

Padrões e Desenho de Software

José Mendes 107188

2023



1 Introdução ao Desenho de Software

1.1 Desenho é uma atividade universal

Qualquer produto que é um agregado de elemento mais simples, pode beneficiar da atividade que é o Desenho (Design).

1.2 O que é o Desenho de Software?

No Software, este Desenho começa pela definição de **requisitos**. Este processo deve ser efetuado pelo programador junto dos Stakeholders, pelo que é fundamental este ter uma boa capacidade de comunicação e filtragem, devendo durante o mesmo ficar escalrecidas quais as necessidades do sistema a implementar.

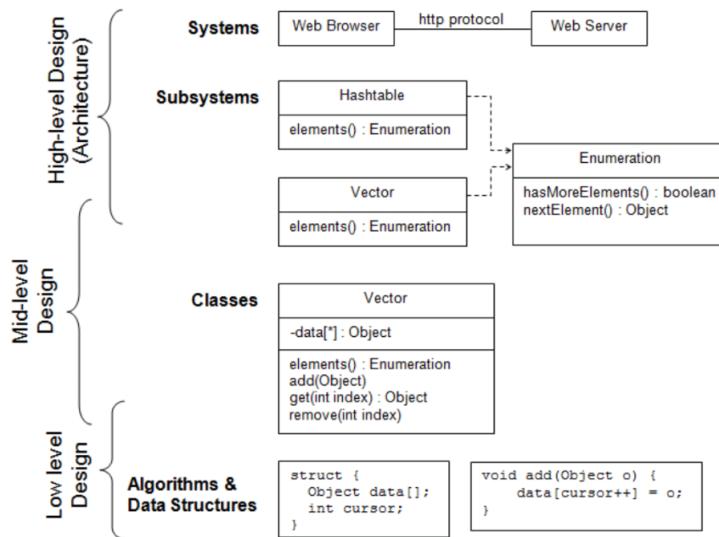
A seguinte fase é o **desenho**, onde se definem as classes, os métodos, as relações entre elas, os fluxos de trabalho, etc.

Todo o processo finda 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:

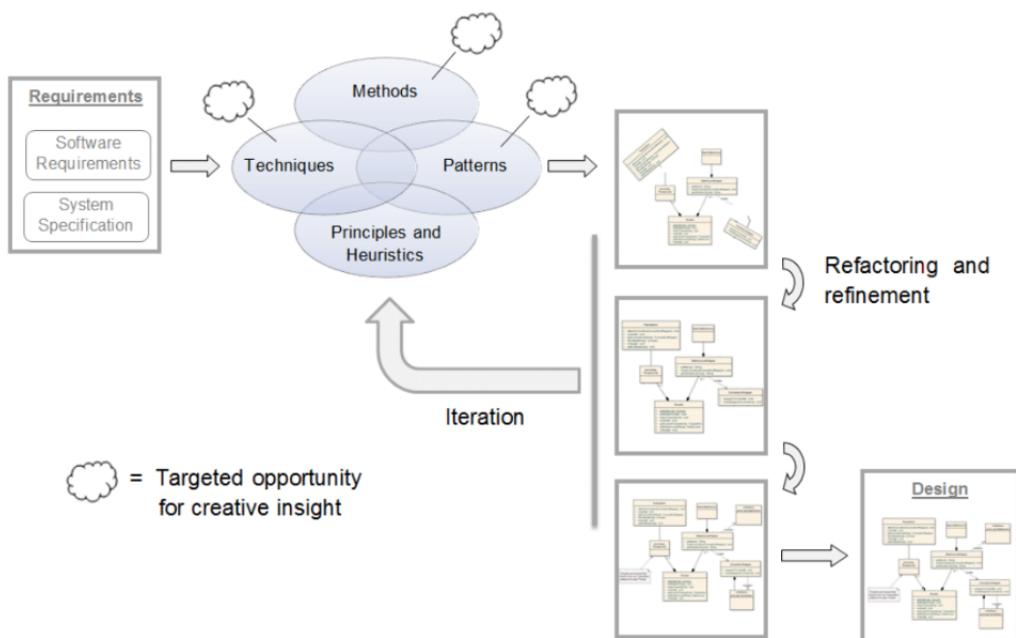
1. Sistema
2. Subsistemas/pacotes: User Interface, Data Storage, Application-level Classes, Graphic...
3. Classes dentro de pacotes, Relações entre classes, A Interface de cada Classe, métodos públicos
4. Atributos, Métodos privados, "Inner classes"...
5. Código fonte que implementa os métodos

1.3 O Desenho ocorre em diferentes níveis



1.4 A importância do Desenho de Software

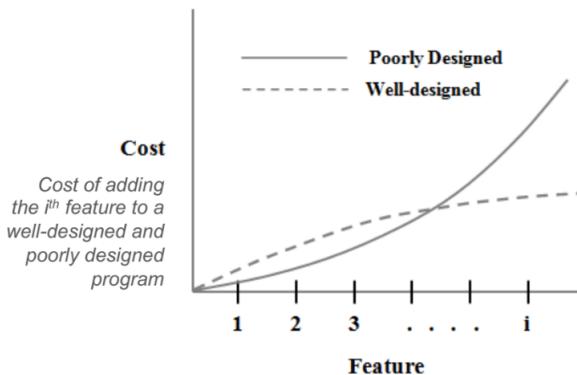
O processo de Desenho pode ser tornado mais **sistemático** e **previsível** através da aplicação de métodos, técnicas e padrões, todos aplicados de acordo com princípios e heurísticas.



1.5 A importância de controlar a complexidade

Um programa que seja mal desenhado, ou seja, muito complexo, é difícil de entender e modificar, podendo, a curto prazo, se mais rápido de desenvolver, mas acabando por se tornar insustentável a longo prazo.

Quanto maior for o programa, mais pronunciadas são as consequências de um mau desenho.



1.6 Dois tipos de complexidade no Software

O desenho é, portanto, uma ferramenta poderosa para o **controlo da complexidade** no desenvolvimento de Software, gerindo:

Complexidades Essenciais - São herdadas no problema

Complexidades Acidentais - Associadas à solução e à forma como é implementada

A quantidade total de complexidade numa solução de Software é:

$$\text{Complexidades Essenciais} + \text{Complexidades Acidentais}$$

O objetivo é portanto, **controlar as Complexidades Essenciais, enquanto impedimos a introdução de Complexidades Acidentais adicionais**.

1.7 Lidar com Complexidade de Software

Para evitar a complexidade no Desenho de Software devemos ter em conta os seguintes princípios:

Modularidade - Subdividir a solução em componentes mais pequenos e fáceis de controlar (DaC - Divide and Conquer).

Abstração - Usar abstração para omitir detalhes em sítios que estes não são necessários

Esconder Informação - Esconder detalhes e complexidades através de interfaces simples

Herança - Componentes genéricos podem ser reutilizados para definir elementos mais específicos

Composição - Reutilizar outros componentes para construir uma nova solução

1.8 O Desenho é um "Wicked Problem"

Um "Wicked Problem" é um problema que apenas pode ser definido claramente, através de uma solução.

"TEX would have been a complete failure if I had merely specified it and not participated fully in its initial implementation. The process of implementation constantly led me to unanticipated questions and to new insights about how the original specifications could be improved."

-Donald Knuth

1.9 Características do Desenho de Software

O desenho de Software caracteriza-se por ser:

Não determinístico - Uma vez que a maioria dos processos de desenho para o mesmo programa não atingem o mesmo resultado (output diferente)

Heurístico - Uma vez que as técnicas de desenho apoiam-se em heurísticas e em regras práticas (rules-of-thumb) em vez de processos repetitivos

Emergente - Uma vez que o resultado final evolui de acordo com a experiência e o feedback. O desenho é um processo iterativo e incremental onde a complexidade do sistema surge de interações relativamente simples

Desta forma, é fundamental acompanhar todas as fases de desenvolvimento, pois elas estão fortemente interligadas e durante o desenvolvimento de uma pode haver necessidade de reaçizar alterações nas anteriores.

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

1.10 Processo de Desenho Genérico

1. **Perceber** o problema (definição de requisitos de Software);
2. Construir um **modelo de solução**, "black-box", sendo as especificações de sistema geralmente representadas por casos de uso (use cases);
3. **Procurar por soluções existentes**, como por exemplo padrões de arquitetura ou desenho que abrangem alguns/todos os problemas identificados pelo desenho de Software;
4. Considerar construir **protótipos**;
5. Documentação e **revisão** do desenho;
6. **Refactor** (Iterate over the solution), ou seja, evoluir o desenho até ter os requisitos funcionais e maximizar os requisitos não funcionais;

1.11 Entradas (Inputs) para o processo de desenho

1. **Requisitos do utilizador** e especificações de sistema, incluindo qualquer restrição no desenho e implementação de opções;
2. **Conhecimento do domínio**, por exemplo, se for uma aplicação de um centro de saúde, o designer necessita de algum conhecimento dos termos e conceitos destes centros;
3. **implementação do conhecimento**, capacidades e limitações de eventuais ambientes de execução;

1.12 Desenho de características internas desejadas

Complexidade mínima - Manter simples. Talvez não seja preciso altos níveis de generalidade

Loose Coupling - Minimizar dependências entre modelos

Manutenção fácil - O teu código vai ser mais vezes lido do que escrito

Extensibilidade - Desempenho para hoje mas olhando para o futuro. Nota: Esta característica pode estar em conflito com a primeira, complexidade mínima. Engenharia é balanciar objetivos em conflito

Reutilizável - Reutilização é um sinal da maturidade da Engenharia

Portátil - Funciona ou pode facilmente funcionar em outros ambientes

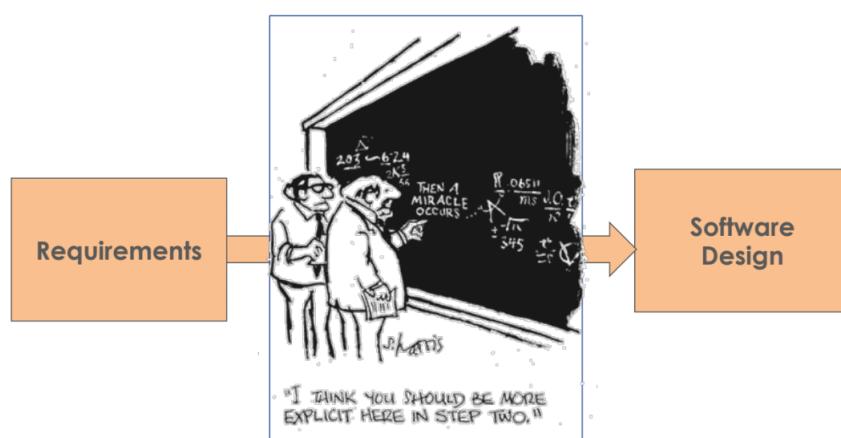
Fan-in alto - Em alguns módulos utility-type e low-to-medium fan-out em todos os módulos.

Fan-out elevado está tipicamente associado a complexidade elevada

Magreza - Quando em dúvida, deixar de fora. O custo de adicionar outra linha de código é muito mais do que os poucos minutos necessários para a escrever

Estratificação - Em camadas. Mesmo se todo o sistema não seguir a arquitetura por camadas, componentes individuais podem fazê-lo

Técnicas base - Por vezes é bom ser um conformista! Ser secante é bom. Código de produção não é o local para tentar experimentar técnicas

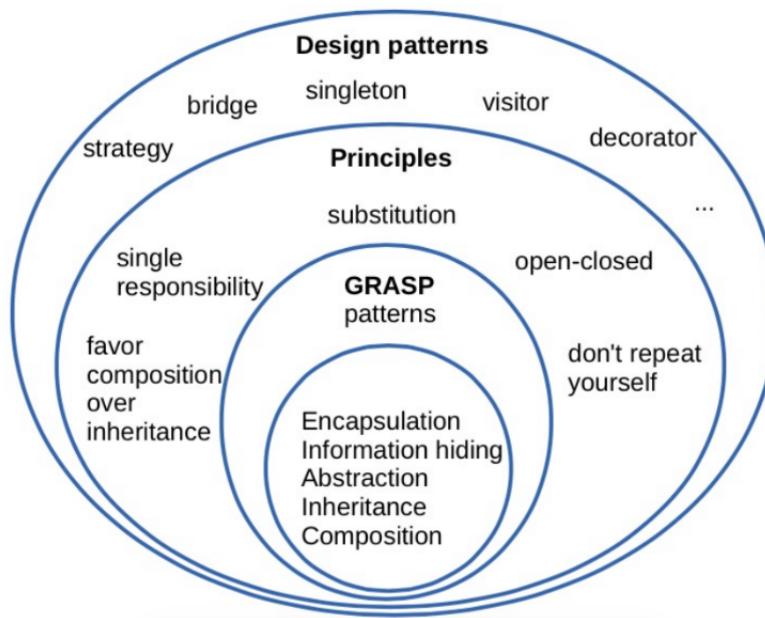


1.13 Padrões

Padrão de Desenho- Solução para um problema de desenho comum reutilizável

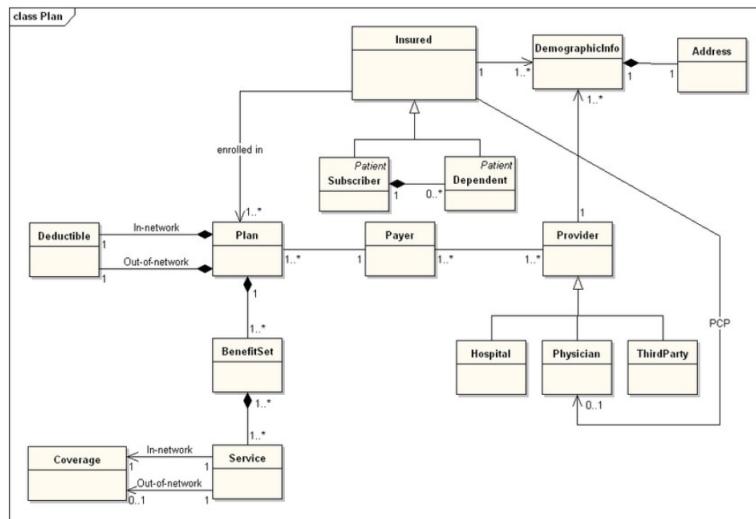
Os padrões de desenho são adaptados às características do problema que resolvem e dependendo do nível de desenho a que se aplicam, os padrões de desenho podem ser divididos em **padrões arquiteturais, padrões de desenho ou idiomas de programação.**

Não existe uma metodologia melhor para obter o melhor desenho de orientado a objetos, no entanto existem princípios, padrões, heurísticas.



1.14 Os diferentes modelos e a sua representação

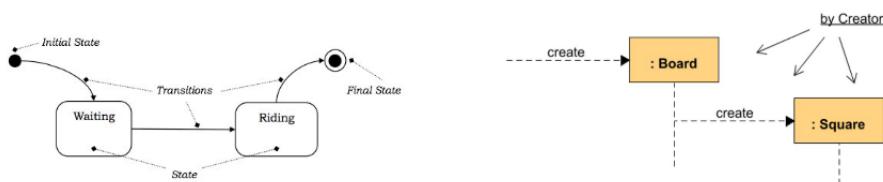
Em diferentes fases do desenho de Software são utilizados diferentes modelos para a sua representação. O **domínio de modelo** representa um modelo conceptual, incorporando tanto o comportamento, como a informação de cada entidade.



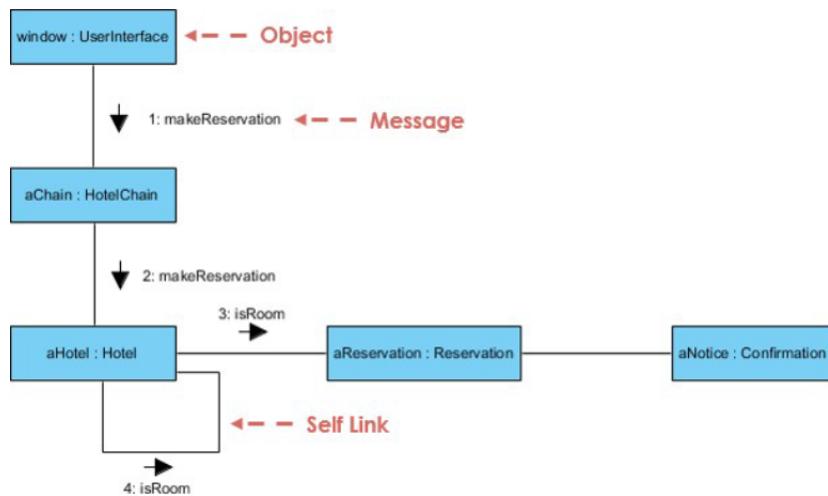
Nestes, a herança e a composição são respetivamente representados por:



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 Princípios GRASP

Uma das metodologias de desenvolvimento de Software mais comuns designa-se **Responsibility-driven Design, RDD** e consiste na definição dos objetos em termos de responsabilidade, papéis e colaboração.

Nota: 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, etc.

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

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

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

De uma forma geral:

1. Atribuir responsabilidade a uma classe
2. Evitar ou Minimizar dependências adicionais
3. Maximizar a coesão (cohesion) e Minimizar o acoplamento (coupling)
4. Aumentar a reutilização e reduzir a manutenção
5. Maximizar a compreensão do código...

Nota: 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.

Os princípios GRASP são:

1. Creator
2. Information Expert
3. Low Coupling
4. High Cohesion
5. Controller
6. Polymorphism
7. Pure Fabrication
8. Indirection
9. Protected Variations

2.1 Creator

Nota: Como resposta, a maioria dos programadores pensa imediatamente no container criar o elemento contido. Este raciocínio tem por base o **low representational gap** e não está errado. Na verdade, existem mais alguns critérios, sendo as condições para B criar A listadas abaixo.

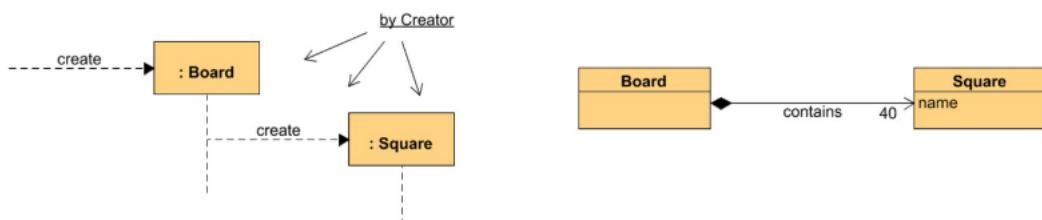
Problema: Quem cria a instância de A? (Fazer)

Solução: Atribuir à classe B a responsabilidade de criar uma instância da classe A se uma destas for verdade (quantas mais melhor):

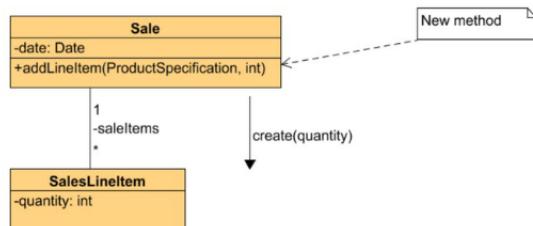
1. B **contém ou agrupa** (contém um conjunto de) A;
2. B **regista** A;
3. B **utiliza** com frequência A;
4. B **tem os dados de inicialização** de A;

Exemplo:

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.



Promove a minimização do acoplamento ao fazer uma instância de uma classe responsável por criar os objetos que referenciam. (**Pró**)

Coneectar um objeto ao seu Creator quando:

1. Aggregator aggregates Part
2. Container contains Content
3. Recorder records
4. Initializing data passed in during creation

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. (**Contra**)

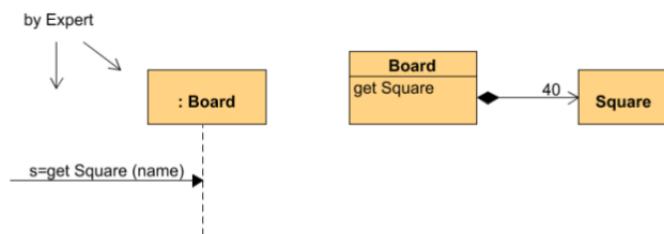
2.2 Information Expert

Problema: Qual o princípio para atribuir responsabilidades a objetos? (Saber e Fazer)

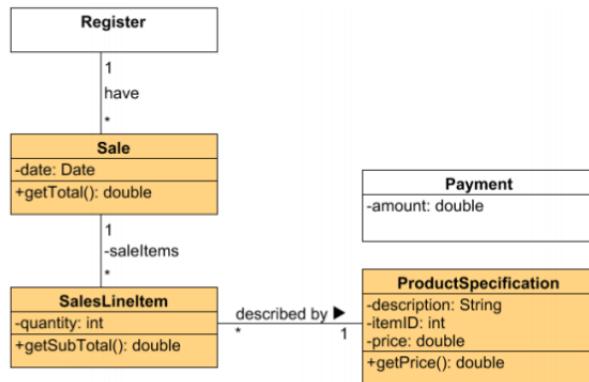
Solução: Atribuir a responsabilidade à classe que tem a informação necessária para a assumir.

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

Exemplo: 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.

Benefícios:

- Facilita o encapsulamento de informação:
 1. Cada classe utiliza assim a informação que dispõe para cumprir tarefas, tornando-se mais coesas
 2. Escrever código mais fácil de modo a ser percebido apenas lendo-o
- Promove **low coupling**

Contraindicações:

- Pode tornar uma classe excessivamente complexa, o que necessita outro tipo de separação, domínio e persistência.

2.3 Low Coupling

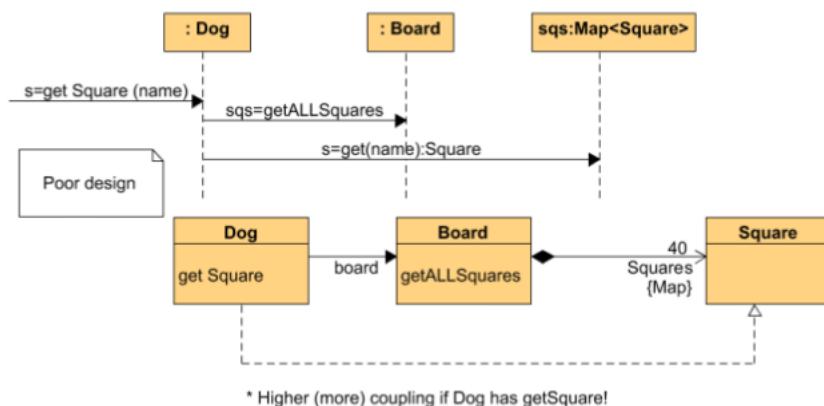
Problema: Como reduzir o impacto da mudança e encorajar a reutilização?

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

Nota: O **acoplamento** define-se como o quanto 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.

Exemplo: 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**.



Benefícios:

1. Compreensão: Classes são mais fáceis de compreender quando isoladas
2. Manter: Classes não são afetadas pelas mudanças em outros componentes
3. Reutilizável: Fácil de aceder a outras classes

Contraindicações: Em classes estáveis, um acoplamento maior não é um grande problema.

Nota: 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!

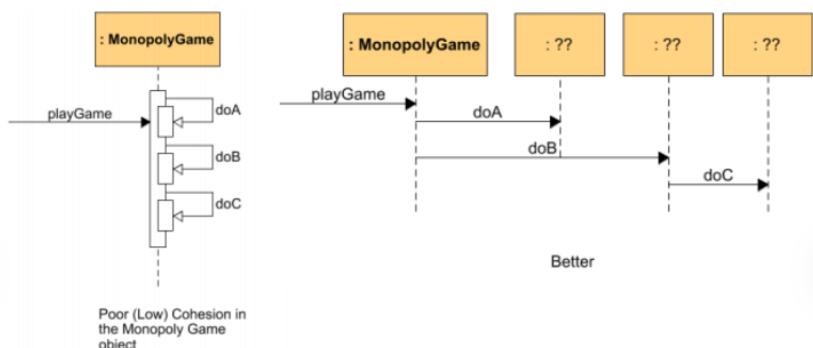
2.4 High Cohesion

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.

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



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

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

Contraindicações: 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.

2.5 Controller

Problema: Quem deve ser responsável por eventos UI?

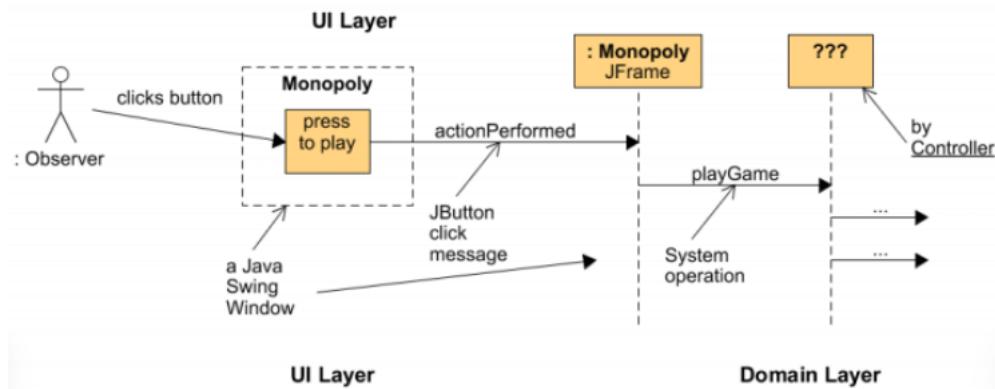
Solução: Se um programa recebe eventos de fontes externas além do seu GUI, adicionar uma classe que desassocia o evento, separando e fazendo a ponte entre as fontes do evento e os objetos que lidam com estes.

Atribuir a responsabilidade de lidar com a mensagem do evento do sistema a uma classe representando uma destas duas escolhas:

1. Classe representativa do modelo de negócio em que se insere, denominada de **façade controller**
2. Classe artificial, seguindo o método Pure Fabrication, denominada de **case controller**

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

Exemplo: 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.



Benefícios: 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.

Contraindicações: Não tem

2.6 Polymorphism

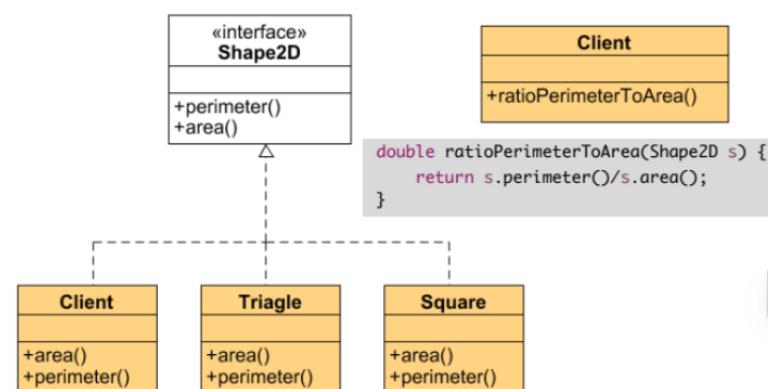
Problema: Como gerir o comportamento com base no tipo (i.e classe) mas sem usar os blocos if-then-else ou switch?

Solução: Quando comportamentos alternativos são selecionados com base no tipo do objeto, usar métodos polimórficos para escolher o comportamento, em vez de usar instruções condicionais para testar o tipo.

Métodos polimórficos: Atribuir o mesmo nome a serviços (métodos) distintos em classes diferentes. Os serviços são implementados por métodos.

Exemplo: 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.



Benefícios: Mais fácil e mais confiável do que usar lógica de seleção explícita. É mais fácil adicionar comportamentos mais tarde.

Se o polimorfismo não for utilizado, e forem usados blocos condicionais, então, esta secção de código vai crescer à medida que mais tipos são adicionados ao sistema. A secção de código fica mais acopelada (sabe de mais tipos) e menos coesa (está a fazer demasiado).

Contraindicações: Não tem

2.7 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 bastante coeso de responsabilidades a uma classe artificial que não representa nada no domínio no problema.

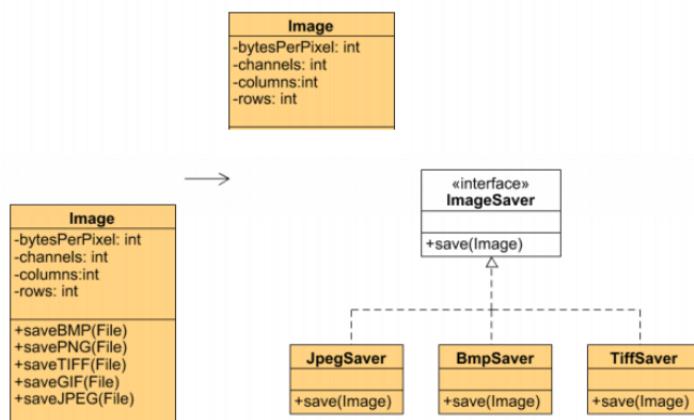
Por exemplo, no POS, se quisermos guardar o registo de cada venda numa base de dados relacional, pelo princípio Information Expert faria sentido atribui-la à classe **Venda**, no entanto, com esta atribuição a classe iria tornar-se pouco coesa e fortemente acoplada à base de dados.

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 Imagem, que segundo este princípio deve ser abstraída das operações de ser guardada em diferentes formatos.



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

Potencial para a reutilização deve aumentar, devido à presença de várias classes. Pure Fabrication.

Contraindicações: Não tem

2.8 Indirection

Problema: Como evitar acoplamento direto, desacoplando objetos de forma a suportar low coupling, e manter o potencial de reutilização elevado? **Solução:** Atribuir a responsabilidade a um objeto intermédio. Para ser o mediador entre outros componentes e serviços, de forma a não ser diretamente acoplados.

Indirection pode ser categorizada em:

1. Behavioural Extension
2. Interface Modification
3. Technology Encapsulation
4. Complexity Encapsulation

Exemplo: 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 sistema de pagamento para validar uma transação. Para tal, deve ser criada uma classe responsável por esta operação.

Benefícios: Permite baixo acoplamento e promove a reutilização.

Contraindicações: Não tem

2.9 Protected Variations

Problema: Como desenhar software de forma a que as suas variações não tenham um impacto negativo noutros 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!

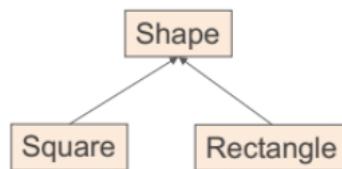
2.9.1 Liskov Substitution Principle (LSP)

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:

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

Exemplo: 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.



2.9.2 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.

Exemplo: 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!).

Benefícios: Mantém o coupling entre classes baixo e torna o desenho robusto
Adiciona uma pequena quantidade de overhead no formato de métodos indiretos

Contraindicações: Não tem

2.10 Outros princípios

2.10.1 SOLID

Single Responsibility

Cada classe deve assumir uma única responsabilidade e ser encapsulada interamente pela classe (encapsulamento máximo).

Open/closed (OCP)

As entidades de Software devem estar abertas à extensão, mas fechadas à alteração.

Liskov substitution (LSP)

Interface segregation

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

Dependency inversion

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

2.10.2 Minimalismo

1. Keep it simple, stupid (KISS)
2. Worse is better (Less is more)
3. You aren't gonna need it (YAGNI)
4. Principle of good enough (POGE)
5. Quick-and-dirty

2.10.3 DRY (Don't Repeat Yourself)

Cut and paste of code is evil.

2.10.4 IoC (Inversion of control)

3 Padrões de desenho

Padrão - Princípios e soluções estruturalmente codificados que descrevem uma solução para um determinado problema.

Um par problema/solução pode ser aplicado em diferentes contextos.

São apenas soluções que os programadores do passado se separaram e decidiram anotar para ajudar designers em novas situações.

A ideia por de trás dos padrões de desenho é simples, escrever e catalogar interações comuns entre objetos que os programadores acharam útil.

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:

Arquiteturais - Expressam uma estrutura organizacional do sistema como um todo;

De desenho - Estabelece um esquema para os subsistemas ou componentes de um sistema, ou as suas relações;

Idiomáticos - Descrevem como implementar um aspeto particular de um componente, ou as relações entre eles;

A pattern is a recurring solution to a standard problem, in a context.

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

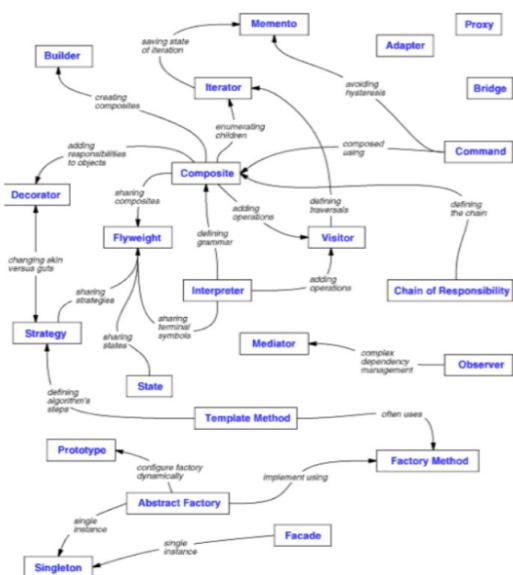
Creational - Relacionados com a criação de objetos;

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 encontram-se fortemente ligados:



Nota: 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

No que toca à construção de objetos, há dois grandes problemas: O construtor de um objeto não pode devolver um subtipo do seu tipo (**Factory method**);

O construtor devolve sempre uma nova instância daquela classe, não permitindo a sua reutilização (**Singleton**);

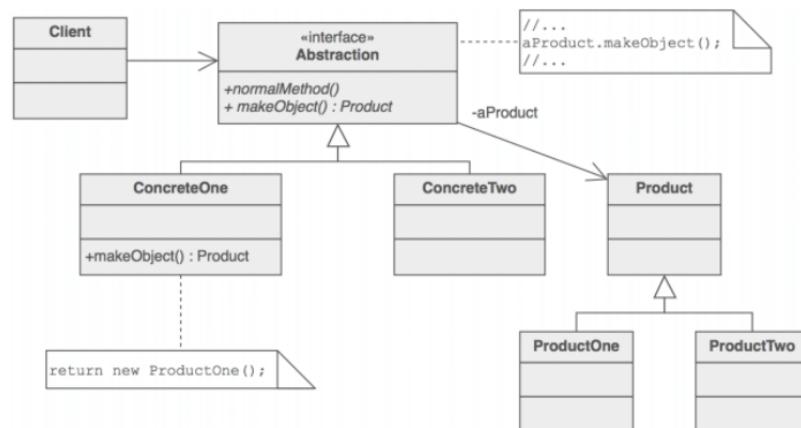
4.1 Class: Factory Method

Intenção - O operador "new" é perigoso. 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 objetos de domínio e permitir a sua instanciação.

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.

Estrutura:



Check List:

1. Se um construtor poder resultar em objetos inconsistentes, considerar desenhar um método fábrica;
2. Considerar fazer todos os construtores private ou protected;
3. Se hierarquia tiver herança, considerar definir o método de fábrica na classe base;
4. Considerar a criação de uma reserva de objetos criados (object pool), evitando assim a sua criação duplicada;

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!

Exemplo:

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 liberto de fazer new, limitando-se a invocar o método estátido no **Viveiro**.

```
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(); //...  
    }  
}
```

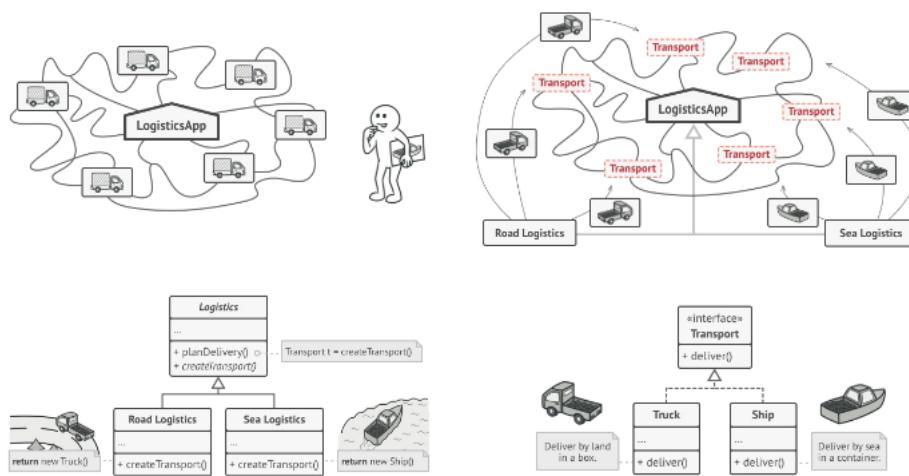


```
class Race {  
    Bicycle createBicycle() {  
        return new Bicycle();  
    }  
    Race createRace() {  
        Bicycle bike1 = createBicycle();  
        Bicycle bike2 = createBicycle(); //...  
    }  
}  
class TourDeFrance extends Race {  
    Bicycle createBicycle() {  
        return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

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

Uma transportadora 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.



4.2 Object: Abstract Factory

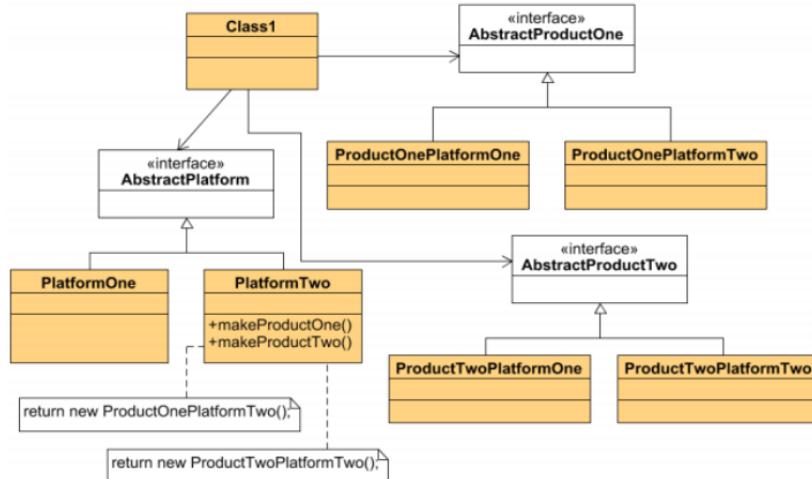
Intenção - O operador "new" é perigoso. Criar uma interface para criar famílias de objetos relacionados sem especificar a sua classe, através de uma hierarquia que encapsula várias **plataformas** e a construção de vários **produtos**.

Problema - Devendo uma aplicação ser portátil, é necessário que encapsule as suas dependências.

Estas plataformas podem ser: 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.

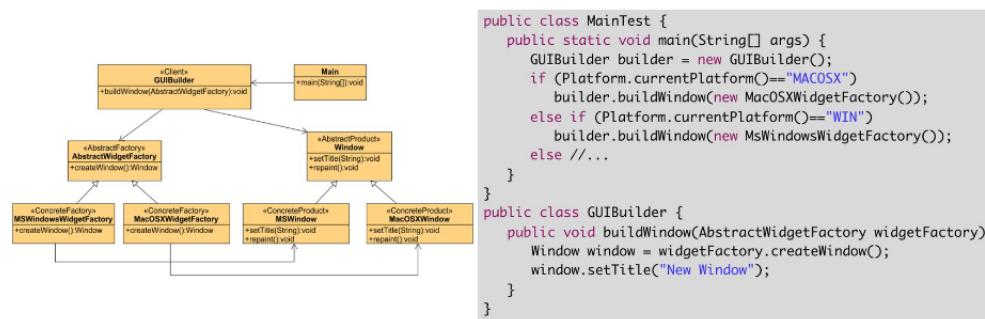
Estrutura:

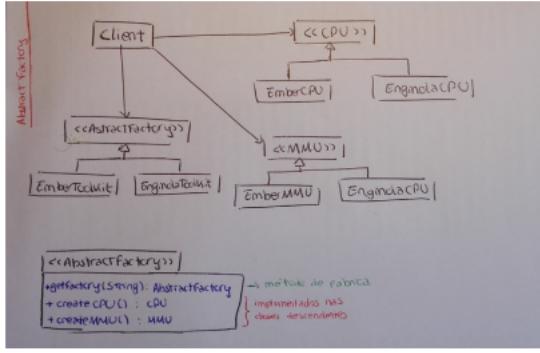


Check List:

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

Exemplo:





```

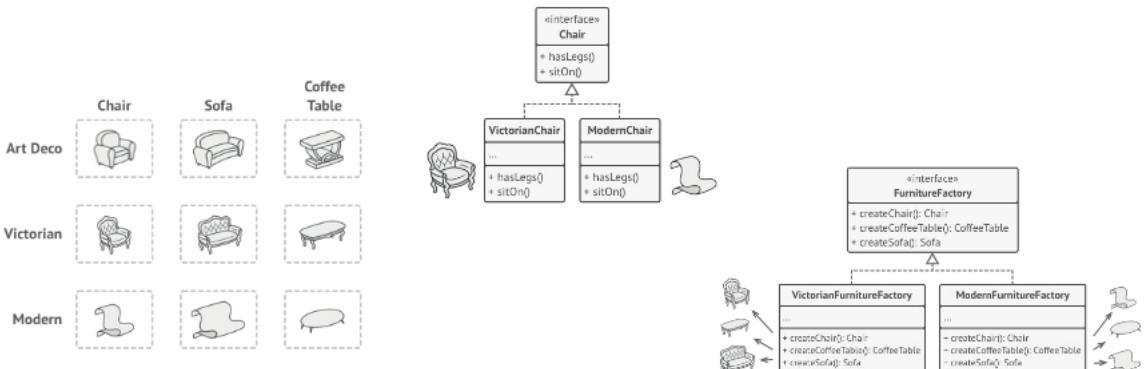
abstract class AbstractFactory {
    private static final EmberToolkit EMBER_TOOLKIT = new EmberToolkit();
    private static final EnginolaToolkit ENGINOLA_TOOLKIT = new EnginolaToolkit();

    // 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();
    }
}

```



4.3 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.

Dando uma representação complexa, cria uma de muitas representações possíveis.

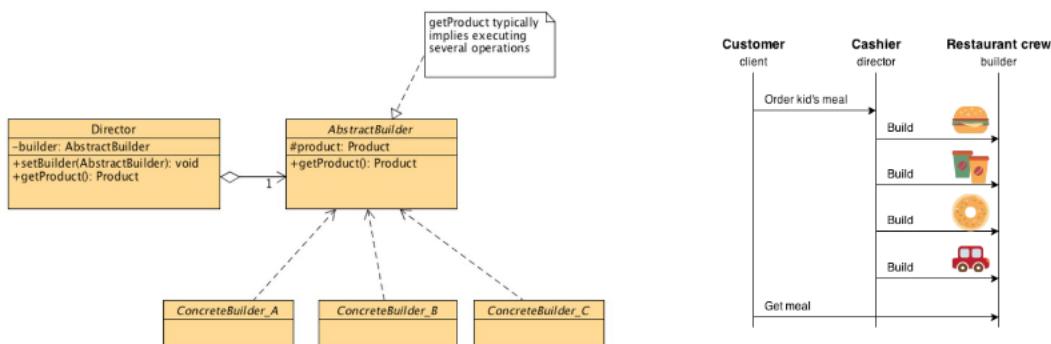
Problema - A construção de um objeto complexo geralmente implica um construtor com imensos parâmetros, alguns dos quais nem sempre necessitamos.

Uma aplicação precisa de criar elementos de um agregado complexo. A especificação para o agregado existe num armazenamento secundário e uma de várias representações precisam de ser contruidas no armazenamento primário.

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.

Estrutura:



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

Nota: 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()`.

Check List:

1. Verificar se o problema consiste em ter um input comum e vários outputs (representações) possíveis;
2. Encapsular o input numa classe Director;
3. Desenhar um protocolo para criar todas as representações possíveis (capturar os passos do protocolo num Builder interface);
4. Definir uma classe derivada do builder para cada representação;
5. O cliente cria um Director/Reader e um Builder, passando o último ao primeiro, pedindo de seguida ao Director que construa e devolva o produto;
6. O cliente pede ao Director para "construir";
7. O cliente pede ao Builder para retornar o resultado;

Exemplo:

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("pan 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

```

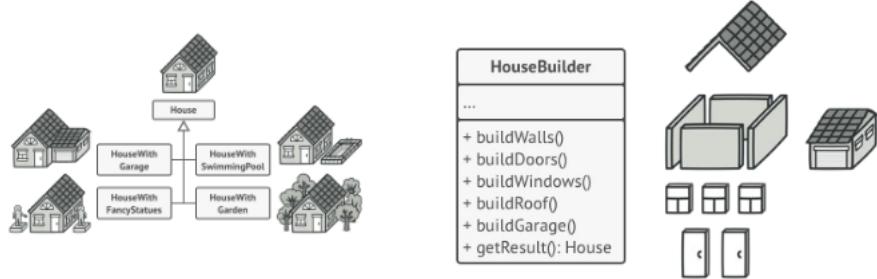
public class NutritionFacts { // Builder Pattern
    private final int servingsSize;
    private final int servings;
    private final int calories;
    private final int fat;
    private final int sodium;
    private final int carbohydrate;

    public static class Builder {
        // Required parameters
        private final int servingsSize;
        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 servingsSize(int val) {
            servingsSize = val;
            return this;
        }
        public Builder servings(int val) {
            servings = 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) {
        servingsSize = builder.servingsSize;
        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();

```



4.4 Object: Singleton

Intenção - Garantir que uma classe tem uma única instância e fornecer um ponto de acesso global à mesma.

Encapsulação "just-in-time initialization" ou "initialization on first use".

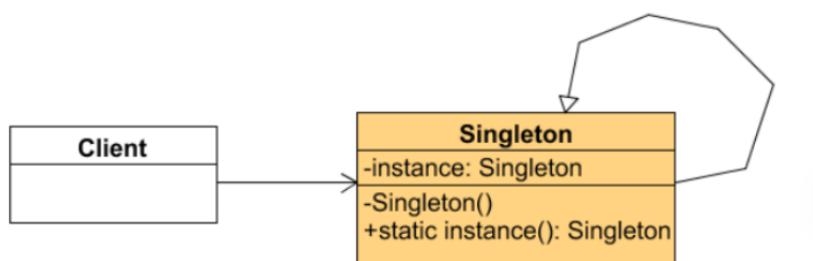
Problema - É impossível de implementar com um construtor, uma vez que este cria uma nova instância do objeto por defeito. A criação de um ponto de acesso global também é um desafio, tal como a lazy initialization.

Solução - Definir o construtor como privado (private Singleton(String name)).

Definir uma private static reference para um único objeto da classe (static private Singleton instance).

Definir um método que vá buscar a instância (static public Singleton getInstance()) (Os utilizadores apenas podem acessar o objeto Singleton através deste método)

Estrutura:



Check List:

1. Definir um atributo estático do seu tipo na classe;
2. 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.
3. Definir todos os contrutores como protected ou private;

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

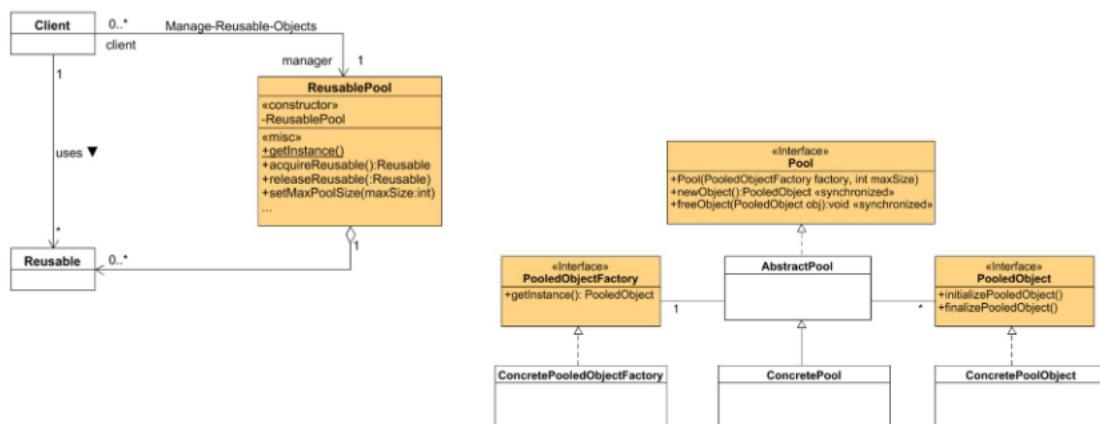
4.5 Object: Object pool

Intenção - Aumentar a eficiência na utilização de objetos com custos elevados de inicialização, que são utilizados frequentemente, cuja instanciação simultânea é reduzida. Por exemplo conexões de rede/BD, threads...

Problema - Um cliente com acesso à Object Pool evita a criação de novos objetos, basta pedir à Pool por um objeto que já foi previamente instanciado. 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.

Estrutura:



Check List:

1. Criar uma classe do tipo pool com uma coleção de pool objects;
2. 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;
    }

    /**
     * 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;
    }

    /**
     * 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;
            }
        }
    }
}
```

4.6 Object: Prototype

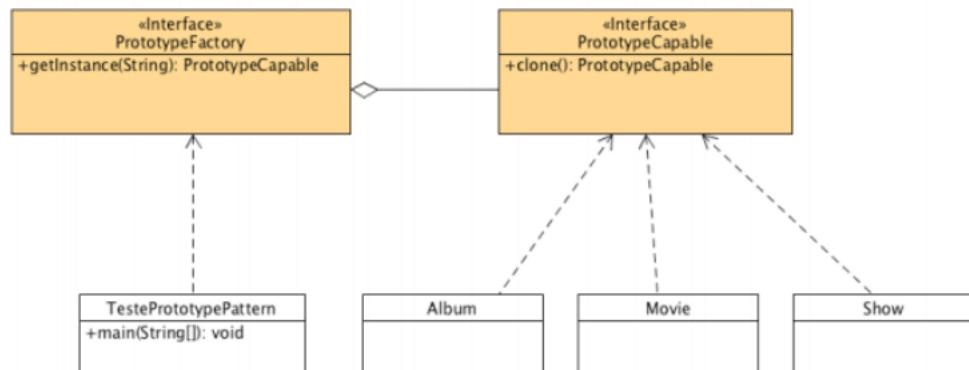
Intenção - Criar objetos novos a partir de existentes, como protótipos, sem estar dependente das suas classes.

O operador "new" é considerado perigoso.

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.

Estrutura:



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:

1. Adicionar um método `clone()` ao produto;
2. Desenhar um "registo" que mantém a cache dos objetos prototipáveis.
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;
3. O cliente substitui todas as referências ao operador "new" e substitui-las por invocações ao método de fábrica;

Exemplo:

1.

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

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 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();
    }
}
```

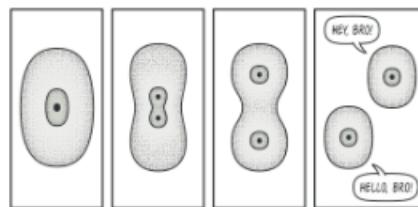
```
public class TestPrototypePattern {
    public static void main(String[] args) {
        try {
            PrototypeCapable proto;
            proto = PrototypeFactory.getInstance(ModelType.MOVIE);
            System.out.println(proto);

            proto = PrototypeFactory.getInstance(ModelType.ALBUM);
            System.out.println(album#prototype);

            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*.



4.7 Resumindo

Abstract Factory - Cria uma instância de várias famílias de classes;

Builder - Separa a construção do objeto da sua representação;

Factory Method - Cria uma instância de várias classes derivadas;

Singleton - Uma classe da qual apenas existe uma única instância;

Object Pool - Evita a aquisição e libertação cara de requisitos através de objetos reciclados que já não estão em uso;

Prototype - Uma inicialização da instância completa para ser copiada ou clonada;

5 Padrões Estruturais

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

5.1 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.

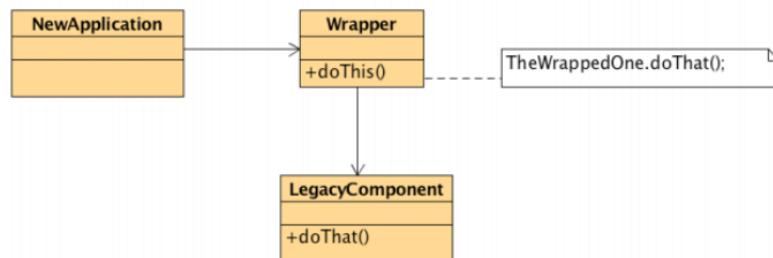
Problema - Quando surge a necessidade de integrar um componente externo ao nosso ecossistema e este já existe, podemos reutilizá-lo. No entanto, por vezes surgem problemas de compatibilidade.

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:

1. Fornecer uma interface, compatível com um dos objetos;
2. Este objeto acede aos métodos desta interface;
3. 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.

Estrutura:



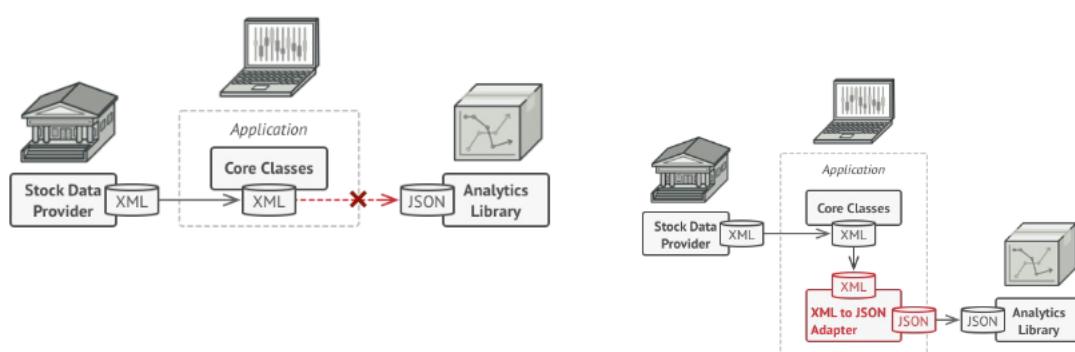
Check List:

1. Decide se a "independência da plataforma" e a serviços de criação são problemas;
2. Escreve uma matriz "plataformas" versus "produtos";
3. Define uma Interface fábrica que consiste num Factory Method por produto;
4. Define uma classe derivada da fábrica para cada plataforma que encapsula todas as referências ao operador "new";
5. O cliente não deve utilizar nenhum "new", e usar Factory Methods para criar os objetos de cada produto;

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).



5.1.1 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());  
    }  
}
```

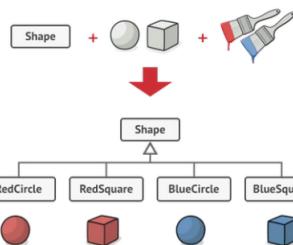
❖ Object adapter adapts via delegation:
– it forwards work to delegate

```
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(); }  
    // ...  
}
```

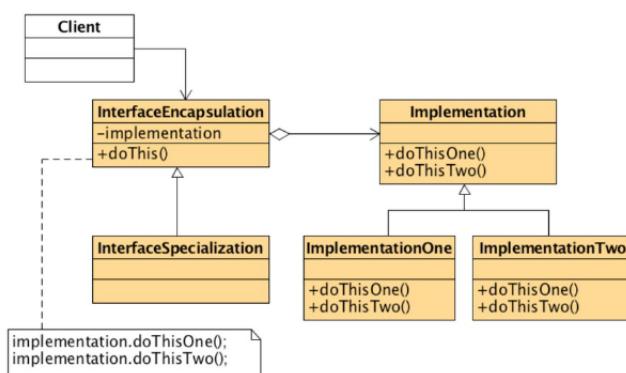
5.2 Object: Bridge

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.

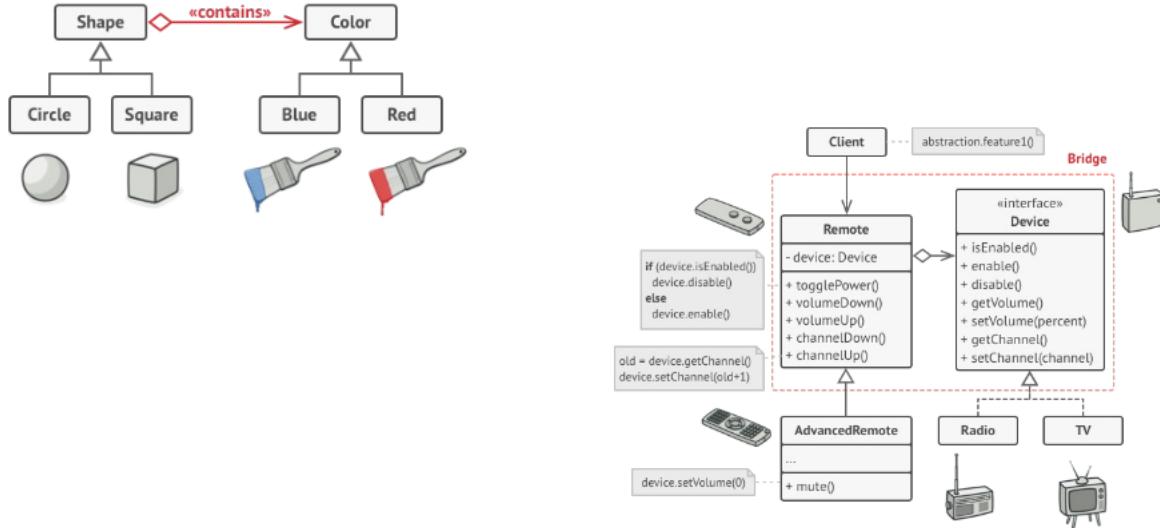


Check List:

Depois de verificar que existem duas dimensões variáveis no problema:

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

Exemplos:



5.3 Object: Composite

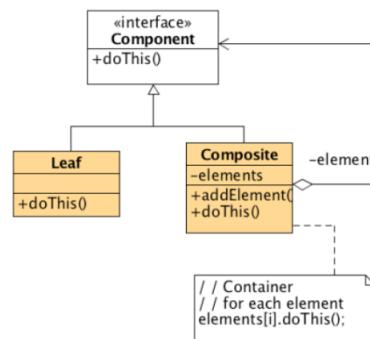
Intenção - Ter componentes de produtos mas de forma a que um componente em si seja também um produto, ou seja, compor objetos em estruturas de árvore e utilizá-las como se fossem objetos (composição recursiva).

"Diretórios possuem entradas, cada uma destas pode ser um diretório".

1-to-many "has a" up the "is a" hierarchy.

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.

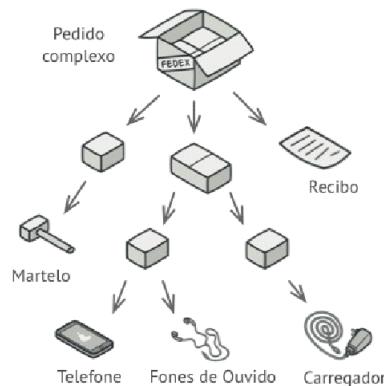


Check List:

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

1. Considerar a heurística “caixas que contém produtos podem ser um produto em si”;
2. Criar uma interface que torne os produtos e as caixas intercambiáveis, 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;
3. Os métodos para adicionar e remover produtos da caixa devem ser declarados na classe da caixa (não na interface!);

Exemplo: 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)



5.4 Object: Decorator

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

Client-specified embellishment of a core object by recursively wrapping it.

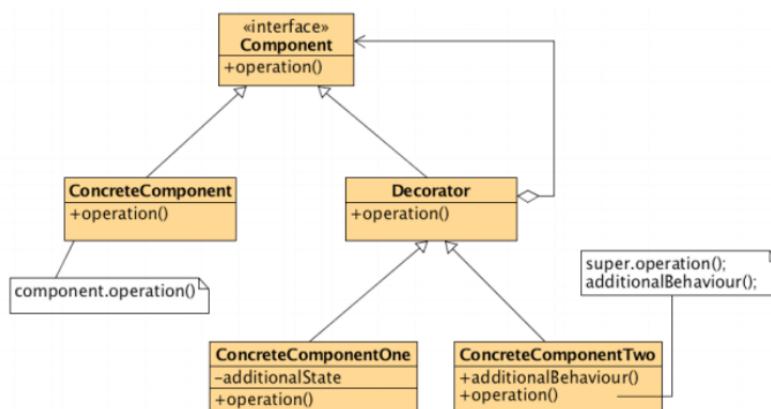
Wrapping a gift, putting it in a box, and wrapping the box.

Problema - Para concretizar a intenção, uma opção é a herança. No entanto, se considerarmos várias funcionalidades que podem ser combinadas entre si (ou não), estariamos a falar de demadiadas 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:

1. Criar uma interface que torne todas as classes intercambiáveis (estilo composite);
2. Criar uma subclasse decorator, que suporte wrappers adicionais.
Este têm como atributo um elemento da interface.
Para cada componente opcional criar uma classe derivada;
3. 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 nome;
    Jogador(String n) { nome = n; }
    @Override public void joga()
        { System.out.print("\n"+nome+" 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(j2);
        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();
    }
}
  
```

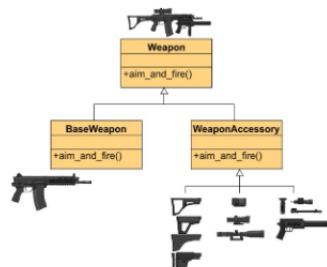
```

class Futebolista extends JogDecorator {
    Futebolista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("futebol "); }
    public void remata() { System.out.println("-- Remata"); }
}

class Xadrezista extends JogDecorator {
    Xadrezista(JogadorInterface j) { super(j); }
    @Override public void joga() { j.joga(); System.out.print("xadrez "); }
}

class Tenista extends JogDecorator {
    Tenista(JogadorInterface j) { super(j); }
    @Override public void joga()
        { j.joga(); System.out.print("tenis "); }
    public void serve() { System.out.println("-- Serve!"); }
}
  
```

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).

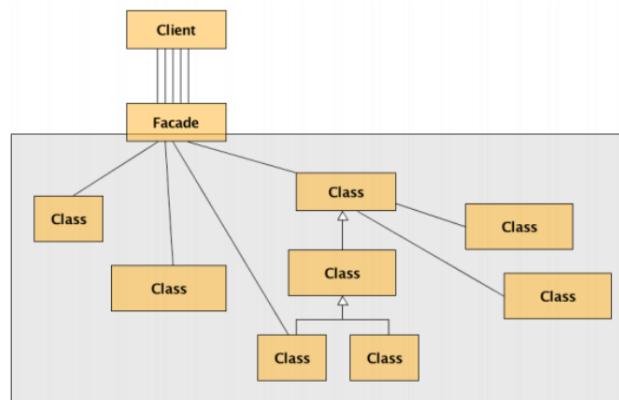
5.5 Object: Façade

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

Envolver um subsistema complicado com uma interface simples.

Problema - Complexidade do sistema.

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



Check List:

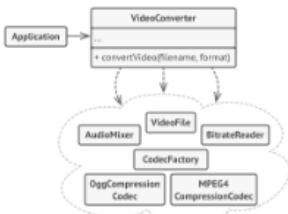
1. Identificar uma interface mais simples e unificada para o sistema (Só esta vai ser utilizada pelo cliente!);
2. Desenhar um wrapper que encapsule todos os subsistemas.
Responsável por delegar os métodos apropriados.
Considerar se dentro dos subsistemas acrescentaria valor a criação de outras classes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

Benefícios:

1. Ao esconder a implementação do cliente podemos alterar os subsistemas sem este se aperceber, promovendo um baixo acoplamento e reduz as dependências.
2. Não se pode dizer no entanto que adicione alguma funcionalidade ou que previna os clientes de aceder a determinados métodos.

Exemplos:

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 bilhão 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
        converter = new VideoConverter()
        mp4 = converter.convert("Funny-cats-video.ogg", "mp4")
        mp4.show()
}

```

2.

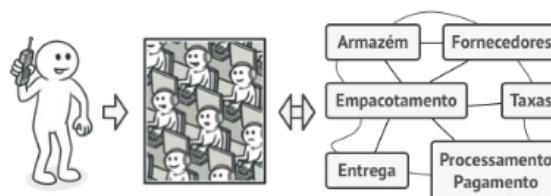
```

// ...
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.

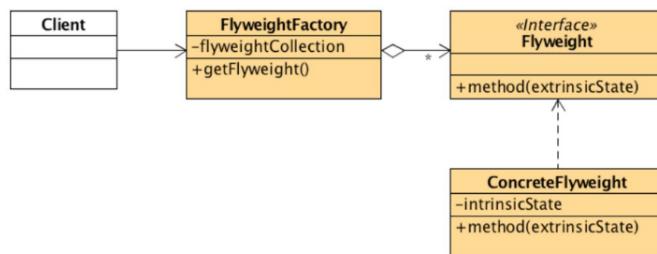


5.6 Object: Flyweight

Intenção - Utilizar a partilha de forma a suportar muitos objetos compactos de forma eficiente. A estratégia Motif GUI, que substitui heavy-weight widgets por light-weight gadgets.

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.



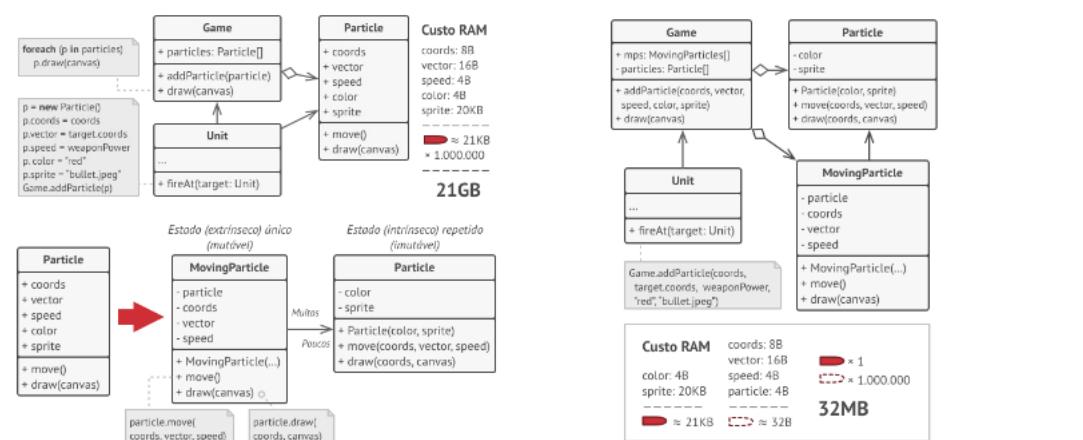
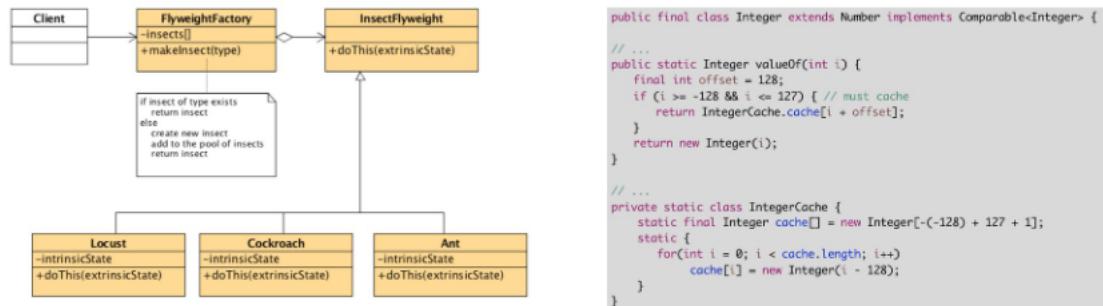
Podemos ainda associar-lhes o padrão object pool.

Check List:

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

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

Exemplo:



```

// 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 uni-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)

```

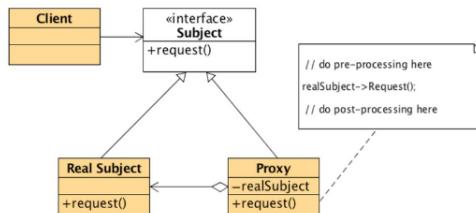
5.7 Object: Proxy

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

Usar um nível extra de indirection para suportar acesso distribuído, controlado ou inteligente. Adicionar um wrapper e uma delegation para proteger o componente verdadeiro da complexidade.

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.

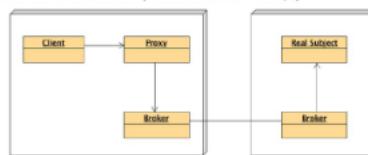
Check List:

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

Exemplo:

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.)

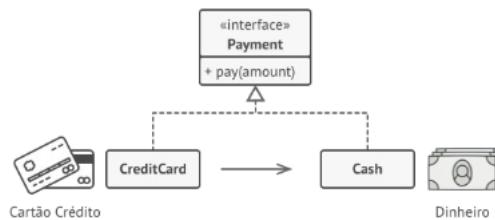
Known Uses: Secure Objects

- ❖ Different clients have different levels of access privileges to an object
- ❖ Clients access the object through a proxy
- ❖ The proxy either allows or rejects a method call depending on what method is being called and who is calling it (i.e., the client's identity)

Known Uses: Copy-on-Write

- ❖ Multiple clients share the same object as long as nobody tries to change it
- ❖ When a client attempts to change the object, they get their own private copy of the object
- ❖ Read-only clients continue to share the original object, while writers get their own copies
- ❖ Allows resource sharing, while making it look like everyone has their own object
- ❖ When a write operation occurs, a proxy makes a private copy of the object on-the-fly to insulate other clients from the changes

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.



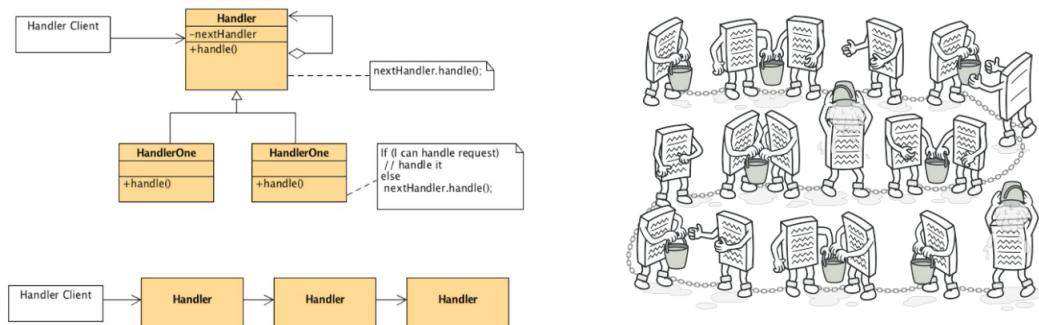
6 Padrões de Comportamentais

6.1 Class: Chain of Responsibility

Intenção - Cria uma corrente de objetos que passam os pedidos através da mesma até que um deles 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:

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

```

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<>();
        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);
        }
    }
}

```

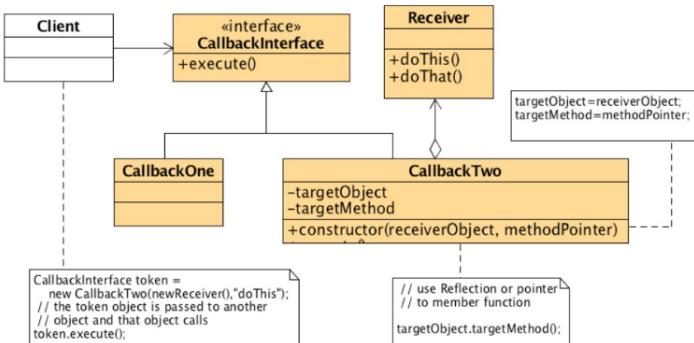
A text parser for: someFile.txt
A JSON parser for: otherFile.json
A CSV parser for: csvFile.csv
No parser for the file: somethingelse.doc

6.2 Class: Command

Intenção - Encapsular um pedido como 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:

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

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(); // get the object's "Class"  
            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 cmd = new CommandReflect();  
    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

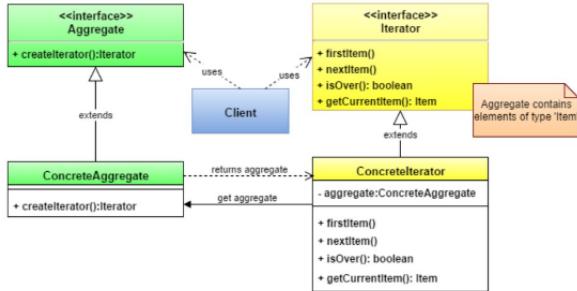
```
public class CommandFactory {  
    private Map<String, Command> commands;  
    private CommandFactory()  
    { commands = new HashMap<String, Command>(); }  
    public void addCommand( String name, Command command )  
    { commands.put( name, command ); }  
    public void executeCommand( String 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 lambda  
        // 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, Light off, Sound off  
Light turned on  
Sound turned on  
Light turned off
```

6.3 Class: 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:

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

```

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

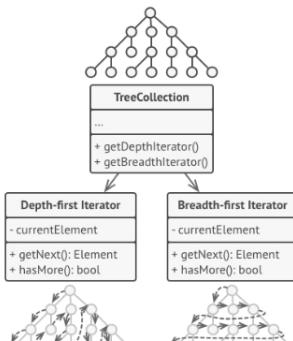
    @SuppressWarnings("unchecked")
    public VectorGeneric() {
        vec = (T[]) new Object[dimVec];
        nElem = 0;
    }

    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;
}
  
```

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();
}

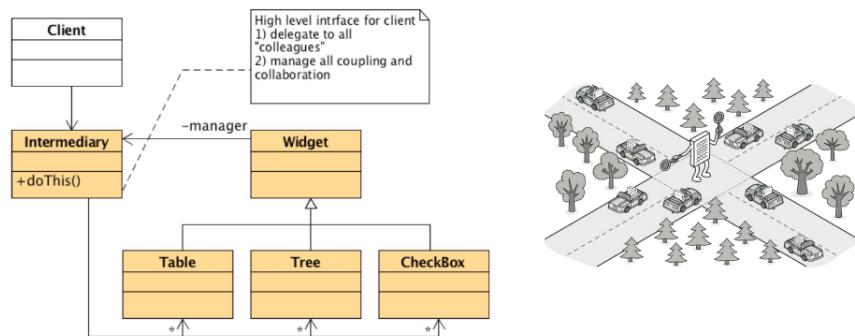
// For both the keys and values of a map
for (Iterator it=map.entrySet().iterator(); it.hasNext(); ) {
    Map.Entry entry = (Map.Entry)it.next();
    Object key = entry.getKey();
    Object value = entry.getValue();
}
  
```

6.4 Class: Mediator

Intenção - Reduzir 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.

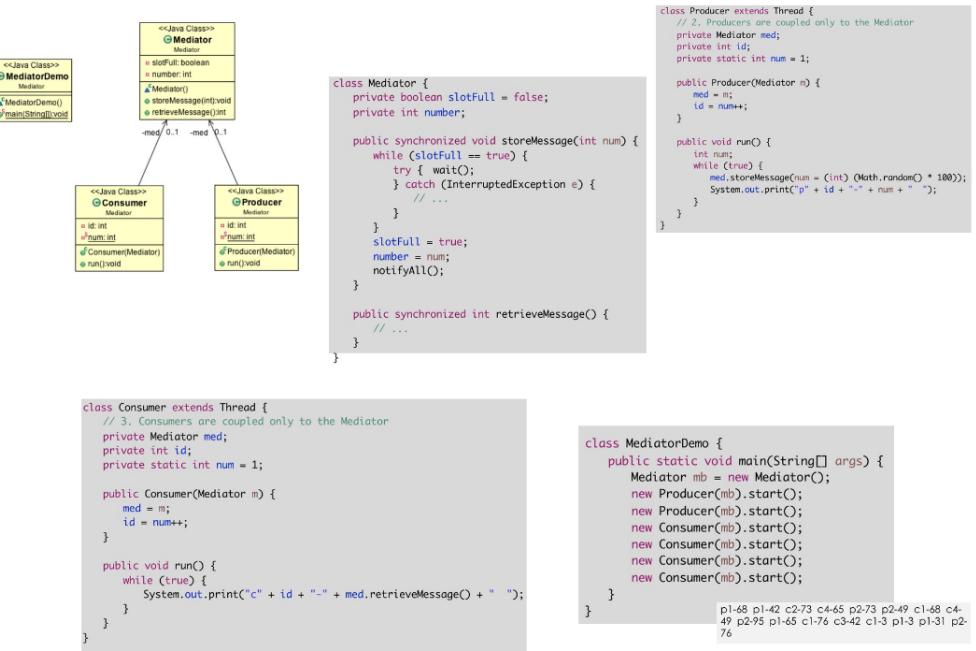
Solução - Colaboração indireta, através de um **mediador**.



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.

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



6.5 Class: Memento