-> O que é Proxy
  - wrapper à volta
  - padrão estrutural
  - um intermediário que permite fazer alguma coisa antes ou depois duma requisição

-> exemplo do cartão de crédito

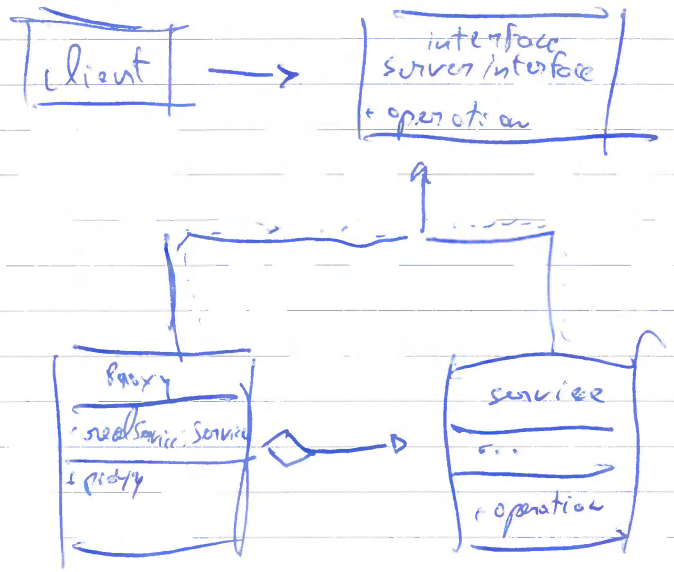
-> O proxy funciona mesmo que o objeto que ele mascareia esteja off

-> A class proxy implementa a mesma interface da classe original

-> Permite controlar o acesso a um objeto/serviço.

-> Podemos associar mais que um proxy ao mesmo objeto

-> Podemos controlar o objeto/serviço sem o acessar diretamente, porém aumentamos a complexidade do código.



Diogo Martins, 108548
Diogo Guedes, 114256
Guilherme Amaral, 113207
Francisca Sales, 112841
Daniel Amorim, 112610
Henrique Freitas, 114990
Gabriel Santos, 113682
Afonso Baixo, 108237
Luís Leal, 103511

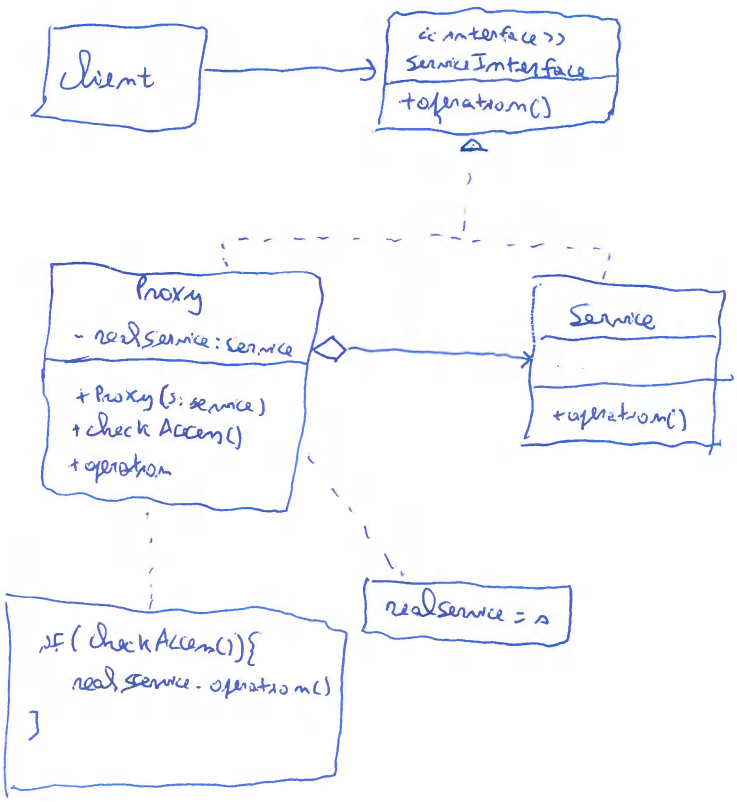
Proxy Design Pattern:

O Proxy é um padrão estrutural, que media o acesso a um objeto e permite a adição de lógica antes e/ou depois do acesso ao objeto original, mas mantendo a mesma interface. Como tal, é possível a adição de lógica extra sem necessidade de alterar a classe inicial, permitindo o princípio open code.

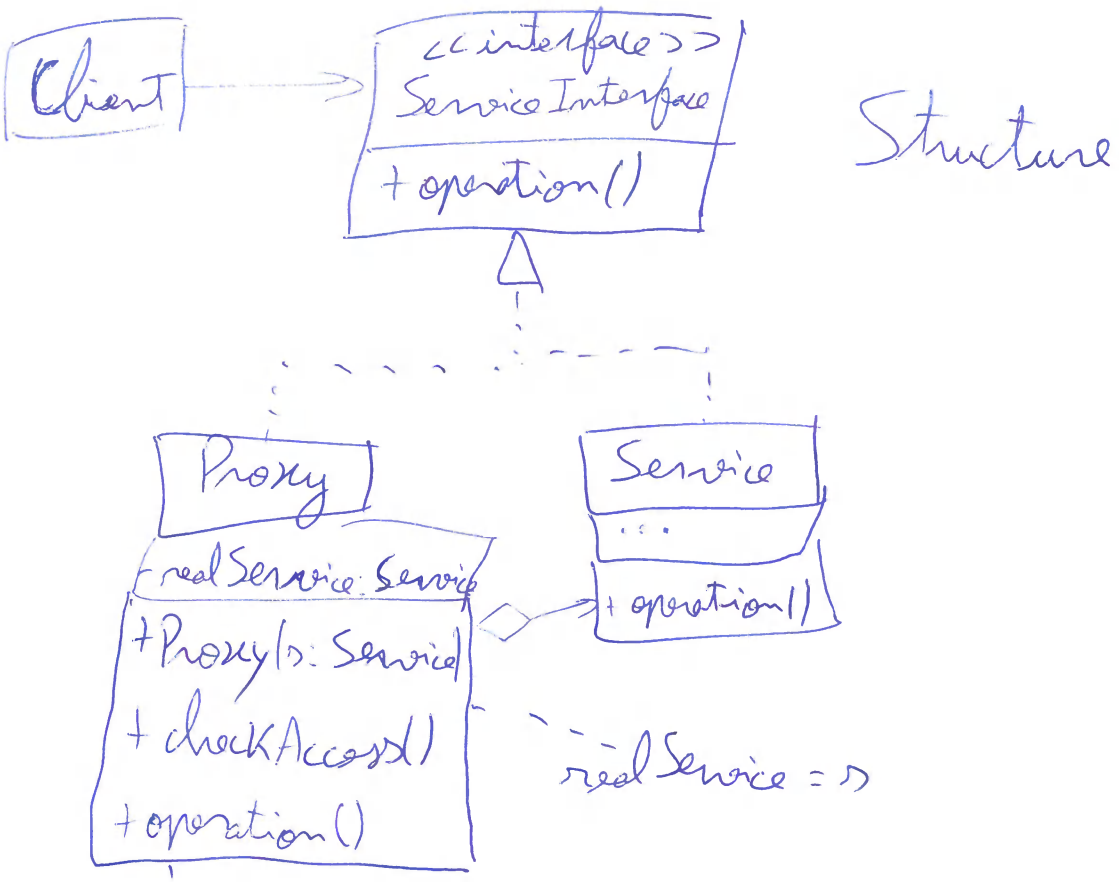
O padrão é constituído pela interface do serviço original, e duas classes que a implementam. O client irá interagir com a classe Proxy, que por sua vez irá interagir com o serviço original.

A interface original declara todas as operações possíveis de realizar. A classe Proxy irá implementar esta interface de modo a mascarar a existência da lógica extra. A classe Proxy possui, por composição, uma instância da classe serviço e recebe as diversas pedidas para o serviço original. Como tal permite, não só lazy-loading do serviço, mas também, controlo de acesso e caching. O cliente irá pela interface do serviço, interagir sempre com a classe Proxy.

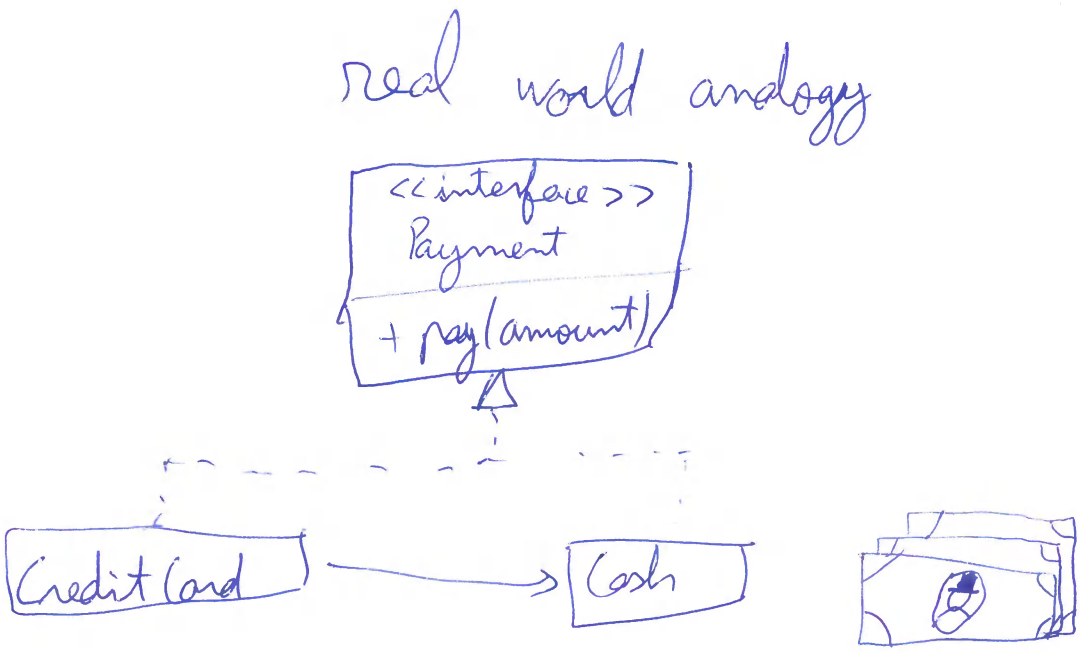
Este padrão pode ser útil quando temos acesso concorrente a uma mesma estrutura de dados, ou então um objeto pesado que pretendemos carregar de modo lazy.



Elementos do grupo:
→ Tiago Albuquerque; Nº 112901
→ Luís Godinho; Nº 112959
→ Filipe Sousa; Nº 114196
→ Abel Teixeira; Nº 113655
→ Hugo Sousa; Nº 112733
→ João Capucho; Nº 113713
→ José Marques; Nº 114321
→ Igor Coelho; Nº 113532
→ Zakhar Kruptsala; Nº 114478
→ Tiago Lopes; Nº 113586




```
if(checkAccess()){
    realService.operation()
}
```



Groupe 6

115243, 113402, 112665, 113480, 115178, 98059, 113626, 103070

Proxy:

Objetivo - Fornece um objeto placeholder, uma "dupla" para o objeto verdadeiro, controlando o acesso a este. Isto é necessário por exemplo quando um cliente usa a maioria dos recursos do sistema, mas nem sempre é necessário estar ligado a este.

Assim podemos criar uma classe "Proxy" que vai servir como gatekeeper para o serviço. Um exemplo concreto é o realizar de um pagamento. ~~Podemos usar dinheiro ou um cartão para realizar o pagamento e a interface Payment vai regular o acesso à conta bancária~~

Quando efetuamos um pagamento com um cartão de crédito, a interface Payment vai ~~regular o acesso ao dinheiro~~ chamar o método de pagamento do proxy (Credit card), que regula o acesso ao dinheiro. O cartão tem de verificar se a conta está congelada ou não e apenas devolver o dinheiro se tivermos acesso a este.

Aplicações:

1. Restringir o acesso a um serviço para quando realmente precisamos deste
2. Restringir o acesso a um serviço para quando o cliente necessita de certas caracteristicas
3. Execução local de um serviço remoto (o proxy é que lida com os métodos do serviço)
4. Registo de acessos ao serviço: o proxy pode manter um registo de quem acedeu ao serviço

Prós:

Abstração do serviço do POV do cliente
Gestão do serviço sem necessidade de intervenção do cliente
Funciona mesmo quando o serviço não funciona

Grupo 6

115243, 113402, 112665, 113480, 115178, 98059, 113628, 103070

11

113717
112994
98543
107876
98584

Proxy

Fornece um substituto para outro objeto. O proxy controla o acesso ao objeto original, permitindo fazer outras coisas antes ou depois do pedido ter chegado ao objeto original

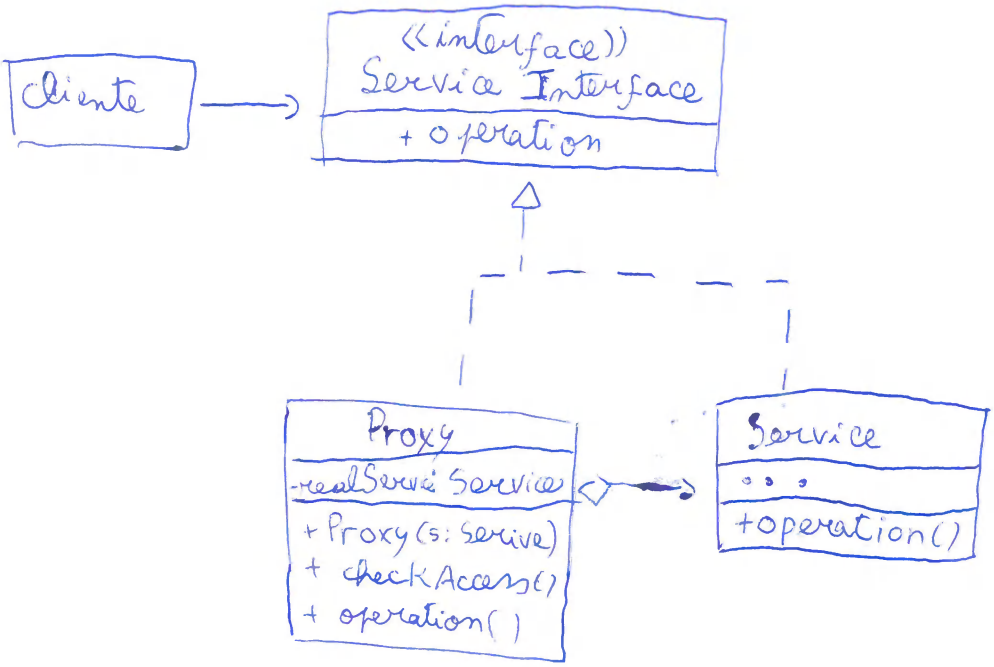
Surge um problema, porquê querer controlar o acesso a um objeto? Vejamos o exemplo seguinte:

- Temos um objeto que consome uma vasta quantidade de recursos de um sistema em questão. Este objeto é preciso de vez em quando, mas não constantemente. Poderíamos implementar inicialização "preguiçosa", por outras palavras, criar este objeto apenas quando ele for necessário. Talvez funcionasse, mas isto causa bastante repetição desnecessária de código, o que teria um impacto negativo na performance do nosso sistema.

- Este padrão sugere que seja criada uma nova classe com a mesma interface de um objeto do serviço original. Posteriormente, a aplicação deve ser atualizada, de forma a que ela passe o objeto proxy para todos os clientes do objeto original. Ao receber a solicitação de um cliente, o proxy cria um objeto de serviço real e transfere-lhe toda a informação.

Diagrama

Tem que funcionar com objetos e proxys através da mesma interface



Pros

1. É possível manipular o serviço sem o cliente perceber
2. Segue o princípio Open / Closed, pois é possível criar vários proxys sem modificar o código do cliente ou do serviço

Contras

1. O código fica mais complexo
2. A resposta do serviço pode atrasar-se

113403
114137
113613
113736
114622
114246
113780

113585
104152
114514
103592
103730
113475

Facade Pattern

O padrão Facade (ou Fachada) é um padrão de design estrutural utilizado na programação orientada a objetos.

Tem o objetivo de fornecer uma interface simplificada para um conjunto complexo de classes, uma biblioteca de software ou qualquer outro sistema complexo.

Este padrão tem como propósito reduzir a complexidade de interação com subsistemas complexos, fornecendo uma interface única e simplificada. Isto ajuda a interação entre o utilizador e o sistema sem necessidade de entender as interdependências entre os seus componentes. Pretende também desacopopler o código do cliente das complexidades do subsistema.

Uma única classe que representa todo o subsistema e oferece uma interface simples para o cliente. Esta direciona as solicitações do cliente para os componentes apropriados dentro do subsistema.

Vantagens:

- Simplicidade: facilita a interação com o sistema, pois o cliente lida com uma única interface simples ao invés de múltiplos interfaces
- Redução de dependências: isola o código do cliente das complexidades do subsistema, facilitando a manutenção e evolução do sistema

Desvantagem:

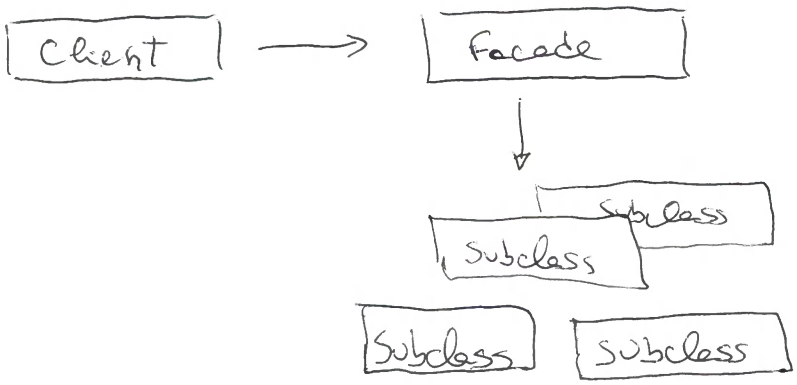
- É uma classe que acaba por se tornar um ponto centralizado para todas as funcionalidades, tornando-a complexa e pesada se não for bem projetada.

Exemplo prático

Um sistema de Home theater que inclui componentes como DvD player, projetor, etc. Em vez de interagir com cada um desses componentes individualmente, podemos ter uma classe Facade, chamada HomeTheaterFacade, que oferece métodos como watchMovie() e endMovie(). Estes métodos geram as opções necessárias nos componentes do sistema de home theater simplificando a utilização para o utilizador final

Estrutura

Temos uma Facade que oferece acesso a partes das funcionalidades do subsistema. Sabe como redirecionar um request do cliente e como operar todas as partes. Temos uma Facade adicional para prevenir poluir a Facade principal com funcionalidades desnecessárias que possam torna-la noutra estrutura complexa. O sistema complexo não sabe da existência das Facades. Ora, o cliente utiliza a Facade em vez de chamar os objetos do subsistema.



10

Façade

Classe FacadeHome → Fornece uma interface simplificada para interagir com o sistema Home

→ Encapsula a complexidade das interações entre os diversos componentes do sistema como a cozinha, cinema, sala (todos estes componentes seriam Facades).

classe AdicionalFacade

→ Cozinha, Cinema, sala

Complex Subsystem

→ ~~Consiste nos diversos componentes do sistema Home como a cozinha, o cinema~~

~~Consiste em cada componente da Home~~

~~Cozinha~~

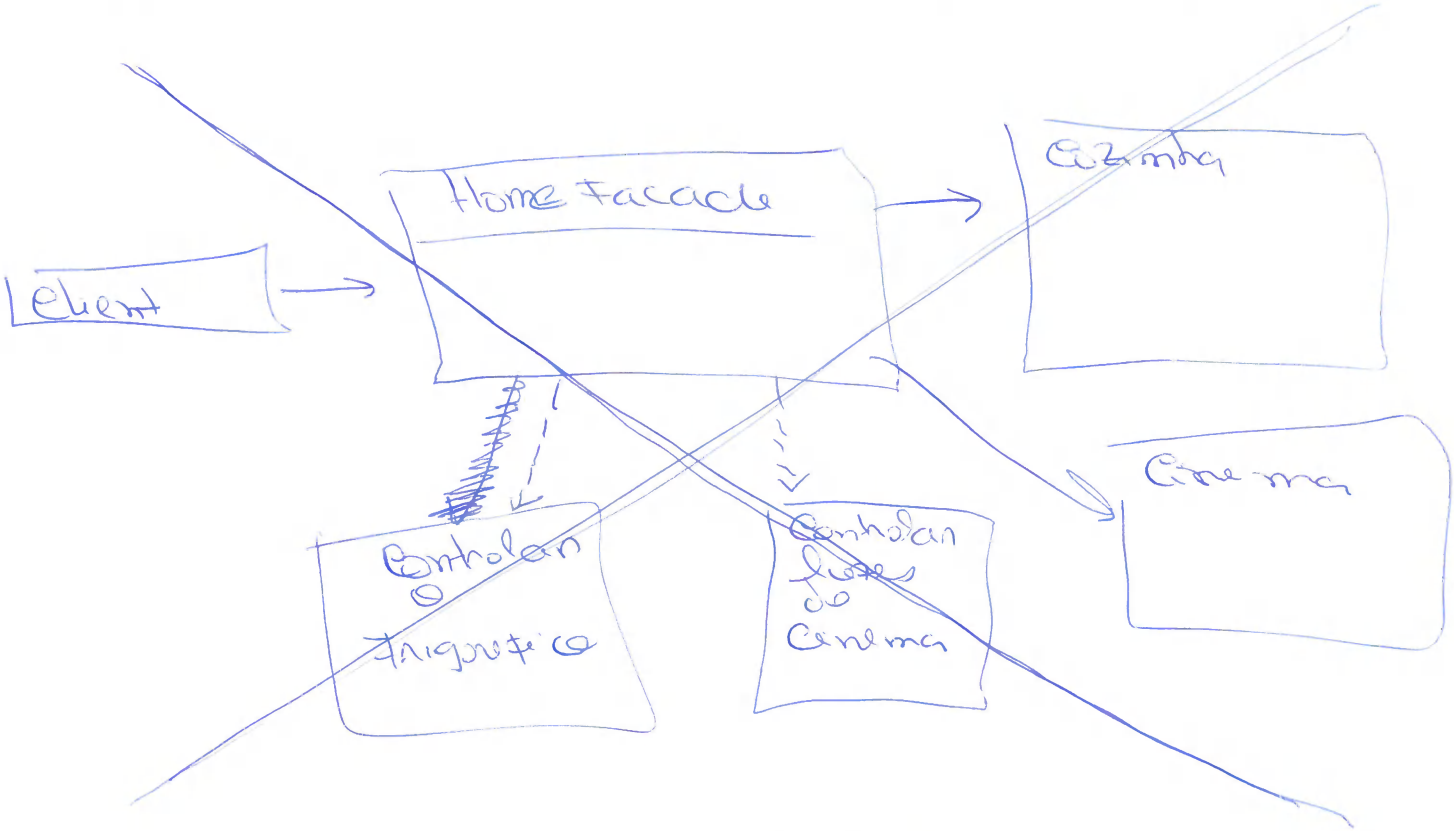
Os subsistemas complexos são os subsistemas constituintes de cada subfacade

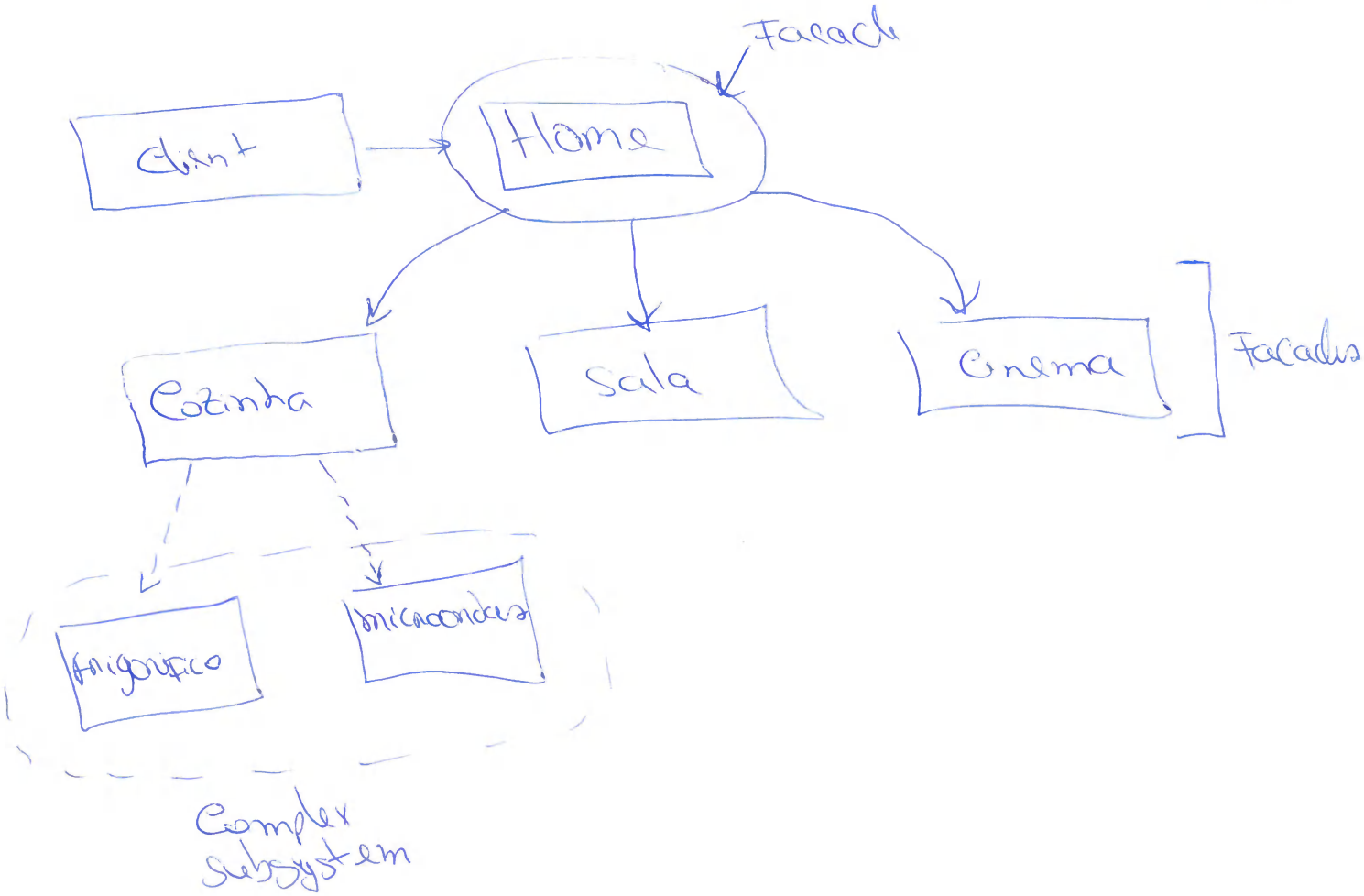
Ex: controlam o Frigorífico (temperatura) da SubFacade cozinha

Client → Que usa a Facade Home

Em vez de interagir com cada componente interage com a Facade Home

ex: Aplicação


Client
Home Facade
Cozinha
Cinema
Controlar frigorífico
Controlar luzes do Cinema


Facade
Client
Home
Cozinha
Sala
Cinema
Facades
Frigorífico
Microondas
Complex subsystem

Facade

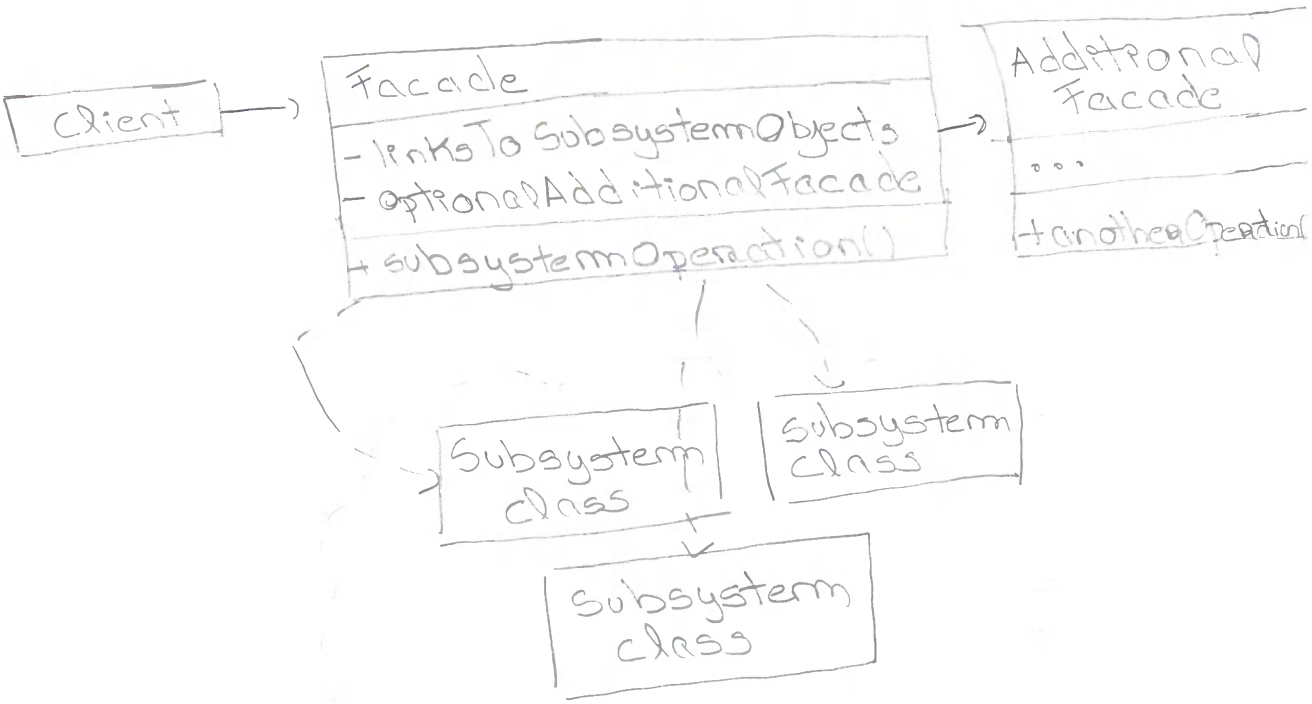
Problema:

→ Temos um sistema muito complexo, com várias interfaces diferentes e diversas dependências, tornando difícil lidar com este e geri-lo.

Solução:

→ Produzir uma interface intermediária simplificada e bem definida para usar as suas funcionalidades enquanto oculta a complexidade dos detalhes de implementação

Estrutura:



Exemplo: Quando ligamos a um callCenter, este reenvia-nos para o serviço específico que necessitamos.

Vantagens e desvantagens:

→ A Facade está coupled a todas as subclasses, se houver um problema com a Facade, todas as outras tornam-se inacessíveis (desvantagem)

→ Isola o código da complexidade do subsistema (vantagem)

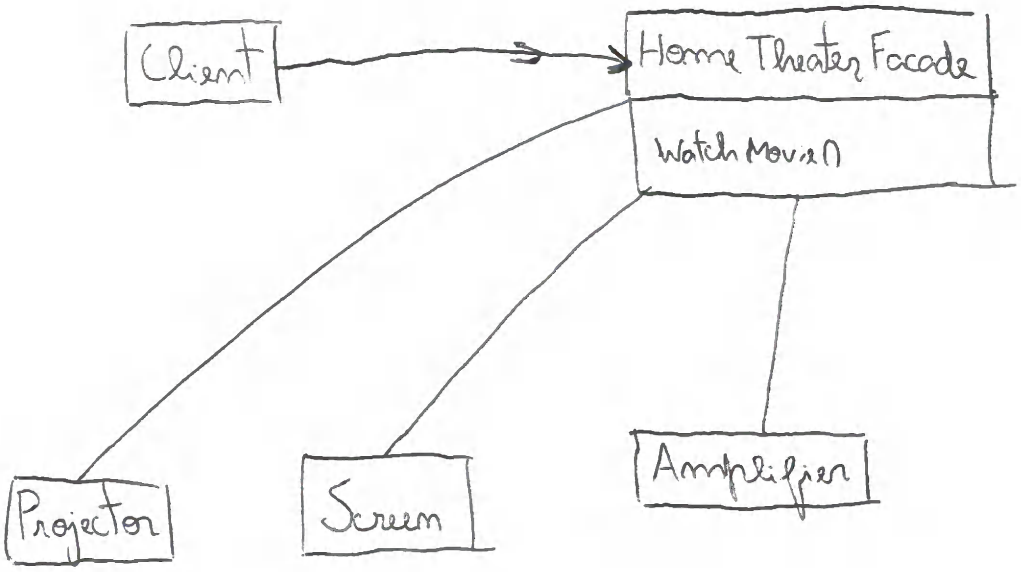
Facade

O padrão de design Facade é usado para simplificar a interação com sistemas complexos, oferecendo uma interface simples para operações complicadas. É um padrão estrutural que encapsula um conjunto de interfaces de um subsistema em uma única interface mais fácil de usar, reduzindo a complexidade e melhorando a modularidade.

Problemas e Soluções

- A complexidade de um sistema pode ser simplificada já que o padrão implementa o princípio do *least knowledge* criando uma interface mais simples que irá utilizar para interagir com o resto do sistema.
- O Alto Acoplamento e o elevado número de dependências também pode ser resolvido por este padrão já que como a implementação é escondida do cliente, podemos alterar os subsistemas à vontade já que a interface será a mesma.

Exemplo:
Por exemplo, o sistema de home theater, o uso do padrão Facade simplifica significativamente a experiência do usuário.
Sem o Facade, o utilizador deve aprender e gerenciar cada componente separadamente para assistir a um filme, o que é complicado.
Com o Facade:
Uma classe HomeTheaterFacade oferece um metodo simples como watchMovie(), que configura todos os dispositivos com as configurações adequadas


Client
Home Theater Facade
Watch Movie()
Projector
Screen
Amplifier

- Flyweight

→ O que é?

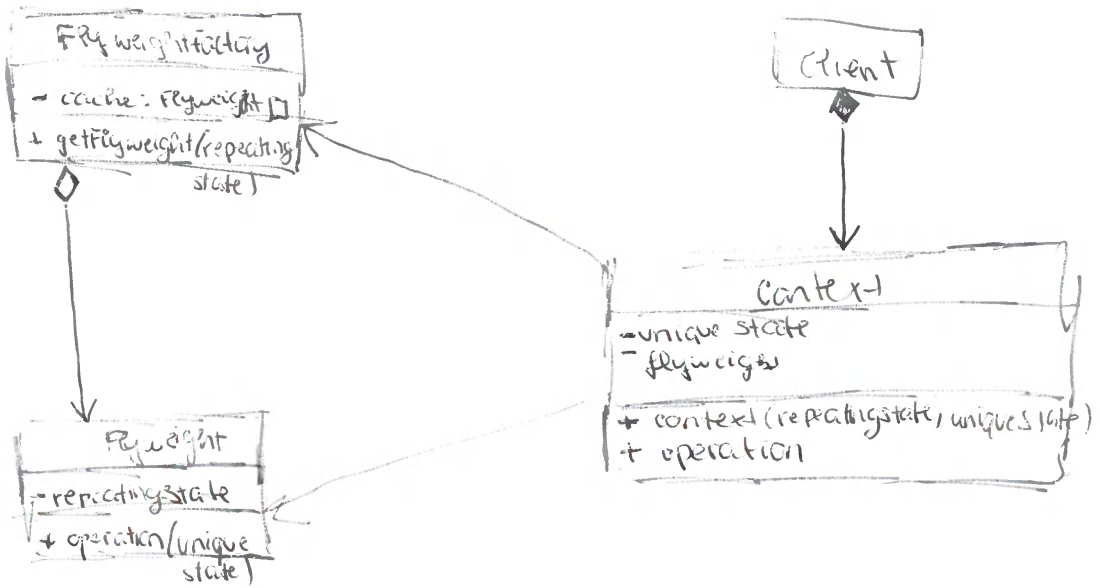
↳ Padrão Estrutural

→ usado para otimizar o uso de memória e desempenho em sistemas que lidam com muitos objetos de estado semelhante

→ Ideia central:

↳ Reduzir o número de objetos que precisam de ser criados, compartilhando o máximo possível de estados entre eles.

→ Estrutura:



→ Exemplo:

- Processamento de caracteres num processador de texto.
  ↳ cada caracter pode ser representado como um Flyweight com o seu estilo de fonte e tamanho como estado compartilhado, enquanto que a posição pode ser passada como uniqueState.

→ Vantagens:

- Poupar memória RAM

→ Desvantagens:

- Troca-se RAM por ciclos de CPU quando a informação tem de ser recalculada, cada vez que se chama o método flyweight
- Código muito mais complexo.

Grupo 8

113534

113920

113968

104429

107548

114614

114588

114624

113705

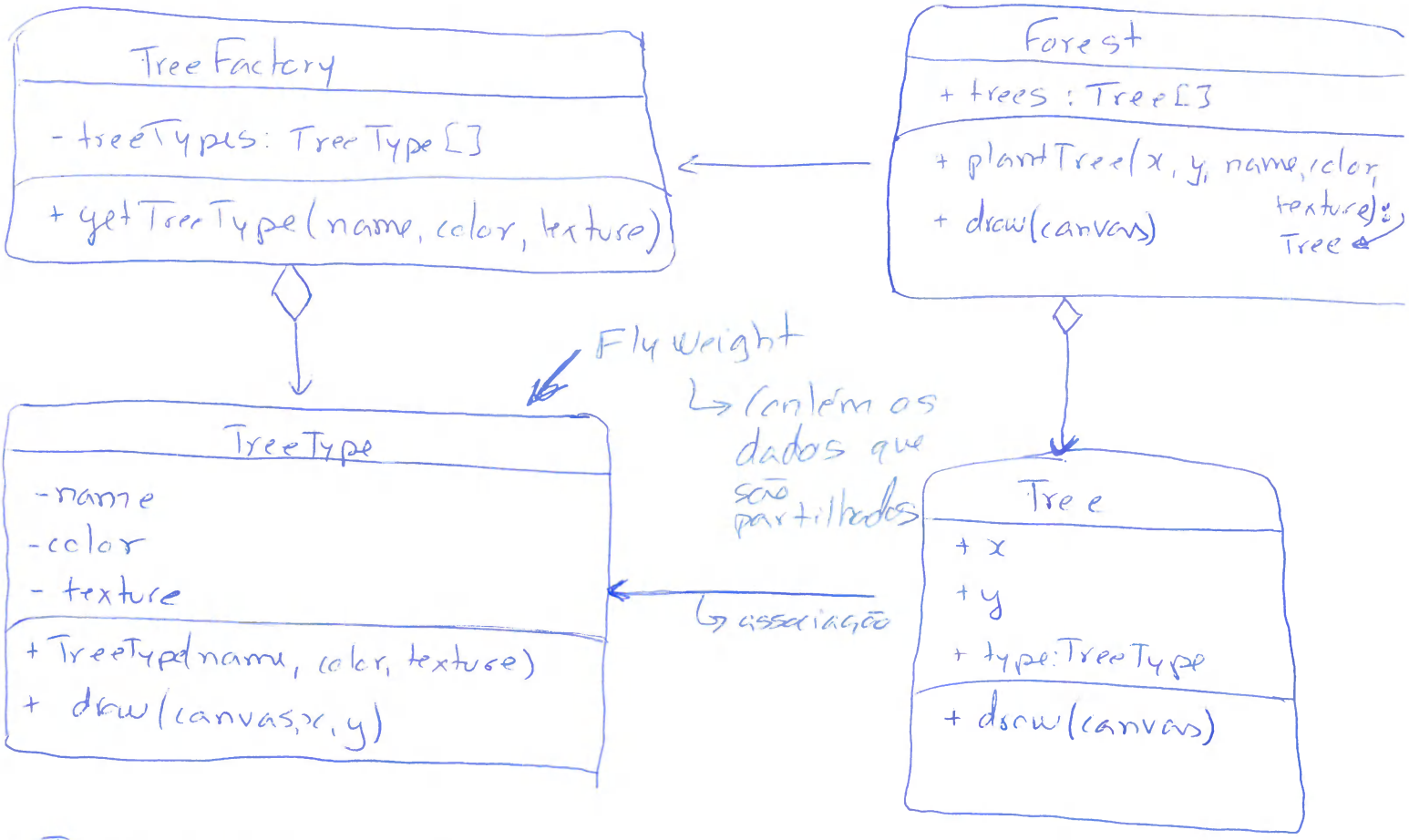
114640

103415

113786

Exemplo prático

115304	113278
113384	115697
112981	113637
113944	110150
114547	



→ Temos um jogo onde queremos renderizar muitas árvores, cada uma com um tipo.

→ Cada tipo partilha características comuns: nome, cor, textura

→ A solução consiste em:

- Na vez de renderizarmos todos os objetos árvore como um tipo, sendo que todas as características desse tipo são guardadas em cada objeto, criar um TypeTree que possuirá os atributos e, a cada árvore, atribuís um tipo.
- Desta forma, teremos atributos partilhados por todas as árvores do mesmo tipo

Atributos "intrinsic state": name, color, texture → esta é a diferença entre vários grupos de árvore

Atributos "extrinsic state": x, y & type → esta é a diferença de árvore para

Flyweight

CPU cycles

Repeated

- Intro

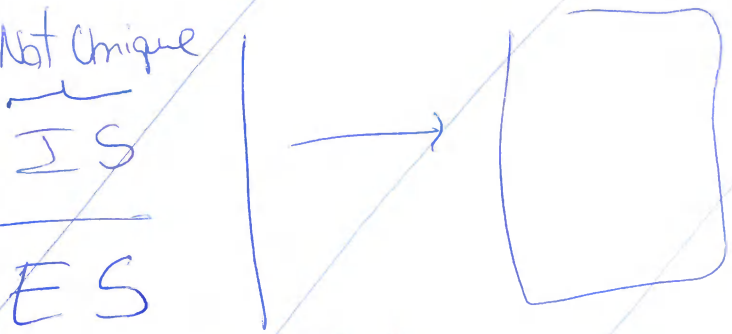
"Quote"

Only

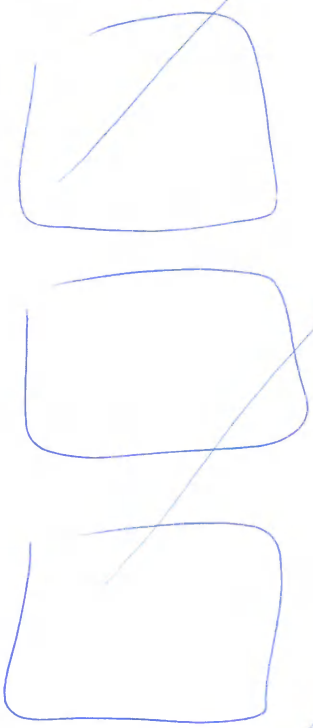
Not Unique

IS

ES



Type



Intrisic

Extrinsic

Flyweight

115304	115697
113893	114547
104384	113278
110150	112981
115697	113384
	113744

- Padrão de design estrutural
- Optimização do uso da memória RAM
- Reduz a repetição (em memória) de componentes de objetos
- Separação dos entidades com atributos únicos e partilhados

(Intrinsic vs Extrinsic)
(shared) (unique)

- Factory que vai ajudar na criação dos tipos que armazenam os estados Intrinsicos.

```
if ( cache[repeatingState] == null ) {
    cache[repeatingState] = new (...)
}
return cache[repeatingSize];
```

↳ Na factory

↳ Ao criar um novo objeto, observa os atributos pertencentes ao estado intrinseco e verifica se já existe algum tipo com eles. Caso exista, atribui ao objeto esse tipo (i.e., as suas caracteristicas). Caso não existe, cria um novo tipo e atribuiu ao objeto

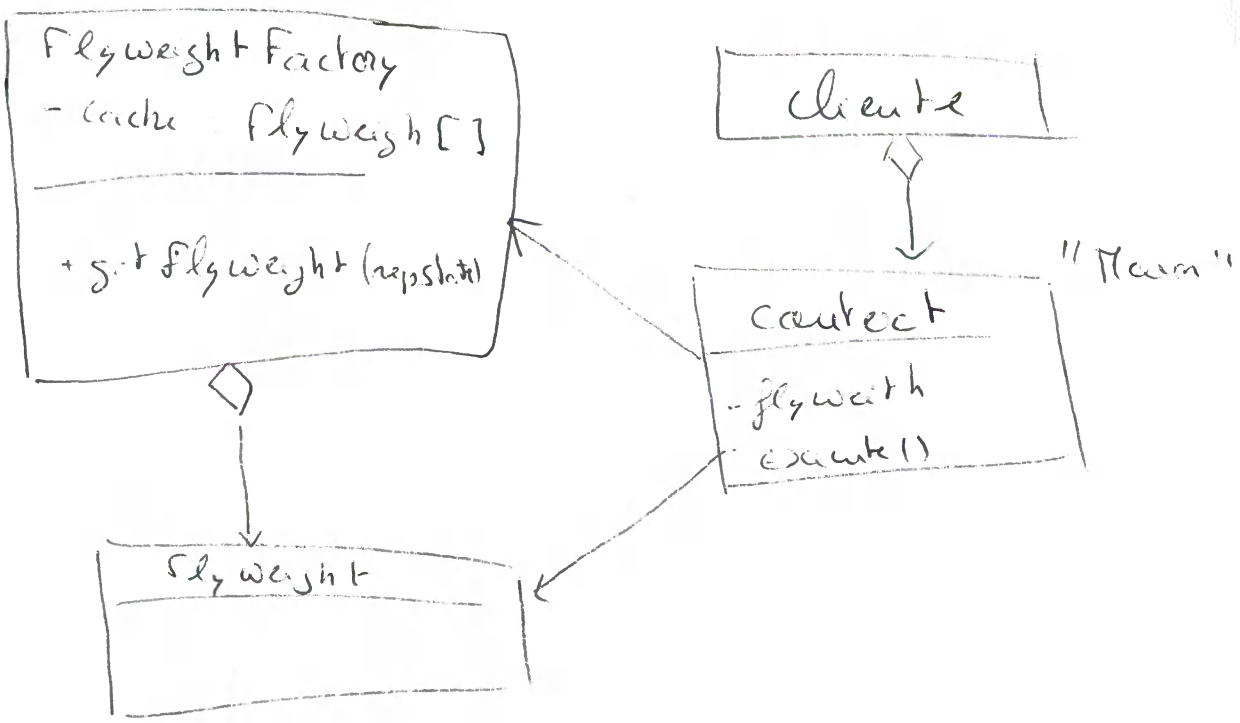
- Pós:
  - → Optimiza RAM
- Contras:
  - → ⊕ CPU cycles
  - → ⊕ Complexidade

Problema que o Flyweight tenta resolver

No implementação de um sistema complexo como um jogo ou uma simulação requer a utilização de muita ram para guardar em memória os dados objetos.

Solução: Em vez de guardar a informação que não muda de cada objeto multiplas vezes guardamos essa informação ~~partilhada entre os dados objetos para~~ em metodos que a usam, assim apenas a informação intrinseca fica "dentro" do objeto permitindo a sua reutilização. Assim é necessário menos objetos pois a unica diferença dos objetos é intrinseca

Componentes: Classe que tratam da lógica do sistema
Classe que implementa a informação mutavel do objeto que pretendemos aplicar o padrão ~~flyweight~~ flyweight, e este vai buscar a informação que é imutavel para todas as instancias deste objeto.

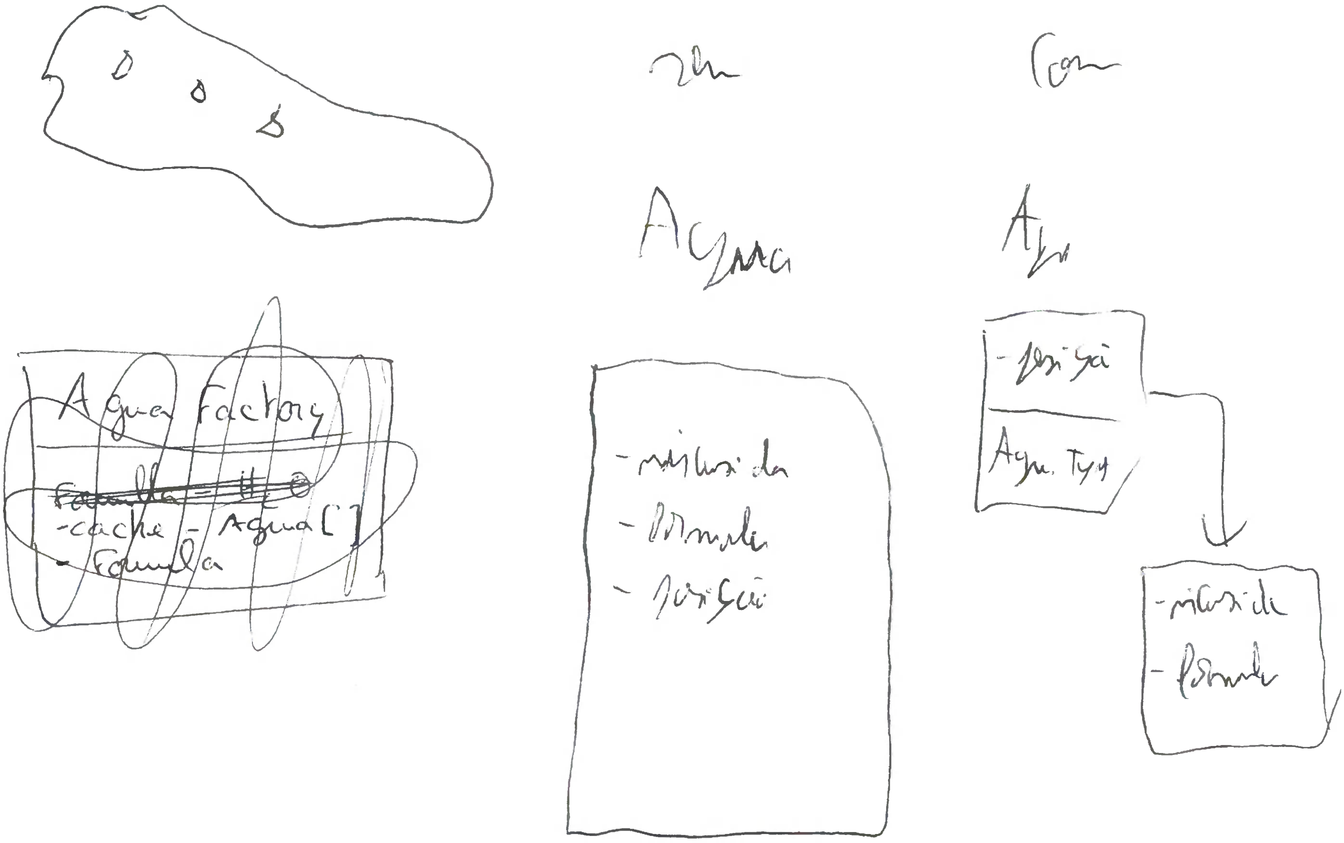


NMec: 113962; 113765; 108536; 113372; 112714; 114192; 113664
111590; 112876; 111590

Pros: Podemos salvar muita RAM

Cons: o processamento pode ser maior pois é necessário recalcular os dados do objeto

O código fica mais complicado



NMec: 113962; 113763; 108536; 113372; 112714; 114192; 113664;
111590; 112876; 111590