Dezembro de 2014

# Project Manager

Academia RePrograma a tua carreira

Daniel Gomes
Filipe Maia
Pedro Antunes

# 1. Índice

# 2. Apresentação do problema

Com o objectivo de introduzir os formandos aos princípios SOLID e de os fazer analisar código produzido por outros, foi-lhes pedido a análise dum dos trabalhos de grupo desenvolvidos no âmbito do módulo 1 do programa “RePrograma a tua carreira”.

No presente relatório foi analisado o trabalho Project Manager, onde se desenvolveu uma aplicação que possibilita a gestão de projectos e o controlo de custos. Um projecto é definido por um local, uma equipa de consultores e por sub-projectos. A aplicação possibilita a adição e modificação de projectos, a obtenção de custos e o fornecimento de informações sobre todos os projectos.

A análise ao trabalho Project Manager verificou se este cumpre os princípios de boas práticas (SOLID) e, nos pontos este não seguiu estes princípios, a aplicação Project Manager foi modificada de modo a respeitar os princípios SOLID.

Nos próximos capítulos procede-se à descrição dos pontos onde o trabalho Project Manager não verificou os princípios SOLID e das alterações feitas à aplicação.

# 3. **Modelo de domínio, em UML**

Este trabalho faz parte de uma análise a um trabalho já efectuado, para melhor compreensão das alterações realizadas são apresentados dois UML, o primeiro referente ao estado inicial da aplicação e o segundo o fruto das alterações realizadas.

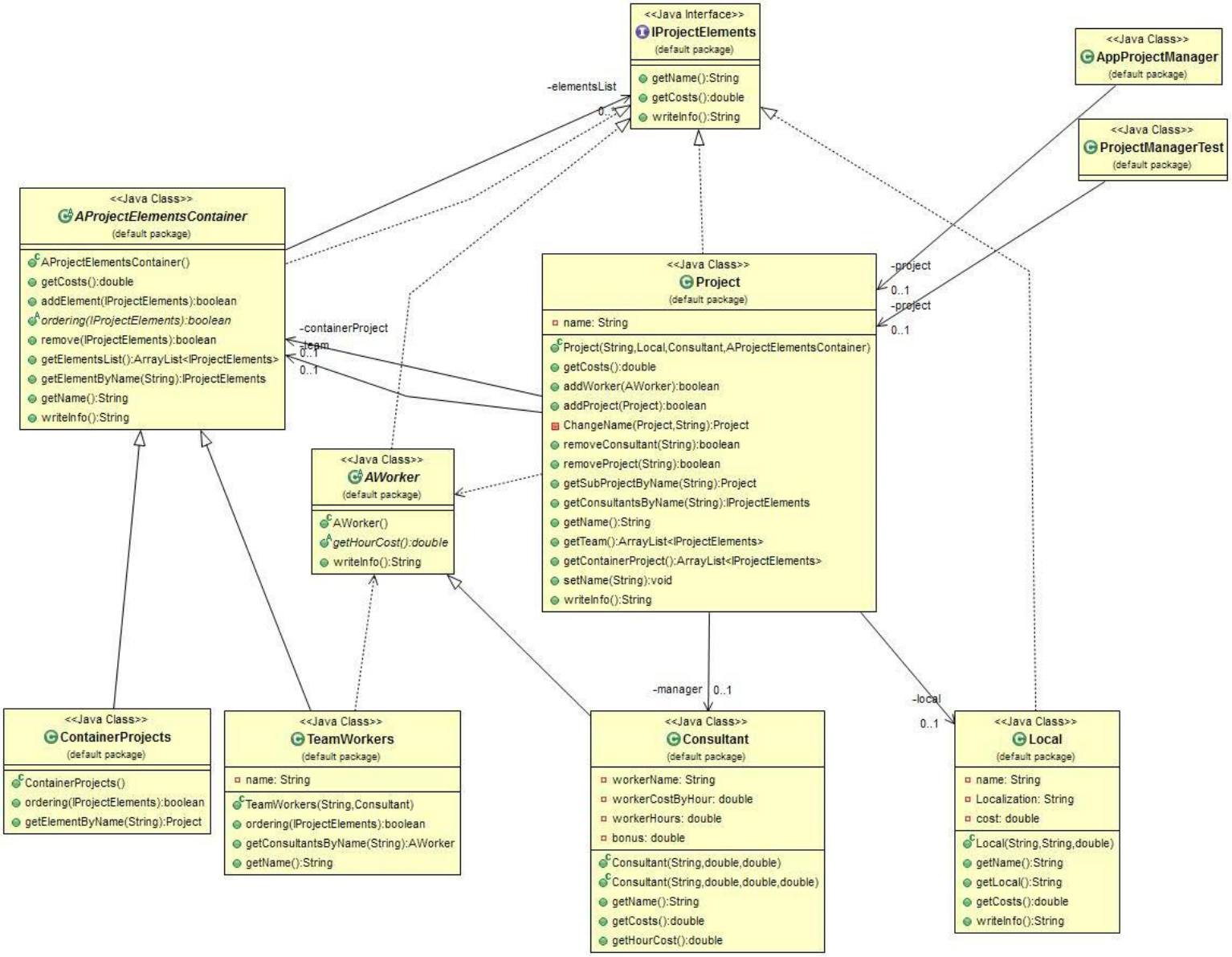


**FIGURA 1 - UML INICIAL**

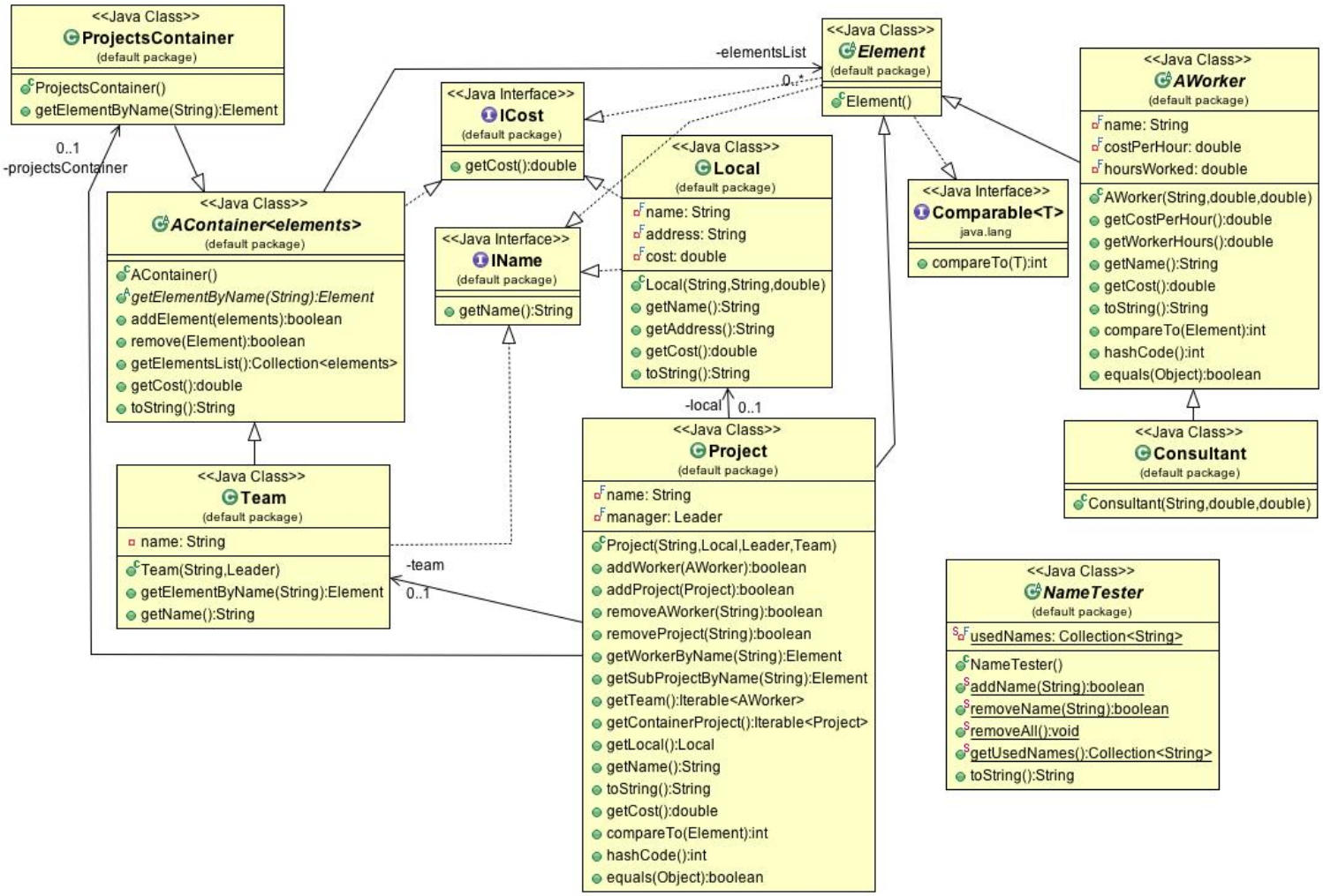


**FIGURA 2 - UML ALTERADO**

# 4. Descrição da solução

## 4.1. Análise dos princípios SOLID

Na análise efectuada ao trabalho Project Manager verificou-se que em alguns casos os princípios SOLID foram cumpridos e em outros não. De seguida são apresentados os vários casos, já apresentados no relatório realizado anteriormente, e apresentada a correcção efectuada.

1. Método getContainerProject e getTeam da classe Project


**Figura 3 - Classe Project, implementada anteriormente.**

Este método retorna um ArrayList, segundo um princípio **D** (The Dependency Inversion Principle), a dependência deve ser de uma interface e não e uma classe concreta. Assim estou a condicionar o posterior uso deste método, restringindo-a a um ArrayList. Seria melhor retornar um Objecto do tipo Collection.

Para vir ao encontro dos princípios *Solid* optou-se por, nos métodos em que se verifique necessário, retornar sempre uma interface possibilitando assim que no futuro este menos restringido ao uso dos métodos.

## 2. Classes Consult e AWorker

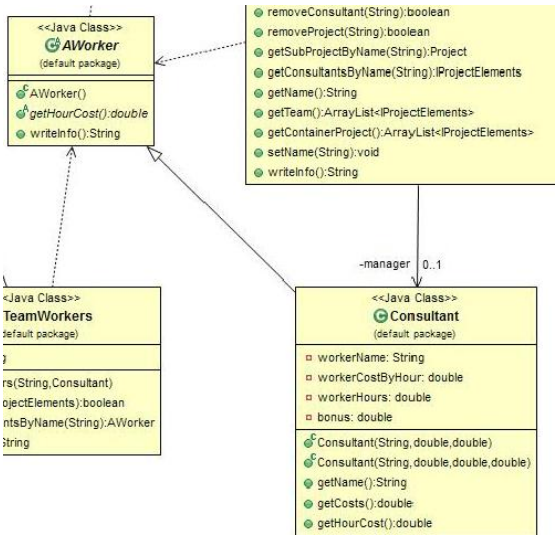


**FIGURA 4 - PARTE DO UML IMPLEMENTADO ANTERIORMENTE**

**(CLASSE CONSULTANT E AWORKER)**

A classe Consultant representa quer os consultores, quer os Lideres. Assim sendo, não cumpre o princípio **S** (The Single Responsibility Principle). Para resolver este problema poder-se-ia criar uma subclasse Líder que estendia a Consult.

Na classe Consultant existem métodos que são características a qualquer trabalhador, o que limita futuras extensões da classe AWorker, no caso surgir um novo tipo de trabalhador. Vindo ao encontro do princípio **O** (The Open Closed Principle), os métodos que são genéricos a um trabalhador deveriam passar para a classe base.

Ao colocar duas responsabilidades á classe Consult estamos a permitir que um consultor possa obter bónus, situação que não se pode verificar. Podemos verificar a ausência do princípio **L** (The Liskov Substitution Principle). A criação da subclasse Líder poderia resolver este problema, se o campo bónus apenas fizesse parte desta nova subclasse.

Para resolver este problema, foi criada a classe *Leader* que representa um líder. Como este elemento partilhas as mesmas características de um *Consultant* e apenas tem uma particularidade que o difere, pode-se considerar que um líder é também ele um consultor. Assim sendo a classe *Leader* será uma subclasse de *Consultant*.

Ao efectuar esta alteração está-se por um lado a resolver o problema relativo ao princípio ***S***, pois cada uma das classes vai representar um tipo concreto de trabalhador e por outro também se resolver o problema relativo ao princípio ***L***, pois impede-se que um consultor possa adquirir as mesmas características que um leader.

O princípio **O** que não foi cumprido na classe Consultant, foi resolvido passando os métodos que são comuns a todos os trabalhadores para a classe *AWorker*, assim no caso de surgir um diferente tipo de trabalhador, este vai receber os métodos que lhe são comuns a todos os trabalhadores.

### 3. Interface IProjectElements

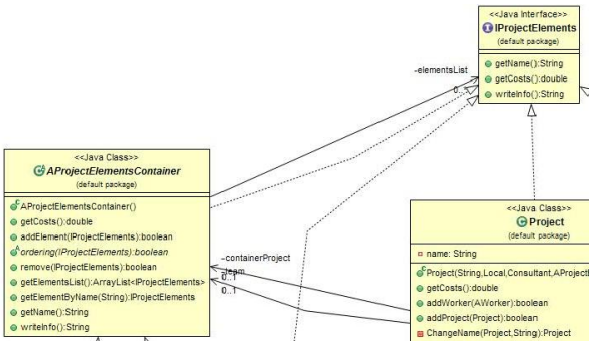


**FIGURA 5 - PARTE DO UML IMPLEMENTADO ANTERIORMENTE**

**(CLASSE APROJECTELEMENTCONTAINER E INTERFACE IPROJECTELEMENTS)**

A interface IProjectElements contém o método getName, que não é comum a todas as classes que a implementam (como é o caso da classe AProjectElementsContainer). Assim sendo, e segundo o princípio **I** (The Interface Segregation Principle), seria melhor criar uma interface, por mais pequena que fosse, que continha apenas este método, que seria implementada pelas classes que contivessem esta característica.

Para resolver este problema, foram criadas duas interfaces, a *ICost* e *IName*. A primeira onde se se coloca o método *getCost* e a segunda onde coloco o método *getName*. Assim posso resolver o problema que surgia na classe *AProjectElementsContainer,* pois já não necessito de reescrever métodos dos quais não necessito.

### 4. Classe AProjectElementsContainer

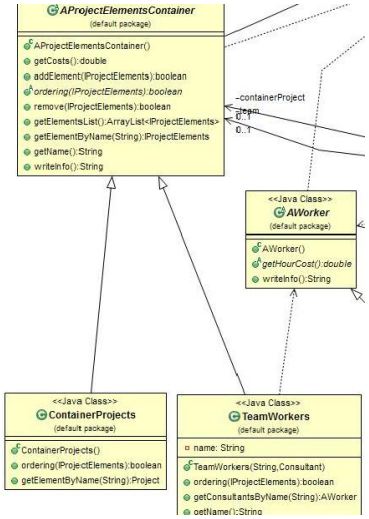


**FIGURA 6 - PARTE DO UML IMPLEMENTADO ANTERIORMENTE**

**(CLASSE APROJECTELEMENTSCONATINER)**

A subclasse TeamWorker contém o método getConssultantByName e a subclasse ContainerProject contém o método getElementByName. Estes métodos realização as mesmas operações que o método da superclasse AProjectElementsContainer. Assim sendo, segundo o princípio O (The Open Closed Principle) teria de se criar o método na classe mãe, e redefinir nas subclasses se necessário.

Poder-se-ia igualmente eliminar os métodos das subclasses e ficar apenas com o método da classe mãe. Seria valido pois, o tipo de objecto retornado é uma abstracção (principio **D** (The Dependency Inversion Principle )).

Em relação a esta classe, optou-se por criar uma classe genérica de *elements* (*AContainer<elements>*).

*Elements* é uma classe abstracta que representa o tipo de elementos que o contentor AContainer pode conter. Existem dois tipos de *elements, AWorker e Projects.* Com a criação de *AContainer<elements>* pretende-se que seja criado um contentor que retenha os trabalhadores e outro que retenha projectos.

Á classe genérica *AContainer<elements>* é associado um nome e um custo, como tal esta classe pode implementar a classe *IName* e *ICost*.

## 4.2. Outras alterações realizadas

De forma a melhorar a aplicação realizada anteriormente realizou-se um conjunto de alterações que não estavam previstas a quando da análise SOLID. Foram elas:

- O método writeinfo() passou a toString(). O método writeinfo() era utilizado em bastantes classes e realizava basicamente o mesmo que método toString(). Assim sendo preferiu-se realizar o Override deste método.
- O método Ordering() procedia a ordenação de uma *Collection*. Sempre que era chamado, lia o critério e era realizada a ordenação.

  Anteriormente, foi utilizado um *arrayList* que não ordenava automaticamente a colecção. O grupo decidiu que mudar o tipo de colecção e passou a utilizar um *TreeSet*, que procedia automaticamente á ordenação dos elementos.

  Assim sendo, é apenas necessário redefinir o método *compareTo*().

# 5. Metodologia de trabalho

No início do trabalho todos os elementos do grupo analisaram o trabalho realizado pelo outro grupo e foram assinalados os principais critérios que não estavam a ser cumpridos. Posteriormente realizou-se a elaboração do relatório e a apresentação.

Numa segunda fase, realizou-se a emenda do trabalho consoante os aspectos assinalados anteriormente. Nesta fase o grupo decidiu dividir responsabilidades. O Daniel procedeu à alteração dos princípios mencionados, o Filipe realizou os testes necessários para a nova aplicação, enquanto o Pedro realizou a aplicação. Ao realizar os testes e a aplicação estes dois elementos do grupo deparam-se com alguns problemas com a execução do novo programa, e tentaram sempre da melhor maneira soluciona-lo. No final O Filipe e o Pedro realizaram o relatório final.

# 6. Conclusão

Os princípios SOLID (Single Responsibility Principle, Open-closed Principle, Liskov Substitution Principle, Interface Segregation Principle e Dependency Inversion Principle) são cinco princípios básicos para programação em linguagens Object-Oriented.

Quando aplicados juntos, têm como função tornar mais simples a manutenção dos sistemas e estender a sua vida útil, facilitando a extensão do software em vez da sua modificação.

Durante a realização deste trabalho foi possível constatar a dificuldade em pegar em código produzido por outros e alterá-lo, sem perder funcionalidades e sem "quebrar" o código.

Após as modificações produzidas neste trabalho com o intuito de respeitar os princípios SOLID, será mais simples no futuro manter e estender este código. A adição de funcionalidades no futuro (por exemplo: o controlo de equipamentos e dos seus custos, ou adição de novos tipos de trabalhadores) é agora mais intuitiva. Torna-se também mais fácil a modificação desta aplicação de modo a se focar mais numa área em particular (como a construção civil, ou outra área qualquer), não sendo tão genérica como é agora.

Durante a execução do trabalho, surgiu a dúvida se poderia ou não haver projectos com nomes iguais desde que pertençam a projectos mãe diferentes. Optou-se por não poderem existir nomes iguais, até por uma questão de evitar que o utilizador confunda qual o projecto que quer tratar. No entanto, este é um ponto que pode vir a ser melhorado no futuro.