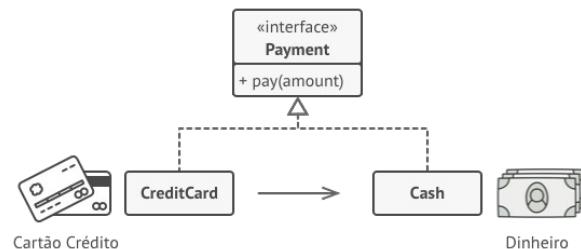
- ❖ The Client and Real Subject are in different processes or on different machines, and so a direct method call will not work
- ❖ The Proxy's job is to pass the method call across process or machine boundaries, and return the result to the client (with Broker's help)



Known Uses: Lazy Loading

- ❖ Some objects are expensive to instantiate (i.e., consume lots of resources or take a long time to initialize)
- ❖ Create a proxy instead, and give the proxy to the client
 - The proxy creates the object on demand when the client first uses it
 - Proxies must store whatever information is needed to create the object on-the-fly (file name, network address, etc.)

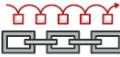
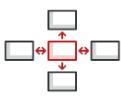
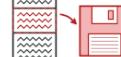
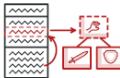
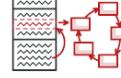
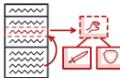
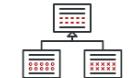
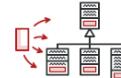
4. Um cartão de crédito é um proxy para uma conta bancária, que é um proxy para uma porção de dinheiro. Ambos implementam a mesma interface porque não há necessidade de carregar uma porção de dinheiro por aí. Um cliente se sente bem porque não precisa ficar carregando montanhas de dinheiro por aí. Um dono de loja também fica feliz uma vez que a renda da transação é adicionada eletronicamente para sua conta sem o risco de perdê-la no depósito ou de ser roubado quando estiver indo ao banco.



6. Padrões comportamentais

Capítulo com base nos slides teóricos e no site refactoring.guru

Os padrões de desenho comportamentais identificam **padrões de comunicação entre objetos**.

 <p>Chain of Responsibility</p> <p>Permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou passa para o próximo handler da corrente.</p>	 <p>Command</p> <p>Transforma o pedido em um objeto independente que contém toda a informação sobre o pedido. Essa Transformação permite que você parametrize métodos com diferentes pedidos, atraso ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas.</p>	 <p>Interpreter</p> <p>Uma forma de incluir elementos de linguagem num programa.</p>
 <p>Iterator</p> <p>Permite que você percorra elementos de uma coleção sem expor as representações estruturais deles (lista, pilha, árvore, etc.)</p>	 <p>Mediator</p> <p>Permite que você reduza as dependências caóticas entre objetos. O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.</p>	 <p>Memento</p> <p>Permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.</p>
 <p>Null Object</p> <p>Fornecer representação do objeto nulo.</p>	 <p>Observer</p> <p>Permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.</p>	 <p>State</p> <p>Permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.</p>
 <p>Strategy</p> <p>Permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.</p>	 <p>Template Method</p> <p>Define o esqueleto de um algoritmo na superclasse mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.</p>	 <p>Visitor</p> <p>Permite que você separe algoritmos dos objetos nos quais eles operam.</p>

Chain of responsibility

#Intenção

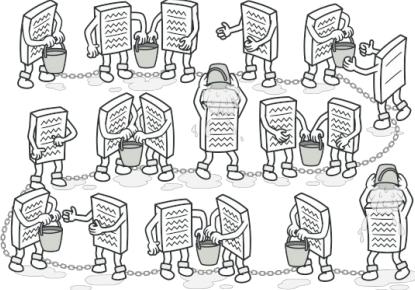
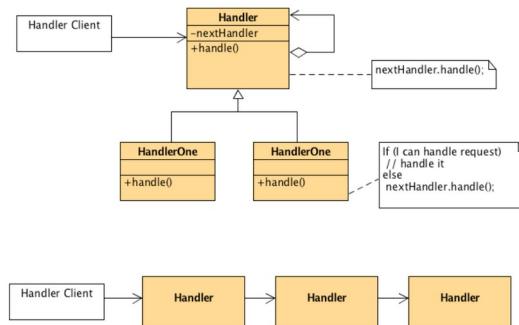
Criar uma **corrente de objetos que passam os pedidos** através da mesma até que um o processe, providenciando um único caminho para vários processamentos possíveis.

#Problema

É necessário processar os pedidos de forma eficiente, sem mapeamento ou relações de dependência.

#Solução

Criar uma referência em cada *handler* para o seguinte, tendo cada um o poder de passar ou não o pedido ao seguinte.



#Check list

- ✓ A classe base tem um ponteiro para o *next*;
 - ✓ Cada classe derivada dá o seu contributo para o processamento do pedido;
 - ✓ Caso o pedido deva ser passado ao *handler* seguinte, a classe derivada invoca a classe base para o fazer;
 - ✓ O cliente cria e liga a corrente;
 - ✓ O cliente “entrega” cada pedido na raiz da corrente;

#Exemplo

1.

```
abstract class Parser {
    private Parser successor = null;

    public void parse(String fileName) {
        if (successor != null)
            successor.parse(fileName);
        else
            System.out.println("No parser for the file: " + fileName);
    }

    protected boolean canHandleFile(String fileName, String format) {
        return (fileName == null) || (fileName.endsWith(format));
    }

    public Parser setSuccessor(Parser successor) {
        this.successor = successor;
        return this;
    }
}

class TextParser extends Parser {

    @Override
    public void parse(String fileName) {
        if (canHandleFile(fileName, ".txt")) {
            System.out.println("A text parser for: " + fileName);
        } else {
            super.parse(fileName);
        }
    }
}
```

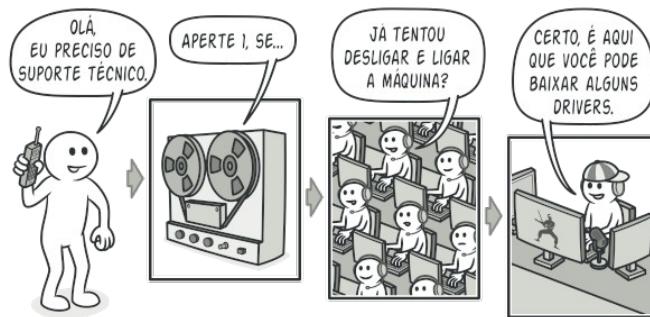
```
class JsonParser extends Parser {
    @Override
    public void parse(String fileName) {
        if (canHandleFile(fileName, ".json"))
            System.out.println("A JSON parser for: " + fileName);
        else
            super.parse(fileName);
    }
}

class CsvParser extends Parser {
    @Override
    public void parse(String fileName) {
        if (canHandleFile(fileName, ".csv"))
            System.out.println("A CSV parser for: " + fileName);
        else
            super.parse(fileName);
    }
}

public class ChainOfResponsibilityDemo {
    public static void main(String[] args) {
        List<String> fileList = new ArrayList<String>();
        fileList.add("somefile.txt");
        fileList.add("otherfile.json");
        fileList.add("csvfile.csv");
        fileList.add("somethingelse.doc");
        Parser textParser =
            new CsvParser().setSuccessor(
                new TextParser().setSuccessor(
                    new JsonParser()));
        for (String fileName : fileList) {
            textParser.parse(fileName);
        }
    }
}
```

A text parser for: somefile.txt
A JSON parser for: otherfile.json
A CSV parser for: csvfile.csv
No parser for the file: somethingelse.doc

2. Numa analogia ao mundo real temos uma chamada para o suporte técnico pode ser atendida por diversos operadores, dependendo da sua competência para responder ao nosso pedido, encaminhando a chamada para outro caso não nos consigam ajudar.



Command

#Intenção

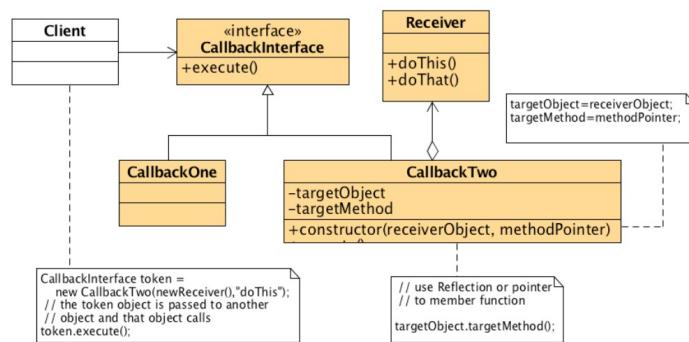
Encapsular um pedido num objeto, permitindo parameterizar métodos com diferentes pedidos, atrasar ou colocar a execução de um pedido numa fila e suportar operações que não podem ser feitas.

#Problema

Necessitamos de fazer pedidos a objetos abstraíndo-nos sobre a operação a realizar e qual o recetor do pedido.

#Solução

Dividir a aplicação em camadas, distinguindo a interface invocadora da implementação do processamento da invocação.



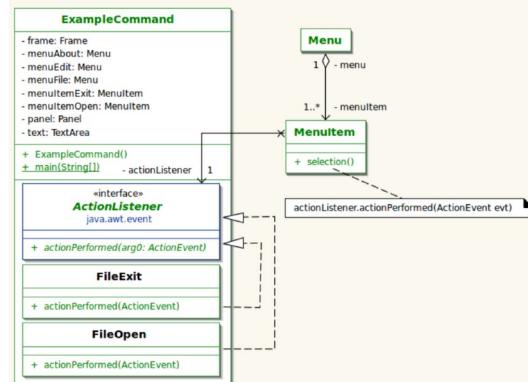
#Check list

Se os pedidos tiverem de ser processados em vários momentos e/ou em diversas ordens.

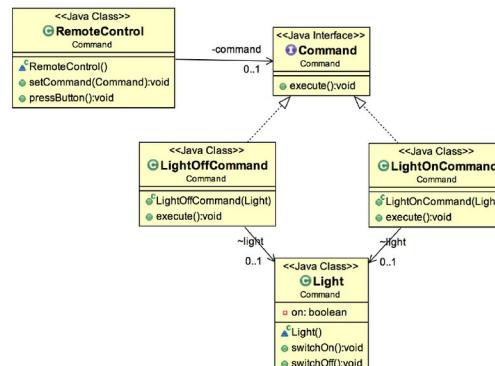
#Exemplo

1. Menu de uma interface gráfica

```
menuFile = new Menu("File", true);
menuItemOpen = new MenuItem("Open...");
menuItemExit = new MenuItem("Exit");
menuFile.add(menuItemOpen);
menuFile.add(menuItemExit);
menuItemOpen.addActionListener(new FileOpen());
menuItemExit.addActionListener(new FileExit());
...
class FileOpen implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileDialog fdlg = new FileDialog(frame, "Open", FileDialog.LOAD);
        fdlg.show();
    }
}
class FileExit implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        System.exit(0);
    }
}
```



2. Comando de televisão



```
//Command
interface Command {
    public void execute();
}

// Concrete Command
class LightOnCommand implements Command {
    // reference to the light
    Light light;
    public LightOnCommand(Light light) { this.light = light; }
    public void execute() { light.switchOn(); }
}

// Concrete Command
class LightOffCommand implements Command {
    // reference to the light
    Light light;
    public LightOffCommand(Light light) { this.light = light; }
    public void execute() { light.switchOff(); }
}
```

```
// Receiver
class Light {
    private boolean on;
    public void switchOn() { on = true; }
    public void switchOff() { on = false; }
}

// Invoker
class RemoteControl {
    private Command command;
    public void setCommand(Command command) {
        this.command = command;
    }
    public void pressButton() {
        command.execute();
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        RemoteControl control = new RemoteControl();

        Light light = new Light();

        Command lightsOn = new LightOnCommand(light);
        Command lightsOff = new LightOffCommand(light);

        //switch on
        control.setCommand(lightsOn);
        control.pressButton();

        //switch off
        control.setCommand(lightsOff);
        control.pressButton();
    }
}
```

3. Reflection

```
public class CommandReflect {
    private int state;
    public CommandReflect( int in ) { state = in; }
    public int addOne( Integer one ) { return state + one; }
    public int addTwo( Integer one, Integer two ) { return state + one + two; }
    static public class Command {
        private Object receiver; // the "encapsulated" object
        private Method action; // the "pre-registered" request
        private Object[] args; // the "pre-registered" arg list
        public Command( Object obj, String methodName, Object[] arguments ) {
            receiver = obj;
            args = arguments;
            Class<?> cls = obj.getClass();
            Class<?>[] argTypes = new Class[args.length];
            for( int i=0; i < args.length; i++ ) // get the "Class" for each
                argTypes[i] = args[i].getClass(); // supplied argument
            try {
                action = cls.getMethod( methodName, argTypes );
            } catch( NoSuchMethodException e ) { System.out.println( e ); }
        }
        public Object execute() {
            try { return action.invoke( receiver, args ); }
            catch( IllegalAccessException e ) { System.out.println( e ); }
            catch( InvocationTargetException e ) { System.out.println( e ); }
            return null;
        }
    }
}
```

```
public static void main( String[] args ) {
    CommandReflect[] objs = { new CommandReflect(1), new CommandReflect(2) };
    Command[] cmds = {
        new Command( objs[0], "addOne", new Integer[] { 3 } ),
        new Command( objs[1], "addTwo", new Integer[] { 4, 5 } );
    };
    System.out.print( "\nReflection results: " );
    for ( Command cmd: cmds )
        System.out.print( cmd.execute() + " " );
    System.out.println();
}
```

Reflection results: 4 11

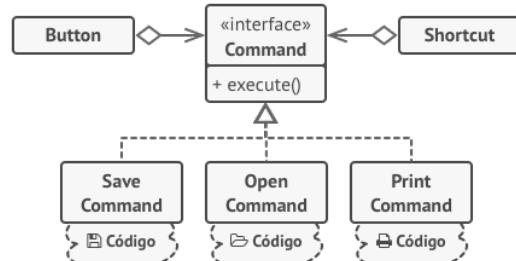
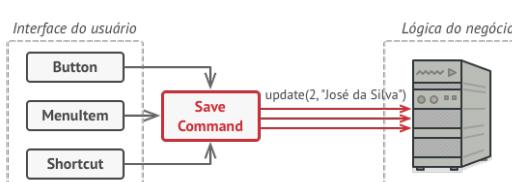
4. Interface funcional

```
public class CommandFactory {
    private Map<String, Command> commands;
    private CommandFactory() {
        commands = new HashMap<String, Command>();
    }
    public void addCommand( String name, Command command ) {
        commands.put( name, command );
    }
    public void executeCommand( String name ) {
        if ( commands.containsKey( name ) )
            commands.get( name ).execute();
    }
    public String listCommands() {
        return "Enabled commands: " +
            commands.keySet().stream().collect( Collectors.joining( ", " ) );
    }
}
public static CommandFactory init() { /* Factory pattern */
    final CommandFactory cf = new CommandFactory();
    // commands are added here using lambdas.
    // It is also possible to dynamically add commands without editing the code.
    cf.addCommand( "Light on", () -> System.out.println( "Light turned on" ) );
    cf.addCommand( "Light off", () -> System.out.println( "Light turned off" ) );
    cf.addCommand( "Sound on", () -> System.out.println( "Sound turned on" ) );
    cf.addCommand( "Sound off", () -> System.out.println( "Sound turned off" ) );
    return cf;
}
```

```
public class Main {
    public static void main( final String[] arguments ) {
        CommandFactory cf = CommandFactory.init();
        System.out.println( cf.listCommands() );
        cf.executeCommand( "Light on" );
        cf.executeCommand( "Sound on" );
        cf.executeCommand( "Light off" );
    }
}
```

Enabled commands: Light on, Sound on, Sound off
 Light turned on
 Sound turned on
 Light turned off

5. Botões



6. Numa analogia ao mundo real, quando vamos a um restaurante fazemos o pedido ao empregado de mesa, que o anota num papel e o entrega no balcão da cozinha, onde este espera que o cozinheiro o leia e comece a processar. O pedido funciona como um comando.

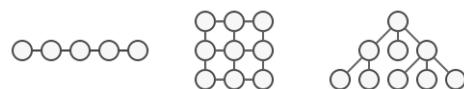
Iterator

#Intenção

Percorrer objetos de uma coleção sem expor a sua representação (lista, árvore...).

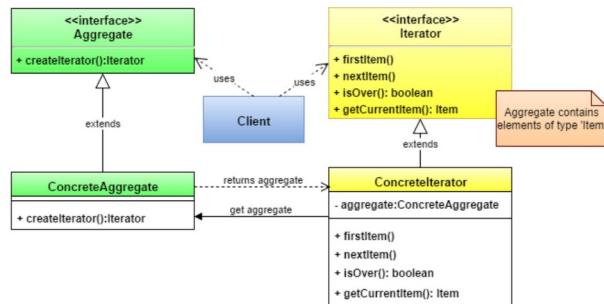
#Problema

Em coleções mais complexas, podem haver várias formas de as percorrer. No entanto, implementá-las todas retira o foco da classe no armazenamento dos objetos (a sua principal utilidade).



#Solução

Extrair o comportamento de travessia da coleção para outro objeto: um **iterator**.



#Check list

- ✓ Adicionar um método *iterator()* na classe da coleção, que devolva a classe do *iterator*;
- ✓ Desenhar a *innerclass* do *iterator* na classe agregadora;
- ✓ O cliente pede à classe agregadora para criar o *iterator*;
- ✓ O cliente utiliza os métodos *hasNext()* e *next()* para aceder aos elementos da coleção.

#Exemplo

1.

```
public class VectorGeneric<T> {
    private T[] vec;
    private int nElem;
    private final static int ALLOC = 50;
    private int dimVec = ALLOC;

    @SuppressWarnings("unchecked")
    public VectorGeneric() {
        vec = (T[]) new Object[dimVec];
        nElem = 0;
    }

    public boolean addElem(T elem) {
        if (elem == null)
            return false;
        ensureSpace();
        vec[nElem++] = elem;
        return true;
    }
}
```

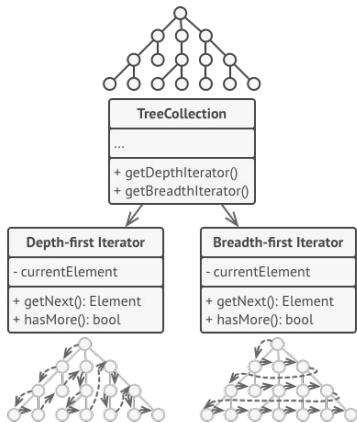
```
private void ensureSpace() {
    if (nElem>=dimVec) {
        dimVec += ALLOC;
        @SuppressWarnings("unchecked")
        T[] newArray = (T[]) new Object[dimVec];
        System.arraycopy(vec, 0, newArray, 0, nElem );
        vec = newArray;
    }
}

public boolean removeElem(T elem) {
    for (int i = 0; i < nElem; i++) {
        if (vec[i].equals(elem)) {
            if (nElem-i-1 > 0) // not last element
                System.arraycopy(vec, i+1, vec, i, nElem-i-1 );
            vec[--nElem] = null; // libertar ultimo objecto para o GC
            return true;
        }
    }
    return false;
}
```

```
public Iterator<T> iterator() {
    return (this).new VectorIterator<T>();
}

private class VectorIterator<K> implements Iterator<K> {
    private int indice;
    VectorIterator() {
        indice = 0;
    }
    public boolean hasNext() {
        return (indice < nElem);
    }
    public K next() {
        if (hasNext())
            return (K)VectorGeneric.this.vec[indice++];
        throw new NoSuchElementException("only " + nElem + " elements");
    }
    public void remove() { // default since Java 8
        throw new UnsupportedOperationException("Operacao nao suportada!");
    }
}
```

2. Vários tipos de *iterators*



3. Formas de iterar as coleções mais comuns em JAVA

```
// For a set or list
for (Iterator it=collection.iterator(); it.hasNext(); ) {
    Object element = it.next();
}

// For keys of a map
for (Iterator it=map.keySet().iterator(); it.hasNext(); ) {
    Object key = it.next();
}

// For values of a map
for (Iterator it=map.values().iterator(); it.hasNext(); ) {
    Object value = it.next();
}

// For both the keys and values of a map
for (Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
}
```

4. Numa analogia ao mundo real, quando visitamos uma nova cidade podemos recorrer a uma aplicação para nos guiarmos, mas também comprar um guia turístico ou até contratar um guia.

Mediator

#Intenção

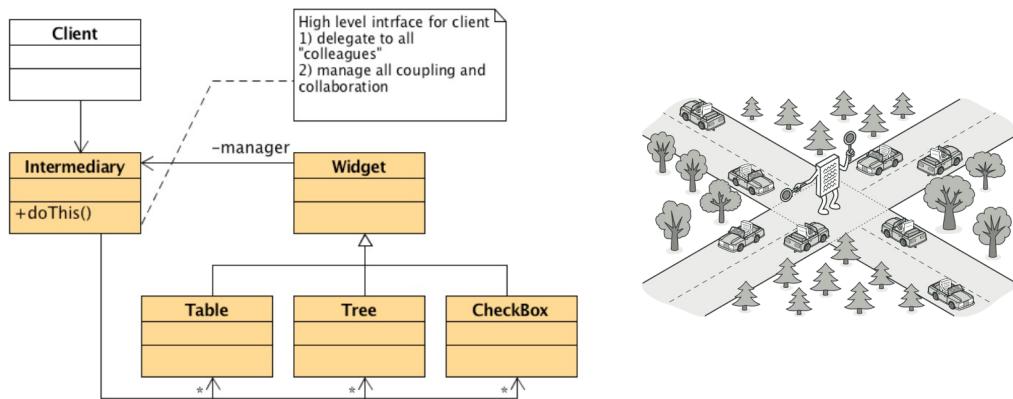
Reducir a dependência entre objetos, restringindo a comunicação direta, forçando que esta seja feita através de um mediador.

#Problema

As dependências entre os objetos (pouca coesão) tornam-nos difíceis de reutilizar.

#Solução

Colaboração indireta, através de um **mediador.**



O mediador define a interface que o *Widget* utiliza para comunicar, que por sua vez define uma classe abstrata com referência ao mediador.

Do mediador pode descender um *ConcreteMediator*, que encapsula a lógica entre os *Widgets*. Todos os descendentes do *Widget* comunicam exclusivamente através do *Intermediary*.

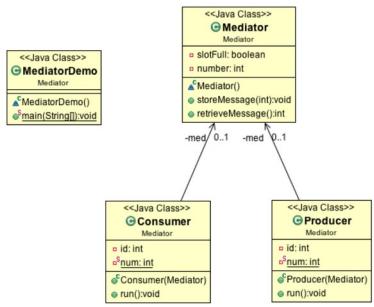
#Check list

Se tivermos um conjunto de objetos que interagem entre si e beneficiariam do desacoplamento.

- ✓ Encapsular as interações numa nova classe;
- ✓ Criar uma instância desta nova classe em todas as que interagiam;
- ✓ Equilibrar o desacoplamento com o princípio da distribuição equitativa de responsabilidades;
- ✓ Ter atenção para não criar um controlador ou objeto “deus”.

#Exemplos

1. Producer/consumer



```
<<Java Class>>
@MediatorDemo
Mediator
```

```
<<Java Class>>
@Consumer
Mediator
```

```
<<Java Class>>
@Producer
Mediator
```

```
class Mediator {
    private boolean slotFull = false;
    private int number;

    public synchronized void storeMessage(int num) {
        while (slotFull == true) {
            try { wait(); }
            catch (InterruptedException e) {
                // ...
            }
        }
        slotFull = true;
        number = num;
        notifyAll();
    }

    public synchronized int retrieveMessage() {
        // ...
    }
}
```

```
class Producer extends Thread {
    // 2. Producers are coupled only to the Mediator
    private Mediator med;
    private int id;
    private static int num = 1;

    public Producer(Mediator m) {
        med = m;
        id = num++;
    }

    public void run() {
        int num;
        while (true) {
            med.storeMessage(num = (int) (Math.random() * 100));
            System.out.print("p" + id + "-" + num + " ");
        }
    }
}
```

```
class Consumer extends Thread {
    // 3. Consumers are coupled only to the Mediator
    private Mediator med;
    private int id;
    private static int num = 1;

    public Consumer(Mediator m) {
        med = m;
        id = num++;
    }

    public void run() {
        while (true) {
            System.out.print("c" + id + "-" + med.retrieveMessage() + " ");
        }
    }
}
```

```
class MediatorDemo {
    public static void main(String[] args) {
        Mediator mb = new Mediator();
        new Producer(mb).start();
        new Producer(mb).start();
        new Consumer(mb).start();
        new Consumer(mb).start();
        new Consumer(mb).start();
        new Consumer(mb).start();
    }
}
```

p1-68 p1-42 c2-73 c4-65 p2-73 p2-49 c1-68 c4-49 p2-95 p1-65 c1-76 c3-42 c1-3 p1-3 p1-31 p2-76

2. Numa analogia ao mundo real temos as torres de controlo nos aeroportos, que coordenam a aterragem e descolagem de aviões sem que estes comuniquem entre si.

Memento

#Intenção

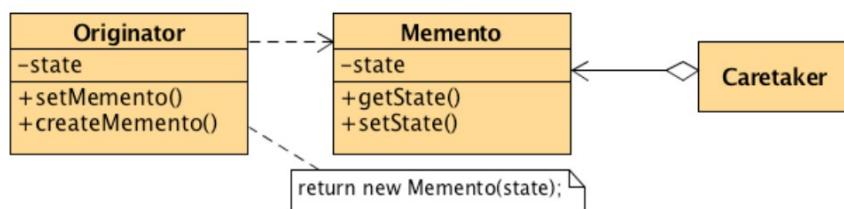
Salvar e restaurar o estado anterior de um objeto, promovendo operações de *undo*.

#Problema

Acesso a todos os atributos nem sempre é possível e por vezes podemos fazer *refactoring* às classes.

#Solução

Delegar a criação do retrato do estado ao dono do estado.



São assim definidos três papéis:

Originator O objeto que detém o estado

Caretaker O objeto que sabe quando e porque é que o *Originator* precisa de guardar o seu estado

Memento O cofre, lido e escrito pelo *originator* e gerida pelo *Caretaker*

#Check list

- ✓ Identificar os papéis do *Caretaker* e do *Originator*;
- ✓ Criar uma classe *Memento*;
- ✓ O *Caretaker* sabe quando guardar o estado do *Originator*;
- ✓ O *Originator* cria um *Memento* e copia o seu estado para este;
- ✓ O *Caretaker* recebe o *Memento* e armazena-o;
- ✓ O *Caretaker* disponibiliza método para obter último estado;
- ✓ O *Originator* retoma o estado anterior com recurso ao *Memento*.

#Exemplo

1.

```
class Memento {
    private String state;
    public Memento(String stateToSave) { state = stateToSave; }
    public String getSavedState() { return state; }
}
```

```
class Originator {
    private String state; // simple example
    public void set(String state) {
        this.state = state;
    }
    public Memento saveToMemento() {
        return new Memento(state);
    }
    public void restoreFromMemento(Memento m) {
        state = m.getSavedState();
    }
    @Override public String toString() { return state; }
}
```

```
public class MementoDemo {
    public static void main(String[] args) {
        Caretaker caretaker = new Caretaker();

        Originator originator = new Originator();
        for (int i= 1; i<=5; i++) {
            originator.set("State " + i);
            System.out.println("Originator: state set to " + originator);
            caretaker.addMemento( originator.saveToMemento() );
            System.out.println("Memento saved");
        }

        while (caretaker.hasMemento()) {
            originator.restoreFromMemento( caretaker.getMemento() );
            System.out.println("Originator: after restore: " + originator)
        }
    }
}
```

```
class Caretaker {
    private Stack<Memento> savedStates = new Stack<Memento>();

    public void addMemento(Memento m) {
        savedStates.push(m);
    }
    public boolean hasMemento() {
        return !savedStates.isEmpty();
    }
    public Memento getMemento() {
        return savedStates.pop();
    }
}
```

```
Originator: state set to State 1
Memento saved
Originator: state set to State 2
Memento saved
Originator: state set to State 3
Memento saved
Originator: state set to State 4
Memento saved
Originator: state set to State 5
Memento saved
Originator: after restore: State 5
Originator: after restore: State 4
Originator: after restore: State 3
Originator: after restore: State 2
Originator: after restore: State 1
```

Null object

#Intenção

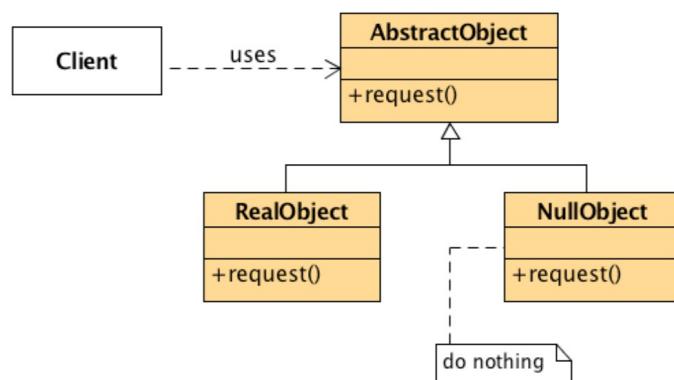
Encapsular a ausência de um objeto, fornecendo uma alternativa adequada ao comportamento de **não fazer nada**.

#Problema

Tratar a ausência de um objeto de forma transparente, **abstraindo** o cliente.

#Solução

Devolver um objeto que represente o comportamento de não fazer nada na ausência de outro.



#Checklist

- ✓ Criar uma classe abstrata representativa dos objetos;
- ✓ Criar as classes derivadas que implementam funcionalidades para os métodos da anterior e uma que não faça nada na invocação dos mesmos métodos.

#Exemplo

```

private Request getRequest(String command) {
    if (command.equals("A"))
        return new ARequest();
    if (command.equals("B"))
        return new BRequest();
    if (command.equals("C"))
        return new CRequest();
    return null;
}
  
```



```

private Request getRequest(String command) {
    if (command.equals("A"))
        return new ARequest();
    if (command.equals("B"))
        return new BRequest();
    if (command.equals("C"))
        return new CRequest();
    return new NullRequest();
}
  
```

Observer

Também conhecido por **Event-subscriber** ou **Listener**

#Intenção

Definir uma relação **one-to-many** entre dois objetos, de forma a que quando o **one** muda o seu estado, todos os seus dependentes são notificados automaticamente.

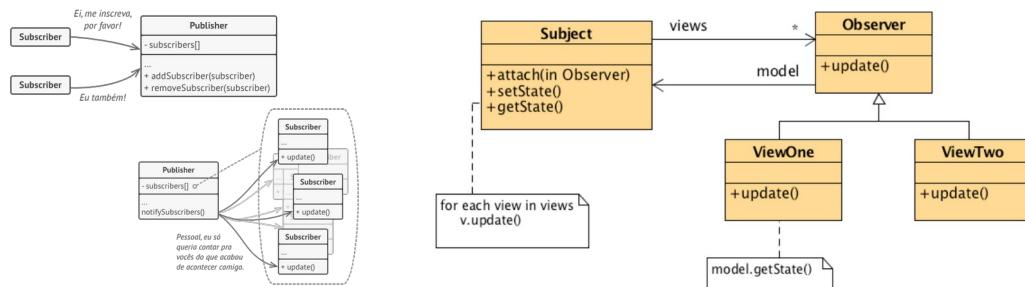
#Problema

Um desenho monolítico não é escalável quando novos gráficos ou requisitos de monitorização são atualizados.

Os clientes não querem estar constantemente a verificar se o estado mudou (quere ser notificados), mas podem crer ser notificados sobre um determinado estado específico (e não todos).

#Solução

Criar um mecanismo de **subscrição** no **publicador** (o **one** da relação), que oferece um **conjunto de métodos da gestão das mesmas**, invocados pelos **subscritores**, que por sua vez devem implementar uma interface, que declara o método de notificação.



#Check list

- ✓ Distinguir as funcionalidades principais e independentes das opcionais e dependentes;
- ✓ Modelas as independentes nos **sujeitos** e as dependentes nos **observadores**;
- ✓ O **sujeito** tem uma referência (lista) à classe base dos **observadores**;
- ✓ Os **observadores** registam-se nos **sujeitos**;
- ✓ Os **sujeitos** notificam os **observadores** de alterações.

Os sujeitos podem enviar a informação aos observadores ou apenas notificá-los e estes fazerem um pedido posterior pela informação.

#Exemplo

1.

```
class Subject {
    private List<Observer> observers = new ArrayList<>();
    private int state;
    public void attach(Observer o) {
        observers.add(o);
    }
    public int getState() {
        return state;
    }
    public void setState(int in) {
        state = in;
        notifyObservers();
    }
    private void notifyObservers() {
        for (Observer obs : observers) {
            obs.update();
        }
    }
}
```

```
abstract class Observer {
    protected Subject subj;

    public abstract void update();
}

class HexObserver extends Observer {
    public HexObserver(Subject s) {
        subj = s;
        subj.attach(this); // Observers register themselves
    }

    public void update() {
        System.out.println("HexObserver saw "
            + Integer.toHexString(subj.getState()));
    }
}
```

```
class OctObserver extends Observer {
    public OctObserver(Subject s) {
        subj = s;
        subj.attach(this);
    }
    public void update() {
        System.out.println("OctObserver saw "
            + Integer.toOctalString(subj.getState()));
    }
}

class BinObserver extends Observer {
    public BinObserver(Subject s) {
        subj = s;
        subj.attach(this);
    }
    public void update() {
        System.out.println("BinObserver saw "
            + Integer.toBinaryString(subj.getState()));
    }
}
```

```
public class ObserverDemo {
    public static void main(String[] args) {
        Subject sub = new Subject();
        // Client configures the number and type of Observers
        new HexObserver(sub);
        new OctObserver(sub);
        new BinObserver(sub);
        Scanner scan = new Scanner(System.in);
        while (true) {
            System.out.print("\nEnter a number: ");
            sub.setState(scan.nextInt());
        }
    }
}
```

```
Enter a number: 25
HexObserver saw 19
OctObserver saw 31
BinObserver saw 11001

Enter a number: 77
HexObserver saw 4d
OctObserver saw 115
BinObserver saw 1001101
```

2. Numa analogia ao mundo real, quando assinamos uma revista a publicadora mantém uma lista dos subscriptores, para os quais envia uma revista cada vez que sai um novo número. Neste caso somos notificados com o envio do conteúdo.

3. Outro exemplo são as notificações por SMS de que a nossa entrega está disponível para levantamento no posto de correios. Neste caso, fomos apenas notificados de que houve uma alteração no estado da nossa encomenda, devendo nós ir buscá-la (fazer um *pull*).

State

#Intenção

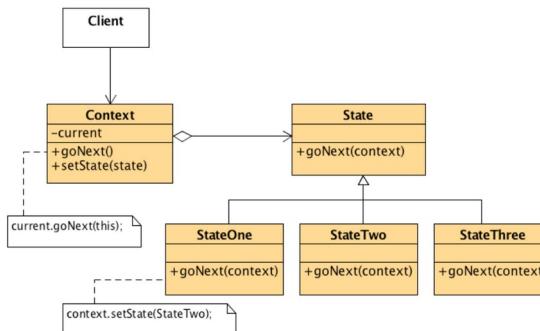
Permitir que um **objeto altere o seu comportamento com base nas alterações do seu estado interno**.

#Problema

Sendo o seu próximo estado dependendo do estado atual, há medida que adicionamos estados possíveis ao nosso objeto vamos ter instruções condicionais gigantes e difíceis de manter a longo prazo com eventuais alterações.

#Solução

Criar novas classes para cada estado e extrair o comportamento específico de cada estado (o seu contexto) para dentro dessas classes.



#Check list

- ✓ Identificar ou criar a classe **state machine** que será o *wrapper* dos estados: **Context**;
- ✓ Criar a classe base que replica os métodos da interface definida anteriormente, cada um com uma instância da classe *wrapper* como argumento extra;
- ✓ Criar uma classe derivada para cada estado;
- ✓ A classe *wrapper* mantém o estado atual do objeto;
- ✓ Todos os pedidos do cliente são delegados no estado atual, sendo passado ao *wrapper* o estado que resultou da execução do método invocado;

É assim responsabilidade dos métodos das classe **State** fazer a alteração do estado.

#Exemplo

1. Ventoinha

```
class CeilingFanPullChain {
    private State currentState;

    public CeilingFanPullChain() {
        currentState = new Off();
    }

    public void setState(State s) {
        currentState = s;
    }

    public void pull() {
        currentState.pull(this);
    }

    interface State {
        void pull(CeilingFanPullChain wrapper);
    }
}
```

```
class Off implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Low()); System.out.println(" low speed");
    }
}

class Low implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Medium()); System.out.println(" medium speed");
    }
}

class Medium implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new High()); System.out.println(" high speed");
    }
}

class High implements State {
    public void pull(CeilingFanPullChain wrapper) {
        wrapper.setState(new Off()); System.out.println(" turning off");
    }
}
```

2. Numa analogia ao mundo real, o **botão de desbloqueio** dos nossos telemóveis tem diferentes funcionalidades, dependendo do estado atual do telemóvel, podendo mostrar o ecrã de desbloqueio quando o telemóvel está bloqueado, bloquear o ecrã quando está desbloqueado e ainda mostrar o ecrã indicador de carga quando está desligado em carregamento.

Strategy

#Intenção

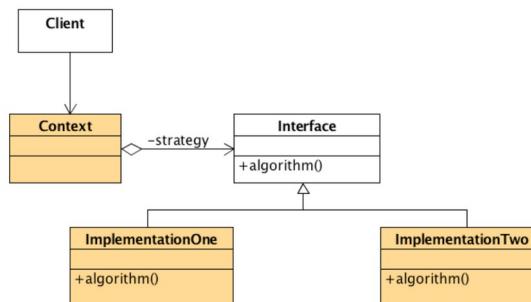
Definir uma **família de algoritmos** cujos objetos sejam imutáveis.

#Problema

A cada novo algoritmo a nossa classe cresce e torna-se desorganizada.

#Solução

Extrair os algoritmos para classes separadas, **strategies**. Como o nome indica, cada uma vai oferecer uma estratégia para resolver o problema.



#Exemplos

1. Contas

```

public interface Strategy { double compute(double elem1, double elem2); }

public class Sum implements Strategy {
    @Override
    public double compute(double elem1, double elem2) {
        return elem1 + elem2;
    }
}

public class Multiplication implements Strategy {
    @Override
    public double compute(double elem1, double elem2) {
        return elem1 * elem2;
    }
}

public class Subtraction implements Strategy {
    @Override
    public double compute(double elem1, double elem2) {
        return elem1 - elem2;
    }
}
  
```

```

public class StrategyDemo {
    public static void main(String[] args) {
        double e1 = 5, e2 = 33;
        Context c = new Context(new Sum());
        System.out.println("Result: " + c.compute(e1, e2));
        c.setStrategy(new Subtraction());
        System.out.println("Result: " + c.compute(e1, e2));
        c.setStrategy(new Multiplication());
        System.out.println("Result: " + c.compute(e1, e2));
    }
}
  
```

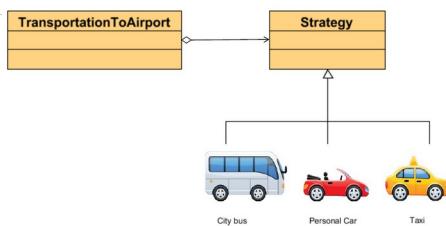
Result: 38.0
Result: -28.0
Result: 165.0

2. Compressão de ficheiros

```

// compressing files in several formats
public class Client {
    public static void main(String[] args) {
        // get a list of files (filelist)
        ...
        CompressionContext ctx = new CompressionContext();
        ctx.setCompressionStrategy(new ZipCompressionStrategy());
        ctx.createArchive(filelist);
        ctx.setCompressionStrategy(new RarCompressionStrategy());
        ctx.createArchive(filelist);
        ctx.setCompressionStrategy(new NoCompressionStrategy());
        ctx.createArchive(filelist);
    }
}
  
```

3. Analogia ao mundo real



Template Method

#Intenção

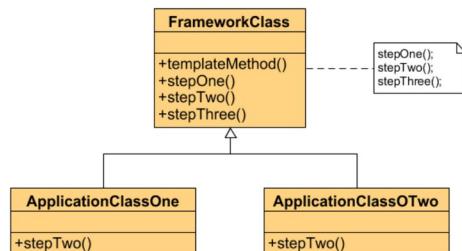
Programar classes que **partilham métodos e atributos comuns**, definindo um esqueleto declarativo e com algumas implementações, que podem ser *overriden* nas descendentes.

#Problema

As classes são semelhantes havendo duplicação de parte da sua implementação.

#Solução

Criar uma superclasse que implemente a maioria dos métodos e atributos, que podem ser *overriden* nas classes descendentes.



#Check list

- ✓ Estandardizar o esqueleto de um algoritmo numa classe base;
 - Definir a implementação dos métodos comuns;
- ✓ Criar uma classe base para cada implementação específica;
 - Fazer *override* dos métodos com implementação distinta da classe base;

#Exemplos

1. Jogos

```

class Monopoly extends Game {
    void initializeGame() {
        // Initialize players
        // Initialize money
    }
    void makePlay(int player) {
        // Process one turn of player
    }
    boolean endOfGame() {
        // Return true if game is over
        // according to Monopoly rules
    }
    void printWinner() {
        // Display who won
    }
    ...
}

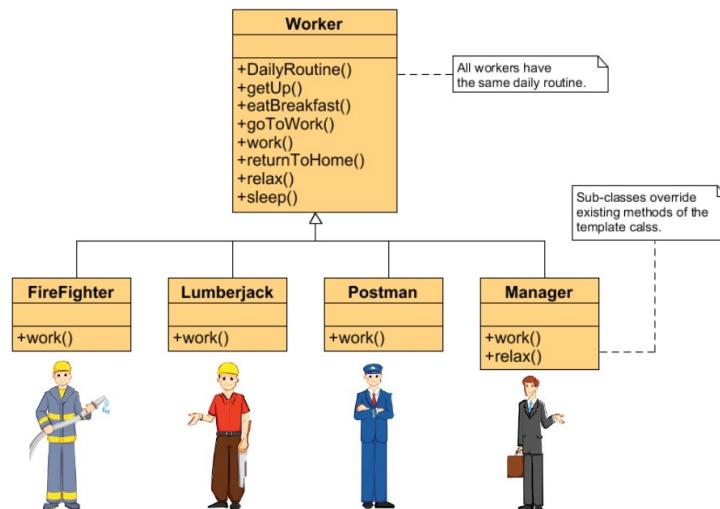
class Chess extends Game {
    void initializeGame() {
        // Initialize players
        // Put the pieces on the board
    }
    void makePlay(int player) {
        // Process a turn of player
    }
    boolean endOfGame() {
        // Return true if Checkmate or
        // Stalemate has been reached
    }
    void printWinner() {
        // Display the winning player
    }
    ...
}

public abstract class Game {
    protected int playersCount;

    public abstract void initializeGame();
    public abstract void makePlay(int player);
    public abstract boolean endOfGame();
    public abstract void printWinner();
    /* A template method : */
    public final void playOneGame(int playersCount) {
        this.playersCount = playersCount;
        initializeGame();
        int j = 0;
        while (!endOfGame()) {
            makePlay(j); j = (j + 1) % playersCount;
        }
        printWinner();
    }
}

```

2. Analogia com o mundo real



Visitor

#Intenção

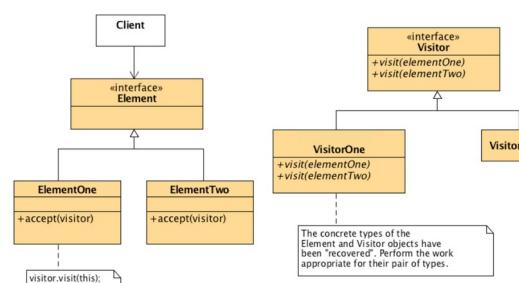
Separar algoritmos dos objetos sobre os quais estes operam.

#Problema

A possibilidade da realização de várias operações sobre um objeto vai “encher” a nossa classe e por vezes não a podemos alterar para adicionar uma nova.

#Solução

Criar uma classe **visitor** para cada comportamento, que define um método para cada elemento, sendo o elemento que invoca o *visitor*, passado como argumento.



Estamos perante uma situação de **Double Dispatch**.

#Check list

Caso a hierarquia seja estável e a interface pública das classes seja suficiente para a o acesso à informação que o *visitor* necessita.

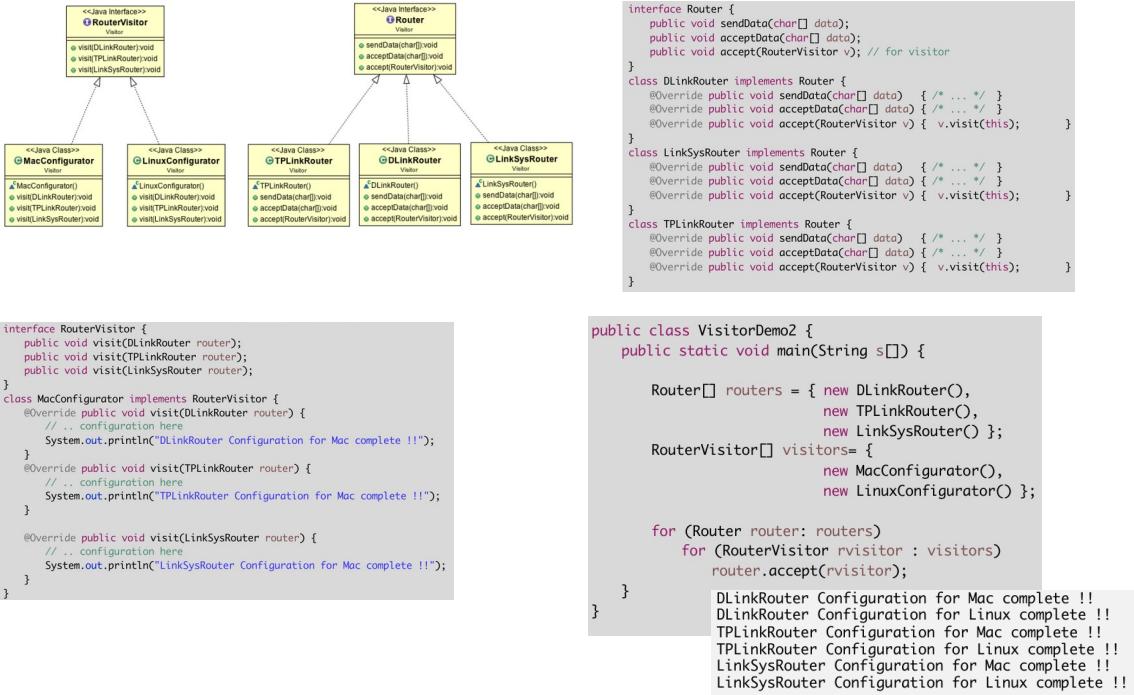
- ✓ Criar uma classe base **visitor** com um método *visit()* para cada elemento;
- ✓ Adicionar um método *accept(Visitor)* aos elementos;
- ✓ Criar uma classe derivada do **visitor** para cada operação a realizar.

A classe **Element** está acoplada à classe base **Visitor**, mas as suas classes descendentes estão acopladas a todas as classes derivadas da primeira.

Por este motivo, adicionar uma nova operação é fácil, bastando a criação de um novo **Visitor**. No entanto, adicionar um novo **Element** implica a criação de um método para este em todos os **Visitors**.

#Exemplos

1. Routers e SO



2. Numa analogia ao mundo real, um agente de seguros vende diferentes tipos de seguros a diferentes entidades: a um banco vende contra roubo, a uma pessoa particular de saúde e a um estabelecimento comercial contra incêndios, por exemplo.

Considerações finais

State, Strategy e Bridge

As suas estruturas são semelhantes, mas diferem na sua intenção, resolvendo problemas distintos.

7. Padrões de arquitetura de software

Capítulo com base nos slides teóricos e no livro [Software Architecture Patterns, de Mark Richards](#)

Sem uma arquitetura clara definida, o desenvolvimento de aplicações tem por base muitas das vezes a arquitetura em camadas. São desenvolvidas em resultado um conjunto de módulos com papéis pouco definidos, altamente acoplados e muitas vezes instáveis.

Os **padrões de arquitetura** são a resposta a estes problemas, definindo as características básicas e comportamento das aplicações, procurando responder a algumas das seguintes questões:

A arquitetura é escalável? Qual a performance da aplicação? Quão facilmente a aplicação responde a mudanças? Quais as características de *deploy*? Quão responsiva é a arquitetura?

	Layered	Event-driven	Microkernel	Microservices	Space-based
Overall Agility	↓	↑	↑	↑	↑
Deployment	↓	↑	↑	↑	↑
Testability	↑	↓	↑	↑	↓
Performance	↓	↑	↑	↓	↑
Scalability	↓	↑	↓	↑	↑
Development	↑	↓	↓	↑	↓

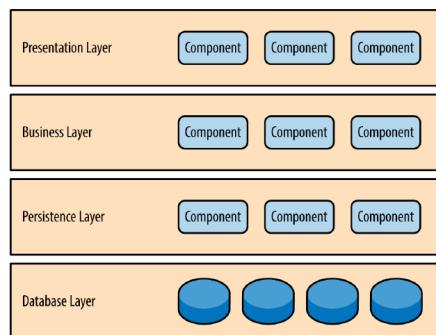
Layered Architecture

É o padrão mais conhecido, também denominado por ***n-tier architecture pattern***.

É o *standart* do desenvolvimento JAVA EE (*Enterprise Edition*) e vai de acordo à organização das estruturas de IT de muitas empresas.

#Descrição

Os seus componentes são organizados em camadas horizontais, cada uma com um papel específico na aplicação.



A sua característica principal é a **separação dos papéis** que cada camada desempenha, sendo este específico para cada uma, que lida apenas com a lógica adjacente a si mesma. Cada camada forma assim uma camada de **abstração**.

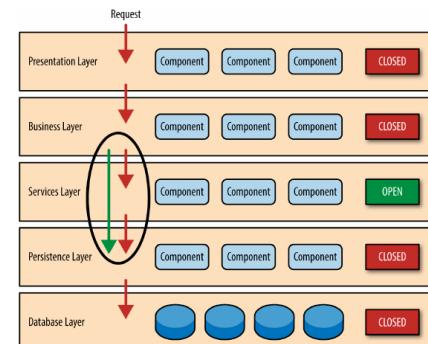
Por exemplo, a camada de apresentação não precisa de saber como obter os dados do utilizador, tendo como única preocupação a forma como os vai apresentar, a única lógica que implementa.

#Conceitos

A maioria das camadas são **fechadas** (*closed layers*), caracterizando-se por na passagem dos pedidos entre camadas, estes **só podem ser passados às camadas seguintes**.

Existem no entanto serviços que fazem sentido estar integrados numa camada para serem acessíveis apenas pela sua camada superior, mas que não são sempre necessários, podendo por isso por vezes ser “ignorados”. A estas camadas chamamos ***open layers***.

Criam-se assim ***layers of isolation***, um conceito que define que as alterações feitas numa camada não têm qualquer impacto nos componentes das outras camadas, havendo **independência entre as camadas**.



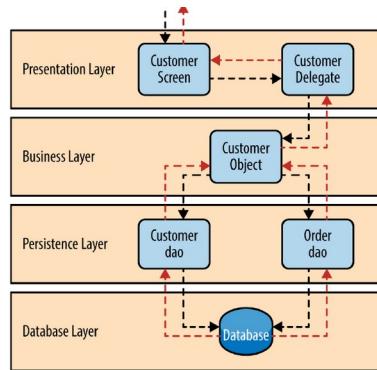
#Exemplos

Para obter informação de um cliente na aplicação de uma empresa, o monitor do cliente é responsável por receber o pedido e mostrar a informação associada. Este não sabe onde os dados são armazenados, nem como são obtidos, limitando-se a encaminhar o pedido para o módulo customer delegate, que contacta um módulo na camada inferior.

O customer object, por sua vez, agrupa toda a informação necessária para responder ao pedido, chamando dois módulos da camada inferior para a obter.

Finalmente os módulos customer dao (data access object) e order dao, vão executar *queries* de consulta dos dados da base de dados, na camada inferior.

As respostas são depois enviadas no sentido inverso, até chegar à camada de apresentação, onde a informação é então apresentada.



Um outro exemplo pode ter por base o desenvolvimento de interfaces para o acesso a bases de dados, tarefa é proposta pela unidade curricular de Base de Dados.



#Considerações

É um padrão que define uma solução sólida e relativamente abrangente.

No entanto, devemos ter em atenção ao ***sinkhole anti-pattern***, que ocorre quando os **pedidos passam por muitas camadas sem serem processados** (apenas encaminhados).

Por exemplo em consultas SQL simples, a camada *business* pouco ou nada faz, limitando-se a encaminhar o pedido à *persistence layer*.

Aplica-se a **regra 80-20**, que define como máximo 20% dos pedidos a apenas serem encaminhados.

#Análise

Agilidade Facilidade de responder a mudanças constantes no ambiente	Baixa	Apesar das mudanças nas camadas serem isoladas, continua a haver um acoplamento elevado dos componentes.
Facilidade de deploy	Baixa	Particularmente complicado em aplicações maiores, pois uma pequena alteração num componente pode criar a necessidade de <i>deploy</i> de toda a aplicação.
Facilidade de testes	Alta	Devido à independência entre as camadas, estas podem ser testadas individualmente.
Performance	Baixa	O facto dos pedidos terem de percorrer várias camadas pode levar a quebras no desempenho.
Escalabilidade	Baixa	Devido ao acoplamento elevado entre os seus componentes, geralmente a escalabilidade é reduzida. Uma possibilidade é a divisão das camadas em implementações físicas distintas, o que no entanto aumenta a granularidade, elevando o custo da mesma.
Facilidade de desenvolvimento	Alta	Devido à sua popularidade e simplicidade de implementação. Tem uma ligação forte à maneira como as empresas comunicam e se organizam (ver Google "Conway's law").

Event-Driven Architecture

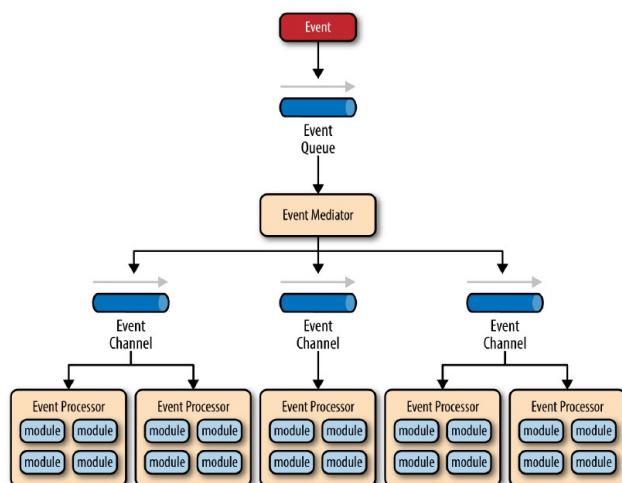
Consiste num padrão de arquitetura distribuída assíncrona utilizado para criar aplicações escaláveis, também conhecido por **message-driven** ou **stream processing**. É composto por componentes individuais desacoplados que recebem e processam eventos assíncronamente.

#Mediator Topology

Útil em eventos com várias etapas e que necessitam de orquestração para ser processados. Existem para esta arquitetura quatro tipos de componentes.

Event queues para onde o cliente envia o evento, sendo este depois entregue ao mediador
Event mediator recebe o evento inicial e orquestra-lo, enviando novos eventos assíncronos (*processing events*) para os canais de forma a executar cada passo do processo
Event channels agregam processadores

Event processors recebem eventos do mediador, sobre os quais executam uma determinada lógica. São *self-contained*, independentes e desacoplados dos restantes.



Para exemplificar como funciona esta topologia, suponhamos que uma pessoa muda de casa e informa a sua seguradora, através de um evento *relocation*. Os passos envolvidos no seu processamento estão contidos num *event mediator*, como mostrado na figura acima, que por cada passo vai criar um *processing event*, que é depois enviado para o canal correspondente, onde será processado pelo seu *event processor*. O processo continua até que todos os passos estejam concluídos.

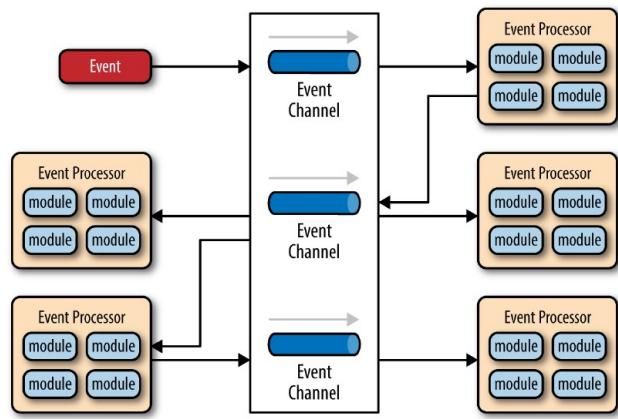
#Broker Topology

Para eventos com um **fluxo de processamento simples**, esta é a topologia mais indicada.

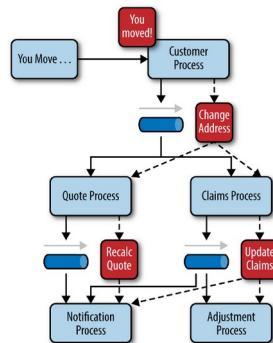
Sem qualquer mediador, o fluxo de mensagens é distribuído por componentes de processamento em cadeia através de um **message broker**. Destacam-se dois componentes.

Broker

Processador de eventos responsável pelo processamento de um evento e pela publicação de um novo indicativo do evento processado



Pegando no exemplo anterior, quando a seguradora é informada do evento *relocation*, este é recebido diretamente pelo processador de eventos responsável por alterar a morada dos clientes, gerando um novo evento *change address*. Suponhamos agora que há dois processadores de evento interessados neste novo evento, que vão em simultâneo realizar operações sobre este, gerando novos eventos e assim sucessivamente, até que não hajam mais eventos a realizar.



#Considerações

Este padrão é bastante **complexo**, principalmente devido à sua natureza assíncrona.

Devido ao desacoplamento e independência dos componentes, é **difícil manter uma unidade de trabalho transacional entre eles**.

Se durante o seu desenvolvimento nos aperceber-mos de que estamos a separar uma unidade de trabalho em vários processadores de eventos, então este não será o padrão indicado para essa solução.

Os aspetos chave são a **criação, manutenção e gestão** dos contratos associados a cada evento (que definem os valores e formatos dos dados aceites).

#Análise

Agilidade	Facilidade de responder a mudanças constantes no ambiente	Alta	Uma vez que os componentes de processamento têm um único propósito e estão desacoplados, as modificações são isoladas a um número restrito de processadores.
Facilidade de deploy		Alta	Novamente a natureza desacoplada dos componentes permite um <i>deploy</i> simples. A topologia do <i>broker</i> é mais simples do que a do <i>mediator</i> , pois no último os mediadores e os processadores estão de alguma forma conectados, pelo que uma alteração num componente implica a alteração no mediador.
Facilidade de testes		Baixa	Testes dos componentes individuais são relativamente simples. No entanto, são necessários testes especializados para gerar eventos.
Performance		Alta	A possibilidade de realizar operações assíncronas, desacopladas e em paralelo permite uma alta performance.
Escalabilidade		Alta	Novamente o desacoplamento e a independência dos componentes permitem uma escalabilidade alta e específica para certos componentes.
Facilidade de desenvolvimento		Baixa	A natureza assíncrona cria algumas dificuldades no desenvolvimento, nomeadamente devido à necessidade da gestão de erros como processadores de eventos não responsivos ou <i>brokers</i> indisponíveis.

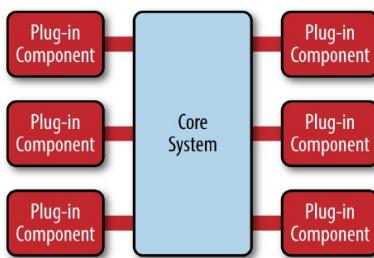
Microkernel Architecture

Também conhecido por **plug-in**, este padrão é utilizado para **implementar funcionalidades extra para uma aplicação de forma isolada**, sendo a sua produção, manutenção e atualização 100% independentes desta. É muito utilizada para criar aplicações *product-based*².

O nome deste padrão tem origem na arquitetura *microkernel* dos sistemas operativos.

#Descrição

O padrão tem por base dois componentes, o **core system** e os **módulos plug-in**, os últimos independentes do primeiro, fornecendo extensibilidade, flexibilidade e isolamento das funcionalidades da lógica da aplicação.



O **core system** geralmente **providencia apenas as funcionalidades básicas** e estritamente necessárias ao funcionamento do sistema. Deve ainda ter conhecimento dos *plug-ins* disponíveis e de como os obter (nome, contratos e protocolos de acesso).

Como exemplo consideremos o *plug-in* AuditCheck, com um determinado contrato que define os formatos e especificidades dos dados de entrada e saída e o protocolo de comunicação XML.

Os **módulos plug-in** devem também manter a independência entre si, podendo no entanto requerer a existência de outros para funcionar, devendo mesmo assim restringir as comunicações ao máximo.

A forma como estão conectados ao *core system* pode variar de acordo com a implementação, sendo comum a utilização de *adapters*.

² Que são encapsuladas num *package* e organizadas em versões, geralmente disponibilizadas como um produto de terceiros.

#Exemplo

Um exemplo da aplicação prática deste padrão que utilizamos com frequência é um navegador ou um IDE. Ambos providenciam as funcionalidades básicas para o acesso à internet e escrita de código, respetivamente, disponibilizando uma “loja” de *plug-ins* que podem ser instalados, adicionando novas funcionalidades à aplicação base.

No mundo empresarial podemos pensar numa seguradora internacional, que naturalmente tem de seguir a legislação de todos os países onde opera. Assim, para processar um pedido de reembolso devido a um sinistro, tem de seguir diferentes procedimentos dependendo do país de origem. Para evitar a criação de regras complexas e código denso, pode ser desenvolvido um *plug-in* para cada país, simplificando assim a implementação e manutenção, pois uma alteração legislativa num país tem impacto num único *plug-in*.

#Considerações

Um dos grandes benefícios deste padrão é a **grande abertura ao desenvolvimento incremental**.

As arquiteturas *microkernel* podem ser embutidas ou utilizadas como parte de outro padrão arquitetural. Podem facilmente ser convertidas noutra.

Para **product-based applications** é a melhor escolha, especialmente quando há a previsão de lançamento de funcionalidades adicionais ao longo do tempo apenas para alguns utilizadores.

#Análise

Agilidade	Facilidade de responder a mudanças constantes no ambiente	Alta	Devido à natureza modular das soluções desenvolvidas com este padrão, as mudanças são isoladas e rapidamente implementáveis. Devido à sua simplicidade, o <i>core system</i> tem tendência a ter um desenvolvimento estabilizado rapidamente, com alterações pontuais ao longo do tempo.
Facilidade de deploy		Alta	Novamente a natureza desacoplada dos componentes permite um <i>deploy</i> simples e em tempo de execução.
Facilidade de testes		Alta	Os testes podem ser isolados.
Performance		Alta	A modularidade permite a criação de aplicações com as funcionalidades estritamente necessárias, dispensando a implementação de código desnecessário.
Escalabilidade		Baixa	Geralmente as implementações são pequenas, implementáveis em unidades individuais e por isso pouco escaláveis.
Facilidade de desenvolvimento		Baixa	O desenvolvimento destas aplicações implica um planeamento e desenho cuidado dos contratos, nomeadamente quanto ao registo dos <i>plug-ins</i> , à sua granularidade e diversas possibilidades de comunicação interna.

Microservices Architecture Pattern

Este padrão apresenta-se como uma alternativa a aplicações monolíticas (que recorrem à *layered architecture*) ou orientadas a serviços (SOA), procurando simplificar o seu **desenvolvimento através da sua divisão em pequenos serviços**.

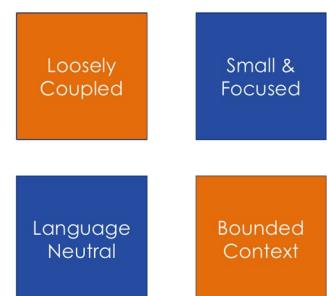
A evolução das aplicações monolíticas ocorreu devido à introdução do desenvolvimento incremental, uma vez que estas aplicações consistem em unidades de *deploy* único, trazendo problemas de manutenção futura.

Quando às SOA (*Service Oriented Applications*), surgiu da necessidade de simplificar a complexidade, custo e dificuldade de compreensão e implementação desta arquitetura, eliminando a necessidade de orquestração e simplificando o acesso entre os vários componentes.

#Descrição

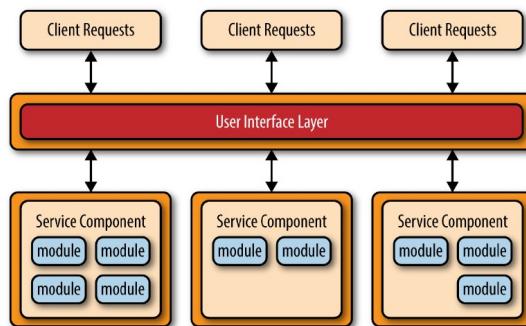
O principal conceito deste padrão é a noção de **deploy de unidades separadas**. Estas unidades são **componentes que prestam um serviço** e podem consistir num pequeno módulo ou numa grande porção da aplicação.

Em qualquer um dos cenários deve ser assegurada uma **arquitetura distribuída**, para a qual é fundamental o **desacoplamento** entre os módulos e a definição do seu protocolo de acesso (REST, SOAP, JMS, ...).



O grande desafio é atingir o grau certo de granularidade.

Se for necessário orquestrar ou passar mensagens entre os componentes na camada da interface ou API, então o serviço está demasiado granulado. Caso esta não consiga ser reduzida (ou seja, por mais que tentemos a orquestração não deixa de ser necessária), então este não será o padrão adequado para desenvolver a nossa aplicação.



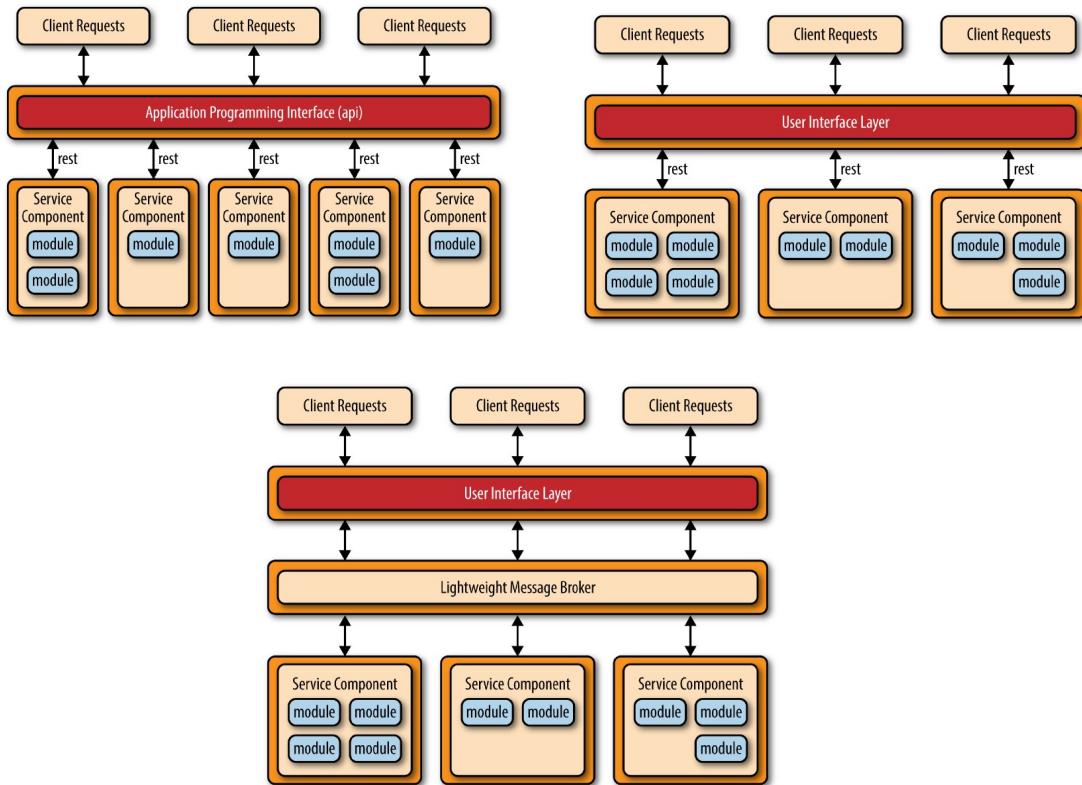
#Topologia

Há três formas de implementar esta arquitetura.

API REST-based, que consiste em disponibilizar uma API para aceder a um número reduzido de serviços individuais e independentes

application REST-based, que consiste em disponibilizar uma interface (*deployed* em separado)

centralized messaging topology, semelhante à anterior, mas em vez de utilizar REST para acesso remoto, faz uso de um **message broker**, que funciona como um simples roteador



#Considerações

Devido à sua natureza, este padrão é geralmente **robusto**, e facilmente **escalável**, sendo o suporte algo fácil de manter ao longo de tempo.

O isolamento dos componentes permite também **deploys em real time**, pois apenas os serviços alterados precisam de ser modificados e as interfaces de acesso podem ser configuradas com páginas de acesso temporariamente indisponível.

No entanto, por ser uma arquitetura distribuída, tal como a *event-driven*, partilha de problemas complexos como a criação de contratos, manutenção, disponibilidade e autenticação.

#Análise

Agilidade	Facilidade de responder a mudanças constantes no ambiente	Alta	Uma vez que os componentes de processamento têm um único propósito e estão desacoplados, as modificações são isoladas a um número restrito de processadores.
Facilidade de deploy		Alta	Novamente a natureza desacoplada dos componentes permite um <i>deploy</i> simples.
Facilidade de testes		Alta	Testes dos componentes individuais são muito mais simples que numa aplicação monolítica, sendo ainda excluída a possibilidade de uma alteração num componente interferir com o funcionamento de outro.
Performance		Baixa	Devido à sua natureza distribuída as aplicações desenvolvidas segundo este padrão tendem a ter uma performance mais reduzida quando comparadas com outras.
Escalabilidade		Alta	Novamente o desacoplamento e a independência dos componentes permitem uma escalabilidade alta e específica para certos componentes.
Facilidade de desenvolvimento		Alta	Devido à funcionalidade estar isolada em cada componente, o desenvolvimento é focado em unidades funcionais e por isso mais simples.

Space-Based Architecture

A maioria das aplicações web gera os pedidos num fluxo **servidor > aplicação > base de dados**, fornecendo uma resposta em tempo útil para um fluxo normal de acesso.

No entanto, quando o fluxo começa a crescer começamos a assistir a um aumento do congestionamento do tráfego. Inicialmente no servidor, que pode ser expandido facilmente e sem grandes custos, mas que faz com que se comecem a sentir limitações na aplicação, esta mais difícil e cara para expandir e por fim na base de dados, que também requere um grande investimento para a sua expansão.

A **space-based architecture**, também conhecido por **cloud architecture** procura então resolver os problemas de escalabilidade e concorrência, consistindo numa melhor alternativa à de tentar escalar os elementos do referido fluxo.

#Descrição

O nome deste padrão vem do conceito de **tuple space**, um paradigma de memória partilhada distribuída que implementa através da remoção da base de dados central, criando uma rede de dados replicada em memória.

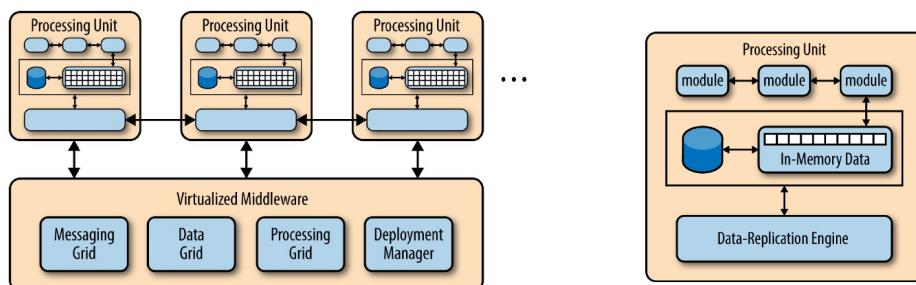
Cada **unidade de processamento** vai então ter uma cópia da base de dados, pelo que estas podem ser inicializadas e desligadas dinâmicamente de acordo com a variação do tráfego de acesso.

Este padrão tem por base dois componentes...

Processing-unit, que contém os componentes da aplicação, *web-based* e lógica do negócio

Virtualized-middleware, que funciona como um controlador, gerindo as comunicações, sessões, replicação, processamento dos pedidos distribuídos e *deploy* das unidades de processamento

Uma aplicação simples pode ter apenas uma **processing-unit**. No entanto, aplicações de maiores dimensões podem dividir as funcionalidades em várias unidades de processamento, dependendo das suas áreas funcionais.



#Virtualized middleware

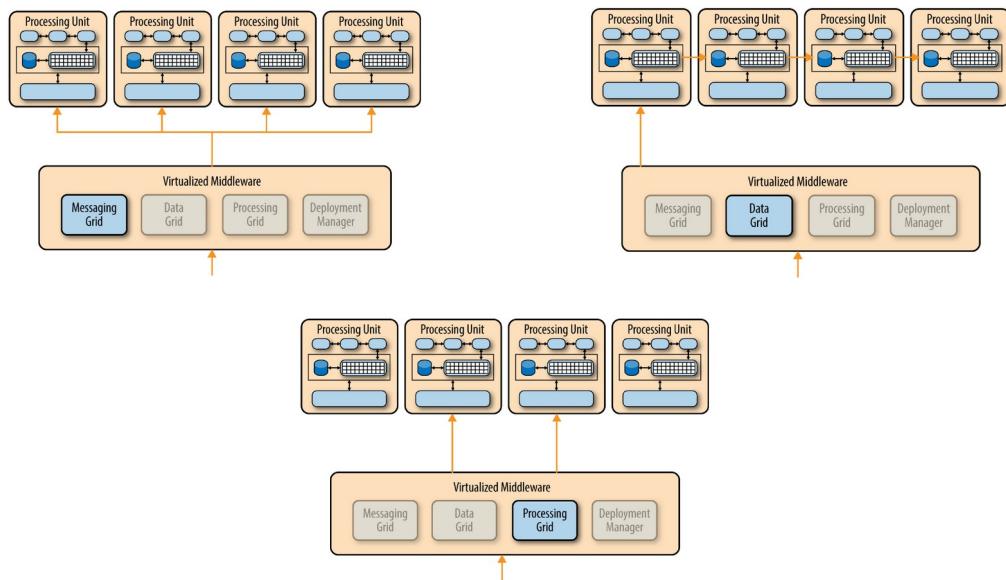
Como podemos observar na figura anterior, o **virtualized-middleware** divide-se em quatro componentes, que trabalham em conjunto para garantir um processamento rápido dos pedidos num sistema distribuído, mas consistente.

Messaging grid encaminha os pedidos recebidos para as unidades de processamento, podendo implementar mecanismos simples como *Round Robin* ou mais complexos, considerando a disponibilidade das unidades

Data grid interage com os mecanismos de replicação nas unidades de processamento, sendo responsável por gerir a replicação dos dados entre elas (feita de forma assíncrona, mas rápida)

Processing grid gera o processamento distribuído entre as unidades de processamento, mediando e orquestrando o trabalho entre as várias unidades

Deployment manager gera o *set up* dinâmico das unidades de processamento, estando para isso a monitorizar constantemente os tempos de resposta e a carga de dados transacionada, para garantir uma eficiência máxima do sistema



#Considerações

Este padrão é complexo e requere muitos recursos para ser implementado, por isso, apesar de ser uma boa solução para pequenas aplicações que têm variações pontuais no número de acessos, não é a melhor solução para grandes aplicações com um fluxo de dados grande constante.

#Análise

Agilidade Facilidade de responder a mudanças constantes no ambiente	Alta	Devido ao <i>deployment manager</i> , as aplicações conseguem responder muito bem a alterações no fluxo de acessos. Também consumam responder bem a alterações no código devido ao tamanho reduzido das aplicações e à sua natureza dinâmica.
Facilidade de deploy	Alta	Apesar de a sua natureza não ser desacoplada, o seu dinamismo e as ferramentas dos sistemas <i>cloud-based</i> permitem fazer um <i>deploy</i> relativamente simples.
Facilidade de testes	Baixa	Atingir um fluxo de acessos elevado em ambiente de testes é bastante dispendioso e <i>time consuming</i> .
Performance	Alta	Devido ao acesso <i>in-memory</i> e a mecanismos de <i>caching</i> este padrão consegue oferecer soluções de alta performance.
Escalabilidade	Alta	Devido à inexistência de dependências a uma base de dados central, este padrão é facilmente escalável.
Facilidade de desenvolvimento	Baixa	A memória partilhada replicada em todas as unidades de processamento e os mecanismos de <i>caching</i> são funcionalidades complexas de desenvolver.

- i **Heurística**, s. f. | No *software* aplica-se no desenvolvimento de soluções com base na experiência, que podem ser sub-ótimas, mas que sabemos capazes de resolver um problema. ±
- ii **Indeterminismo**, s. m. | Sistema filosófico segundo o qual a vontade humana não é estritamente determinada pelos motivos das nossas ações.
- iii **Emergente**, adj. 2 g. | Que resulta ou procede.
- iv **Refactoring**, n. Eng. | Processo de modificação de um sistema de *software* para melhorar a estrutura interna do seu código sem alterar o seu comportamento externo.
- v **Acoplamento**, s. m. | Junção ou engate de mecanismos para produzir trabalho.
- vi **Coesão** s. f. | União; Harmonia.
- vii **Encapsulamento**, s. m. | Encerrar em cápsula.
- viii **Low Representational Gap** | É uma ideia base no desenho orientado a objetos, procurando aproximar os modelos mentais dos de software, nomeadamente através da utilização de nomes ilustrativos e facilmente percetíveis.
- ix **Indirection** | Dissimulação, s. f. | Fingimento, disfarce, caractér dissimulativo.
- x **Overhead** | Combinação excessiva de tempo de computação indireta.