Universidade dos Açores

Cantina: Compra de Senhas

Informática - Redes e Multimédia
Programação Centrada em Objectos

Entrega final

Helder Correia 20102556 **Nuno Silva** 20082439 **Paulo Silva** 20082455 Grupo 1

Índice

Introdução

Este projecto tem como objectivo desenvolver uma aplicação de compra e gestão de senhas para uma cantina, em ambiente Universitário. A adopção deste sistema pretende melhorar e tornar mais eficiente o funcionamento deste processo.

A aplicação deverá ter como funcionalidades, entre outras, autenticação de um aluno, consulta do saldo, alteração do código de acesso, alteração de e-mail, e a própria compra de senhas.

Análise

Foi adoptada a ferramenta de controlo de versões Git para colaboração do código. Um VCS é algo de essencial nos dias que correm.

Tomou-se a decisão de adoptar a boa prática de implementar todo o código da aplicação em inglês, à excepção dos dados que serão mostrados ou introduzidos pelo utilizador.

Houve o intuito de criar testes automatizados com JUnit de forma a garantir o correcto funcionamento da aplicação, facilitar o desenho de código mais flexível e adaptável a mudanças, e para servir como documentação de como usar o código (i.e. na interface), mas acabou por ser abandonado. No entanto todo o desenho do modelo foi feito pensando em como seria testado, e portanto obteve-se parte do benefício do *Test Driven Development*, (TDD, no que toca ao desenho) mesmo sem os testes.

Teve-se desde o início a separação de responsabilidades em mente, o que promove a extensibilidade, modularidade e facilita a manutenção. Decidiu-se separar os diferentes domínios da aplicação em camadas, implementadas através de diferentes *packages*. Três grandes camadas são a *apresentação*, a *lógica de negócio* e os *dados*.

A **apresentação** é responsável pela interface do utilizador. A sua principal função é traduzir tarefas em algo que o utilizador consiga compreender, pedindo os dados necessários e mostrando os resultados no ecrã. Esta camada pode aceder às outras (por convenção).

A **lógica de negócio**, é onde se encontram os objectos de domínio. Esta camada processa pedidos, efectua decisões e avaliações lógicas, e cálculos. Ela não está consciente

da apresentação ou interface, nem da persistência de dados. É a mais independente de todas.

Foi decidido lidar com a **persistência de dados** separadamente da lógica de negócio, na sua própria camada. É aqui que a informação será guardada e recolhida do sistema de ficheiros ou base de dados, mas facilmente alterado para outro sistema como chamadas remotas, por exemplo.

Posteriormente as validações também foram separadas para uma camada em separado, tornando os objectos de domínio mais simples e leves.

Esta separação de responsabilidades foi muito central no desenvolvimento deste trabalho. A apresentação comunica livremente com o modelo, mas não o contrário. As validações apenas têm consciência dos objectos de domínio e estes últimos são completamente agnósticos a tudo fora deles. A persistência de dados, tal como os validadores apenas têm consciência dos objectos de domínio. Qualquer interacção entre estas camadas é feita na raiz do modelo, ou já na interface gráfica que interage com a totalidade do mesmo.

Cada *mapper* da persistência de dados contém um mapa de instâncias carregadas. Cada instância de um *mapper* é isolada, e consequentemente o seu mapa de objectos carregados. Portanto para obter uma referência a uma única instância de um *mapper*, facilmente acessível, usa-se o `MapperRegistry`. Essa classe usa *lazy loading* para instanciar cada *mapper* com valores que preencham os requisitos. Será nesta única classe que se pode alterar os caminhos dos ficheiros onde são guardados os objectos, por exemplo. Há no entanto a excepção do `Config`, porque não é um tipo de objecto de domínio.

`Config` fornece uma instância globalmente acessível, o caminho ao ficheiro de configurações, assim como define configurações por defeito e tem acesso às configurações do sistema. À primeira vez que este objecto é usado, se não existir ficheiro de configurações, o mesmo será criado automaticamente com os valores em si definidos.

`Application` serve como *Gateway*. Esta classe não é obrigatória, mas fornece um conjunto de métodos que agrupam certas operações que fazem parte dos requisitos da aplicação, de forma a simplificar o código necessário na interface gráfica.

Criaram-se duas interfaces gráficas separadas, com um *Backend* para os administradores e um *Frontend* para os estudantes. Assim, as diferentes permissões para cada grupo de utilizadores é implementada através das funcionalidades disponíveis em cada interface.

Programadores diferentes, geralmente têm estilos de programação diferentes. Para evitar ter uma base de código com formatação inconsistente, adoptaram-se as convenções de código dadas pela própria Oracle. As convenções tornam a base de código mais consistente, e ajuda a seguir boas práticas na formatação de um código mais legível para outros e portanto mais fácil de interpretar, mas também mais fácil de manter.

A escolha dos nomes dos *packages* é um exemplo de uma das decisões influenciadas pelas convenções de código da Oracle.

Conclusão

O grupo sentiu-se motivado em aproveitar este projecto para explorar e aprender as formas mais correctas de programar em projectos reais. Por esse motivo houve um esforço em atingir um bom equilíbrio entre adoptar soluções mais adequadas para projectos maiores e mais complexos, e soluções mais simples mas talvez menos flexíveis, para um projecto universitário como este.

A escolha de programar em inglês torna a leitura do código mais natural, uma vez que a linguagem Java está implementada nessa língua. A maior vantagem, no entanto, é ter um código "internacionalizado", possível de ser partilhado com programadores de outros países. Projectos universitários geralmente têm tempo de vida curta e público alvo bem definido (i.e. os seus avaliadores). No entanto há interesse em publicar no GitHub o projecto final, para referência, tornando o tempo de vida longo.

O grupo batalhou com a ideia de *"desenho planeado"*, onde se tenta planear todo o domínio antes de se começar a programar. Essa abordagem tem os seus problemas, o principal dos quais é o facto de que é impossível pensar em tudo o que precisaremos ao programar. É inevitável, à medida que se programa, ir encontrando coisas que questionam o desenho. Demasiado planeamento leva ao risco de haver o chamado *"analysis paralysis"* inicialmente, mas também durante o desenvolvimento, onde se nota cada vez mais falhas no desenho inicial que precisam ser corrigidas e adaptadas.

O grupo acha mais natural e eficiente em termos de tempo a abordagem de *"desenho evolutivo"*, onde se usa inicialmente um ou vários diagramas UML para discutir ideias, mas apenas o suficiente para começar a trabalhar na melhor direcção possível. A esse nível, é

uma visão mais abstracta do problema. As soluções mais concretas seriam descobertas na criação dos testes automatizados, como descrito pela metodologia TDD.

O grupo achou interessante explorar o TDD (Test Driven Development), devido ao enorme valor que esse conhecimento traz profissionalmente. No entanto, sendo que apenas um elemento tinha algum conhecimento nesta área, tentou-se evitar que os outros elementos do grupo não esperassem pelos testes para começar a criar os objectos de domínio, ou que tivessem que desviar muito tempo a aprender. Esta limitação de tempo, impediu a correcta implementação de um verdadeiro TDD, e tentou-se comunicar o melhor possível, soluções através do UML.

Apesar disso, continuou a haver o investimento em criar testes unitários, que provaram o correcto funcionamento dos objectos do domínio, mesmo após estes já terem sido criados. Muitas vezes aconteceu verificar que uma dada implementação era difícil de testar, o que indicava que seria também mais difícil de usar. Isso levou a várias refactorizações, o que inevitavelmente levou a uma implementação mais extensível e mais adaptável a mudanças, provando a eficácia do TDD.

Contudo, a meio do projecto acabou-se por dar prioridade a tornar a aplicação funcional, e portanto os testes foram abandonados, com a ideia de que este poderá ser no futuro um projecto de exemplo onde se possa praticar a criação de testes automatizados.

Inicialmente, a vertente gráfica da aplicação começou a ser desenvolvida “de raíz” de forma a aplicar os conceitos leccionados e obter maior controle sobre os componentes Swing. Contudo, esta medida foi abandonada por falta de viabilidade e substituída pelo *Swing GUI Builder* do NetBeans. Assim conseguiu-se não só um desenvolvimento mais eficiente, como também uma interface mais apelativa no geral para o utilizador.

Houve dificuldade em decidir qual a melhor solução de fluxo de menus a implementar pois o grupo acredita que a interface deve ser clara, auto-explicativa e de fácil usabilidade.

Bibliografia

Livros

1. Martins FM. JAVA6 e Programação Orientada pelos Objectos. 2ª Edição. Lisboa: FCA - Editora de Informática; 2009.

2. Freeman S, Pryce N. Growing Object-Oriented Software, Guided By Tests. Addison Wesley; 2009.

3. Fowler M. Patterns of Enterprise Application Architecture. Addison Wesley; 2002.

Web

4. Oracle. The Java Tutorials. 2011 [Acedido em Dezembro, 2011]. Disponível em: http://download.oracle.com/javase/tutorial/

5. Fowler M. Is Design Dead? 2004 [Acedido em Dezembro, 2011]. Disponível em: http://martinfowler.com/articles/designDead.html

6. Oracle, Code Conventions for the Java Programming Language. 1999 [Acedido em Dezembro, 2011]. Disponível, em: http://www.oracle.com/technetwork/java/codeconv-138413.html