UNIVERSIDADE
LUSÓFONA

Trabalho Final de Curso

Relatório Final

Pandora - Yet Another Automated Assessment Tool

DEISI198

Alexandre Brigolas (a21803430)

Ricardo Nunes (a21805213)

Orientador: Prof. Dr. Pedro Arroz Serra

25 de junho de 2021

Índice

Índice de Figuras

Índice de Tabelas

Índice de Excertos de Código

Resumo

A automatização de processos domina todos os aspectos da vida actual, principalmente no negócio. No entanto, a educação não é excepção. As ferramentas de avaliação automática - *Automated Assessment Tool* (AAT) - são bastante procuradas, não só, mas principalmente no meio académico, e esta procura tem vindo a crescer nos últimos anos. Estas AATs dão, no imediato, tanto ao aluno, como ao professor, um feedback das avaliações qualitativas e quantitativas dos testes realizados. Algumas destas, como objectivo da Pandora, permitem ao aluno detectar falhas na sua submissão e ressubmeter, incentivando assim uma melhoria contínua e uma aprendizagem escalável ao ritmo do aluno, tornando-a mais autónoma e ligada ao mundo real. A Pandora visa não só incorporar estes aspectos, mas também outros que se encontram implementados em outras plataformas existentes. No entanto, nenhuma plataforma *open-source* implementa todas estas funcionalidades numa só. Algumas destas sendo:

- Permitir a execução de projectos desenvolvidos em linguagem C;
- Ser expansível para outras linguagens de programação;
- Ser de fácil implementação, administração, e manutenção.

A Pandora foi inicialmente desenvolvida em 2019 pelo Prof. Pedro Serra, e apesar de já estar em produção e a ser utilizada em algumas unidades curriculares na Universidade, necessita não só de algumas melhorias, mas também adição de novas funcionalidades. Neste projecto o nosso grupo de trabalho visa realizar esse tipo de melhorias, tanto a nível visual como comportamental em algumas das suas funcionalidades, e desta forma melhorar a sua experiência de usabilidade e consequentemente melhorar a aprendizagem dos alunos.

Abstract

Process automation dominates all aspects of today's life, especially in business. However, education is no exception. *Automated Assessment Tool*s (AATs) are highly sought after, not only but mainly in academia, and this demand has been growing in recent years. These AATs immediately provide both the student and the teacher feedback on the qualitative and quantitative assessments of the tests performed. Some of these, as a goal of Pandora, allow the student to detect flaws in their submission and resubmit, thus encouraging continuous improvement and scalable learning experience adjusted to the student's pace, making it more autonomous and linked to the real world. Pandora aims not only to incorporate the aspects mentioned above, but also others that are implemented on other existing platforms. However, no open-source platform implements all of these features in one. Some of these being:

- Allow for the execution of projects developed in C language;
- Be expandable to other programming languages;
- Easily implemented, administrated, and maintained.

Pandora was initially developed in 2019 by Prof. Pedro Serra, and despite being already in production and being used in some curricular units at the University, needs not only some improvements, but also the addition of new features that aim to improve the interaction and user experience. In this project, our work group aims to make these improvements, both visually and behaviorally in some of Pandora's functionalities, and in this way improve their usability, promote a better user-experience, and consequently improve students' learning.

Capítulo 1

Identificação do problema

Existem várias aplicações e plataformas categorizadas como *Automated Assessment Tool*. No entanto, estas plataformas apresentam alguns problemas, como a difícil implementação e integração, interfaces pouco *user-friendly*, ou expansibilidade reduzida para outras linguagens de programação.

A Pandora visa reunir todas as condições necessárias para se tornar a AAT *open-source* de referência. No entanto, de modo geral, apesar de cumprir algumas das funções para a qual foi concebida, a Pandora está incompleta e com funcionalidades essenciais em falta, que a impossibilita de alcançar o seu objectivo de ser a plataforma de avaliação automática definitiva.

A mesma é desenvolvida em Python e assenta na *framework* Django, que é baseada no padrão de desenho *Model View Template*. Este padrão é uma conjunto de três componentes importantes, o *Model*, a camada de acesso de dados responsável pela ligação à base de dados e o respectivo tratamento de dados, o *Template*, a camada de apresentação constituída por um ficheiro *HyperText Markup Language* (HTML) combinado com *Django Template Language* (DTL) que representa a *User Interface* (UI), e a *View*, responsável pela lógica de negócio e interagir com o *model* para obter os dados necessários e renderizar o *template*[5, 10].

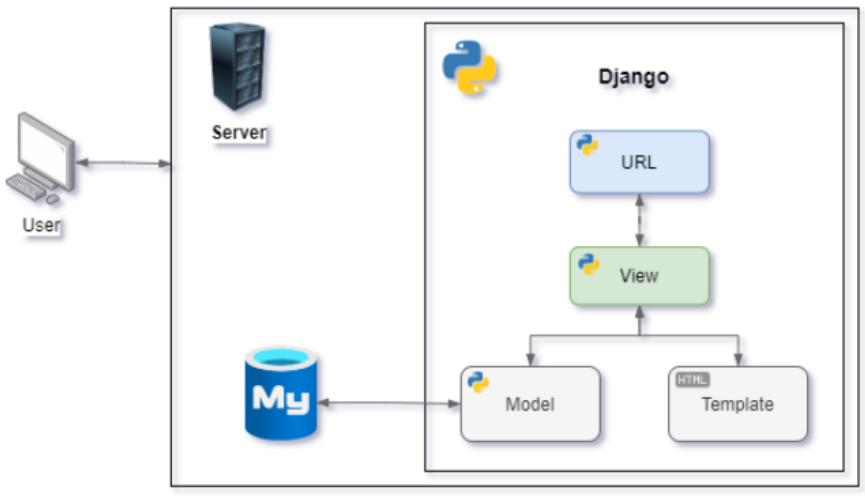

Figura 1.1: Diagrama da arquitectura do Django

A nível aplicacional, está organizada em *contests* (concursos). Um *contest* é um exercício composto por um enunciado, um conjunto de regras, e tem associado uma série de testes. As submissões ao *contest* são efectuadas sempre em equipa, mesmo que individuais, estas restritas por um número mínimo e máximo de elementos.

1.1 Compatibilidade com Múltiplas Linguagens

A introdução de compatibilidade com múltiplas linguagens, de preferência com implementação fácil de cada linguagem adicional, é um dos principais objectivos da Pandora, e provavelmente o objectivo mais desafiante deste trabalho.

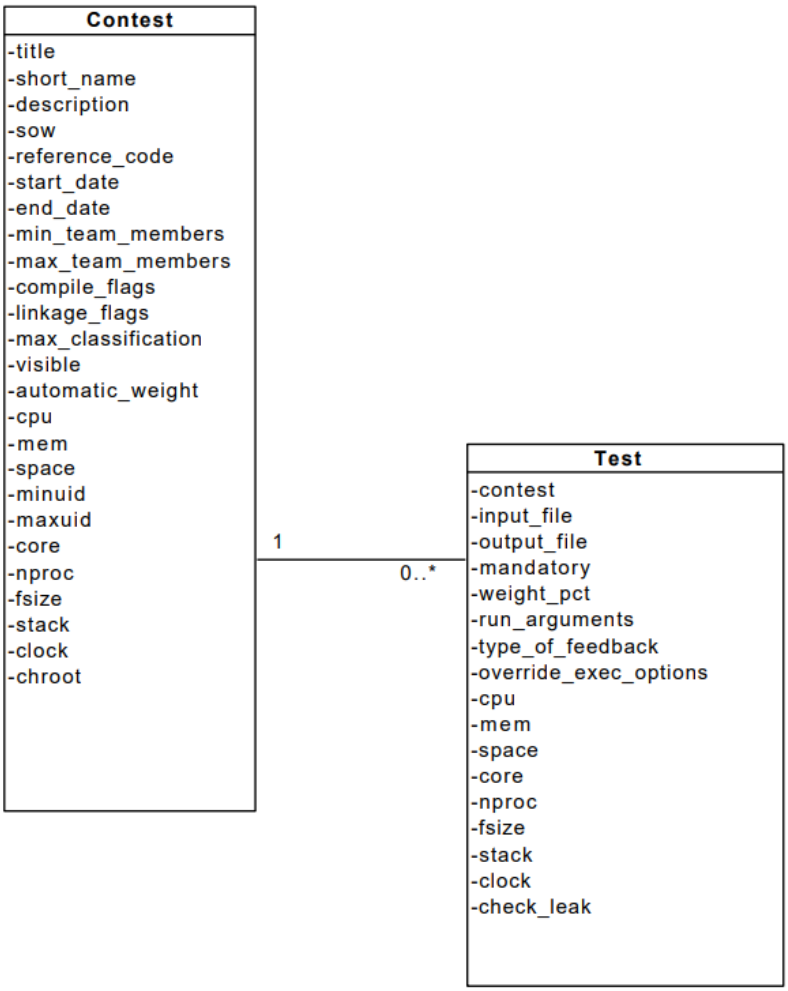

Figura 1.2: Diagrama UML da arquitectura original da Pandora com foco nos *contests*

Na figura 1.2 é evidenciado um obstáculo considerável para o alcance desta meta. A Pandora suportava apenas uma linguagem, C, sem base para inclusão de outras linguagens, visto que, tanto em estrutura como em código, estava *hard-coded* para C, evidenciado pela ausência de escolha de linguagem no *contest*, assim como configurações específicas ao *GNU Compiler Collection* (GCC) presentes em todos os *contests* (≪*compile_flags*≫ e ≪*linkage_flags*≫). Assim, ficámos determinados a arranjar uma solução para este embaraço, tendo em conta que teríamos de consolidar a reestruturação da lógica de negócio referente às linguagens da Pandora e processa-

mento de submissões com a edição e adição de modelos, considerando que o grau de interdependência entre estes é bastante elevado.

1.2 Controlo de Acesso

Encontra-se em falta controlo de acesso em algumas áreas da Pandora, a mais destacada sendo referente aos *contests*. Cada *contest* criado pelo administrador fica visível para todos os utilizadores aprovados (entenda-se aqui como aprovado, permitido acesso à plataforma), e estes podem realizar o *contest* sem qualquer tipo de restrição. Aqui podemos identificar um dos problemas com a Pandora, visto que não é realizado controlo de acesso de qualquer tipo relativamente aos *contests*.

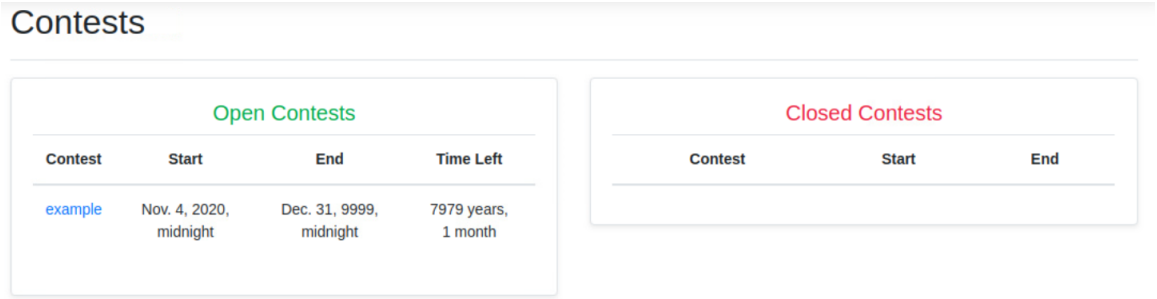

Figura 1.3: Ecrã dos *contests*, susceptível a *cluttering* devido à falta de controlo de acesso e difícil manutenção

No seu estado actual, a Pandora seria inviável para implementação numa instituição como o sistema centralizado para realização de avaliações automáticas (entenda-se aqui como sistema centralizado, um sistema utilizado em várias áreas da instituição, tendo apenas uma instância do sistema), visto que a falta de controlo de acesso ao nível de *contests* implica inúmeros problemas, como por exemplo a difícil manutenção da plataforma devido ao *cluttering* de *contests* na UI, removíveis apenas por acção manual, tal como a prevenção de fraude ou infracção semelhante, visto que qualquer utilizador aprovado, mesmo que aprovado por outra área que não a emissora do *contest*, pode realizar qualquer *contest*.

1.3 Área de Administração

Juntamente com a falta de controlo de acessos para os *contests*, a Pandora também carece de uma área de administração não só capaz de minimizar os problemas descritos anteriormente mas também facultativa de uma gestão fácil, intuitiva, e adaptada aos seus objectivos como projecto. A área de administração actual não fornece confiança ao utilizador. Sendo conjunta com a área de utilizador, separada através de menus visíveis apenas para utilizadores com nível de acesso equivalente a administrador, a área de administração sofre de vários obstáculos. Formulários incompletos, privados de *feedback*, e não funcionais, poucas funcionalidades para além da criação de *contests* e testes, e uma interface pouco ou nada intuitiva, devido à falta de *feedback* por parte da barra lateral de navegação, agravada também pela estrutura de *views* não hierárquica, onde a ausência de *breadcrumbs* dificulta a navegação.

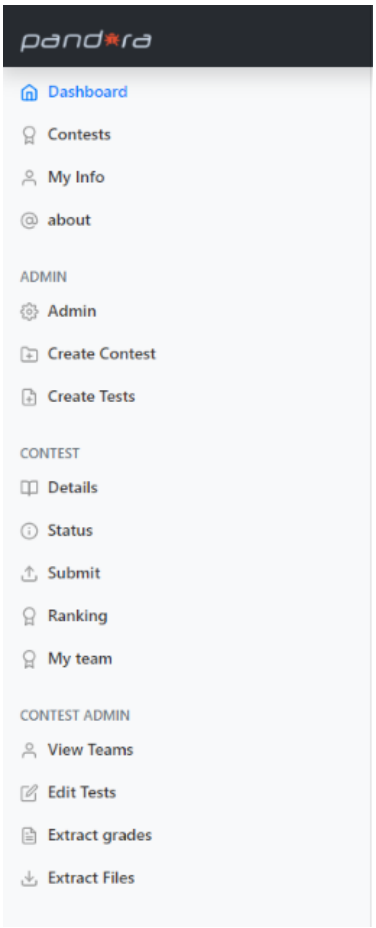

Figura 1.4: *Sidebar* de navegação, com as opções para utilizador e administrador

Neste formato, a área de administração tem pouca utilidade, tendo apenas duas funcionalidades não encontradas na área de administração fornecida pelo Django, a extracção de notas e a extracção de ficheiros. Grande parte das outras funcionalidades são ofuscadas pelas mesmas funcionalidades oferecidas por esta, apesar de a área de administração Django ser uma plataforma para editar directamente os objectos presentes na base de dados, normalmente reservada para casos onde a edição manual dos dados é necessária. Visto isto, concluímos que uma nova área de administração seria crucial para a progressão da Pandora.

1.4 *User Interface*

Aquando dos problemas referentes à área de administração, a UI da plataforma também se encontra com várias lacunas a nível de *interface*, notavelmente na área designada para utilizadores. Apesar de cumprir o seu propósito principal de possibilitar a submissão de código e visualização de resultados, sofre de uma *interface* pouco intuitiva, e não funcional em certos casos, como em dispositivos de dimensão reduzida, onde a barra lateral de navegação não é visível. Factores como a ausência de uma *dashboard* para os utilizadores, navegação confusa e falta de *feedback* por parte da *sidebar*, e páginas muito simples sem grande interactividade para o utilizador contribuem para uma *interface* com espaço para bastantes melhorias.

1.5 Estrutura de Código e Projecto

Um problema invisível para utilizadores, mas presente e crítico para a Pandora atingir o alvo de ser *open-source*, é a estrutura da mesma referente ao código e projecto. Após uma análise inicial, guiada pelo orientador, foi possível verificar que a Pandora encontra-se num estado com parecer de uma prova de conceito, com poucas optimizações feitas a nível de código, sendo este desenvolvido com foco total na rapidez de *deployment*, desprezando boas práticas de programação. Apesar do Python ser uma linguagem multi paradigma, o Django beneficia bastante de programação orientada a objectos, e situações como falta de encapsulamento, inutilização de herança, e irregularidades na denominação de métodos são omnipresentes, o que torna a leitura do código desagradável, dificulta a adição de novas funcionalidades, cria impedimentos a nível de *debug*, e, mais grave, cria uma barreira para a transição para *open-source*, não no sentido de ser impossível a Pandora ser *open-source* com o código no seu estado actual, mas sim no sentido de ser uma grande causa de atrito na criação de uma comunidade *open-source* que suportasse o projecto.

1.6 Testes Unitários

No relatório intermédio referimos a criação de testes unitários como um dos objectivos para este trabalho. No entanto, após reflexão e discussão com o orientador, deduzimos que este objectivo não se alinhava com os outros objectivos deste trabalho, e a sua realização seria demasiado inconveniente por alguns motivos. A reestruturação da lógica de negócio referente às linguagens para compatibilizar a Pandora com múltiplas linguagens, que inicialmente não fazia parte deste trabalho mas decidimos acrescentar como objectivo devido à sua importância, iria atrasar o desenvolvimento dos testes unitários, junto também com o facto da incorporação destes ser algo que reserva estudo sobre as ferramentas a utilizar para tal, efectivamente criando um objectivo que quase certamente não iria ser cumprido até à *deadline* deste trabalho. Decidimos então focar os recursos que iriam ser gastos no desenvolvimento dos testes unitários nos outros processos, nomeadamente o desenvolvimento da compatibilidade multi-linguagem, que de certa forma substituiu esta meta.

Capítulo 2

Levantamento e análise dos Requisitos

Depois de reflexão mais aprofundada sobre o objectivo deste trabalho e de discussão construtiva com o orientador, vários dos requisitos foram modificados, adicionados e removidos. Notou-se uma falta de preparação e pouca análise funcional no levantamento de requisitos realizados na nossa entrega anterior. Com isto decidimos rever e efectuar o levantamento de novos requisitos, não só para estarem de acordo os objectivos deste trabalho, mas também para, de um modo geral, estarem mais correctos, assertivos e concretos.

2.1 Requisitos

2.1.1 Gestão de utilizadores

Requisito Funcional 01

Filtro de utilizadores

Role: Administrador

Pré-condições: Existirem utilizadores.

Descrição: O filtro de utilizadores deve permitir aos administradores pesquisarem por uma das seguintes informações:

- Nome
- Email
- Número de aluno

Requisito Funcional 02

Selecção de múltiplos utilizadores

Role: Administrador

Pré-condições: Existirem utilizadores.

Descrição: Deve ser possível ao administrador seleccionar vários utilizadores, por meio de *checkboxes*, para efectuar operações em "massa".

2.1.2 Gestão de grupos (Turmas)

Requisito Funcional 03

Criação de Grupos

Role: Administrador

Pré-condições: Nenhuma.

Descrição: Deve ser possível a criação de grupos de alunos, para que desta seja possível por exemplo, em vez de associar cada aluno, um de cada vez, a um *contest*, associar o grupo de alunos. Deve ainda ser possível visualizar todos os grupos numa área destinada para o efeito.

Requisito Funcional 04

Área de visualização dos Grupos

Role: Utilizador

Pré-condições: Existirem grupos e utilizadores.

Descrição: Deve ser possível visualizar os grupos numa área destinada para o efeito. Os utilizadores devem ver apenas os grupos a que se encontram associados.

2.1.3 Gestão de *contests*

Requisito Funcional 05

Criação de *contests*

Role: Administrador

Pré-condições: Nenhuma.

Descrição: Deve ser possível a criação de *contests* na plataforma através de um formulário para o efeito.

Requisito Funcional 08

Opção para reavaliar trabalhos

Role: Administrador

Pré-condições: Nenhuma.

Descrição: Deve ser possível reavaliar todos os trabalhos submetidos num *contest*, executando-os contra os testes atuais no *contest*. A nota atribuida ao trabalho terá de ser atualizada de acordo com o resultado dos testes.

Requisito Funcional 10

Adicionar restrição de número de submissões

Role: Administrador

Pré-condições: Nenhum.

Descrição: Deverá ser possível definir no *contest*, aquando da sua criação, um limite máximo de submissões. No caso de ser 0 (zero), a aplicação deve assumir submissões ilimitadas.

Requisito Funcional 11

Limitar a execução a 2 (dois) testes, por erro de tamanho do *output* excedido

Role: Administrador

Pré-condições: Existir um grupo, um *contest* associado a esse grupo, mais do que dois testes nesse *contest* e pelo menos um utilizador.

Descrição: A plataforma deverá parar a execução dos testes quando, em mais do que dois, o output gera um ficheiro com tamanho superior ao limite definido no respectivo *contest*.

Requisito Funcional 12

Criar equipa automaticamente quando o *contest* é individual

Role: Administrador

Pré-condições: Existir um grupo, um *contest* associado a esse grupo, pelo menos um utilizador, associado também ao grupo.

Descrição: Aquando da associação de utilizadores a um *contest*, que seja definido com 1 (um) de limite máximo de elementos por grupo, deverá ser criado automaticamente, para cada utilizador, uma equipa com um nome aleatório.

Requisito Funcional 13

Guardar resultado de testes na base de dados

Role: Administrador

Pré-condições: Existir um grupo, um *contest* associado a esse grupo, pelo menos um utilizador.

Descrição: O sistema deve guardar o resultado de cada teste na base de dados, criando para isso um novo objecto *Classification*, e associando o mesmo ao respectivo teste.

Requisito Funcional 14

Compilação e execução de testes em linguagem C

Role: Administrador

Pré-condições: Existir um grupo, um *contest* associado a esse grupo, pelo menos um utilizador.

Descrição: Deve ser possivel a compilação e execução de testes na linguagem C, a partir de um container Docker.

Requisito Funcional 15

Compilação e execução de testes em linguagem Java

Role: Administrador

Pré-condições: Existir um grupo, um *contest* associado a esse grupo, pelo menos um utilizador.

Descrição: Deve ser possivel a compilação e execução de testes na linguagem Java, a partir de um container Docker.

2.1.4 Interface

Requisito Funcional 16

Melhorar a Interface

Role: Administrador, Utilizador

Pré-condições: Nenhuma.

Devem ser aplicadas melhorias à interface, nomeadamente espaçamento entre botões, indicação de campos obrigatórios em formulários, menu lateral simplificado.

2.2 Requisitos obsoletos e removidos

Requisito Funcional 06

Criação Automática de Ficheiros de Output

Role: Administrador

Pré-condições: Requisito criação de *contests* completo.

Descrição: A criação de ficheiros com os outputs corretos deve ser automatizada e feita através da execução do código de referência do *contest*, submetido no momento de criação do mesmo, com os ficheiros de input fornecidos.

Requisito Funcional 07

Criação de Testes Output via Ficheiros JSON

Role: Administrador

Pré-condições: Requisito criação de *contests* completo.

Descrição: Deve ser possível criar testes para um *contest* via a submissão de um ficheiro .json com uma listagem de testes com os inputs e outputs de cada teste. O sistema deve, após a submissão deste ficheiro JSON, gerar ficheiros de input e output.

Requisito Funcional 09

Criação de Testes Unitários

Role: Administrador

Pré-condições: Requisito criação de *contests* completo.

Descrição: Deve ser possível a criação de testes unitários através de um formulário para o efeito. Estes testes serão os que substituirão os testes atuais presentes na plataforma.

Capítulo 3

Viabilidade e Pertinência

A Pandora é uma ferramenta de avaliação flexível, rápida e sem custos de manutenção elevados, que já está a ser utilizada na Universidade. Está a ser desenvolvida em Python, com base na framework Django, e onde a sua execução é feita num servidor Unix (Linux), revelando-se estável, fiável e rápida.

Este trabalho vem adicionar não só melhorias, mas novas funcionalidades à Pandora, tornando-a mais prática para disciplinas de programação, tanto dentro como fora dos cursos de informática da ULHT, e já se encontra a ser utilizada na Universidade pela disciplina de Linguagens de Programação I. Como referido anteriormente, aprimorar as funcionalidades base e a introdução de novas, bem como expandir a plataforma para que seja capaz de realizar testes noutras linguagens para além de C, são alguns dos objectivos principais deste trabalho, possivelmente resultando numa adopção maior por parte das diferentes disciplinas de programação na faculdade e noutras instituições. A reestruturação do código e do projecto em si realizada neste trabalho aplica-se também à transição para *open-source* que, apesar de não fazer directamente parte do escopo deste trabalho, quando realizada, pode originar a criação de uma comunidade de contribuidores para a Pandora, favorecendo a manutenção e desenvolvimento deste projecto.

No ano lectivo 2019/2020, no final do 2º semestre, foi feito um inquérito aos alunos de Linguagens de Programação I desse mesmo ano, realizado pelo Bruno Leal e pelo Prof. Pedro Serra[6]. Este inquérito foi realizado quando a Pandora já se encontrava em produção e estava a ser utilizado nessa mesma disciplina. As respostas ao inquérito, analisadas em anexo, revelam um *feedback* por parte dos alunos positivo, fortalecendo a viabilidade da Pandora como ferramenta de avaliação. É preciso ter em conta o método de realização do inquérito, podendo o mesmo se revelar tendencioso e levar a conclusões incorrectas, visto que foi realizado, em parte, pelo responsável da unidade curricular onde foi utilizada. De qualquer modo, a Pandora revela-se com potencial para ser uma AAT competitiva, embora necessite de trabalho.

Capítulo 4

Solução Desenvolvida

4.1 Planeamento

4.1.1 Entendimento e análise funcional da Pandora

Depois de todo o processo de aprendizagem sobre o Django, mas antes de analisarmos o código da Pandora, começámos primeiro por interagir com a já existente aplicação, fazendo-lhe alguns testes de funcionalidade para a qual foi concebida, tais como:

- Criação de Utilizadores
- Criação de *Contests*
- Criação de Testes
- Submissão de código funcional e não funcional

Estes testes permitiram-nos desde logo perceber que a Pandora estava incompleta, dado que:

- Encontravam-se formulários em falta, e dos existentes alguns sem feedback, outros não funcionais;
- No caso do formulário de criação de um novo *contest*, era permitida a sua criação sem o preenchimento da data de inicio e/ou data de fim, o que leva a que a página da listagem de *contests* lançasse uma excepção;
- O feedback da *sidebar* de navegação não indicava a página correta, e o menu de navegação era pouco intuitivo;
- A edição de grande parte (quase todos) os modelos da Pandora só era possível através da área de administração fornecida pelo Django, mesmo que os formulários para a edição estivessem presentes na Pandora. Isto tornava algumas funções simples bastante trabalhosas;
- A *responsiveness* da Pandora estava pouco trabalhada, evidenciada pelo facto do menu lateral de navegação (único), não ser acessível em dispositivos de reduzida dimensão. Assim sendo não era possível utilizar a aplicação a partir de uma determinada resolução de ecrã.

Depois de perceber o conceito funcional da Pandora, analisámos, ecrã-a-ecrã a sua correspondência com cada pedaço de código. Reparámos numa programação muito funcional e pouco, ou quase nada, orientada a objectos. Não só nisso, mas também que a estrutura de ficheiros estava, na nossa opinião, mal organizada, havendo até apenas um único ficheiro que continha todas as rotinas usadas na aplicação, levando a sessões de *debug* muito tediosas e difícil adição de novas funcionalidades.

Sendo a função principal da Pandora, a compilação, execução e realização de testes de input/output ao código submetido pelos utilizadores, verificámos que essa mesma funcionalidade estava presente na forma de um comando Shell que lançava uma *sandbox*, com uma série de comandos Linux fortalecendo a segurança e correto processamento das submissões. Percebemos então que a adequação com alguns comandos Linux seria crucial para o desenvolvimento deste trabalho, visto que a segurança e correta execução dos testes depende dos mesmos.

Um pouco de conhecimento do *GNU Compiler Collection* (GCC) era, e é, essencial mas, uma vez já utilizada em unidades curriculares anteriores, já estávamos algo familiarizados com a sua utilização. Não sendo a causa de nenhum dos problemas que deram origem a este trabalho, e visto que a linguagem C se encontrava correctamente implementada na Pandora, o GCC não foi alvo de estudo significativo durante a realização deste trabalho, dado que a implementação de C na nova arquitectura baseada em Docker se trataria apenas de uma migração de funcionalidades já desenvolvidas, tendo na mesma em consideração o cuidado necessário para efectuar a migração sem criar novos *bugs* e problemas.

4.1.2 Concepção do UML

Concluída a análise, e mediante os requisitos definidos para a Pandora, considerámos crítico a concepção de um novo, modificado diagrama UML, antes de qualquer tipo de intervenção a nível de código, para que pudéssemos ter uma visão mais *macro* da aplicação e reflectir sobre as alterações necessárias para implementação destes mesmos requisitos. Esta foi uma das mais importantes etapas no nosso trabalho. Aproveitámos o UML inicialmente concebido e redesenhá-mo-lo no sentido de corresponder aos requisitos solicitados. Na figura abaixo, podemos verificar o seu estado final, que será detalhado nas subsecções seguintes.

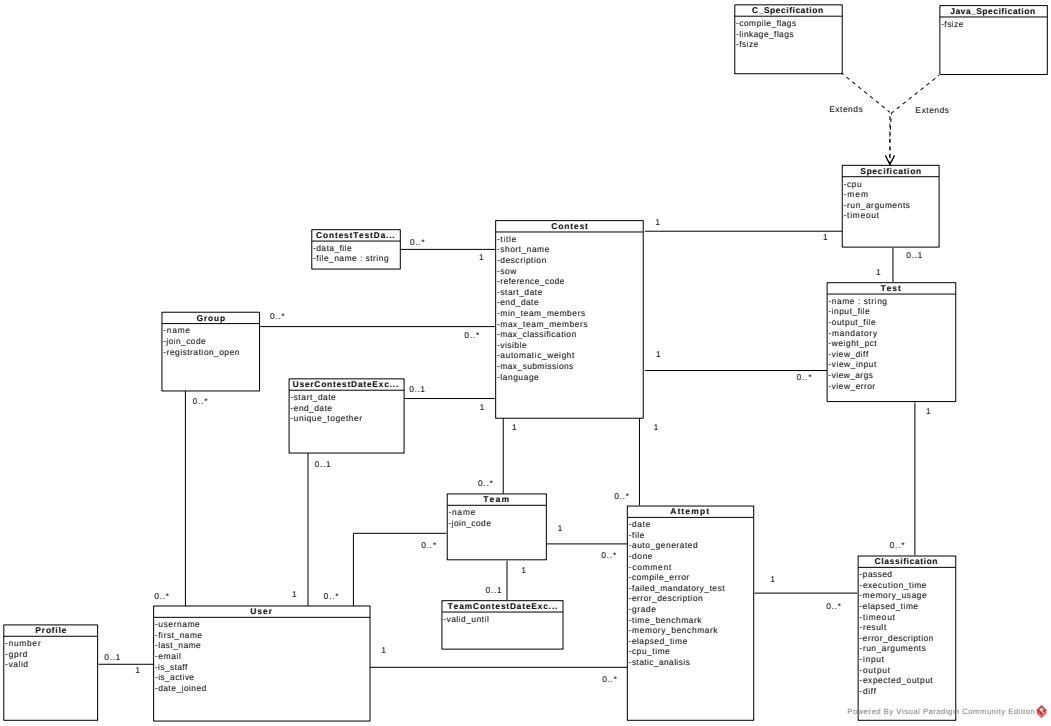

Figura 4.1: Diagrama UML completo da Pandora

Grupos/Turmas

Um dos requisitos deste projecto era a possibilidade de criação de Grupos (Turmas), com o objectivo de adicionar algum controlo de acesso aos contests. Nesse sentido, e conforme se pode verificar na figura 4.2, apenas adicionámos um novo modelo, *Groups*, que se relaciona, de muitos para muitos (N-N), com *Users* e *Contests*. A escolha do tipo de relação é explicada pelo facto de um utilizador poder pertencer a vários grupos e um grupo poder conter vários utilizadores, o mesmo se aplicando aos *contests*.

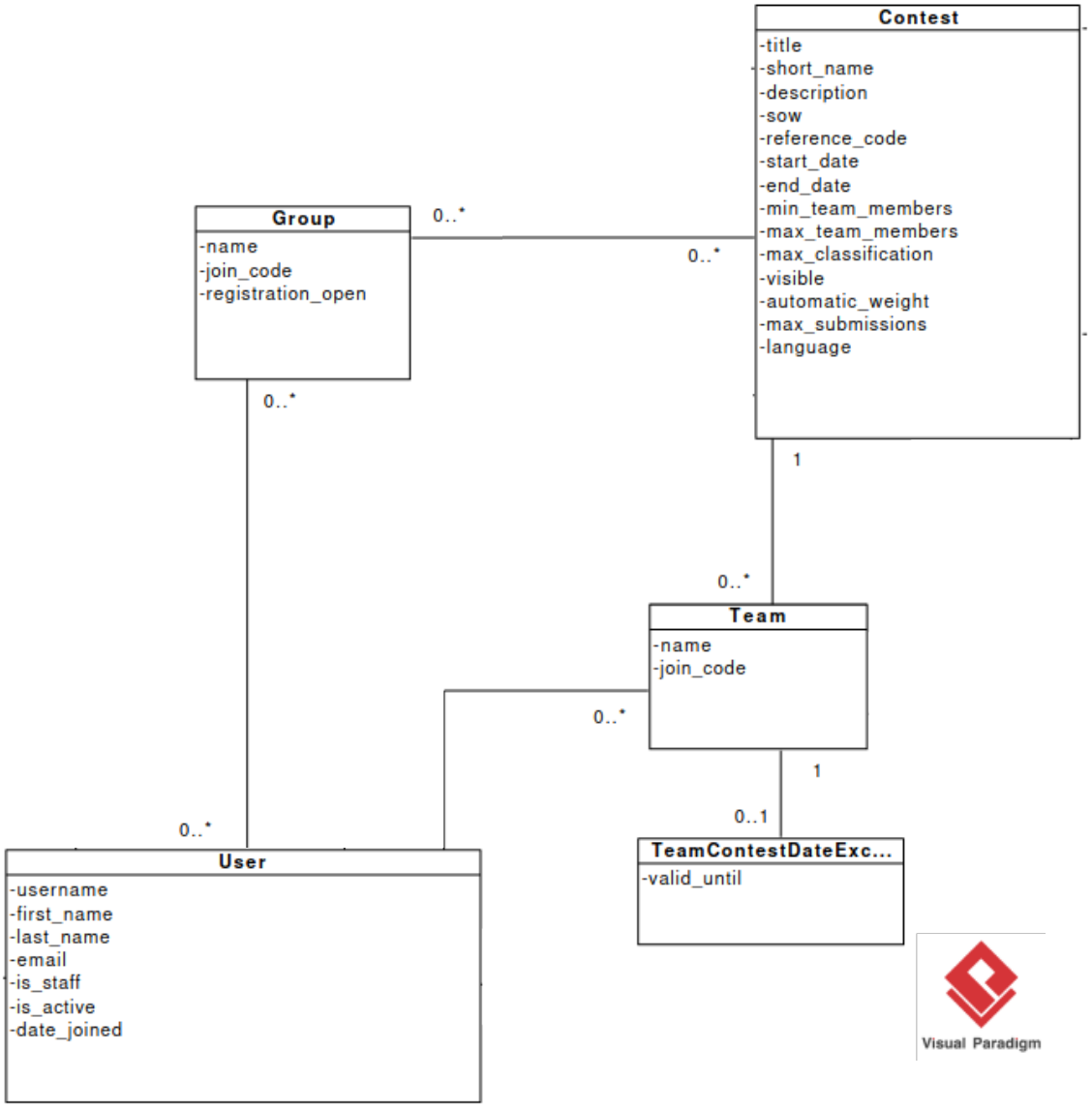

Figura 4.2: Diagrama UML da Pandora com foco nos Grupos/Turmas

Os grupos têm como propriedades um nome («name») para identificação do mesmo, um código de adesão («join_code») para facilitar a admissão de utilizadores a um grupo, e um booleano («registration_open») para determinar se a admissão de utilizadores através do código de adesão é aceite.

Múltiplas Linguagens

Com base também no requisito de ser possível a implementação de novas linguagens, foi necessário modificar o diagrama de tal forma que acomodasse e fosse escalável a mais do que uma linguagem. Tendo isto em consideração, a pergunta que nos colocámos foi:

Dos parâmetros de configuração que temos, quais são aplicáveis a todos os ambientes de execução?

Depois de alguma investigação deduzimos que apenas podíamos ter um parâmetro «timeout» (referente ao comando Linux com o mesmo nome) que termina a execução do teste assim que o mesmo ultrapassasse o tempo limite estipulado, ficando algo protegidos contra eventuais erros causadores de execuções demasiado demoradas como, por exemplo, *loops* infinitos no código submetido. Poderiam também haver argumentos personalizados que o administrador pudesse querer correr juntamente com a execução dos testes, ao qual chamámos «run_arguments». Acompanhámos estes dois comandos por outros dois comandos referentes ao Docker: «mem», que limita a memória RAM do *container*, e «cpu», que limita o número de cores (físicos) que são utilizados pelo *container*.

Assim, e com base nesta lógica e na aprendizagem desenvolvida ao longo dos 3 anos decorridos, decidimos implementar, para este caso, herança nos modelos da Pandora, de forma a reduzir também a dependência directa entre os *contests* e a linguagem C. Para isso, decidimos criar uma nova classe abstracta, intitulada de *Specification*, que seria a base para a implementação de todas as linguagens na Pandora. Esta classe incorpora as propriedades acima mencionadas, no entanto, sendo abstracta, obriga a que cada linguagem tenha de ser implementada através dos modelos da Pandora, mas oferecendo também liberdade às mesmas de incluírem características e propriedades específicas.

Ou seja, o modelo/classe *Specification* é a classe abstracta que tem, de momento, duas derivadas:

- *C_Specification* - Contém todas as propriedades e métodos específicos para a linguagem C
- *Java_Specification* - Contém todas as propriedades e métodos específicos para a linguagem Java

Intuitivamente, seguindo esta lógica, quando, no futuro, for necessário implementar outra linguagem na Pandora, é possível acrescentar um novo *model* que contenha todas as propriedades e métodos específicos para essa mesma linguagem, juntamente de uma configuração referente ao Docker.

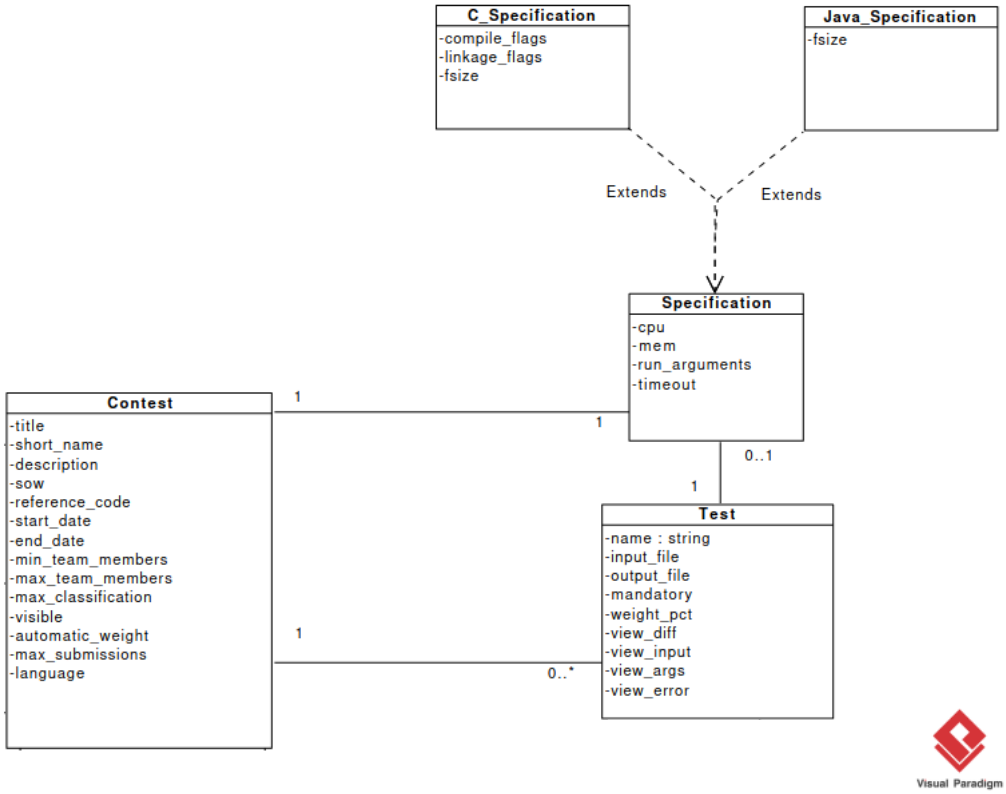

Figura 4.3: Diagrama UML da Pandora com foco na multi-linguagem

Na figura 4.3 é apresentada a secção do diagrama UML referente às classes *Test*, *Contest*, *Specification* e derivadas. É possível verificar que tanto a classe *Contest* como a *Test* ambas têm relação com a *Specification*, a primeira tendo uma relação de um para um, a segunda também uma relação um para um mas que pode não existir.

De base, todos os *contests* têm obrigatoriamente um *Specification*, que será a configuração de referência para a execução dos testes desse *contest*. No entanto, os testes podem ter ou não. No caso dos testes terem um *Specification* associado, esse será utilizado em substituição do de referência associado ao *contest* ao qual o teste pertence. Assim, os testes podem ser alocados mais recursos, dados tempo limite de execução maiores, serem executados com parâmetros de compilação ou execução diferentes, entre outros, efectivamente atribuindo alguma liberdade ao administrador de modificar as configurações de execução de cada teste caso seja necessário.

Tentativa/Teste

Acompanhados do *Contest*, nesta secção podemos observar três modelos principais ligados ao processamento de submissões: *Attempt*, *Classification*, e *Test*, . Quando um utilizador envia o seu código para ser avaliado, é criado uma nova *Attempt*. Este modelo está relacionado com o *Contest* por isso conseguimos aceder directamente aos testes relativos ao mesmo.

Aquando da execução de cada teste, é criada uma nova *Classification*, que regista se o teste passou ou não, o seu tempo de execução, quanta memória utilizou, o seu output, entre outros. No fim, após todos os testes terem sido executados, é calculada a nota com base na propriedade «*weight_pct*» de cada teste, que é a percentagem da nota atribuída a esse teste, da nota máxima do respectivo *contest*, acessível na propriedade «*max_classification*».

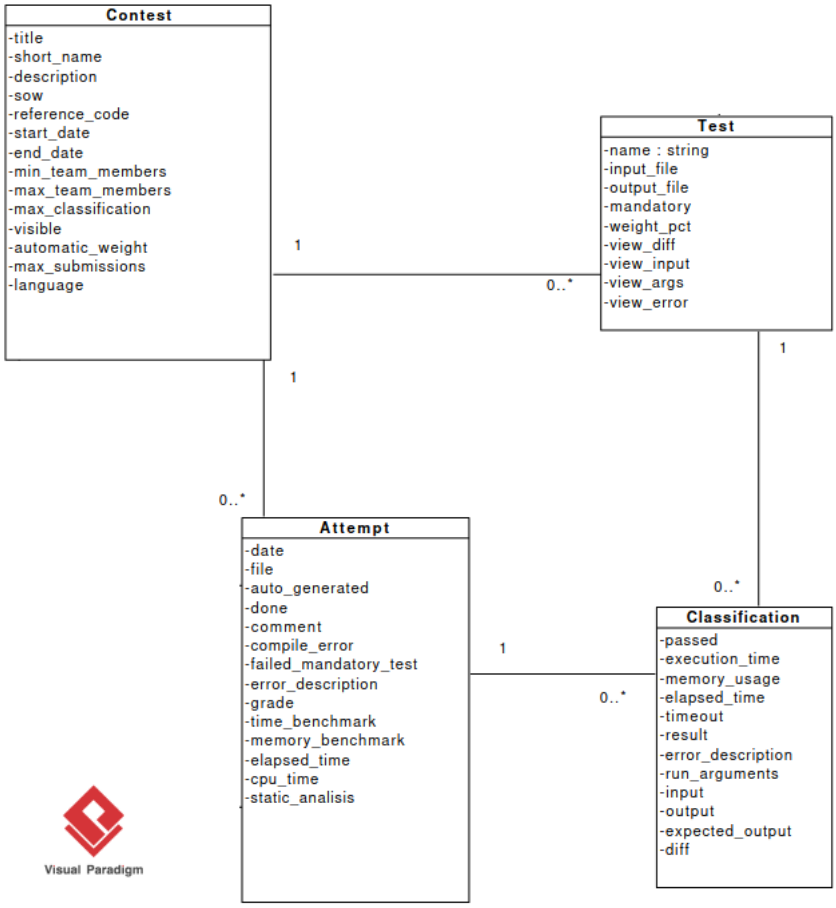

Figura 4.4: Diagrama UML base da Pandora com foco nas tentativas e respectivos testes

4.1.3 Docker como solução para múltiplas linguagens

O Docker

O Docker é uma ferramenta desenhada para tornar mais fácil a criação, implementação e execução de aplicações utilizando para isso, *containers*. Os *containers* permitem ao programador "empacotar"não só a aplicação, mas todas as partes que lhe são necessárias, como as bibliotecas, dependências e outros ficheiros necessários, bem como o ambiente de execução em si (até um certo ponto). Assim, o programador tem confiança que a aplicação é executada correctamente em praticamente todos os sistemas operativos, independentemente das configurações que essa a máquina possa ter. Os *containers* do Docker são muito semelhantes a máquinas virtuais, o que veio beneficiar muito os programadores no sentido em que estes se podem concentrar mais em desenvolver o código, sem se preocuparem com o sistema onde corre.

Docker para a Pandora

As tecnologias de *containerization* de aplicações, como o Docker, enquadram-se bem no contexto de uma AAT devido às imagens e aos *containers*. As imagens seriam configuradas de modo a garantir um ambiente virtual para cada linguagem que permitisse segurança, viabilidade, e consistência da execução de código submetido, enquanto que os *containers* seriam as instâncias dessas imagens, utilizáveis de modo a que qualquer resíduo resultante da compilação e execução de código submetido seja eliminado juntamente com o *container* no final da sua execução, isolando essa execução num ambiente virtual próprio de modo a reforçar a segurança da Pandora.

A questão da segurança era bastante importante, dado que a versão anterior da Pandora assentava na utilização do *safeexec*, uma *sandbox* Unix que permite executar comandos num ambiente controlado e limitado. Contudo, esta *sandbox* está obsoleta, onde o Docker a virá a substituir (ou complementar se necessário, visto que a utilização do Docker não impossibilita automaticamente a utilização de *sandboxes* dentro do *container*).

O facto dos *containers* serem fácil e rapidamente criados e destruídos contribuiu bastante para a escolha do Docker para a Pandora, pois sabemos, por experiência própria com utilização de outras AATs, que a concorrência de submissões pode alcançar níveis bastante elevados e consigo trazer uma série de problemas. Com o Docker é possível ter um ambiente isolado para cada submissão, facilmente tornando as submissões paralelizáveis através de outras ferramentas, amortecendo os problemas que poderiam surgir.

4.2 Desenvolvimento

4.2.1 Análise e reestruturação do código

Estrutura do Projecto

A estrutura de projectos em Django organiza-se em duas partes - o projecto e as aplicações. Como projecto interpretamos o agregado de todas as aplicações que o compõem. As aplicações são *packages*, escritas em Python, que oferecem funcionalidades à aplicação. Originalmente, a Pandora era composta por apenas uma aplicação, a aplicação *contest*. Nesta aplicação estavam incluídas todas as *views*, *templates*, e funcionalidades relativas ao funcionamento do Pandora como AAT, com excepção de algumas páginas "soltas"e os *models*, que se encontravam na raiz do projecto, fora da aplicação *contest*.

A Pandora foi então reestruturada para ter 3 aplicações: *Shared*, *Admin*, e *User*. A aplicação *Shared* será utilizada para conter todos os modelos, funções, e funcionalidades reutilizáveis e partilhadas entre as outras duas aplicações, *Admin* e *User*. Como o nome indica, a aplicação *Admin* guardaria o código e referente à secção de administração da Pandora, enquanto que a aplicação *User* faria o mesmo para a secção de utilizador.

Templates e *Views*

A nível de *templates*, estavam pouco optimizados, não fazendo uso de herança para reduzir a quantidade de código duplicado. Decidimos adoptar uma *component-based approach*, organizando os *templates* de cada aplicação em 3 secções: *components*, *layouts*, e *pages*. *Components* referem-se às peças (botões, tabelas, gráficos, entre outros) que trazem funcionalidade aos *templates*. *Layouts* a estrutura geral dos *templates* (áreas de navegação, *sidebars*, *headers*, e *footers*) - estes que por sua vez herdam de outros *layouts*, até um *layout* base. *Pages* que herdam um *layout* e incluem componentes para criar um *template* que represente uma página da aplicação. Assim, as componentes seriam reutilizáveis entre diferentes páginas, as inconsistências entre *layouts* de difrentes páginas seriam eliminadas devido à herança hierárquica dos mesmos, e a duplicação de código ficaria bastante reduzida.

Com esta abordagem, tornou-se possível implementar novas *páginas* de forma simples, visto que para criar uma página seria apenas necessário herdar de um *layout* e incluir as componentes necessárias.

Excerto de código 4.1: Código exemplo de uma página

```
{% extends "path/to/a/layout.html" %}
{% block thisCodeWillBePlacedInTheLayout %}
    <div class="row">
        <div class="col-lg-12">
            {% include 'path/to/your/component.html' %}
        </div>
    </div>
    <div class="row">
        <div class="col-lg-12">
            {% include 'path/to/another/component.html' %}
        </div>
    </div>
{% endblock %}
```

No entanto, esta nova organização revelou um obstáculo na construção de páginas - algumas componentes e *layouts* necessitam de informação dinâmica, acedida nos *templates* através de interpolação. Estes dados estão disponíveis através do *context*, um dicionário que é fornecido ao *template* no momento de renderização pela view. O problema surge quando duas componentes ou *layouts* necessitam de dados desse dicionário que estão gravados na mesma chave. Como solução implementámos o que chamámos de *context functions*.

Excerto de código 4.2: Código exemplo de uma *context function*

```
def get_example_component_context():
    """
    This function returns the necessary context for
    example_component, which displays all contests
    that are still open (i.e. whose end date has not
    yet been reached).
    """
    open_contests = Contest.get_open_contests()
    return {
        'path_to_example_component': {
            'open_contests': open_contests
        }
    }
```

Estas *context functions* servem para prefixar o contexto requerido pela componente ou *layout* com o caminho e nome do ficheiro, tornando assim a chave única, e depois tratar e inserir os dados necessários na secção do dicionário acedida através dessa chave. Deste modo, é possível garantir que todas as componentes e *layouts* tem uma secção do dicionário própria, e os dados não se misturam. A interpolação feita no template da componente ou *layout* necessita também de utilizar esse mesmo prefixo.

Excerto de código 4.3: Código exemplo de uma *view*

```
def example_view(request):
    """
    In this example view, we will render a page template
    referenced in template_name, while updating the
    context to allow for the components it includes
    to have the data they need.
    """
  template_name = 'path/to/my/page/template.html'
  context = {}
  context.update(get_example_component_context())
  context.update(get_another_component_context())
  return render(request, template_name, context)
```

Como podemos verificar pelo código acima, a lógica de tratamento de dados para cada componente fica também isolada dentro da sua função de contexto, deixando o código da view mais limpo e tornando a componente assim totalmente reutilizável, onde do lado do template será apenas necessário incluir a componente, e do lado da view, caso a componente necessite e tenha uma função de contexto, actualizar o contexto através dessa função.

Dito previamente, outro dos problemas identificados foi a falta de controlo de acesso a certos recursos. O caso de acesso aos *contests* foi resolvido através da implementação dos grupos, colecções de utilizadores aos quais seriam atribuídos *contests*, permitindo a esses mesmos utilizadores participarem nos *contests* dos grupos aos quais pertenciam. As *queries* utilizadas para *fetching* dos *contests* aos quais os utilizadores tinham acesso foram modificadas para obterem apenas aqueles que estivessem associados a um grupo ao qual o utilizador pertence. Mas, sabendo o «id» de um *contest*, o utilizador poderia tentar aceder a certas *views* relativas ao *contest* na mesma. Para resolver este problema, e outros semelhantes, foram utilizados *view decorators*.

Excerto de código 4.4: *View decorator* para controlar acesso aos *contests*

```
def user_has_access_to_contest(function):
    def _inner(request, *args, **kwargs):
        contest_id = kwargs.get('contest_id')
        contest = Contest.get_by_id(contest_id)
        if contest and contest.user_has_access(request.user):
            return function(request, *args, **kwargs)
        raise PermissionDenied
    return _inner
```

Esta função, quando utilizada como decorador ligado à *view*, intercepta a invocação da função da mesma, executando toda a sua lógica antes da *view*, e permite extrair informações do *request* passado à *view*, de modo a obter dados necessários como, por exemplo, o utilizador que efectuou o *request*. Podemos também extrair parâmetros passados no URL, neste caso o «contest_id», para saber a qual objecto o pedido se destina. Aplicando a lógica necessária, e se for detectado um acesso indevido, é lançada uma excepção, que impede qualquer lógica na view de ser processada. Caso o acesso seja válido, a execução segue para decoradores posteriores ou a view.

Excerto de código 4.5: Aplicação de *view decorators*

```
1
2 @login_required
3 @user_complete_profile_required
4 @user_approval_required
5 @user_has_access_to_contest
6 def example_view(request):
7     ...
8   return render(request, template_name, context)
```

Como podemos verificar no excerto de código 4.5, podemos aplicar vários decoradores a uma *view*, que serão executados sequencialmente, formando uma cadeia de acessos necessários para aceder à *view*. Visto que são executados sequencialmente, e dado que as excepções lançadas não são apanhadas por nenhum dos decoradores subsequentes nem a *view*, basta não cumprir uma das condições necessárias para o acesso à *view* ser negado. A utilização destes decoradores no projecto tornou-se frequente, pois, na nossa opinião, é uma solução simples e elegante para o problema de controlo de acessos.

Modelos e Classes

A transição do UML conceptual para o código, a nível de modelos, foi rápida, tendo apenas de editar as propriedades de cada modelo, bem como criar os modelos em falta. Em Django os modelos são na verdade classes que derivam da classe Model, disponível no pacote *models* do Django, conforme se pode ver no seguinte excerto de código:

Excerto de código 4.6: Classe *Attempt* (simplificada)

```
1 class Attempt(models.Model):
2     contest = models.ForeignKey(Contest)
3     user = models.ForeignKey(User)
4     team = models.ForeignKey(Team)
5     date = models.DateTimeField()
6     submitted_file = models.FileField(null=False)
7     auto_generated = models.BooleanField(default=False)
8     finished_processing = models.BooleanField(default=False)
```

A investigação inicial que fizemos sobre o Django não nos permitiu perceber de imediato a potencialidade desta classe. Na verdade esta classe permite-nos definir o tipo de dados, as relações (através de chaves estrangeiras e outras técnicas), e propriedades de base de dados para cada atributo, conforme o exemplo anterior.

Grande parte do código referente às classes e modelos não estava encapsulado e encontrava-se em funções isoladas, o que torna o código pouco legível e susceptível a *bugs*. Por exemplo, se fosse necessário saber o *ranking* de uma equipa num determinado contest, seria chamada uma função semelhante a *get_ranking_for_team(team)*, onde por argumento seria passada a equipa. De modo a optimizarmos o código, todas as funções referentes a modelos foram passadas para dentro destes. Assim, invocações deste estilo passariam a ser efectuadas directamente na variável que contêm a instância desse modelo, passando a ser algo semelhante a: *team.get_ranking()*. Para funções que se revelaram estáticas, nas quais não existe instância do modelo, foi utilizada a anotação *@classmethod*, resultando em invocações análogas a

Team.get_by_id(id), onde, em vez de referenciarmos a variável que contém a instância, é referenciado o modelo/classe.

Excerto de código 4.7: Código exemplo de um modelo

```
class ExampleModel(models.Model):
    name = models.CharField()
    users = models.ManyToManyField(User)

    def get_name(self):
        return self.name

    def get_users(self):
        return self.users.all()

    def has_user(self, user):
        return self.users.filter(id=user.id).exists()

    @classmethod
    def get_by_id(cls, id):
        return cls.objects.get(id=id)
```

Os modelos também permitem produzir *queries* orientadas a objectos. Por exemplo:

Excerto de código 4.8: Exemplo de uma *query* no Django

```
n_mandatory_passed = self.getClassifications().filter(passed=
    True, test__mandatory=True).count()
```

Que em linguagem SQL ficaria:

Excerto de código 4.9: Tradução da *query* para SQL

```
SELECT COUNT(c.id) FROM classifications c, tests t WHERE c.
    passed=1 AND t.mandatory=1 AND c.test_id = t.id
```

Sendo os *models*, tal como já dissemos, classes, podem ter também métodos, exemplificado acima, que também utilizamos para abstrair este tipo de *queries* e nos retornar o resultado que necessitamos de modo mais directo. Por exemplo, e seguindo os exemplos anteriores:

Excerto de código 4.10: Exemplo de método e sua chamada

```
# Metodo
def get_all_passed_tests_count(self):
    all_tests = self.get_classifications().filter(passed=True
    ).count()
    mandatory = self.get_classifications().filter(passed=True
    , test__mandatory=True).count()
    non_mandatory = self.get_classifications().filter(passed=
    True, test__mandatory=False).count()
    return all_tests, mandatory, non_mandatory
# Chamada do metodo
classification = Classification.get_by_id(1)
```

```
9 passed, mandatory, non_mandatory = classification.
    get_all_passed_tests_count()
```

Que poderia ser simplificado tendo em conta programação orientada a objectos, ignorando, por questões de exemplificação, a optimização de calcular os testes obrigatórios passados através da subtracção dos não passados ao total de passados e vice-versa, para algo semelhante a:

Excerto de código 4.11: Optimização do código na figura 4.8

```
1 def get_all_passed_tests_count(self):
2     all_tests = self.get_passed_tests_count()
3     mandatory = self.get_passed_mandatory_tests_count()
4     non_mandatory = self.get_passed_non_mandatory_tests_count
    ()
5     return all_tests, mandatory, non_mandatory
```

Como podemos ver neste excerto de código, desta forma podemos programar e obter, o número de testes total passados o número de testes obrigatórios passados e o número de testes não obrigatórios passados, para que com isso, por exemplo, calcularmos a nota final da *Attempt* que criou a *Classification* #1.

Foi assim possível encapsular o código dentro dos respectivos modelos/classes, aproveitando também em alguns casos para realçar a segurança e robustez destes métodos, visto que agora, encapsulados, podiam facilmente tirar proveito de outros métodos pertencentes à mesma classe.

4.2.2 Introdução do Docker à Pandora

Configuração e Criação de Imagens

Como mencionado anteriormente, o Docker irá ser utilizado na Pandora para construir imagens, efectivamente ambientes virtuais para cada linguagem de programação, e inicializar instâncias (*containers*) das mesmas para cada submissão. A construção das imagens é feita através de um Dockerfile. Um Dockerfile é simplesmente um conjunto de instruções que é usado para criar uma imagem. Quando o *Dockerfile* é executado, é construída uma imagem com base nas instruções fornecidas pelo mesmo. Essa imagem é depois utilizada para inicializar *containers*, garantindo que todos os *containers* inicializados através dessa imagem têm um ambiente de execução igual.

Excerto de código 4.12: Dockerfile para C (simplificado)

```
1 FROM gcc:latest
2 COPY time /usr/bin/time
3 COPY c_test_script.sh /usr/bin/c_test_script.sh
4 RUN mkdir /usr/src/compiled/
```

Por exemplo, este *Dockerfile* toma como base a versão mais recente da imagem docker *gcc*, que como o nome indica, inclui o GCC. Também copia dois ficheiros, «time» e «c_test_script.sh», para dentro da directoria «/usr/bin», sendo o primeiro um programa para medir o tempo de execução (e outros parâmetros) do código submetido e o segundo o *script* configurado para compilar o código, quando necessário, e correr os testes. Por fim, cria a directoria «/usr/src/compiled/», onde será colocado o código do aluno para execução e compilação. Estes comandos em específico são executados na construção da imagem (existem outros que são executados no momento de inicialização do *container*), o que leva a que todos os *containers* inicializados a partir desta imagem sejam portadores destes ficheiros e directorias.

Excerto de código 4.13: Construção da Imagem

```
1 docker build -t docker_c .
```

Este comando faz a construção da imagem, com base no *Dockerfile*, onde o parâmetro -t (proveniente de *tag*) serve para referenciarmos a imagem através de um nome, neste caso «docker_c», e o «.» (ponto) referência a directoria onde está o *Dockerfile*. Esta *tag* é fixa para cada linguagem, sendo referenciada no código para o sistema saber em qual imagem, consoante a linguagem do *contest*, se deve basear para a inicialização do *container*.

Inicialização de *Containers*

Para inicializarmos um novo *container*, executamos o seguinte comando:

Excerto de código 4.14: Criação do *container* (simplificado)

```
1 docker run --rm -i -d --name attempt1 -v ./:/disco docker_c
    sleep 600
```

Este comando tem várias funções, explicadas abaixo:

- O comando «docker run» inicializa o *container*;
- A *flag* «–rm», de *remove*, é efectivamente uma forma *cleanup*, removendo o *container* assim que a sua execução terminar;
- A *flag* «-d», de *detached*, evita que o *container* ligue o seu «STDIN», «STDOUT», e «STDERR» à consola, para evitar que a chamada deste comando seja bloqueante na Pandora;
- A *flag* «-i», de *interactive*, mantêm o «STDIN» do *container* aberto, o que permite interagir com este através do envio de comandos;
- A *flag* «–name» atribui o parâmetro passado ao *container*, neste caso «attempt1», sendo possível no futuro identificá-lo pelo mesmo;
- A *flag* «-v» permite efectuar o *mount* de um volume para ser acessível dentro do *container*;
- Por fim, o comando «sleep 600» adormece o *container* por 600 segundos, assim que este é criado, evitando que este termine a sua execução após a sua inicialização dado que o *container* em si não tem processos para executar no momento em que é inicializado.

A junção de todas estas opções permite à Pandora inicializar um *container*, de forma não bloqueante, com uma rotina de *cleanup* automática aplicada no fim da sua execução, receptível a comandos.

Para a execução dos testes, enviamos os comandos para o *container*, do seguinte modo:

Excerto de código 4.15: Envio de comando para um *container*

```
1 docker exec -i attempt1 c_test_script.sh
```

Um comando simples, envia a instrução «c_test_script.sh» (ficheiro previamente colocado na directoria «/usr/bin/» durante a construção da imagem) para o *container* com nome «attempt1», de modo interactivo devido à *flag* «-i». A este comando devem ser também acrescentados os parâmetros do *script* em questão, como por exemplo o identificador do teste a executar, que seriam depois processados por este. Faz-se notar aqui a ausência da *flag* «-d», para tornar a execução deste comando bloqueante, cuja importância será explicada na secção seguinte.

Comunicação com o Celery

O Celery é uma ferramenta *open-source* onde, ela própria, é uma *queue* (fila) de tarefas com foco no processamento em tempo real, mas que permite também o seu agendamento.

A adopção do Celery para a Pandora foi pensada devido à concorrência de processos, nomeadamente as submissões. É possível atingir simultaneidade no processamento das submissões, relegando estas tarefas para o Celery, onde este as gere e executa de acordo com as definições implementadas. O Celery já se encontrava integrado na Pandora no início da realização deste trabalho. No entanto, devido à reestruturação da forma como seriam processadas as submissões em função da inclusão do Docker, foi necessário estudarmos o funcionamento da mesma e fazer uma nova implementação da mesma.

Assim, o Celery é uma ferramenta essencial na Pandora, sendo que todos os processos deste tipo são encaminhados para esta. No Django existe uma biblioteca para Celery, onde se define uma função como sendo uma *Task*, que, no momento de inicialização do Celery, é registada como uma tarefa, e pode ser chamada em qualquer momento. A sua chamada é feita via um protocolo especifico do mesmo, e dá inicio ao processamento dessa tarefa. Visto que a tarefa é agora tratada pelo Celery, podemos usufruir do sistema de *workers* do mesmo, que permite paralelizar as tarefas, executando várias em simultâneo. Definindo como tarefa a criação e execução do *container* responsável pela execução do código submetido e a avaliação da submissão, alcançamos a simultaneidade do tratamento de submissões, efectivamente sendo possível avaliar várias submissões em simultâneo quando necessário, até um limite definido na configuração do Celery.

O processamento, tanto da compilação (em linguagens que necessitem), como da execução dos testes ao código que os alunos submetem, pode ser um processo demorado, por diversos motivos, e sendo o Django a renderizar o *front-end* e a processar também todo o *back-end*, a demora em qualquer processo seria um problema, pois a renderização do template para ser apresentado no *front-end* ficaria bloqueado à espera do fim da sua execução, caso este tratamento de submissões estivesse implementado na *view* resultante da submissão. Não sendo o caso, visto que é o Celery que trata das submissões através das tarefas anteriormente descritas, o bloqueio da lógica de negócio na *view* não é um problema. No entanto, apresentar *feedback* relativo ao estado da submissão aos utilizadores é essencial para uma boa *user experience*. Para apresentar o estado da submissão aos utilizadores enquanto esta é processada, necessitaríamos de desenvolver um *endpoint* que poderia ser chamado para obter o estado actual da submissão. No entanto, o Celery já inclui esta funcionalidade, onde é possível atribuir um estado à tarefa dentro do código da função que foi marcada como tal, sendo apenas necessário consultar o estado do mesmo através de um pequeno *script*, escrito em Javascript, que consulta o estado periodicamente.

Excerto de código 4.16: *Script* para consulta do estado da submissão

```
<script type="text/javascript">
  function processProgress(progressBarElement,
   progressBarMessageElement, progress) {
        // Apresenta o progresso numa progress bar
    }
  function processResult(resultElement, result) {
    // Redireciona a pagina atual para a pagina de resultados
  }
  $(function () {
    var progressUrl = "{% url 'celery_progress:task_status'
   task_id %}";
    CeleryProgressBar.initProgressBar(progressUrl, {
      onProgress: processProgress,
      onResult: processResult,
    })
  });
</script>
```

Um dos desafios durante o processo de inclusão do Docker na Pandora foi garantir que seria possível obter *feedback* do estado das submissões agora que seriam processadas num *container* próprio, visto que a comunicação a que temos acesso é através da invocação de métodos oferecidos pelo pacote do Celery para Django, invocados na função definida como tarefa, sem comunicação directa com o *container*. No entanto, como o comando que executa o *container* está incluído na função, e tendo comportamento bloqueante (assíncrono), teríamos pelo menos acesso ao momento em que o comando terminou a sua execução.

No início, enquanto nos familiarizávamos com o Docker, optámos por executar um *container* por teste, o que rapidamente se verificou demasiado demorado, devido ao tempo necessário para a inicialização de cada *container*, e ineficiente, visto que estavam a ser gastos recursos a fazer o *setup* do *container* desnecessariamente. No entanto o *feedback* era possível, pois após cada teste, poderíamos actualizar o estado da submissão.

Trocámos então, tendo agora em mente a eficiência e rapidez, para um modelo onde todos os testes eram executados dentro de um só *container*, e era dada a instrução para executar todos os testes no momento da sua inicialização, só voltando a ter feedback destes quando o *container* terminasse de executar todos os testes. Isto revelou-se novamente inviável, pois não iríamos conseguir desenvolver uma forma, em tempo útil, do *container* comunicar com o Celery e dar feedback dos testes à medida que eram executados.

Após mais alguma investigação na documentação do Docker, finalmente chegámos à solução apresentada anteriormente. Segundo a documentação do Docker, é possível enviar comandos para um *container* em execução, através do comando *docker exec ¡tag¿ ¡command¿*. Assim, resolvemos inicializar o *container* e de seguida adormecê-lo, sem que executasse nenhuma função. Depois, dentro da lógica da tarefa, seriam dadas instruções ao *container* através do comando supra-mencionado, sequencialmente, para cada teste. Como já foi descrito, a acção de enviar o comando para o *container* é bloqueante devido à ausência da *flag* «-d», conseguindo acesso ao momento de término de cada teste, e consequentemente o estado da submissão.

Processamento dentro do *container*

A execução do código dentro do *container* é feita através de um *Shell script*, composto principalmente por comandos para aumentar a segurança da execução e identificação de possíveis erros, através do *trapping* de códigos de erro dos diferentes processos de modo a identificar as falhas no processamento, como erros de compilação, excepções durante a execução de código, limites de recursos excedidos, entre outros, sendo assim possível prestar o *feedback* correcto do erro ao utilizador.. Estas informações, juntamente com o resultado dos testes quando estes não quebram nenhuma das restrições lhes impostas, são gravadas em disco no servidor, para processamento posterior pela tarefa.

O *script* recebe parâmetros configurados de acordo com as propriedades especificadas nas subclasses da classe *Specification*, e também outros que identificam qual a submissão a processar e teste a realizar. É colocado um *script* dentro de cada imagem, sendo que existe uma imagem por linguagem, e cada script é adaptado de acordo com a linguagem em questão. Por exemplo, em linguagens compiladas, se enviarmos no parâmetro «test_id» um 0, o script irá compilar o programa. Para linguagens interpretadas, basta editar o *script* de modo a ignorar esse caso, ou então não enviar a instrução de todo. Na figura seguinte, temos um exemplo gráfico da comunicação, indirecta, do Celery com o Docker, com os respectivos comandos (simplificados).

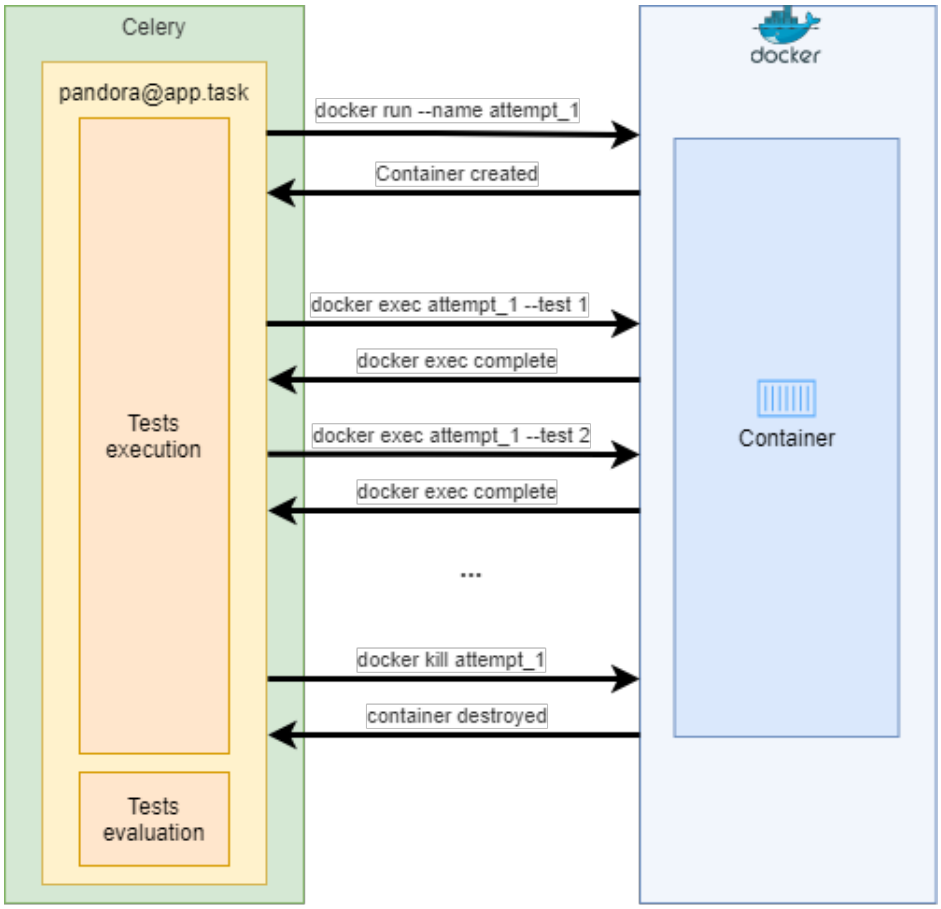

Figura 4.5: Comunicação Celery-Docker

4.2.3 Novas linguagens

Para não ficarmos apenas pela teoria, e testarmos se realmente a reestruturação da Pandora atingiu o seu objectivo referente à implementação de novas linguagens, decidimos implementar algumas.

Java

Como primeira linguagem adicional decidimos implementar Java (Java 8), tanto por ser uma linguagem muito utilizada no ensino de programação, mas também por ser uma linguagem com que estamos algo familiarizados.

Foi necessário, antes de qualquer outra intervenção, a criação de uma classe/modelo *Java_Specification* para representar as configurações para a linguagem. Isto seguiu-se da criação de uma nova imagem Docker, baseada na imagem «openjdk», disponível no dockerhub[2], com instruções de criação muito semelhantes à imagem para C, copiando para dentro desta, no momento da sua criação, o *script* necessário para compilação de código e execução de testes, entre outros ficheiros.

O *script* foi configurado tendo em mente agora a nova linguagem, fazendo o *trapping* dos *exit codes* dos vários processos nos momentos acertados de modo a detectar erros de compilação e excepções não apanhadas causadoras do término da execução do programa submetido.

A implementação de Java revelou-se simples e realizou-se de acordo com as expectativas. A abstracção do tipo de linguagem utilizado na rotina de processamento de submissões e subsequente simplificação desta rotina para apenas inicializar um *container* e enviar o comando de execução de cada teste, fornecendo ao mesmo comando como parâmetro o identificador do teste, da submissão, e do contest, bem como as propriedades presentes na classe derivada da *Specification*, revelou-se bastante útil, visto que foram apenas necessárias actualizações ligeiras ao código para obter uma versão inicial, mas funcional, de compatibilidade com Java na Pandora.

4.2.4 *User Interface* e *User Experience*

User Interface

Aproveitando o facto da Pandora assentar na biblioteca Bootstrap[8], nomeadamente a versão 4, e estarmos também algo familiarizados com o funcionamento da mesma, decidimos criar uma UI com base no *theme* SB Admin 2[1]. Deste modo, conseguimos reaproveitar grande parte do código preexistente e, ao mesmo tempo, revitalizar a interface com um template moderno, especificamente desenvolvido para utilização com Bootstrap.

Este *theme* inclui duas ferramentas que ajudaram bastante não só na construção do novo UI mas também no desenvolvimento de algumas funcionalidades essenciais para a administração da Pandora - DataTables[7], um *plugin* para o jQuery[3] que permite visualizar e interagir com tabelas HTML de forma dinâmica, e Chart.js[9], que permite criar gráficos para interpretação facilitada de dados.

User Experience

A *user experience* foi melhorada de forma passiva através do desenvolvimento do novo UI. Aspectos como o realçar da página ou secção actual na *sidebar* de navegação, *responsiveness* da aplicação, ou posicionamento acertado de componentes para fácil visualização foram todos melhorados durante o desenvolvimento do novo UI.

4.2.5 Nova Área de Utilizador

Para a área de utilizador aproveitamos alguns componentes que já tinham sido construídos, e seguimos aproximadamente a mesma lógica de navegação que já estava implementada, mas onde havia espaço para muitas melhorias e implementação de novas funcionalidades.

Menu de navegação

Decidimos, no sentido de melhorar e simplificar a utilização da Pandora, colocar apenas os seguintes menus, na navegação principal da Pandora:

- *Dashboard*
- *Contests*
- *Groups*
- *About*

Dashboard

Este ecrã destina-se exclusivamente a apresentar os dados das submissões e *contests* aos quais o utilizador está associado. Em modo *card*, ou gráfico, utilizador pode visualizar o número de *contests* activos, a média da sua nota global (apenas relativa às submissões feitas na Pandora), a média do *ranking* global da Pandora, o número médio de submissões por *contest*, o tempo restante para cada *contest*, e a última nota obtida em cada *contest* que participou.

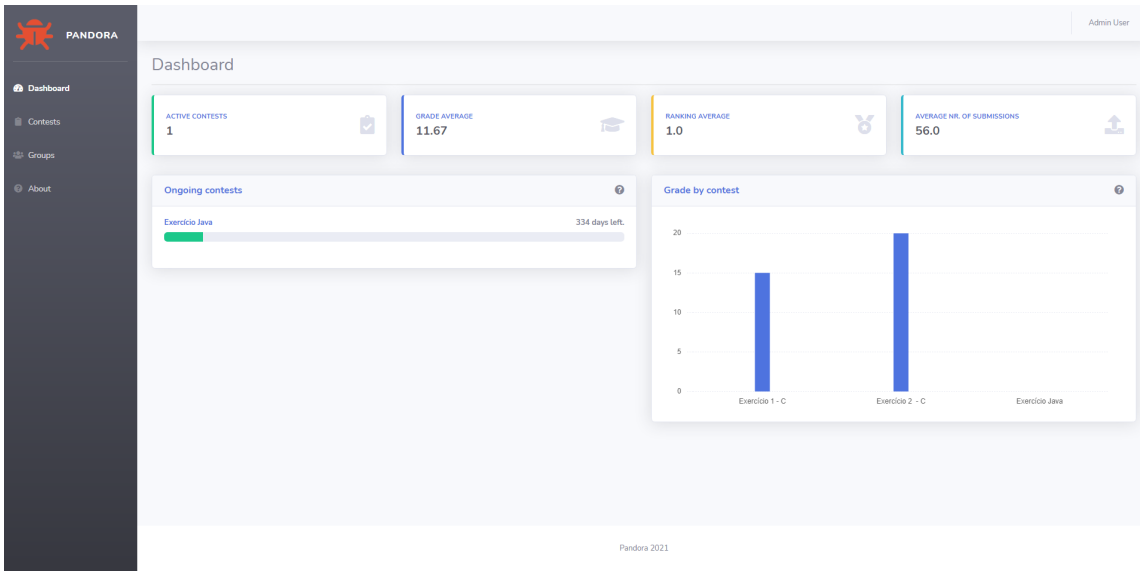

Figura 4.6: Área de utilizador - *Dashboard*

Contests

Este conjunto de ecrãs são os principais em toda a área de utilizador. No topo da hierarquia destes ecrãs encontra-se a lista de *contests* (Figura 4.7), onde o utilizador pode visualizar os *contests*, tanto a decorrer como terminados, alocados aos grupos aos quais pertence.

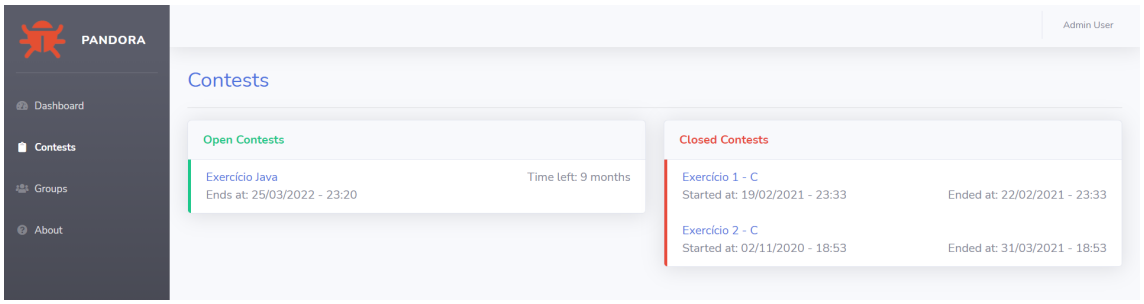

Figura 4.7: Área de utilizador - Lista de *contests*

Acedendo à pagina do *contest* pretendido (Figura 4.8), é lhe apresentado, à semelhança do *dashboard* anteriormente referido, um ecrã do tipo *masonry*, com *cards* de informação relativa ao respectivo *contest*. Informação essa:

- Os detalhes do *contest* - Linguagem, *deadline*, Número máximo de elementos por grupo, etc. (Figura 4.8)
- O estado actual - classificação, número de submissões, etc. (Figura 4.8)
- O nome da sua equipa e respectivos membros. (Figura 4.9)
- A sua posição no ranking desse *contest*. (Figura 4.9)
- Um gráfico com o progresso da sua nota. (Figura 4.9)
- Uma tabela com o histórico das suas submissões. (Figura 4.9)

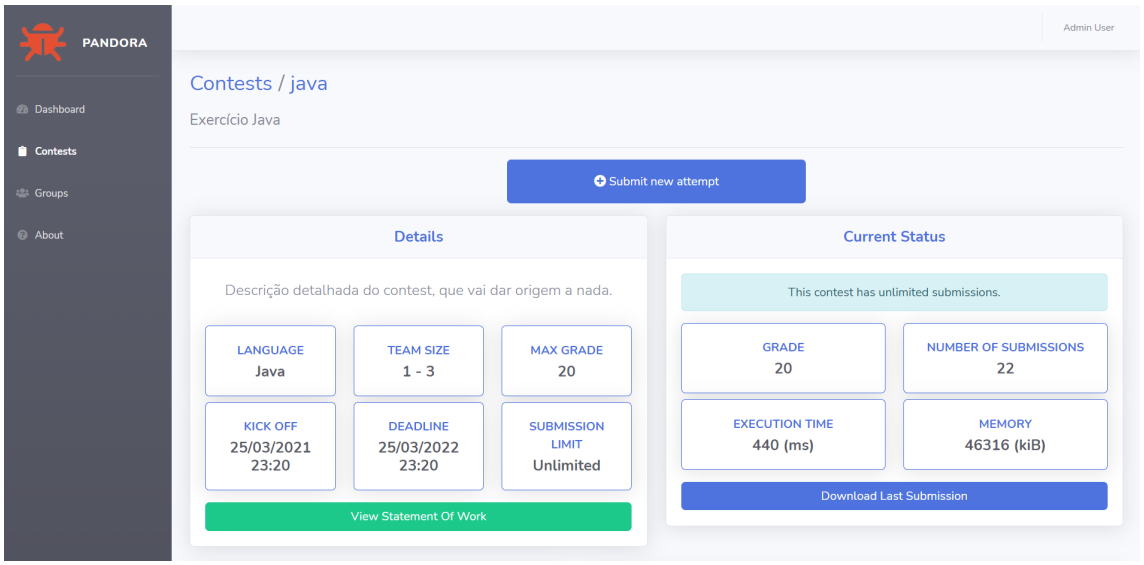

Figura 4.8: Área de utilizador - Detalhes de um *contest*

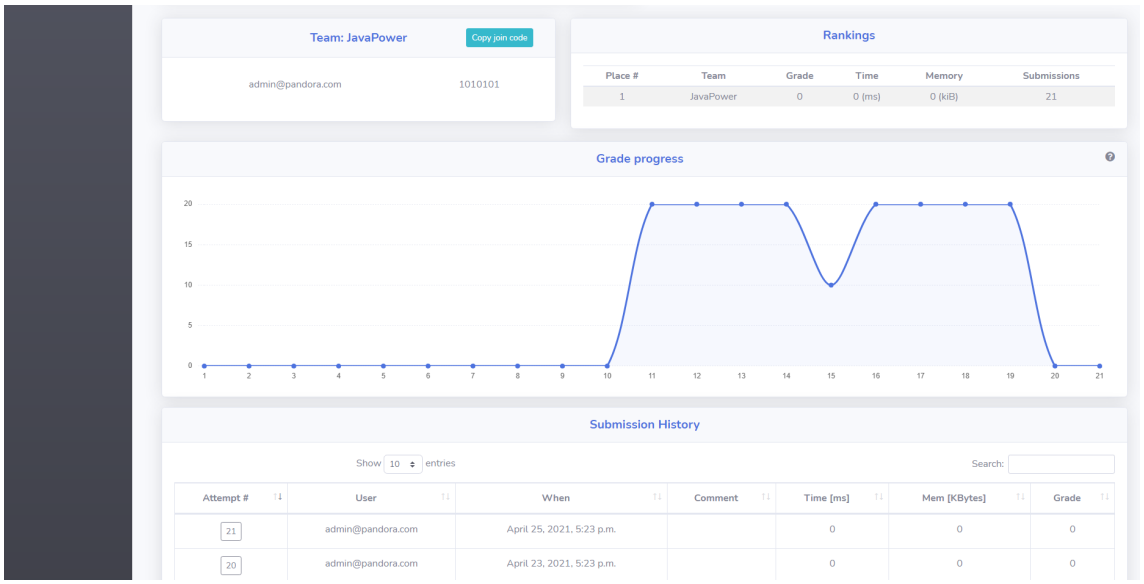

Figura 4.9: Área de utilizador - Detalhes de um *contest* (cont.)

Como é possível observar também na Figura 4.8, é no topo deste ecrã que se encontra o botão que permite ao utilizador enviar uma submissão.

Submissão

Dividimos esta secção da Pandora em três partes:

- Formulário de submissão do código.

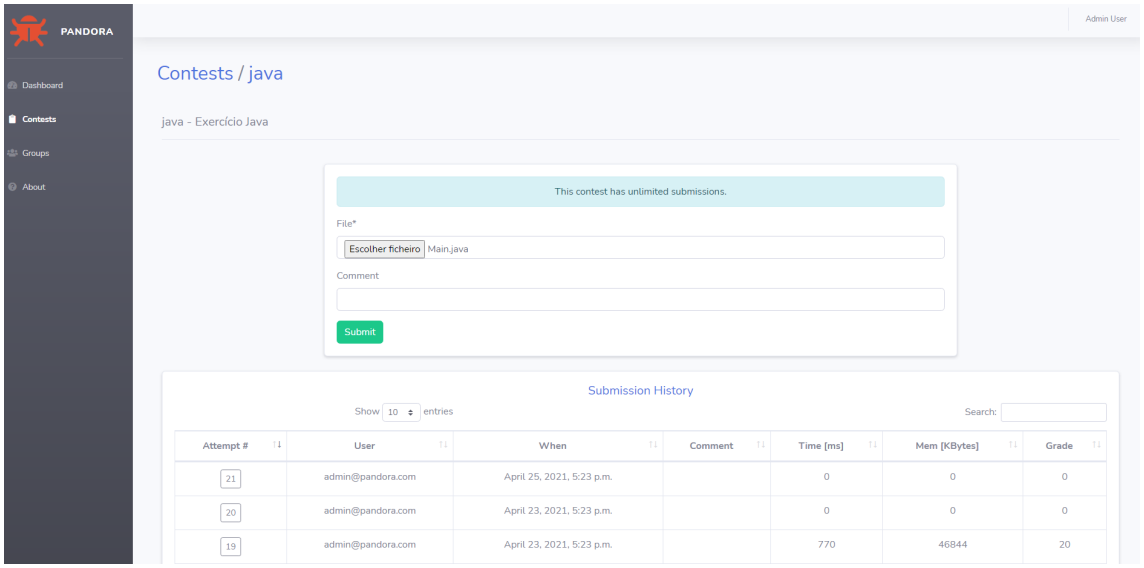

Figura 4.10: Área de utilizador - Formulário de submissão

Neste ecrã, o utilizador tem à sua disposição tanto o formulário de submissão, como a tabela com as suas últimas submissões.

- Página de feedback da execução dos testes

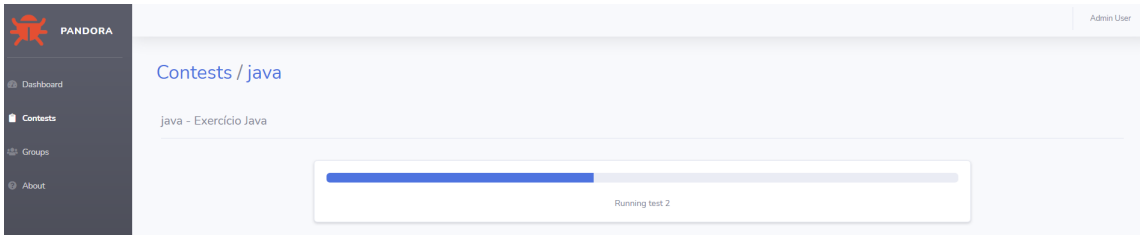

Figura 4.11: Área de utilizador - *Feedback* da execução dos testes

Depois do código submetido, é apresentado ao utilizador uma *progress bar* que indica o progresso da execução dos testes feitos ao código do mesmo.

- Resultados da tentativa

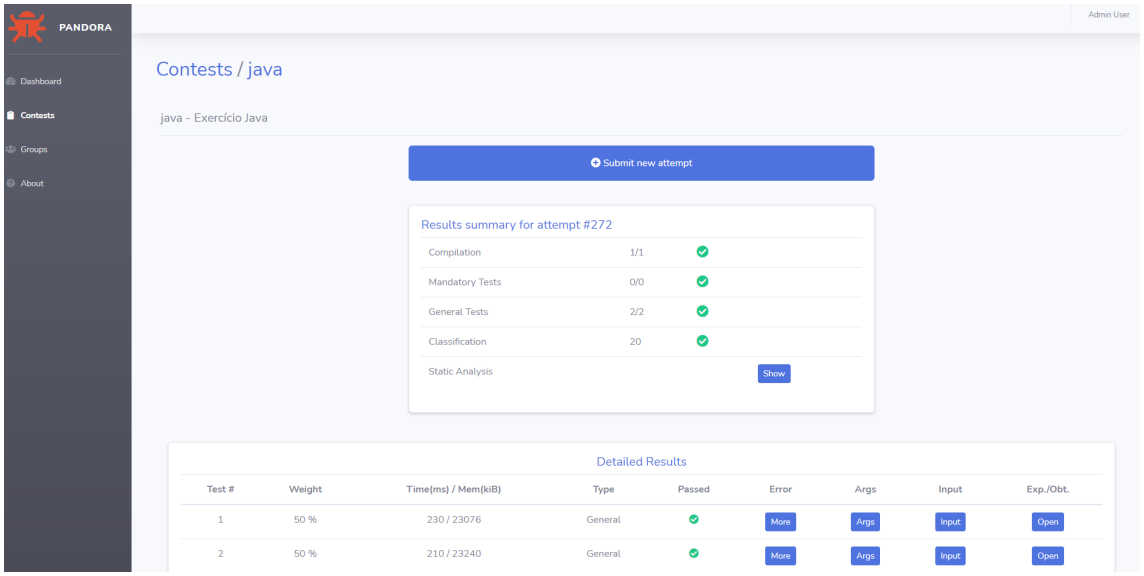

Figura 4.12: Área de utilizador - Resultados da execução dos testes

No fim da execução de todos os testes, são apresentados ao utilizador os resultados dos mesmos, como se passou na compilação do código, o número de testes passados, qual a sua avaliação final, e os detalhes de cada teste executado. Tem também disponível, mediante configuração, para cada teste, o erro (caso exista), os argumentos utilizados na sua execução, o input utilizado e o output esperado e real.

Groups

Esta é apenas uma área onde são apresentados, ao utilizador, os grupos aos quais está associado.

About

Área reaviva à apresentação da Pandora ao utilizador, com uma breve descrição.

Profile

Nesta área, o utilizador pode alterar os seus dados de perfil - Primeiro e último nome, e o número de aluno.

4.2.6 Nova Área de Administração

No desenvolvimento inicial da Pandora, o menu de administração estava conjunto com o de utilizador, obviamente com as devidas regras internas que apenas permitiam a um utilizador de administração ver essas opções no menu, o que o tornava bastante confuso e pouco prático na sua utilização. Posto isto, e conforme já referido nos capítulos anteriores, decidimos dividir a Pandora em duas áreas distintas - Utilizador e Administração.

Menu de navegação

Na navegação principal da área de administração, o administrador tem à sua disposição os seguinte menus:

- *Dashboard*
- *Contests*
- *Groups*
- *Users*

Dashboard

À semelhança do *dashboard* da área de utilizador, é apresentado ao administrador um conjunto de *cards* e gráficos com o número de utilizadores e *contests* activos, o número de submissões da última semana (7 dias), um gráfico de barras com o número de submissões de cada *contest*, e por fim um gráfico com a média das notas globais por *contest*.

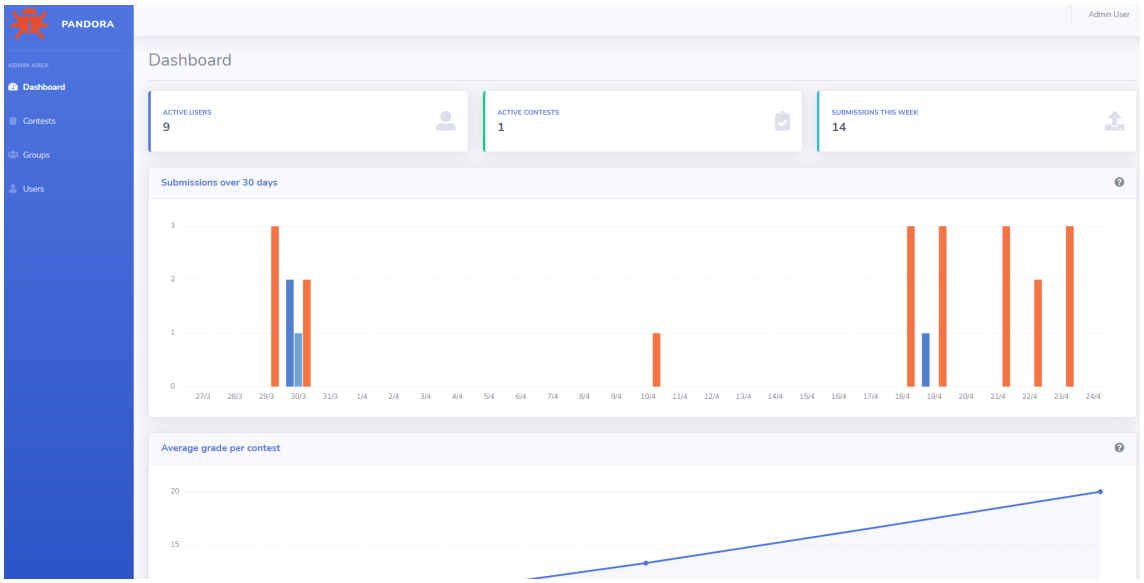

Figura 4.13: Área de administração - *Dashboard*

Contests

O administrador nesta área tem à sua disposição todos os *contests* criados na Pandora, assim como também pode adicionar um novo. Não só pode editar as configurações de cada um (individualmente), como também adicionar e/ou editar testes, ver e/ou editar as equipas, ver os utilizadores associados ao mesmo, e ver todas as submissões tanto em gráfico como em tabela. Esta última dá a possibilidade de apresentar também os resultados e detalhes de cada uma das submissões feitas.

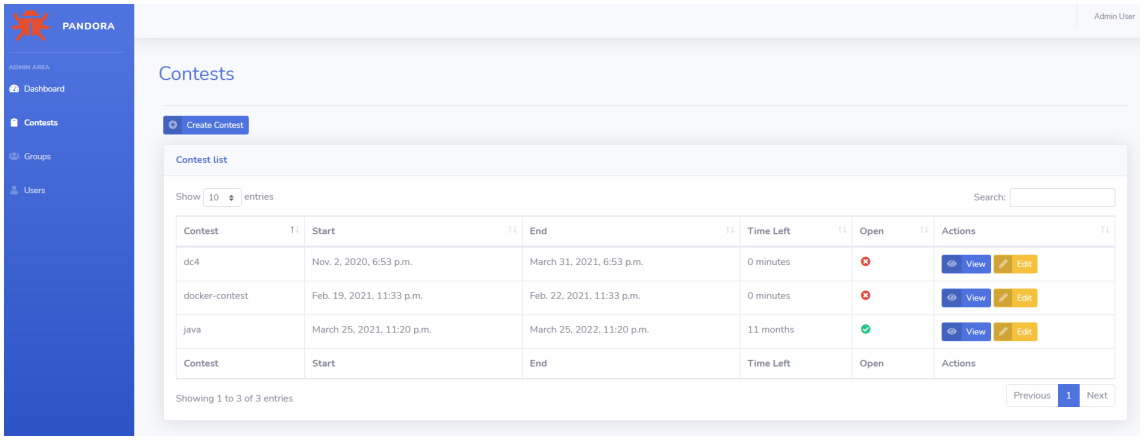

Figura 4.14: Área de administração - Lista de *contests*

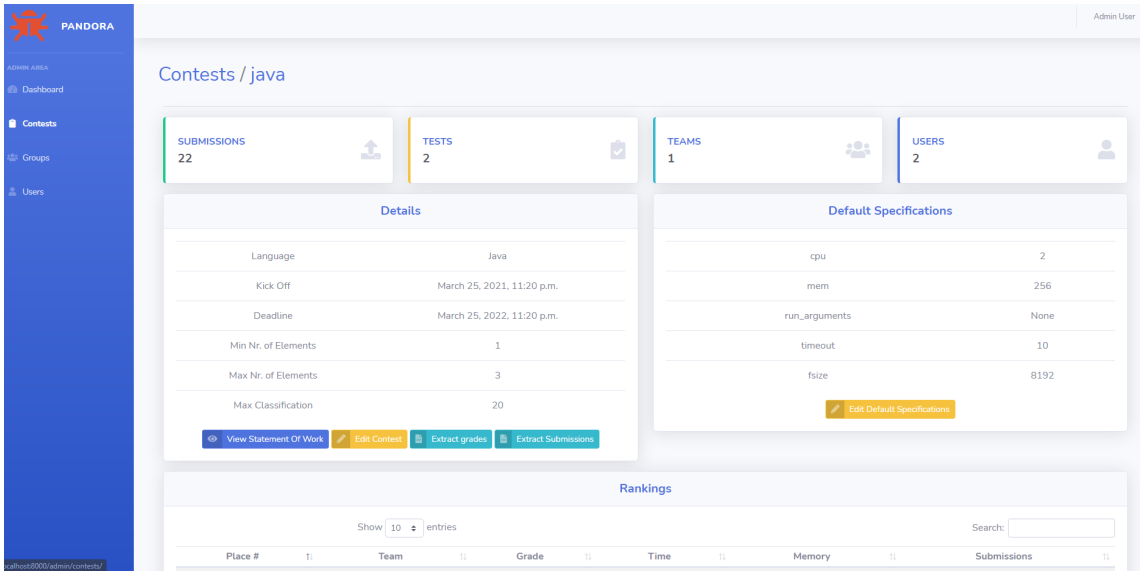

Figura 4.15: Área de administração - Detalhes de um *contest*

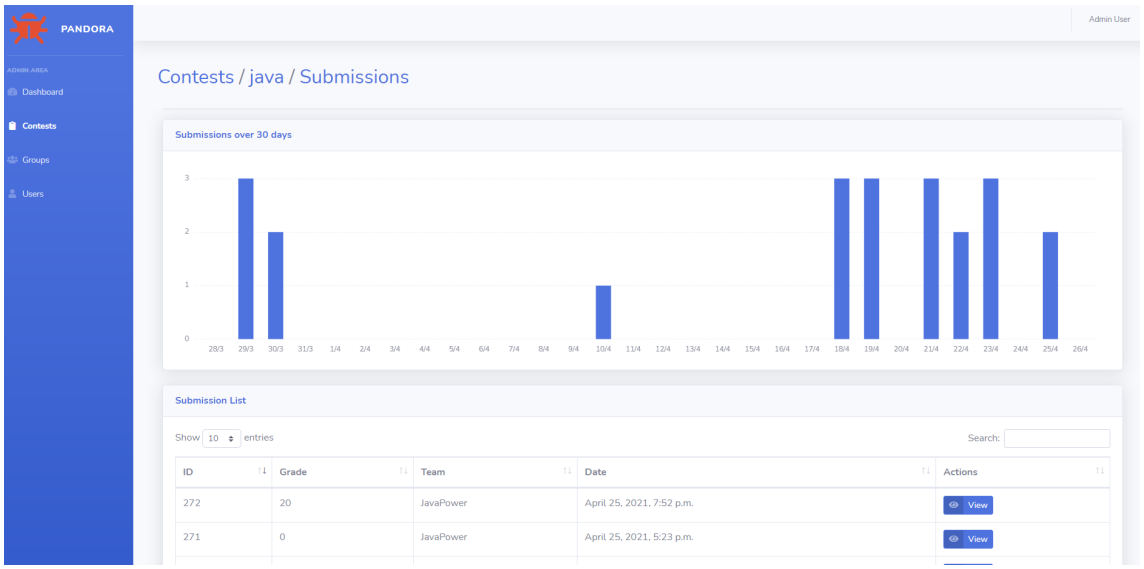

Figura 4.16: Área de administração - Submissões de um *contest*

Groups

É nesta área que o administrador pode criar novos grupos, ver e/ou editar os que já estão criados, e associar utilizadores e/ou *contests* ao mesmo. Ao clicar num grupo, tem também à disposição um pequeno *dashboard* apenas com *cards* onde lhe é mostrado quantos utilizadores e *contests* estão associados.

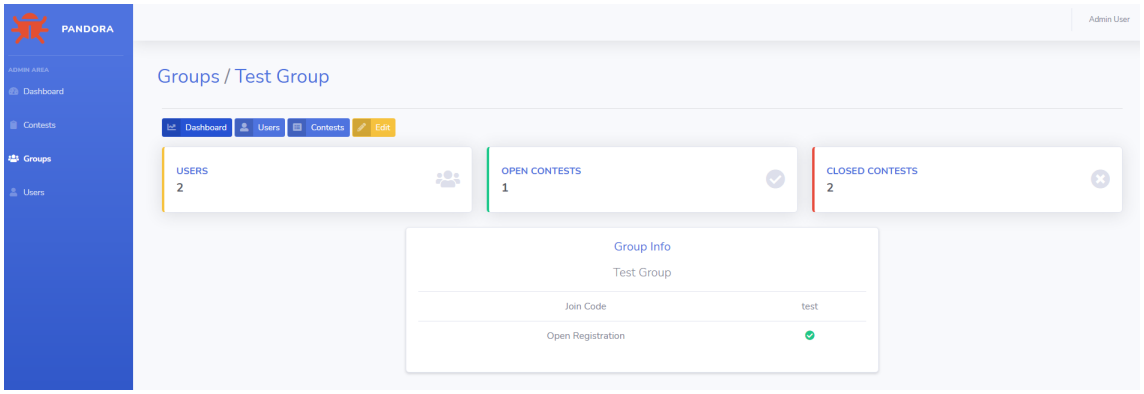

Figura 4.17: Área de administração - Detalhes de um grupo

Users

Nesta área o administrador pode criar e/ou editar (individualmente) cada utilizador, podendo também, em volume, validar ou invalidar o mesmos através da selecção de *checkboxes*.

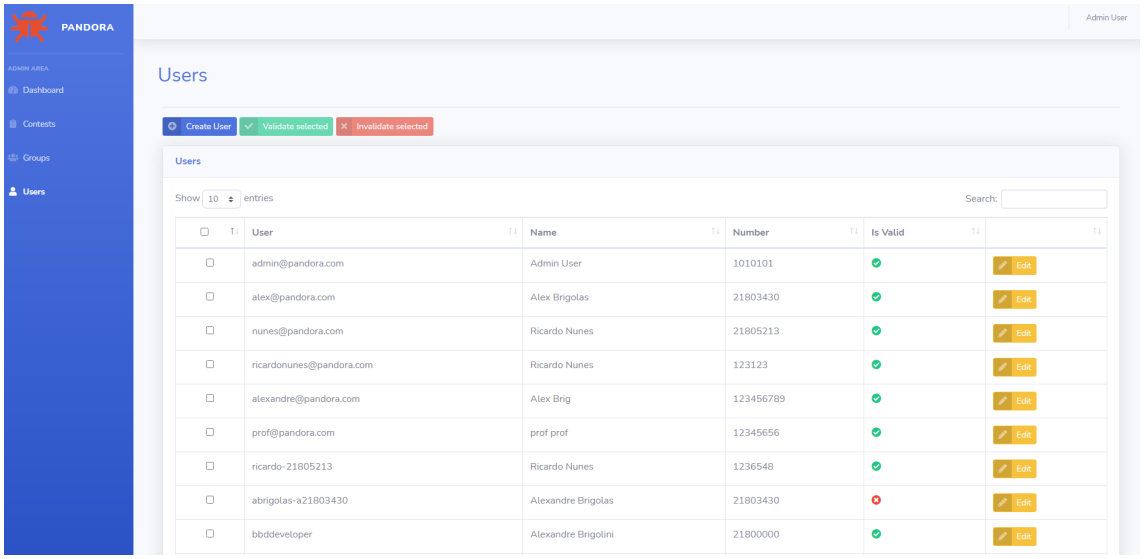

Figura 4.18: Área de administração - Lista de utilizadores

4.2.7 Licenças

Todas as adições efectuadas no desenvolvimento desta solução não produzidas por nós, nomeadamente código desenvolvido por terceiros na forma de pacotes, *scripts*, *templates*, ou outro formato, foram escolhidas com licenças compatíveis com a licença *GNU Public License v3 (GPLv3)*. Para tal, consideraram-se adições com licenças *MIT* ou a própria *GPLv3*[4].

4.3 *Deployment*

A implementação da solução foi feita em duas fases. Inicialmente, pouco depois do início do segundo semestre, foram colocadas em produção as novas áreas e as melhorias à UI juntamente com a aplicação reestruturada a nível de organização de código e arquitectura de modelos, utilizando já o Docker para inicializar os *containers* para processamento de submissões. Apresentando no começo do *deploy* alguns problemas devido à incompatibilidade de algumas alterações com a estrutura da base de dados em produção, que foram rapidamente resolvidos com a ajuda do orientador. Esta primeira actualização ao *software* em produção foi crucial para nos apercebermos de vários problemas presentes nesta versão da Pandora, tanto reportados pelos utilizadores (incluindo o orientador), como detectados por nós. A demora no tempo de processamento de submissões (nesta versão o processamento de submissões utilizava um *container* por teste e não por submissão), *shutdowns* do servidor devido à falta de recursos, elementos do UI e funcionalidades em falta foram alguns dos erros dos quais só nos apercebemos em ambiente de produção.

Numa segunda fase foram feitas várias actualizações mais pequenas, orientadas a corrigir os *bugs* causados pela actualização inicial, rectificar embaraços na natureza de como a Pandora utilizaria o Docker, implementar algumas melhorias obtidas através de feedback dos alunos ao orientador, bem como adicionar alguns requisitos "soltos"que se encontravam em falta. Não estando livre de ocasionais *bugs*, esta fase revelou-se menos problemática na implementação, visto que grande parte das *breaking changes* já tinham sido efectuadas.

De modo geral, a implementação da nova versão da Pandora foi realizada com sucesso, em parte devido também ao facto da primeira versão, já com grande parte das alterações, ter sido colocada em produção no início do segundo semestre. Isto permitiu-nos recolher feedback dos alunos (que, de certa forma, também fizeram o papel de *testers*), detectarmos relativamente cedo alguns constrangimentos a nível estrutural, e obter também um novo ponto de vista de desenvolvimento que tem em conta os recursos disponíveis no servidor e o nível de concorrência que este pode alcançar.

Capítulo 5

Benchmarking

A tabela 5.1 apresenta a comparação de diferentes funcionalidades das AATs que considerámos mais relevantes para comparação com a Pandora. Na tabela, os qualificadores foram escolhidos tendo em conta os objectivos da Pandora, de modo a ser possível efectuar uma avaliação de onde é que a Pandora se encontra relativamente às outras AATs na concretização destes mesmos objectivos. A tabela não deve ser interpretada como comparação de usabilidade, viabilidade, ou valor das mesmas, visto que diferentes AATs, mesmo que desenvolvidas com o intuito de avaliar código, podem ter casos de uso diferentes, bem como outras funcionalidades aqui não descritas.

Tabela 5.1: Trade-off entre diferentes AATs.

	AAT			
Qualificadores	Pandora	DropProject	Mooshak	Repl.it
Permitir configurar vários níveis de visibilidade dos testes	Sim	Sim	Sim	Não
Permitir submissões em equipa	Sim	Sim	Limitado (o administrador é que cria as equipas)	Sim
Open-source	Não	Sim	Sim	Não
Permite testes unitários	Não	Sim	Não	Não
Permite testes input/output	Sim	Não	Sim	Sim
Mostrar *ranking*	Sim	Sim	Sim	Não
Linguagens compatíveis	C, Java	Java, Kotlin	C, C++, Java	C, Java, Kotlin, (50+).

Comparativamente ao estado da Pandora antes do desenvolvimento deste trabalho, a mesma assenta agora numa base sólida para a adição de múltiplas linguagens através da utilização do Docker e reestruturação do projecto. Acreditamos que com esta nova base, a inclusão de novas linguagens na Pandora seja de fácil realização, demonstrado em parte pela incorporação do Java, e esperemos que, na continuidade deste projecto, o mesmo venha a ser revelado. No entanto, ainda há trabalho a ser feito, evidenciado pela falta de testes unitários e, apesar das melhorias prestadas a nível de qualidade de código, atrito residual na transição para *open-source*.

Capítulo 6

Método e planeamento

A realização deste trabalho foi, desde o começo, planeada seguindo metodologias Agile, começando por analisar cada requisito, defini-lo, desenvolve-lo, testá-lo e finalmente colocá-lo em produção, o que nos permitiu um desenvolvimento rápido e eficiente dando-nos tempo para produzir todas as funcionalidades requeridas. Registámos também todos os requisitos num mapa Kanban, que juntamente com a disponibilização do código no Github nos permitiu, sem qualquer dificuldade, trabalharmos de uma forma independente mas ao mesmo tempo em grupo, dividindo de modo uniforme o trabalho pelos elementos do grupo, paralelizando e agilizando o desenvolvimento da solução. Em alturas onde o desenvolvimento de um requisito se revelasse complexo utilizou-se *pair-programming*, que se mostrou bastante útil, permitindo tanto um ponto de vista externo atento a detalhes por vezes pouco perceptíveis ao programador como também a sincronização dos elementos do grupo.

Grande parte das datas inicialmente definidas em calendário não foram cumpridas. A analise inicial e planeamento de entrega de requisitos foi feita tendo em conta o pouco ou nenhum conhecimento de algumas das ferramentas com que iríamos trabalhar, nomeadamente o Django, o Docker, e a própria Pandora. Isto levou a que, no desenvolvimento dos requisitos, detectássemos lacunas no nosso planeamento inicial, mais uma vez provindos da nossa ignorância referente à utilização destas ferramentas. Começámos então por reestruturar a calendarização de entrega dos requisitos, após discussão com o orientador, de modo a que trabalhássemos primeiro com os requisitos que reservam conhecimento sobre linguagens e ferramentas com as quais já estávamos familiarizados, como o desenvolvimento de uma nova UI e novas *views*, progressivamente transitando para requisitos mais complexos que incorporam tecnologias que desconhecíamos, sendo o melhor exemplo a inclusão da arquitectura multi-linguagem utilizando o Docker. Assim, a curva de aprendizagem necessária para o desenvolvimento de alguns requisitos foi bastante amortecida, dado que à medida que implementávamos os requisitos mais "fáceis", estávamos também a assimilar as outras ferramentas de modo gradual, evitando um possível período alargado de *downtime* no desenvolvimento, proveniente do estudo reservado a essas mesmas ferramentas.

Capítulo 7

Resultados

Os requisitos solicitados foram desenvolvidos, cumpridos e implementados em ambiente de produção com sucesso, utilizando, de certa forma, os alunos de Linguagens de Programação I como *testers*. Esta utilização e todo o *feedback* enviado por parte dos alunos permitiu-nos testar de um modo mais profundo a Pandora, num ambiente mais "real", aprimorando e contribuindo com maior eficácia para o sucesso da maior parte dos requisitos.

Foi também realizado um novo inquérito de satisfação, baseado nas questões presentes no inquérito do ano lectivo anterior, onde se observa, de um modo geral, respostas bastante positivas. Porém, alguns dos pontos são marcados por alguma controvérsia por parte dos alunos, tais como a motivação no desenvolvimento do projecto ser influenciada pela existência de um *ranking*, ou o número de submissões afectar o posicionamento do aluno/grupo no *ranking*, assim como os níveis de ansiedade dos alunos. Dos pontos positivos destacamos o sentimento de justiça na nota atribuída pela Pandora, ou esta estimular no aluno o gosto pela programação. Em anexo é possível observar em maior detalhe as questões colocadas aos alunos, assim como uma breve análise de cada uma.

Capítulo 8

Conclusão e trabalhos futuros

A envolvente de novas tecnologias, como Docker, assim como todas as dificuldades que encontrámos, que, para nós, foram consideradas e aceites sempre como desafios, contribuíram muito para a nossa aprendizagem e aprofundamento do nosso conhecimento, onde tivemos oportunidade de interagir com *software* em produção e experienciar alguns dos imprevistos e acontecimentos que só se observam em ambiente "real". Este projecto permitiu-nos tentar inovar e explorar novas soluções de forma autónoma, sempre algo enquadradas nos requisitos inicialmente propostos pelo professor orientador, que nos deu liberdade para tal.

Enquanto que consideramos que a solução desenvolvida contribui algo para a progressão da Pandora como AAT, existem ainda vários pontos a melhorar. Deixamos algumas sugestões para trabalhos futuros:

- Desenvolvimento de documentação.
- Implementação de novos métodos de autenticação na Pandora (por exemplo, via *Single Sign On* da própria ULHT).
- Criar uma imagem Docker base para a Pandora e tornar as existentes herdeiras desta.
- Adição e parametrização de novas linguagens na Pandora.
- Implementação de testes unitários em linguagens existentes na Pandora.
- Compatibilização dos testes com conjuntos de inputs de modo a reduzir a detecção dos mesmos por parte dos utilizadores.
- Auditoria e respectivas melhorias de segurança.
- Segmentação da área de administração de modo a facilitar administração concorrente entre várias unidades curriculares.

Bibliografia

[1] Start Bootstrap. *SB Admin 2*. Start Bootstrap LLC 2020, 2021. URL: https://startbootstrap.com/theme/sb-admin-2.

[2] Docker community. *Dockerhub - openjdk*. Dockerhub, 2021. URL: https://hub.docker.com/_/openjdk.

[3] OpenJS Foundation e contributors. *jQuery*. OpenJS Foundation, 2021. URL: https://jquery.com/.

[4] GNU. *GPL Compatible Licenses*. Free Software Foundation, Inc, 2021. URL: https://www.gnu.org/licenses/license-list.html#GPLCompatibleLicenses.

[5] javaTpoint. *Django MTV*. javaTpoint, 2018. URL: https://www.javatpoint.com/django-mvt.

[6] B Leal. *Automatic Assessment tool*. Universidade Lusófona de Humanidades e Tecnologias, 2019.

[7] SpryMedia Ltd. *DataTables*. SpryMedia Ltd, 2021. URL: https://datatables.net/.

[8] Bootstrap Team. *Bootstrap*. Bootstrap, 2021. URL: https://getbootstrap.com/.

[9] Chart.js Team. *Chart.js*. Chart.js, 2021. URL: https://www.chartjs.org/.

[10] Tutorialspoint. *Django Overview*. Tutorialspoint, 2021. URL: https://www.tutorialspoint.com/django/django_overview.htm.

Acrónimos

AAT *Automated Assessment Tool*. 1, 11, 19, 20, 44, 47

DTL *Django Template Language*. 1

GCC *GNU Compiler Collection*. 2, 13, 26

HTML *HyperText Markup Language*. 1

UI *User Interface*. 1, 3, 5, 43, 45

ULHT Universidade Lusófona de Humanidades e Tecnologias. 11, 47

UML *Unified Modeling Language*. 1, 3, 14, 15, 17, 18, 23

UX *User Experience*. 32