

Relatório Final

Trabalho de Grupo

Project Manager

Enunciado 02

Etapa 2

**Formação**

*“ReProgama a tua careira”*

**Elementos**

Filipa Gonçalves

Filipe Maia

Lucas Andrade

Data de Entrega : 7 de Janeiro de 2015

Índice

[Índice 2](#_Toc408132542)

[1. Introdução 3](#_Toc408132543)

[1.1. Estrutura Do Relatório 3](#_Toc408132544)

[2. Apresentação do Problema: 4](#_Toc408132545)

[2.1. Descrição do Domínio (Alterações) 4](#_Toc408132546)

[Suporte a diversas formas de representação 4](#_Toc408132547)

[Comandos 5](#_Toc408132548)

[Requisitos diversos 5](#_Toc408132549)

[3. Alterações ao código implementado anteriormente 8](#_Toc408132550)

[4. Aplicativo centrado no utilizador 8](#_Toc408132551)

[5. Método de trabalho 8](#_Toc408132552)

[4.1. – Distribuição De Trabalho 9](#_Toc408132553)

[4.2. - Recursos utilizados 9](#_Toc408132554)

[6. Discussão e Conclusão 10](#_Toc408132555)

1. Introdução

No âmbito da avaliação do módulo *M2- Desenvolvimento de Aplicações*, da formação *Randstad* *Reprograma a tua carreira*, foi-nos proposto um projecto de trabalho em grupo onde se pretende a implementação de uma aplicação para gestão de projectos.

A realização deste projecto está dividida em três etapas, sendo este o relatório referente à 2ªetapa do mesmo.

Os objetivos de cada etapa são descritos por enunciados autónomos que incluem adições de requisitos (funcionais e não funcionais) ao presente enunciado.

O trabalho consiste na realização de uma aplicação de gestão (*userinterface)* com recurso a comandos que permitirá a utilização do código anteriormente desenvolvido no âmbito de trabalhos de grupo anteriores para a resolução deste enunciado.

Nesta etapa foi-nos pedido a adição de mais comandos na aplicação (*Patch* e *Delete*) e que fosse permitido a escolha, por parte do utilizador, do formato (*text, html* ou *Json*) e do destino (consola ou ficheiro) da informação pedida à aplicação.

Foi também alterada a interface implementada por todos os comandos (de *Command* para *Callable*) e consequentemente a sua estrutura de classes abstratas e métodos a implementar nos diferentes comandos.

Tentou-se estruturar o código de modo a que, sendo alterada a outputStream ou a tipologia de repositório utilizada, fosse necessário efectuar o mínimo de alterações possíveis.

A resolução de todo o projecto foi feita com base no exemplo apresentado e construído nas aulas teóricas *Travel Agency*, fazendo apenas as adaptações necessárias.

* 1. Estrutura Do Relatório

A estrutura do relatório inicia-se com a apresentação do problema e consequentemente a sua interpretação. De seguida, são apresentadas as justificações das decisões tomadas ao longo do projecto assim como das alterações ao código inicial. Aborda-se ainda a metodologia de trabalho utilizada, e, por fim, faz-se uma abordagem técnica do desenvolvimento do aplicativo e apresenta os resultados obtidos.

1. Apresentação do Problema:

***Enunciado original:***

O nosso cliente pretende criar uma aplicação que seja capaz de:

1. Adicionar diferentes elementos a um projecto (local de trabalho, equipa de consultores e subprojectos);
2. Determinar os custos associados ao desenvolvimento de projectos;
3. Mostrar os diferentes elementos que compõem o projecto

Segundo o cliente *“um local de trabalho é caracterizado pelo seu nome, localização e preço. Um consultor é definido pelo seu nome, o seu preço por hora e o número de horas de trabalho que realizou. Uma equipa tem um nome e é composta por um conjunto de consultores, onde um deles é o líder de equipa. A diferença entre um consultor e um líder é que o líder ainda recebe um prémio. O custo da equipa é a soma do custo de cada um dos consultores. Um projecto tem um nome e é gerido por um consultor que não está inserido em nenhuma equipa. O custo total de um projecto corresponde à soma do custo de cada um dos seus elementos.”*

Um outro requisito exigido é referente a apresentação dos resultados do aplicativo, isto é a apresentação “do*s elementos deve ser feita pela ordem alfabética do nome do elemento, enquanto a apresentação dos consultores da equipa deve ser feita de forma crescente pelo seu custo por hora.”*

* 1. Descrição do Domínio (Alterações)

Suporte a diversas formas de representação

Adicionar a todos os comandos de método GET os seguintes parâmetros opcionais:

* accept - define o tipo de representação usada no retorno dos comandos GET. Os valores admitidos são:
  + text/html - *Hypertext Markup Language* (HTML)
  + text/plain - texto simples
  + application/json - *Javascript Object Notation* (JSON)
* output-file - define a localização do ficheiro onde é colocado o resultado do comando. Caso o parâmetro não esteja presente, o resultado é apresentado no *standard output*.

Comandos

Acrescentar os seguintes comandos:

* PATCH /users/{username} - Actualiza a palavra passe do utilizador identificado porusername. Recebe como parâmetro os seguintes elementos:
  + oldPassword - identifica a palavra passe antiga
  + newPassword - identifica a nova palavra passe
* PATCH /projects/{pid} - Altera a informação do projecto identificado por pid. Recebe os seguintes parâmetros, todos opcionais:
  + latitiude - latitude da localização do projecto
  + longitude - longitude da localização do projecto
  + name - nome do projecto
  + price - preço do espaço do projecto
* DELETE /projects/{pid} - Apaga o projecto identificado por pid. Note que apagar um projecto significa apagar também os seus subprojectos.
* PATCH /consultants/{cid} - Altera os dados do consultor identificado por cid. Recebe como parâmetros os seguintes elementos, todos opcionais:
  + name - nome do consultor
  + priceHour - preço hora do consultor

Requisitos diversos

* Acrescentar suporte para modo interactivo quando se executa a aplicação sem parâmetros. Neste modo, a aplicação apresenta um *prompt* onde são inseridos os comandos. Após a realização de um comando é apresentado novamente o *prompt*. A aplicação termina com a execução do comando de saída EXIT.
* Acrescentar o comando OPTION / que apresenta no *output* e no formato definidos nos parâmetros, uma descrição de todos os comandos existentes.

Para uma melhor compreensão sobre os comandos necessários e o caminho a ser percorrido em cada um deles, completámos a árvore de nós apresentada no relatório anterior, cujos nós a preto representam os elementos da *path* inserida (estando a negrito o método) e a laranja as *factory* dos comandos disponíveis em cada um dos caminhos percorridos. Note-se que os nós cujo nome aparece entre chavetas, {}, representam *placeholders*, ou seja, permitem a inserção de qualquer valor desde que diferente do “nome” dos nós fixos irmãos. Nestes casos, é já dentro do comando que é feita a verificação da validade do valor inserido.

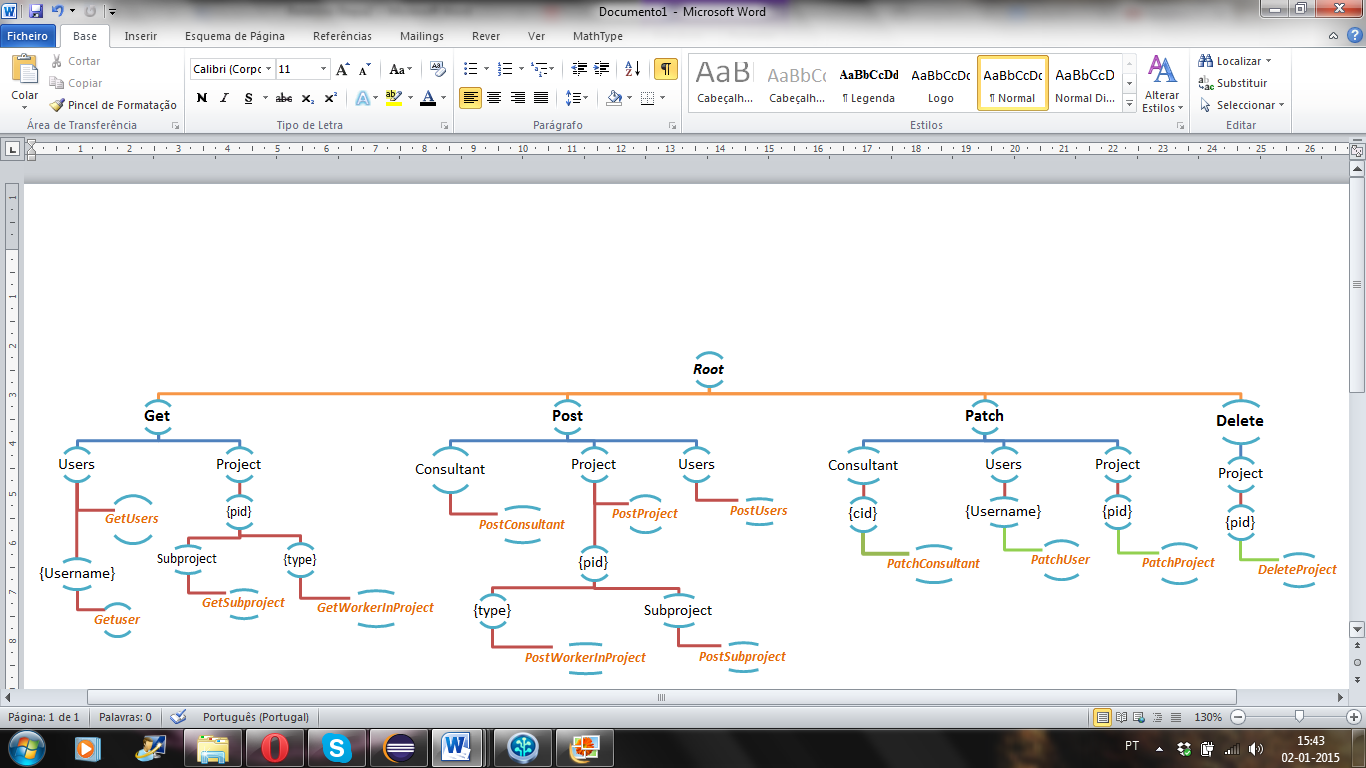


Figura 1- Árvore de Comandos (actualizada).

1. Alterações ao código implementado anteriormente

Tendo sempre por base o exemplo apresentado na aula para o projecto *Travel Agency*, o grupo decidiu que nesta etapa teriam de ser desenvolvidas as seguintes tarefas:

1. Implementação da interface *Callable* em todos os comandos;

Na etapa anterior, para a criação dos comandos foi utilizada a interface *Command* que estabelecia a utilização do método *execute* que não tinha qualquer retorno. Assim estabelecemos que o método *execute* receberia um objecto do tipo *ResultOutputMethodToStream* para a qual o método enviaria toda a informação por ele recebida e processada, foi esta a forma que encontrámos de modo a obter o retorno pretendido dos comandos.

Nesta etapa, implementámos então a interface *Callable* em vez da interface *Command* nos comandos, uma vez que esta estabelece a utilização do método *call()* que retorna um objecto que pode depois ser tratado quer por outro método ou, como foi nossa escolha, por outra classe, para manter o princípio de *Single Responsability*.

Tal como na etapa anterior todos os comandos têm dois métodos “executáveis”, o *call()* que realiza as instruções comuns a todos os comandos de um determinado grupo e o *internalCall()* que realiza as instruções específicas de cada comando. Todos os comandos retornam um App*Element* um conjunto de App*Elements na forma de array*. Assim, decidimos que o *internalCall()* retornaria um array de *AppElement*s que seria depois tratado pelo método *call()*. Nos métodos que retornam apenas um elemento, foi criado um array de apenas um App*Element*.

Para se poder retornar uma mensagem quando necessário, foi também criada a classe *Message* que implementa a interface App*Element* e que, como tal, pode ser retornada pelo método *internalCall()*. É assim mais fácil para os comandos passarem para a consola ou ficheiro uma mensagem de sucesso ou insucesso da operação, caso seja necessário. Inicialmente pensou-se em lançar uma excepção com esse tipo de mensagem, a qual poderia ser apresentada na consola, mas descartou-se essa possibilidade pois considerou-se que não se trata de um bom uso das excepções. Estas devem comunicar que se passou algo que não se estava a espera e que o programa não pôde prosseguir com o funcionamento normal, e não para comunicar ao utilizador que a operação foi realizada com sucesso.

A classe *User*, passou a implementar também a interface *AppElement* para que pudesse ser feito o retorno pretendido dos métodos *internalcall*() dos comandos que produzem um utilizador*.*

O método *call()*, por sua vez, retorna um objecto *Result* definido a partir de um array de *AppElement* e por duas Strings: uma que define onde será impresso o output (na consola, se for null ou num output-file) e a outra que define o formato da impressão (ver ponto 5). No caso de comandos onde não seja possível alterar o método de output dos resultados (os POST, PATCH e DELETE), ambas as Strings são null.

Também o método *newInstance()* das classes que implementam *CommandFactory*, cujo objectivo é criar as novas instâncias dos diferentes comandos, retornam agora Callable<Result>.

1. Criação dos comandos *PatchUser, PatchProject, PatchConsultant* e *DeleteProject* que estendem a classe abstrata *BaseCommandUserAuthentication;*

Na etapa anterior foi-nos pedido que todos os comandos POST, por uma questão de segurança, exigissem a autenticação dos dados do *user*. Nesta linha de pensamento e, uma vez que os novos comandos envolvem tarefas de modificação ou eliminação de informação, decidimos que os novos comandos deveriam também exigir a autenticação dos dados do *user*, logo todos estendem a classe abstrata *BaseCommandUserAuthentication.*

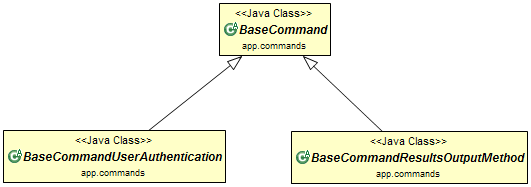
1. Alteração do nome da interface DatabaseElement para AppElement, inclusão do método *getJson()* em AppElement e adição da dependência *org.json*:

Esta interface continua a ter a mesma função que tinha anteriormente, no entanto agora obriga a todos os que a implementam a terem o método *getJson()* que devolve um objecto JSONObject com as informações do AppElement, à semelhança do que acontecia com o método *toString()* dos AppElements.

Para utilização de objectos JSONObject adicionou-se uma dependência nova à nossa aplicação, *org.json* (versão de 13 do Novembro de 2014). Os objectos JSONObject contêm um repositório de pares nome/valor e conjuntos só de valores que permitem a gravação de informações relativas aos AppElements, e facilitam o retorno numa String destas informações formatadas em linguagem JSON.

O método *getJson()* é usado na classe Result (ver ponto 5) para obter as informações dos AppElements, deixando de ser usado para apresentação dos resultados da execução dos comandos o método *toString()*. Apesar de agora não ser utilizado na aplicação, o método *toString()* não foi apagado.

1. Criação de uma classe abstrata da qual estendessem todos os métodos GET, OPTION e futuros comandos cujos parâmetros definam o formato (*accept*) e destino (*output*) da informação de output.



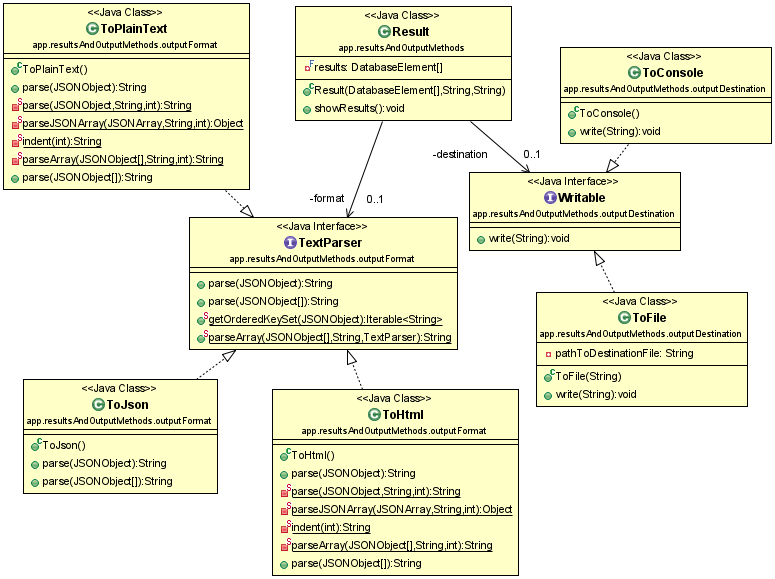
Na etapa anterior todos os comandos GET estendiam directamente a classe abstracta BaseCommand, porque não reuniam nenhuma característica que obrigasse a criação de outra abstração intermédia. Contudo, tendo por base o enunciado actual, como alguns comandos podem definir o formato e destino da informação de Output, foi criada uma classe abstrata BaseCommandResultsOutputMethod, cujo método *call()* estabelece a possibilidade de definição do formato e destino da informação de Output (ver ponto 5). O método *call()* da BaseCommandResultsOutputMethod retorna um Result com os resultados da execução do *internalCall()* e com os parâmetros de formato (texto, HTML ou JSON) e destino (standard output ou ficheiro).

A classe abstracta BaseCommandResultsOutputMethod foi criada por se considerar que é da responsabilidade dos comandos a recolha dos parâmetros inseridos pelo utilizador. Assim, as Strings com o formato e o destino inseridas pelo utilizador são adicionadas ao objecto Result retornado pelo método *call()*.

No entanto não é da responsabilidade da BaseCommandResultsOutputMethod saber para onde e como devem ser enviados os resultados, sendo essa responsabilidade do objecto Result.

1. Criação da classe *Result* e suas classes derivadas que têm como objectivo a representação da informação fornecida pela aplicação em *Json, Html* e *Text Plain* para a consola ou para um ficheiro*;*

Devido à implementação da interface Callable no lugar da interface Command, o método de apresentação dos resultados da execução teve de ser alterado. Tendo em consideração as exigências do enunciado da etapa 2, onde são pedidos três tipos de representação no retorno dos comandos GET (HTML, texto simples e JSON), foram criadas as classes da imagem seguinte.



No conjunto, estas classes vão adicionar funcionalidade extra à aplicação (no sentido que agora também é possível apresentar os resultados no formato HTML e JSON e é possível enviá-los para um ficheiro) e vão substituir a interface ResultOutputMethod, a classe abstracta ResultOutputMethodToStream e a classe ResultOutputAsStringToStream.

A classe Result guarda os resultados da execução do método *internalCall()* dos comandos num array de AppElements e possui o método *showResults()* que envia os resultados no formato desejado (texto, HTML ou JSON) para um destino (standard output (neste caso o standard output é a consola) ou ficheiro).

O construtor da classe Result, para além de receber os resultados da execução do método *internalCall()*, também recebe duas Strings: uma indica o formato e a segunda o destino. Utilizando a String do formato, o construtor do Result instancia um novo TextParser (ToPlainText, ToHtml ou ToJson) através de métodos das classes Class e Package, se o conteúdo da String for inválido é dada a excepção ClassNotFoundException, se a String for vazia é instanciado o formato padrão (neste caso, o ToPlainText). Se a String do destino estiver vazia, o construtor do Result instancia um ToConsole, caso contrário é instanciado um novo ToFile.

O método showResults() utiliza as classes que implementam o TextParser e as que implementam o Writable para enviar os resultados.

A classe TextParser possui o método *parse* que recebe JASONObjects e devolve uma String com os resultados no formato pretendido (ToJson, ToPlainText ou ToHtml). Para que a informacao possa ser apresentada nos vários formatos pedidos (JSON, HTML e Plain Text) pensou-se em usar um formato canónico do qual pudesse em seguida ser feito parse para outro formato. Uma vez que se tem o acesso à biblioteca org.json, utilizaram-se objectos deste tipo como uma espécie de formato canónico, evitando assim ter que fazer parse de Strings para um diferente formato. Ao utilizar a biblioteca, bastou criar um método *getJson()* em cada classe relevante, onde são inseridos as propriedades relevantes dessa classe e os seus valores. Desta forma, quando é necessário escrever a informação lá armazenada, basta recolher o conjunto das chaves existentes no JSONObject e apresentar os valores associados a essa chave. Esta abordagem trás um problema adicional pois o JSON não impõe nenhuma ordem nas suas chaves. Então, a ordem pela qual se introduz as chaves no JSONObject é perdida, e, aquilo que se fez para manter alguma ordem ao fazer parse para HTML ou Plain Text foi obter as chaves por ordem alfabética.

Existem na interface TextParser dois métodos *parse*, um que recebe um JSONObject e outro que recebe um array de JSONObjects, devido ao facto que o método *internalCall()* utilizado pelos comandos devolve um array de DatabaseElement. Em vez de se fazer parse de cada objecto desse array individualmente e acumular todos esses resultados na mesma String, o uso de *parse(JSONObject[] json)* permite que esses resultados sejam acumulados na String de uma forma especial para cada formato. Por exemplo, no formato JSON é importante que, ao fazer parse de vários objectos, estes sejam colocados num array (em linguagem JSON), e este método parse(JSONObject[] json) permite precisamente tratar dessa situação.

Para fazer parse do JSONObject para o formato desejado utilizou-se o padrão Strategy na interface TextParser, com os métodos *parse(JSONObject json)* e *parse(JSONObject[] json)*. As classes que implementam esta interface implementam a estratégia necessária para apresentar a informação contida no(s) objecto(s) JSONObject para uma String no respectivo formato (cada subclasse faz parse para um e só um formato). No futuro se for necessária a adição de mais formatos para apresentação dos resultados, basta criar classes que implementem a interface TextParser e que convertam JASONObjects nos formatos desejados.

Para permitir que a informação seja impressa para a consola ou para ficheiro utilizou-se novamente o padrão de desenho Strategy, criou-se a interface Writable com o método *write(String info)*.

1. Alteração da aplicação para que permita um comportamento distinto quando se executa a aplicação com/sem parâmetros.

A classe AppProjectManager foi modificada de modo a que sempre que receba argumentos quando inicia, os argumentos recebidos são tratados como um comando e esse comando é executado. Caso não receba argumentos quando inicia, mantém o mesmo comportamento que tinha antes.

1. Acrescentar o comando OPTION / que apresenta no *output* e no formato definidos nos parâmetros, uma descrição de todos os comandos existentes.

Ao implementar o comando OPTION escolheu-se introduzir à mão no método call() todos os possíveis comandos e as suas opções. Pensou-se na possibilidade de, em tempo de execução, obter essa informação a partir dos comandos que se encontravam registados no CommandParser, mas isso não fornece informação sobre os parâmetros aceites pelo comando. Mesmo assim, poder-se-ia obter parte dessa informação no campo DEMANDING\_PARAMETERS de cada comando, mas aí encontram-se apenas os parâmetros obrigatórios, e não os opcionais. Então considerou-se que o resultado que se obteria de tal implementação não compensaria o esforço de a implementar.

1. De acordo com os comandos a ser implementados e as especificações do enunciado desta etapa, tivemos de fazer algumas alterações nas classes anteriormente implementadas:

**Classe Abstrata AWorker**

**Classe Project**

**Classe Team**

**Classe Local**

**Classe ProjectsContainer**

1. Aplicativo centrado no utilizador
2. Método de trabalho

No início do trabalho todos os elementos do grupo analisaram o enunciado e o trabalho realizado pelos grupos anteriores e foram assinalados os pontos que teriam de ser alterados, nesta fase, para possibilitar a implementação dos comandos.

Posteriormente construiu-se a App, fizeram-se os testes e redigiu-se do relatório.

O trabalho foi efectuado praticamente em simultâneo, com contribuições valiosas de cada elemento do grupo, devido às diferentes valências e níveis de conhecimento.

* 1. – Distribuição De Trabalho

Ao longo de todo o processo foram distribuídas tarefas distintas a cada elemento do grupo, contudo procurou-se sempre que todos os elementos participassem e estivessem a par de todas as vertentes.

A formanda Filipa Gonçalves ficou responsável pelas tarefas associadas à parte Office (com os contributos dos restantes elementos do grupo), pelas alterações nos métodos execute (alterados para call) dos diferentes comandos e criação da classe *BaseCommandResultsOutputMethod* e participou nas discussões dos métodos, abordagens para a solução do problema.

O formando Filipe Maia ficou responsável pela implementação do código (criação dos novos comandos Patch e Delete), elaboração da documentação do código e participou nas discussões dos métodos e abordagens para a solução do problema.

O formando Lucas Andrade ficou responsável pela implementação do código (Classes e métodos associados à representação do output – Htlm, Text e Json) e dos testes e participou nas discussões dos métodos e abordagens para a solução do problema.

* 1. - Recursos utilizados

Foi utilizado, como repositório de versões do nosso código, o GitHub:

https://github.com/Lucas-Andrade/ProjectManager\_FLM.git

1. Discussão e Conclusão