Docentes:
Prof. José Luís Oliveira & Prof. Sérgio Matos

Projeto Teórico
Tópico A

Rui Fernandes, 92952

DETI
Universidade de Aveiro

06 – 2020

Padrão 1 – Builder

Introdução

O builder é um *creational pattern* que assenta na construção de objetos complexos num método “passo a passo” separando a mesma da representação do objeto. Isto permite que o processo de construção de objetos gere representações distintas e variádas.

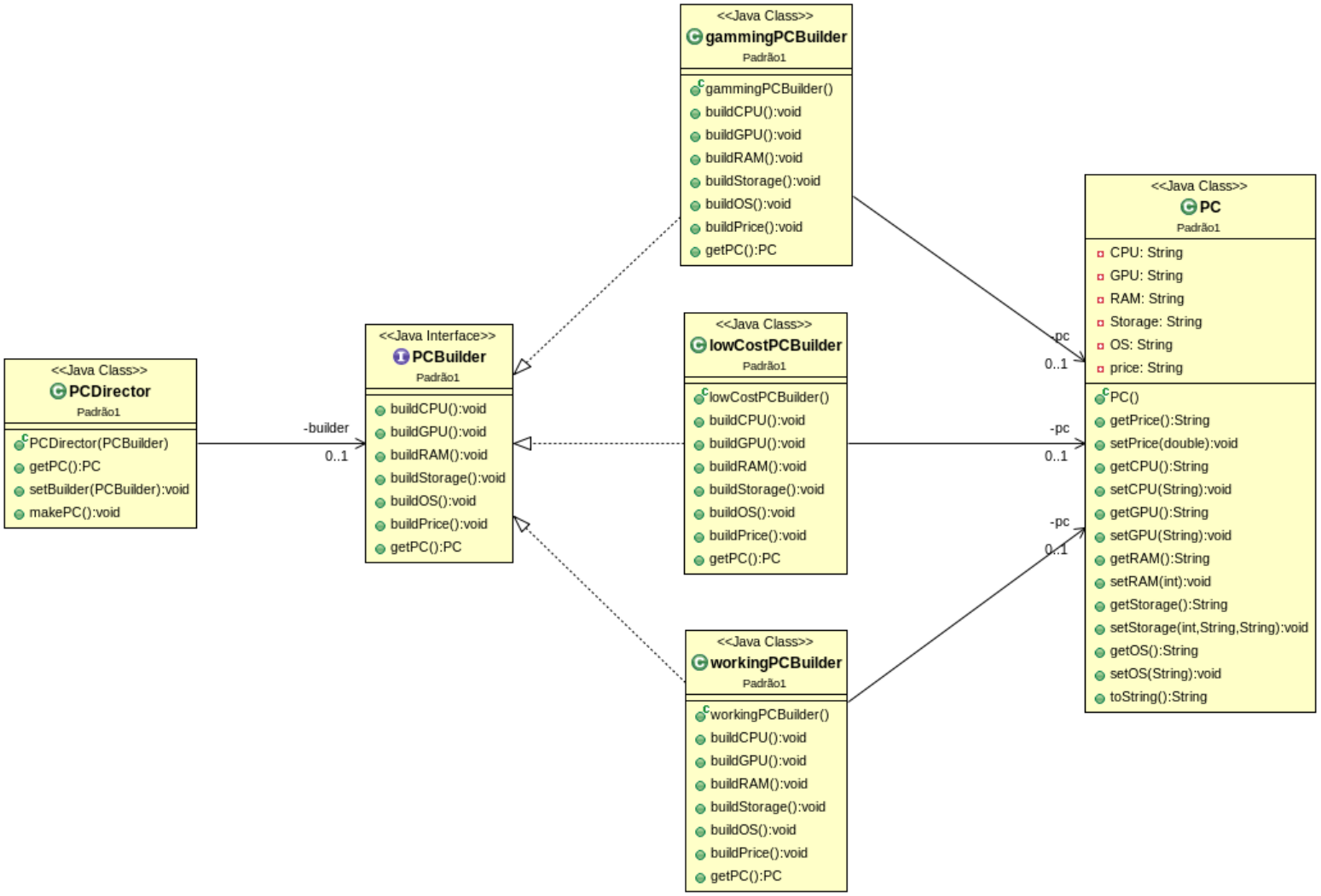
Para este padrão é necessária uma classe abstrata (ou interface) *Builder* que serve como base para classes de *builders* concretos, para assistência é típico criar uma classe *Director* que encapsula um *Builder* e faz chamadas ao mesmo para a criação dos objetos. Assim pode dizer-se que o *Director* define a ordem na qual os passos são executados e por sua vez o Builder fornece a implementação dos mesmos.

Problema

Uma pequena empresa chamada “Tech building” especializa-se em construir computadores fixos customizados de especificações variáveis. Pretende-se então fazer um programa que simule a construção dos vários computadores e o armazenamento em stock dos mesmos.

Os três principais tipos de computadores da empresa são “low cost PC”, “working PC” e “gamming PC”. Os computadores são definidos pelo seu CPU, placa gráfica, RAM, Memória, Sistema Operativo e preço.

Solução



Para representação do objeto complexo que neste caso é um computador criou-se a classe *PC* que contém os componentes essenciais para a construção do mesmo.

A interface *PCBuilder* define os métodos necessários de construção dos componentes de *PC* bem como um método de retorno de um *PC*.

Os três *builders* concretos (*gammingPCBuilder*, *lowCostPCBuilder* e *workingPCBuilder*) implementam a interface *PCBuilder* e cada um constrói um *PC* com especificações distintas. A título de exemplo, o lowCostPCBuilder constrói um *PC* com 2GB de RAM, enquanto que o workingPCBuilder fá-lo com 12GB de RAM.

O *PCDirector* serve para facilitar a interação do lado do cliente, sendo que tem um *PCBuilder* interno que pode ser alterado, e métodos para construir um *PC* (consoante o builder atual) e retorná-lo.

Existe ainda uma simples classe *Client* que testa a utilização das restantes classes relativas ao padrão.

Referências

1) https://refactoring.guru/design-patterns/builder
2) https://www.tutorialspoint.com/design_pattern/builder_pattern.htm
3) https://sourcemaking.com/design_patterns/builder
4) https://en.wikipedia.org/wiki/Builder_pattern
5) Slides Teórico-Práticos de PDS

Padrão 2 – Proxy

Introdução

O proxy é um *structural pattern* que fornece um objeto substituto ou intermediário para outro objeto que consuma muitos recursos. Isto é, o Proxy vai aparentar ser o objeto em causa, no entanto decide e gere quando realmente deve fazer chamadas para o objeto real.

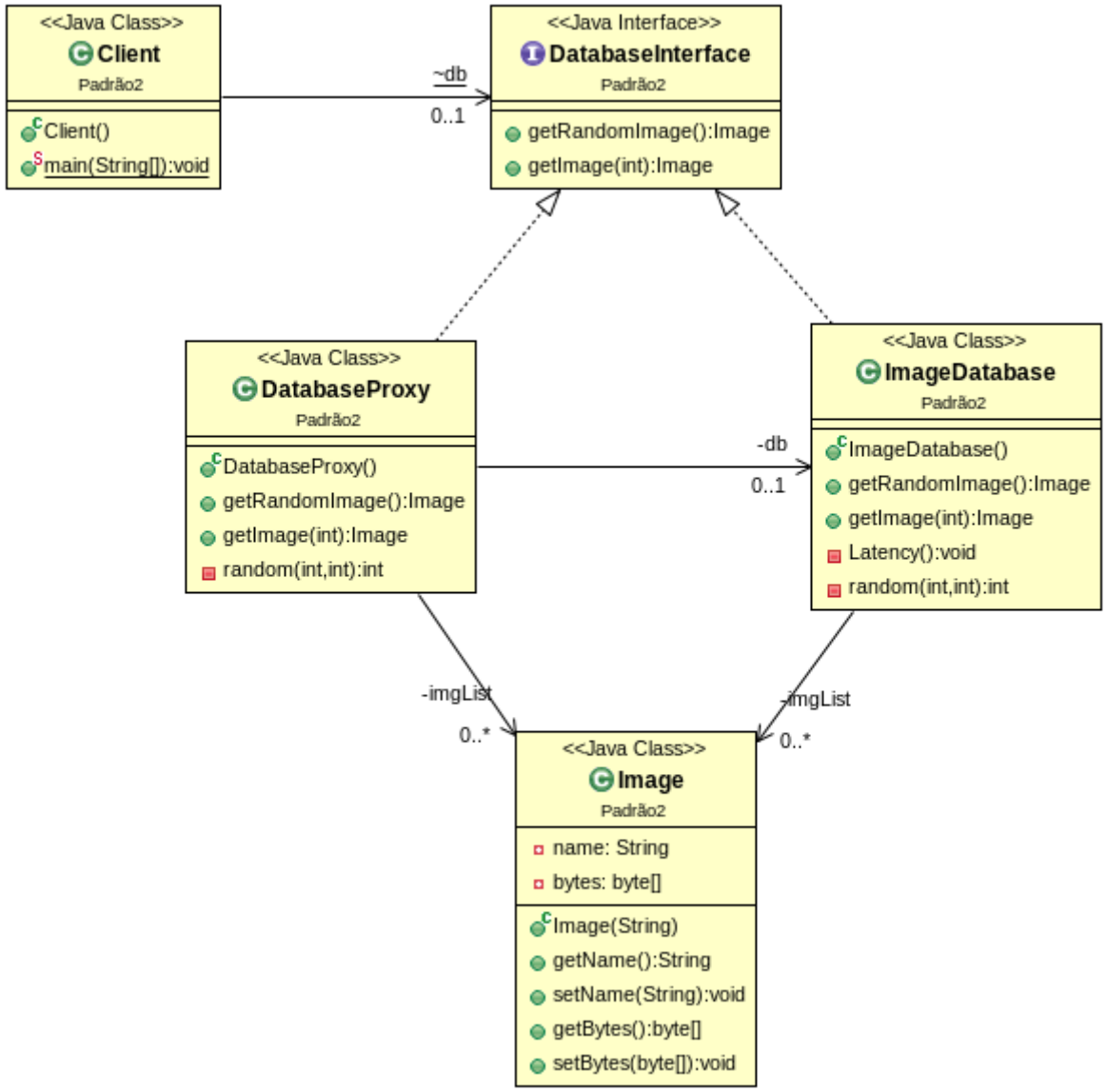
Este padrão é bastante simples sendo que apenas envolve uma classe representativa de um objeto *“resource heavy”,* a classe Proxy, e uma Interface implementada por ambos, permitindo ao Proxy aparentar ser do tipo do objeto real.

Problema

Assumindo que temos uma base de dados que armazena imagens, pretendemos fazer um pequeno programa que nos permita fazer download de imagens para o nosso armazenamento local.

A operação de transformação de imagens em bytes para que possa ser transmitido ao utilizador e posteriormente transformado de novo numa imagem localmente, é algo demorado. Tendo isto em conta, se o utilizador quiser fazer download da mesma imagem varias vezes, vai experienciar um processo bastante demoroso.

Solução



A classe *Image* foi criada para guardar o nome (usado também para encontrar o *path* da imagem) e a informação em bytes de uma dada imagem.

Para simular uma base de dados de imagens criou-se a interface *DatabaseInterface* e a classe *ImageDatabase* que a implementa. Esta classe tem métodos que obtêm a informação em bytes de imagens e devolvem objetos do tipo *Image*. Tem ainda um método usado internamente para simular o tempo de espera para o processo de transformação e transmissão da imagem.

A classe DatabaseProxy implementa a interface DatabaseInterface e vai então ser o nosso *proxy,* contém um objeto do tipo ImageDatabase. Tem uma "*cashe*" local na forma dum HashMap que vai guardar a informação das imagens já transmitidas pela base de dados real, permitindo ao *Client* descarregar a mesma imagem sem ter de passar pelo processo demoroso.

Referências

1) https://refactoring.guru/design-patterns/proxy
2) https://www.geeksforgeeks.org/proxy-design-pattern/
3) https://www.baeldung.com/java-proxy-pattern
4) https://en.wikipedia.org/wiki/Proxy_pattern
5) Slides Teórico-Práticos de PDS