Universidade do Minho
Escola de Engenharia

Diogo Francisco de Carvalho Almeida

Catálogo de Usability Smells

Outubro de 2014

Universidade do Minho
Escola de Engenharia
Departamento de Engenharia Informática

Diogo Francisco de Carvalho Almeida

Catálogo de Usability Smells

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor José Francisco Creissac Freitas de Campos
Professor João Alexandre Baptista Vieira Saraiva

Outubro de 2014

DECLARAÇÃO

Nome

Diogo Francisco de Carvalho Almeida

Endereço eletrónico: diogoal20@gmail.com Telefone: 910121633

Número do Bilhete de Identidade: 13602573

Título dissertação ☒ /tese ☐

Catalogo de Usability Smells

Orientador(es):

Professor José Francisco Creissac Freitas de Campos
Professor João Alexandre Baptista Vieira Saraiva

Ano de conclusão: 2014

Designação do Mestrado ou do Ramo de Conhecimento do Doutoramento:

Mestrado de Engenharia Informática

Nos exemplares das teses de doutoramento ou de mestrado ou de outros trabalhos entregues para prestação de provas públicas nas universidades ou outros estabelecimentos de ensino, e dos quais é obrigatoriamente enviado um exemplar para depósito legal na Biblioteca Nacional e, pelo menos outro para a biblioteca da universidade respectiva, deve constar uma das seguintes declarações:

1. É AUTORIZADA A REPRODUÇÃO INTEGRAL DESTA TESE/TRABALHO APENAS PARA EFEITOS DE INVESTIGAÇÃO, MEDIANTE DECLARAÇÃO ESCRITA DO INTERESSADO, QUE A TAL SE COMPROMETE;

Universidade do Minho, 31 / Outubro / 2014

Assinatura: Diogo Francisco de Carvalho Almeida

AGRADECIMENTOS

A realização e conclusão deste trabalho nunca teria sido possível sem a contribuição direta ou indireta de várias pessoas. Para todos, os meus sinceros agradecimentos! Porém, um agradecimento muito especial:

- Aos meus pais, por não medirem esforços para que alcançasse esta etapa da minha vida, pelas palavras de incentivo nas horas mais difíceis e pelo carinho, compreensão e paciência disponibilizada. Também à minha irmã pelas mesmas razões e, principalmente, os gestos tidos e pelas palavras ditas nos momentos certos em que mais precisei de sorrir;
- Ao meu Orientador Professor José Francisco Creissac Freitas de Campos e Co-Orientador Professor João Alexandre Baptista Vieira Saraiva pelas suas orientações traduzidas numa presença sempre constante e numa resposta sempre atempada aos meus apelos, acompanhados sempre de palavras de confiança e incentivo. Agradeço-lhes igualmente os ensinamentos, críticas e sugestões dadas ao longo deste trabalho;
- A todos aqueles que se demonstraram disponíveis para esclarecer dúvidas ou para fornecer material imprescindível para o trabalho, em especial ao Professor João Carlos Silva do Dep. de Tecnologias/DIGARC, Instituto Politécnico do Cávado e do Ave pelos ensinamentos, críticas e orientações durante todo este trabalho;
- À Alexandra por acreditar em mim desde o momento em que nos conhecemos, por me apoiar sempre nas horas boas e más. Um especial obrigado pela sua amizade incondicional, pelo seu suporte, pela sua coragem e força que teve e tem para comigo e pelo ombro amigo sempre disponível. Obrigado a ti namorada, amiga e companheira.
- A todos os meus avós que nem sempre perto estiveram sempre a pensar e olhar por mim. Pelas suas palavras amigas e de incentivo e todo o suporte e apoio que me deram e dão;
- Sr. Domingos, à Dona Graça e ao Luís pelo apoio, amizade e incentivo ao longo deste percurso;
- E a todos os meus familiares e amigos.

RESUMO

A dissertação tem como objetivo explicar o que são este tipo de *Smells* e como estes podem prejudicar a usabilidade nas aplicações interactivas/web. Um *Smell* é algo no código ou na aplicação que não sendo um erro torna difícil a sua compreensão, manutenção e evolução. Este trabalho apresenta um estudo de arte com base nos *Code Smells* de Martin Fowler e na abordagem que Hermans *et al.* apresenta para deteção de *smells* em folhas de cálculo. É então desenvolvido o catálogo de smells potencialmente relevantes para interfaces gráficas com o utilizador que irá servir de base ao estudo dos *Usability Smells*. É ainda explicado o que são *Usability Smells*, como podem ser encontrados e possíveis soluções para a sua eliminação baseadas no estudo de usabilidade realizado. Foi concluído que todos os *Usability Smells* apresentados no teste de usabilidade foram considerados anomalias que afetam a interface das aplicações prejudicando, assim, a compreensão, manutenção e evolução das mesmas. Além disso, foi também definido um *Refactoring* para cada um dos *Usability Smells* com o objetivo de os eliminar.

Palavras-chave: Usability, Software, Software Quality, Code Smells, Smells, Refactoring, Inter-worksheets Smells, Martin Fowler

ABSTRACT

This dissertation aims to explain the Usability Smells and how they can harm the usability in interactive applications/web. A Smell is something in the code or in the application that, despite not being an error, makes its comprehension, maintenance and evolution difficult. This paper presentes a study of art based on the Code Smells of Martin Fowler and on the approach that Hermans et al. introduced for detecting smells in spreadsheets. Then a catalog of potentially relevant smells for graphical user interfaces is developed, which will be the foundation to study the Usability Smells. Furthermore, it is also explained what are Usability Smells,how they can be found and possible solutions for their elimination based on usability study conducted. It was concluded that all Usability Smells presented in the usability test were considered anomalies that affect the interface of the applications, thus harming its comprehension, maintenance and evolution. Moreover, it was also defined a Refactoring for each one of the Usability Smells aiming to eliminate them.

Keywords: Usability, Software, Software Quality, Code Smells, Smells, Refactoring, Inter-worksheets Smells, Martin Fowler

CONTEÚDO

Contents ix

1 INTRODUÇÃO 1
- 1.1 Introdução 1
- 1.2 Motivação 3
- 1.3 Perguntas de Investigação 3
- 1.4 Estrutura da Dissertação 4

2 CODE SMELLS 5
- 2.1 Introdução 5
- 2.2 O que são Code Smells 5
- 2.3 Trabalhos Relacionados 6
  - 2.3.1 Code Smells de Martin Fowler 6
  - 2.3.2 Refactoring por Martin Fowler 8
  - 2.3.3 Inter-Worksheets Smells de Felienne Hermans *et al.* 15

3 USABILITY SMELLS 18
- 3.1 Introdução 18
- 3.2 O que são Usability Smells? 19
- 3.3 Catálogo de Usability Smells 19
  - 3.3.1 Usability Smell Shotgun Surgery 20
  - 3.3.2 Usability Smell Too Many Layers 22
  - 3.3.3 Usability Smell Middle Man 22
  - 3.3.4 Usability Smell Information Overload 23
  - 3.3.5 Usability Smell Inappropriate Intimacy 25
  - 3.3.6 Usability Smell Feature Envy 26

4 VALIDAÇÃO DOS USABILITY SMELLS 28
- 4.1 Introdução 28
- 4.2 Dados, Metodologia e Pré-Teste 28
- 4.3 Aplicação caso de estudo - OH Open Hospital 29
- 4.4 Teste de Usabilidade 29
- 4.5 Teste de Usabilidade - Tarefas 31
  - 4.5.1 Tarefa 1: Middle Man 31
  - 4.5.2 Tarefa 2: Too Many Layers 34
  - 4.5.3 Tarefa 3: Shotgun Surgery 36
  - 4.5.4 Tarefa 4: Inappropriate Intimacy 38

4.5.5 Tarefa 5: Information Overload 40
5 RESULTADOS DA AVALIAÇÃO DOS USABILITY SMELLS 42
5.1 Resultados Obtidos 42
5.1.1 Resultados Tarefa 1: Middle Man 42
5.1.2 Resultados Tarefa 2: Too Many Layers 46
5.1.3 Resultados Tarefa 3: Shotgun Surgery 49
5.1.4 Resultados Tarefa 4: Inappropriate Intimacy 52
5.1.5 Resultados Tarefa 5: Information Overload 56
5.2 Discussão dos resultados 60
6 CONCLUSIONS AND FUTURE WORK 66
i APENDICES 70
A TESTE DE USABILIDADE 71
B CODE SMELLS DE MARTIN FOWLER 77
C REFACTORINGS DE MARTIN FOWLER 85

LISTA DE FIGURAS

Fig. 1 Move Method 10
Fig. 2 Move Field 11
Fig. 3 Extract Class 11
Fig. 4 Inline Class 12
Fig. 5 Hide Delegate 13
Fig. 6 Remove Middle Man 13
Fig. 7 Change Bidirectional Association to Unidirecional 14
Fig. 8 Replace Delegation with Inheritance 14
Fig. 9 Usability Smell - Shotgun Surgery 21
Fig. 10 Usability Smell - Too Many Layers 22
Fig. 11 Usability Smell - Middle Man 24
Fig. 12 Usability Smell - Information Overload 24
Fig. 13 Usability Smell - Inappropriate Intimacy: a página 1 à esquerda é usada para inserir informação do autor, a página 2 ao centro tem como objetivo inserir informação do livro e a página 3 à direita é uma página de apresentação de mensagens de erro. 25
Fig. 14 Usability Smell - Feature Envy 27
Fig. 15 Aplicação OH - Open Hospital 30
Fig. 16 Imagens da Tarefa 1 representando um exemplo do Middle Man 32
Fig. 17 Refactoring apresentado para o Middle Man. 33
Fig. 18 Imagens da Tarefa 2 representando um exemplo do Too Many Layers 34
Fig. 19 Refactoring apresentado para o Too Many Layers. 35
Fig. 20 Imagens da Tarefa 3 representando um exemplo do Shotgun Surgery 36
Fig. 21 Refactoring apresentado para o Shotgun Surgery. 37
Fig. 22 Imagens da Tarefa 4 representando um exemplo do Inappropriate Intimacy 38
Fig. 23 Refactoring apresentado para o Inappropriate Intimacy. 39
Fig. 24 Imagens da Tarefa 5 representando um exemplo do Information Overload 40
Fig. 25 Refactoring apresentado para o Information Overload. 41
Fig. 26 Resultados primeiro grupo de avaliação do Middle Man 43
Fig. 27 Resultados terceiro grupo de avaliação do Middle Man 45
Fig. 28 Resultados primeiro grupo de avaliação do Too Many Layers 46
Fig. 29 Resultados segundo grupo de avaliação do Too Many Layers 48
Fig. 30 Resultados primeiro grupo de avaliação do Shotgun Surgery 49

Lista de Figuras

Fig. 31	Resultados terceiro grupo de avaliação do Shotgun Surgery	51
Fig. 32	Resultados primeiro grupo de avaliação do Inappropriate Intimacy	52
Fig. 33	Resultados segundo grupo de avaliação do Inappropriate Intimacy	55
Fig. 34	Resultados primeiro grupo de avaliação do Information Overload	56
Fig. 35	Resultados segundo grupo de avaliação do Information Overload	59
Fig. 36	Exemplo do Pattern Responsive Enabling	61
Fig. 37	Exemplo do Pattern One-Window Drilldown	62
Fig. 38	Exemplo do Pattern Dropdown Chooser	63
Fig. 39	Exemplo do Pattern Responsive Disclosure	64
Fig. 40	Exemplo do Pattern Closable Panels	65
Fig. 41	Exemplo do Pattern Clear Entry Points	65
Fig. 42	Introduce Local Extension	88
Fig. 43	Replace Data Value with Object	88
Fig. 44	Duplicate Observer Data	90
Fig. 45	Encapsulate Collection	90
Fig. 46	Replace Type Code with Class	91
Fig. 47	Replace Type Code with Subclasse	91
Fig. 48	Replace Type Code with State/Strategy	92
Fig. 49	Rename Method	93
Fig. 50	Rename Parameter	94
Fig. 51	Remove Setting Method	95
Fig. 52	Pull Up Field	97
Fig. 53	Push Down Method	97
Fig. 54	Push Down Field	98
Fig. 55	Extract Subclass	98
Fig. 56	Extract Superclass	99
Fig. 57	Extract Interface	99
Fig. 58	Collaps Hierarchy	100
Fig. 59	Form Template Method	101

LISTA DE TABELAS

Tab. 1 Frequências relativas Middle Man - Grupo 1 43
Tab. 2 Frequências relativas Middle Man - Grupo 3 45
Tab. 3 Frequências relativas Too Many Layers - Grupo 1 46
Tab. 4 Frequências relativas Too Many Layers - Grupo 3 48
Tab. 5 Frequências relativas Shotgun Surgery - Grupo 1 49
Tab. 6 Frequências relativas Shotgun Surgery - Grupo 3 51
Tab. 7 Frequências relativas Inappropriate Intimacy - Grupo 1 53
Tab. 8 Frequências relativas Inappropriate Intimacy - Grupo 3 55
Tab. 9 Frequências relativas Information Overload - Grupo 1 56
Tab. 10 Frequências relativas Information Overload - Grupo 3 59

LIST OF LISTINGS

2.1 Antes do Extract Method . . . 9
2.2 Depois do Extract Method . . . 9
2.3 Antes do Inline Method . . . 9
2.4 Depois do Inline Method . . . 10

C.1 Antes do Replace Temp with a Query . . . 85
C.2 Depois do Replace Temp with a Query . . . 85
C.3 Antes do Replace Method with Method Object . . . 86
C.4 Depois do Replace Method with Method Object . . . 86
C.5 Antes do Substitute Algorithm . . . 86
C.6 Depois do Substitute Algorithm . . . 87
C.7 Antes do Introduce Foreign Method . . . 87
C.8 Depois do Introduce Foreign Method . . . 87
C.9 Antes do Replace Array with Object . . . 89
C.10 Depois do Replace Array with Object . . . 89
C.11 Antes do Encapsulate Field . . . 89
C.12 Depois do Encapsulate Field . . . 89
C.13 Antes do Decompose Conditional . . . 92
C.14 Depois do Decompose Conditional . . . 92
C.15 Antes do Replace Parameter with Explicit Methods . . . 94
C.16 Depois do Replace Parameter with Explicit Methods . . . 94
C.17 Antes do Preserve Whole Object . . . 94
C.18 Depois do Preserve Whole Object . . . 95
C.19 Antes do Replace Parameter with Method . . . 95
C.20 Depois do Replace Parameter with Method . . . 96
C.21 Antes do Introduce Parameter Object . . . 96
C.22 Depois do Introduce Parameter Object . . . 96

1

INTRODUÇÃO

1.1 INTRODUÇÃO

O custo de manutenção do software é hoje em dia um tema muito discutido. A manutenção do software é dos processos que mais dificultam o controlo dos custos aumentando-os gradualmente (Lee et al., 2014). A ausência de métricas de qualidade no processo de desenvolvimento e manutenção de software resulta de várias causas: a falta de tempo disponível para concretizar a tarefa, a falta de definição dessas métricas ou até mesmo a questões de inexperiência e falta de *know how* nesta área.

O percurso de aprendizagem de algoritmia nem sempre é curto e varia de pessoa para pessoa, contudo uma coisa é certa: no início todos cometem erros. Todo o programador, no começo dos seus estudos, entende que o código mesmo executando e realizando as tarefas propostas nem sempre é o mais correto. Vejamos o seguinte exemplo em pseudo-código:

```
void funcao_Com_A_Maior_Assinatura_Que_Nao_Faz_Nada(int totalTarefas, int var_1,
    int var_2, ...) {

    int Tarefas[totalTarefas];

    Tarefas[0] = var_1;
    Tarefas[1] = var_2;
    Tarefas[2] = var_3;

    (...)

    se Tarefas == null
      procurarTarefas();
    senao
      acabarTarefas(Tarefas);
}
```

Este é um exemplo trivial do que acontece no início da nossa vida de programadores, criamos código com o objetivo de resolver problemas e raramente olhamos para ele com outros olhos e pensamos: "Será que é a maneira mais correta e legível de realizar esta tarefa?" ou "Precisarei de me preocupar com a sua manutenção?". Ao não pensar nestas questões o programador continua a sua vida e o código continua igual, útil mas sem futuro.

Ao analisar a função acima descrita encontra-se diversos pormenores que não estando corretos funcionam. Começamos pela assinatura do método: o nome que lhe foi atribuído além de ser grande de mais não explica o seu objetivo. Se é uma função para realizar tarefas porque é que se deve chamar *"funcao_Com_A_Maior_Assinatura_Que_Nao_Faz_Nada"*. Depois da assinatura passamos para os parâmetros que recebe. Quantas mais tarefas existissem mais variáveis seriam passadas? Continuando, olhando por exemplo para o interior do método, se tivermos mais variáveis a ser passadas nos parâmetros mais atribuições fazíamos ao *Array* de Tarefas. Não prolongando mais a lista verifica-se que o código está correto e executa mas a maneira como foi escrito poderá não ser a melhor. Estas "coisas" que nos parecem erradas são designadas de *Code Smells*. Os *Code Smells* não são erros mas anomalias, bocados de código que "cheiram mal". Estas anomalias prejudicam a legibilidade e manutenção do código.

Como quase tudo na vida também existe forma de corrigir e resolver os *Code Smells*, chama-se a esse processo: *"Refactoring"*. Segundo Simon et al. (2001) o processo de *Refactoring* é uma questão chave para aumentar a qualidade interna do software durante todo o seu ciclo de vida. Já Opdyke (1992) acrescenta: os *Refactorings* não alteram o comportamento do programa, assim o conjunto de inputs e outputs deverá ser igual antes e depois de aplicar o *refactoring*. Como os *refactorings* não alteram o comportamento do programa, melhoram o desenho e evolução do software reestruturando o programa deixando-o mais limpo e legível para outros programadores. Um exemplo de um *Refactoring* para o *smell* do nome do método seria: reduzir o nome e alterá-lo para algo que seja percetível ao programador ou a outra pessoa perceber para que serve sem necessitar de olhar para o corpo do método. Uma solução para reduzir a infindável lista de parâmetros seria utilizar linguagens orientadas a objetos, como por exemplo o Java que permite ao programador passar os atributos que quiser nos objetos e caso o objeto necessite de outros atributos este pode pedi-los a outros objetos.

O estudo de *Code Smells* e respetivos *Refactorings* há muito que é feito. Fowler (2002) foi quem apresentou pela primeira vez este conceito definindo que são peças de software que podem ter problemas. Fowler na sua obra apresenta 22 *Code Smells* em código e respetivas soluções para os corrigir. Os *Code Smells* não são estudados apenas no código, existem diversos trabalhos onde são definidos e adaptados por exemplo a folhas de cálculo, desenho e manutenção de software, etc. Cunha et al. (2012b) tentaram identificar *spreadsheet smells* em folhas de cálculo. Para isso criaram um catálogo e uma metodologia para identificar *smells*. Depois desenvolveram uma ferramenta que identificasse os *smells* e executaram-na num dos maiores repositórios de folhas de cálculo, o EUSES, onde por exemplo detetaram que 20% das células com *smell* apontam para um possível problema nas folhas de cálculo, Cunha et al. (2012a). Além de Cunha et al. (2012b), Hermans et al. (2012) apresentam uma

abordagem para a deteção de *smells* em folhas de cálculo. Este trabalho será apresentado com maior detalhe na secção 2.3.3 devido a ser um estudo base nesta dissertação.

Outro exemplo é Ganesh et al. (2013) onde o estudo dos *smells* em linguagens de programação orientadas a objetos mostrou uma carência na organização e definição dos *smells* que afetam este tipo de linguagens. Um estudo onde sejam organizados os *smells* existentes nas linguagens de Programação Orientadas a Objetos (POO), avaliados de acordo com a sua influência nas aplicações, ajudaria o *designer* a entender melhor as potenciais falhas no design da aplicação. De forma a colmatarem essa lacuna Ganesh et al. (2013) adotaram uma abordagem para classificar e organizar *smells* onde apresentam os que violam princípios de programação orientada a objetos.

Este relatório de dissertação no âmbito da tese de mestrado em Engenharia Informática pretende abordar a questão dos *code smells* em aplicações interactivas/web.

1.2 MOTIVAÇÃO

A motivação para a escolha deste tema ocorreu quando procurava estudos sobre usabilidade de aplicações e *code smells*. Nessa procura apercebi-me que a quantidades de estudos que relaciona-se esses dois temas era reduzida ou quase nula. Foi nessa altura que me questionei se não existia possibilidade de relacionar os *code smells* com aplicações interativas. Para tentar relacionar esses dois temas foquei-me nos trabalhos de Fowler (2002) onde é apresentado um catálogo de *code smells* e Hermans et al. (2012) que tenta criar uma abstração dos *code smells* a folhas de cálculo. Com este estudo pretendo desenvolver um catálogo de *Usability Smells* (explicado no cap. 3) capaz de ajudar os programadores a melhorar a interface gráfica de aplicações interativas, e quem sabe ajudar no desenvolvimento de novos paradigmas focados em melhorar ou desenvolver novos tipos de interfaces.

1.3 PERGUNTAS DE INVESTIGAÇÃO

Pretende-se com esta tese de mestrado desenvolver e validar um catálogo de *Usability Smells* com o intuito de ajudar os programadores a melhorar a usabilidade e qualidade da interface das aplicações interactiva. Tipicamente a análise da interface é um processo que exige testes com utilizadores reais. Para proceder a esses testes iremos inicialmente identificar um conjunto de *usability smells* mais comuns. A identificação dos *smells* será focada nos trabalhos de Fowler (2002) e Hermans et al. (2012) e pretende-se com isso responder às seguintes questões:

- R1 Existirá possibilidade de identificar *usability smells* em aplicações interactivas?
- R2 Quais os *usability smells* mais comuns? E será que expõem ameaças na qualidade e integridade da aplicação?
- R3 Qual a melhor solução para identificar os *usability smells*

1.4 ESTRUTURA DA DISSERTAÇÃO

Este documento tem a seguinte estrutura: o primeiro capítulo é composto por uma introdução à tese, a motivação que levou à realização deste estudo, as perguntas a que se pretende responder no final e a estrutura do documento. O segundo capítulo abordará o tema *Code Smells* explicando: o que são os *Code Smells*, em que ponto se enquadra esta tese no seu contexto e discussão sobre alguns trabalhos chave que ajudaram na realização do estudo da arte, entre eles os trabalhos de Fowler (2002) e Hermans et al. (2012). No terceiro capítulo é explicada a transição *Code Smells* para *Usability Smells* e o porquê do seu estudo usando para isso um catálogo desenvolvido. O quarto capítulo descreve o estudo realizado para validar o catálogo presente no capítulo anterior e finalmente no quinto capítulo são apresentados os resultados do estudo realizado.

Para não sobrecarregar a tese foram desenvolvidos 2 anexos referentes aos restantes *code smells* e respetivos *refactorings* de Fowler (2002). Estes *code smells* não têm tanta importância como os apresentados no terceiro capítulo, contudo não poderiam deixar de ser estudados. Assim, para possível leitura, foram transferidos para os dois anexos: B e C.

2

CODE SMELLS

2.1 INTRODUÇÃO

Neste capítulo será introduzido o tema de *Code Smells*. Será explicado o que são este tipo de anomalias, o contexto em que se inserem e os dois principais trabalhos, que abordam este tema, que ajudaram na realização deste relatório: Fowler (2002) e Hermans et al. (2012). Como tipicamente a maioria dos problemas têm resolução o mesmo também acontece com estas anomalias, dessa forma será também apresentado um conjunto de *Refactorings*[1] propostos por Fowler (2002).

2.2 O QUE SÃO CODE SMELLS

Em meados dos anos 90 a indústria do software começou uma infinita escalada tornando-se um dos sectores que mais dinheiro faz circular em toda a economia mundial. Quanto mais evoluía o mundo das tecnologias menos tempo as empresas tinham para se atualizarem ou finalizarem as suas encomendas. O tempo passava e cada vez mais havia concorrência, devido a este aumento apareceu um novo paradigma de programação onde era mais importante o trabalho feito que a qualidade do mesmo. Este paradigma "obrigou" os programadores a serem cada vez mais preguiçosos seguindo a máxima do *Quick and Dirty*. Esta máxima força os programadores a procurar maneiras mais rápidas de completarem os seus objetivos originando o aparecimento de falhas na manutenção e evolução do seu código sem falar na compreensão para ele e para os outros Arora and Gambardella (2004).

Atualmente, porque têm objetivos e prazos a cumprir, os programadores tentam apressar-se deparando-se normalmente com erros no seu código, outras vezes mesmo sem erros o programa executa mas algo não "cheira bem", é daqueles casos que o deixam a pensar: "Algo aqui não está bem." ou "Falta aqui algo!", em todos estes casos ele pode estar perante um *code smell*.

Um *Code Smell* é algo no código ou na aplicação que não sendo um erro torna difícil a sua compreensão, manutenção e evolução. Tipicamente estes *code smells* apresentam a necessidade de o programador alterar alguma coisa no seu código, ou seja, aplicar um *Refactoring* na estrutura do seu programa.

1 Propostas de reformulação de código com o intuído de resolver *code smells*.

Um dos nomes mais falados no mundo dos *Code Smells* é o de Martin Fowler. Fowler instanciou uma lista de 22 *code smells* na sua obra (Fowler, 2002) que serão abordados na Secção 2.3.1 e apresentou os respetivos *refactorings*.

Segundo Fowler "Refactoring é o processo de alterar o software de forma a não modificar o comportamento externo do código contudo melhorando a sua estrutura interna. É uma maneira disciplinada de limpar o código minimizando as chances de aparecimento de bugs." (Fowler, 2002, página 9).

2.3 TRABALHOS RELACIONADOS

Nesta secção serão apresentados dois trabalhos que serão a base desta tese de mestrado. Em primeiro lugar será apresentado o trabalho sobre *Code Smells e Refactoring* de Martin Fowler onde iremos focar principalmente os *code smells* e respetivos *refactorings* que ele aponta (Fowler, 2002). O trabalho do Fowler apresenta 22 *code smells* e os respetivos *refactoring*. Para facilitar a apresentação serão apresentados 4 dos 22 *smells* com os respetivos *refactorings*, os restantes encontra-se explicados nos anexos B e C. Por último será apresentado o trabalho de Hermans et al. (2012) sobre *Inter-worksheets smell* onde são apresentados um conjunto de *code smells* instanciados para folhas de cálculos.

2.3.1 *Code Smells de Martin Fowler*

Shotgun Surgery

O *Shotgun Surgery* é um smell ocorre quando um método é invocado em diversos lugares e a alteração da sua assinatura obrigada a alteração em todos esses locais. É necessário saber todas as classes e métodos que se deve alterar quando o método ou classe a que fazem referência esta a ser alterado. Segundo Fowler o *Shotgun surgery* é um smell que dá bastante trabalho a corrigir, para a resolução deste *code smell* é indispensável para o programador saber todos os locais onde as classes ou métodos foram instanciadas.

A solução passaria por usar o *Refactoring Move Method*, Secção 2.3.2, e o *Move Field*, Secção 2.3.2, de forma a colocar todas as alterações numa única classe. Caso não se encontre a classe ideal para onde mudar as alterações o ideal passa por criar uma nova. Além disso usar o *Inline Classe*, Secção 2.3.2, para transferir todo o comportamento para essa nova classe.

Feature Envy

O Feature Envy ocorre quando um método tem mais interesse nos métodos de outras classe que nos da própria classe. Segundo Fowler são inúmeros os casos de métodos que invocam métodos de outras classes para realizar as suas próprias tarefas.

Uma possível solução de *Refactoring* para este *code smell* seria usar o *Move Method* 2.3.2 dessa forma passaríamos o método para a classe onde se encontra a maioria dos métodos que ele utiliza.

Se apenas metade ou uma parte do método sofrer deste smell a solução seria usar o *Refactoring Extract Method* 2.3.2 nessa parte especial e depois o *Move Method*, Secção 2.3.2, passando essa fração de código para a devida classe.

Middle Man

O *Middle Man* é um *code smell* que ocorre quando existem pelo menos duas classes (ou métodos) e uma delas delega competências à outra, competências que poderiam ser realizadas pela primeira página. Entende-se como competência a passagem de lógica interna à classe.

Segundo Fowler a encapsulação vem sempre com delegação. Um exemplo trivial deste smell é por exemplo o de um patrão e a sua secretária. O patrão delega todo o comportamento para a secretária. O problema aparece se o patrão ficar sem secretária, ele não saberá como realizar os recados que outrora delegava à secretária.

Fowler sugere que uma das soluções passe por usar o *Refactoring Remove Middle Man*, Secção 2.3.2, contudo este processo é o inverso do *Hide Delegate*, Secção 2.3.2, a resolução utilizando um destes *Refactorings* faz aparecer o outro e vice-versa. Se forem apenas alguns métodos que sofram deste smell o *Inline Method*, Secção 2.3.2, é suficiente. Por último caso exista comportamento adicional entre classes deve-se usar o *Replace Delegation with Inheritance*, Secção 2.3.2. Esta solução passa por transformar o *Middle Man* numa subclasse do objeto principal possibilitando desta forma alterar o objeto principal sem alterar as suas delegações.

Inappropriate Intimacy

O Inappropriate Intimacy é um smell que aparece quando uma classe tem muitas dependências ou detalhes de implementação de outras classes tornando-se muito íntimas. Assim as classes gastam mais tempo nos métodos das outras classes que nos próprios.

De forma a reduzir a "intimidade" entre classes, Fowler sugere a utilização dos seguintes *Refactorings*: *Move Method* e o *Move Field* na Secção 2.3.2.

Quando existe uma ligação bidirecional entre classes e apenas se faça uso de um sentido da ligação, então deve ser usado o *Change Bidirectional Association to Unidirecional*, Secção 2.3.2. Caso existam interesses múltiplos entre classes, isto é: lógica comum às duas classes, deve-se usar o *Extract Class* ou o *Hide Delegate*, respetivas Secções 2.3.2 e 2.3.2, para extrair essa lógica e transforma-la numa subclasse que possa ser invocada por ambas.

2.3.2 *Refactoring por Martin Fowler*

Esta subsecção destina-se a apresentar os *Refactorings* para os *Code Smells* apresentados por Fowler. Todos estes *refactorings* provêm da obra Fowler (2002). As imagens dos *refactorings* foram retiradas do site *Refactoring* de Martin Fowler[2].

Para começar é preciso definir o que é um *Refactoring*: todo o processo à volta da remoção de um *Code Smell*, desde a sua identificação até a eliminação, com o objetivo de melhorar e aumentar a qualidade da estrutura interna da aplicação sem alterar a sua interface. Segundo Simon et al. (2001) este processo é uma questão chave para continuar a progredir qualquer aplicação mantendo a melhor qualidade interna do software durante todo o seu ciclo de vida. Já Opdyke (1992) acrescenta: os *Refactorings* não devem alterar o comportamento do programa, assim o conjunto de inputs e outputs deverá ser igual antes e depois de aplicar o *refactoring*. Como os *refactorings* não alteram o comportamento do programa permitem a evolução do software reestruturando o programa deixando-o mais limpo e legível para outros programadores.

Fowler divide a seguinte lista de *refactorings* nos seguintes grupos:

- Composição de Métodos:

  Extract Method, *Inline Method*, *Replace Temp with Query*, *Replace Method with Method Object*, *Substitute Algorithm*.

- Mover recursos entre objetos:

  Move Method, *Move Field*, *Extract Class*, *Inline Class*, *Hide Delegate*, *Remove Middle Man*, *Introduce Foreign Method*, *Introduce Local Extension*.

- Organizar Dados:

  Replace Data Value with Object, *Replace Array with Object*, *Duplicate Observer Data*, *Change Bidirectional Association to Unidirection*, *Encapsulate Field*, *Encapsulate Collection*, *Replace Type Code with Class*, *Replace Type Code with Subclasses*, *Replace Type Code with State/Strategy*.

- Simplificar expressões condicionais:

  Decompose Conditional, *Replace Conditional with Polymorphism*, *Introduce Null Object*.

- Simplificar as chamadas a métodos:

  Rename Method, *Remove Parameter*, *Replace Parameter with Explicit Methods*, *Preserve Whole Object*, *Replace Parameter with Method*, *Introduce Parameter Object*, *Hide Method*, *Remove Setting Method*.

2 http://refactoring.com/catalog/ visitado pela última vez a 18 de Março de 2014.

- Lidar com generalizações:

  Pull Up Field, *Push Down Method*, *Push Down Field*, *Extract Subclass*, *Extract Superclass*, *Collapse Hierarchy*, *Form Template Method*, *Replace Delegation with Inheritance*.

Extract Method

Utiliza-se este *Refactoring* quando a estrutura de um método começa a ser demasiado grande afetando assim a legibilidade do código. A Solução: Dividir o método em dois ou mais métodos de forma a facilitar a manutenção e compreensão.

```
void printOwing(double amount) {
    printBanner();
    //print details
    System.out.println ("name:" + name);
    System.out.println ("amount" + amount);
}
```

Listing 2.1: Antes do Extract Method

```

void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println ("name:" + name);
    System.out.println ("amount" + amount);
}
```

Listing 2.2: Depois do Extract Method

Inline Method

Este *Refactoring* tem como objetivo simplificar o complicado isto é: imaginando uma classe que têm dois métodos que se completam e que sempre que se utiliza um também se utiliza o outro, estes dois métodos podem ser alterados e transformados num só. A Solução: juntar os dois métodos para simplificar o código.

```
int getRating() {
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
  return _numberOfLateDeliveries > 5;
```

```
}
```

Listing 2.3: Antes do Inline Method

```
int getRating() {
  return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Listing 2.4: Depois do Inline Method

Move Method

Quando uma classe usa mais recursos ou métodos de outras classes do que da própria. A Solução: Criar um método similar ao da classe mais usada ou transformar o método antigo numa simples delegação ou remover o método. Ver Figura 1, Secção 2.3.

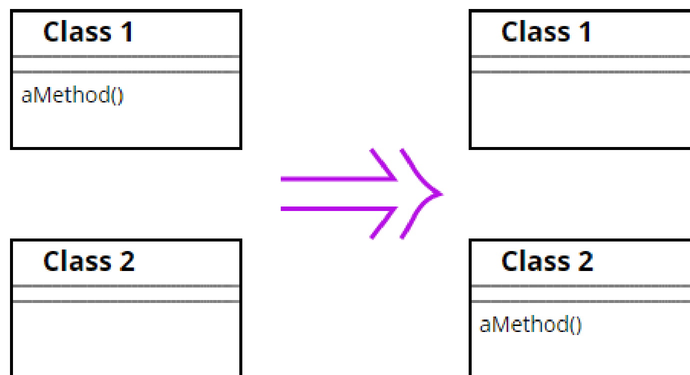


Fig. 1.: Figura Refactoring Move Method.

Move Field

Quando um(a) campo/variável é mais utilizado(a) por outra classe do que na qual se encontra. A Solução: criar esse(a) campo/variável na classe onde é mais utilizado. Ver Figura 2, Secção 2.3.

Extract Class

Quando a lógica dentro de uma classe é demasiado grande, essa classe pode ser dividida em duas de forma a melhorar a sua compreensão. A Solução: transformar a classe em duas de forma a simplificar a leitura, utilização e manutenção do código. Ver Figura 3, Secção 2.3.

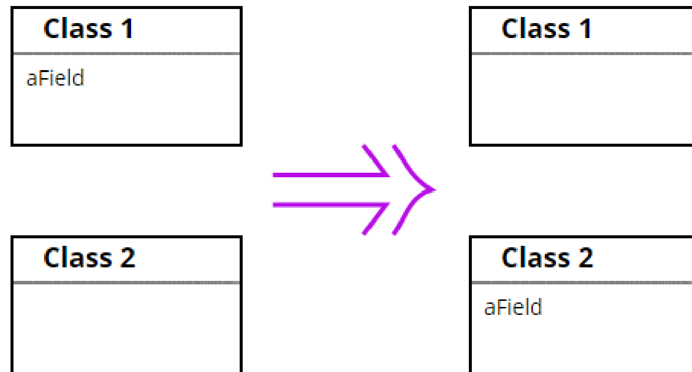


Fig. 2.: Figura Refactoring Move Field.

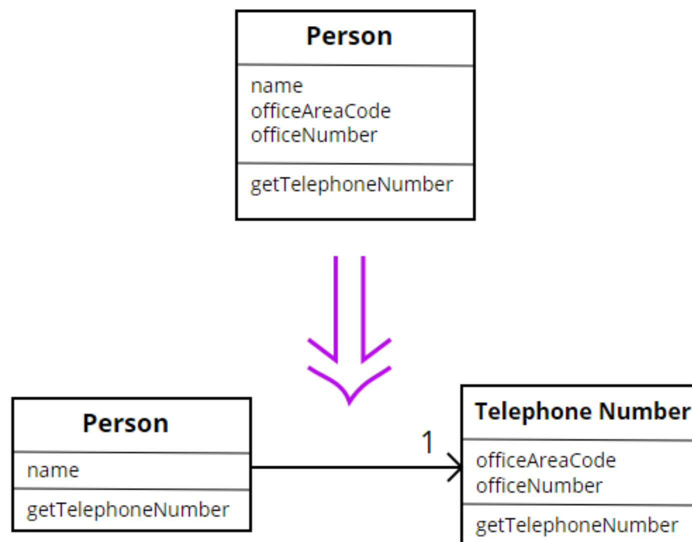


Fig. 3.: Figura Refactoring Extract Class.

Inline Class

Quando a lógica de uma classe depende quase na totalidade da lógica de outra ou outras classes o melhor será juntar essas classes. A Solução: Juntar as classes numa só classe. Ver Figura 4, Secção 2.3.

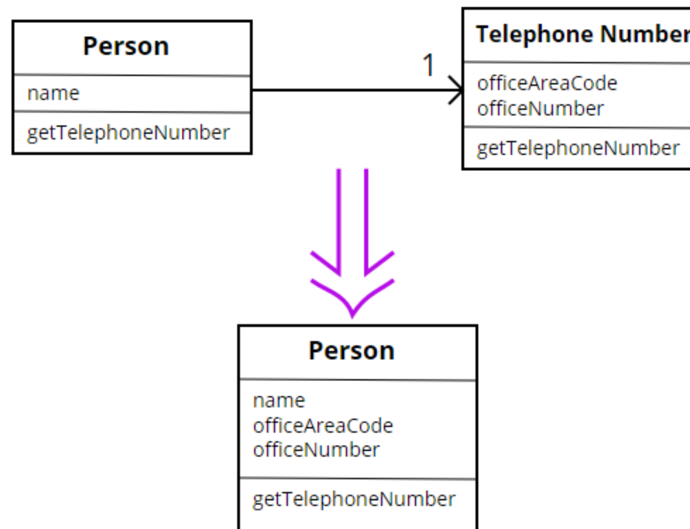


Fig. 4.: Figura Refactoring Inline Class.

Hide Delegate

Este *Refactoring* é usado quando uma classe é composta por várias subclasses. Cada uma dessas subclasses contêm pelo menos um método invocado noutra subclasse. Assim para impedir que métodos comuns a diferentes subclasses estejam espalhados a solução passa por transformar uma subclasse numa classe intermédia transferindo esses métodos para lá resolvendo assim os problemas de delegação de competências. Ver Figura 5, Secção 2.3.

Remove Middle Man

Este *Refactoring* é usado quando num programa existe a seguinte hierarquia: classes, classe intermédia e subclasse. Nessa hierarquia os métodos comuns da classe e da subclasse são delegados à classe intermédia. O problema ocorre quando os métodos comuns da classe intermédia recorrem aos métodos da classe e da subclasse e não contêm lógica suficiente para a manter a classe intermédia como uma classe separada. Nesta situação os métodos comuns devem ser divididos para a outra subclasse e a classe intermédia deve passar a subclasse. A Solução: remover a classe intermédia. Ver Figura 6, Secção 2.3.

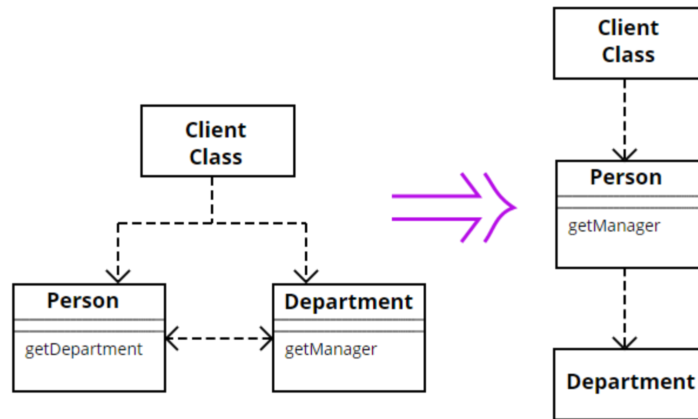


Fig. 5.: Figura Refactoring Hide Delegate.

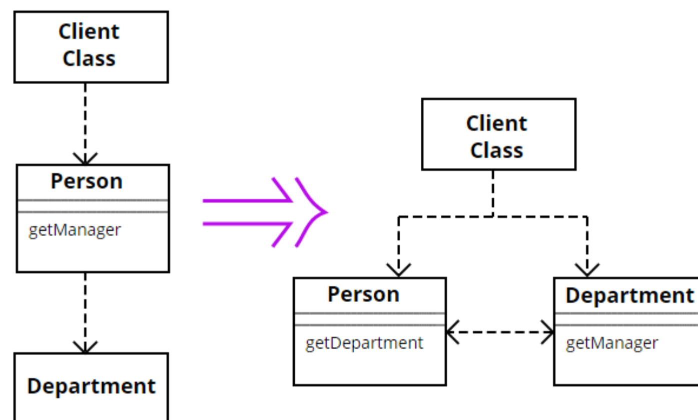


Fig. 6.: Figura Refactoring Remove Middle Man.

Change Bidirectional Association to Unidirecional

Quando existe uma ligação bidirecional entre duas classes através dos seu métodos e uma das classes já não necessita da outra. A Solução: quebrar a ligação que já é não útil. Ver Figura 7, Secção 2.3.

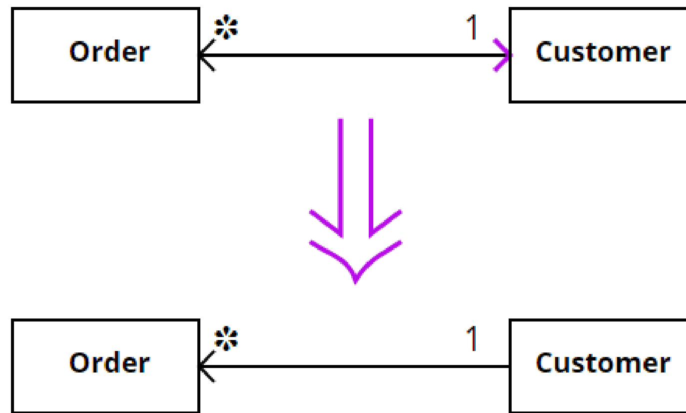


Fig. 7.: Figura Refactoring Change Bidirectional Association to Unidirecional.

Replace Delegation with Inheritance

E por fim o *Replace Delegation with Inheritance*, usado quando uma subclasse usa apenas parte da interface da superclasse ou não quer herdar dados dela. Neste caso o que se deve fazer é copiar os campos da superclasse que a subclasse utiliza para esta segunda, ajustar os métodos e romper a hierarquia. Ver Figura 8, Secção 2.3.

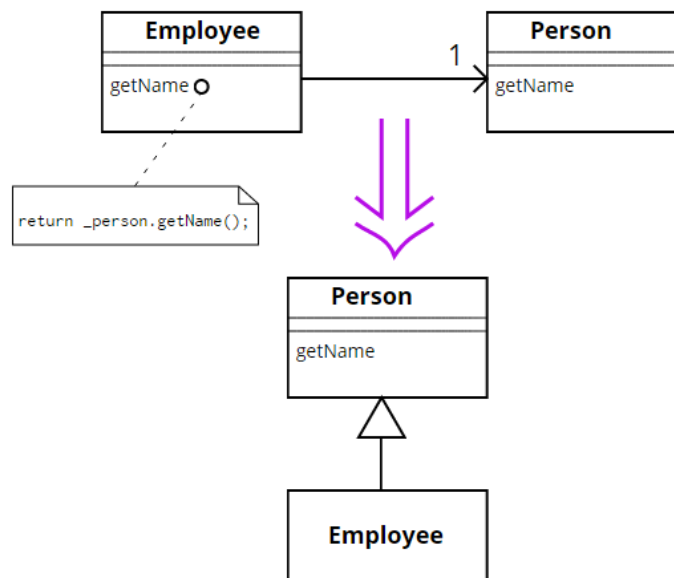


Fig. 8.: Figura Refactoring Replace Delegation with Inheritance.

2.3.3 *Inter-Worksheets Smells de Felienne Hermans* et al.

Nesta secção é apresentado o trabalho de Hermans et al. (2012) sobre *Inter - Worksheets Smells* isto é: deteção de *code smells* em folhas de cálculo.

O objetivo principal do projeto foi responder às seguintes questões:

- R1 - Qual o *smell* mais encontrado nas folhas de cálculo?
- R2 - De que forma um *inter-worksheet smell* expõe ameaças na qualidade de uma folha de cálculo (spreadsheet) e na integridade dos cálculos?
- R3 - De que forma a ferramenta *Data Flow Diagrams* (criada pelos autores) é útil e apropriada para visualizar *inter-worksheets smells*?

Inicialmente o grupo começou por estudar os *code smells* de Martin Fowler apresentados na Secção 2.3.1. Além do trabalho de Fowler, o grupo analisou o trabalho de outros autores sobre métricas de software, métricas para detetar classes suspeitas(ou classes com falhas), obtenção de thresholds para métricas, entre outros. O primeiro passo que os autores fizeram foi criar uma ponte entre a programação orientada a objetos e as folhas de cálculo, ou *spreadsheets*. Assim eles definiram que as *worksheets* seriam as classes compostas por métodos que seriam as células.

Depois de criar este elo de ligação entre a POO e as *spreadsheets* os autores definiram uma lista de 4 *smells* que melhor se identificavam:

- *Inappropriate Intimacy*: Uma *worksheet* tem muitas dependências ou detalhes de implementação de outras *worksheets*.
- *Feature Envy*: Quando uma célula tem mais interesse em aparecer noutra *worksheet* do que na qual se encontra.
- *Middle Man*: Quando uma *worksheet* atribui maior parte das operações a outras *worksheets* e não contêm lógica suficiente para ser uma *worksheet* separada.
- *Shotgun Surgery*: Quando uma pequena alteração cria muitas alterações noutras *worksheets*.

Após escolherem os *code smells* mais apropriados passaram a uma fase de deteção automática dos mesmos. Para cada um dos quatro casos criaram métricas que os identificassem. Para detetar o *Inappropriate Intimacy* investigaram a quantidade de dados ligados entre duas *worksheets*, contaram a quantidade de ligações e calcularam a *Intimacy*:

$$Intimacy(w_0, w_1) \equiv |\{(c_0, c_1) \in Ks : c_0 \in w_0 \wedge c_1 \in w_1 \wedge w_0 \neq w_1\}|$$

Ou seja a *Intimacy* é o número de pares no *ConnectionSet* (total de ligações entre *worksheets*) onde a célula c_0 pertence a worksheet w_0 e a c_1 pertence à w_1 e $w_0 \neq w_1$. Para calcular o máximo de *Intimacy* de uma *worksheet* calcula-se o máximo de *Intimacy* entre ela e as ligações que lhe acedem.

Passando para o *Feature Envy* o grupo definiu que quando uma célula tem este *smell* então a *worksheet* a que pertence é "*smelly*". Para detetarem este *smell* inspecionaram todas as fórmulas e verificaram todas as células que fazem e lhe fazem referência. Contaram assim todas as ocorrências das células com fórmulas que fazem referência a outras células noutras *worksheets* dentro da mesma *spreadsheet* e chegaram há seguinte métrica:

$$FE(c_0) \equiv |\{(c_0, c_1) \in Ks : c_0 \in w \wedge c_1 \notin w\}|$$

Após o *Feature Envy* analisaram o *Shotgun Surgery* onde chegaram a conclusão que para detetar este *smell* era necessário contar as células ou fórmulas que alteram o seu comportamento quando as células ou fórmulas a que fazem referência também alteram. Para isso chegaram a duas fórmulas, a primeira para contar o total de fórmulas alteradas e a segunda com o objetivo de contar o total de worksheets onde o smell ocorre.

$$ChangingFormulas = |\{(c_0, c_1) \in Ks : c_0 \in W \wedge c_1 \notin W\}|$$

$$ChangingWorksheets = |\{w_1 \in S : (\exists (c_0, c_1) \in Ks : c_0 \in w_1 \wedge c_1 \in w_0)\}|$$

Por último analisaram e definiram o *Middle Man*, este é detetado quando uma worksheet é usada para passar valores a outra worksheet. O risco de *Middle Men Smell* aumenta sempre que existem cadeias de cálculos entre fórmulas. Para detetar este *smell* contaram o número de *Middle Mens* que aparecem nas fórmulas de uma worksheet que são referênciadas por outro *Middle Men* criando a seguinte métrica:

$$MM(w) = |\{(c_0, c_1) \in Ks : c_1 \in w \wedge MMF(c_0) \wedge MMF(c_1)\}|$$

MMF() return Boolean value

Depois de determinarem as métricas para encontrar cada um dos *smells*, o grupo estabeleceu um *threshold* através da análise da distribuição das métricas em várias spreadsheets. Os resultados obtidos foram os seguintes:

- Todas as métricas seguiam uma distribuição *Power-law*.
- Os *smells* com maior risco de aparecerem são o *Inappropriate Intimacy* e o *Middle Man*.
- Os *smells* com menor risco de aparecerem foram o *Feature Envy* e o *Shotgun Surgery*.

O passo seguinte foi desenvolver uma ferramenta que criasse *Data Flow Diagrams* [3] das relações entre as worksheets e apresentasse os *smells* em cada uma delas e as respetivas células.

3 Os Data Flow Diagrams são representações gráfica do "fluxo" de dados, fornecendo assim uma visão estruturada do sistema.

Procederam à avaliação de duas formas: a primeira quantitativa onde foi analisada a ocorrência de *smells* no *Euses Spread Corpus* (uma base de dados com milhares de spreadsheets de diversas áreas) permitindo assim a resposta a **R1**, a segunda avaliação foi mais qualitativa com o intuito de entender melhor **R1** e responder a **R2** e **R3**. Nesta segunda avaliação foram "usados" 10 programadores que utilizaram as suas mais importantes spreadsheets. Os 10 profissionais pertenciam ao banco *Robeco* (Alemão). Os resultados foram que o *Feature Envy* é o smell mais comum de aparecer e em seguida o *Inappropriate Intimacy*. Os dois estão relacionados visto que o *Feature Envy* pode originar o *Inappropriate Intimacy*. Finalmente questionados os 10 participantes sobre cada um dos *smells* e sobre a utilização de *data flow diagrams* chegaram aos seguintes resultados:

- Todos os participantes ficaram admirados com a representação dos *data flow diagrams* e com eles encontraram ligações que nem eles se lembravam que tinham.
- O *Inappropriate Intimacy* é difícil de detetar pelos utilizadores de spreadsheets, além disso demoram muito tempo a perceber quais as células que estão ligadas. Alguns utilizadores não acreditam que este *smell* seja um problema.
- O *Feature Envy* aparece quando pelo menos 3 das referências a uma formula estão noutras worksheets. O *highlight* das células nos *data flow diagrams* ajudou os utilizadores a visualizar as fórmulas onde ocorria este *smell*. Os *data flow diagrams* serviram para detetar o *smell* contudo obriga os utilizadores a "mergulhar" nas worksheets para entenderam as ligações.
- No caso do *Middle Man* os utilizadores não se importaram de ver os valores das células a passarem de uma *worksheet* para o outra porque os ajudava a relembrar os valores em causa. Um dos pontos que levou a esta conclusão foi o elevado detalhe e tamanho das spreadsheets e a quantidade de *worksheets* que compões cada uma delas. Quanto aos *data flow diagrams* os utilizadores acharam uma boa solução.
- Finalmente no *Shotgun Surgery* os utilizadores acharam que o *smell* provém de uma deficiência no desenho/criação das spreadsheets. Notaram também que sempre que os utilizadores visualizam este *smell* ficam com necessidade de alterar a spreadsheet.

3

USABILITY SMELLS

3.1 INTRODUÇÃO

O desenvolvimento de grandes e complexas aplicações interativas tem de seguir rigorosos princípios de engenharia de software, como, documentação do software, testes de software, *design patterns*, melhores práticas de qualidade de software, de forma a minimizar o tempo e custo de produção de software. A utilização destes princípios além de reduzir estes dois fatores torna possível a manutenção e evolução destes enormes e complexos sistemas.

A comunidade de programadores tem vindo a realizar um enorme trabalho definindo novas técnicas, atualizando as antigas, e melhorando ferramentas que ajudem os engenheiros de software a desenvolver estes complexos sistemas. Por exemplo, o custo de manutenção de software é um tópico bastante discutido por toda a comunidade Lee et al. (2014)

Quando se constrói uma aplicação, deve-se manter o devido cuidado em todos os aspetos relevantes na sua implementação, desde a sua arquitetura, até à interface com o utilizador. Para exercer este cuidado uma possível solução passa por utilizar métricas de software tanto na aplicação como na seus requisitos funcionais e não funcionais.

Métricas de software podem ser ferramentas úteis, não apenas durante o desenvolvimento mas também durante a manutenção dos sistemas para detetar potenciais problemas. Estas métricas providenciam uma visão abstrata do código que permite detetar potenciais problemas na sua qualidade. Martin Fowler capturou esta noção sob o conceito de *Code Smells*,Fowler (2002).

Quando se desenvolvem aplicações têm-se a maioria das vezes o cuidado de manter a aplicação limpa, isto é tornar o código legível e de fácil manutenção para quem o escreve e lê. Desta maneira os utilizadores percebem melhor para que serve a ferramenta e como funciona sem a necessidade de se agarrem logo aos manuais. O estudo de Myers and Rosson (1992) aponta que 50% do tempo gasto na implementação da aplicação destina-se à *user interface* mostrando assim a importância da usabilidade e do estudo dela.

Este capítulo começa por apresentar a definição de *Usability Smells* e o porquê do seu estudo e em seguida apresenta o catálogo de *Usability Smells* desenvolvido avaliado no capítulo 4.4. O Catálogo é um dos pontos propostos na metodologia de trabalho, onde se começa a definir a ponte entre código e usabilidade de aplicações interativas.

3.2 O QUE SÃO USABILITY SMELLS?

Usability Smells como o próprio nome indica são *smells* de usabilidade. A ideia surgiu ao estudar os *code smells* de Fowler. Tanto no código como na interface os erros e anomalias são comuns e obrigam a uma manutenção constante. O programador é assim obrigado a aumentar o tempo gasto na manutenção se tiver como objetivo "tentar" manter a legibilidade da aplicação. Assim, surgiu a questão de saber se esses *smells* não poderiam existir na interface gráfica das aplicações.

Antes de apresentar a definição de *Usability Smells* é preciso referir que um dos primeiros passos a realizar nesta tese foi desenvolver uma ponte entre a programação orientada a objetos e as aplicações interativas/web. Sem este elo de ligação não seria possível mapear os *code smells* para *usability smells*. Assim este mapeamento é feito da seguinte maneira: considera-se que uma classe é o equivalente a uma página e um método a todos os componentes que mexam com a lógica da aplicação, exemplo: botões, *text fields*, *tabs*, etc.

Os *Usability Smells* como os *Code Smells* não são considerados erros mas sim anomalias na aplicação que tornam difícil a sua compreensão, manutenção e evolução. O aparecimento deste tipo de *smells* dá indicação ao programador que a aplicação necessita de alterações. Olhando para algumas aplicações notamos que a sua estrutura às vezes não é a mais correta e isso influência as tarefas a realizar, por exemplo:

- páginas web a apresentar informação sobre outras páginas em posições privilegiadas dando-lhes um maior ênfase,
- botões, caixas de texto, entre outros componentes inseridos em locais fora do seu contexto,
- páginas a delegar tarefas a outras páginas desnecessariamente...

Nestes casos as aplicações não estão erradas contudo algo não "está bem" e nestas situações o utilizador provavelmente percebe que as tarefas que tenta realizar não estão a ter a performance desejada. O estudo de *Usability smells* tem o objetivo de encontrar estes e outros casos de anomalias, perceber a que ponto os *Usability smells* se encaixam neles e estudar possíveis resoluções ou *refactorings*.

3.3 CATÁLOGO DE USABILITY SMELLS

Na deteção de *Usability Smells* o primeiro passo a realizar é a instanciação dos *code smells* para aplicações interativas. Com este passo é possível numa primeira fase perceber o que é o *smell* e como é que ele ocorre. Para que tal aconteça optou-se por, em vez dos 22 *code smells* de Fowler, seguir a metodologia de Hermans et al. (2012) e instanciar os mesmo quatro *smells* que eles utilizaram (*Inappropriate Intimacy*, *Feature Envy*, *Middle Man* e o *Shotgun Surgery*) e posteriormente, consoante os resultados, acrescentar mais smells ao catálogo. Além destes identificamos mais dois no caso de estudo que será apresentado posteriormente, *Information Overload* e *Too Many Layers*.

Dividiram-se estes seis *smells* em três grupos: *smells* de implementação, de design e de domínio.

O primeiro grupo, o grupo dos *Smells* de implementação ocorre no esqueleto da aplicação, na forma como é construido o código e afeta de certa forma a usabilidade da aplicação através da sua organização. Este grupo é composto por: *Shotgun Surgery* e *Too Many Layers*.

O segundo grupo, o grupo de *Smells* de design representa um conjunto de *Usability Smells* que não estão relacionados com a estrutura da aplicação contudo afetam a sua interface. Este grupo é composto por: *Middle Man* e *Information Overload*.

Finalmente o terceiro grupo, o grupo dos *smells* de domínio ocorre em páginas em que a informação que contêm pertence ao mesmo contexto, isto é, páginas que apresentam tipo de informação comum. Imagine-se duas páginas que apresentam informação sobre a área de informática: a primeira página apresenta tutoriais sobre Java e a segunda página apresenta um caso de estudo sobre as programações orientadas a objetos onde a linguagem em foco é o Java. Estas duas páginas pertencem a área de informática, ao mesmo domínio científico e apresentam informação comum. Este grupo é composto por: *Inappropriate Intimacy* e o *Feature Envy*.

O seis *Usability smells* serão a seguir apresentados.

3.3.1 *Usability Smell Shotgun Surgery*

O primeiro *smell* a ser catalogado foi o *Shotgun Surgery*. Introduziu-se este smell no grupo dos *Smells* de Implementação. Relembrando como Fowler e Hermans o definem:

- **Fowler:** Este smell ocorre quando um método é invocado em diversos lugares e a alteração da sua assinatura obrigada a alteração em todos esses locais. É um smell que dá bastante trabalho a corrigir e é necessário saber todas as classes e métodos que se deve alterar quando o método ou classe a que fazem referência está a ser alterado.

- **Hermans:** Quando uma alteração afeta uma grande quantidade de células ou *worksheets* levando também à sua alteração. (Impacto na manutenção do código).

Acredita-se que este *smell* ocorre por causa de reaproveitamento de código. Considera-se o reaproveitamento de código o ato de desenvolver funções que possam ser utilizadas em diversos locais da estrutura de uma aplicação através da sua invocação. O problema surge quando o volume de código de uma aplicação começa a ser elevado. Nesta situação o programador perde noção de todos os locais onde invocou as funções.

Se uma função for invocada em 10, 100 ou mais classes a certo ponto o engenheiro de software não se vai conseguir lembrar de todos os lugares onde a chamou. O maior problema acontece quando a lógica desse bloco de código é alterada, neste caso todas as classes onde a função foi invocada também irão mudar. Se o programador não se lembrar de todas as classes poderá estar a afetar gravemente a aplicação. O mesmo acontece nas aplicações interactivas/web, é normal haver reaproveitamento de código. Diversas vezes, por exemplo, utiliza-se o mesmo *form* em diferentes páginas. Quando o

número de páginas começa a ser muito grande provavelmente a utilização do mesmo *form* poderá não ser o mais adequado. Pode existir uma das página que utiliza o *form* com objetivos diferentes das outras páginas. Nesta situação se mudarmos o *form* para se adaptar a essa página a sua representação visual também muda nas outras páginas onde está a ser utilizado. É nesta altura que surge o *Shotgun Surgery*, uma página com um visual correto e outras com o visual errado. O maior problema causado por este *smell* é conseguir identificar todas as páginas onde o *form* foi utilizado para serem corrigidas.

Para se compreender melhor este smell vejamos a Figura 9. Este exemplo apresenta três páginas de uma aplicação de gestão pessoal de conteúdo multimédia. Na aplicação o utilizador pode registar músicas e filmes. Em cada um dos casos caso o utilizador pretenda pode acrescentar informação acerca do autor dos conteúdos. Neste exemplo o programador considerou reaproveitar o *form* utilizado para acrescentar músicas e filmes à sua base de dados para inserir também informação sobre os autores de cada conteúdo. Imagine-se que a dimensão da aplicação é muito grande e que o programador utilizou esse *form* em mais do que estas 3 páginas apresentadas. O programador ao testar a aplicação verifica que o *form* no autor não é o mais apropriado alterando-o de forma a responder às suas expetativas. Como a dimensão da aplicação é muito grande e o programador não se recorda de todos os locais onde invocou o *form* a alteração vai afetar as outras páginas que também o utilizam. É nesta altura que aparece o *Shotgun Surgery*. A alteração do *form* obriga à alteração de todas as páginas onde é invocado.

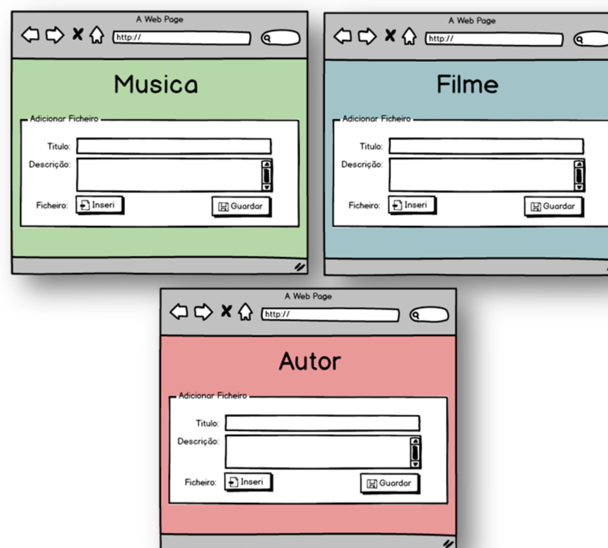


Fig. 9.: Exemplo do Shotgun Surgery.

3.3.2 *Usability Smell Too Many Layers*

O *Too Many Layers* foi um *smell* encontrado após executar várias vezes o caso de estudo utilizado para validar todos estes *Usability Smells* (ver Capítulo 5).

Este é o segundo *smell* introduzido no grupo de implementação porque o seu aparecimento afeta diretamente toda a estrutura da aplicação. Considera-se que este *smell* aparece quando uma aplicação tem demasiadas páginas e, para realizar uma tarefa, os utilizadores finais têm de passar por pelo menos três delas. Passar por várias páginas para realizar uma determinada tarefa afeta severamente o seu desempenho levando até a um possível esquecimento do que o utilizador está a fazer.

Para compreender melhor o *Too Many Layers* vejamos a Figura 10. A figura é do sistema operativo Ubuntu[1]. Neste exemplo pode-se ver como introduzir um novo repositório de aplicações no sistema operativo. Para realizar esta tarefa o utilizador tem de passar por 4 páginas. Este é um exemplo do *Too Many Layers* onde a grande quantidade de páginas afeta a realização da tarefa e a legibilidade da aplicação.

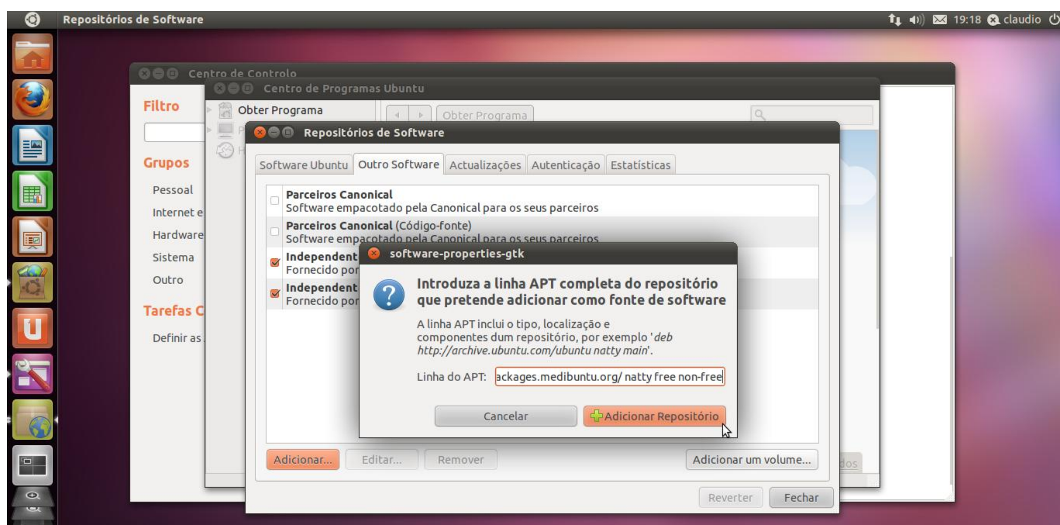


Fig. 10.: Exemplo do Too Many Layers retirado do site: http://ubuntued.info/wp-content/uploads/2011/04/9_adicionar_repositorio.jpg

3.3.3 *Usability Smell Middle Man*

O terceiro *smell* a ser catalogado foi o *Middle Man*, segundo Fowler e Hermans:

- **Fowler:** ocorre quando existem pelo menos duas classes (ou métodos) e uma delas delega competências à outra, competências que poderiam ser realizadas pela primeira classe. Entende-se como competência a passagem de lógica interna à classe.

1 Sistema operativo de código aberto construído a partir de um núcleo Linux.

- **Hermans:** Quando uma *worksheet* delega a maioria das suas operações a outras *worksheets* e não contém lógica suficiente para ser uma *worksheet* separada.

O *Middle Man* é um dos smells que mais facilmente se consegue inserir no catálogo de *Usability Smells*. Considera-se que, para que este smell ocorrer, existam pelo menos duas páginas e que a primeira delega competências à segunda. A anomalia ocorre quando a informação ou tarefa que foi delegada à segunda página aparece na primeira página.

Introduziu-se o *Middle Man* no grupo dos *Smells* de Design. Em todos os caso que o *Middle Man* ocorreu verificou-se que o problema era devido a uma má disposição dos conteúdos da página ou para simplificar tarefas, havendo assim delegação de alguns componentes a outras páginas.

Para se perceber melhor este *smells* vejamos a Figura 11. As páginas foram retiradas da aplicação web *Twitter*[2] que apresenta o sistema de autenticação. Neste exemplo a página de *Login* do sistema delega a uma segunda página a informação de autenticação do utilizador. Como é percetível a página **A** delega tarefas para a página **B** quando esta a poderia fazer sem qualquer problema. A questão que se coloca é: a que página o utilizador irá dar mais atenção? Neste caso o o utilizador é obrigado a focar a atenção numa página diferente, ele só saberá que entrou no sistema quando vir as páginas que se sucedem à entrada no sistema contudo, caso a autenticação não seja aceite este não saberá qual terá sido o erro que levou a tal situação. A página incumbida dessa tarefa já não existe e a página que lhe incumbiu esse objetivo não o sabe concretizar.

3.3.4 *Usability Smell Information Overload*

O excesso de informação é um dos mais típicos *Usability Smell*. Este *smell* ocorre quando os engenheiros de software introduzem informação a mais do que aquela que é necessária. Este *smell* pode ser encontrado em muitas aplicações *web* e moveis. No último caso este *smell* aparece por exemplo sob a forma de sistemas de publicidade. Estes excessos de publicidade afetam a usabilidade da aplicação e forçam os utilizadores finais a visitar *websites* só para poder realizar a tarefa que pretendem. Outro exemplo é quando existem excesso de componentes como botões ou links que não são úteis ou não têm tarefas associadas.

Para compreender melhor o *Information Overload* vejamos por exemplo a imagem 12. A imagem foi retirada do site: `http://www.mimarch.net`. O site da empresa *Mimarch* foi considerado um dos 25 piores *websites* de 2013 por Vincent Flanders' no site *Web Pages That Sucks*[3]. Neste exemplo é percetível um excesso de informação visual em forma de menus e imagens. Este excesso de informação é o principal obstáculo para o utilizador perceber que tarefas existem e que são possíveis de realizar na aplicação.

2 `https://twitter.com/` Visualizada pela última vez a 10 de Agosto de 2014
3 `http://www.webpagesthatsuck.com/worst-websites-of-2013.html`

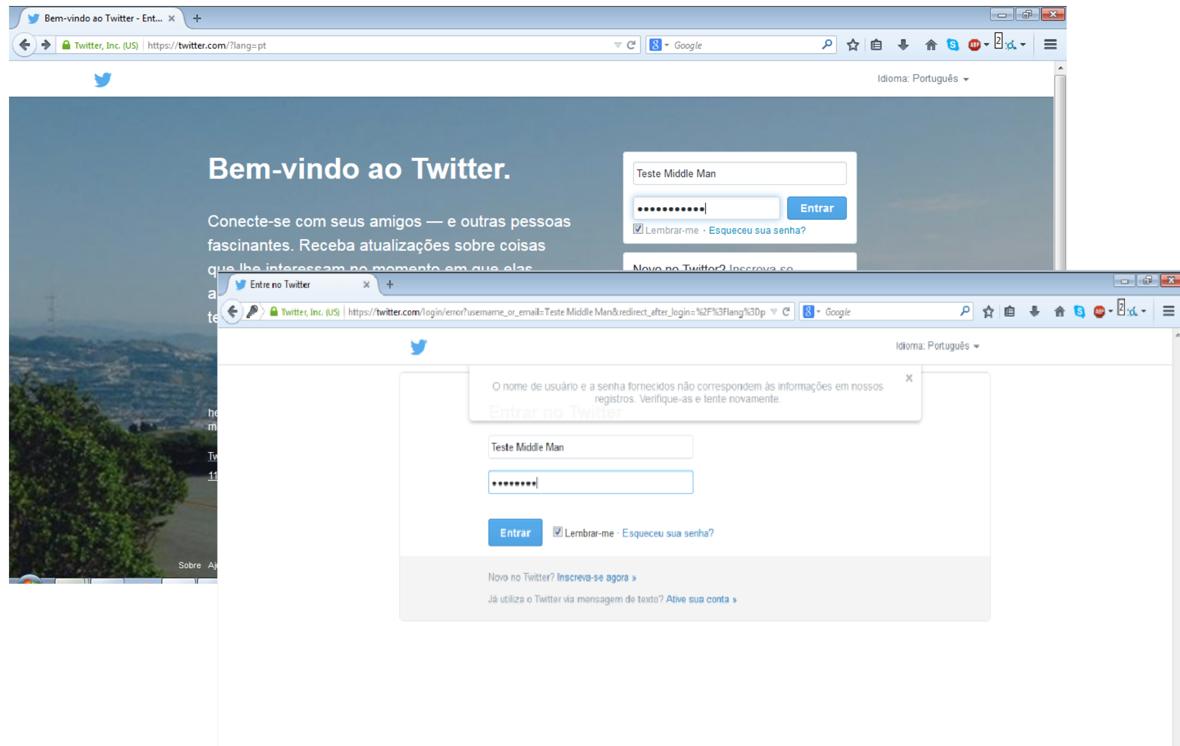


Fig. 11.: Exemplo do Middle Man.

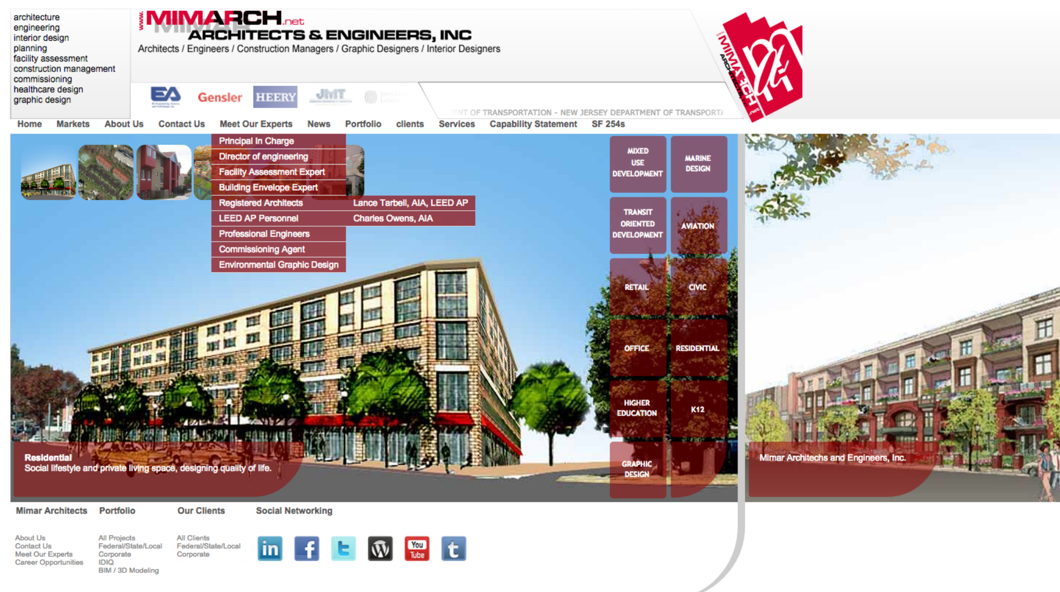


Fig. 12.: Exemplo do Information Overload. Imagem retirada do site: http://www.mimarch.net

3.3.5 *Usability Smell Inappropriate Intimacy*

O quinto *Usability Smell* a apresentar é o *Inapproriate Intimacy*, recordando como Fowler e Hermans o descrevem:

- **Fowler:** As classes normalmente tornam-se muito intimas de outras classes, gastando mais tempo nos métodos privados dessas classes que nos próprios métodos.
- **Hermans:** O Inappropriate Intimacy é um smell que aparece quando uma *worksheet* tem muitas dependências ou detalhes de implementação de outras *worksheets*.

O *Inappropriate Intimacy* ocorre quando duas páginas pertencem ao mesmo domínio, o caminho para elas é diferente e têm pelo menos uma tarefa de uma das páginas que depende da outra página. Se as duas páginas pertencerem ao mesmo domínio é mais fácil para o utilizador final realizar a tarefa se as duas páginas estiverem juntas ou o caminho para elas for o mesmo.

Vejamos por exemplo a Figura 13, neste exemplo concreto de *Inappropriate Intimacy* são apresentadas 3 páginas com sequência temporal que pertencem a um sistema de publicações online de livros (o domínio a que pertencem é o do registo do livro): a primeira página tem como objetivo registar um autor, ao clicar no botão "Seguinte" o utilizador é reencaminhado para a segunda página onde deve inserir uma publicação e finalmente uma terceira página informativa. Neste exemplo podemos reparar que o utilizador introduziu um livro contudo não introduziu um autor. O sistema reconhece essa falha e quando o utilizador clica em "Seguinte" na página do livro é apresentada a informação que não foi introduzido nenhum autor e que para continuar deve retroceder à devida página e completar a informação.

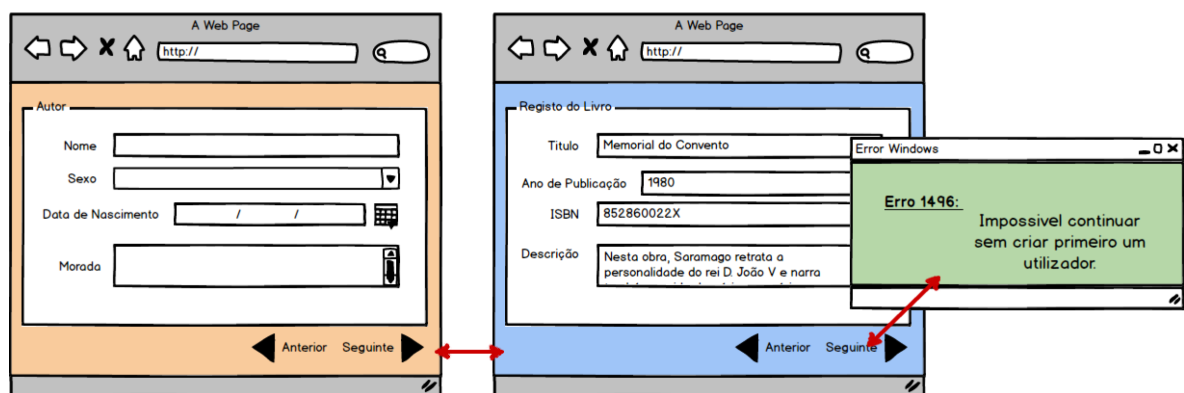


Fig. 13.: Exemplo do Inappropriate Intimacy.

O *Inappropriate Intimacy* ocorre porque as tarefas da página do livro(2) dão mais interesse à informação da página do autor(1) do que à da própria, influenciando assim o aparecimento da página de erro. Uma possível solução para este problema seria transferir a validação do autor presente na página dois para a primeira página, assim a página de erro apareceria ao clicar em "Seguinte" na

primeira página caso não fossem preenchidos os erros. Como podemos ver na Figura 13 a página 2 tem mais interesse na informação da página 1 que na dela própria. Se as tarefas destas páginas pertencentes ao mesmo domínio estivessem juntas seria mais fácil para o utilizador final.

3.3.6 *Usability Smell Feature Envy*

Além do *Inappropriate Intimacy* o *Feature Envy* é outro *smell* de domínio. Segundo Fowler e Hermans:

- **Fowler:** O Inappropriate Intimacy é um smell que aparece quando uma classe tem muitas dependências ou detalhes de implementação de outras classes tornando-se muito íntimas.
- **Hermans:** Uma *worksheet* tem muitas dependências ou detalhes de implementação de outras *worksheets*.

O *Feature Envy* ocorre quando uma página contêm pelo menos uma operação pertencente a outra página e essa operação não se encontra no respetivo local. Introduzimos o *smell* neste grupo porque as anomalias criadas por este *smell* estão relacionadas diretamente com a interface entre páginas do mesmo domínio e não com a estrutura do código. A relação entre páginas e a não colocação dos operações no devido lugar são os principais fatores para que seja introduzido nesse grupo.

Vejamos o exemplo da Figura 14 onde aparece um sistema de gestão de clientes de uma seguradora. Neste exemplo pode-se ver duas páginas: a primeira da “Correctora” onde são geridos os seguros dos clientes e a segunda a “Carteira” onde estão todas as operações para visualizar as carteiras dos clientes. No exemplo em questão a página da “Correctora” apresenta dois botões: “inserir carteira” e “remover carteira”, podemos considerar que estes dois botões pertencem à página de “Carteira”. Estes dois botões, ou operações como referido acima, encontram-se fora do lugar. Se pertencem à página da “Correctora” então deveriam estar nessa página e não na da “Carteira”. Relembrando o mapeamento entre as POO e os *Usability Smells* onde as páginas são classes verificamos que existem duas classes e que uma contêm duas operações que seriam os métodos que têm mais interesse na informação da outra classe que na informação da própria, ou seja, tal e qual como Fowler definiu o *Feature Envy*, ver Subsecção 2.3.1.

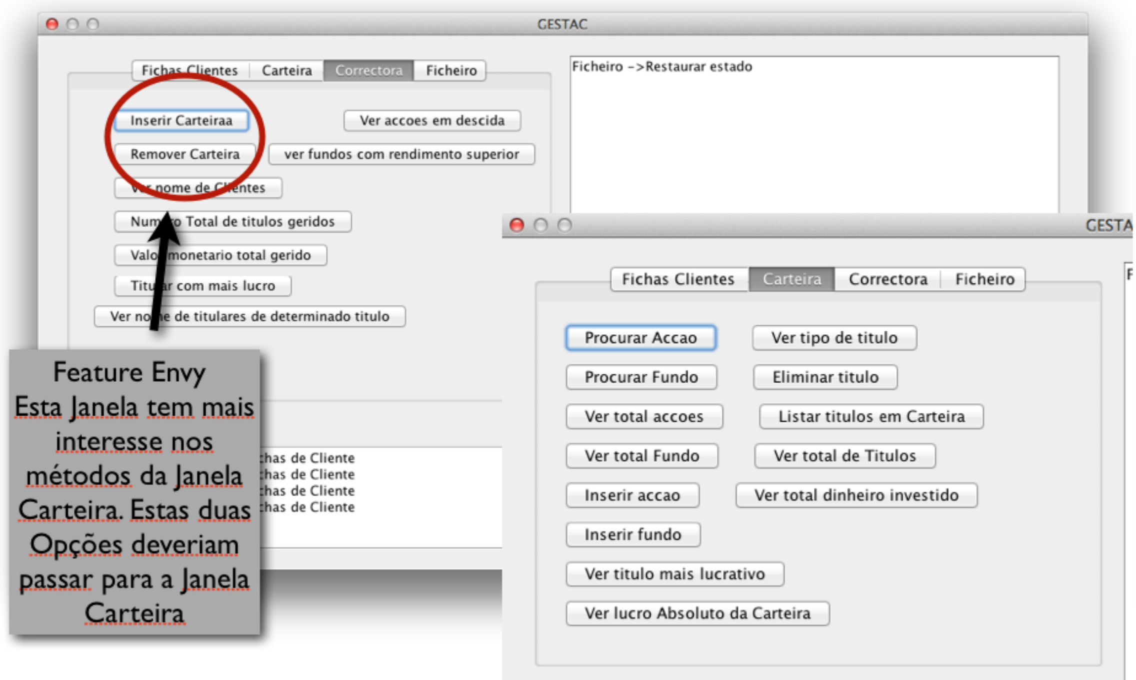


Fig. 14.: Exemplo do Feature Envy.

4

VALIDAÇÃO DOS USABILITY SMELLS

4.1 INTRODUÇÃO

Neste capitulo irá ser descrito o estudo realizado desde a sua definição até à avaliação do questionário com o objetivo de validar os *Usability Smells*. Pretende-se com este estudo obter reações e respostas a algumas questões como:

1. Qual a opinião dos programadores acerca destes *Usability Smells*?
2. Os *Usability Smells* realmente afetam a usabilidade da aplicação?
3. Que tipo de *Refactoring* os programadores apresentam para resolver estas anomalias?
4. Qual o melhor *Refactoring* para cada um dos *Usability Smells*?

4.2 DADOS, METODOLOGIA E PRÉ-TESTE

Em primeiro foi definido o tipo de dados mais apropriado para responder a estas questões e como seriam recolhidos. Foram escolhidos dados primários para validar as respostas. Optou-se por criar de raiz todo o questionário como ferramenta de recolha de dados sem recorrer a dados secundários provenientes de fontes externas. Um dos motivos para usar dados primários e criar o questionário deve-se a não ter sido encontrado nenhum estudo sobre *Usability Smell* onde fosse usada a aplicação *Open Hospital* (descrita na Secção 4.3). Outro motivo que levou à escolha do questionário como ferramenta foi o trabalho de Yin (2013) que aponta existirem três critérios para ajudar na seleção do método de recolha de dados: o tipo de questão de investigação, o grau de controlo que o investigador tem sobre os acontecimentos comportamentais reais e o nível de focalização que se quer fazer a acontecimentos atuais versus acontecimentos históricos. Relativamente aos dados obtidos no questionário estes foram tratados estatisticamente. A analise dos dados foi qualitativa permitindo assim avaliar a qualidade do caso de estudo e analisar o desempenho dos participantes durante o estudo.

O questionário criado é composto por 5 tarefas cada uma delas fazendo referência a um *Usability Smell*. Cada tarefa é focada num exemplo encontrado na aplicação de cada um dos *Usability Smells*

listados no catálogo (Secção 3.3). Cada uma das 5 tarefas é composta por 3 grupos. O primeiro grupo apresenta ao participante o *Usability Smell*. O objetivo deste grupo, recorrendo a perguntas de escolha múltipla dicotómica avaliadas numa escala de *Likert* de cinco níveis (Discordo Totalmente **DT**, Discordo Parcialmente **DP**, Indiferente **I**, Concordo Parcialmente **CP** e Concordo Totalmente **CT**) é validar se os participantes consideram o *Usability Smell* realmente uma anomalia. O segundo grupo composto por uma pergunta de resposta aberta tem como objetivo obter um conjunto de *Refactorings* capazes de eliminar o *Usability Smell*. Finalmente no terceiro grupo é apresentado uma proposta de *Refactoring*. Este último grupo é composto por questões de escolha múltipla dicotómicas também avaliadas numa escala de *Likert* e uma questão de resposta aberta. O objetivo deste grupo é validar o *Refactoring* proposto.

Optou-se por fazer inicialmente um pré-teste. O pré-teste teve como objetivo verificar se a aplicação usada e o questionário estavam enquadrados e adequados ao estudo, e se os participantes entendiam o que era pedido em todas as tarefas que tinham de realizar. Considera-se que a escolha dos participantes para a amostra foi por conveniência uma vez que se pretendia algum conhecimento de linguagens de programação. Desta forma foi pedido a 5 alunos da Universidade do Minho da área da Informática que o realizassem.

O resultado deste pré-teste foi o esperado. O questionário sofreu apenas ligeiras alterações, a maior parte das alterações foi a nível gramatical corrigindo algumas frases onde não era percetível aos participantes o que era pedido. Além destas alterações optou-se por acrescentar um segundo conjunto de escolhas múltiplas para validar melhor os *Refactorings* apresentados para cada tarefa.

4.3 APLICAÇÃO CASO DE ESTUDO - OH OPEN HOSPITAL

A aplicação *OH - Open Hospital* tem como objetivo ajudar a gerir o Hospital *St. Luke* em Angal no Uganda. A aplicação encontra-se desenvolvida em Java e tem uma interface gráfica como se pode ver na Figura 15. A aplicação *Open Hospital* permite realizar pesquisas sobre por exemplo doenças, exames, operações realizadas, vacinas, marcar exames, gerir contas de utentes, gerir medicamentos, ver estatísticas e gerir as finanças do hospital entre outras opções. Considerou-se usar esta aplicação após analise onde foi possível encontrar exemplos de quase todos os *Usability Smells* do catálogo, exceto o *Feature Envy*, apresentados no capítulo 5.

4.4 TESTE DE USABILIDADE

Após realizar o pré-teste e refinar o questionário realizou-se uma avaliação qualitativa. Esta avaliação foi realizada com uma amostra de 12 programadores entre os 18 e 22 anos. De acordo com Nielsen (2012) o tamanho da amostra não deveria importar muito porque é um teste qualitativo com o objetivo de obter perceções que ajudem a validar os *Usability Smells* e busquem formas de os eliminar. Contudo se o teste fosse quantitativo então o tamanho da amostra deveria ter no mínimo 20 partici-

Fig. 15.: Aplicação OH - Open Hospital

pantes (Nielsen, 2012). Para um participante pertencer à amostra o principal requisito foi ter algum conhecimento sobre paradigmas de linguagens de programação, sentir-se à vontade com aplicações interativas/web e ter pelo menos mais de 2 anos de experiência nesta área. Para tal os 12 participantes foram alunos do curso de Engenharia Informática da Universidade do Minho.

Optou-se por realizar o teste durante uma aula por dois motivos. O primeiro motivo é o tempo necessário para realizar todo o teste, segundo o pré-teste cada utilizador necessitaria de aproximadamente uma hora para realizar todas as tarefas. Assim tornar-se-ia bastante difícil em relação ao tempo planeado fazer os testes individualmente. O segundo motivo era a angariação de participantes suficientes para a amostra, durante uma aula é mais fácil conseguir participantes com tempo e disponibilidade.

Iniciou-se o teste apresentando a aplicação. Durante aproximadamente 10 minutos foram apresentadas aos participantes algumas das suas principais funcionalidades. Além da apresentação disponibilizaram-se mais alguns minutos para os participantes poderem interagir com a aplicação e se sentirem à vontade para realizar o teste.

O questionário usado no teste (encontra-se nos anexos), que foi alterado de acordo com os resultados do pré-teste, é também composto por 5 tarefas cada uma delas fazendo referência a um *Usability Smell*. Cada tarefa é focada num exemplo, encontrado na aplicação, de cada um dos *Usability Smells* listados no catálogo (Secção 3.3) com a exceção do *Usability Smell Feature Envy* devido a não ter

sido detetado na aplicação caso em estudo. Cada uma das 5 tarefas é composta por 3 grupos. O primeiro grupo apresenta ao participante o *Usability Smell*. O objetivo deste grupo, recorrendo a perguntas de escolha múltipla dicotómica avaliadas numa escala de *Likert*, é validar se os participantes consideram o *Usability Smell* realmente uma anomalia. O segundo grupo composto por uma pergunta de resposta aberta tem como objetivo obter um conjunto de *Refactorings* capazes de eliminar o *Usability Smell*. Finalmente no terceiro grupo é apresentado uma proposta de *Refactoring*. Este último grupo é composto por questões de escolha múltipla dicotómicas também avaliadas numa escala de *Likert* e uma questão de resposta aberta. O objetivo deste grupo é validar o *Refactoring* proposto. Em média cada participante demorou aproximadamente 15 minutos a responder a cada tarefa, e no total aproximandamente 1 hora e 20 min.

4.5 TESTE DE USABILIDADE - TAREFAS

Nesta secção irá ser apresentada cada uma das 5 tarefas do teste de usabilidade, as perguntas de cada uma das tarefas. O resultados obtidos serão apresentados no capítulo seguinte.

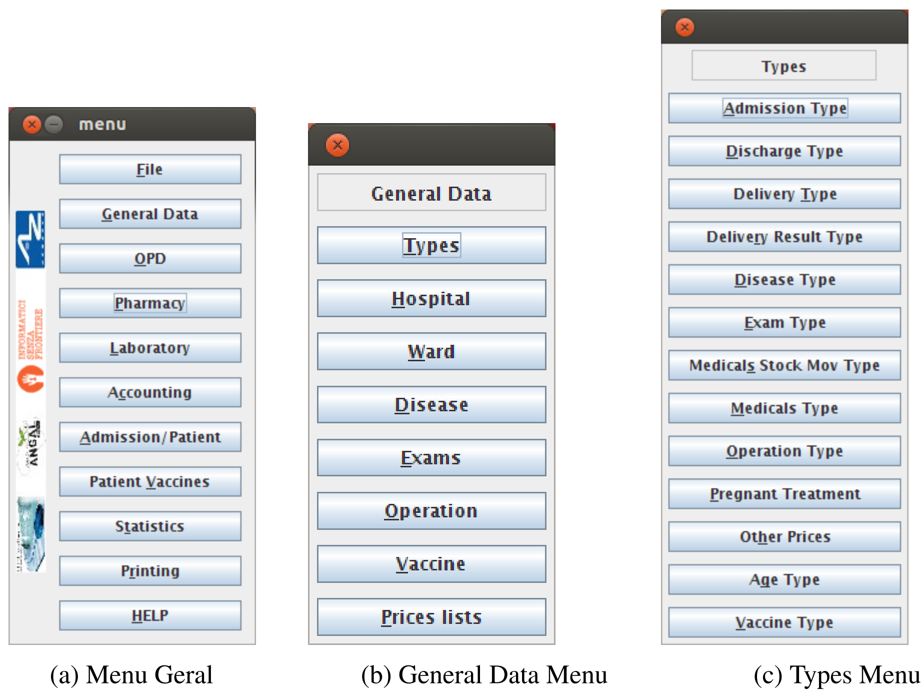
Relembrando, cada uma das tarefas tem como objetivo validar um *Usability Smell*. Serão apresentadas cinco tarefas representando cada um dos cinco *Usability Smell* com a exceção do *Feature Envy* que não foi encontrado neste caso de estudo.

4.5.1 *Tarefa 1: Middle Man*

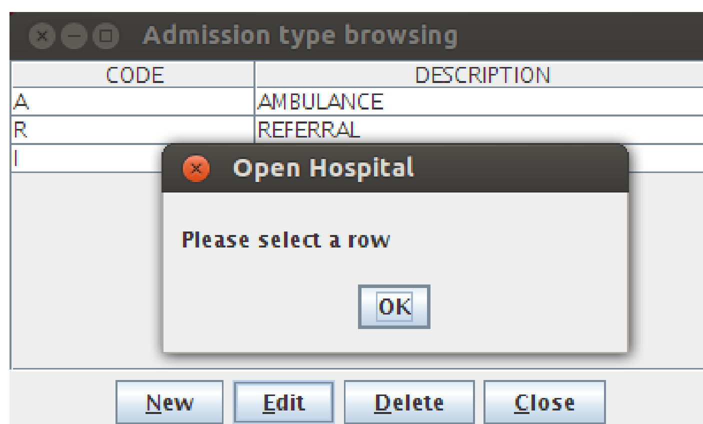
Na primeira tarefa foi pedido aos participantes para editar uma tabela na aplicação *OH*. Para tal os participantes tinham de na aplicação *OH* selecionar as seguintes opções:

General Data >>Types >>Admission Type

e nessa página pressionar o botão *Edit* como mostra a Figura 16d.



(a) Menu Geral (b) General Data Menu (c) Types Menu



(d) Admission Type Menu e exemplo de Middle Man

Fig. 16.: Imagens da Tarefa 1 representando um exemplo do Middle Man

No primeiro grupo da Tarefa 1 foram apresentadas quatro questões para o participante avaliar relativamente ao caso de estudo da aplicação *OH*:

1. O aparecimento de páginas pop-up é a melhor forma de fornecer indicações ao utilizador?

2. Concorda que o aparecimento de uma página só para indicar que nenhum objeto se encontra selecionado o ajuda a realizar a tarefa de forma mais efetiva?

3. Preferia que a informação da página fosse apresentada na mesma página onde se encontra a tabela?

4. Considera este exemplo uma anomalia na conceção da interface?

Ao fim deste grupo de questões foi pedido a cada participante que indicasse que alterações visuais efetuariam para corrigir esta anomalia. O objetivo desta pergunta é obter um conjunto de *Refactorings* provenientes do utilizador (grupo 2).

Após questionar os participantes acerca da aplicação foi-lhes apresentado um exemplo de *Refactoring* para o *Middle Man* apresentado na Figura 17. O *Refactoring* apresentado consiste em transferir a mensagem informativa para a página onde se encontra a tarefa a realizar e atribuir algum realce, neste caso apresentar a mensagem numa fonte de texto mais carregada (ou a "bold" por exemplo) e com uma cor que chame a atenção como por exemplo o vermelho, uma cor que algumas participantes aconselharam. Ao transferir a mensagem para a página da tarefa o utilizador final teria menos trabalho e não andaria a saltar de página em página sempre que a tarefa fosse realizada de forma incorreta.

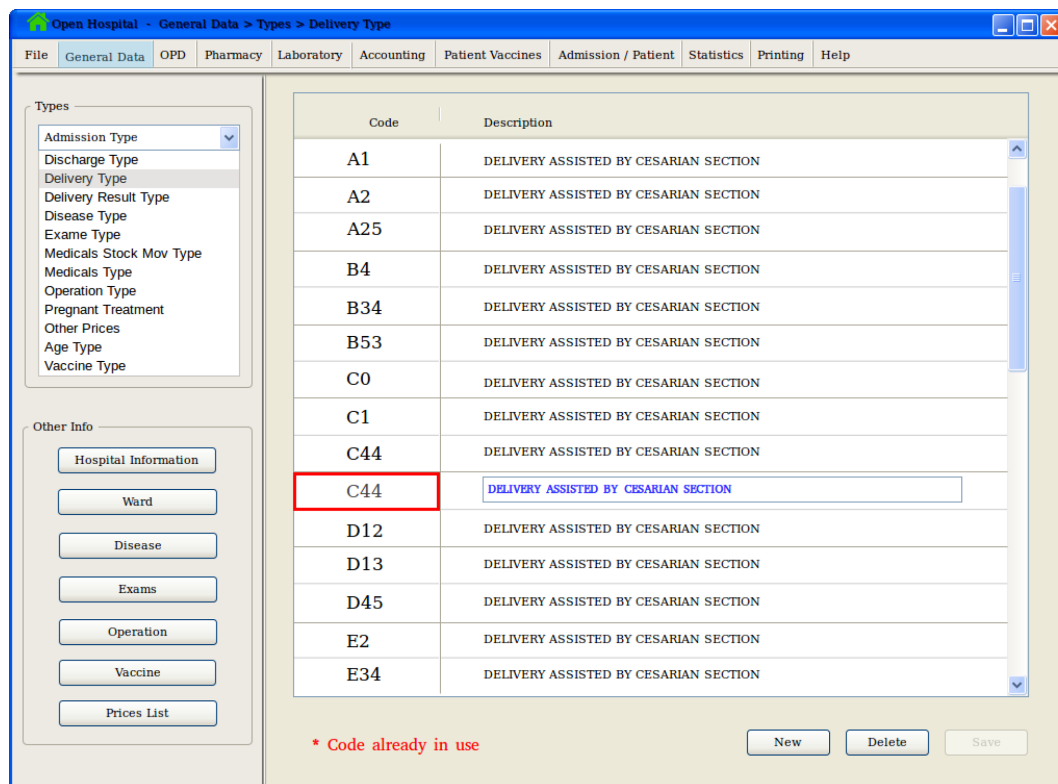


Fig. 17.: Refactoring apresentado para o Middle Man.

Por fim para testar o *Refactoring* apresentado foram feitas duas questões sobre o mesmo às quais foi pedido para avaliar e justificar (grupo 3):

A Considera que apresentar a mensagem, quer seja ela informativa ou de erro, na própria página é a solução mais correta?

B O que acha da alteração feita relativamente à aplicação original?

4.5.2 *Tarefa 2: Too Many Layers*

Nesta segunda tarefa foi pedido aos participantes para avaliarem o total de passos a realizar para inserir um novo tipo de dados numa tabela. Para tal os participantes, na aplicação, tinham de selecionar as seguintes opções:

General Data >>Types >>Exam Type >>

tal e qual como apresentado na Figura 18.

(a) Menu Geral (b) General Data Menu (c) Types Menu

(d) Exam Type Menu e exemplo de Too Many Layers

Fig. 18.: Imagens da Tarefa 2 representando um exemplo do Too Many Layers

No primeiro grupo da Tarefa 2 foram apresentadas três questões para o participante avaliar relativamente ao caso de estudo da aplicação *OH*:

1. Concorda que o total de páginas que tem de percorrer para realizar a tarefa é o mais correto?
2. Concorda que todos estes passos poderiam ser poupados se estas opções estivessem agrupadas num menu?
3. Considera este exemplo uma anomalia na conceção da interface?

Ao fim deste grupo de questões foi pedido a cada participante que indicasse um conjunto de alterações visuais que efetuariam para corrigir esta anomalia. O objetivo desta pergunta é obter um conjunto de *Refactorings* provenientes do utilizador (grupo 2).

Após questionar os participantes acerca da aplicação foi-lhes apresentado um exemplo de *Refactoring* para o *Too Many Layers*, ver a Figura 19. O *Refactoring* apresentado consiste em transformar todas as páginas num sistema de menus e sub-menus laterais e superiores como se pode visualizar nas figuras. Optou-se também por utilizar *dropdowns* para unir páginas dentro do mesmo domínio como no caso das que estão no menu *Types*, ver Figura 19. Para validar a proposta de *Refactoring* foi questionado aos participantes qual a opinião que tinham quanto a agrupar páginas em menus/tabs.

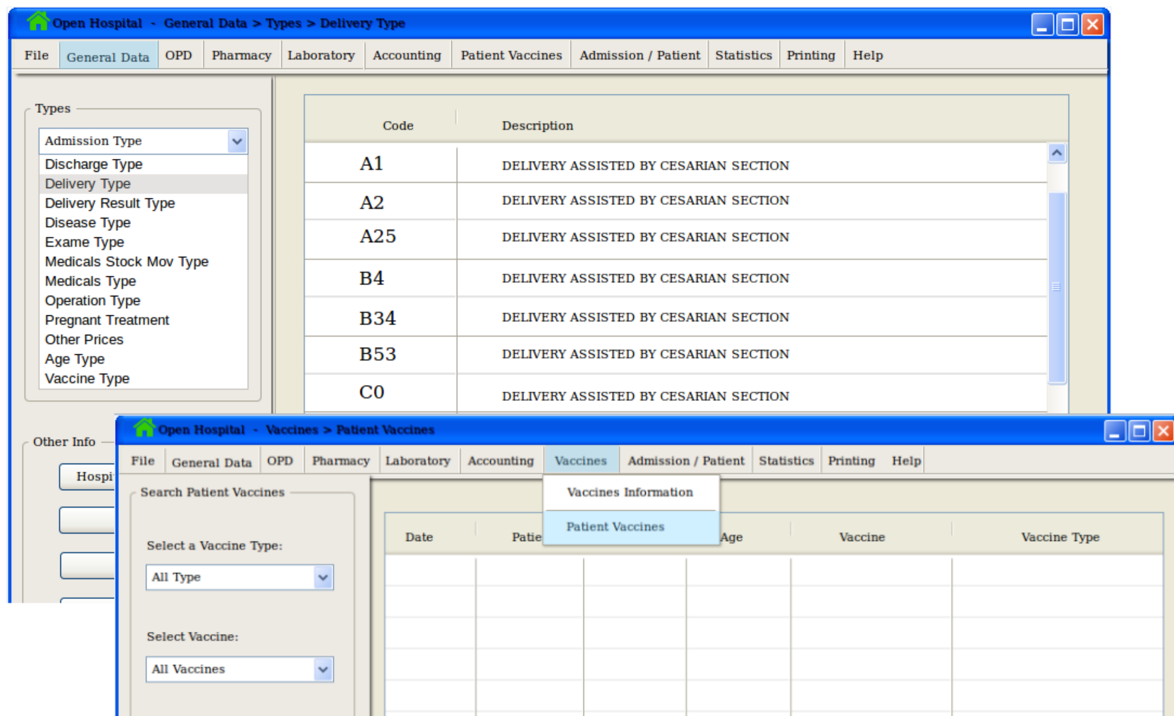


Fig. 19.: Refactoring apresentado para o Too Many Layers.

Por último questionou-se os participantes sobre a proposta de *Refactoring* apresentada (grupo 3):

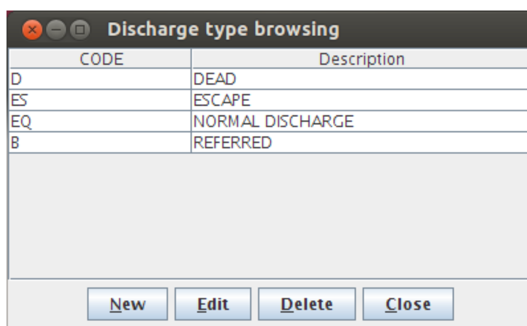
A Qual a sua opinião quanto a agrupar estas páginas em menus/tabs?

4.5.3 *Tarefa 3: Shotgun Surgery*

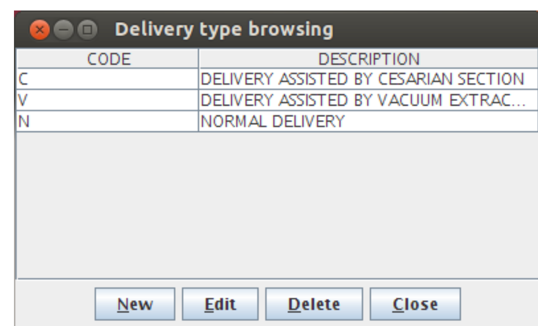
Na terceira tarefa foi pedido aos participantes para visualizarem três páginas na aplicação *OH* e prestarem atenção ao "path" para cada uma delas. Para tal os participantes tinham de na aplicação *OH* realizar as seguintes tarefas:

General Data >>Types >>Discharge Type >>Close
General Data >>Types >>Delivery Type >>Close
General Data >>Types >>Exam Type >>Close

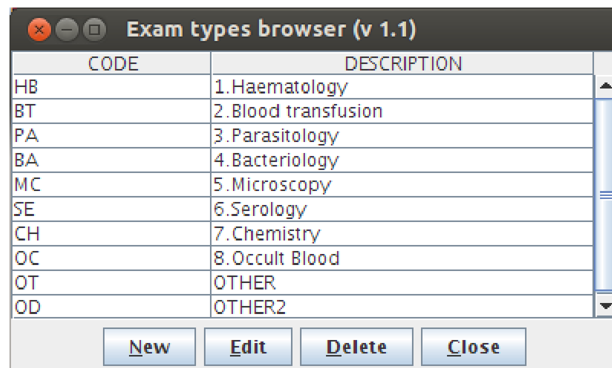
As páginas finais de cada uma das tarefas estão representadas na Figura 20.



(a) Menu Discharge Type



(b) Menu Delivery Type



(c) Menu Exam Type

Fig. 20.: Imagens da Tarefa 3 representando um exemplo do Shotgun Surgery

No primeiro grupo da Tarefa 3 foram apresentadas quatro questões para os participantes avaliarem o aparecimento do *Shotgun Surgery* no caso de estudo da aplicação *OH*:

1. Concorda que a disposição das páginas e a maneira como tem de procurar informação dentro do *Types* prejudica a produtividade e eficiência das tarefas?

2. As páginas são todas semelhantes, concorda que se devam juntar numa só?

3. Provavelmente o mesmo código foi usado para programar uma parte comum das páginas, se o tivesse de repetir em 5, 10 ou mais lugares lembrar-se-ia de todos eles?

4. Considera este exemplo uma anomalia na conceção da interface?

Ao fim deste grupo de questões foi pedido a cada participante que indicasse um conjunto de alterações visuais que efetuariam para corrigir esta anomalia. O objetivo desta pergunta é obter um conjunto de *Refactorings* provenientes do utilizador (grupo 2).

Após questionar os participantes acerca da aplicação foi-lhes apresentado um exemplo de *Refactoring* para o *Shotgun Surgery* apresentado na Figura 21. Nesta sugestão de *Refactoring* agrupou-se todas as opções do menu "Types" um componente estilo *dropdown* lateral a uma página comum. Quando selecionada uma opção nesse menu a página do lado direito atualizava com a informação da tarefa. Neste *Refactoring* optou-se também por usar hiperligações para outras opções, como podemos ver simplificou-se a estrutura do menu *General Data*, com esta solução tanto é possível ver o menu "Types" como aceder diretamente a informações como por exemplo: "Hospital Information", "Exames", "Operations", etc.

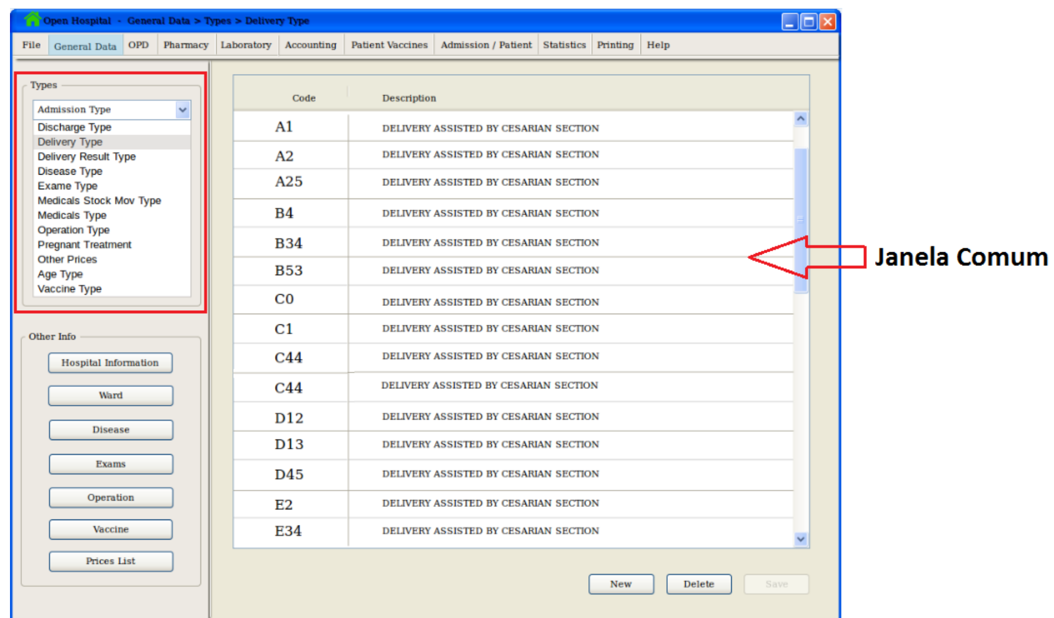


Fig. 21.: Refactoring apresentado para o Shotgun Surgery.

Por fim para testar o *Refactoring* apresentado foi feita uma questão sobre o mesmo à qual foi pedido para avaliar e justificar (grupo 3):

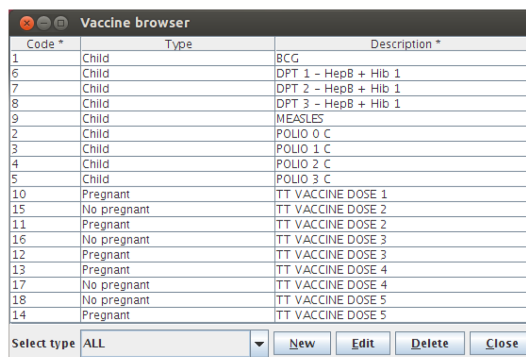
A Qual a sua opinião sobre agrupar páginas idênticas numa única página?

4.5.4 *Tarefa 4: Inappropriate Intimacy*

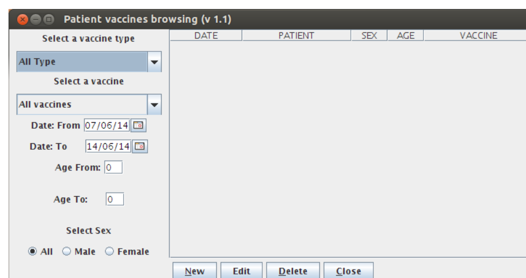
Na quarta tarefa foi pedido aos participantes para visualizarem duas páginas na aplicação *OH* e prestarem atenção à disposição e aos elementos de cada uma delas. Para tal os participantes tinham de na aplicação *OH* realizar as seguintes tarefas:

General Data >>Vaccine >>Close
Patient Vaccines >>Close

As páginas finais de cada uma das tarefas estão representadas na Figura 22.



(a) Menu Vaccine



(b) Patient Vaccines

Fig. 22.: Imagens da Tarefa 4 representando um exemplo do Inappropriate Intimacy

No primeiro grupo da Tarefa 4 foram apresentadas quatro questões para os participantes avaliarem o aparecimento do *Inappropriate Intimacy* no caso de estudo da aplicação *OH*:

1. Concorda que estas duas páginas pertencem ao mesmo domínio?
2. Concorda que quando duas páginas pertencem ao mesmo domínio, o caminho para elas deve ser idêntico?
3. Agrupando estas duas páginas numa opção *Vaccines* no menu principal aumentaria a produtividade e legibilidade da aplicação?

4. Considera este exemplo uma anomalia grave?

Ao fim deste grupo de questões foi pedido a cada participante que indicasse um conjunto de alterações visuais que efetuariam para corrigir esta anomalia (grupo 2).

Após questionar os participantes acerca da aplicação foi-lhes apresentado um exemplo de *Refactoring* para o *Inappropriate Intimacy* apresentado na Figura 23. Neste exemplo criou-se um menu superior chamado "Vaccines" com ligação às duas páginas. Neste menu superior o utilizador pode aceder tanto às informações das vacinas através do sub-menu "Vaccines Information" onde pode gerir todas as informações de vacinas, criar novas ou eliminar, como também pode aceder à gestão das vacinas dos pacientes através do sub-menu "Patient Vaccines" onde pode realizar pesquisas, editar informações e eliminar vacinas administradas a pacientes e administrar novas vacinas. Usando esta solução os utilizadores finais não precisariam de andar a saltar de página em página, e de menu em menu, para ver ou gerir todos os dados referentes às vacinas.

Open Hospital - Vaccines > Patient Vaccines

File | General Data | OPD | Pharmacy | Laboratory | Accounting | Vaccines | Admission / Patient | Statistics | Printing | Help

Vaccines Information
Patient Vaccines

Search Patient Vaccines
Select a Vaccine Type: All Type
Select Vaccine: All Vaccines
Date:
From:
To:
Age From: To:
Select Sex: All
Search
Count: 0

Date	Patie		Age	Vaccine	Vaccine Type

New | Edit | Delete | Save

Fig. 23.: Refactoring apresentado para o Inappropriate Intimacy.

Por último foi pedido aos participantes para avaliar e justificar e justificar o *Refactoring* apresentado (grupo 3):

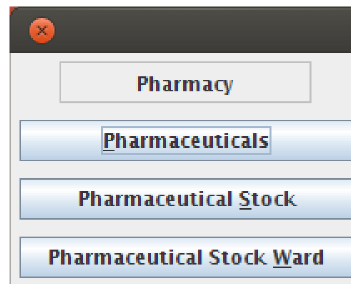
A Qual a sua opinião sobre agrupar páginas dentro do mesmo domínio?

4.5.5 *Tarefa 5: Information Overload*

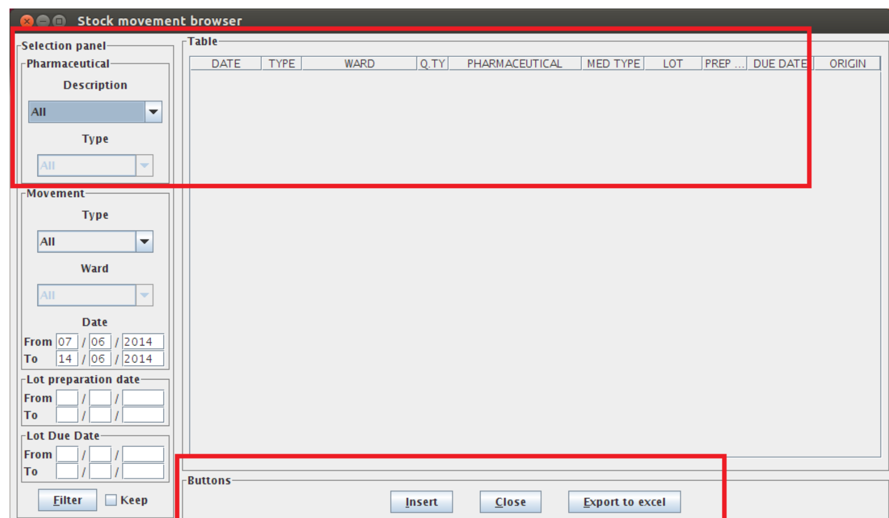
Na quinta e última tarefa foi pedido aos participantes para acederem a uma página na aplicação *OH*. Para tal os participantes tinham de na aplicação *OH* selecionar as seguintes opções:

Pharmacy >>Pharmaceuticals Stock >>Close

e prestar atenção aos detalhes da última página, ver Figura 24. Os detalhes encontra-se marcados dentro das caixas a vermelho. Dentro dessas caixas os participantes tinham como objetivo identificar informação extra desnecessária para realizar as tarefas.



(a) Pharmacy Menu e exemplo de Information Overload



(b) Pharmaceuticals Stock Menu e exemplo de Information Overload

Fig. 24.: Imagens da Tarefa 5 representando um exemplo do Information Overload

No primeiro grupo da Tarefa 5 foram apresentadas quatro questões para o participante avaliar relativamente ao caso de estudo da aplicação *OH*:

1. O excesso de informação visual nas aplicações prejudica a legibilidade e produtividade dos utilizadores?

2. Considera que todas as operações possíveis nessa página são facilmente identificáveis?

3. O excesso de alguns componentes visuais nessa página, como por exemplo a informação de “Tabela” ou de “Jpanel” ajuda a entender melhor como é composto o layout?

4. Considera este exemplo uma anomalia grave?

Ao fim deste grupo de questões foi pedido a cada participante que indicasse que alterações visuais efetuariam para corrigir esta anomalia. O objetivo desta pergunta é obter um conjunto de *Refactorings* provenientes do utilizador (grupo 2).

Após questionar os participantes acerca da aplicação foi-lhes apresentado um exemplo de *Refactoring* para o *Information Overload* apresentado na Figura 25. O *Refactoring* apresentado consiste em remover toda a informação excessiva que dificultava a compreensão das tarefas, reorganizar melhor os componentes e alterar alguns componentes por outros mais apropriados para cada tarefa.

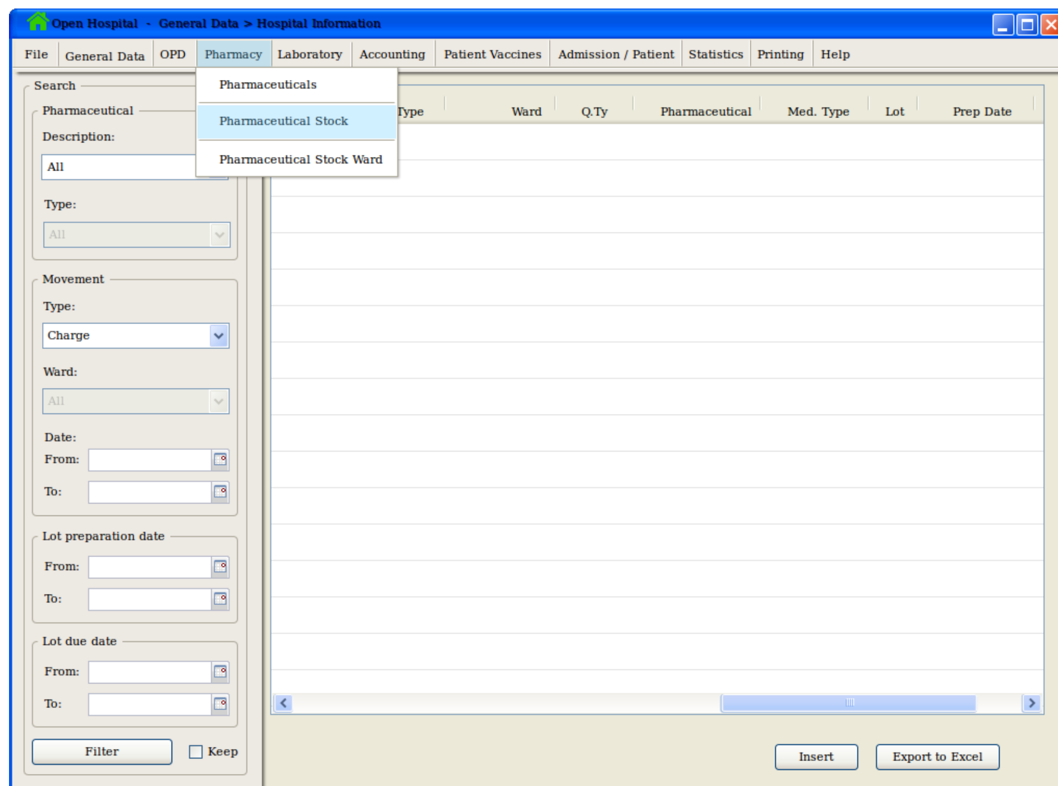


Fig. 25.: Refactoring apresentado para o Information Overload.

Por fim para testar o *Refactoring* apresentado foram feitas duas questões sobre o mesmo às quais foi pedido para avaliar e justificar (grupo 3):

A Qual a sua opinião sobre estas alterações?

B Considera que as alterações melhoraram a legibilidade e produtividade da aplicação?

5

RESULTADOS DA AVALIAÇÃO DOS USABILITY SMELLS

Neste capítulo será apresentado os resultados obtidos ao teste de usabilidade. Recordando, o teste foi realizado com a participação de 12 alunos da Universidade do Minho. Todos os participantes são da área de informática com algum conhecimento em diversas paradigmas de programação e pelo menos mais de 2 anos de experiência nesta área.

A estrutura desde capítulo é composta por uma secção onde é apresentada, separadamente, a avaliação de cada um dos cinco *Usability Smells* do teste de usabilidade, nomeadamente: *Middle Man*, *Too Many Layers*, *Shotgun Surgery*, *Inappropriate Intimacy* e *Information Overload*.

5.1 RESULTADOS OBTIDOS

Nesta secção serão apresentados os resultados obtidos no teste de usabilidade. Para cada uma das perguntas ou conjunto de perguntas pertencente a uma tarefa relativa a um *Usability Smell* serão apresentados e discutidos os resultados com a ajuda de gráficos e tabelas resumo.

5.1.1 *Resultados Tarefa 1: Middle Man*

Os resultados obtidos do grupo 1 de perguntas encontram-se na Figura 26 em forma de gráfico de barras numa escala de *Likert* e na tabela 1 a qual mostra as frequências relativas de cada uma das perguntas.

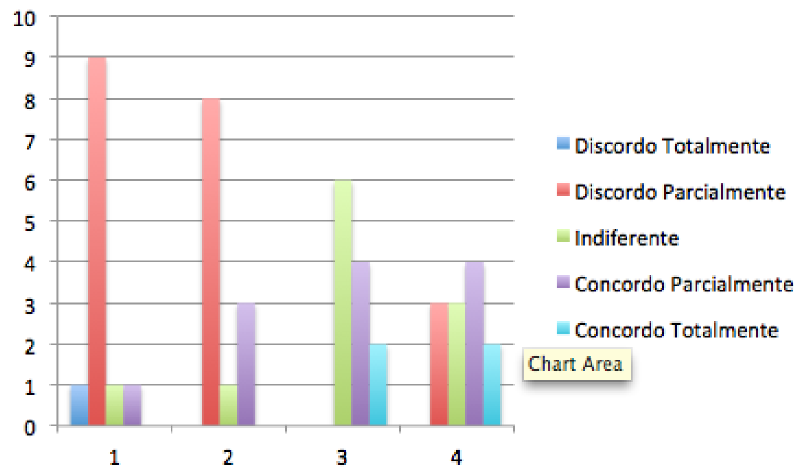


Fig. 26.: Resultados obtidos às perguntas do grupo 1 sobre o Middle Man.

	DT	DP	I	CP	CT
Pergunta 1	8.0%	75.0%	8.0%	8.0%	0.0%
Pergunta 2	0.0%	67.0%	8.0%	25.0%	0.0%
Pergunta 3	0.0%	0.0%	50.0%	33.0%	17.0%
Pergunta 4	0.0%	25.0%	25.0%	33.0%	17.0%

Tab. 1.: Frequências relativas Middle Man - Grupo 1

Pergunta 1 - O aparecimento de páginas pop-up é a melhor forma de fornecer indicações ao utilizador?

Nesta primeira pergunta foi questionado aos participantes se o aparecimento de páginas pop-up é a melhor forma de fornecer indicações ao utilizador. Pode-se verificar que para 8% dos participantes é indiferente se as aplicações usam páginas pop-up ou outro tipo de componentes para apresentar indicações aos utilizadores. Como esperado, a resposta mais votada (ou o valor da moda) foi o discordo parcialmente onde 75% dos participantes, discordam que seja a melhor solução contudo ainda assim os restantes 8% concordam com a sua utilização mas de forma reticente.

Pergunta 2 - Concorda que o aparecimento de uma página só para indicar que nenhum objeto se encontra selecionado o ajuda a realizar a tarefa de forma mais efetiva?

Na pergunta dois foi questionado aos participantes se concordam que o aparecimento de uma página só para indicar que nenhum objeto se encontra selecionado os ajuda a realizar a tarefa de forma mais efetiva. Nesta segunda pergunta os participantes olharam para a aplicação e começaram a perceber que provavelmente essa solução não seria a mais atrativa. Como resposta mais frequente, 67% dos

participantes escolheram discordar parcialmente quando questionados se utilizar este tipo de estrutura de páginas é o mais vantajoso para realizar as tarefas. Para 8% dos participantes a forma como apresenta a tarefa é indiferente. Por outro lado 25% dos participantes mesmo olhando para a aplicação concordam parcialmente que as páginas pop-up os ajuda a realizar a tarefa de forma mais efetiva. Segundo alguns ajuda a perceber que falharam algum passo na edição da tabela mas logo em seguida afirmam que não deveria ser necessário esse tipo de comportamento da aplicação se os mecanismos de edição fossem diferentes.

Pergunta 3 - Preferia que a informação da página fosse apresentada na mesma página onde se encontra a tabela?

Quando os participantes foram questionados se preferiam que a informação fosse apresentada na mesma página onde se encontra a tabela os resultados dividiram-se. Como resposta mais frequente, para 50% dos participantes é indiferente que essa informação seja transferida para o local onde a tarefa é realizada ou esteja numa página separada. Em contrapartida os restantes 50% dos participantes, divididos entre concordo parcialmente e concordo totalmente, preferem que essa informação seja transferida da página pop-up para a página onde se encontra a tabela, a página da tarefa.

Pergunta 4 - Considera este exemplo uma anomalia na conceção da interface?

Por último foi questionado aos participantes se acham o exemplo apresentado uma anomalia na conceção da interface. Metade dos participantes (50%) concorda que o *Middle Man* é uma anomalia que afeta a interface e prejudica a realização de tarefas a que está associado. Dos 50% que aceita, 33% concorda parcialmente sendo essa a resposta com maior frequência. Dos restantes 50%, 25% discordam parcialmente ficando a pensar se realmente é ou não. Destes 25% alguns associam esta pergunta apenas ao exemplo apresentado e não a um caso geral. Por último, os restantes 25% dos participantes acham que esta anomalia é indiferente e não vêm grande problema nela.

Ao fim deste grupo de questões foi pedido a cada participante que indicasse que alterações visuais efetuariam para corrigir esta anomalia (grupo 2). Segundo os participantes as alterações necessárias para eliminar o *Middle Man* passariam por transferir toda a informação para a página da tarefa, apresentar o erro onde se encontra a tabela de forma a evitar mais um "click" utilizando para isso *labels* e a mensagem deveria estar com letras a vermelho. Outra solução passaria por permitir a edição de conteúdo na tabela ou apenas ativar os botões quando se seleciona algum dado da mesma. Dois participantes sugerem a utilização de um pop-up temporário de forma a que o utilizador fosse alertado sem recorrer ao "click" para fechar o aviso ou, um balão de texto com a mensagem informativa a aparecer quando posiciona o rato sobre o botão de editar. Contudo também houve quem considerasse que os pop-ups são desagradáveis contudo não considera que o exemplo da tarefa 1 seja um problema e que o pop-up é uma boa forma de fornecer informações sobre funcionalidades principais.

Por fim foi apresentado aos participantes o *Refactoring* proposto para corrigir o *Middle Man* ao qual se fez duas questões (grupo 3).

Os resultados das questões encontram-se na Figura 27, na tabela 2 encontra-se o resultado das respetivas frequências relativas:

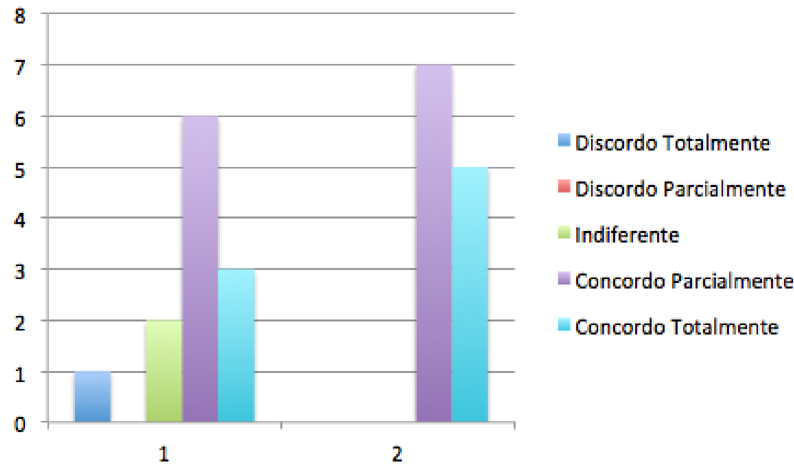


Fig. 27.: Resultados obtidos às perguntas do grupo 3 sobre o Middle Man.

	DT	DP	I	CP	CT
Pergunta A	8.0%	0.0%	17.0%	50.0%	25.0%
Pergunta B	0.0%	0.0%	0.0%	58.0%	42.0%

Tab. 2.: Frequências relativas Middle Man - Grupo 3

Pergunta A - Considera que apresentar a mensagem, quer seja ela informativa ou de erro, na própria página é a solução mais correta?

Nesta primeira pergunta foi questionado aos participantes se consideram que apresentar a mensagem, quer seja ela informativa ou de erro, na própria página é a solução mais correta tendo em conta o *Refactoring* apresentado. A esta pergunta a maioria dos participantes, 75%, concorda que seja a maneira mais correta de o fazer, sendo que a resposta mais frequente com 50% das respostas é os participantes concordarem parcialmente com a sugestão apresentada. Dos restantes 25%, 17% ficam indiferentes a esta alteração e 8% discordam parcialmente que essa alteração seja feita.

Pergunta B - O que acha da alteração feita relativamente à aplicação original?

Quando questionados sobre o que achavam relativamente à alteração da aplicação principal 100% dos participantes é a favor dela justificando que as alterações são significativamente notórias, a solução torna-se mais intuitiva, bem estruturada e organizada. 58% dos participantes concordou parcialmente

que o *Refactoring* apresentado está melhor que o original, sendo esta a resposta com maior frequência. A alteração oferece ao utilizador final uma melhoria da experiência em relação à aplicação original e que o não aparecimento das páginas pop-up poupa um "click" na realização da tarefa melhorando o tempo de reação na deteção do erro.

5.1.2 *Resultados Tarefa 2: Too Many Layers*

Os resultados obtidos das perguntas do grupo 1 encontram-se presente na Figura 28 em forma de gráfico de barras numa escala de *Likert* e na tabela 3 a qual mostra as frequências relativas de cada uma das perguntas.

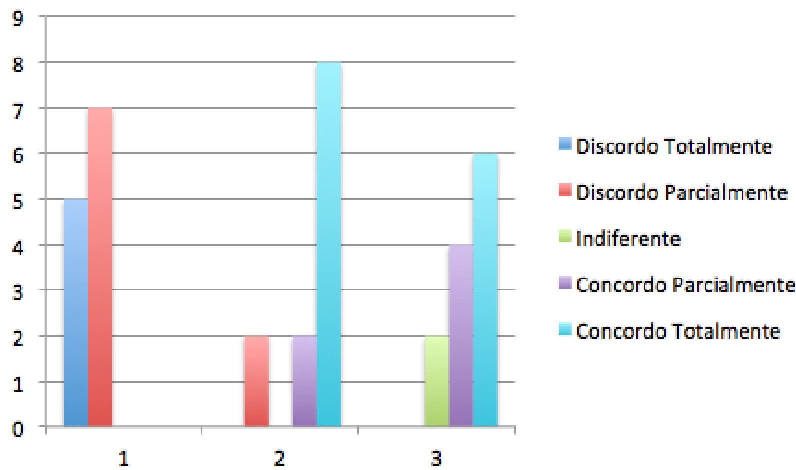


Fig. 28.: Resultados obtidos às perguntas do grupo 1 sobre o Too Many Layers.

	DT	DP	I	CP	CT
Pergunta 1	42.0%	58.0%	0.0%	0.0%	0.0%
Pergunta 2	0.0%	17.0%	0.0%	17.0%	67.0%
Pergunta 3	0.0%	0.0%	17.0%	33.0%	50.0%

Tab. 3.: Frequências relativas Too Many Layers - Grupo 1

Pergunta 1 - Concorda que o total de páginas que tem de percorrer para realizar a tarefa é o mais correto?

A primeira questão da segunda tarefa, com o objetivo de avaliar o *Too Many Layers*, foi perceber se os participantes concordavam com o total de páginas que tinham de percorrer para realizar a tarefa. Esta primeira questão obteve os 100% de participantes a discordar, 42% discordaram na totalidade e os restantes 58% de forma parcial sendo esta a resposta mais frequente. Um dos primeiros pontos

captados nesta tarefa apareceu quando os participantes viram a aplicação. A maioria questionou a estrutura da aplicação e se esta não poderia ser desenhada de outra forma.

Pergunta 2 - Concorda que todos estes passos poderiam ser poupados se estas opções estivessem agrupadas num menu?

Na segunda pergunta foi questionado aos participantes se concordavam em agrupar as opções em menus de forma a evitar o excesso de passos até a realização da tarefa. Com uma maior frequência de adesão, 67% dos participantes concordou totalmente com o uso de menus para facilitar a tarefa, juntaram-se ainda 17% que concordaram de forma parcial. Os restantes 17% discordou de forma parcial. Estes participantes duvidaram, de certa forma, que a utilização de menus ajuda a realizar a tarefa de forma mais eficaz, reduzindo o número de "clicks" na aplicação.

Pergunta 3 - Considera este exemplo uma anomalia na conceção da interface?

Por último e não menos importante questionou-se os participantes se consideravam o exemplo uma anomalia na conceção da interface. A maioria dos participantes considerou este *smell* uma anomalia na interface. 50% dos participantes concordou na totalidade com essa questão, sendo a resposta com maior frequência. 33% dos participantes concordou de forma parcial e os restantes 17% dos participantes ficaram indiferentes se este exemplo seria uma anomalia ou não. Notou-se que, para os participantes que ficaram indiferentes a esta questão nem a estrutura da aplicação era grave nem a anomalia preocupante, segundo alguns o tempo que demoravam até chegar à tarefa era demorado, contudo não era "difícil" de realizar.

No final deste grupo de questões foi pedido aos participantes para indicarem as alterações visuais que efetuariam para resolver esta anomalia (grupo 2). No conjunto das respostas a utilização de menus seria a solução ideal. Houve quem dissesse que deveriam adicionar uma opção num menu para adicionar novos "types" ou colocar as operações numa única página. Com esta alteração alguns participantes afirmaram que um menu superior pouparia "clicks" em excesso. Houve também quem sugerisse usar menus de separadores, tabs ou até mesmo uma versão usando *MDI Applications Layout*[1] com recurso a menus.

Por fim foi apresentado aos participantes o *Refactoring* proposto para corrigir o *Too Many Layers* ao qual se fez uma questão (grupo 3), ver Figura 19.

Os resultados da questão encontram-se na Figura 29, na tabela 4 encontra-se o respetivo resultado das frequências relativas:

1 Multiple-Document Interface

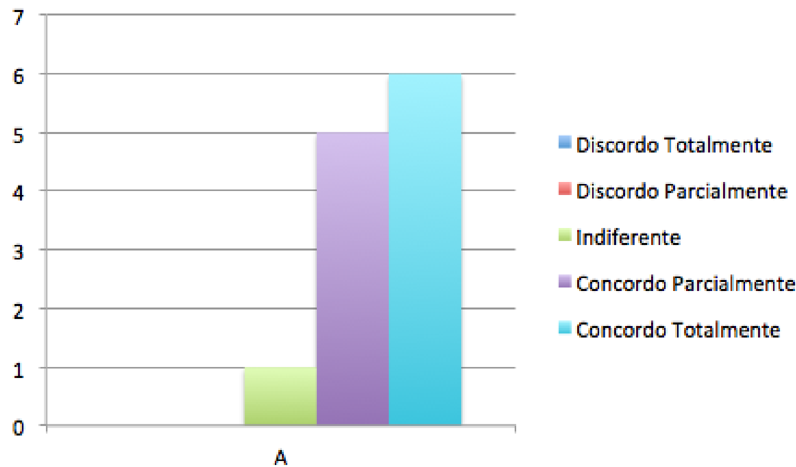


Fig. 29.: Resultados da pergunta do terceiro grupo sobre o Too Many Layers.

	DT	DP	I	CP	CT
Pergunta A	0.0%	0.0%	8.0%	42.0%	50.0%

Tab. 4.: Frequências relativas Too Many Layers - Grupo 3

Pergunta A - Qual a sua opinião quanto a agrupar estas páginas em menus/tabs?

Como podemos ver nos resultados da tabela 4 a maioria dos participantes concordou com a proposta de *Refactoring*. Os 98% dos participantes, que concordaram com o *Refactoring* apresentado para o *Too Many Layers*, preferem agrupar páginas em menu/tabs para evitar a sua proliferação o que, segundo eles, facilita a usabilidade da aplicação e soluciona os problemas do excesso de páginas. Sendo a reposta mais frequente (50%) concordarem totalmente com o agrupar de páginas em menus e tabs. Além disso apontam que as páginas em tabs/menus permitem uma melhor organização da aplicação tornando-a mais visível, limpa e intuitiva, acrescentam que a navegação fica rápida na medida que não é necessário estar a retroceder na aplicação para navegar entre várias funcionalidades. Os restantes 8% dos participantes demonstraram-se indiferentes a esta alteração indicando que o uso intensivo de menus/tabs possivelmente também se pode tornar numa desvantagem, se por exemplo houver um caso em que o utilizador necessite de ver várias tabs ao mesmo tempo, das duas uma, ou se adota a possibilidade de transformar tabs em páginas para se poderem posicionar lado a lado ou será impossível para o utilizador realizar a tarefa.

5.1.3 *Resultados Tarefa 3: Shotgun Surgery*

Os resultados obtidos das perguntas do grupo 1 encontram-se presente na Figura 30 em forma de gráfico de barras numa escala de *Likert* e na tabela 5 a qual mostra as frequências relativas de cada uma das perguntas.

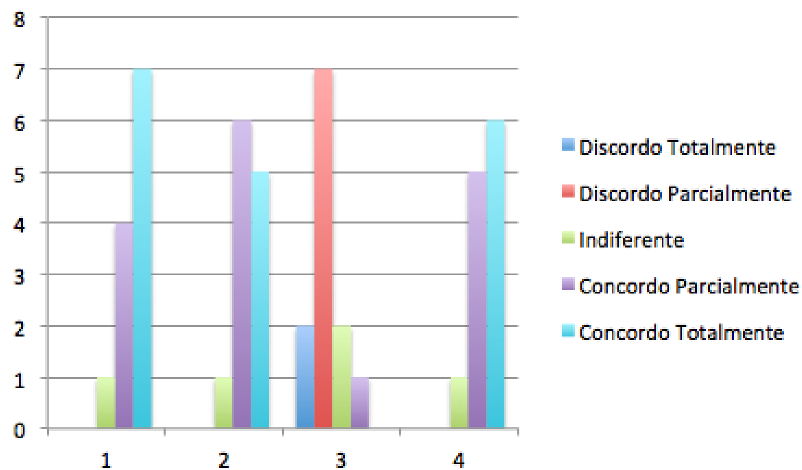


Fig. 30.: Resultados obtidos às perguntas do grupo 1 sobre o Shotgun Surgery.

	DT	DP	I	CP	CT
Pergunta 1	0.0%	0.0%	8.0%	33.0%	58.0%
Pergunta 2	0.0%	0.0%	8.0%	50.0%	42.0%
Pergunta 3	17.0%	58.0%	17.0%	8.0%	0.0%
Pergunta 4	0.0%	0.0%	8.0%	42.0%	50.0%

Tab. 5.: Frequências relativas Shotgun Surgery - Grupo 1

Pergunta 1 - Concorda que a disposição das páginas e a maneira como tem de procurar informação dentro do Types prejudica a produtividade e eficiência das tarefas?

Na primeira pergunta foi questionado aos participantes se concordariam com a disposição das páginas e se a forma como têm de procurar informação dentro do menu "Types" prejudicava a produtividade e eficiência das tarefas. A maioria dos participantes concorda totalmente que a forma como está estruturada a aplicação e como têm de fazer a procura cria uma grande inconveniente na produtividade e eficiência das tarefas. Como resposta mais frequente, 58% dos participantes concorda totalmente que a produtividade é afetada. Outros 33% concorda de forma parcial, provavelmente necessitariam de mais casos para poder concordar na totalidade. Satisfatoriamente apenas 8% dos participantes achou este problema indiferente.

Pergunta 2 - As páginas são todas semelhantes, concorda que se devam juntar numa só?

Na segunda questão perguntamos aos participantes se como as páginas são todas iguais se se poderiam juntar numa só. A maioria dos participantes concordou de forma parcial. 42% dos participantes concordou totalmente que estas páginas poderiam ser agrupadas alguns afirmando que iria simplificar bastante no caso das pesquisas visto que não seria preciso andar a saltar de página em página. 50% concordou de forma parcial sendo considerada a resposta com maior frequência, como na pergunta anterior esta percentagem de participantes precisava de ver mais exemplos. Contudo os 50% dos participantes que concordaram parcialmente não excluíram que a estrutura da aplicação fosse a mais correta. Por último para 8% dos participantes agrupar páginas semelhantes é indiferente.

Pergunta 3 - Provavelmente o mesmo código foi usado para programar uma parte comum das páginas, se o tivesse de repetir em 5, 10 ou mais lugares lembrar-se-ia de todos eles?

Ao ser analisada a aplicação, mais propriamente estes exemplos, notou-se que a estrutura interna das páginas era idêntica. Para estes casos subentendeu-se que estaríamos perante um exemplo onde existiria reaproveitamento de código para certas partes comuns das páginas. Questionámos assim os participantes, caso fossem eles a programar a aplicação e caso tivessem de repetir o mesmo código em 5, 10 ou mais lugares se se lembrar-se-iam de todos os sítios onde o replicavam. A maioria dos participantes discordou de forma parcial. Os 58% dos participantes que discordaram de forma parcial sendo esta a resposta mais frequente. Os participantes que discordaram parcialmente afirmaram que é muito difícil quando se fala em reaproveitamento de código decorar todos os sítios onde isso acontece; além destes participantes outros 17% discordaram totalmente, alguns afirmando que seria uma "missão impossível" a não ser que tivessem um bom mapeamento de toda a aplicação. Para outros 17% dos participantes essa questão é indiferente, segundo alguns essa questão não seria um problema grave porque de alguma forma conseguiriam lembrar-se de todos os locais. Os últimos 8% concordaram de forma parcial que não teria problemas em lembrar-se de todos os locais onde foi reaproveitado o código.

Pergunta 4 - Considera este exemplo uma anomalia na conceção da interface?

Por último foi questionado aos participantes se consideravam este exemplo uma anomalia da interface. A maioria dos participantes concordou totalmente que este *smell* é uma verdadeira anomalia na interface. A resposta mais frequente foi a concordo totalmente com 50% dos participantes a indicar que o *Shotgun Surgery* é realmente uma anomalia. Estes participantes frisaram que quando o *smell* ocorre a estrutura da aplicação fica bastante confusa e que isso prejudica bastante a produtividade das tarefas. Outros 42% dos participantes concordaram mas de forma parcial e para os restantes 8% o aparecimento do *Shotgun Surgery* é um problema que lhes é indiferente. Para estes últimos participantes, este *smell* é indiferente porque segundo a pergunta 3 eles conseguiriam relembrar todos os locais onde

o código era replicado.

Ao fim deste grupo de questões foi pedido aos participantes que indicassem que alteração ou alterações visual(is) efetuariam para corrigir esta anomalia. As opiniões a esta questão foram quase todas à volta da criação de menus e junção de páginas comuns. Para alguns a melhor solução seria a criação de páginas com atalhos rápidos e intensivos ao estilo de um menu ou criar um menu ou lista onde se colocariam todos os "Types" ou até mesmo usar *dropdowns*. Segundo os participantes qualquer um destes exemplos seria implementado com a seguinte estrutura e lógica: colocar um menu com os "Types" do lado esquerdo e uma página comum do lado direito, ao selecionar um "type" a página do lado direito atualizaria consoante o pretendido.

Por último foi apresentado aos participantes o *Refactoring* proposto para corrigir o *Too Many Layers* ao qual se fez uma questão (grupo 3), ver Figura 21.

Os resultados da questão encontram-se na Figura 31, na tabela 6 encontra-se o resultado das frequências relativas:

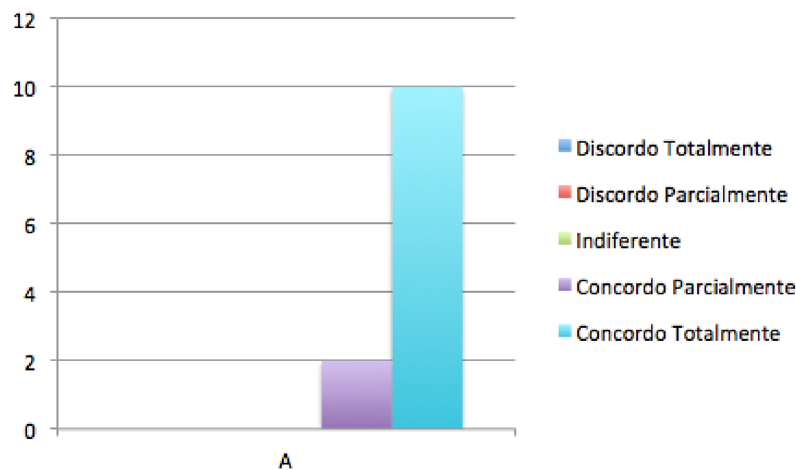


Fig. 31.: Resultados da pergunta do terceiro grupo sobre o Shotgun Surgery.

	DT	DP	I	CP	CT
Pergunta A	0.0%	0.0%	0.0%	17.0%	83.0%

Tab. 6.: Frequências relativas Shotgun Surgery - Grupo 3

Pergunta A - Qual a sua opinião sobre agrupar páginas idênticas numa única página?

Nesta pergunta foi questionado aos participantes, relativamente ao *Refactoring*, qual a opinião que tinham sobre agrupar páginas idênticas numa única página. Como podemos visualizar na tabela 6 a

maioria dos participantes concordou totalmente em agrupar as páginas. 83% dos participantes não hesitou quando questionados se essa era a melhor opção, sendo a resposta ̈concordo totalmenteä resposta com maior frequência. Os restantes 17% dos participantes também concordaram de forma parcial com essa alteração. Segundo as justificações dos participantes é bastante mais produtivo e mais simples não ter de fechar e voltar a navegar para aceder a outra página, a navegação torna-se mais fluida, simples e direta, desde que esteja bem implementada, ficando menos confusa para o utilizador uma vez que tinha ao seu dispor todas as informações necessárias. Os participantes apontam também que ao apresentar os vários tipos numa lista evita-se que a página da aplicação seja demasiado grande e aproveita-se a zona visível da mesma para mostrar tabelas evitando mais uma vez cliques desnecessários, principalmente para mudar de "Types". Por último apontam que se evita trabalho ao programador e ao utilizador então é mais benéfico.

5.1.4 *Resultados Tarefa 4: Inappropriate Intimacy*

Os resultados obtidos para as questões do grupo 1 encontra-se presente na figura 32 em forma de gráfico de barras numa escala de *Likert* e na tabela 7 a qual mostra as frequências relativas de cada uma das perguntas.

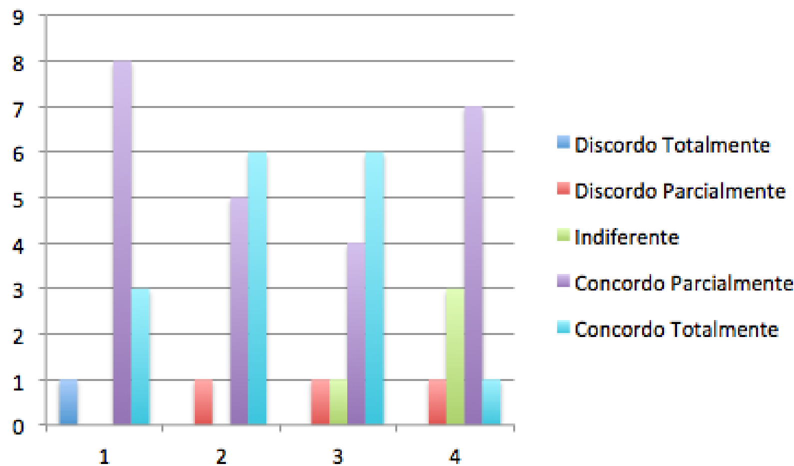


Fig. 32.: Resultados obtidos às perguntas do grupo 1 sobre o Inappropriate Intimacy.

	DT	DP	I	CP	CT
Pergunta 1	8.0%	0.0%	0.0%	67.0%	25.0%
Pergunta 2	0.0%	8.0%	0.0%	42.0%	50.0%
Pergunta 3	0.0%	8.0%	8.0%	33.0%	50.0%
Pergunta 4	0.0%	8.0%	25.0%	58.0%	8.0%

Tab. 7.: Frequências relativas Inappropriate Intimacy - Grupo 1

Pergunta 1 - Concorda que estas duas páginas pertencem ao mesmo domínio?

Na primeira pergunta foi questionado aos participantes se concordam que as duas páginas apresentadas pertencem ao mesmo domínio. A maioria dos participantes concordou de forma parcial com esta questão sendo a resposta com maior frequência. 67% dos participantes olharam para as duas páginas analisando as tarefas e o conteúdo. A análise fez com que os participantes concordassem de forma parcial que as duas páginas pertenciam ao mesmo domínio. 25% dos participantes concordou totalmente com esta questão. Houve contudo 8% dos participantes que discordaram totalmente com esta questão do domínio das páginas Estes participantes justificaram que as duas páginas não pertenciam ao mesmo domínio através da posição das duas na aplicação, segundo eles como estavam em locais diferentes já não poderiam ser do mesmo domínio.

Pergunta 2 - Concorda que quando duas páginas pertencem ao mesmo domínio, o caminho para elas deve ser idêntico?

Na segunda questão foi perguntado aos participantes para imaginarem duas páginas pertencentes ao mesmo domínio e se nessa situação eles concordavam que o *path* para ambas deveria ser o mesmo. A maioria dos participantes, 50% concordou totalmente com essa ideia, esta foi a resposta com maior frequência. Segundo alguns "se pertencem ao mesmo domínio então devem estar juntas para facilitar a pesquisa entre ambas". Houve mais 42% dos participantes que concordaram de forma parcial com esta ideia e os restantes 8% discordaram de forma parcial. Os últimos 8% dos participantes provavelmente não concordaram com esta ideia por lhes ser difícil identificar duas páginas pertencentes ao mesmo domínio.

Pergunta 3 - Agrupando estas duas páginas numa opção Vaccines no menu principal aumentaria a produtividade e legibilidade da aplicação?

Na terceira pergunta questionou-se os participantes se concordavam que ao agrupar duas páginas numa e adicionar um sub-menu *Vaccines* ao menu principal aumentaria a produtividade e legibilidade da aplicação. A maioria dos participantes(50%) concordou totalmente com esta pergunta, tornando-se esta a resposta mais frequente. A estes 50% de participantes juntaram-se 30% que de forma parcial, com ligeiras dúvidas, concordaram que esta seria a melhor solução para resolver esta anomalia. Houve

contudo 8% dos participantes que ficaram indiferentes com a solução proposta e finalmente 8% que discordaram parcialmente apontando que "poderia" a solução apresentada não ser a mais adequada à resolução da anomalia. Os 8% dos participantes que discordaram foi devido a não terem associado as duas páginas ao mesmo domínio de acordo com as suas respostas anteriores.

Pergunta 4 - Considera este exemplo uma anomalia grave?

Finalmente questionou-se os participantes se consideravam o *Inappropriate Intimacy* uma anomalia grave na conceção da interface. A resposta Ëoncordo parcialmente…foi a mais respondida, ou seja com maior frequência. A maioria dos participantes (cerca de 58% dos participantes) concordou parcialmente que o *Inappropriate Intimacy* prejudicava gravemente a conceção da interface, alguns apontando que a própria usabilidade ficava "destruída" e que as tarefas ficavam bastante mais difíceis de realizar. Juntaram-se 8% dos participantes que não tiveram quais quer dúvidas sobre os graves problemas que este *Usability Smell* trazia. Contudo houve ainda 25% de participantes que ficaram indiferentes sobre considerar o *Inappropriate Intimacy* uma anomalia grave na conceção. Por último houve ainda 8% que discordaram parcialmente que fosse uma anomalia grave. Em conversa com estes últimos participantes, notou-se que apesar de discordarem que o *Inappropriate Intimacy* seja uma anomalia grave na conceção da interface não descartam a possibilidade de o mesmo não ser uma anomalia.

Ao fim deste grupo de questões foi pedido aos participantes que indicassem que alteração ou alterações visual(is) efetuariam para corrigir esta anomalia. Diversas opiniões surgiram contudo, a maioria dos participantes sugeriu agrupar as duas páginas("Vaccines" e " Patient Vaccines") no mesmo menu "Vaccines" ou recorrendo a separadores de modo a que seja mais fácil consultar informação e mover as duas páginas para a mesma para ficar a informação mais legível. Outros participantes sugeriam permitir a funcionalidade de edição de vacinas dentro da página de gestão das vacinas dos pacientes, juntar as duas páginas e aplicação filtros para selecionar o que se pretende e evitar excesso de informação, ou inserir somente uma opção vacines e dentro dessa opção haver a possibilidade de listar as vacinas ou ver as vacinas de certo paciente. Além destas sugestões, houve mais uma de um participante que não considerou o *Inappropriate Intimacy* uma anomalia grave, segundo este participante "pode ser vantajosa manter-se assim, onde numa secção possa ter a informação geral e noutra com mais detalhe e pesquisa avançada".

Por fim foi apresentado aos participantes o *Refactoring* proposto para corrigir o *Inappropriate Intimacy* ao qual se fez uma questão (grupo 3), ver figura 21.

Os resultados da questão encontram-se na figura 33, na tabela 8 encontra-se o resultado das frequências relativas:

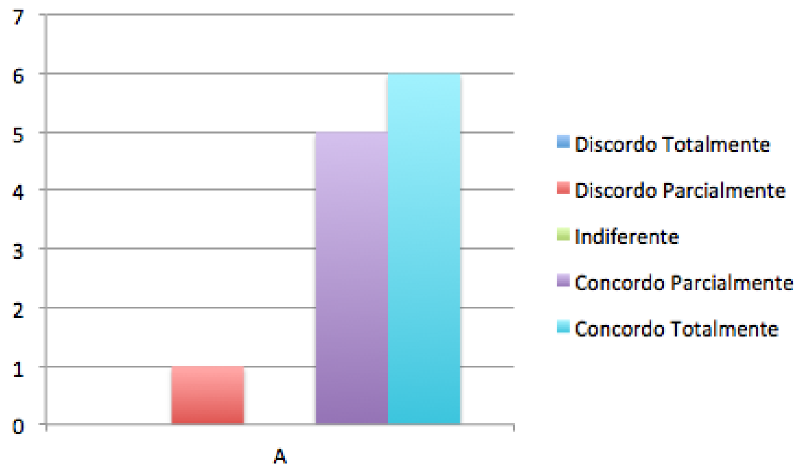


Fig. 33.: Resultados da última pergunta sobre o Inappropriate Intimacy.

	DT	DP	I	CP	CT
Pergunta A	0.0%	8.0%	0.0%	42.0%	50.0%

Tab. 8.: Frequências relativas Inappropriate Intimacy - Grupo 3

Pergunta A - Qual a sua opinião sobre agrupar páginas dentro do mesmo domínio?

Nesta pergunta foi questionado aos participantes, relativamente ao *Refactoring*, qual a opinião que tinham sobre agrupar páginas dentro do mesmo domínio. A maioria dos participantes 50% concordou totalmente com esta solução, alguns afirmando que o *Refactoring* proposto "reduz a possibilidade de haver confusão sobre qual dos caminhos vai ter a cada página", "facilita a navegação pois o contexto(domínio) da opção de listar as vacinas ou ver as vacinas de um paciente é o mesmo", sendo do mesmo domínio e "se o utilizador quer inserir um novo registo de vacina e quer consultar informação acerca desta, como são do mesmo domínio faz sentido que as páginas estejam próximas.". Uma vez que se trata do mesmo domínio torna-se "mais percetível ao utilizador a localização do destino, ao invés de ter caminhos diferentes para atingir objetivos do mesmo domínio". Esta resposta foi considerada a resposta com maior frequência de ocorrer. Outros 42% dos participantes concordou de forma parcial. Apesar de ficarem ligeiramente reticentes sobre agrupar páginas do mesmo domínio para facilitar a resolução das tarefas apontam que "as informações estão mais próximas umas das outras o que facilita a navegação entre as duas páginas" e que "a informação ficou no mesmo domínio evitando menus desnecessários noutros domínios". Em termos de tempos de realização das tarefas e lógica de procura, os participantes afirmaram que "facilita alternar entre as duas opções" e "evita o utilizador de procurar por um menu desorganizado". Contudo houve ainda 8% dos participantes que discordaram parcialmente com o *Refactoring* apresentado. Segundo a análise das questões anteriores estes partici-

pantes discordam porque para eles as páginas não pertencem ao mesmo domínio levando-os também a rejeitar o *Refactoring* para uma anomalia que para eles é inexistente.

5.1.5 *Resultados Tarefa 5: Information Overload*

Os resultados obtidos do grupo 1 de perguntas encontra-se presente na figura 34 em forma de gráfico de barras numa escala de *Likert* e na tabela 9 a qual mostra as frequências relativas de cada uma das perguntas.

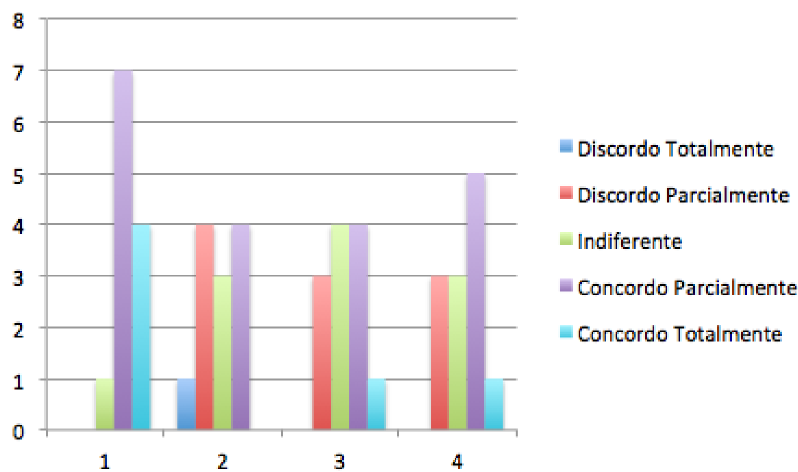


Fig. 34.: Resultados obtidos às perguntas do grupo 1 sobre o Information Overload.

	DT	DP	I	CP	CT
Pergunta 1	0.0%	0.0%	8.0%	58.0%	33.0%
Pergunta 2	8.0%	33.0%	25.0%	33.0%	0.0%
Pergunta 3	0.0%	25.0%	33.0%	33.0%	8.0%
Pergunta 4	0.0%	25.0%	25.0%	42.0%	8.0%

Tab. 9.: Frequências relativas Information Overload - Grupo 1

Pergunta 1 - O excesso de informação visual nas aplicações prejudica a legibilidade e produtividade dos utilizadores?

Nesta primeira pergunta foi questionado aos participantes se o excesso de informação visual nas aplicações prejudica a legibilidade e produtividade dos utilizadores. Com maior frequência de resposta, 58% dos participantes concordou parcialmente que esta anomalia iria causar problemas nesses dois pontos. Outros 33% não tiveram dúvidas e concordaram totalmente, segundo alguns participantes, em algumas aplicações o excesso de informação fazia-os perder o rumo das tarefas a realizar. Ao

olharem para a aplicação a maioria dos participantes ficou preocupada por ver tantos detalhes. Segundo alguns este excesso de detalhe prejudica a perceção do objetivo da aplicação. Houve ainda 8% dos participantes que não consideraram o excesso de informação um problema que afetasse a legibilidade e produtividade da aplicação, quando questionados o porquê da sua resposta justificaram que demoravam mais a realizar a tarefa porque tinham de perceber tudo mas no final conseguiam sempre cumprir com o proposto.

Pergunta 2 - Considera que todas as operações possíveis nessa página são facilmente identificáveis?

Na segunda pergunta pediu-se aos participantes para olharem para a aplicação. Questionaram-se os participantes se todas as operações possíveis nessa página são facilmente identificáveis. Esta questão foi considerada bi-modal, as duas respostas com maior frequência foram: discordo parcialmente e o concordo parcialmente, ambas com 33% de respostas. Um dos pontos que se notou foi o tempo que cada participante utilizava a analisar a página. Quanto mais tempo o participante demorava mais rapidamente chegava à conclusão que afinal as operações da página não eram assim tão facilmente identificáveis. Assim sendo os 33% dos participantes que discordou parcialmente demorou mais tempo a analisar todas as operações da página que os 33% que concordaram parcialmente. Houve ainda 25% dos participantes que ignorou o excesso de informação contudo também não despenderam muito tempo a perceber para que servia cada tarefa. Destes 25% alguns não se importavam de ter excesso de informação desde que fosse possível realizar as tarefas. Finalmente 8% dos participantes discordaram totalmente afirmando que não percebiam para que serviam as opções nem com tanta informação disponível. No total pode-se afinar que 41% dos participantes discordou com esta pergunta.

Pergunta 3 - O excesso de alguns componentes visuais nessa página, como por exemplo a informação de "Tabela" ou de "Jpanel" ajuda a entender melhor como é composto o layout?

Na terceira pergunta apontamos alguns componente mais propriamente *JPanels* com informação como "Tabela" ou "Jpanel" que explicava aos utilizadores o que continham. Questionaram-se os participantes se essa informação os ajudava a entender melhor a estrutura do layout. Novamente as opiniões dividiram-se sendo obtida uma resposta bi-modal em que as respostas mais frequentes foram: discordo parcialmente e concordo parcialmente ambas com 33% das respostas. A maioria dos participantes (cerca de 41%) concordou que essa informação os ajudava a entender melhor dos quais 8% concordou totalmente e os restantes 33% de forma parcial. Houve ainda 25% dos participantes que discordaram parcialmente, não considerando essa informação uma ajuda afirmando que lhes fazia perder muito tempo a perceber a estrutura do layout e os objetivos de cada componente. Finalmente, os restantes 33% consideram que essa informação é indiferente e que não afeta a maneira como interpretam a aplicação.

Pergunta 4 - Considera este exemplo uma anomalia grave?

Por último foi questionado aos participantes se achavam este exemplo uma anomalia grave. A maioria dos participantes, 42%, concordou parcialmente que o *Information Overload* realmente prejudicava a interpretação da aplicação e consequentemente afetava a usabilidade da mesma. Esta foi a resposta mais frequente. Além dos 42% que concordou parcialmente juntaram-se 8% que não tiveram quaisquer dúvidas quando questionados sobre a gravidade da anomalia. Houve ainda 25% que acharam indiferente esta anomalia, para eles o *Information Overload* não era grande problema e não afetava gravemente a usabilidade da aplicação. Segundo alguns participantes o smell fazia com que a tarefa demorasse um pouco mais de tempo, dava um pouco mais de trabalho mas não tornava a aplicação impercetível. Finalmente os últimos 25% discordaram parcialmente que o *Information Overload* fosse uma anomalia grave na usabilidade da aplicação.

Ao fim deste grupo de questões foi pedido a cada participante que indicasse que alterações visuais efetuariam para corrigir esta anomalia (grupo 2). Segundo os participantes as alterações necessárias para eliminar o *Information Overload* passariam por retirar os menus dos *JPanels* como "Table" e "Buttons" deixando opções de pesquisa o mais legível possível para o utilizador, adicionar dropdowns em vez de textfields para que a quantidade de dados não origine mais confusão, mas acima de tudo "reduzir a quantidade de informação extra mostrada". Um participante sugere espaçar um pouco mais os componentes para ajudar na compreensão da informação. Além dos pontos apresentados na tarefa 5, os participantes ainda apontaram outros componentes que estariam mal utilizados levando-nos a questionar se o *Information Overload* não ocorreria por falta de conhecimento dos componentes que se podem utilizar e do seu mau uso. Os componentes apontados pelos participantes foram: os *Input Fields* para introduzir datas onde a melhor solução segundo certas linguagens de programação seria a utilização de *Data Pickers*, *Combo Box* aninhadas sem explicação do que cada item fazia e excesso de *Labels* na aplicação.

Por fim foi apresentado aos participantes o *Refactoring* proposto para corrigir o *Middle Man* ao qual se fez duas questões (grupo 3), ver figura 25.

Os resultados das duas questões encontram-se na figura 35, na tabela 10 encontram-se as frequências relativas referente à questão:

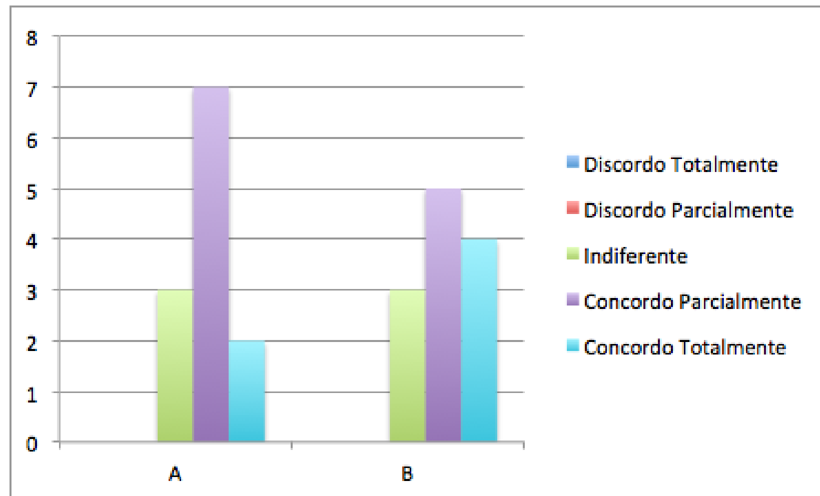


Fig. 35.: Resultados obtidos às perguntas do grupo 3 sobre o Information Overload.

	DT	DP	I	CP	CT
Pergunta A	0.0%	0.0%	25.0%	58.0%	17.0%
Pergunta B	0.0%	0.0%	25.0%	42.0%	33.0%

Tab. 10.: Frequências relativas Information Overload - Grupo 3

Pergunta A - Qual a sua opinião sobre estas alterações?

A primeira pergunta foi questionar os participantes sobre as alterações apresentadas no *Refactoring* da figura 25. A resposta Ëoncordo Parcialmentefoi considerada a resposta com maior frequência. A maioria (cerca de 58%) concordou parcialmente que estas alterações melhoravam significativamente a aplicação, segundo eles "página torna-se visualmente menos agressiva uma vez que não contêm excesso de informação" melhorando "ligeiramente o aspeto visual". Dos restantes participantes, 17% concordaram totalmente com as alterações propostas. Os restantes 25% dos participantes ficaram indiferentes ao *refactoring* apresentado provavelmente por não considerarem o *Information Overload* um *Usability Smell* preocupantes. Quanto à má utilização de componentes, mais precisamente os *Input Fields* que foram trocados para *Date Pickers* os participantes acrescentam que "a possibilidade de escolher a partir deste reduzem a quantidade de informação introduzida pelo utilizador" e melhora a interface.

Pergunta B - Considera que as alterações melhoraram a legibilidade e produtividade da aplicação?

Para ajudar a perceber melhor se o *Information Overload* era uma anomalia preocupante e para avaliar o *Refactoring* questionou-se os participantes se concordavam que as alterações melhoravam a legibilidade e produtividade da aplicação. A maioria dos participantes (42%) concordou parcialmente com

as alterações sendo a resposta com maior frequência, dessa forma pode-se reforçar mais uma vez a ideia que o *Information Overload* realmente afeta a legibilidade da estrutura da aplicação e consequentemente a produtividade das tarefas. A estes 42% juntaram-se mais 33% dos participantes que concordaram totalmente afirmando que com a remoção deste *Usability Smell* as opções ficam mais legíveis e fáceis de entender e que não necessitam de tanto tempo para entender cada um dos componentes. Houve ainda 25% que considerou indiferente as alterações propostas para *Refactoring*, dois participantes não encontraram grandes mudanças em relação à aplicação original.

5.2 DISCUSSÃO DOS RESULTADOS

O teste de usabilidade realizado com o intuito de validar o catálogo de *Usability Smells* foi planeado de forma a apresentar cada um dos *Usability Smells* aos participantes e captar opiniões e reações sobre a ocorrência de cada um dos *smells*. Depois de perceber as opiniões dos participantes foi-lhes pedido para indicarem possíveis resoluções ou *Refactorings* que elimina-se cada uma das anomalias. O objetivo deste segundo passo era obter uma lista de *Refactorings* iguais aos que seriam propostos posteriormente e outros que poderiam ser melhores. Assim com este segundo passo foi feita uma primeira validação aos *Refactorings* que seriam posteriormente apresentados e a obtenção de novos. Finalmente apresentou-se o *Refactoring* para cada um dos *Usability Smells* e questionou-se os participantes sobre o que achavam acerca da resolução apresentada. Desta forma foi realizada uma segunda avaliação ao *Refactoring*. Em todos os *Usability Smells* apresentados foi obtido com satisfação os resultados pretendidos. Ao analisar as propostas de *Refactorings* fornecidas pelos participantes e as apresentadas contra os *Patterns* de desenho de Interfaces apresentados por Jenifer Tidwell (Tidwell, 2010) foram encontrados alguns mecanismos de produção de interfaces contra a e favor dos *Refactorings*. A seguir será analisado cada um dos *Usability Smells* e o melhor *Refactoring* para o eliminar.

Começando pelo *Middle Man*, todos os participantes entenderam rapidamente o conceito deste *Usability Smell* conseguindo identifica-lo facilmente na aplicação. Quando questionados acerca do *Middle Man* ser uma anomalia, 50% dos participantes concordou. A margem pode não ser parecer muita alta mas analisando as restantes respostas verificou-se que apenas 25% discordou de forma parcial, ou seja, com ligeiras dúvidas. Ao analisar as reações dos participantes reparou-se que os participantes que discordaram com este *Usability Smell* não estavam 100% seguros das suas respostas, talvez fosse necessário apresentar-lhes mais casos de estudo, contudo não seria necessário porque a percentagem de participantes que discordou foi reduzida. Ao analisar os patters de Tidwell (Tidwell, 2010) verificou-se que o *Responsive Enabling* poderia ajudar na resolução do *Middle Man*. Por definição este *pattern* é usado em tarefas *step-by-step*, ou seja umas a seguir às outras, e aconselha a desabilitação de botões antes de passar à tarefa seguinte. Antes de habilitar o botão deve ser feita uma validação interna pelo sistema a todas as condições necessárias, se tudo estiver de acordo com o planeado então o botão fica ativo. Ver Figura 36 Assim o *Refactoring* proposto para eliminar o *Middle Man* é: o programador

deve utilizar o *Responsive Enabling* para validar todas as condições, caso as condições não estejam de acordo com o previsto deve ser apresentada uma mensagem ao utilizador no local mais conveniente à sua perceção na mesma página das tarefas. Após validar as condições e estas estejam favoráveis à continuação das tarefas os botões devem ser habilitados no sistema permitindo o utilizador continuar as tarefas.

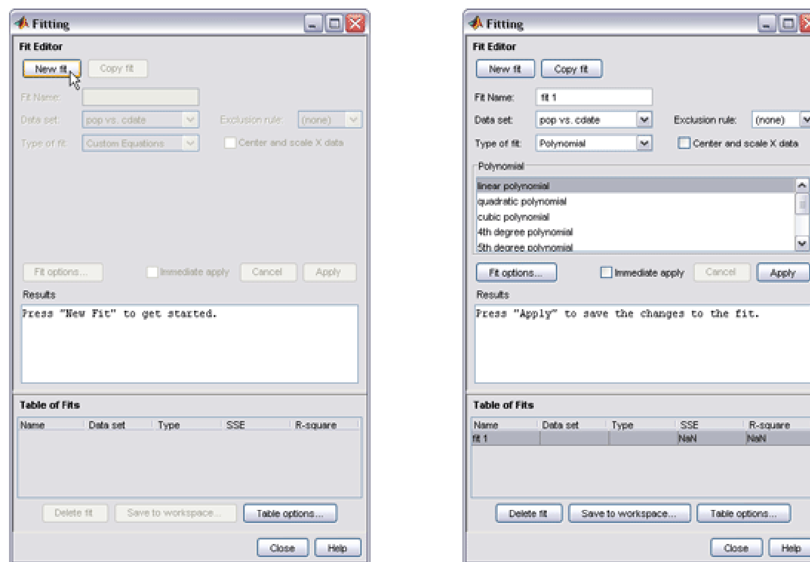


Fig. 36.: Exemplo do Pattern Responsive Enabling, retirado de `http://designinginterfaces.com/firstedition/index.php?page=Responsive_Enabling` a 8 de Outubro de 2014.

O segundo *Usability Smell* a validar foi o *Too Many Layers*. As respostas obtidas a este *Usability Smell* mostram que os participantes compreenderam como ele afeta a usabilidade das aplicações. De acordo com as respostas verificou-se que os participantes ficam aborrecidos quando têm de passar por diversas páginas numa aplicação para concluir uma tarefa. 83% dos participantes concordou que o *Too Many Layers* é realmente uma anomalia. Analisando os patterns de Tidwell (Tidwell, 2010) verificou-se que por exemplo o *One-Window Drilldown* faz ocorrer este *Usability Smell*. Segundo Tidwell este pattern deve ser utilizado em aplicações com muitas páginas ou painéis de informação por as quais o utilizador tem de passar. Se a aplicação for muito grande então ela deve ser dividida em menus pelos quais o utilizador tem de navegar. Tidwell apresenta os menus dos primeiros iPods como exemplo. Ver Figura 37.

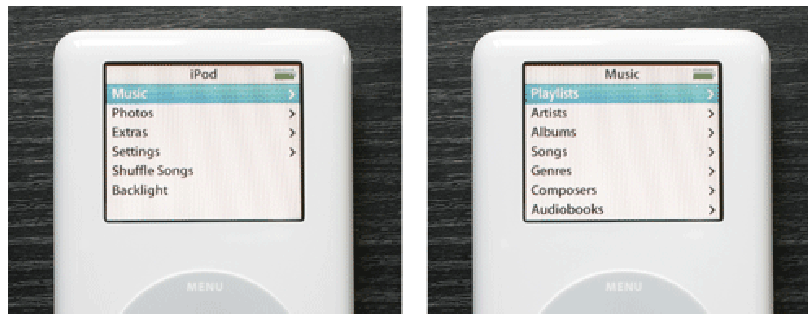


Fig. 37.: Exemplo do Pattern One-Window Drilldown, retirado de http://designinginterfaces.com/firstedition/index.php?page=One-Window_Drilldown a 10 de Outubro de 2014.

Contudo Tidwell apresenta outros patters que ajudam a eliminar o *Too Many Layers* como por exemplo: *Global Navigation* e o *Card Stack*. O primeiro pattern indica que uma grande aplicação deve conter um menu global por onde o utilizador se possa guiar para andar de sessão em sessão assim não necessita de passar por muitas páginas para chegar ao destino. O segundo pattern sugere adicionar na aplicação menus em forma de T̈absägrupadas por tipos. Este pattern deve ser usado em aplicações com muitas páginas ou com muita informação na mesma página. De acordo com estes dois patterns, as soluções apresentadas pelos participantes e o *Refactoring* apresentado no teste de usabilidade definiu-se o *Refactoring* para eliminar o *Too Many Layers*: o programador de forma a evitar um excesso de páginas na aplicação deve recorrer a mecanismos como menus superiors ou laterais para acesso direto a outras páginas ou em forma de T̈abs̈. Além disso o programador deve evitar ao máximo a transmissão de informação para outras páginas quando essa pode ser apresentada na própria. Quando isso acontece o programador está perante um *Middle Man*.

O *Shotgun Surgery* foi o terceiro *Usabily Smell* validado. Um dos receios neste teste era a questão de os participantes não conseguirem identificar este *Usability Smell* na aplicação por ser um *smell* mais relacionado com o código. Apesar do *Shotgun Surgery* estar mais relacionado com o reaproveitamento do código os participantes facilmente o identificaram. Quando questionados sobre o *Shotgun Surgery* ser uma anomalia 92% dos participantes não hesitou concordando com a afirmação. Além de considerarem uma anomalia, 91% dos participantes afirmou que o *Shotgun Surgery* prejudica realmente a produtividade e eficiência das tarefas. Tidwell (Tidwell, 2010) apresenta por exemplo o *Dropdown Chooser* um pattern capazes de ajudar na eliminação deste *Usability Smell*. O *Dropdown Chooser* aconselha o programador a utilizar *Dropdowns* nas aplicações para escolher ou selecionar tarefas a realizar. Ver Figura 38. Analisando este pattern, as respostas dos participantes e a proposta de *Refactoring* apresentada no teste de usabilidade definiu-se o seguinte *Refactoring* para eliminar o *Shotgun Surgery*: o programador deve criar menus ou links para páginas de forma a ajudar na navegação, caso as páginas sejam idênticas o programador deve junta-las numa página comum e utilizar meca-

nismos como por exemplo *Dropdowns* que atualizem a parte comum permitindo o utilizador navegar entre essas páginas sem ter necessidade de sair da página comum.

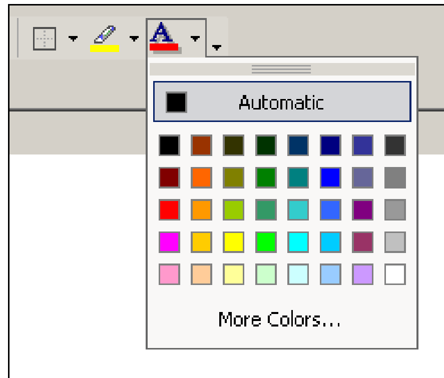


Fig. 38.: Exemplo do Pattern Dropdown Chooser, retirado de http://designinginterfaces.com/firstedition/index.php?page=Dropdown_Chooser a 10 de Outubro de 2014.

O penúltimo *Usability Smell* a ser apresentado no teste de usabilidade foi o *Inappropriate Intimacy*. Verificou-se que também este *Usability Smell* foi facilmente identificado pelos utilizadores, contudo não tão facilmente como o *Middle Man* ou o *Too Many Layers*. 66% dos participantes concordou que este *Usability Smell* é realmente uma anomalia, um valor razoável e esperado. A maior dificuldade foi explicar aos participantes o que são páginas pertencentes ao mesmo domínio. Apesar de demorarem a entender a maioria conseguiu identificar no exemplo o *smell*. Quando questionados se as duas páginas do exemplo pertenciam ao mesmo domínio 92% dos participantes concordou. Tidwell (Tidwell, 2010) apresenta dois patterns que podem ajudar na eliminação do *Inappropriate Intimacy*: *Intriguing Branches* e *Responsive Disclosure*. O primeiro pattern aconselha a utilização de *links* e *labels* em locais que chamem a atenção do utilizador. No caso do *Inappropriate Intimacy* seriam *links* e *labels* para as páginas pertencentes ao mesmo domínio. O segundo patterns apresentado por Tidwell aconselha o programador em aplicações *step-by-step* a usar mecanismos para esconder parte da informação. Essa informação só deve ser apresentada quando for selecionada a opção a que está associada. Ver a Figura 39 onde é apresentado um sistema de entrada na aplicação, consoante a ação é apresentada uma mensagem ou outra. Se duas páginas pertencem ao mesmo domínio pode-se usar o *Intriguing Branches* para criar *links* para ambas as páginas e o *Responsive Disclosure* para apresentar a página escolhida no *link*. De acordo com os patterns de Tidwell, as soluções apresentadas pelos participantes e o *Refactoring* apresentado chegou-se ao seguinte *Refactoring* para eliminar o *Inappropriate Intimacy*: se pelo menos duas páginas pertencerem ao mesmo domínio, apresentando assim o mesmo tipo de informação, ambas devem conter links direcionados para cada uma delas agrupados no mesmo menu, se for possível o programador deve tentar agrupar as páginas numa só página com o cuidado de não apresentar excesso de informação senão origina o *Information Overload*.

Fig. 39.: Exemplo do Pattern Responsive Disclosure, retirado de http://designinginterfaces.com/firstedition/index.php?page=Responsive_Disclosure a 12 de Outubro de 2014.

O último *Usability Smell* a ser validado foi o *Information Overload*. Este *Usability Smell* consiste no aparecimento de excesso de informação na página. 92% dos participantes concordaram que este *Usability Smell* prejudica a legibilidade e produtividade das tarefas e da aplicação. Apesar de ser um valor baixo 50% dos participantes concorda que o *Information Overload* é realmente uma anomalia grave. Dos restantes 50% dos participantes, apenas 25% discorda que seja uma anomalia grave. Provavelmente a falta de mais exemplos desde *Usability Smell* tenha originado a reduzida percentagem de participantes que concorda que o *Information Overload* seja uma anomalia grave. Segundo analise dos patterns de Tidwell (Tidwell, 2010) o *Information Overload* poderia ser removido com o auxilio de por exemplo: *Extras On Demand*, *Closable Panels* ou o *Clear Entry Points*. O *Extras On Demand* é um patterns que aconselha o programador a apresentar a informação mais importante na página inicial e a restante deve estar oculta até ser selecionada o botão ou *link* com a ação associada. O programador pode por exemplo utilizar o *Closable Panels* para ocultar essa informação evitando o excesso de informação. Ver Figura 40. De forma a deixar a aplicação ainda mais limpa o programador pode utilizar o *Clear Entry Points*. Segundo Tidwell este pattern deve ser usado para simplificar a aplicação, ele recomenda que na interface devam existir apenas dos pontos mais importantes assim o utilizador à primeira vista não fica assustado com os passos a realizar para determinadas tarefas. A ideia do pattern passa por colocar alguns pontos e consoante o que o utilizador escolhe apresentar nova informação. Ver Figura 41. Analisando todos estes patterns, as soluções apresentadas pelos participantes e o *Refactoring* apresentado chegou-se ao seguinte *Refactoring* para eliminar o *Information Overload*: o programador deve em primeiro lugar de forma a encontrar mais facilmente o excesso de informação realizar testes de usabilidade com diversos participantes tal e qual se fez neste estudo. Se com o estudo encontrar excesso de informação, o programador deve remover essa informação sem prejudicar a aplicação. A informação pode ser escondida ou removida se não for necessária e apenas deve ser apresentada os dados mais importantes em cada página.

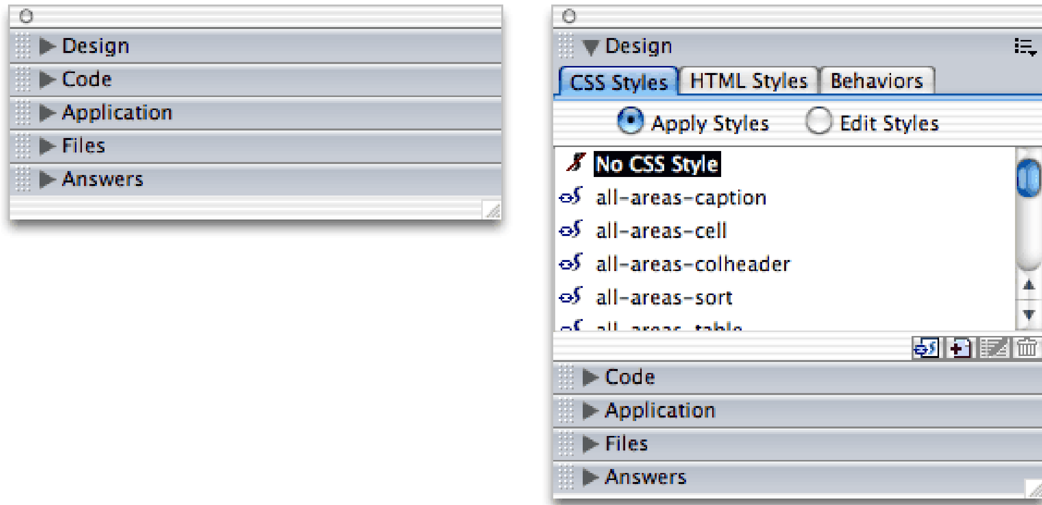


Fig. 40.: Exemplo do Pattern Closable Panels, retirado de http://designinginterfaces.com/firstedition/index.php?page=Closable_Panels a 13 de Outubro de 2014.

Fig. 41.: Exemplo do Pattern Clear Entry Points, retirado de http://designinginterfaces.com/firstedition/index.php?page=Clear_Entry_Points a 13 de Outubro de 2014.

6

CONCLUSIONS AND FUTURE WORK

Esta dissertação teve como objetivo mapear um conjunto de *code smells* e adaptá-los a aplicações interativas/web. Para tal começou por se analisar um conjunto de trabalhos relacionados com o tema entre os quais, o de Fowler (2002) e Hermans et al. (2012). No primeiro trabalho Fowler apresenta um conjunto de *code smells* e os respetivos *refactorings*. Este trabalho ajudou a entender melhor o que são essas anomalias que ocorrem no código das aplicações, além disso ajudou a definir um lote de *code smells* a analisar. No segundo estudo Hermans et al. (2012) realizaram um mapeamento dos *code smells* de Fowler (2002) para folhas de cálculos (*spreadsheets*), segundo eles as classes seriam *worksheets* e os métodos seriam células e formulas. O mapeamento feito por Hermans et al. (2012) ajudou a definir a estrutura do estudo que viria a validar o lote de *code smells* retirados do estudo de Fowler (2002).

Depois de analisar e entender o que eram *code smells* o passo seguinte foi definir o que seriam os *Usability Smells* e como eles ocorrem nas aplicações interativas/web. O mais difícil neste passo foi criar uma relação entre as linguagens orientadas a objetos e as aplicações com *user interface* que permitisse relacionar os *code smells* com os *usability smells*.

Após definir o que seriam os *Usability Smells* escreveu-se um catálogo com o mapeamento, o que é cada um dos *Usability Smells* e como aparecem nas aplicações com alguns exemplos. O catálogo foi posteriormente validado recorrendo a um teste de usabilidade. No teste foi utilizado como caso de estudo uma aplicação de gestão hospitalar em Angal no Uganda. O teste consistiu num conjunto de tarefas onde era apresentado aos participantes cada um dos cinco *Usability Smells*. Cada tarefa referente a cada um dos *Usability Smell* consistia num conjunto de perguntas de escolha múltipla avaliadas numa escala de *Likert* e de resposta aberta. Cada tarefa tinha como objetivo explicava o *Usability Smell* ao participantes, questionava-os sobre como eliminar essas anomalias e por último apresentava uma proposta de *Refactoring*.

Com a realização do teste de usabilidade foi possível validar separadamente cada um dos *Usability Smells* e definir o *Refactoring* para cada um deles. Ao construir o teste de usabilidade verificou-se que o *Middle Man* é dos *Usability Smells* mais fáceis de encontrar e definir ao contrário de por exemplo o *Inappropriate Intimacy*. Os resultados do teste foram todos conforme esperado. Os participantes graças ao teste conseguiram entender facilmente os *Usability Smells* e identificá-los no caso de estudo. Além disso todos os *Usability Smells* foram considerados anomalias segundo os participantes. Os *Re-*

factorings propostos pelos participantes coincidiram com os *Refactorings* apresentados o que ajudou também a validar as soluções apresentadas e a definir os *Refactorings* finais para cada *Usability Smell*.

Relativamente às perguntas de investigação apresentadas na Secção 1.3 os resultados foram os seguintes:

[R1] Existirá possibilidade de identificar usability smells em aplicaçoes interactivas?

A resposta é afirmativa, contudo nem todos os *Usability Smells* foram validados como o caso do *Feature Envy*. Este *Usability Smell* não apareceu na aplicação caso de estudo contudo isso não significa que não exista. Os restantes *Usability Smells* foram todos identificados e validados.

[R2] Quais os usability smells mais comuns? E será que expõem ameaças na qualidade e integridade?

Os *Usability Smells* mais comuns e por sua vez mais fáceis de encontrar foram o *Middle Man*, o *Too Many Layers* e o *Information Overload*. Os restantes *Usability Smells* apresentaram ligeiras dificuldades em definir e encontrar. Todos os *Usability Smells* apresentaram de certa forma ameaças na qualidade das aplicações. A integridade das mesmas foram postas à prova e em alguns casos a usabilidade foi realmente afetada. Dos *Usability Smells* que os participantes comentaram mais o *Too Many Layers* foi o que afetava mais a aplicação.

[R3] Qual a melhor solução para identificar os usability smells?

Até agora a identificação foi toda manual, escolheu-se a aplicação e visualmente tentou-se encontrar cada um dos *smells*. O resultado desta análise foi realizado com sucesso.

Como trabalho futuro e de forma a automatizar todo este processo sugere-se a criação de ferramentas de análise de interfaces onde fosse possível mapear o comportamento de uma aplicação. No mapeamento é preciso ter em atenção as ações de cada componente, o conteúdo de cada página e as ligações entre elas. Juntando estes três objetivos e uma *framework* capaz de interagir com diversos tipos de aplicações seria possível desenvolver uma ferramenta de análise de interfaces para cada *Usability Smell*. É de relembrar que o tema é bastante abstrato, desta forma é possível que não seja possível definir uma ferramenta comum a todos os *Usability Smells*.

BIBLIOGRAFIA

Asish Arora and Alfonso Gambardella. The globalization of the software industry: Perspectives and opportunities for developed and developing countries. Working Paper 10538, National Bureau of Economic Research, June 2004. URL http://www.nber.org/papers/w10538.

J. Cunha, J.P. Fernandes, P. Martins, J. Mendes, and J. Saraiva. Smellsheet detective: A tool for detecting bad smells in spreadsheets. In *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pages 243–244, 2012a. doi: 10.1109/VLHCC.2012.6344535.

Jácome Cunha, João P Fernandes, Hugo Ribeiro, and João Saraiva. Towards a catalog of spreadsheet smells. In *Computational Science and Its Applications–ICCSA 2012*, pages 202–216. Springer, 2012b.

Martin Fowler. Refactoring: Improving the design of existing code. In *Proceedings of the Second XP Universe and First Agile Universe Conference on Extreme Programming and Agile Methods - XP/Agile Universe 2002*, pages 256–, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-44024-0. URL http://dl.acm.org/citation.cfm?id=647276.722308.

Martin Fowler. Catalog of refactorings, @MISC, June 2009. URL http://refactoring.com/catalog/.

S. G. Ganesh, Tushar Sharma, and Girish Suryanarayana. Towards a principle-based classification of structural design smells. *Journal of Object Technology*, 12(2):1: 1–29, 2013.

Felienne Hermans, Martin Pinzger, and Arie van Deursen. Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 441–451, Piscataway, NJ, USA, 2012. IEEE Press. ISBN 978-1-4673-1067-3. URL http://dl.acm.org/citation.cfm?id=2337223.2337275.

Taeho Lee, Taewan Gu, and Jongmoon Baik. Mnd-scemp: an empirical study of a software cost estimation modeling process in the defense domain. *Empirical Software Engineering*, 19(1):213–240, 2014. ISSN 1382-3256. doi: 10.1007/s10664-012-9220-1. URL http://dx.doi.org/10.1007/s10664-012-9220-1.

Brad A. Myers and Mary Beth Rosson. Survey on user interface programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '92, pages 195–202, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. doi: 10.1145/142750.142789. URL http://doi.acm.org/10.1145/142750.142789.

Jakob JAKOB NIELSEN Nielsen. How many test users in a usability study?, @MISC, June 2012. URL http://www.nngroup.com/articles/how-many-test-users/.

William F Opdyke. *Refactoring object-oriented frameworks*. PhD thesis, University of Illinois, 1992.

F. Simon, F. Steinbruckner, and C. Lewerentz. Metrics based refactoring. In *Software Maintenance and Reengineering, 2001. Fifth European Conference on*, pages 30–38, 2001. doi: 10.1109/.2001.914965.

J. Tidwell. *Designing Interfaces*. O'Reilly Media, 2010. ISBN 9781449302733. URL http://www.google.pt/books?id=5gvOU9X0fu0C.

R.K. Yin. *Case Study Research: Design and Methods*. SAGE Publications, 2013. ISBN 9781483302003. URL http://www.google.pt/books?id=AjV1AwAAQBAJ.

Parte I

APENDICES

TESTE DE USABILIDADE

Teste de Usabilidade nº1

Aplicação: Open Hospital - Hospital Information System:
Fonte: http://sourceforge.net/projects/openhospital/?source=directory

Objetivo 1: Este teste tem como objetivo validar um catálogo de *Usability Smells*. Um *Usability Smell* é uma anomalia na interface gráfica da aplicação que prejudica a sua legibilidade, manutenção e evolução.

Objetivo 2: Durante este teste será pedido ao participante que execute um conjunto de tarefas e posteriormente as avalie. O participante deve cingir-se ao que é pedido e ignorar as restantes funcionalidades do sistema.

Objetivo 3: Todas as perguntas são avaliadas numa escala de 1 a 5, onde 1 significa "Não Concordo" e 5 "Concordo Plenamente". O valor 3 significa que a questão é indiferente para o participante.

Tarefa nº1 – Middle Man

Nesta tarefa o participante deve editar uma tabela, para isso deve selecionar as seguintes opções na aplicação:

General Data >> Types >> Admission Type >> Edit

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) O aparecimento de janelas pop-up é a melhor forma de fornecer indicações ao utilizador?					
2) Concorda que o aparecimento de uma janela só para indicar que nenhum objeto se encontra selecionado o ajuda a realizar a tarefa de forma mais efetiva?					
3) Preferia que a informação da janela fosse apresentada na mesma janela onde se encontra a tabela?					
4) Considera este exemplo uma anomalia na concepção da interface?					

Que alteração visual efetuaria para corrigir esta anomalia?

Visualize a imagem "Main", prestando atenção à forma como é apresentada a mensagem de erro.

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Considera que apresentar a mensagem, quer seja ela informativa ou de erro, na própria janela é a solução mais correta?					
2) O que acha da alteração feita relativamente à aplicação original?					

Justifique as suas respostas

Tarefa nº2 – To Many Layers

Nesta tarefa o participante deve avaliar o total de passos para realizar a tarefa de inserir um novo tipo de dados:

General Data >> Types >> Exame Type >> New

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Concorda que o total de janelas que tem de percorrer para realizar a tarefa é o mais correto?					
2) Concorda que todas estes passos poderiam ser poupados se estas opções estivessem num menu?					
3) Considera este exemplo uma anomalia na concepção da interface?					

Que alteração visual efetuaria para corrigir esta anomalia?

Visualize a imagem “Main”, prestando atenção à barra superior de opções.

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Qual a sua opinião quanto a agrupar estas janelas em menus/tabs?					

Justifique a sua resposta:

Tarefa nº3 – Shotgun Surgery

Nesta tarefa o participante deverá realizar as respectivas tarefas e prestar atenção há disposição e aos elementos das janelas:

Tarefa 1: *General Data >> Types >> Discharge Type >> Close*
Tarefa 2: *General Data >> Types >> Delivery Type >> Close*
Tarefa 3: *General Data >> Types >> Exame Type >> Close*

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Concorda que a disposição das janelas e a maneira como tem de procurar informação dento dos *Types* prejudica a produtividade e eficiência das tarefas?					
2) As janelas são todas semelhantes concorda que se devam juntar numa só?					
3) Provavelmente o mesmo código foi usado para programar a parte comum das janelas, se o tivesse de repetir em 5, 10 ou mais lugares lembrasseía de todos eles?					
4) Considera este exemplo uma anomalia na concepção da interface?					

Que alteração visual efetuaria para corrigir esta anomalia?

___
___
___
___
___
___
___

Visualize a imagem "Main", nessa imagem todos os tipos de dados foram agrupados na área "Types".

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Qual a sua opinião sobre agrupar janelas idênticas numa única janela?					

Justifique a sua resposta:

___
___
___
___
___
___
___

Tarefa nº4 – Inappropriate Intimacy

Nesta tarefa o participante deverá realizar as respectivas tarefas e prestar atenção a disposição e elementos das janelas:

Tarefa 1: *General Data >> Vaccine >> Close*
Tarefa 2: *Patient Vaccines >> Close*

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Concorda que estas duas janelas pertencem ao mesmo domínio?					
2) Concorda que quando duas janelas pertencem ao mesmo domínio, o caminho para elas deve ser idêntico?					
3) Agrupando estas duas janelas numa opção *Vaccines* no menu principal aumentaria a produtividade e legibilidade da aplicação?					
3) Considera este exemplo uma anomalia grave?					

Que alteração visual efetuaria para corrigir esta anomalia?

Visualize a imagem "Main Vaccines", nessa imagem as janelas "Vaccines" e "Patient Vacciones" foram agrupadas num sub-menu.

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Qual a sua opinião sobre agrupar janelas dentro do mesmo domínio?					

Justifique a sua resposta:

Tarefa nº5 – Information Overload

Nesta tarefa o participante deverá prestar atenção à seguinte janela com informação do stock farmacêutico:

Pharmacy >> Pharmaceuticals Stock >> Close

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) O excesso de informação visual nas aplicações prejudica a legibilidade e produtividade dos utilizadores?					
2) Considera que todas as operações possíveis nessa janela são facilmente identificáveis?					
3) O excesso de alguns componentes visuais nessa janela, como por exemplo a informação de "Tabela" ou de "JPanel" ajuda a entender melhor como é composto o layout?					
4) Considera este exemplo uma anomalia grave?					

Que alteração visual efetuaria para corrigir esta anomalia?

Visualize a imagem "Pharmacy", nessa imagem foram retirados alguns elementos visuais que identificavam o que era cada conjunto de opções (compare com a janela "Pharmaceuticals Stock" da interface original).

Perguntas: (Discordo Totalmente) 1 ☐☐☒☐☐ 5 (Concordo Totalmente)	1	2	3	4	5
1) Qual a sua opinião sobre estas alterações?					
2) Considera que as alterações melhoraram a legibilidade e produtividade da aplicação?					

Justifique as suas respostas

B

CODE SMELLS DE MARTIN FOWLER

Neste anexo serão apresentados os restantes *code smells* de Fowler (2002).

DUPLICATED CODE

Às vezes o programador ou por questão de esquecimento ou de preguiça não se apercebe que está a repetir código em mais que um lugar. Esta repetição de código piora quando o mesmo acontece dentro da mesma classe, ou pior ainda, dentro da mesma função. Relembrando a noção de *code smell*, a repetição de código não é um erro mas que "cheira muito mal cheira"! Ao unificarmos esse código a aplicação fica melhor e mais legível.

Para resolver este *smell* Fowler sugere a utilização do *refactoring Extract Method* 2.3.2 invocando depois esse método nos locais que sofreram essa alteração.

Outro problema comum é quando a duplicação ocorre um duas subclasses pertencentes há mesma ou mesmas classes e sem filhos. Neste caso o processo passa por eliminar a duplicação usando o *Extract Method* 2.3.2 nas classes e depois aplicar *Pull Up Field* C.

Se o código for similar mas não exatamente igual, o que Fowler sugere é utilizar o *refactoring Extract Method* 2.3.2 de forma a separar os bits iguais dos que são diferentes e depois aplicar o *Form Template Method* C

Finalmente se o programa tem dois métodos que fazem exatamente a mesma coisa mas com algoritmos diferentes então a solução ideal é limpar/remover um dos algoritmos usando depois o *Substitute Algorithm* C.

Um último caso possível deste smell segundo Fowler, seria quando duas classes que não estão relacionadas entre si têm o mesmo código. Nesta situação o ideal é aplicar a uma das classes o *Extract Class* 2.3.2 e depois usar esse novo componente na outra classe. Além disto se existir um método pertencente a uma classe ou sub-classe que seja chamado em mais que um sítio o ideal seria perceber qual o melhor sítio onde ele devesse estar e movê-lo para lá.

LONG METHOD

Com o passar dos anos os programadores vão ganhando mais experiência, experiência essa que os ajuda a perceber que quanto menor forem os métodos mais fácil é a sua compreensão. Contudo nem sempre isso acontece e quando o programador percebe já tamanho do método ultrapassa o limite do razoável e legível. Se bem que existe solução para este smell, mas antes de passarmos à solução temos de relembrar uma coisa: por mais pequeno que seja o método nem sempre o código é percetível, por isso quanto melhor e mais intuitivo do objetivo for o nome do método melhor é a sua compreensão. Segundo Fowler "95% das vezes" o que se tem a fazer para encortar o método é usar o *Extract Methot* 2.3.2, descobrir partes do método que trabalham bem juntas e fazer um novo método.

Se o método tem muitos parâmetros e variáveis temporais estas vão também nos métodos extraídos. Porém pode acontecer irem demasiados parâmetros para esses métodos extraídos, nesse caso o novo método pode ficar menos legível que o inicial usando assim o *Replace Temp with Query* C apagando assim as variáveis temporárias e os *Introduce Parameter Object* C e *Preserve Whole Object* C para encurtar a lista de parâmetros. Se mesmo assim a lista não encortar então a solução passar por usar o *Replace Method with Method Object* C.

Outra questão que o programador deve ter em atenção é quando tem ciclos e condições no seu código, estes dois exemplos dão-lhe ótimos sinais para que sejam extraídos simplificando a interpretação de por exemplo hierarquias de ciclos. Para corrigir este problema a solução passa por usar o *Decompose Conditional* C extraindo o ciclo e o código dentro dele, do método onde se encontra.

LARGE CLASS

Como nos métodos as classes também podem chegar a determinado ponto em que obriga-las a fazer todas as tarefas pode ser pior tanto para a manutenção e interpretação do código como para o desempenho da aplicação. Por exemplo, quando um programador aprende a escrever código no início toda a lógica é introduzida no ciclo principal deixando o código pouco percetível. Com o evoluir do seu conhecimento começa a conhecer ferramentas das linguagens como por exemplo funções que permitem deixar o código mais percetível, limpo e reaproveitável. O mesmo acontece com as classes, Fowler para resolver este problema sugere aplicar o *Extract Classe* 2.3.2 juntado as variáveis que trabalhariam bem em conjunto num novo componente. Se for mais apropriado transformar o novo componente numa subclasse então a solução é aplicar o *Extract Subclass* C. Se a classe não utilizar todas as suas variáveis de instância ao mesmo tempo então o *Extract Class* 2.3.2 ou o *Extract Subclass* C podem ser utilizados bastantes vezes.

Como descrito anteriormente, quando uma classe tem muitas variáveis a melhor solução é aplicar o o *Extract Class* 2.3.2 ou o *Extract Subclass* C, contudo um truque que Fowler aconselha a fazer para ajudar a encontrar a melhor solução é: ver como são utilizadas as classes e de acordo com essa utilização perceber como podem ser quebradas usando para isso o *Extract Interface* C.

Um último caso que Fowler apresenta é a das GUI class, nestes casos de forma a entender melhor a classe deve-se separar alguma informação. Contudo esta separação muitas vezes leva à duplicação de código obrigando o programador a uma constante sincronização em ambas as partes. O *refactoring* que Fowler aconselha para este exemplo é o *Duplicate Observer Data* C.

LONG PARAMETER LIST

Normalmente os programadores tendem a passar todas as variáveis necessárias pelos parâmetros, caso isto não fosse possível teriam de usar variáveis globais o que tornaria o que daria muitas dores de cabeça. Com os prós vêm os contra e neste exemplo as variáveis podem ocorrer em longas listas de informação deixando o método incompreensível. A programação orientada a objetos veio ajudar nessa resolução, o programador pode passar os atributos que quiser nos objetos e caso o objeto necessite de outros atributos este pode pedi-los a outros objetos. Fowler sugere a utilização do *Replace Parameter with Method* C quando se pretende dados de um parâmetro de um objeto que se conhece. Quando pretendemos tirar uma grande quantidade de dados e substituir pelo objeto em si, o melhor processo é utilisar o *Preserve Whole Object* C, caso existam dados que não contenham objetos lógicos então deve-se usar o *Introduce Parameter Object* C.

Fowler contudo acrescenta que existe uma importante exceção ao aplicar estas alterações: quando o programador não quer explicitamente criar uma dependência do objeto que invoca ao objeto maior. Neste caso Fowler sugere que é aceitável descompactar os dados e enviar uma longa lista de parâmetros contudo irá tornar-se bastante doloroso para o programador a manutenção e interpretação do código e nessa altura ele deve pensar novamente na estrutura do código.

DIVERGENT CHANGE

Todo o programador tende a estruturar o seu software de forma a que as alterações sejam fáceis de aplicar e o código limpo, mas muitas vezes depara-se que ao adicionar ou alterar por exemplo uma nova base de dados ou outros componentes do software diversas classes terão também de ser alteradas para se adaptarem a essa nova mudança. Quando essas alterações são difíceis de realizar então o programador encontra-se com este *code smell*. Fowler sugere a utilização do *Extract Class* 2.3.2 para colocar tudo como deve ser.

DATA CLUMPS

Fowler iguala grupos de dados ou itens de dados a crianças, estas andam sempre em grupos. Podemos encontrar esses grupos de três ou quatro itens em vários lugares como por exemplo campos de uma classe ou parâmetros nas assinaturas dos métodos tendendo a adquiri-los como seus. Simplificando e unificando estes itens as classes torna-se mais legíveis e o código mais limpo. Para corrigir melhorar

o código, Fowler sugere que o programador em primeiro lugar procure onde esses grupos de itens se encontram, depois usar o *Extract Class* 2.3.2 nos campos e transformar os grupos em objetos, em seguida o programador deve dar atenção às assinaturas dos métodos usando o *Introduce Parameter Object* C ou o *Preserve Whole Object* C de forma a transferir os parâmetros da assinatura para o corpo do método.

PRIMITIVE OBSESSION

Normalmente um programador depara-se com a seguinte pergunta "Qual a melhor estrutura de dados para representar um conjunto de informação?", a resposta baseia-se simplesmente a estruturas de tipos primitivos ou estruturados. Um ponto a favor das linguagens orientadas a objetos é a quebra de ligações entre estes dois tipos de estruturas. O programador pode escolher o melhor tipo de dados para representar o que pretende, isto é utilizar um tipo inteiro para guardar um número de telefone ou criar uma classe chamada "Telefone" que faça o mesmo trabalho. Contudo quando o programador começa a confundir e misturar estas estruturas o código começa a ficar confuso e aparece o *Primitive Obsession*.

Quando um programador começa a utilizar as programações orientadas a objetos fica meio relutante quanto ao tipo de dados que deve usar. A resolução, segundo Fowler, para este *smell* passa por usar o *Replace Data Value with Object* C nos atributos individuais. No caso da informação ser um conjunto de dados de um objeto então o que se deve fazer é retirar esse conjunto de dados e transforma-los numa classe usando o *Replace Type Code with Class* C, isto é se o valor não afetar o comportamento do objeto. Se existirem condições que dependam desses conjuntos de dados então deve-se usar o *Replace Code with Subclasses* C ou o *Replace Type Code with State/Strategy* C.

Se existir um conjunto de campos que obrigatoriamente necessitam de estar juntos a solução é utilizar o *Extract Class* 2.3.2, se essas primitivas forem uma lista de parâmetros o ideal é aplicar o *Introduce Parameter Object* C ou se forem *Arrays* o *Replace Array with Object* C.

SWITCH STATEMENTS

Segundo Fowler este *smell* ocorre principalmente nas linguagens orientadas a objetos. Este *smell* baseia-se na falta de comandos *switch*. Este tipo de mecanismo de controlo tipicamente aparece duplicado no código e é utilizado principalmente em condições "Testar se A senão passar para o teste seguinte. Testar se B senão para o teste seguinte." e por aí fora. Normalmente estes testes implicam ir ao método ou classe onde se encontra o teste, Fowler sugere usar o *Extract Method* 2.3.2 onde se encontra o teste *switch* e coloca-lo na classe onde é necessário haver polimorfismo usando o *Move Method* 2.3.2. Neste ponto o programador necessita de decidir que quer substituir esse código por uma subclasse ou por um objeto estado que represente as alterações de comportamento usando para isso: *Replace Type Code with Subclasse* C ou *Replace Conditional with State/Strategy* C. Para

além disso se for necessário definir uma estrutura hierárquica a solução passa por usar também o *Replace Conditional with Polymorphism* C. Por último se apenas algumas condições afetarem um único método ou se uma condição ficar a *NULL* deve-se usar os *refactorings Replace Parameter with Explicit Methods* C e *Introduce Null Object* C.

PARALLEL INHERITANCE HIERARCHIES

Em alguns casos o programador ao criar uma subclasse pertencente a uma classe tem de criar uma subclasse pertencente a outra. Por exemplo a classe *Figuras* com as subclasses: *Rectângulo*, *Circulo* e *Triângulo*, e a classe *Desenho* com as subclasses: *DesenharCirculo*, *Desenhar Rectangulo* e *DesenharTriangulo*. Sempre que se cria uma nova subclasse de *Figuras* para representar uma figura geométrica o programador tem também de criar a respetiva subclasse para desenhar essa figura na classe *Desenho*. Esta estrutura mostra a existência deste *smell*, o que Fowler sugere para o eliminar e assegurar uma correta hierarquia nas classes é aplicar os *refactorings Move Method* 2.3.2 e o *Move Field* 2.3.2.

LAZY CLASS

Quando o programador cria classes tende a que estas sejam fáceis de manter e entender, contudo quando uma classe começa a apresentar problemas na sua atualização, manutenção e interpretação deve ser eliminada.

Fowler sugere a utilização do *Collapse Hierarchy* C quando comecem a existir subclasses que não façam muita coisa e o *Inline Class* 2.3.2 quando existirem variáveis ou métodos inutilizadas.

SPECULATIVE GENERALITY

Algumas vezes quando é pedido a um programador para fazer um determinado software o cliente pede mil e uma funcionalidades que nunca serão utilizadas. Estes pedidos obrigam o programador a criar classes e métodos que acabarão por não realizar nenhum trabalho. Além dos métodos que não são usados outros terão cabeçalhos com listas quase infinitas de parâmetros com a mesma sorte dos métodos. Neste caso estamos perante este *smell*.

Nestes casos o que se deve fazer para simplificar as coisas é adaptar o seguinte pensamento: "Não será útil então não será feito." senão a manutenção desses componentes fará com que os custos de atualização do software cresçam, desnecessariamente, mais que o previsto.

Fowler sugere a utilização do *Collapse Hierarchy* C quando existem classes que não estão a fazer nada, o *Inline Class* 2.3.2 quando existem delegações a outras classes que não sejam úteis e o *Remove Parameter* C para eliminar os parâmetros inúteis.

Também pode acontecer existirem métodos com nome fora do normal em que o nome sugere um objetivo e ele acaba por realizar outro, nesse caso deve ser usado o *Rename Method* C.

TEMPORARY FIELD

Normalmente quando se criam variáveis num objeto são para ser usadas, no entanto existem alturas em que algumas variáveis recebem valores e outras não. Estas variáveis são variáveis de circunstância pois o seu valor varia de caso para caso. Entender este tipo de variáveis normalmente é demoroso e segundo Fowler pode até enlouquecer o programador. Nestes casos Fowler sugere que se aplique o *Extract Class* 2.3.2 e se crie um local para essas variáveis com todo o código associado a elas. Possivelmente se existirem condições associadas a essas variáveis para isso deve-se usar o *Introduce Null Object* C para criar um componente alternativo quando as condições são falsas.

MESSAGE CHAINS

Este *smell* ocorre normalmente quando existem cadeias de pedidos entre objetos ou seja: um cliente questiona um objeto por outro que por sua vez questiona outro objeto sobre um quarto objeto e por aí fora. O cliente fica dependente dessa cadeia indeterminável de pedidos *get* entre objetos. Nestes casos uma alteração nos objetos intermédios obriga à alteração do cliente. Fowler sugere o seguinte procedimento: aplicar o *Hide Delegate* 2.3.2 em vários pontos da cadeia transformando as delegações em subclasses, contudo a utilização deste *refactoring* faz aparecer um *Middle Man* 2.3.1. Em alternativa a este processo deve-se ver onde é usado o objeto final e que código ele utiliza, depois aplicar o *Extract Method* 2.3.2 para remover esse código e o *Move Method* 2.3.2 para o colocar na cadeia, deste modo deve ser introduzido um método para ajudar os clientes de um objeto sempre que estes queiram navegar na cadeia de objetos.

Alternative Classes with Different Interfaces

Este *smell* ocorre quando duas classes são parecidas no interior mas com diferentes interfaces, neste caso deve ser possível modifica-las para usarem a mesma interface. Para efetuar esta alteração Fowler aconselha usar o *Rename Method* C nos métodos que façam a mesma coisa mas que tenham assinaturas diferentes e caso não seja suficiente continuar a aplicar o *Move Method* 2.3.2 no comportamento das classes até que fiquem idênticas. Se nestas alterações for preciso mexer em código redundante a solução é usar o *Extract Superclass* C.

INCOMPLETE LIBRARY CLASS

Normalmente as bibliotecas de software tendem a ser grandes e a conter inúmeros métodos para resolverem os objetivos para que são criadas. Como os programadores são humanos e "errar é humano" normalmente ocorrem falhas: assinaturas de métodos erradas ou até mesmo faltas de métodos. Por vezes estas falhas ocorrem quando as bibliotecas são demasiado grandes e estruturadas sendo a alteração quase impossível de realizar. Quando o programador se apercebe que é tarde de mais e não consegue adicionar o método que pretende este acaba por ficar colado a outra classe que não a correta usando o *Introduce Foreign Method* C. Quando se pretende alterar um método já existente a solução é isola-lo primeiro para o atualizar depois, *Introduce Local Extension* C.

DATA CLASS

Segundo Fowler este *smell* ocorre em classes que contêm apenas campos, *gets* e *sets* e nada mais que isso. Fowler compara estas classes a crianças, funcionam até certa altura, mas quando querem operar como gente graúda necessitam adquirir alguma responsabilidade.

Normalmente os dados dessas classes são manipulados a maior parte das vezes por outras classes que por ela própria. Quando isto acontece ele sugere aplicar o *Encapsulate Field* C nos campos das *Data Class* antes que os dados sejam usados por outras classes. Se esses dados forem coleções então deve-se usar o *Encapsulate Collection* C. No caso de existirem métodos nas classes que não devam ser alterados deve-se usar o *Remove Setting Method* C.

Para além dos campos das *Data Class* o programador também deve olhar para os métodos *gets* e *sets* e ver quais são usados por outras classes. Assim usando o *Move Method* 2.3.2 transfere o comportamento para dentro da *Data Class*. Se o método não poder ser todo movido então deve-se usar o *Extract Method* 2.3.2 para criar um novo o que permitirá usar o *Hide Delegate* 2.3.2 nos restantes métodos.

REFUSED BEQUEST

Este *smell* acontece quando uma classe herda de outra classe mas não usa nenhum método dela. Neste caso fica a questão: "Para quê usar hierarquias se não usufrui dela?". Acontece que a hierarquia está errada, dessa forma o programador deve criar uma nova subclasse da classe mãe e usar o *Push Down Method* C e o *Push Down Field* C para transferir os métodos não usados para lá mantendo apenas na classe mãe os métodos comuns.

COMMENTS

Comentários no código não devem ser considerados de *smell* a não ser que o código tenha sofrido *refactorings*. Vejamos, um programador acrescenta comentários para explicar o código ou seja o código não é 100% legível. Se o código for corrigido então já não são precisos comentários porque já é legível. Contudo pode acontecer que mesmo assim ainda sejam precisos comentários para explicar blocos de código, nesses casos deve-se usar o *Extract Method* 2.3.2, se isso não chegar então deve ser aplicado o *Rename Method* C para alterar a assinatura do mesmo.

C

REFACTORINGS DE MARTIN FOWLER

Neste anexo serão apresentados os restantes *Refactorings* de Fowler (2002).

REPLACE TEMP WITH QUERY

Utiliza-se este *refactoring* quando no código se utiliza uma variável temporária para guardar uma o resultado de uma expressão. O que se deve fazer é substituir a variável por um método e todos os sítios onde ela é chamada pelo nome do método. Assim essa variável pode ser utilizada em mais que um método.

```
bool estadoFogao;

if temperatuda > 250
  estadoFogao = False;
else
  estadoFogao = True;
```

Listing C.1: Antes do Replace Temp with a Query

```
void Fogao(bool estado){
  estadoFogao = estado;
};

if temperatuda > 250
  Fogao(False);
else
  Fogao(True);
```

Listing C.2: Depois do Replace Temp with a Query

REPLACE METHOD WITH METHOD OBJECT

Utiliza-se o *Replace Method with Method Object* quando um método é muito longo e usa enumeras variáveis locais que impossibilitam a aplicação dos *Extract Method* 2.3.2. A solução passa por transformar essas variáveis num objeto dessa método, dessa forma pode utilizar esse objeto também nos outros métodos.

```
void fazAlgoBastanteComplexo{
  int fazer_1;
  int fazer_2;
  int fazer_3;
  int fazer_4;
  char nome_1;
  bool estado_3;
  (...)
}
```

Listing C.3: Antes do Replace Method with Method Object

```
obj estado(){
  int fazer_1 = Calcular(1);
  int fazer_2 = Calcular(2);
  int fazer_3 = Calcular(3);
  int fazer_4 = Calcular(4);
  char nome_1 = ``Actividade''
  bool estado_3 = true;
  (...)
}

void fazAlgoBastanteComplexo{
  obj estado_coisas = estado();
}
```

Listing C.4: Depois do Replace Method with Method Object

SUBSTITUTE ALGORITHM

Utiliza-se este *refactoring* quando o programador pretende substituir um algoritmo por outro mais simples. Para isso substitui o corpo do método pelo novo algoritmo.

```
String fazAlgoBastanteComplexo( String [] total_coisas{
  for (int i = 0; i < total_coisas.length; i++) {
```

```
    if (total_coisas[i].equals ("Arrumar o Casa")){
      return "Arrumar a Casa";
    }
    if (total_coisas[i].equals ("Lavar o Carro")){
      return "Lavar o Carro";
    }
    if (total_coisas[i].equals ("Regar as Plantas")){
      return "Regar as Plantas";
    }
  }
    return "Ir descansar!";
}
```

Listing C.5: Antes do Substitute Algorithm

```
String fazAlgoBastanteComplexo( String [] total_coisas{
  List coisas = Arrays.asList(new String[] {"Arrumar a Casa", "Lavar o Carro", "
      Regar as Plantas"});
  for (int i=0; i<total_coisas.length; i++)
    if (coisas.contains(total_coisas[i])) return total_coisas[i];
  return "Ir Descansar";
}
```

Listing C.6: Depois do Substitute Algorithm

INTRODUCE FOREIGN METHOD

Utilizador quando temos duas classes: uma a fazer de servidor e outra de cliente, queremos introduzir um novo método à classe servidor sem a alterar. Para isso criamos o novo método na classe cliente com uma instância da classe servidor nos seus argumentos.

```
Server servidor = new Server(utilizador.getName, utilizador.getIp, utilizador.
    getPort);
```

Listing C.7: Antes do Introduce Foreign Method

```
Server servidor = novoServidor(utilizador);

private static Server novoServidor(Server arg){
  return new Server (arg.getName, arg.getIp, arg.getPort);
}
```

Listing C.8: Depois do Introduce Foreign Method

INTRODUCE LOCAL EXTENSION

Quando a classe Servidor que está a ser usada necessita de métodos adicionais e não pode ser alterada, neste caso cria-se uma nova classe com os métodos adicionais e transforma-se numa subclasse ou extensão dessa classe. (Fig. 42)

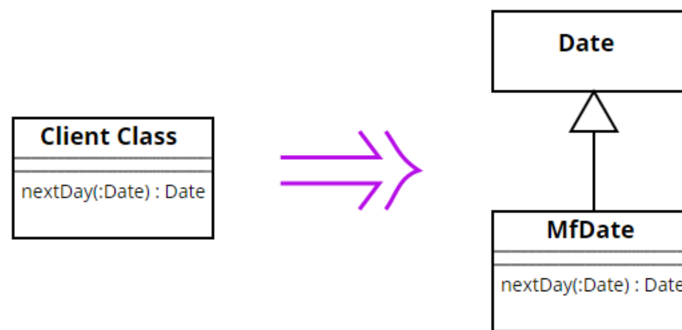


Fig. 42.: Figura Refactoring Introduce Local Extension.

REPLACE DATA VALUE WITH OBJECT

Utiliza-se quando temos dados que necessitam de informação ou comportamento adicional. A solução passa por transformar esses dados num objeto. (Fig. 43)

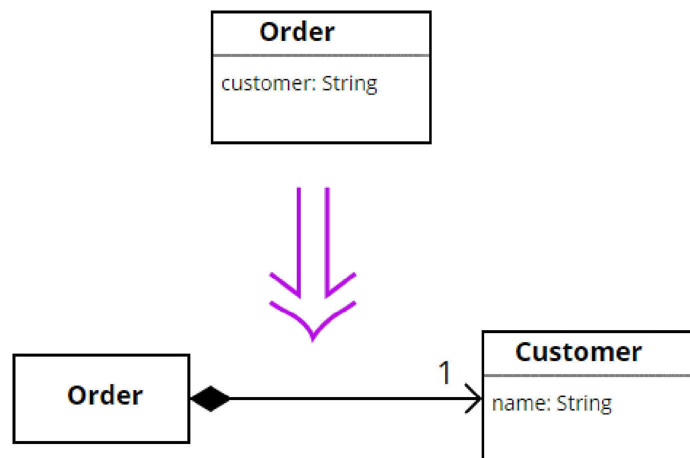


Fig. 43.: Figura Refactoring Replace Data Value with Object.

REPLACE ARRAY WITH OBJECT

Quando se tem um array em que cada posição representa coisas diferentes, o melhor é transformar esse array num objeto e para cada posição do array criar uma variável com um nome percetivo para aquilo que se quer guardar.

```
String[] cidade = new String[4];
cidades[0] = ``Viseu'';
cidades[1] = ``Manuel'';
cidades[2] = 324;
cidades[3] = 817263;
```

Listing C.9: Antes do Replace Array with Object

```
Cidade viseu = new Cidade();

viseu.setName(``Viseu'');
viseu.setPresidente(``Manuel'');
viseu.setTotalRuas(324);
viseu.setTotalHabitantes(817263);
```

Listing C.10: Depois do Replace Array with Object

DUPLICATE OBSERVER DATA

Aplica-se quando se tem um subconjunto de dados apenas disponível no controlador GUI e um subconjunto de métodos que lhes querem aceder. Para isso deve-se copiar o subconjunto de dados para um objeto e aplicar uma classe *Observer* para sincronizar as duas partes. (Fig. 44)

ENCAPSULATE FIELD

Utilizado quando se tem um campo público e se quer transformar num privado que apenas seja alterado a partir de alguns métodos.

```
public String nome;
```

Listing C.11: Antes do Encapsulate Field

```
private String nome;
```

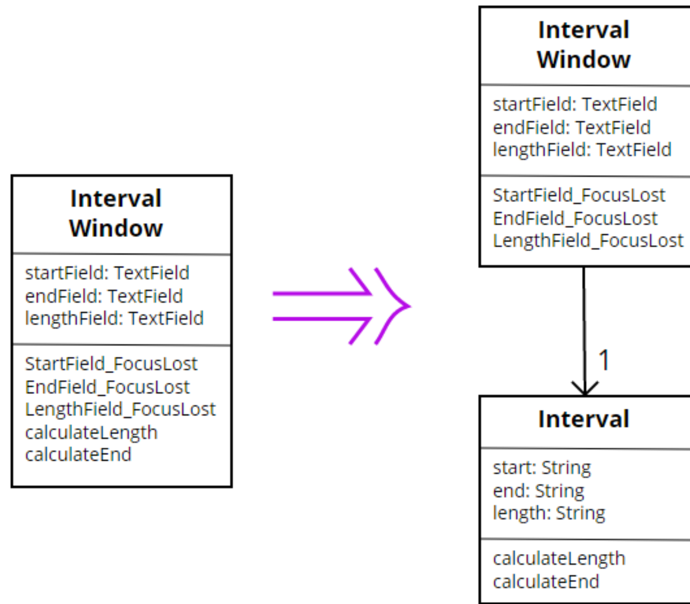


Fig. 44.: Figura Refactoring Duplicate Observer Data.

```
public String getNome() {return nome;}
public void setNome(String name) {nome = name; }
```

Listing C.12: Depois do Encapsulate Field

ENCAPSULATE COLLECTION

Quando um método retorna um conjunto de dados deve ser alterado para retornar um método de leitura e ao mesmo tempo criar-lhe métodos para adicionar e remover os dados do conjunto. (Fig. 45)

Fig. 45.: Figura Refactoring Encapsulate Collection.

REPLACE TYPE CODE WITH CLASS

Quando uma classe contem enumeradores que não afetam o seu comportamento, esses devem ser transformados numa nova classe. (Fig. 46)

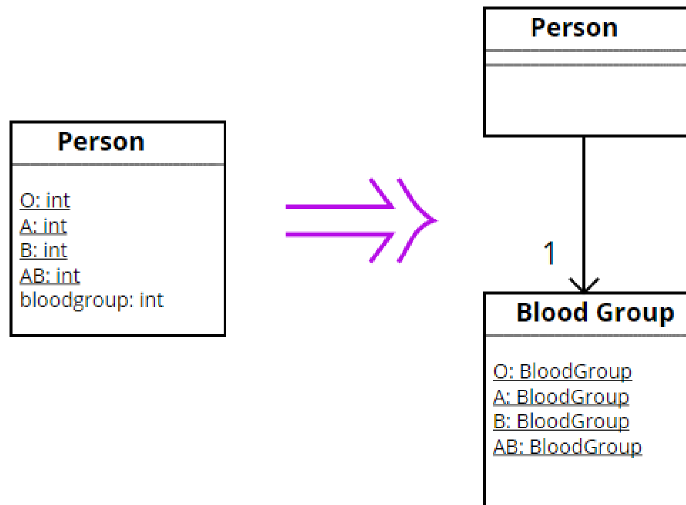


Fig. 46.: Figura Refactoring Replace Type Code with Class.

REPLACE TYPE CODE WITH SUBCLASSES

Quando existe código que altera o comportamento da classe esse código deve ser transformado numa subclasse dessa classe. (Fig. 47)

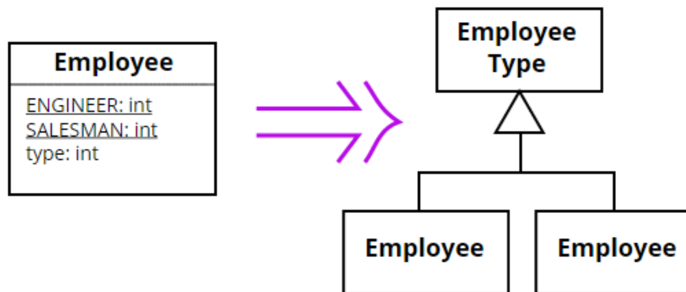


Fig. 47.: Figura Refactoring Replace Type Code with Subclasse.

REPLACE TYPE CODE WITH STATE/STRATEGY

Ao contrário o *Replace Type Code with Subclasses* C este *refactoring* é utilizado quando se tem um conjunto de dados que altera o comportamento da classe mas não pode ser transformado numa subclasse. Nesta situação substitui-se esse código por um objeto estado. (Fig. 48)

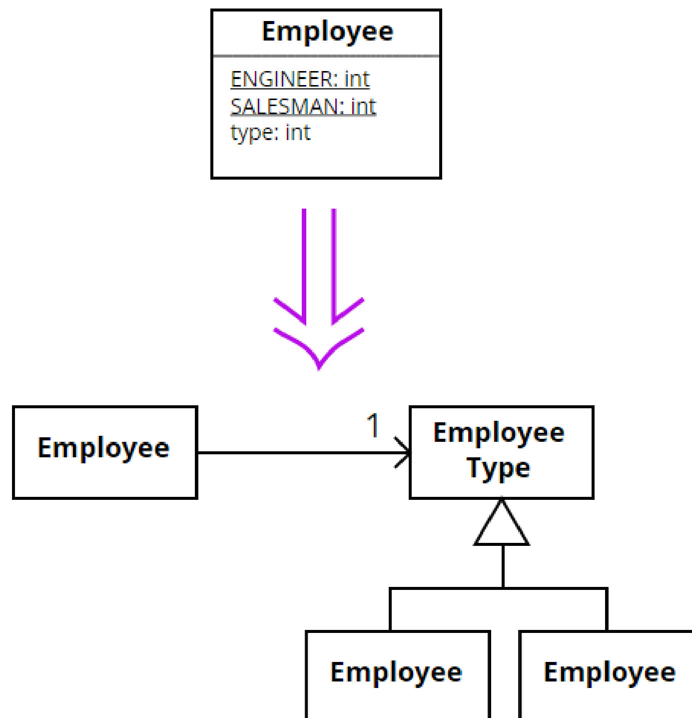


Fig. 48.: Figura Refactoring Replace Type Code with State/Strategy.

DECOMPOSE CONDITIONAL

Quando existe uma condição (if-else) de difícil compreensão. Nessa situação deve-se simplificar recorrendo à criação de métodos auxiliares.

```
if (data.antes(INICIO_NATAL) || (data.depois(FIM_NATAL)))
  return ``Continuar a trabalhar.'';
else
  return ``Preparar Natal.'';
```

Listing C.13: Antes do Decompose Conditional

```
void Natal(data inicio, data fim){
  if (inicio > 20 && fim < 26)
    return true;
  return false;
};

if( Natal(INICIO_NATAL, FIM_NATAL) )
  return ``Continuar a trabalhar.'';
else
```

```
return ``Preparar Natal.'';
```

Listing C.14: Depois do Decompose Conditional

REPLACE CONDITIONAL WITH POLYMORPHISM

Ocorre quando se tem uma condição que depende de um objeto para alterar o comportamento. Por exemplo: uma condição do tipo *switch-case* que verifica a nacionalidade de uma pessoa e consoante a nacionalidade atribui-lhe um salário. A condição apenas aceita 3 nacionalidades. Essa condição pode ser substituída por uma classe com 3 subclasses: a classe Cidadao com as subclasses Europeu, Americano e Asiatico. Dessa forma simplifica-se a condição.

INTRODUCE NULL OBJECT

Quando se questiona várias vezes se o valor de um objeto está a null, esse valor deve ser substituído por um objeto null.

RENAME METHOD

Quando o nome de um método não ajuda na compreensão do seu objetivo esse nome deve ser alterado. (Fig. 49)

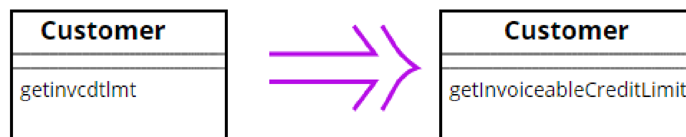


Fig. 49.: Figura Refactoring Rename Method.

REMOVE PARAMETER

Quando um método tem um ou mais parâmetros que já utiliza esses parâmetros devem ser removidos. (Fig. 50)

REPLACE PARAMETER WITH EXPLICIT METHODS

Utilizado quando um método executa várias condições aos parâmetros. Essas condições devem ser retiradas do método e transformadas em métodos auxiliares.

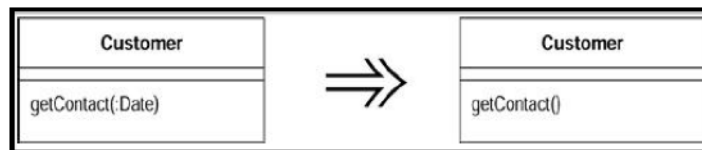


Fig. 50.: Figura Refactoring Rename Parameter.

```
bool verificaPessoa (String nome, int contribuinte){
  if ( nome.equals(``Jose'') && contribuinte == 1238481)
    return true;
  false;
}
```

Listing C.15: Antes do Replace Parameter with Explicit Methods

```
bool confirmarNome (String nome){
  if nome.equals(``Jose'');
    return true;
  false;
}

bool confirmarContribuinte (int contribuinte){
  if contribuinte == 1238481;
    return true;
  false;
}

bool verificaPessoa (String nome, int contribuinte){
  if  confirmarNome(nome) && confirmarContribuinte(contribuinte)
    return true;
  false;
}
```

Listing C.16: Depois do Replace Parameter with Explicit Methods

PRESERVE WHOLE OBJECT

Quando são usados e passados por parâmetros vários valores de um objeto nessa situação o que se deve fazer é enviar todo o objeto.

```
Pessoa p = new Pessoa(``Jose'', 16, 1.90);
```

```
int idade = p.getIdade();
float altura = p.getAltura();

float pesoIdeal = PesoCerto(idade, altura);
```

Listing C.17: Antes do Preserve Whole Object

```
Pessoa p = new Pessoa(``Jose'', 16, 1.90);

float pesoIdeal = PesoCerto(p);
```

Listing C.18: Depois do Preserve Whole Object

REMOVE SETTING METHOD

Quando o valor de uma variável é atribuído unicamente em tempo de criação e nunca mais alterado então não devem existir métodos que o alterem. (Fig. 51)

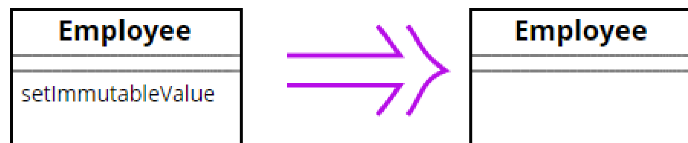


Fig. 51.: Figura Refactoring Remove Setting Method.

REPLACE PARAMETER WITH METHOD

Este *refactoring* é utilizado quando uma variável recebe o valor retornado por um método e o passa nos parâmetros de outro método. Nesta situação, como a variável não é alterada desde o momento que recebe o valor do primeiro método até ao momento em que o passa para o segundo método, em vez de se passar a variável nos parâmetros deve ser chamado o método.

```
int idade = 17;

float peso = Pessoa.GetPeso();

double pesoIdeal = PesoCerto(idade, peso);
```

Listing C.19: Antes do Replace Parameter with Method

```
int idade = 17;

double pesoIdeal = PesoCerto(idade, Pessoa.GetPeso());
```

Listing C.20: Depois do Replace Parameter with Method

INTRODUCE PARAMETER OBJECT

Utilizado quando existe um grupo de parâmetros que andam sempre juntos em qualquer sítio. Nesta situação o que se deve fazer é agrupar esse parâmetros e transforma-los num objeto.

```
class Corrida{
  int box;
  int totalRodas;
  string cor;
  string condutor;
  float premio;
}
```

Listing C.21: Antes do Introduce Parameter Object

```
class Participante{
  int totalRodas;
  string cor;
  string condutor;
}

class Corrida{
  int box;
  Participante p = new Participante();
  float premio;
}
```

Listing C.22: Depois do Introduce Parameter Object

HIDE METHOD

Quando um método é apenas utilizado pela própria classe esse método deve ser único e exclusivamente privado.

PULL UP FIELD

Quando duas subclasses da mesma classe têm a mesma variável essa variável deve ser transferida para a classe mãe. (Fig. 52)

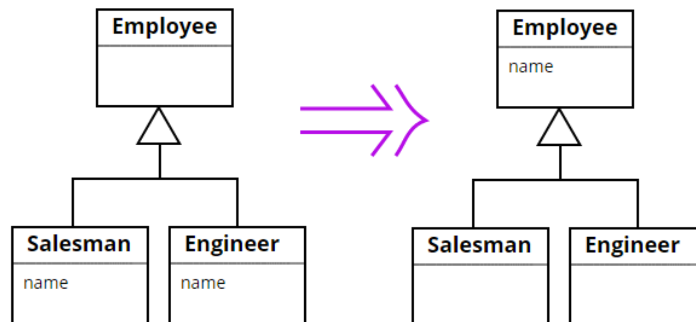


Fig. 52.: Figura Refactoring Pull Up Field.

PUSH DOWN METHOD

Quando um método de uma classe está acessível a todas as suas subclasses mas, apenas uma usufruí dele, então esse método deve ser transferido para a subclasse. (Fig. 53)

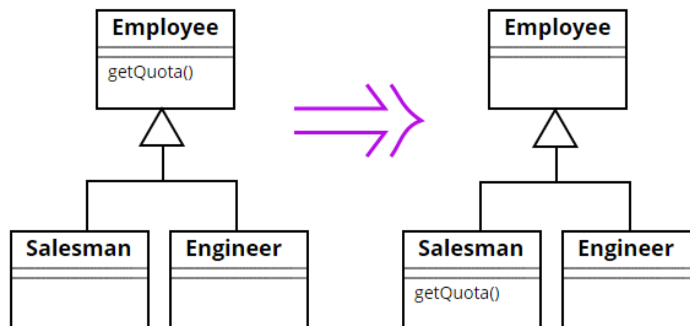


Fig. 53.: Figura Refactoring Push Down Method.

PUSH DOWN FIELD

Ao contrário do *Pull Up Field* C este *refactoring* aconselha que: quando a variável de uma classe disponível a todas as subclasses apenas é utilizada por uma subclasse então essa variável deve ser transferida para lá. (Fig. 54)

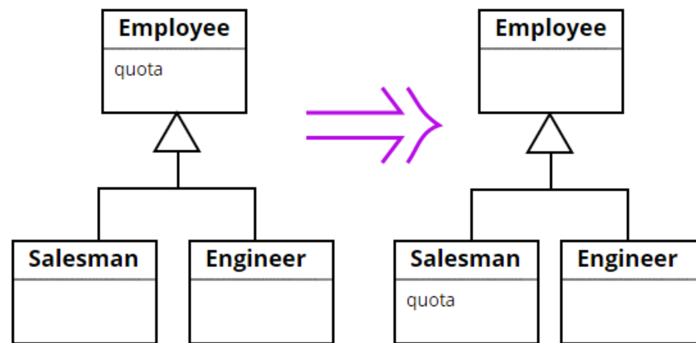


Fig. 54.: Figura Refactoring Push Down Field.

EXTRACT SUBCLASS

Este *refactoring* é utilizado quando existem atributos numa classe usados apenas em algumas situações. Neste caso esses atributos devem ser reagrupados e transformados numa subclasse. (Fig. 55)

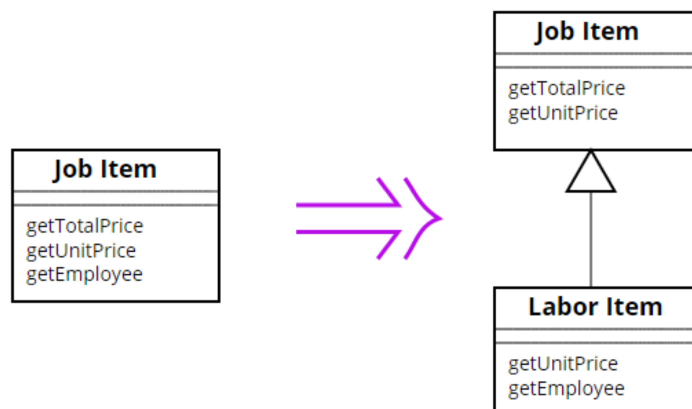


Fig. 55.: Figura Refactoring Extract Subclass.

EXTRACT SUPERCLASS

Quando duas classes têm métodos semelhantes o que se deve fazer é criar uma superclasse comum a ambas e transferir para lá esses métodos. (Fig. 56)

EXTRACT INTERFACE

Utiliza-se este *refactoring* quando numa aplicação vários clientes usam o mesmo conjunto de métodos da interface de uma classe ou existem duas classes que tenham partes das suas interfaces em comum.

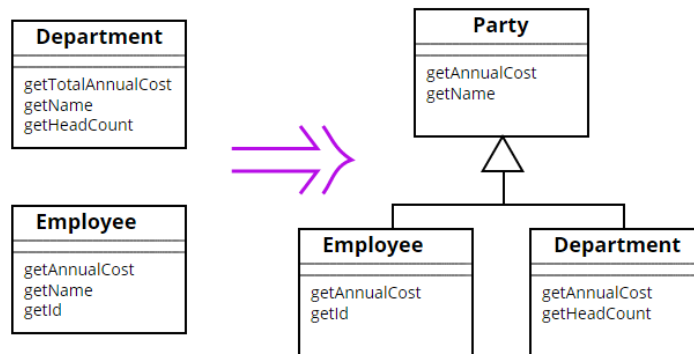


Fig. 56.: Figura Refactoring Extract Superclass.

Nestes casos devemos extrair esses conjunto de métodos e transforma-los numa interface comum às classes. (Fig. 57)

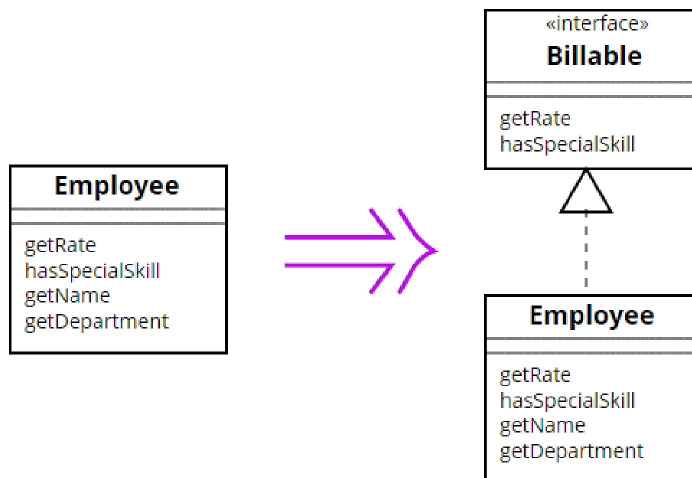


Fig. 57.: Figura Refactoring Extract Interface.

COLLAPS HIERARCHY

Este *refactoring* é utilizado quando uma superclasse e uma subclasse não são diferentes, nesse caso deve-se unir as duas. (Fig. 58)

FORM TEMPLATE METHOD

Utiliza-se o *Form Template Method* quando duas subclasses têm métodos com a mesma assinatura a procederem da mesma forma com ligeiras alterações. O que se deve fazer é subdividir esses métodos em partes comuns e não comuns. As partes não comuns são transformadas em métodos com o mesmo

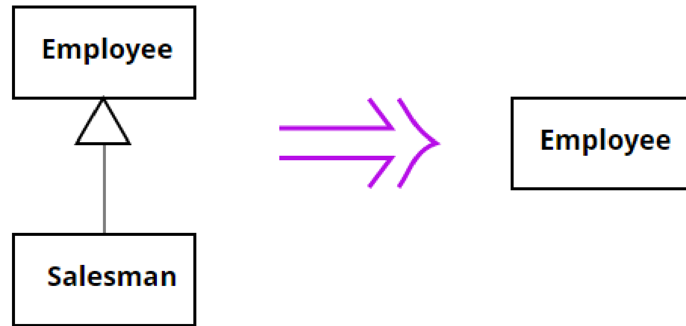


Fig. 58.: Figura Refactoring Collaps Hierarchy.

nome e deixadas nas subclasses, a parte comum é transformada num método e transferida para a classe assim como as assinaturas dos métodos não comuns. (Fig. 59)

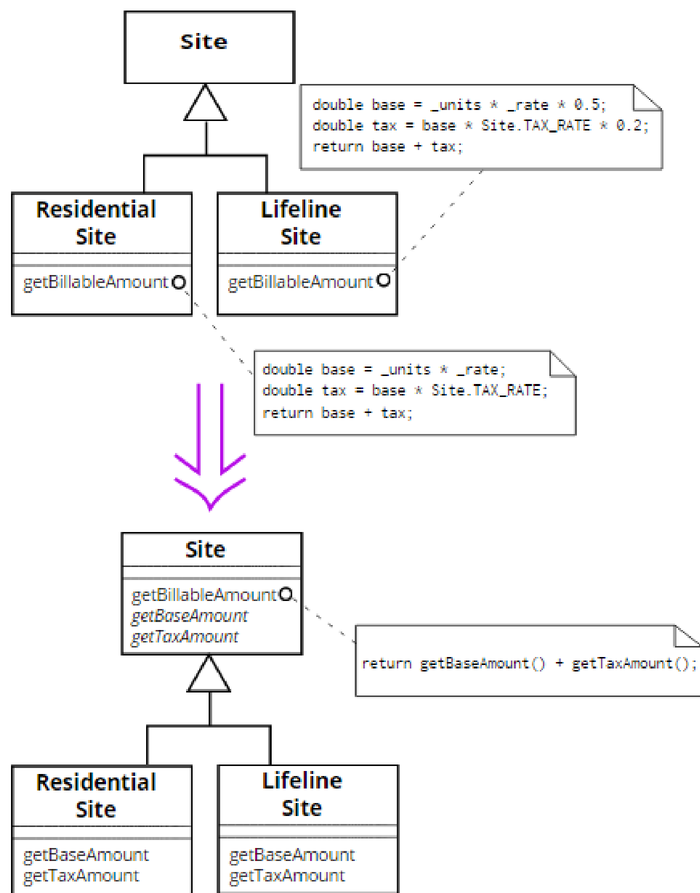


Fig. 59.: Figura Refactoring Form Template Method.