

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE DE MEDEIROS SCHMIDT

Instatiating the Page Object Pattern in Desktop Applications

Monografia apresentada como requisito parcial
para a obtenção do grau de Bacharel em Ciência
da Computação

Trabalho realizado na Technische Universität
Berlin dentro do acordo de dupla diplomação
UFRGS - TU Berlin.

Orientador brasileiro: Prof. Dr. Ingrid Oliveira de
Nunes

Orientador alemão: Prof. Dr. rer. nat. Thomas
Karbe

Porto Alegre
2016

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

de Medeiros Schmidt, Felipe

Instatiating the Page Object Pattern in Desktop Applications / Felipe de Medeiros Schmidt. – Porto Alegre: CIC da UFRGS, 2016.

53 f.: il.

Trabalho de conclusão (graduação) – Universidade Federal do Rio Grande do Sul. Curso de Ciência da Computação, Porto Alegre, BR–Brasil, 2016. Orientador: Ingrid Oliveira de Nunes.

1. Padrão-Page-Object. 2. Aplicações-Desktop. 3. Teste-Headless. 4. View-Object. I. Oliveira de Nunes, Ingrid. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitor: Prof. Carlos Alexandre Netto

Vice-Reitor: Prof. Rui Vicente Oppermann

Pró-Reitor de Graduação: Prof. Vladimir Pinheiro do Nascimento

Diretor do Instituto de Informática: Prof. Luis da Cunha Lamb

Coordenador do Curso de Ciência de Computação: Prof. Carlos Arthur Lang Lisbôa

Bibliotecária-chefe do Instituto de Informática: Beatriz Regina Bastos Haro

AGRADECIMENTOS

A realização deste trabalho tornou-se viável graças ao apoio de pessoas queridas que sempre estiveram ao meu lado e que serei eternamente grato. Agradeço profundamente aos meus pais, meus irmãos, minha namorada, meus amigos, chefes, professores e orientadores por todo o carinho, incentivo e atenção que me deram ao longo de toda minha trajetória. A todos aqueles que cruzaram o meu caminho nesse mundo, obrigado pela experiência e pelo aprendizado. Lembrarei com carinho de cada um. Graças a vocês, meu desejo de vencer inúmeros desafios, conquistar novos objetivos e me superar se tornam cada vez maiores.

RESUMO

Teste de software está se tornando cada vez mais importante no desenvolvimento de software. Aplicações web têm o desafio de testar aplicações onde interfaces de usuário (UIs) são definidas em linguagens específicas (por exemplo, JSP e ASP), o que dificulta o seu teste. Padrões (por exemplo, o padrão Page Object) e tecnologias (por exemplo, Selenium) fornecem suporte para lidar com esta questão. Embora as aplicações web sejam populares, aplicações desktop ainda tem um papel crucial na indústria de software. Novas bibliotecas e frameworks são baseadas em outras linguagens, como o XML, para definição de interface. Isso traz o problema de aplicações web para as aplicações desktop também. Assim, este trabalho explora como instanciar o padrão objeto Page (POP) no contexto de aplicações POP. Page Objects são referidos como View Objects (VOs) neste caso. Um dos principais benefícios deste método é em relação ao modo de teste. Ao seguir os passos para implementar os VOs, preparamos nossos testes para executar no modo headless. Isso significa que a aplicação desktop não necessita ser instanciado para ser testado. Todas as operações são simuladas e feitas sem exibir a interface de usuário, simulando variáveis importantes da UI para ser usado dentro de VOs.

Palavras-chave: Padrão-Page-Object. Aplicações-Desktop. Teste-Headless. View-Object.

ABSTRACT

Software testing is becoming increasingly important in software development. Web applications have the challenge of testing applications where user interfaces (UIs) are specified in specific languages (e.g. JSP, ASP), which complicates its test. Patterns (e.g. Page Object Pattern) and technologies (e.g. Selenium) provide support to deal with this issue. Although web applications are popular, desktop applications still have a crucial role in the software industry. New libraries and frameworks are based on other languages, such as XML, for interface definition. It brings the web application's problem to the desktop applications as well. Thus, this work explores how to instantiate the Page Object Pattern (POP) in the context of desktop applications. POs are referred to as View Objects (VOs) in this case. One of the main benefits of this method is regarding to the test mode. When following the steps to implement view objects, we prepare our tests to run in headless mode. It means that our desktop application interface does not need to be instantiated to be tested. All operations are simulated and done without displaying the UI, mocking important variables of the UI to be used within VOs.

Keywords: Headless-Testing, Desktop-Applications, Page-Object-Pattern, View-Object.

RESUMO ESTENDIDO

Este é um resumo estendido em português para a Universidade Federal do Rio Grande do Sul. O trabalho de conclusão original, em inglês, foi apresentado na Technische Universität Berlin através do programa de dupla diplomação UNIBRAL II entre as duas universidades.

Introdução

Teste de software tem ganhado um papel cada vez mais importante no desenvolvimento de software. Para facilitar esta tarefa nada fácil, diferentes abordagens são cada vez mais utilizadas, tais como Test-Driven Development (TDD) (Janzen and Saiedian, 2005), Behaviour-Driven Development (BDD) (Solís and Wang, 2011), testes *headless* (Johansen, 2010) e outras técnicas. Abordagens propostas e tecnologias desenvolvidas muitas vezes focam em aplicações web e, assim, eles se tornam mais populares. Aplicações web têm o desafio de testar aplicações onde interfaces de usuário (UIs) são definidas em linguagens específicas (por exemplo, JSP, ASP), o que dificulta o seu teste. Esta dificuldade surge porque a integração entre ambas as linguagens tem que ser feito corretamente e os testes são, então, mais difíceis de ser preparado. Portanto, padrões (e.g. Padrão Page Object) e tecnologias (e.g. Selenium) fornecem suporte para lidar com esta questão.

Embora aplicações web sejam mais populares, aplicações desktop ainda tem um papel crucial na indústria de software. Antigas bibliotecas de interfaces de usuário, tais como Java Swing, foram puramente baseadas em linguagens de programação utilizadas para implementar a aplicação, e isso torna mais fácil a execução de testes automatizados. No entanto, novas bibliotecas e frameworks são baseados em outras linguagens, como o XML, para definição de interface. Assim, precisamos lidar com duas linguagens diferentes e integrá-los. Isso traz o problema de aplicações web para as aplicações desktop também.

Assim, este trabalho explora como instanciar o padrão Page Object (POP) citep yu2015incremental no contexto de aplicações desktop. O POP modela página web em objetos, que contêm funções específicas de uma área da página e agem de forma independente. Os Page Objects (PO) são uma camada intermediária entre as páginas web e os códigos de teste. Como resultado, propomos um método que fornece orientação para os desenvolvedores para instanciar esse padrão em aplicações desktop. Neste caso, as POs

são referidos como *View Objects* (VOs). Um dos principais benefícios deste método é em relação ao modo de teste. Ao seguir os passos para implementar os VOs, preparamos nossos testes a serem executados no modo headless. Isso significa que a interface da aplicação desktop não precisa ser instanciada para ser testada. Todas as operações são simuladas e feitas sem exibir a interface do usuário, simulando variáveis importantes da interface do usuário para ser usado dentro dos VOs.

Este trabalho começa dando uma base em conceitos relevantes que são necessários para compreender este trabalho (Capítulo 2), tais como técnicas de teste e explicações sobre as tecnologias adotadas. Depois disso, o método para a instanciação do POP é dada com um exemplo (Capítulo 3). A parte de avaliação é realizada para uma aplicação evoluída do exemplo anterior e como aplicar este método neste caso específico (Capítulo 4). Um capítulo discussão é dirigida para abordar algumas idéias e considerações (Capítulo 5). Para terminar, o Capítulo 6 apresenta conclusões finais.

Revisão da Literatura

Hoje em dia, erros de software podem ser responsáveis pelos custos de tempo e dinheiro. Embora não seja possível remover todos os erros de uma aplicação, existem formas de testes que permitam reduzir consideravelmente a quantidade de erros e os erros mais graves. Visando a melhoria da qualidade e poupar dinheiro (e também, indiretamente, economizando tempo), TDD (Janzen and Saiedian, 2005) e BDD (Solís and Wang, 2011) são técnicas de testes adequados para prever e identificar erros - na fase de desenvolvimento - que pode acontecer quando se executa a aplicação desenvolvida. Concentrando-se em bons resultados de testes desenvolvidos, há um padrão proposto por Bill Wake chamada 3A Padrão¹. O principal objetivo deste padrão é estruturar os testes adequadamente, de uma forma podemos identificar aspectos importantes que precisamos para testar o aplicativo. Basicamente, os testes devem *Arrange*, *Act* e *Assert*. Segundo ele, *arrange* significa criar objetos que serão testadas. *Act* é usar esses objetos e dar-lhes ordens. Fazendo asserts podemos garantir algumas circunstâncias sobre o objeto. Assim, com o uso do padrão 3A, casos de teste consistentes pode ser escrito.

Técnicas como BDD, quando usadas para testar uma aplicação desenvolvida, podem ter extrema relevância para se encontrar possíveis erros e ser extremamente poderoso. Com BDD aplica-se a maneira de escrever testes primeiro, mas concentra-se em testes que

¹Ver <http://xp123.com/articles/3a-arrange-act-assert/>

descrevem o comportamento. Dado que a abordagem BDD ainda não é clara e diferentes autores têm distintas opiniões sobre a sua definição. Principais características do BDD foram identificados por Solís and Wang (2011, p. 02): linguagem Ubíqua, processo de decomposição iterativo, Descrição de histórias de usuário e templates de cenários, testes de aceitação automatizada com regras de mapeamento.

A palavra *headless* significa simplesmente que alguma coisa (por exemplo, aplicação, browser, operação) funciona sem uma interface gráfica do usuário. O termo *teste headless* está associado com o desafio de testar uma interface de usuário sem exibi-la. A idéia principal do teste headless de uma aplicação é que os testes podem ser mais consistentes. Isso significa que, os testes não verificam se um determinado botão foi realmente clicado, mas qual é o significado real e ação desse clique. Por conseguinte, a economia de tempo é uma vantagem, porque a interface não é mostrada, e nós não precisamos esperar por cliques ou seleções eventos e suas respostas.

Visto que a comunicação entre a interface de usuário e a aplicação em si é bastante direta, isso torna os testes mais acoplados e dependentes da interface de usuário. Sendo esse problema mais complexo de se resolver, padrões podem facilitar essa tarefa. O padrão Page Object (POP) (Leotta et al., 2013) é um padrão de design que tem sido amplamente utilizado e o termo foi dissipado por *Selenium*. Este padrão foi projetado para criar testes automatizados para navegadores web. Ele pode ser melhor aplicado quando um aplicativo tem muitas páginas ou muitos estados. A idéia desse padrão é que página da web é modelada em objetos, que podem ser partes específicas da página da web que podem agir independentemente. Isto significa que todas as funções que podem ser executadas dentro de um objecto são encapsulados para algo, chamado Page Object. Desta forma, todas as alterações necessárias na página da web (UI) não afetarão as classes de teste, mas apenas o objeto da página referida no essa funcionalidade (Figura 2.1).

Testando Aplicações Desktop com View Objects

Pode-se separar o método para a criação do VOs em quatro etapas, a fim de alcançar o nosso resultado desejado, os quais são descritos como se segue.

Desenvolver a Aplicação Desktop A interface do usuário e a lógica da aplicação deve ser dividido em diferentes módulos. Esta separação é importante a ser feito por causa do passo em relação à implementação dos objetos de exibição, quando nos

preocupamos com o mock (simulação) dos elementos da lógica da aplicação e os elementos da interface gráfica para executar os testes headless.

Escolher os View Objects A escolha dos VOs precisa ser observado, tendo em consideração o modo como os elementos da View interagem uns com os outros. Assim, o expert deve navegar através da(s) janela(s) da aplicação e considerar quais VOs podem ser criados para encapsular funcionalidades específicas da aplicação.

Implementar os View Objects A seleção de variáveis e operações da aplicação relevantes são usados como um mock para implementar as classes VOs. Nós nos preocupamos com os elementos significativos da interface e da parte lógica, ou seja, elementos que têm um papel importante na interação com o usuário. Por exemplo, em relação aos elementos GUI, o utilizador pode escrever nos campos de texto e clicar em botões numa determinada janela, de modo que o VO deve conter uma variável com relação aos campos de texto e outras para os botões. Em relação à parte da lógica, se temos um banco de dados, devemos fazer o mesmo, a criação de variáveis em relação a isso também. Além disso, é preciso considerar que são as possíveis operações naquela janela, por exemplo, Se em uma determinada janela, o usuário é capaz de *selecionar* uma entrada na tabela, *clicar* em um botão, *escrever* em um campo de texto e assim por diante. Em seguida, os métodos associados a estas operações vão lidar com as variáveis mockadas criadas antes. É por isso que o primeiro passo é crucial (em relação ao desenvolvimento adequado da aplicação).

Testar O objetivo é usar os métodos criados dentro dos VOs a fim de testar a nossa aplicação. Assim, os testes são escritos instanciando os VOs e usando-os.

Avaliação e Discussão

O Source code 3.3.1 é um exemplo de um View Object implementado para testar uma aplicação dada como exemplo.

Com o desenvolvimento do HTML4, o HTML5 (que não é padrão até o momento) foi lançado e suporta muitas outras features que não eram suportadas antes, tais como mídia e JavaScript. Assim, áudio, vídeo e gráficos vectoriais (2D e 3D interactivo) são agora integrados para esta tecnologia e capazes de ser reproduzidos e armazenados na aplicação.

Portanto, podemos notar que, com HTML5, as características de desenho de páginas web se tornou muito mais parecida com recursos para criar interfaces de usuário desktop, mas ainda precisa ser considerado qual é a intenção do conteúdo que é mostrado sobre a aplicação, porque VOs lida com janelas e POs com páginas da web, ou seja, se o conteúdo seria melhor apresentado em uma janela ou sobre navegadores da Web, tendo em conta as necessidades de cada um.

Também é importante destacar que as páginas web são muitas vezes páginas completas. Então, depois de interagir com uma página, temos uma completamente nova. Por outro lado, interfaces de usuário podem normalmente serem separadas em diferentes partes, tais como sliced windows, tabbed panes e outros elementos. Assim, as ações do usuário podem alterar uma ou mais dessas partes e deixar os outros sem quaisquer alterações. Com o desenvolvimento das páginas web, elas estão se tornando mais modernas neste aspecto de ter as mesmas características que interfaces de usuário standalones. Sendo assim, os VOs podem ser úteis para eles também.

Um papel importante com o uso de VOs para testar a nossa aplicação é que podemos alcançar naturalmente *teste headless*. Isto é possível porque nós mockamos as variáveis da interface do usuário e implementamos as mesmas operações da interface do usuário, simulando essas operações. Em seguida, cada operação é feita mais rapidamente e faz a interface do usuário não precisar ser instanciada. Outra vantagem é em relação aos testes: se é preciso modificar a implementação da interface do usuário, os testes permanecerão os mesmos, porque o VOs encapsulam os métodos de interface do usuário. Assim, apenas o VO deve ser modificado. Temos importantes vantagens ao usar VOs para testar aplicações desktop, como a organização do código fonte, bem-estrutura e os testes são de fácil manutenção.

Um dos pontos negativos do uso de VO é que, por exemplo, uma classe de implementação poderia tornar-se uma enorme quantidade de if e elses para cobrir todos os botões (ver Source Code 3.3.1c). Assim, a complexidade do código-fonte pode aumentar e a organização pode ser um problema.

Conclusão

Sendo software testando uma área que está se tornando crucial no campo de desenvolvimento de software, a automação de testes a fim de economizar tempo é essencial. Diferentes abordagens e ferramentas foram propostas para aplicações web. O POP veio

para facilitar a parte de testes da aplicação, porque a aplicação é geralmente definida em linguagens específicas, o que complica o teste. Por esta razão, VOs são propostos como uma instância do POP. A principal diferença entre os dois métodos é basicamente que um visa navegadores da web (POs) e outro em aplicações standalone (VOs). Com o desenvolvimento da tecnologia HTML, as páginas da web são cada vez mais idênticas a aplicações standalone. Sem a utilização destes métodos, a arquitetura da aplicação seria altamente acoplada, aumentando a complexidade da aplicação, devido à ligação direta entre as classes de teste e a interface de usuário. Assim, estes padrões ajudam a reduzir a complexidade da aplicação, criando uma outra camada que é responsável por encapsular os detalhes dos componentes da UI. Com a utilização de VOs também temos a vantagem de conseguir testes em modo headless. Isto é possível por causa do uso de métodos e variáveis mockadas que simulam operações de UI. Assim, a interface do usuário não é exibida e todas as operações ocorrem sem instanciá-la. Desenvolvendo a aplicação, nós precisamos estruturar e organizar os módulos, a fim de abstrair alguns conceitos da aplicação. Podemos perceber que o importante no desenvolvimento da aplicação é a separação da lógica da aplicação e interface do usuário. Assim, podemos manipular e aplicar corretamente o VOs. Uma limitação do método pode ser a alta complexidade de uma classe VO implementada dependendo do número de botões que podemos interagir.

Sendo o BDD uma técnica eficaz quando combinado com VOs, quatro características principais ajudam a construir casos de testes consistentes, sendo eles: uma linguagem ubíqua (os termos devem ser usados globalmente no projeto), processo de decomposição iterativa para levantamento de requisitos, descrição de texto simples com modelos história do usuário e do cenário (como um padrão para a criação das características de arquivo) e, finalmente, testes de aceitação automatizados com regras de mapeamento (ou seja, cenários sendo executado automaticamente e sendo mapeados para testar o código). Assim, no BDD, os testes são escritos em primeiro lugar, e que têm a descrição do comportamento do sistema, como um objetivo a ser alcançado.

Nós pudemos ver dois tipos diferentes de aplicação e nós trabalhamos em suas peculiaridades. Se é uma aplicação complexa, a delegação de suas ações para VOs menores têm de ser consideradas na modelagem e implementação do VOs. Portanto, por VOs poderem ser aplicados para aplicações desktop, poderíamos, então, identificá-los como uma instância para o POP.

Para um trabalho futuro, pensa-se em uma implementação de um plugin a fim de automatizar esse método para aplicações desktop.

```
public class InsertDataWindow {
    private String txtName = "txtName";
    private String txtLastName = "txtLastName";
    private String txtWeight = "txtWeight";
    private String txtHeight = "txtHeight";
    private static String BTN_ADD_AND_CONFIRM =
        "BTN_ADD_AND_CONFIRM";
    private static String BTN_SHOW_TABLE = "BTN_SHOW_TABLE";
```

a: Action elements

```
private ObservableList<Person> personDataToRetrieve;
private TablePersonActionController tpc;
private Person personToAdd;
```

b: Logic elements

```
private void printText(String element, String text){
    element = "";
    element = text;
}
private void click(String element){
    if(element == BTN_ADD_AND_CONFIRM){
        tpc.addNewPerson(personToAdd);
    }
    else if(element == BTN_SHOW_TABLE){
        personDataToRetrieve = tpc.getPersonData();
    }
}
public InsertDataWindow fillInInfo(Person person){
    printText(txtName, person.getFirstName());
    printText(txtLastName, person.getLastName());
    printText(txtHeight, String.valueOf(person.getHeight()));
    printText(txtWeight, String.valueOf(person.getWeight()));
    personToAdd = person;
    return this;
}
public TableResultsWindow AddPerson(){
    click(BTN_ADD_AND_CONFIRM);
    return new TableResultsWindow(tpc);
}
public TableResultsWindow showTableResults(){
    click(BTN_SHOW_TABLE);
    return new TableResultsWindow(personDataToRetrieve);
}
}
```

c: Methods implementation

```
trw = idw.showTableResults();
```

d: Small Test Example

Listing 0.1: InsertDataWindow view object implementation

LISTA DE FIGURAS

Figura 2.1	Illustration of the Page Object Pattern modules	26
Figura 2.2	Illustration of the Technische Universität Berlin Website Login Area.....	26
Figura 2.3	Illustration of the modules of a Web Application for automated testing.....	27
Figura 3.1	Illustration of the visualization of the Main Window view	31
Figura 3.2	Illustration of the visualization of the Table Window view	32
Figura 3.3	Illustration of the visualization of the Edit Window view	33
Figura 4.1	Illustration of the visualization evolved BMI application	41
Figura 4.2	Illustration of the visualization evolved BMI application adding a person...	41
Figura 4.3	Illustration of the visualization evolved BMI application editing a person...	42
Figura 4.4	Illustration of the visualization evolved BMI application saving a person ...	42

LISTA DE SOURCE CODES

2.1 POP Example: Login Method of TUB website	26
2.2 POP Example: Test for Login Method of TUB website.....	26
3.1 Writing Scenarios: InsertPerson feature	35
3.2 Step Definitions Class Implementation.....	37
3.3 Console response after running the feature.....	38
4.1 PersonInfoWindow Implementation	45
4.2 TableWindow Implementation	46

LISTA DE ABREVIATURAS E SIGLAS

UI	User Interface
GUI	Graphical User Interface
TDD	Test-Driven Development
ATDD	Acceptance Test-Driven Development
BDD	Behavior-Driven Development
IDE	Integrated Development Environment
POP	Page Object Pattern
PO	Page Object
VO	View Object
RIA	Rich Internet Application
BMI	Body Mass Index

SUMÁRIO

1 INTRODUCTION	17
2 BACKGROUND	19
2.1 Software Testing	19
2.1.1 Test-Driven Development	19
2.1.2 Behaviour-Driven Development	21
2.1.3 Headless Testing	24
2.2 Page Object Pattern	25
2.3 Adopted Technologies	27
2.3.1 JavaFX	28
2.3.2 Cucumber.....	29
3 TESTING DESKTOP APPLICATIONS WITH VIEW OBJECTS	30
3.1 Method Overview	30
3.2 Running Example	31
3.3 Method Steps	32
4 EVALUATION	40
4.1 The Evolved BMI Calculator Application	40
4.2 Develop Desktop Application	42
5 DISCUSSION	49
5.1 Web Page vs. Desktop View	49
5.2 Advantages of Using POP to Test Desktop Applications	50
5.3 Limitations of View Objects	50
6 CONCLUSION	51
REFERÊNCIAS	53

1 INTRODUCTION

Software testing is becoming increasingly important in software development. To make this difficult task easier, there are plenty of different approaches that are increasingly being used, such as Test-Driven Development (TDD) (Janzen and Saiedian, 2005), Behaviour-Driven Development (BDD) (Solís and Wang, 2011), headless testing (Johansen, 2010) and other techniques. Proposed approaches and developed technologies often focus on web applications, as they are more popular. Web applications have the challenge of testing applications where user interfaces (UIs) are specified in specific languages (e.g. JSP, ASP), which complicates its test. This difficulty appears because the integration between both languages has to be done properly and the tests are then harder to be prepared. Therefore, patterns (e.g. Page Object Pattern) and technologies (e.g. Selenium) provide support to deal with this issue.

Although web applications are popular, desktop applications still have a crucial role in the software industry. Former user interfaces libraries, such as Java Swing, were purely based on programming languages used to implement the application, and this made easier the execution of automated tests. However, new libraries and frameworks are based on other languages, such as XML, for interface definition. Thus, we need to handle with two different languages and integrate them. It brings the web application's problem to the desktop applications as well.

Thus, this work explores how to instantiate the Page Object Pattern (POP) (Yu et al., 2015) in the context of desktop applications. The POP models the web page application into objects, which contain specific functions of an area of the page and act independently. The page objects (POs) are a middle layer between the web pages and the test codes. As a result we propose a method that provides guidance for developers to instantiate this pattern in desktop applications. In this case, POs are referred to as *view objects* (VOs). One of the main benefits of this method is regarding to the test mode. When following the steps to implement view objects, we prepare our tests to run in headless mode. It means that our desktop application interface does not need to be instantiated to be tested. All operations are simulated and done without displaying the UI, mocking important variables of the UI to be used within VOs.

This work starts by giving a background on relevant concepts that are required to understand this work (Chapter 2), such as testing techniques and explanations about the adopted technologies. After that, the method for instantiation of the POP is given

with an example (Chapter 3). The evaluation part regards to an evolved application of the previous example and how to apply this method in this specific case (Chapter 4). A discussion chapter is directed to approach some ideas and considerations (Chapter 5). To finish, the Chapter 6 presents final conclusions.

2 BACKGROUND

With the purpose of achieving the best understanding of the view objects, we need to review paradigms and approaches. Environments for developing user interfaces applications are introduced, as well as some concepts regarding to the Model-View-Controller concept to build applications, headless testing and page object pattern.

2.1 Software Testing

Nowadays, software errors might be responsible for time and money costs. Although it is not possible to remove all errors of an application, there are ways of testing that enable to reduce considerably the amount of errors and the most severe errors. Aiming at quality enhancement and money saving (and also, indirectly, time saving), TDD (Janzen and Saiedian, 2005) and BDD (Solís and Wang, 2011) are adequate testing techniques to predict and identify errors – at the development phase – that might happen when running the application developed.

Focusing on good results of tests developed, there is a pattern proposed by Bill Wake called 3A Pattern¹. The main goal of this pattern is to structure the tests properly, in a way we can identify important aspects we need to test the application. Basically, the tests should *Arrange*, *Act* and *Assert*. According to him, arrange means setting up the objects that will be tested. Act is use these objects and give them commands. By making asserts we ensure some circumstances about the object. So with the use of the 3A Pattern, consistent test cases can be written.

With the intention to explain BDD, there are main concepts that should be explained before, such as TDD and ATDD, to be able to understand this approach.

2.1.1 Test-Driven Development

TDD is drawing the attention from many researchers and developers, being broadly used. The central characteristic is the matter that TDD uses short iterations to create a software with simple initial design (Janzen and Saiedian, 2005, p. 01).

Hammond and Umphress (2012, p. 01) define:

¹See <http://xp123.com/articles/3a-arrange-act-assert/>

“Test Driven Development (TDD), also referred to as test-first coding or Test Driven Design, is a practice where a programmer is instructed to write production code only after writing a failing automated test case. This approach offers a completely opposite view of the traditional test-last approach commonly used in software development, where production code is written according to design specifications, and, typically, only after much of the production code is written does one write test code to exercise it.”

Kent Beck is considered the creator of this technique. In one of his books, he provides more details about the practice of TDD and claims that the cycle process is composed of five steps (Beck, 2003, p. 92):

- *Write a new test case that initially does not pass.*
- *Run all test cases and see the new test case fail.*
- *Make a change on the code to make it pass.*
- *Run again the tests and see all of them pass.*
- *Refactor the code to remove duplication.*

Although the original TDD makes the developers to focus on what is significant to a small test pass at a low-level (development-level) – by repeating the steps of writing new test cases that fails, passing them and then *refactoring* – there are also relevant questions related to the application-level, which is at a higher-level in comparison with TDD. In this approach of running application-level tests, a technique similar to TDD is introduced and called Acceptance Test-Driven Development (ATDD).

Therefore, Hammond and Umphress (2012, p. 04) claim that higher-level tests are useful to set a context – ensuring that the code is providing the user’s desired functionality – and this technique is being used by software engineers. From another perspective, in the ATDD approach, users are responsible for writing tests before the beginning of the implementation phase. Beck (Beck, 2003) also notes that the time between a test (that was written by a customer) and a feedback (when the test finally passed) can be very long. So it is better to apply TDD instead (Beck, 2003, p. 185). So the ATDD approach is based on acceptance tests, which allow developers to focus on a specific goal of a customer from each user story.

TDD and ATDD are correlated, but different in their matter. On the one hand, TDD approach is a developer side mechanism, in which test cases are written to perform specified functions and customers might not be able to understand them. On the other hand, ATDD is a mechanism that involves developers, customers and testers, in which they certify that the requirements are well-defined and customers are able to read them.

The combination of both approaches results in a technique called Behavior-Driven Development (BDD), and in the next section it is discussed how it should be constructed in practice.

2.1.2 Behaviour-Driven Development

As mentioned before, the merge of TDD and ATDD generated BDD. North et al. (2006) was the first person who made the BDD approach known. The reason for the creation of a new name for the BDD approach is, according to (Hammond and Umphress, 2012, p. 03): *“The initial reason for the name shift from Test-Driven Development to Behaviour-Driven Development was to alleviate the confusion of TDD as a testing method versus TDD as a design method.”*

BDD applies the way of writing tests first, but it focuses on tests that describe behaviour – and not like TDD and ATDD, in which the tests are focused on the unit implementation.

Given that the BDD approach is still not clear and different authors have distinct opinions about its definition, the descriptions and definitions of BDD are not determined. Six main characteristics of BDD were identified by Solís and Wang (2011, p. 02), but here we limit ourselves to introduce solely the most relevant to our work.

Ubiquitous Language One of the essences of BDD is the concept of ubiquitous language, which its structure comes from a domain model and its knowledge is carried in a dynamic form (Evans, 2004, p. 25). The communication between domain experts and developers is crucial in a process of software development. Due to the gap between both areas, domain experts are not used with some specific and technical terms, in which developers work with and are familiarized. Thus, a miscommunication usually takes place, because the experts are not able to express what they want (assuming the experts know what they want), as well as the developers are not able to understand what the domain experts want.

In order to avoid some linguistic flaws in this process of communication, an accurate translation is required. Therefore, an ubiquitous language helps both parties to speak an universal and "common" language. It is important to highlight as noted in Solís and Wang (2011, p. 02) essay:

“Creating a ubiquitous language for a project is crucial since it should be used throughout the development lifecycle. A dictionary is specified

at the beginning of the project. Most vocabulary of the ubiquitous language should come from the analysis phase. However, new words can be inserted at anytime of the development phases. Creating the ubiquitous language needs to involve anyone (domain experts and developers) who will use the language. In the design and implementation phases, developers will use the language to name classes and methods.”

In short, the terms in a document, for example, should appear in conversations face to face, in the diagrams and in the code. That is the main point that characterizes an ubiquitous language.

Iterative Decomposition Process At the phase of requirements gathering, customers and developers need to communicate with each other, in a way that customers must establish what they want to be developed. However, a starting point for this communication is not always simple to find and it is also difficult to clarify what should be done.

According to Solís and Wang (2011, p. 03):

“Therefore in BDD the analysis starts with identification of the expected behaviours of a system, which are more concrete and easy to identify. The system’s behaviours will be derived from the business outcomes it intends to produce. Business outcomes are then drilled down to feature sets. A feature set splits a business outcome into a set of abstract features, which indicate what should be done to achieve the business outcome. Feature sets are derived from discussions between customers and developers on business outcomes.”

Additionally, they claim that business outcomes are the starting point of BDD process, so the developers must know which set of features should be developed first and, for that, the customers have to specify the priority of the feature. To represent those features, user stories are created to describe the new capability of the system from the perspective of the customer.

For Beck and Fowler (2001, p. 43), features and user stories are synonyms, as stated in their book: “A user story is a chunk of functionality (some people use the word feature) that is of value to the customer.” Each user story might contain different scenarios, which express how the system should behave in a specific context when a circumstance takes place. Some issues should be delineated by user stories, e.g. the role of the user in an user story, the user desired feature and the benefit for the user if the system provides the feature (Solís and Wang, 2011, p. 03). In the next description we provide further details, where the matter is the description of user stories and scenarios. Applying iteratively this process of decomposition outlined above, we can reach an enough initial analysis.

Plain Text Description with User Story and Scenario Templates When writing user stories and scenarios in BDD, it is really useful to have some pre-defined "texts" that have a pattern to be the starting point of the description, which are called templates. Details about user stories are stated by Beck and Fowler (2001, p. 44):

- *Stories must be understandable to the customer;*
- *A user story is nothing more than an agreement that the customer and developers will talk together about a feature;*
- *Each story must provide something of value to the customer;*
- *Stories need to be of a size that you can build a few of them in each iteration;*
- *Stories should be independent of each other;*
- *Each story must be testable.*

Taking into account these guidelines to write good user stories (and scenarios), the templates are written using an ubiquitous language, which developers can understand how the system should behave and what it should support in order to implement it, and users can check if they really need the feature by seeing the benefit they might get from it. Thus, the user stories are created based on the following templates (Solís and Wang, 2011, p. 03):

[StoryTitle] (One line describing the story)

As a [Role]

I want a [Feature]

So that I can get [Benefit]

As mentioned at the template above, the story title describes the story that is done by someone or something (role). The feature is described as a functionality or an activity that will be performed, and with that some benefit is obtained. For scenarios, the template is as the following (Solís and Wang, 2011, p. 03):

Scenario 1: [Scenario Title]

Given*[Context]*

And [Some more contexts]....

When*[Event]*

Then*[Outcome]*

And [Some more outcomes]....
Scenario2: [Scenario Title]

Scenarios are specified to understand how the selected feature should behave, when the feature is performed in a specific context and an event occurs. The context delimits the scope of the operations and sets up preconditions, while the event is the behavior we are focused on. As a result of this scenario, the outcome should describe how the system will act and verify if the right thing happened in the *When* clause.

It is relevant to notice that everything in the brackets are mapped to tests later on, in which the methods, for example, are named as what is written there. It means that when we write tests, the name of the methods are linked with the clauses *Given*, *When*, *Then*.

The *And* clause can be used in any of the three clauses. It serves as an abbreviation not to repeat the others, and it is nested to the last one.

Automated Acceptance Testing with Mapping Rules the automated acceptance testing from ATDD is also a characteristic contained in BDD (Solís and Wang, 2011, p. 03). This step is basically related to the mapping from scenarios to test code, that is basically one of the features of BDD, as claimed (Solís and Wang, 2011, p. 04): "*The classes implementing the scenarios will read the plain text scenario specifications and execute them. In other words, BDD allows having executable plain text scenarios.*" Thus, when running scenarios, the acceptance criteria is automatically analysed by BDD.

2.1.3 Headless Testing

The word *headless* simply means that something (e.g. application, browser, operation) works without a graphical user interface (GUI). The term *headless testing* is associated with the challenge to test a GUI without displaying it. The main idea of testing an application headless is that tests might be more consistent. It means, the tests do not check whether a certain button was actually clicked, but what is the real meaning and action of this click. Consequently, the time saving is an advantage, because the GUI is not shown, and we do not need to wait for clicks or selections events and their responses.

There is a technology that enable us to create automated tests for web browsers and run it headless, called Selenium². It is important to highlight that the focus of Selenium is not to run tests headless, being this mode just one of the features. The concept of this technology was used as a basis to this work in order to automate tests for standalone applications. This technology was also a pioneer for the creation of a pattern called Page Object³, which is going to be discussed in the next section.

2.2 Page Object Pattern

Patterns are usually generic and expert solutions that solve problems, which commonly appears in a context. So it tends to be reused many times under similar circumstances. Thus, patterns for test automation are really useful to solve particular complications, because it means that these patterns were experienced by many people and they succeeded. It is important to highlight that patterns *are not* ready method solutions that can be used and applied to all situations in a stepwise way, but they must be instantiated for each problem in order to solve them. The usage of patterns depends on the scope we are concerned. The *Design Patterns* can be described as generic solutions for a software architecture or design, because they instruct how the automated tests should be constructed, in a way that they will be easy to maintain, effective and efficient. So there is a *design pattern* called **Page Object Pattern** (Leotta et al., 2013). POP is a design pattern that has been generally used and the term was widespread by *Selenium*. This pattern was designed to create automated tests for web browsers. It can be better applied when an application has many pages or many states. The idea of this pattern is that the web page application is modelled into objects, which can be specific parts of the web page that might act independently. It means that all the functions that can be performed within an object are encapsulated to something, called Page Object. In this way, every change required at the web page (UI) will not affect the test classes, but just the page object referred in that functionality.

Yu et al. (2015, p. 01) claims that:

“Page objects introduce a middle layer between web pages and test code so that the web page elements are abstracted by the page objects and the test code only contains testing logic code. In this way, a tester can write test cases based on page objects, without concerning about the actual representation of the web application.”

²See <http://www.seleniumhq.org/>

³See http://docs.seleniumhq.org/docs/06_test_design_considerations.jsp/

Figura 2.1: Illustration of the Page Object Pattern modules

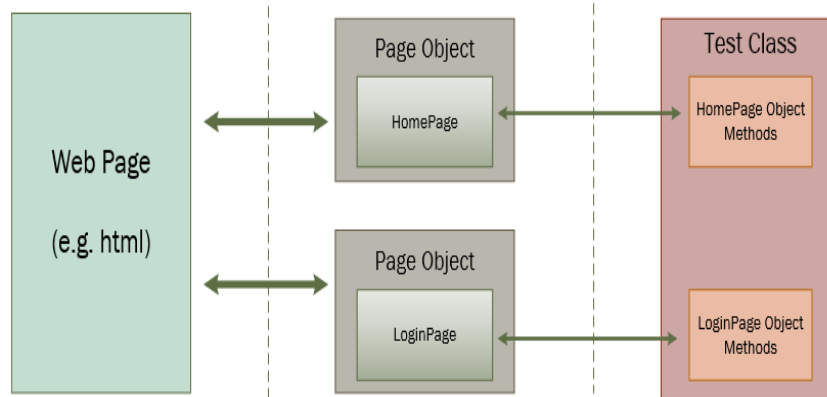


Figura 2.2: Illustration of the Technische Universität Berlin Website Login Area.

Ihr Login

Benutzername	<input type="text"/>
Passwort	<input type="password"/>
<input type="button" value="Einloggen"/>	

For a better understanding of his statement, Figure 2.1 shows how the Page Objects works. So with this picture we can see the separation between the test code and the navigation code (page objects). As an example, we take as example the Technische Universität Berlin website login area, as shows Figure 2.2.

For the LoginPage Class, we would have a login method such as the source code presented in Source Code 2.1:

Source Code 2.1: POP Example: Login Method of TUB website

```

def login(username, password)
  find(:id, 'user').set(username)
  find(:id, 'password').set(password)
  find('.einloggen-btn').click
end
  
```

And as an example of a test, which calls the login method within a test class is exemplified in Source Code 2.2:

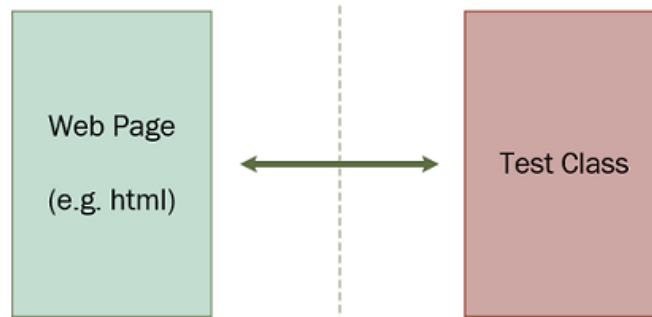
Source Code 2.2: POP Example: Test for Login Method of TUB website

```
login_page.login('felipe', 'pw123456')
```

Selenium Webdriver enables this search by elements of the GUI, facilitating the creation of the PO classes.

We may mention *advantages* regarding to the usage of the POP, starting from the application modules communication. Supposing we have structured our application modules as shown Figure 2.3 (not POP-like), we can notice that the application would have only two modules: one responsible for the whole application, and another one for the tests. This situation might bring future problems regarding to the tests written. Being the GUI directly tested by the tests, it may be hard to maintain the tests, because they are related precisely to the design conditions of the UI (e.g. position of the elements). Directly testing the GUI, the complexity of the application increases, because the tests become hard to understand and not easy to read. So the source code should be usually analysed in order to check the compliance of the tests. These fragilities described above might be solved with the use of the POP.

Figura 2.3: Illustration of the modules of a Web Application for automated testing



Now taking into consideration *disadvantages* of the use of POP, we have to consider that the experts have an important role while modelling and implementing the page objects. It means that they must have enough knowledge to do the tasks properly.

2.3 Adopted Technologies

Our method is generic enough to be used with different technologies to implement user interface or to support automated testing. However, in order to illustrate our approach, we selected a set of widely used technologies.

2.3.1 JavaFX

When we think about creating a graphical user interface, JavaFX comes up as an useful tool for this objective. *JavaFX*⁴ was announced in May 2007 for the first time at JavaOne conference. *JavaFX* is a software platform developed by Oracle based on Java for creation and delivery of rich internet applications (RIAs) that can run on many different devices, using in their first versions a *JavaFX Script* language. Furthermore, *JavaFX* has support on desktop computers with operational systems like Microsoft Windows, Mac OS and Linux, for web browsers and mobile devices. Since there are many other technologies created for the development of user interfaces – e.g. *Java Swing* and *Java 3D* – *JavaFX* came to replace them. The version 8.0 was the one used in our work and this version is part of the Java JDK/JRE 8. So in this version, there is no longer a specific scripting language (JavaFX Script) to develop RIAs.

Being a technology that helps on the development of RIAs on the client-side, *JavaFX* provides many interface resources such as multimedia (sound, video), graphics and animations. When developing an application using this technology, it enables to improve the visual aspect of the application and can be used in different platforms. JavaFX can also be integrated to already created resources in Java, so it means that the reuse of already implemented applications is also possible. Moreover, it might be integrated to a variety of Java Frameworks (e.g. NetBeans, Eclipse) to enable the maintenance, suitability and improvement of visual resources.

A given interface should be clear enough to enable the user to achieve its goals. Furthermore, the usability performance could be measured by its effectiveness (e.g. if a set of intended tasks is able to be achieved, so it means the software is effective), efficiency (e.g. resources as time, memory or money were used in large scales to achieve the goals) and satisfaction (e.g. if the users find the product acceptable). The standard ISO 9241-11 provides the definition for usability, which states Bevan (1995, p. 01): “*The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use.*”

⁴See <http://docs.oracle.com/javase/8/javase-clienttechnologies.htm>

2.3.2 Cucumber

Cucumber Tool⁵ is a framework that allows to write and execute BDD tests. It enables the creation of automated tests for the functional validation in a plain text. This text is written in a language that Cucumber understands: the Gherkin⁶. Basically, the user describes the behavior of the system by creating a feature file, where the user stories and the scenarios are written, based on templates. In the scenarios, different test situations are described. Cucumber is not able to understand itself what to do when it finds the keywords *Given*, *When* and *Then*, for example, so then Gherkin language helps on it. Another keywords such as *Scenario*, *And*, *But*, *Background*, *Examples*, *Scenario Outline* are also identified by Gherkin language.

Since Cucumber is also a tool for test automation, the *Step Definitions Class* is required to guide Cucumber how to act and in which sequence. The step definitions are represented as a class that basically maps the clauses in the feature file to methods in the step definitions class file. Then the clauses can be implemented properly to perform the actions required. It is important to notice that Cucumber uses regular expressions to make this mapping of the steps.

The key advantage of Cucumber is that the description of the features (in the scenario file) can be written and understood by someone who is not involved directly to the *technical* part of the project (e.g. the user).

⁵See <https://cucumber.io/>

⁶See <https://cucumber.io/docs/reference>

3 TESTING DESKTOP APPLICATIONS WITH VIEW OBJECTS

This chapter focuses on describing the main contribution of this work: the view objects (VO), which can be seen as an instantiation for the page object pattern, but for desktop applications. We detail how VOs must be developed as a method, and a running example exemplifies how it works.

3.1 Method Overview

We can separate the method for the creation of the VOs into four steps in order to achieve our desired result, which are described as follows.

Develop the Desktop Application The user interface and the application logic must be divided into different modules. This separation is important to be done because of the step regarding to the implementation of the view objects, when we care about the mock of the elements of the application logic and the GUI elements to run the headless tests.

Choose View Objects The choice of the view objects needs to be observed, taking into consideration how the elements of the view interact to each other. So the expert should navigate through the application window(s) and consider which view objects can be created to encapsulate specific functionalities of the application.

Implement View Objects The selection of relevant variables and operations of the application are used as a mock to implement the view object classes. We care about the significant elements of the GUI and of the logical part, i.e. elements which have an important role on the interaction with the user. For example, in relation to the GUI elements, the user can write on text fields and click at buttons in a certain window, so the view object must contain a variable with relation to the text fields and to the buttons. In relation to the logic part, if we have a database, we must do the same, creating variables regarding to that as well. Furthermore, we need to consider which are the possible operations at that window, e.g. if at a certain window, the user is able to *select* an entry on the table, *click* at a button, *write* on a text field and so on. Then, methods associated with these operations will handle with the mocked variables created before. That is why the first step is crucial (regarding to the proper development of the application).

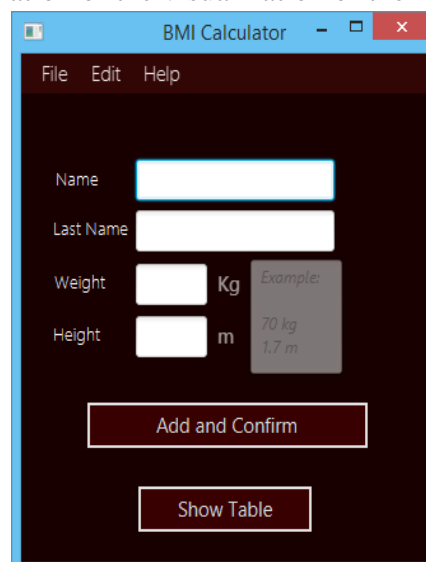
Test The purpose is to use the methods within the view objects created in order to test our application. So the tests are written instantiating the view objects and using them.

In the next section, we show a running example applying these steps described above in order to show how it works in practice.

3.2 Running Example


A simple desktop application was developed in JavaFX using Eclipse IDE and it is used to illustrate our method throughout this chapter. The source code of the application is in Github¹. The application calculates the Body Mass Index (BMI) of a person. So the user informs name, last name, weight, height and she can add a new entry to the table (Figure 3.1). If all information required is valid, another window displaying the updated table opens. The BMI is automatically calculated and it is assigned with such entry. The user can also just click at a button that displays the results (Figure 3.2). The possibility of editing an specific entry is also possible and a new window opens for this edition (Figure 3.3).

Figura 3.1: Illustration of the visualization of the Main Window view



¹See https://github.com/fmschmidt/BMI_Application

Figura 3.2: Illustration of the visualization of the Table Window view



Name	Last Name	Height	Weight	BMI	Date
Felipe	Schmidt	1.7	70.0	24.221453	2015-11-27
Felipe	de Medeiros	1.7	70.0	24.221453	2015-11-27
de Medeiros	Schmidt	1.7	70.0	24.221453	2015-11-27
Schmidt	Felipe	1.7	70.0	24.221453	2015-11-27

Buttons: Edit... Delete

3.3 Method Steps

The overview of the method given above is applied to give an idea how it works in a stepwise way.

1. Develop the Desktop Application

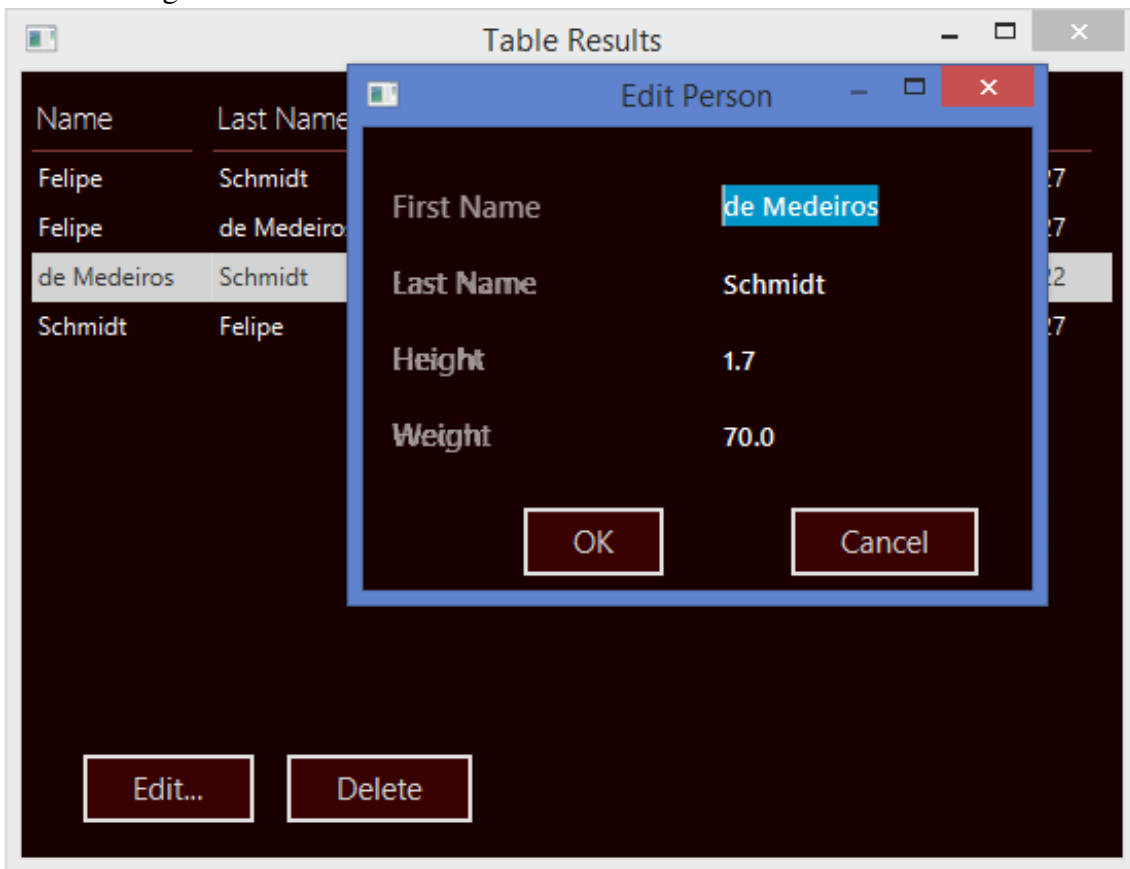
For this step, we have already done in the previous section of the running example. It is important to highlight that the most important thing to consider is about the separation of the application logic and the user interface. Thus, we keep independent modules communicating with each other (See Figure 2.1 of the POP modules). Now it is important to know how to choose the view objects concerned.

2. Choose View Objects

The first thing we should consider is related to the navigation through the application. So this choice is based on the way the content is displayed to the user.

In order to explore every VO class of this application, the following steps are taken for the example: the user clicks at "Show Table" and a new window appears with the results table. Then, clicks "Edit" over an entry, and a new window opens with

Figura 3.3: Illustration of the visualization of the Edit Window view



the details of the entry. After considering this navigation flow, we explored every possibility of view windows on this application. In other words, the possible user interactions to the application, that resulted into different windows, were identified. Therefore, we consider that, for this kind of application, each window corresponds to one VO. So the windows are transformed into VO classes. Being VOs an instance of POP, the nomenclature "Window" is appended at the end of each VO class name to explicitly identify them.

3. Implement View Objects

Basically three important considerations for the proper implementation of VO classes are considered:

- choice of relevant action elements of the UI. Create variables and use them such as mocked variables;
- choice of logical elements of the considered window, non-UI elements, such as database related or controller methods;
- implementation of the operations.

The rough idea is that the variables represent the UI actions (such as button click, writing to text field) and the logic part for instantiation. Thus, the pattern hides these variables from the user. Methods represent user actions (e.g. fill in name). Then, method codes map user actions to UI actions.

For the first bullet point, it is associated with relevant action elements of the UI. To choose the correct elements of the UI, first we have to analyse the source code of the desktop application (regarding to the fxml file, see Github² for the BMI application source code). In JavaFX, these visual elements of the UI (that we refer to as action elements) have an *@FXML* annotation that are synchronized with the fxml file. From this UI class, we must extract the important elements, e.g. if the user may act on text fields, these text fields must be mocked. So then we have variables for these text fields (Source Code 3.3.1a). The same is for any kind of element (e.g. labels, text fields, buttons), which might be operated by an user.

For the second bullet point, we have some variables which are used for the instantiation of our database and for logical operations in our application, such as add new person, delete person and edit person. So these variables are declared because we will use it to retrieve data and operate (Source Code 3.3.1b).

After knowing all relevant variables, we are able to care about the operations that can be done in a given window (third bullet point). For the main window of the application example above (Figure 3.1), we are able to click at buttons and write on text fields. So we implement methods which can be executed at this given window, such as add new person, fill in text fields and show table results (Source Code 3.3.1c). It is relevant to notice that the methods implemented in a specific VO class must be related to the responsibility of the concerned application window view. So it cannot contain methods which are not related to the window (e.g. the main window cannot contain "editPerson" method, because the main window does not have this operation).

After it is done, we have successfully implemented the VO InsertDataWindow class, which corresponds to the main window of the running example (Figure 3.1). The same procedure must be done for the other windows as well, to achieve complete tests.

4. Test

As POP suggests and it was explained previously, the VO methods should contain

²See https://github.com/fmschmidt/BMI_Application

the user actions such as click and write, because inside the future tests should be written in a way that we know what to do (e.g. show the table) and not how to do (e.g. click on "Show Table"). In other words, we are not interested if the mouse went to a given position and clicked at a button labelled "Show Table". We are interested if the action of showing the table was actually done. As an example of how a test like this look like, we present Source Code 3.3.1d.

Our VOs came to ease up the test process, enabling headless testing and not leaving the UI brittle for changes. Additionally, our tests become more readable and understandable, also taking into consideration that our test classes will not test directly the UI, but the view objects, which encapsulate GUI elements. Using our VOs and combining with BDD technique, we can test our application by writing scenarios using an ubiquitous language (to understand how our system should behave), and then making a translation by implementing the step definitions class.

Writing Scenarios According to Chapter 2.1.2, we have some concepts and templates that facilitate us to write the feature files in plain text. The following examples will show how the features would be written to be then translated to test code.

Source Code 3.1: Writing Scenarios: InsertPerson feature

```
Feature: User inserts a new data to the table
```

```
As an User
```

```
I want to insert person data to the table
```

```
So that I can save the development recording their data
```

```
Scenario: the person information is valid
```

```
Given I have log entries:
```

```
  | firstName | lastName | height | weight |
  | Felipe   | Schmidt  | 170    | 70     |
  | Felipe   | de Medeiros | 170    | 70     |
  | De Medeiros | Schmidt | 170    | 70     |
  | Schmidt   | Felipe   | 170    | 70     |
```

```
When I enter a person information:
```

```
  | firstName | lastName | height | weight |
  | test      | test     | 180    | 80     |
```

Then I see a result table:

firstName	lastName	height	weight
Felipe	Schmidt	170	70
Felipe	de Medeiros	170	70
De Medeiros	Schmidt	170	70
Schmidt	Felipe	170	70
test	test	180	80

Scenario: the person information is invalid

Given I have log entries:

firstName	lastName	height	weight
Felipe	Schmidt	170	70
Felipe	de Medeiros	170	70
De Medeiros	Schmidt	170	70
Schmidt	Felipe	170	70

When I enter a person information:

firstName	lastName	height	weight
		180	80

Then I see a result table:

firstName	lastName	height	weight
Felipe	Schmidt	170	70
Felipe	de Medeiros	170	70
De Medeiros	Schmidt	170	70
Schmidt	Felipe	170	70

So based on the steps of the Chapter 2.1.2 specified, we could write the scenarios following the guidelines. Now, the next step is regarding to the Step Definitions class implementation.

Step Definitions Class Implementation This class is responsible for taking each line from the feature files and map to methods in test code, using a Gherkin language to recognize the keywords like Given, When and Then. To show how it would work, Source Code 3.2 represents this translation.

Within this class is where we are going to use the view objects we have previously created. The readability of the tests are more intuitive in a way we can understand what it should do and what is actually doing.

We may notice that each line of the keywords are taken in particular and converted to methods. These methods have an important role to execute what we called the 3A Pattern: Arrange, Act, Assert.

Source Code 3.2: Step Definitions Class Implementation

```
public class StepDefinitions {

    private InsertDataWindow idw;
    private TableResultsWindow trw;
    private EditPersonWindow epw;
    private Person personDataToAdd;
    private ObservableList<Person> personDataList;

    @Given("^I have log entries:$")
    public void I_have_log_entries(List<Person>
        personList) {
        personDataList =
            FXCollections.observableList(personList);
        idw = new InsertDataWindow();
        trw = new TableResultsWindow();
    }

    @When("^I enter a person information:$")
    public void I_enter_a_person_information(List<Person>
        newEntryList) {
        personDataToAdd = new
            Person(newEntryList.get(0).getFirstName(),
                newEntryList.get(0).getLastName(),
                newEntryList.get(0).getWeight(),
                newEntryList.get(0).getHeight());

        idw = idw.fillInInfo(personDataToAdd);
        trw = idw.AddPerson();
    }

    @Then("^I see a result table:$")
```

```
public void I_see_a_result_table(List<Person>
    newEntryList) {
    trw =
        trw.updatedTable().assertTableCount(newEntryList.size());
    }
}
```

As a result, when we execute the Cucumber feature, we have the lines that appears in the console, which are shown in Source Code 3.3.

Source Code 3.3: Console response after running the feature

```
2 Scenarios (2 passed)
6 Steps (6 passed)
```

Thus, the features could be mapped and translated to methods regarding to the view objects. We may notice that the tests are more clear, being the readability also better.

```

public class InsertDataWindow {
    private String txtName = "txtName";
    private String txtLastName = "txtLastName";
    private String txtWeight = "txtWeight";
    private String txtHeight = "txtHeight";
    private static String BTN_ADD_AND_CONFIRM =
        "BTN_ADD_AND_CONFIRM";
    private static String BTN_SHOW_TABLE = "BTN_SHOW_TABLE";

```

a: Action elements

```

private ObservableList<Person> personDataToRetrieve;
private TablePersonActionController tpc;
private Person personToAdd;

```

b: Logic elements

```

private void printText(String element, String text){
    element = "";
    element = text;
}
private void click(String element){
    if(element == BTN_ADD_AND_CONFIRM){
        tpc.addNewPerson(personToAdd);
    }
    else if(element == BTN_SHOW_TABLE){
        personDataToRetrieve = tpc.getPersonData();
    }
}
public InsertDataWindow fillInInfo(Person person){
    printText(txtName, person.getFirstName());
    printText(txtLastName, person.getLastName());
    printText(txtHeight, String.valueOf(person.getHeight()));
    printText(txtWeight, String.valueOf(person.getWeight()));
    personToAdd = person;
    return this;
}
public TableResultsWindow AddPerson(){
    click(BTN_ADD_AND_CONFIRM);
    return new TableResultsWindow(tpc);
}
public TableResultsWindow showTableResults(){
    click(BTN_SHOW_TABLE);
    return new TableResultsWindow(personDataToRetrieve);
}
}

```

c: Methods implementation

```

trw = idw.showTableResults();

```

d: Small Test Example

Listing 3.3.1: InsertDataWindow view object implementation

4 EVALUATION

In the previous chapter, we introduced our method to instantiate the Page Object Pattern in a desktop application, illustrating it with a simple version of the BMI calculator application. In this application, windows (views) are not split into parts and, therefore, one virtual object is used to test each view. In this chapter, we show a preliminary evaluation of our approach by using it in a more complex scenario. The source code of the BMI application can be checked in Github¹.

4.1 The Evolved BMI Calculator Application

The Evolved BMI Calculator Application consists of a single window (view) with two-sides: the left-hand side contains text fields and buttons to fill out. The right-hand side contains the table with entries (Figure 4.1). The user can simply act on text fields, writing a person information and add it to the table. The information added then appears instantly to the table if valid (Figure 4.2). When clicking at a person entry to edit, the person information goes to the text fields to be edited (Figure 4.3). If everything was successfully modified and the save button is pressed, the entry is updated on the table (Figure 4.4).

This is the navigation flow of the application. We can notice that this application works with a single view with smaller views that act together in order to give back a global result. It happens usually in web pages, e.g. when we have a static menu bar and we press any tab. The content changes, but the menu remains the same.

¹See https://github.com/fmschmidt/BMI_Application

Figura 4.1: Illustration of the visualization evolved BMI application

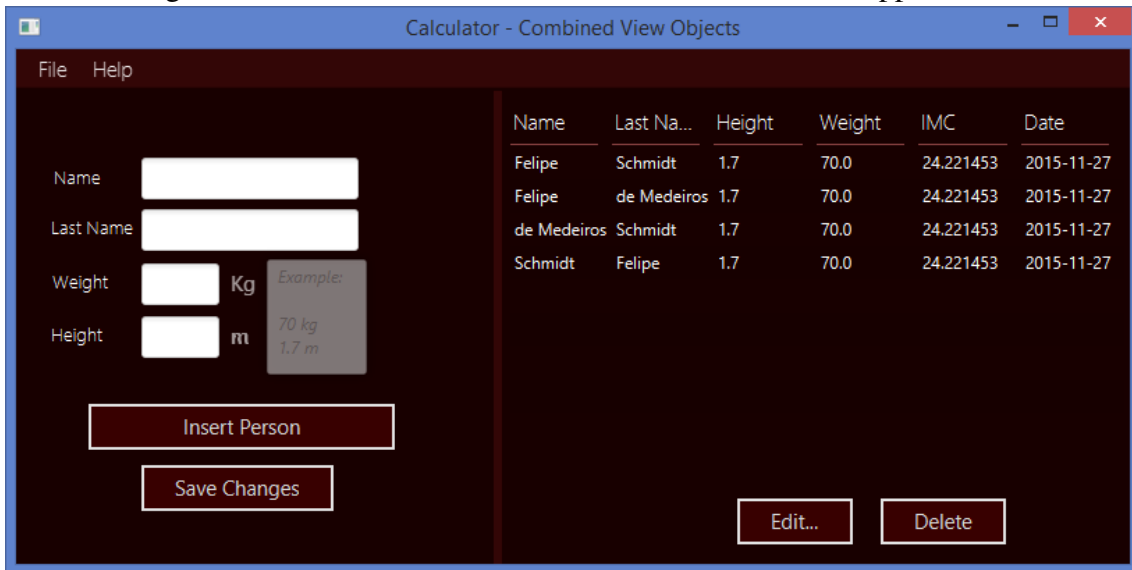


Figura 4.2: Illustration of the visualization evolved BMI application adding a person

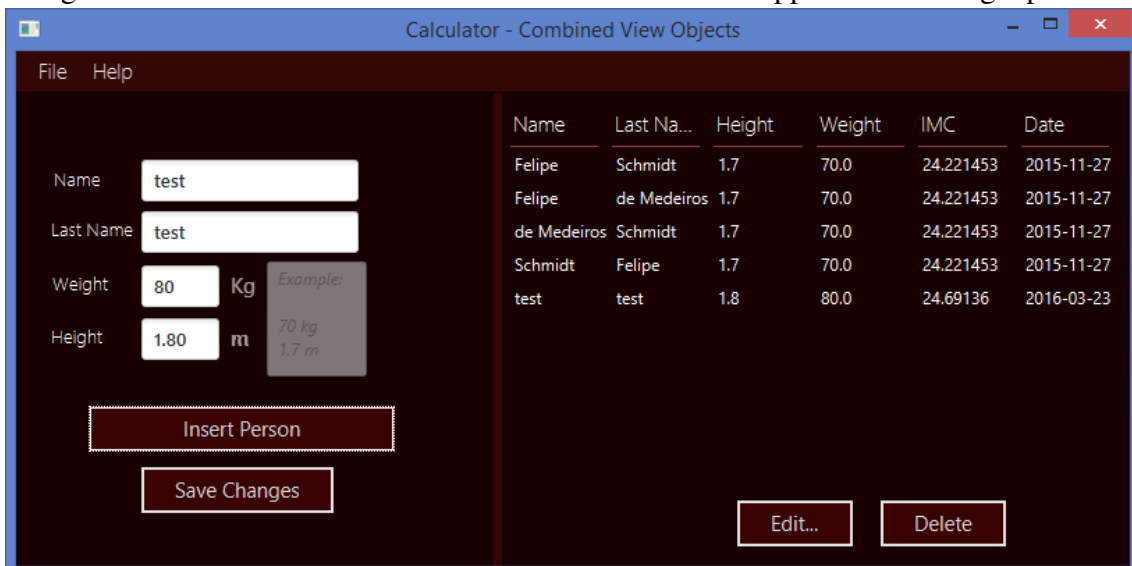


Figura 4.3: Illustration of the visualization evolved BMI application editing a person

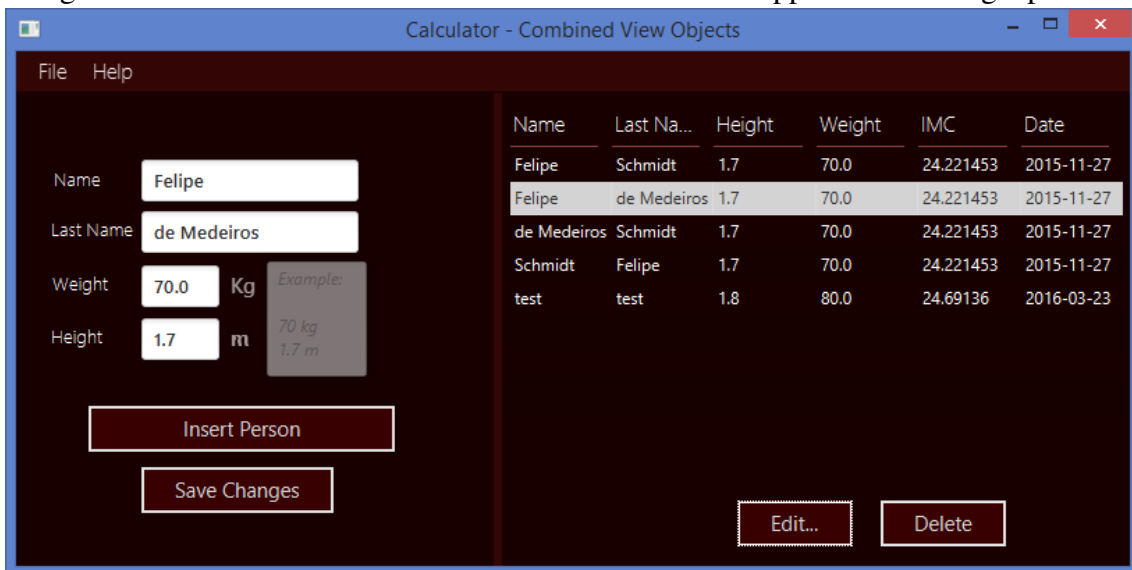
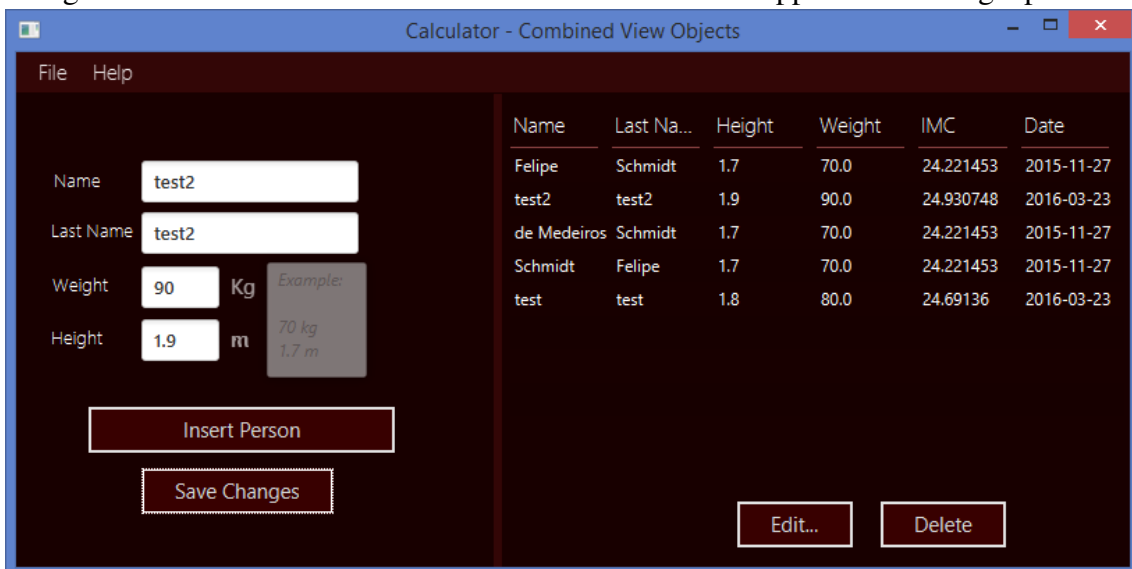


Figura 4.4: Illustration of the visualization evolved BMI application saving a person



4.2 Develop Desktop Application

The overview of the method given in a previous chapter is applied and it flows in the same way as the running example given previously.

1. Develop the Desktop Application

For the development of the application we assume the previous section. It is important to highlight that the logic application and the UI are separated. Thus, we can manipulate them adequately to achieve our desired result.

2. Choose View Objects

For the choice of our VO classes, we must consider the navigation flow of the application and how the windows interact with each other. We noticed two different sides that communicate to each other to exchange information. After knowing this action, we consider we have a larger window that has two small windows that communicate with each other. So basically the large window is a class that delegates its responsibilities to the small ones and receives the view updated. The small windows act (as an interaction) with each other and these results are combined and given back to the great one. Thus, we have three VO classes to implement: `PersonInfoWindow` (left-side), `TableWindow` (right-side) and the `OverviewApplicationWindow` (whole window).

3. Implement View Objects

For this step we should reconsider the three important points described at the previous chapter of the running example: choice of action elements of the UI, logical elements and implementation of the operations.

For each VO class, we followed these steps to implement them. The smaller ones were explained at the previous chapter and they have nothing new (Source Code 4.1 and 4.2 are not complete). We show how the great VO looks like, because it is a new concept. The great VO uses the methods from the smaller ones. So its variables are the instances of the small VO classes (Source Code 4.2.1a).

As long as the `OverviewApplicationWindow` delegates its responsibilities to the small ones, it *does not* have any operation like click at a button. The small ones do that. It contains specific methods regarding the real functionalities of the window, such as fill out text fields, delete person and edit person (Figure 4.2.1b).

4. Test

In this VO classes implementation we are using JUnit as another possibility to test our application (Source Code 4.2.1).

```
public class OverviewApplicationWindow {
    private OverviewActionController overviewController;
    private TableWindow tableWindow;
    private PersonInfoWindow personInfoWindow;
    private ObservableList<Person> personDataToRetrieve;
```

a: Greater VO Variables

```
public OverviewApplicationWindow fillInInfoAndAdd (Person
    person) {
    tableWindow = personInfoWindow.fillInInfo(person);

    tableWindow = tableWindow.assertTableCount(5);

    return new OverviewApplicationWindow(personInfoWindow,
        tableWindow);
}

public OverviewApplicationWindow deletePerson(int index){
    tableWindow = tableWindow.removePerson(index);
    tableWindow = tableWindow.assertTableCount(3);

    return new OverviewApplicationWindow(personInfoWindow,
        tableWindow);
}

public OverviewApplicationWindow editPerson(int index){
    personInfoWindow = tableWindow.editPerson(index);

    return new OverviewApplicationWindow(personInfoWindow,
        tableWindow);
}

public OverviewApplicationWindow editPersonDetails(Person
    oldPersonSelected, Person newPerson){
    tableWindow =
        tableWindow.editPersonDetails(oldPersonSelected,
            newPerson);

    tableWindow = tableWindow.assertTableCount(4);

    return new OverviewApplicationWindow(personInfoWindow,
        tableWindow);
}
}
```

b: Greater VO Methods

Listing 4.2.1: OverviewApplicationWindow view object implementation.

Source Code 4.1: PersonInfoWindow Implementation

```
public class PersonInfoWindow {

    private String txtName = "txtName";
    private String txtLastName = "txtLastName";
    private String txtWeight = "txtWeight";
    private String txtHeight = "txtHeight";
    private static String BTN_INSERT = "BTN_INSERT";
    private ObservableList<Person> personDataToRetrieve;
    private OverviewActionController overviewController;
    private Person personToAdd;
    private TableWindow tw;
    private Person personToEditSent;

    public PersonInfoWindow() {
        overviewController = new OverviewActionController();
        //PageObject corresponds to: driver.navigate().to(url);
        personDataToRetrieve = overviewController.getPersonData();
        personToAdd = new Person();
    }

    public PersonInfoWindow(Person personToEdit) {
        personToEditSent = personToEdit;
        setPersonInfoWindow(personToEditSent);
    }

    public TableWindow fillInInfo(Person person) {
        printText(txtName, person.getFirstName());
        //printText("title", post.getTitle());
        printText(txtLastName, person.getLastName());
        //printText("text", post.getText());
        printText(txtHeight, String.valueOf(person.getHeight()));
        printText(txtWeight, String.valueOf(person.getWeight()));

        personToAdd = person;
    }
}
```

```

click(BTN_INSERT); //click("addPostBtn");

setPersonInfoWindow(person); //sets the window;

return new TableWindow(overviewController);
}

public void printText(String element, String text){
    element = ""; //textBox.clear();
    element = text; //textBox.sendKeys(text);
}

public void click(String element){

    if(element == BTN_INSERT){
        overviewController.addNewPerson(personToAdd);
    }
    else if(element == BTN_SAVE_CHANGES){}
}

```

Source Code 4.2: TableWindow Implementation

```

public class TableWindow {
    private ObservableList<Person> personDataToRetrieve;
    private String BTN_DELETE = "BTN_DELETE";
    private String BTN_EDIT = "BTN_EDIT";
    private static String BTN_SAVE_CHANGES = "BTN_SAVE_CHANGES";
    private OverviewActionController overviewController;
    private int personIndex;
    private Person personToEdit;
    private Person oldPerson;
    private Person personToBeEdited;
    private PersonInfoWindow piw;

    public TableWindow removePerson(int index){
        select(index);
        click(BTN_DELETE);
        return this;
    }
}

```

```

}
public PersonInfoWindow editPerson(int index){
    select(index);
    click(BTN_EDIT);

    setOldPerson(personToEdit);

    return new PersonInfoWindow(personToEdit);
}
public TableWindow editPersonDetails(Person
    oldPersonSelected, Person newPerson){
    oldPerson = oldPersonSelected;
    personToBeEdited = newPerson;

    click(BTN_SAVE_CHANGES);
    return new TableWindow(overviewController);
}

private void select(int index){
    personIndex = index;
}
public void click(String element){
    if(element == BTN_DELETE){
        overviewController.deletePersonFromTheTable(
overviewController.getPersonData(), personIndex);
    }

    if(element == BTN_EDIT){
        //a new dialog to edit a person opens
        personToEdit =
            overviewController.getPersonData().get(personIndex);
        overviewController.editPersonFromTheTable(personToEdit,
            overviewController.getPersonData());
    }

    if(element == BTN_SAVE_CHANGES){
        if(!overviewController.checkExistingPerson(

```

```
overviewController.getPersonData(),
    personToBeEdited.getFirstName().toString(),
    personToBeEdited.getLastName().toString()) {
    overviewController.editPersonFromTheTable(oldPerson,
        personDataToRetrieve);
    overviewController.saveEditedPerson(oldPerson,
        personToBeEdited,
        overviewController.getPersonData());
}
}
}
```

5 DISCUSSION

In this chapter, we discuss important points related to the context of usage of POs and VO as well as the advantages and limitations of using it.

5.1 Web Page vs. Desktop View

Being Page Object a pattern directed to web pages and View Object for standalone applications, there are natural distinctions between them. Some years ago, when web pages and standalone applications were completely different, there were much more differences between both. With the development of the languages for web page creations, those differences are getting smaller, i.e. web pages and standalone applications are being much more similar. From HTML1 until HTML4, building web pages were mainly for designing texts, images and graphics, but it did not support user interactions, because each user interaction loaded a new page. Many people use web browsers for reading, e.g. blogs, forums and news, and the use of a scroll bar is relevant in this case, noticing that blank spaces at the web page are used to separate the content itself. Now comparing the fact of reading on a standalone application, a scroll bar could not be a good idea, because the views are usually desktop windows, and the use of blank spaces are limited to show the content at one window. Thus, the content displayed at a standalone interface tends to be more objective and specific. Therefore, we can notice two main differences between standalone interfaces and web pages: the type of content, and the way this content is shown. Another point to notice is that both contents are presented differently: for web pages, the use of a web browser is necessary (as well as internet connection) to access the content, in contrast to standalone applications that, as the name suggests, stand by itself.

With the development of the HTML4, the HTML5 (that is not standard yet) was released and it supports many other features that were not supported before, such as media and JavaScript. So audio, video and vector graphics (2D and 3D interactive) are now integrated to this technology and able to be played and stored in the application.

Therefore we can notice that with HTML5, the features to design web pages became much more similar to features to design desktop user interfaces, but still need to be considered what is the intention of the content that is showed on the application, because VOs deals with windows, and POs with web pages, i.e. if the content would be better

presented in a window or over web browsers, bearing in mind the requirements of each one.

It is also important to highlight that web pages are often complete pages. So after acting with a page, we get a completely new one. On another hand, standalone GUIs may normally be separated into different parts, such as sliced windows, tabbed panes and other elements. Thus, the user actions might change one or more of these parts and leave others without any changes. And with the development of the web pages, they are becoming more modern on this aspect of having the same features as standalone GUIs. Then, the VOs can be useful for them also.

5.2 Advantages of Using POP to Test Desktop Applications

An important role with the use of VOs to test our application is that we achieve naturally *headless testing*. This is possible because we mock the variables of the UI and we implement the same operations of the UI, simulating these operations. Then, every operation is done faster and the UI does not need to be instantiated. Another advantage is regarding to the tests: if we need to modify the UI implementation, the tests will remain the same, because the VOs encapsulate the UI methods. Thus, just the VOs must be modified. We have important advantages when using VOs to test desktop applications such as organization of the source codes, well-structure and the tests are easily maintained.

5.3 Limitations of View Objects

One of the bad points of using VO is that, for example, an implementation class could become a huge amount of if and elses to cover all buttons (see Source Code 3.3.1c). Then, the complexity of the source code may increase and the organization might be a problem.

6 CONCLUSION

Being software testing an area that is becoming crucial in software development field, the test automation in order to save time are essential. Different approaches and tools were proposed for web applications. The POP came up to facilitate the testing part of the application, because the application is usually defined in specific languages that complicates the test. For this reason, VOs are proposed as an instance of the POP. The main difference between both methods is basically that one aims at web browsers (POs) and the other at standalone applications (VOs). With the development of the HTML technology, the web pages are becoming more identical to standalone applications. Without the usage of these methods, the architecture of the application would be highly coupled, increasing the complexity of the application, because of the direct connection between the test classes and the GUI. So these patterns help to reduce the complexity of the application, creating another layer that is responsible for encapsulating the details of the components of the UI. With the use of VOs we also have the advantage of achieving headless testing. This is possible because of the use of mocked variables and methods that simulates operations of the UI. Thus, the UI is not displayed and all operations occur without instantiating it. Developing the application, we needed to structure and organize the modules in order to abstract some concepts of the application. We could notice that the important thing on the development of the application is the separation of the application logic and the UI. Thus, we can manipulate and implement properly the VOs. A limitation of the method can be complexity of a VO class implemented: the complexity can be high, depending on the number of buttons that we can act with.

Because BDD might be a powerful technique when combined with VOs, it was described in detail in order to give a better idea of this technique. Four main characteristics of BDD bring consistent test cases, being them: an ubiquitous language (the terms should be globally used in the project), iterative decomposition process for requirements gathering, plain text description with user story and scenario templates (as a standard for creation of the file features), and finally automated acceptance testing with mapping rules (i.e. scenarios being run automatically and being mapped to test code). So in BDD, the tests are written first, and they have the behaviour description of the system as a goal to be achieved. The Cucumber Tool is used with the help of Gherkin language to make the translation from a plain text (scenarios) to test code.

We could see two different types of application and we worked on their peculiarities. If it is a complex application, delegation of their actions to smaller VOs have to be considered when modelling and implementing the VOs. Therefore, as long as VOs can be applied for desktop applications, we could then identify an instance for the POP.

For a future work, an implementation of a plugin is intended to be developed in order to automate this method for desktop applications.

REFERÊNCIAS

- Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.
- Kent Beck and Martin Fowler. *Planning extreme programming*. Addison-Wesley Professional, 2001.
- Nigel Bevan. Human-computer interaction standards. *Advances in Human Factors Ergonomics*, 20:885–885, 1995.
- Eric Evans. *Domain-driven design: tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.
- Susan Hammond and David Umphress. Test driven development: the state of the practice. In *Proceedings of the 50th Annual Southeast Regional Conference*, pages 158–163. ACM, 2012.
- David Janzen and Hossein Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *Computer*, (9):43–50, 2005.
- Christian Johansen. *Test-driven JavaScript development*. Addison-Wesley Professional, 2010.
- Maurizio Leotta, Diego Clerissi, Filippo Ricca, and Cristiano Spadaro. Improving test suites maintainability with the page object pattern: An industrial case study. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 108–113. IEEE, 2013.
- Dan North et al. Introducing bdd. *Better Software*, March, 2006.
- Carlos Solís and Xiaofeng Wang. A study of the characteristics of behaviour driven development. In *Software Engineering and Advanced Applications (SEAA), 2011 37th EUROMICRO Conference on*, pages 383–387. IEEE, 2011.
- Bing Yu, Lei Ma, and Cheng Zhang. Incremental web application testing using page object. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 1–6. IEEE, 2015.