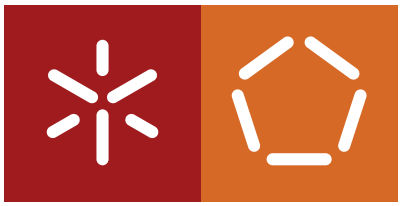
Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marina Machado

MODUS
Generation of Interfaces based on Models

October 2015



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Marina Machado

MODUS
Generation of Interfaces based on Models

Master dissertation
Master Degree in Computing Engineering

Dissertation supervised by
Supervisor José Creissac Campos
Co-supervisor Rui Couto

October 2015

Universidade do Minho

Declaração RepositóriUM: Dissertação de Mestrado

Nome: Marina Machado

Nº Cartão Cidadão /BI: 14179404 Tel./Telem.: 912704099

Correio eletrónico: angel.yorda@gmail.com

Curso : Engenharia Informática Ano de conclusão da dissertação: 2015

Área de Especialização: Informática

Escola de Engenharia, Departamento/Centro: Departamento de Informática

TÍTULO DISSERTAÇÃO/TRABALHO DE PROJECTO:

Título em PT : MODUS – Geração de Interfaces baseadas em Modelos

Título em EN : MODUS – Generation of Interfaces based on Models

Orientadores José Francisco Creissac Campos

Rui Miguel Silva Couto

Declaro sob compromisso de honra que a dissertação/trabalho de projeto agora entregue corresponde à que foi aprovada pelo júri constituído pela Universidade do Minho.

Declaro que concedo à Universidade do Minho e aos seus agentes uma licença não-exclusiva para arquivar e tornar acessível, nomeadamente através do seu repositório institucional, nas condições abaixo indicadas, a minha dissertação/trabalho de projeto, em suporte digital.

Concordo que a minha dissertação/trabalho de projeto seja colocada no repositório da Universidade do Minho com o seguinte estatuto (assinale um):

1. ☒ Disponibilização imediata do trabalho para acesso universal;
2. ☐ Disponibilização do trabalho para acesso exclusivo na Universidade do Minho durante o período de

   ☐ 1 ano, ☐ 2 anos ou ☐ 3 anos, sendo que após o tempo assinalado autorizo o acesso universal.
3. ☐ Disponibilização do trabalho de acordo com o **Despacho RT-98/2010 c)** (embargo___ anos)

Braga/Guimarães, 27 / 10 / 2015

Assinatura: Marina Machado

Acknowledgements

With a few words, I wish to express my gratitude to everyone who helped and motivated me throughout the completion of this thesis.

- To my supervisor, José Creissac Campos, and co-supervisor, Rui Couto, who always helped me with both constructive criticism and friendly advices. For their availability during the time passed and the share of their honest opinions on the issues faced.
- To Carlos, Damien, Dário, David, Hugo, Miguel, Nuno, Paulo, Ramon and Tiago who contributed to my project. For their precious help without which I couldn't get the results I needed.
- To my family who kept on motivating me during the thesis completion. For the strength they gave me to face all challenges, and all their love and caring.
- To all my friends who always gave me a smile when I most needed it and supported me all the way throughout this year.

My most sincere and deepest gratitude.

Abstract

Nowadays the user interface is essential for making a successful application. However, implementing a well thought interface requires a lot of time and effort. Advances have been made for user interface development tools, but they require the explicit identification of every interface element, not solving the issue.

Model-based methodologies have been proposed as a possible solution. To decrease development costs, they reuse some artefacts, in this case models, of the application's development to automatically generate the user interface. The generation process should, however, have a high level of automation to efficiently resolve the problem. Yet, in model-based approaches, this requires the use of detailed interface models, leading to an irregular interface development process.

Following that concept, this dissertation explored an automatic creation process directly based on structural models of the business logic, based on two main premises. First, for specific domains, the interface can be generated from a description of the application data. The identification of the domain replaces the need for including the interface models. Second, the description is to be made in Unified Modeling Language, known as UML, where the class diagram plays a key role.

The objective was to develop an *Eclipse* plugin ***MODUS*** (MOdel-based Developed User Systems) to support the approach. It allows the generation of the application's user interface from its UML descriptions. Unlike other existing tools, *MODUS* promotes the iterative refinement of the generated interfaces by exploiting the separation between the content and form of the interface. Thus leading to the particular interest in browser based Web applications.

Once the plugin was built, tests were conducted to conclude about the viability of the methodology. On the one hand, a survey based analysis was performed to study both validity and efficiency of the *MODUS* approach itself. On the other hand, a case study was used to evaluate the generated user interfaces, examining the degree of correctness, precision and completion that can be achieved using the approach.

Keywords: Automated user interface design, Model-driven engineering, Model-Based User Interface Development

Resumo

Atualmente a interface de utilizador é essencial para o sucesso de uma aplicação, no entanto a implementação de uma interface ponderada e correta requer muito tempo e esforço. Avanços foram feitos ao nível das ferramentas de desenvolvimento de interfaces, contudo exigem a identificação explícita de cada elemento da interface, não resolvendo o problema.

Metodologias baseadas em modelos têm sido propostas como uma solução possível. Para diminuir os custos de desenvolvimento, reutilizam alguns artefactos do desenvolvimento da aplicação, neste caso modelos, para gerar automaticamente a interface do utilizador. O processo de geração deve, no entanto, possuir um elevado nível de automação para solucionar eficientemente o problema. Ora, nas abordagens baseadas em modelos seria necessário utilizar modelos de interface detalhados, conduzindo a um processo irregular do desenvolvimento da interface.

Seguindo essa ideologia, a dissertação explorou um processo de criação automático focado diretamente em modelos estruturais da lógica de negócio, baseado em duas premissas principais. Em primeiro lugar, para domínios específicos, a interface pode ser gerada a partir de uma descrição dos dados da aplicação. A identificação do domínio substitui a necessidade de incluir os modelos de interface. Em segundo lugar, a descrição deve ser feita em *Unified Modeling Language*, conhecido por UML, onde o diagrama de classe desempenha um papel fundamental.

O objetivo foi desenvolver um *plugin* do *Eclipse* ***MODUS*** (*MOdel-based Developed User Systems*) o qual sustenta a abordagem, permitindo a geração da interface apropriada a partir das descrições UML da aplicação. Ao contrário de outras ferramentas existentes, *MODUS* promove o refinamento iterativo das interfaces geradas, tirando partido da separação entre o conteúdo e a forma da interface. Apresentando assim um interesse particular em aplicações Web baseadas em *browsers*.

Uma vez o *plugin* estabelecido, foram realizados testes de forma a concluir sobre a viabilidade da metodologia. Por um lado, uma análise baseada num inquérito foi estabelecida para estudar a validade e a eficácia da abordagem *MODUS*. Por outro lado, um caso de estudo foi usado para avaliar as interfaces de utilizador geradas, examinando o grau de correção, precisão e cumprimento que pode ser conseguido utilizando a abordagem.

Palavras Chaves: Desenvolvimento automático de interfaces, *Model-driven engineering*, *Model-Based User Interface Development*

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

AUI Abstract User Interface. 12, 14

CRUD Create Read Update Delete. 2, 3, 13, 15

CSS Cascading Style Sheets. 8, 26, 32, 39, 40, 41, 46, 50, 53, 54, 71, 73, 76, 79, 81, 82

CUI Concrete User Interface. 12

DOM Document Object Model. 46, 47

DTD Document Type Definition. 40, 41

FUI Final User Interface. 12, 14, 25, 30, 31, 32, 33, 45, 48, 49, 50, 53, 54, 64, 70, 71

GUI Graphical User Interface. 1, 8, 45, 54

HTML HyperText Markup Language. 7, 45, 46, 48, 49, 50, 59, 71, 72, 73, 75, 76, 81

IDE Integrated Development Environment. 34, 44, 52, 55, 57, 75, 80

IFML Interaction Flow Modeling Language. 28, 82

MBUID Model-Based User Interface Development. 3, 11, 12, 13, 14, 15, 23

MDE Model Driven Engineering. 3, 4, 9, 10, 11, 13, 15

PaaS Platform as a System. 15

PSSUQ Post-Study System Usability Questionnaire. 67, 78

UI User Interface. 1, 2, 3, 4, 6, 7, 8, 9, 11, 12, 13, 14, 15, 17, 24, 25, 26, 28, 29, 30, 31, 32, 33, 41, 44, 49, 50, 52, 53, 55, 59, 66, 67, 70, 71, 73, 76, 78

UML Unified Modeling Language. 3, 4, 5, 24, 28, 33, 36, 41, 50, 52, 62, 63, 66, 80

WUI Web User Interface. 7, 8, 9, 16, 24

WYSIWYG What You See Is What You Get. 3, 8, 15

XMI XML Metadata Interchange. 33

XML EXtensible Markup Language. 33, 38, 39, 41, 44, 50

Chapter 1
Introduction

> ”Design can be art. Design can be aesthetics. Design is so simple, that’s why it is so complicated.”
>
> Paul Rand

1.1 Context

The first computers created were meant to complete difficult tasks in an efficient way. The user interface was not a relevant asset, as they were used by trained technicians. Since their appearance a lot of time has passed. From mechanical switches, punch cards, command lines and the first Graphical User Interface (GUI) created, computers suffered a huge evolution. Nowadays, not only computers, but technology in general, is accessible to a diversified audience, from different cultures, professions, economic backgrounds, and so on. People are accustomed to using technology in their daily life.

The User Interface (UI) encompasses all aspects which allow the users to relate to an application. They can see, listen and even touch the user interface, depending on it to communicate with the application [Myers et al., 2000]. The UI became one of the decisive factors of the future of any application, would it be Desktop, Mobile, Web or even Hybrid. Therefore, it should better reflect human-machine communications, enabling a more comfortable interaction.

Studies carried out in the 1980’s and 1990’s have shown the that almost 50% of the development effort was dedicated to the UI [Mittal et al., 1986; Bobrow et al., 1986; Myers and Rosson, 1992]. User interfaces have kept increasing in complexity and features, requiring innovative development solutions. These new challenges hamper the interface creation process. In fact a huge amount of time is still spent in designing, implementing and maintaining the UI. Moreover, web applications pose a particular challenge given their distributed, dynamic and heterogeneous nature, respectively, in terms of browser and server code architecture, code generation (which involves interface aspects such as responsive design) and of programming language for the code implementation.

A lot of effort was invested in the creation of user interface development tools, in order to accelerate and facilitate the generation process. Currently, a rather diversified set of tools

exist. The UI development tools can be classified in many ways, according to their methodology, features, among others. However, in most of those categories, the tools need a detailed specification of the user interface elements. Generation processes based exclusively on these approaches remains still a time consuming process [Myers, 1992, 1994b; Myers et al., 2000; Kennard and Steele, 2008].

A model-based approach may be the key to create the application's UI in a fast and efficient way. Model-based tools stand out by reducing the time spent in the implementation of user interfaces. They use models, created in the first steps of development, to build the UI [Myers, 1992, 1994b; Myers et al., 2000], thus reusing previous development efforts. Above all, these tools allow a certain degree of automation in the generation process, as seen in Model-Driven Engineering [Mohagheghi and Dehlen, 2008].

1.2 Motivation

The user interface is an essential asset of the application. Nonetheless, spending half of the software development time in the UI is not a viable solution [Mittal et al., 1986; Bobrow et al., 1986; Myers and Rosson, 1992]. Graphical Designers should dedicate their attention to building a creative and unique design for the application. Instead, they tend to spend large amounts of time repeatedly building the same layout for each UI they develop. Any form of automation will be profitable, and model-based tools do provide a certain degree of automation. The inclusion of automation is even more relevant if it leads to the creation of high quality user interfaces. Programmers, despite their small experience or little knowledge in user interface design, could rely on the system's implementation to build strong foundations for the UI, supporting a usable way of communicating with the user [Myers, 1992, 1994b; Myers et al., 2000].

Despite their unique advantages, model-based tools did not become a standard in the generation of user interfaces. *UIDE* [de Baar et al., 1992], *HUMANOID* [Szekely et al., 1993] and *MECANO* [Puerta et al., 1994] are examples of highly automated tools, focused on the creation of applications with a Create Read Update Delete (CRUD) based interface. The UI created with these tools are, however, limited in terms of *Look & Feel*, being composed of simple widgets, and restricted to basic operations of insertion, edition and so on. Due to the level of automation and the generalization of the creation process, the user interfaces end up following the same structure, leaving little space for diversity. In general, model-based tools with a high degree of automation are criticized for having a weak integration with the

software development process. First, these tools tend to have problems in integrating the creative process of the *designers*. This usually has a negative impact on the quality of the UI [Molina, 2004]. Second, the models used tend to diverge from the business logic models used in model driven approaches, raising coordination and productivity problems [Meixner et al., 2011; Meixner and Calvary, 2014]. Some model-based tools try to address these issues by lowering their level of automation, such as *MARIAE* [Paterno, 1999] and *TERESA* [Berti et al., 2004]. In these tools, the developer has to manually add auxiliary information about the models to compensate the loss in automation. The process of designing the models becomes even more complex and time consuming [Meixner et al., 2011].

Over time, the model-based tools have evolved into platforms developed in the software industry. In other words, tools adopting some of the more established concepts of this paradigm were created. Nonetheless, these tools were intended to satisfy specific needs of the developers. Examples are *Outsystems* [Resources, 2011] and *Integranova* Solutions [2005]. *Outsystems* aims to increase the productivity of the developers. Though it can create user interfaces from a database model, the level of automation may be low, being the UI built within a What You See Is What You Get (WYSIWYG) editor. The *Integranova* platform aims the creation of all layers of the application: database, business and presentation. It generates a CRUD based UI with a high level of automation, with no possibility of improvement during the user interface creation.

A model-based approach that could balance the level of automation with the appeal of the UI could be a solution for the development costs problem. The system models should be the basis for the automation process, assuring solid foundations for the user interface. The intervention of the developer shouldn't impact the degree of automation. Instead, it should benefit the quality of the UI by integrating the designer's creative mind.

1.3 Objectives

This dissertation presents an approach for the automated generation of user interfaces. Supported by the principles of Model Driven Engineering (MDE) and Model-Based User Interface Development (MBUID), the proposal is to create the UI by resorting to the system's Unified Modeling Language (UML) descriptions. A high level of automation is assured by complementing the model with the identification of the application domain.

The dissertation approach is supported by the "MODUS" (MOdel-based Developed User Systems) tool, implemented as an *Eclipse* [Foundation, 2001] plugin. In the context of model-

based user interface generation, the approach proposes the following list of contributions:

- By focusing on browser based Web applications, it exploits the separation between the definition of the content and form of the user interface. Such exploitation enables the generation UI ”skeletons”, which can be personalized by designers. Therefore, promoting the iterative refinement of the generated interfaces. Since the goal of the approach is focused on developing an interface independently of any technology of the business logic layer, this separation can allow future integration with back end implementation.

- It is centered around one structural model of the system, a class diagram in UML [Fowler, 2003]. By definition, the class diagram states all classes and respective relations required to develop the system, being essential in a MDE context. This model is a suitable choice for this approach, since its composing entities are decisive in the application's content, being specific to it.

- Defining the application domain beforehand limits the need to resort to additional models. This enables a substantially more automated generation process. Indeed, for the same domain, it can be observed that user interfaces tend to be similar in navigation, structure and visual components.

- The concept of *Evolutionary Prototyping* [Davis, 1992] plays a key role in the methodology. The users of the tool can manipulate the result at any stage of the generation process, hence refining the final interface. Due to its iterative nature, the generated interface can be further manipulated, in order to comply with a greater number of user requirements.

- It supports the definition of a number of interface appearance details, such as: front end frameworks, responsive design for different devices, templates, among others. This way, the user has greater control over the visual appearance of the final result.

1.4 Document Structure

The thesis document is composed of the following chapters:

Chapter 1, the current one, is an introductory chapter. It was dedicated to presenting the context, motivation and objectives of the theme, thus introducing the main premises of the dissertation.

Chapter 2 explores the model-based approach in the field of user interface generation. The chapter presents all relevant concepts of automatic generation based on models, serving as the foundation of the proposed approach. It identifies some important tools fitting the context of this work, providing some concrete examples. Besides reviewing model-based approaches, it studies the application domain as a categorization for web applications.

Chapter 3 presents the *MODUS* conceptual process. It defines each step and procedure that compose the approach, ideally analyzing both advantages and disadvantages within the context of the generation of user interfaces. This conceptual approach will allow to establish the main lines for the development of a reference implementation.

Chapter 4 features the implementation of the *MODUS* prototype. By developing each stage of the approach, this chapter allows to validate the choices established in the Chapter 3. Concretely, it specifies what data formats to use, how to extract relevant information, manipulate it and generate the output.

Chapter 5 covers simulations on the *MODUS* prototype from a case study example. This chapter presents, step by step, the instructions on how to obtain the desired user interface from a UML model.

Chapter 6 tests the *MODUS* prototype, through the analysis of a case study example and a survey. With the results obtained, it studies the impact of the automatic generation of interfaces based on a singular system model. This chapter mainly intends to validate the approach, discussing the contributions of the *MODUS* approach in creating a consistent and correct user interface.

Finally, Chapter 7 debates the impact of automatic generation of interfaces based on models. This chapter discusses the obtained results, withdraw conclusions, and explore future work.

Chapter 2
User Interfaces

> "I have this hope that there is a better way. Higher-level tools that actually let you see the structure of the software more clearly will be of tremendous value."
>
> Guido van Rossum

The user interface of an application manages the output displayed to the user as well as the user's input. It includes everything that is designed into the software to enable the interaction between the person and the machine. Which means every interface object that can be seen, heard or touched trough the application. Being the only means of communication with the user, it becomes one of the most important aspects of the application. Programs with a poor UI are difficult to operate, learn, remember, hence failing in human interaction. Thus a good application should be supported by a strong user interface.

Building a strong UI, in other words a usable, responsive and beautiful interface, is a complex task. Despite the existence of usability design principles, such as [Nielsen, 1993], there is no standard technique that will guarantee the success of a user interface. Furthermore, as the time passes, the user interfaces become easier to use and understand, being more complex to build. They are complemented with an increasing number of elements to provide a more comfortable interaction with the user, such as run-time validation or help messages in UI forms [Myers, 1994a; Cerny et al., 2012]. Besides, not only creating the application's UI is a difficult task, it is also a long process. Studies have showed that the average time dedicated to the user interface can reach near 50% of the application's code development time [Schlungbaum and Elwert, 1996; Kennard and Steele, 2008; Cerny et al., 2012]. Consequently, it is not surprising that developers wished to accelerate and facilitate UI development.

2.1 User Interface Generation Tools

Researchers worked hard to find a solution for an efficient UI development, inventing the user interface generation tools. As the name suggests, these tools create or help in the creation of the application's interface. Their main goal is to make the process of developing user interfaces easier, faster and cheaper, while ensuring the quality of the UI produced. Although each user interface is unique in terms of graphical details, such as color patterns,

visual design, among others, in general (either on different platforms or devices) they tend to follow patterns, having a familiar appearance and a similar mode of operation. Due to this homogeneity, user interface generation tools were able to refine their work, reaching a high level of sophistication.

Different layers of UI software components exist to create the user interface of an application. Understanding these layers is essential to debate what the advantages and disadvantages of the various levels of abstraction of the user interface tools. Note that, in this dissertation, there is a particular interest for the generation of user interfaces for web applications. Figure 2.1 maps the generic architecture proposed by [Myers, 1994b] to the specific case of Web User Interface (WUI)'s. The establishment of this bridge helps to ponder on how to possibly integrate the layers and define the most efficient methods of development.

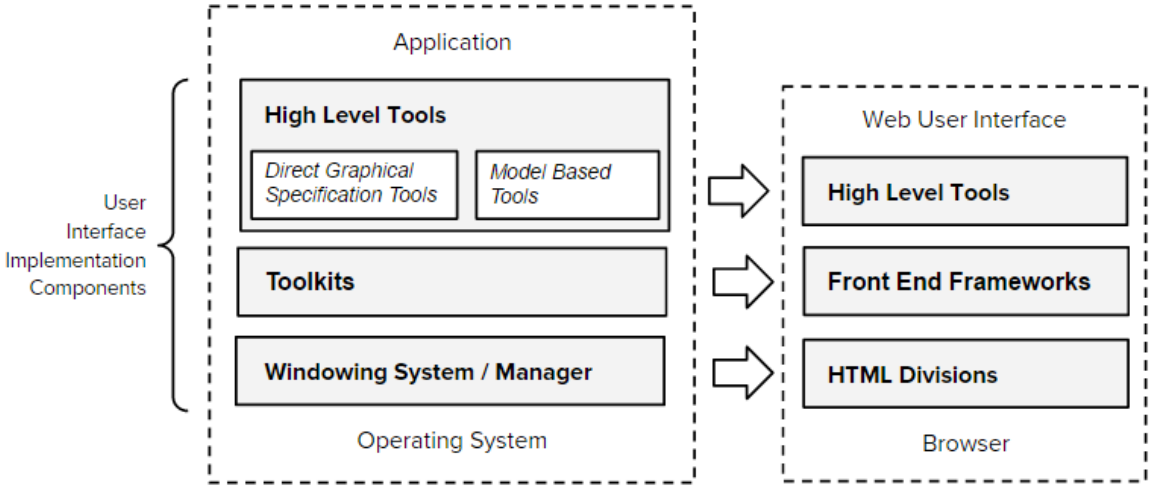


Figure 2.1: Bridge between the representation of the UI Software Components (based on Myers [1994b]) and the web interface development

Window Managers [Myers, 1992, 1994b; Myers et al., 2000] provide a basic programming model for the creation of the display screen's output, as well as for the recognition of the user's input. A **Windowing System** divides processes into a set of distinct regions of the screen, commonly known as windows. Creating interfaces at this level requires that every interface component must be built from scratch. Not only can this lead to various inconsistencies throughout the interface, it reveals to be a slow and annoying process [ROSENTHAL, 1987]. The development of a WUI does not offer direct support for window management. This can however be achieved by explicitly addressing the interface divisions as if they were equivalent to windows. These divisions are displayed by the browser from interpreting the

HyperText Markup Language (HTML) files. Each division can be defined either as a single page, a part of a page or even an HTML tag defining a specific area of the page, depending on the degree of intended detail. This approach does not solve the main problem, since it still leads to inconsistency and development delays in the generation of the UI.

Toolkits [Myers, 1992, 1994b; Myers et al., 2000] provide libraries of widgets[1]. They establish a framework which allows the manipulation of their interactions with the user. Since the UI components are predefined, they only require instantiation. The creation process of the user interface becomes easier and quicker. However, being dependent on the toolkit framework, the UI is restricted to the widget's library. In the development of a WUI there are a lot of front end framework and toolkits which impact the generation of the interface, such as the *Bootstrap* Cascading Style Sheets (CSS) framework [Otto, 2011]. Resorting to these front end helpers to create the WUI is not an optimal solution as a standalone. Nevertheless, it could reveal to be helpful as a complementary component of the generation tool.. A front end framework would be responsible to define the style of all widgets and other interface elements, assuring the overall consistency of the WUI.

As can be understood, programming at these levels reveals to be a humongous and difficult task. This contributes to the idea that high-level tools are, without a doubt, an asset in UI development. Different categories of high-level tools have been created to help develop user interfaces, each with its own specific approach. They can be classified in many ways, based on the techniques they use, the paradigm they belong to or even the style of interfaces they create, among others. Taking into account the context of this dissertation, the user interface generation tools are categorized into two main groups, model based tools and direct graphical specification tools [Myers, 1992, 1994b; Myers et al., 2000]:

- **Direct graphical specification tools** include prototyping tools, data visualization tools, editors for application-specific graphics and interface builders such as the Netbeans GUI builder [Oracle, 2011]. With these tools, the UI is partially or fully created by placing objects directly on canvas, by dragging them with a pointing device. The generation process is long and repetitive, leaving very little margin for automation whatsoever. Furthermore, they do not provide any guidance on creating good and strong UI's. One of the most predominant subcategory of the direct graphical specification tools are user interface builders. The interface builders provide a "drag and drop" WYSIWYG editor to generate the interface. One of the biggest advantages of using any interface

[1] A widget is an interactive component of a GUI responsible to display the input data of the user, such as menus, buttons and text fields.

builders, whether they are intended for web design or not, is that the developer can observe the final result while operating the tool. When developing code, not always "what you write is what you get". Relevant examples of interface builders in the development of WUI are: *Adobe Dreamweaver* and *LayoutIt*. *Adobe Dreamweaver* [Macromedia, 2012] is a web development tool featuring a code editor, thus allowing the programmer to assure the readability and cleanliness of the code. *LayoutIt* [Katz et al., 2015] is an interface builder supported by a CSS framework, proving that different software components can be united to create an improved tool. Once again, it is shown that uniting different approaches can converge to a better solution.

- **Model Based Tools** use high level specification models of the system to generate the user interface. Model-based tools are mostly known for trying to reduce the UI development costs, which is a recurrent problem found in other tools, by integrating automation in their creation process. They can resort to various numbers and variety of models, have distinct levels of automation, build different types of user interfaces, among others. Common known examples of model based tools, which can be used for the development of WUI, are *USIXML* [Limbourg et al., 2004] and *TERESA* [Berti et al., 2004]. Compared to the other categories of tools, they appear to be a valid solution for the fast development of high quality user interfaces [Meixner et al., 2011].

2.2 Models and User Interfaces

Models are often associated with the concepts of "example to be followed", "pattern" and "guideline". Their purpose is to simulate an entity, helping its creator to understand and analyze it. Herbert Stachowiak [Stachowiak, 1973] characterized that models must have the following properties:

- **Mapping**: models are based on an original, the subject. They are representations or mappings of something imaginary, which may or may not be built at some point.
- **Reduction**: models must not mirror all the properties of the subject, but the relevant attributes in the context.
- **Pragmatism**: models should replace the subject in order to fulfill some purpose.

In the field of software engineering, software models help in the study and design of programs. They allow the engineers to analyze the system before starting to develop, ide-

ally realizing the specification that will lead to the final solution. However, these models are usually created from a train of thought, which isn't always straight forward. Some models may become overspecified, others underspecified, and so on. Models are dependent on their creator, the specifier. Acknowledging this fact, model quality can be obtained from the reformulation of the *Seven Sins of the Specifier* [Meyer, 1985]: noise, silence, contradiction, overspecification, ambiguity, forward reference and wishful thinking . By not committing the specifications sins, models could be ensured to be correct. Although a higher experience reduces the likelihood of committing mistakes, human actions may not meet the expected quality requirements. Despite this issue, models demonstrated to be a powerful resource to help create but also to manage large and complex systems. Over time, the use of models as primary elements in the development of applications has increased, and for many became a habit. This has contributed to the establishment of the model-driven software engineering paradigm, frequently known as MDE [Stahl et al., 2006; Rech and Bunse, 2008].

2.2.1 Model Driven Engineering

In MDE, high-level models are converted into lower-level models with the purpose of converting them into running systems [Stahl et al., 2006; Rech and Bunse, 2008]. A v-model of the systems engineering process [Osborne et al., 2005] was adapted to MDE in Figure 2.2. The transformation process of the models can be either automated or manual. Using models ensures consistency in software development, ensuring a better quality and correction of the final result. Above all, resorting to an automatic transformation of the models, a reduction of development time can be ensured [Meixner and Calvary, 2014; Schlungbaum, 1996].

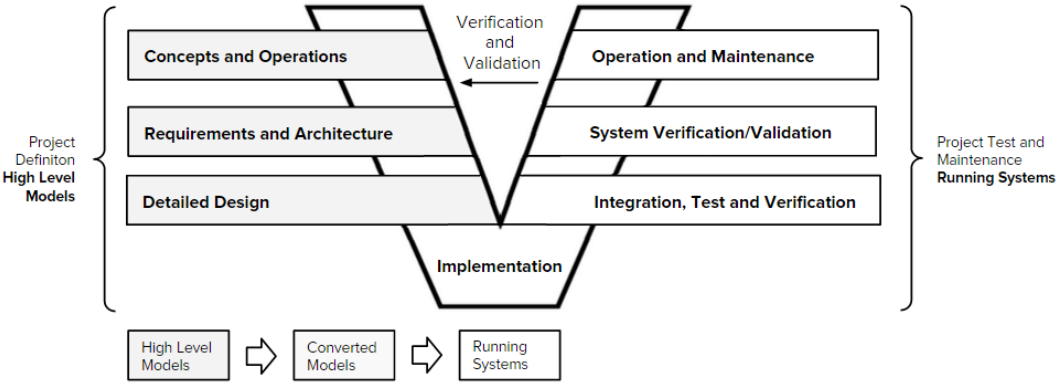


Figure 2.2: V-model of the Systems Engineering Process [Osborne et al., 2005] adapted to MDE

In order to create high level models, teams and individuals (managers, developers and designers) go through an extensive communication about the desired solution. This process isolates the business logic from the technology and other implementation mechanisms, being the models either representations of the problem or possible implementations of the solution. The level of abstraction can be raised, which eases the handling of the inherent complexity of the system by using standardized models. Compatibility between systems is increased, allowing a harmonization of past, present and future technologies [Kidwell, 1996; Hailpern and Tarr, 2006].

MDE is a powerful paradigm with proven results. Nonetheless, it is a target of some criticisms [Mohagheghi and Dehlen, 2008]. The developers tend to have issues with the models themselves. Discovering the exact type of models, and from those what amount of models is required and in what degree of specification can be challenging. A norm established beforehand would help to solve this problem. The developers would know what models they need in order to specify the problem and find the solution. However, to be efficient, it is necessary to ensure the quality of the models, from the higher to lower levels. As presented earlier, the quality of the models depends on the creator. Introducing automation to the model transformation process would assure the quality of the intermediate models, by removing the variable that may lead to more inconsistencies and errors, the specifier. This would guarantee a final result with better quality and less errors.

From the above discussion, one can deduce that MDE is a suitable paradigm for an efficient and sustainable software development. In the software engineering community, the approach is typically applied to the creation of the business logic and data layers [Kidwell, 1996; Hailpern and Tarr, 2006] of the applications. The models developed in this context could be reused to generate the UI, which is one of the premises of the proposed approach.

2.2.2 Model-Based User Interface Development

MBUID can be seen as the MDE of user interfaces. It uses high-level abstract declarative models as the foundation for the creation of the UI [Schlungbaum, 1996; Da Silva, 2001; Meixner and Calvary, 2014]. These models (domain model, task model or both) expose the application's problem. Besides them, some auxiliary interface models can be added to help describe the user interface itself, the system to be developed or other aspects related to either.

The *Cameleon Reference Framework* is widely accepted as the standard architecture for MBUID. It specifies four main levels of modeling, represented in Figure 2.3 [Calvary et al., 2003; Vanderdonckt, 2005].

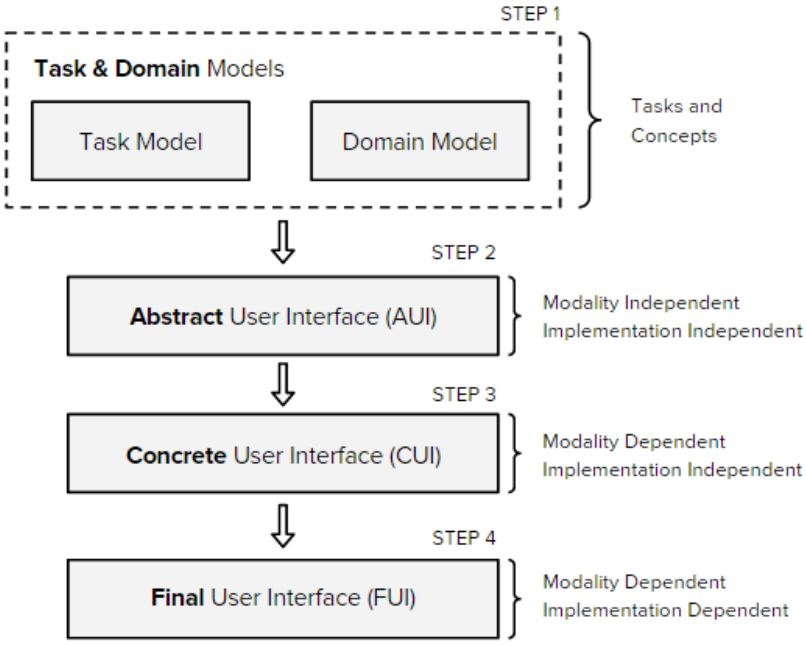


Figure 2.3: The 4 Steps of the MBUID approach

1. **Task and Domain Models** - description of the end user tasks and domain concepts, related to their accomplishment.

2. **Abstract User Interface (AUI)** - description of the user interface in terms of *Abstract Interaction Units* or *Abstract Interaction Objects* and their respective relationships [Vanderdonckt and Bodart, 1993]. Being the objects abstractions, the UI is represented independently of any technology and modality.

3. **Concrete User Interface (CUI)** - description of the user interface in terms of *Concrete Interaction Units* or *Concrete Interaction Objects* [Vanderdonckt and Bodart, 1993], determining the layout and the navigation of the interface. CUI are modality dependent and describe the *Look & Feel* of the user interface.

4. **Final User Interface (FUI)** - description of the user interface in terms of source code. The source code can be in either a programming or a mark-up language. The running user interface can be interpreted or executed after code compilation.

The MBUID approach aims at development of user interfaces, in which the model is an abstraction of the UI. More specifically, it intends to reduce the amount of time and effort spent in the development of the user interfaces, while assuring their quality [Meixner and

Calvary, 2014]. Yet, the user interface is somewhat heterogeneous, evolving through time and cultures. It can be interpreted in different ways, depending on the computing platforms, working environments, end users and programming languages. An MBUID generated UI is generally criticized for having a poor life cycle and weak support for different platforms and devices [Meixner et al., 2011; Meixner and Calvary, 2014]. A possible solution would be complementing the MBUID approach with portability resources, such as front end frameworks. However, the quality issue does not end here. The automation tends to have a negative impact on the quality of the user interface. This is mostly due to the lack of integration with the designer and the "low ceiling" [2] of the tool [Molina, 2004]. Balancing the level of automation with the developer intervention could promote the iterative refinement of the interface, thus improving the quality of the UI produced.

MBUID is a similar methodology to MDE aiming the creation of user interfaces. As with MDE, the generation process is based on high-level models. Nevertheless, in MBUID it is common to use detailed models to define different aspects of the UI. This dependency to interface models limits the acceptance of the approach, since there is an increased cost of modeling, whether it is in number of models or in the development effort [Schlungbaum, 1996; Da Silva, 2001; Meixner and Calvary, 2014].

2.2.3 MBUID Development Tools

Over time, MBUID development tools changed in order to face the new challenges brought by the evolution of the user interfaces. Overall, four generations of MBUID can be identified [Meixner et al., 2011; Schlungbaum, 1996; Paterno et al., 2009].

The first generation of MBUID tools focused on generating *Desktop* applications based on CRUD operations. They emphasized the use of one universal declarative model, leaning on a fully automated process de Baar et al. [1992]; Szekely et al. [1993]; Puerta et al. [1994]. The more automated is the process, the less it is time-consuming. An example of the first generation tools is *MECANO* [Puerta et al., 1994]. Despite using interface models, in other words models that describe either the interface as a whole or one or more of its components, *MECANO* focuses on using domain models as the universal declarative model. It supports the reuse of domain models, since some applications will have the same modeling prototype with some minor changes. This fact contributes to the idea that common patterns can be found in the modeling of the same application domains. In *MECANO* the domain and interface mod-

[2] The restriction on what and how much the tool can do, limiting the variety of widgets that can be produced.

els are processed by the intelligent-designer tool, where an instance of the generic interface model is created. That instance (representation of the domain specific interface design) is generated by a series of mappings between the model's characteristics. Then, the run-time system generates the UI from a declarative language based specification of the instance. After the user interface is finished the user can improve some details. It is noted that, although the interface is generated automatically, the designer always has the possibility to apply some changes according to his creativity.

Both first and second generation tools are meant for Desktop applications. Like for the first, the second generation of MBUID tools tries to solve the problem, each with a different generic approach. By resorting to a more complex interaction with the developer, they manage to create a more flexible and specific user interface. These tools define the interface model as a composition of declarative models, models which describe how and what the application will accomplish to face the system's problem. Although supporting some automation, the generation of the FUI is a cumulative process. To increase the degree of automation it is necessary to include auxiliary information about the user interface. An example of this issue is *TADEUS* [Schlungbaum and Elwert, 1996]. This approach intends to generate complete user interfaces by resorting to a lot of specific interface models. The UI layout and dynamic behaviour is created from the dialogue model, dialogue graph and interaction tables, domain and task model. The generation process itself is composed of seven steps and is similar to *GENIUS* [Janssen et al., 1993]. The *GENIUS* approach [Janssen et al., 1993] is intended for database oriented applications. It uses mainly the data model to generate the UI, comparable to a first generation tool. However, being a second generation tool, it needs external data to create the user interface. In this case in the form of Entity Relationship diagrams and Dialogue nets. Still, the data model is not enough to structure the information around the user tasks. Views are defined, and for each view a window is created with a default layout. These default properties help improve the consistency of the UI, speeding up the generation. This demonstrates that resorting to default content created from the interpreted data can have an overall good impact on the produced user interface. At the end, the final layout is created with the proper arrangement of its different components.

As the time passes new interactive platforms and devices are invented, bringing new challenges in the field of the user interfaces. The third generation of MBUID tools is precisely dedicated to dealing with these constraints. *TERESA* [Berti et al., 2004] is a typical example of this generation. It uses high-level task models to generate the UI for different platforms/devices. To solve the new challenges placed by the need to deploy an application on multiple

platforms, the specification needs to address the multiple platforms of use, being context-dependent. The developer has to create and adapt the task model for the various platforms. Since *TERESA* is a web application oriented tool, it could resort to front end technologies, such as *Javascript* and *CSS3*. They could significantly increase compatibility between different devices and platforms during run-time [Sampaio and Campos, 2014]. The integration of these technologies may be an alternative to the context specification. The *TERESA* tool analyses the task relationships, from the task model, to create the AUI. From the abstract user interface and the target platform, the FUI is generated. This process features a range of automatic solutions, from complete to low automation. The different levels of interaction may allow to apply some highly relevant changes.

The fourth generation of MBUID tools focuses on two main issues: multipath development [Limbourg et al., 2004] (with a particular attention to mobile devices) and the integration with pre-existing web services. The approach proposed in this dissertation is largely orthogonal to these concerns. It focuses on providing an efficient solution for creating interfaces, taking as a starting point structural models of the business layer.

2.2.4 Related Model Based Tools

Although there are numerous MBUID tools, certain model-based approaches are suited for the context of this dissertation. Two in particular can be emphasized due to their relevant and distinctive features: the *Ruby on Rails* framework and the *Outsystems* platform.

Ruby on Rails [Team, 2005] is a full-stack open source framework for the quick creation of Ruby web applications. This framework uses the business model to generate a rough application with standard CRUD operations. It creates both front and back end implementations, complemented with a simple functional interface. *Ruby on Rails* is one of many model-based tools that can create a UI leaning only on one structural model. However, in this particular case, it guides itself by acknowledging that for each entity, a set of web pages will be necessary to fulfill the CRUD operations. Nonetheless *Ruby on Rails* is not enough to generate good user interfaces. Despite each view having the appropriate content, their layout is exactly the same for each web application created. Also, the *Look & Feel* of the generated user interface is very simple, not taking advantage of any front end resource.

The *Outsystems* platform [Resources, 2011] is a MDE compliant Platform as a System (PaaS) for implementing web applications. It produces not only the user interface, but also back end code for Java and .Net programming languages. This platform provides a set of tools to generate an overall fully functional application: tools for managing databases, gen-

erating back end implementation, creating user interfaces, among others. After adding some database information, the developer must specify the UI navigation map. He is able to set specific parameters, as for example the default view. In *Outsystems* the developer can select a view, which already contains some minimal information, and start building the desired user interface in the WYSIWYG editor. The developer can resort to a set of widgets generated from the interpreted data. However, he is not limited to the resources provided by the editor, being able to manually include additional design to the UI.

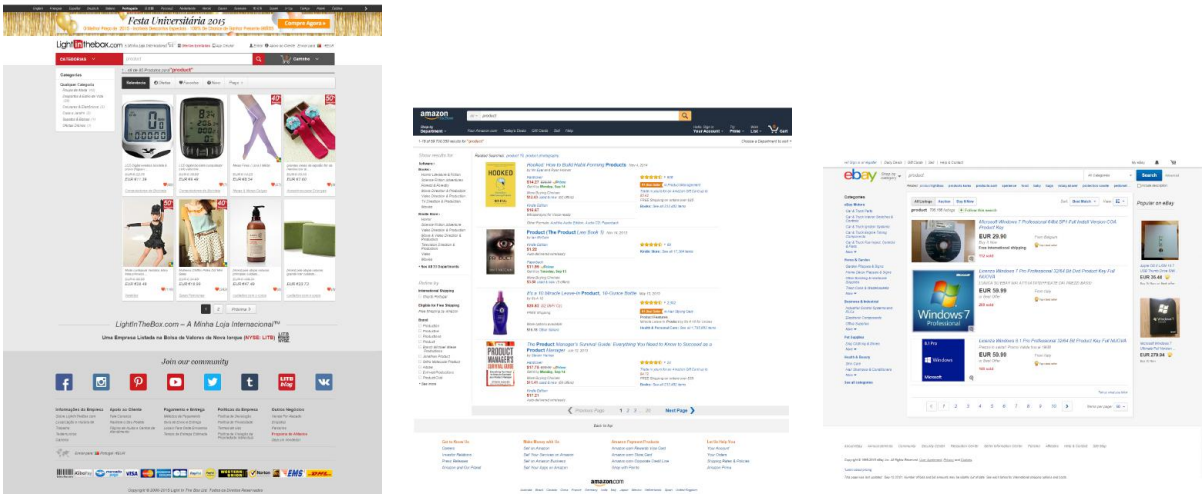
Analyzing these model-based perspectives was essential to define some of the main guidelines of the dissertation's approach. From *Ruby on Rails* one can deduce that a certain set of pages tends to exist for each entity of the application. It can also be extrapolated that some specific pages tend to exist for each application area. Yet, there is a need to include some methodology to increase the flexibility of the user interfaces. Furthermore, resorting to front end resources would take advantage of the separation between content and structure of a WUI. The *Outsystems* platform exploits the concept of a navigation map to define the set of views. Despite having to be laid out manually, these configurations facilitate the connections between views, being a significant advantage. Acknowledging the conclusions drawn on *Ruby on Rails*, one could assume that each domain view could be defined in a model, in terms of default content and navigation.

2.3 Application Domains in Web User Interfaces

An application domain defines a set of applications with a common set of features, requirements, objectives, among others. Retrospectively, the application domain is decisive in the content of any application, influencing many of the aspects related to user interface. From the interaction with the user, the displayed content, the navigation between pages and so on.

A key study to the achievement of the dissertation is understanding how application domains can influence the user interface of web applications. It can be observed that most web user interfaces belonging to the same application domain appear to follow patterns, whether in terms of content, navigation or structure. They are made out of various combinations of components, which are common among themselves. This can be seen in Figure 2.4, where different *eCommerce* web applications feature comparatively the same visual structure. Since the main components and interactions are the same, they should follow conventions. Even if a browser-based web application wanted to escape the structural standards, the users always look for a familiar interface. They should expect an output who matches previous situations

they've already faced. In other words, the user interface should be consistent [Tucker, 1997, p. 132-134].



(a) Light in the Box (b) Amazon (c) Ebay

Figure 2.4: Examples of compliant/similar websites (cropped printscreens)

Web applications, because of their variety, can be categorized in many ways, as shown in [Mcneil, 2008; McNeil, 2010]. In summary, the following forms of categorization can be considered:

Type Categorization by website's functionality or application area, such as e-commerce, forums, search.

Design Categorization by the website's visual aspect and *widget*'s style, such as modern, *sketchy*, *flat*.

Color Categorization by the website's dominant color or color pattern, such as blue, violet, orange.

Structure Categorization by the website's layout structure and its respective components, such as one-page, horizontal scrolling, hybrid.

From the categorization methods mentioned above, the application domain is directly related to the type categorization, defining the content and functionality displayed by the user interface. While the structure is dependent on the website's purpose, thus its application

domain, it is not enough to define the user interface as a whole. Both design and color can actually be affected by the application domain, however, they are dependent on temporal and cultural contexts. Above all, these aspects influence the uniqueness of the UI. It is more important to integrate them with the designer creative process than the actual domain. The type categorization could help identifying the main application domains related to web user interfaces.

Following the conducted analysis, one can identify two major facts. First, the application domain can be represented by the type categorization. Second, websites belonging to the same type tend to share similar interface components. In order to identify application domains one need to update the type categorization presented in [Mcneil, 2008; McNeil, 2010] to also consider the user interface itself. It was decided to select, within all possible types that can be identified, a small representative subset. This group will help to present groupings of browser-based applications with the same domain. Selecting only a subset was a personal choice made by the author, which believes that each of these types would get a higher contribution from the *MODUS* tool. Namely, because the chosen categories tend to have a stricter and more standardized structure, either in content and layout. Furthermore, this set unites different research areas of interest for the author. Again, it is worth noting the fact that the selected types depend on the temporal and social context in which the dissertation was written. The identification of each of the types was inspired by the books mentioned above [Mcneil, 2008; McNeil, 2010; Eccher, 2008], the study of the most popular websites and the analysis of the marketing template sites. The web application types identified can be described as follows:

The Forum Type Web applications dedicated to online discussion between users on a specific topic or set of topics.

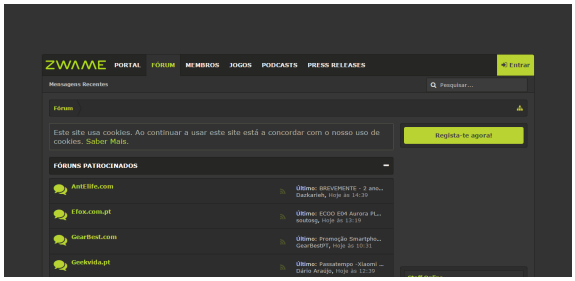
The Blogging/Magazine Type Web applications on which one or more users write about a specific concept, opinion or other.

The Advertisement/Product Placement Type - Web applications dedicated to the presentation and advertisement of a certain product, company, among others.

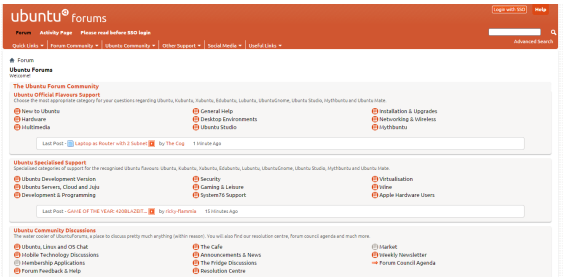
The *eCommerce* Type Sites dedicated to the trading of a specific product, group of products or even services.

2.3.1 The Forum Type

The Forum type is composed of web applications dedicated to online discussion. Each conversation is composed by a set of messages posted by the users. The conversation may be about a particular subject or group of subjects, depending on the application. The range of subjects may lead to a hierarchical structure, in which threads, conversations, are grouped in topics, which are themselves grouped in *subforums*. a "ZWAME"[3] in Figure 2.5a, "Ubuntu Forums"[4] in Figure 2.5b, "Minecraft Forum"[5] in Figure 2.5c, "W3C Forum"[6] in Figure 2.5d.

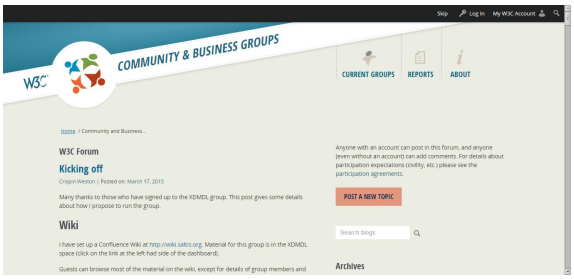


(a) ZWAME



(b) Ubuntu Forums

(c) Minecraft Forum



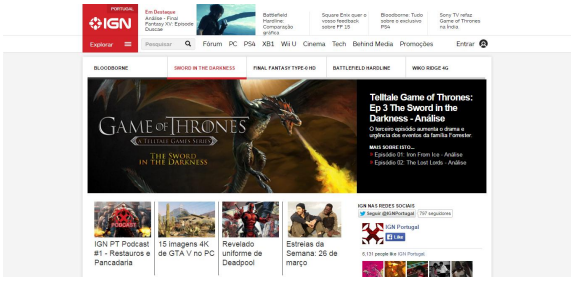
(d) W3C Forum

Figure 2.5: Examples of Forum web applications

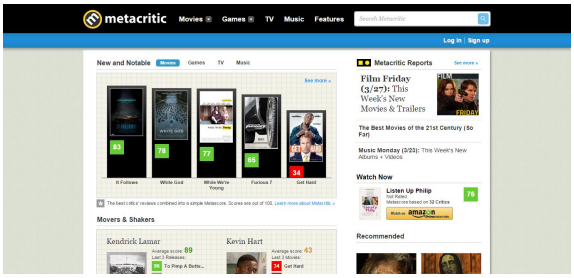
[3]https://forum.zwame.pt - last visited on March 17, 2015
[4]http://ubuntuforums.org - last visited on March 17, 2015
[5]http://www.minecraftforum.net - last visited on March 17, 2015
[6]https://www.w3.org/community/forum/ - last visited on March 17, 2015

2.3.2 The Blogging/Magazine Type

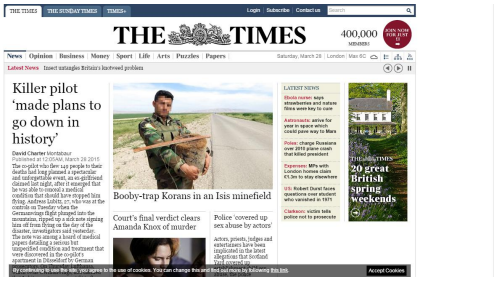
The Blogging/Magazine type is composed of web applications on which one or more users write about a specific concept, opinion or other, in the form of posted messages. The main topic of discussion/information may vary from personal matters, specific subject or area, branding of a company or an individual, editorials, among others. The messages themselves can have text, images, videos, being their content directly related to the topic they approach. In most cases the applications belonging to this type allow visitors or followers to comment on the published posts. The Blogging/Magazine type is similar to the Forum type, however, unlike it, it is focused on the publishing of new content. Some commonly known examples are: "IGN"[7] in Figure 2.6a, "Metacritic"[8] in Figure 2.6b, "The Times"[9] in Figure 2.6c, "How-To Geek"[10] in Figure 2.6d and so on.



(a) IGN



(b) Metacritic



(c) The Times

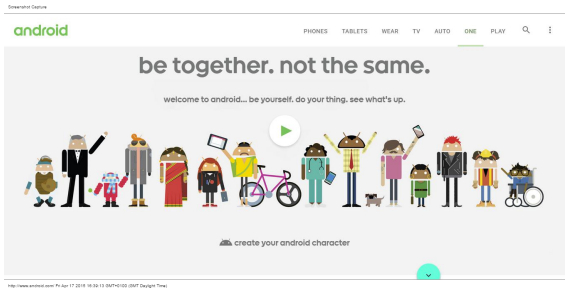
(d) How-To Geek

Figure 2.6: Examples of Blogging/Magazine web applications

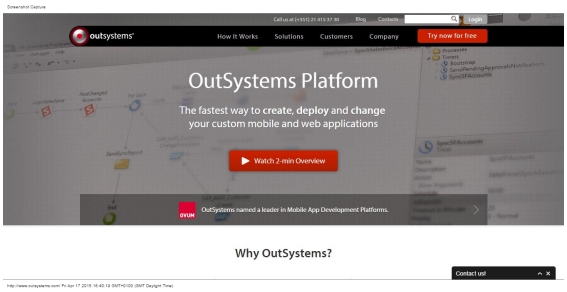
[7] http://pt.ign.com - last visited on March 17, 2015
[8] http://www.metacritic.com - last visited on March 17, 2015
[9] http://www.thetimes.co.uk - last visited on March 17, 2015
[10] http://www.howtogeek.com - last visited on March 17, 2015

2.3.3 The Advertisement/Product Placement Type

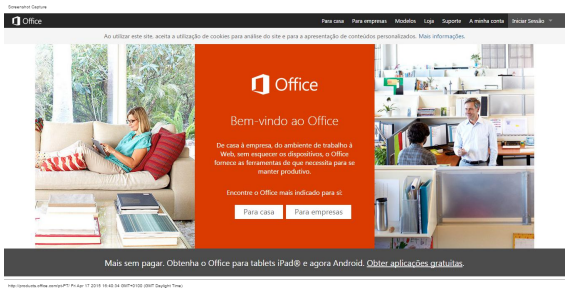
The Advertisement/Product Placement type is composed of web applications dedicated to the presentation or advertisement of a specific product, service or enterprise. Their main goal is to provide information about a product to the customer. Most of these web applications contain the means of accessing a main or other sites related to the product itself, such as *e-store*, blog, among others. Mainly they disseminate the information about the product, in order to induce a possible consumer to have a favorable dynamic attitude towards it. This can lead to a purchase, subscription or even a spread of the word of the product by the public itself. Some commonly known examples are: "Android"[11] in Figure 2.7a, "Outsystems"[12] in Figure 2.7b, "Microsoft Office"[13] in Figure 2.7c, "Kaspersky"[14] in Figure 2.7d and so on.



(a) Android



(b) Outsystems



(c) Microsoft Office

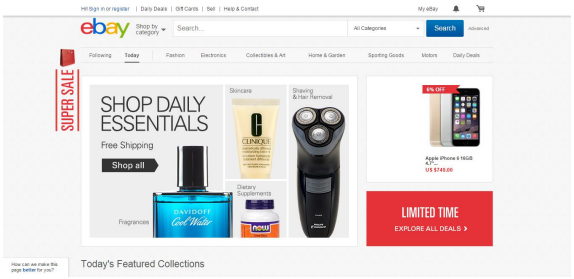
(d) Kaspersky

Figure 2.7: Examples of Advertisement/Product Placement web applications

[11] http://www.android.com - last visited on March 17, 2015
[12] http://www.outsystems.com - last visited on March 17, 2015
[13] https://products.office.com/pt-PT - last visited on March 17, 2015
[14] http://www.kaspersky.com/pt/ in Thursday March 17, 2015

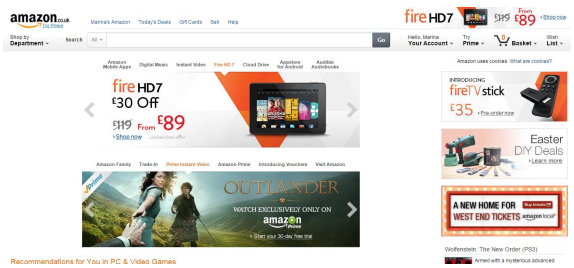
2.3.4 The *eCommerce* Type

The *eCommerce* type is composed of web applications dedicated to the trading of a specific product, group of products or even services through the internet. Users can register to acquire products, without having to access a physical intermediary. They can search, consult and purchase goods. In some of these applications, users are also allowed to trade products, either as a business, independent curator, or simply as a seller of unwanted or used articles. The main goal of these applications is to make products more accessible, either nationally or internationally, increasing the speed of search and the range of selection of products. Some commonly known examples are: "Ebay"[15] in Figure 2.8a, "LightInTheBox"[16] in Figure 2.8b, "Amazon"[17] in Figure 2.8c, "OLX"[18] in Figure 2.8d and so on.

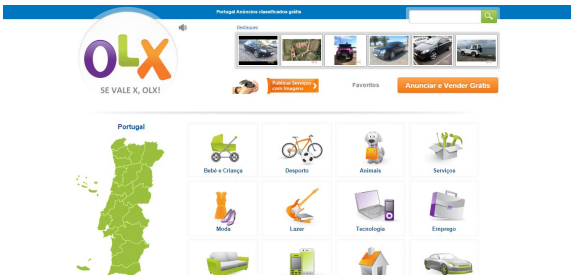


(a) Ebay

(b) LightInTheBox



(c) Amazon



(d) OLX

Figure 2.8: Examples of Blogging/Magazine web applications

[15] http://www.ebay.com - last visited on March 17, 2015
[16] http://www.lightinthebox.com/pt/ in Thursday March 17, 2015
[17] http://www.amazon.co.uk - last visited on March 17, 2015
[18] http://olx.pt - last visited on March 17, 2015

2.4 Conclusion

This chapter intended, above all, to analyze different user interface generation tools in order to conclude on the model-based approach. The model-based methodologies reveal to contribute to a fast and development of quality user interfaces, showcasing however, some issues. The study of the generations of MBUID tools and other tools fitting this context allowed to identify strengths and weaknesses of each perspective, contributing to the development of the thesis approach. The analysis of web applications allowed to recognize their division by domain, which is an essential aspect of the *MODUS* approach. It was deduced that, the application domain itself, besides defining the functionality and content of an application, influences many aspects related to its user interface. To demonstrate the categorization of web applications by domain, a subset was presented complemented with concrete examples

Chapter 3

The *MODUS* Approach

> ”Logic will get you from A to Z. Imagination will get you everywhere.”
>
> Albert Einstein

This chapter enunciates the base concepts of the *MODUS* approach, focusing on the definition of each step of the proposed process. The process consists of five stages: front end resources configuration; business logic model analysis; generic user interface deduction; user interface definition; final user interface generation.

3.1 The Conceptual Approach

The *MODUS* conceptual approach is portrayed in Figure 3.1. To initiate the process it is necessary to provide the following input data:

A) the specification of the application domain;

B) the business logic model, represented as a UML class diagram;

C) the information about the front end resource configurations;

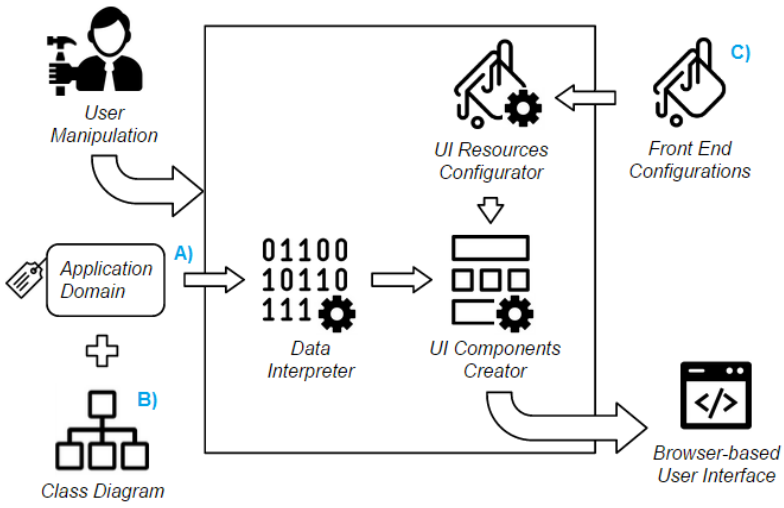


Figure 3.1: Overview of the Approach Process

The *MODUS* process is composed of three main structural units (see Figure 3.1, center): the Data Interpreter, the UI Resources Configurator and the UI Components Creator. These units are responsible for the generation of the appropriate user interface, from the class diagram and the application domain. By integrating the concept of evolutionary prototyping, the user is allowed to interact with the intermediate outcomes produced by each of the units. The final user interface is expressed as a WUI, more specifically as a browser-based web interface.

The **Data Interpreter**, responsible for extracting all relevant information from the user input, is composed of several modules. One of the modules is the Business Logic Model Analysis, presented in Section 3.3, which thoroughly analysis the business logic model. The idea is to set a bridge between the concrete and generic representation of the application. Another module is the Generic User Interface Deduction, presented in Section 3.4, which determines a typical interface of the application domain. The definition of this interface is made by balancing the data of the model and the generalizations of the domain. The overall interpretation process should be designed to be solid and expandable. To achieve so, it is necessary to define a generic interpretation mechanism, capable of being introduced and applied unambiguously to various domains and user interface deductions.

The **UI Resources Configurator** is responsible for enhancing the overall visual appearance of the UI. It supports the configuration of the front end resources, establishing the overall design of the user interface. However, the configuration of this unit is intended to be optional. The UI Resources Configurator is composed by a singular module, the Front End Resources Configuration, presented in Section 3.2. This process is essential to integrate the aesthetic choices of the designer in the creation of the FUI. The presented methodology should be generic to all front end resources. The idea would be implementing a configuration mechanism independent to the associated front end resource, able to support different levels of configuration, from general settings to detailed aspects of the UI.

The **UI Components Creator** is responsible for generating the user interface. At the beginning there is a focus on the intermediate components of the user interface, by the User Interface Definition module, presented in the Section 3.5. Each of the components is created from the descriptions calculated by the Data Interpreter, according to the front end resources established in the UI Resources Configurator. This module should promote the flexibility of user interfaces, avoiding a rigid and repetitive automatic process. In order to do so, it is mandatory to select an appropriate data format to serve as the basis for the creation mechanism. After the completion of this stage, all the elements are combined to originate the

final user interface. This is done by the Final User Interface Generation module presented in Section 3.6.

3.2 The Front End Resources Configuration

The *MODUS* approach integrates the use of front end resources in the generation of the UI. In the context of the approach, they are defined as all resources dedicated to enhancing the front end experience with the application, more specifically the quality and the appeal of its user interface. The resources are defined according to a CSS front end framework, such as *Bootstrap*. These frameworks provide a collection of tools to facilitate and accelerate the creation of browser-based web applications. They are mostly composed of pre-made code and templates for different application widgets, such as buttons, forms, and so on. Usually, they resort to a file or files for the easy definition of the user interface design. The idea is to take advantage of these files to configure the framework, and thus define the front end resources associated with the generated UI.

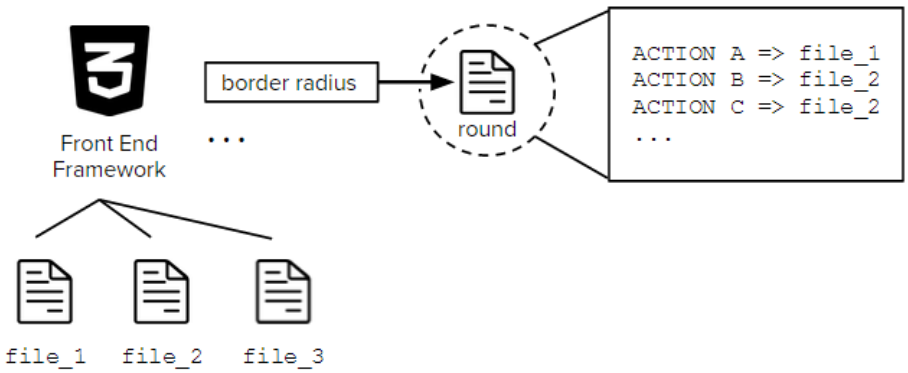


Figure 3.2: Simplified representation of the Style Definition Mappings

To allow the manipulation of the front end resources, the frameworks are registered with a set of mappings (see Figure 3.2), from the style definition, as for example setting the border radius to ”squared”, to the design updates, in this case the framework modifications that will enable the border radius to be ”squared”. Each mapping value is defined in a file, enumerating all actions that need to be applied to the framework files in order to enforce the style definition. Once every style definition is set, all appropriate mapping values are interpreted, registering all the actions to be executed. The actions are then grouped by framework file, applied by ascending time of setup, giving privilege to the latest modifications. This will

assure that the most specific configurations will override the most generic, in case that two or more actions affect the same design specification.

3.3 The Business Logic Model Analysis

At this stage, the business logic model is analyzed in order to identify relevant information that can be related to the domain. This process is mostly based on associating entities, in other words, classes of the class diagram, to standard classes. A standard class represents a class that usually exists in the modeling of a particular application domain, as for example the `Product` standard class of the `eCommerce` domain.

The algorithm of associating an entity with a standard class, illustrated in Figure 3.3, is based on the identification of relationship patterns. A relationship pattern defines a set of common architectural relationships that typically exist between a standard class and the others in a certain application domain. An example is considering that, for the `eCommerce` domain, a `Cart` has many `Products` At the beginning of the process every entity is considered a possible solution for the association (A). When a pattern of a standard class is identified on an entity (1), it is considered a valid solution for the association (B). If no association is met for the pattern (or patterns), the standard class is ignored, considered as nonexistent in the model.

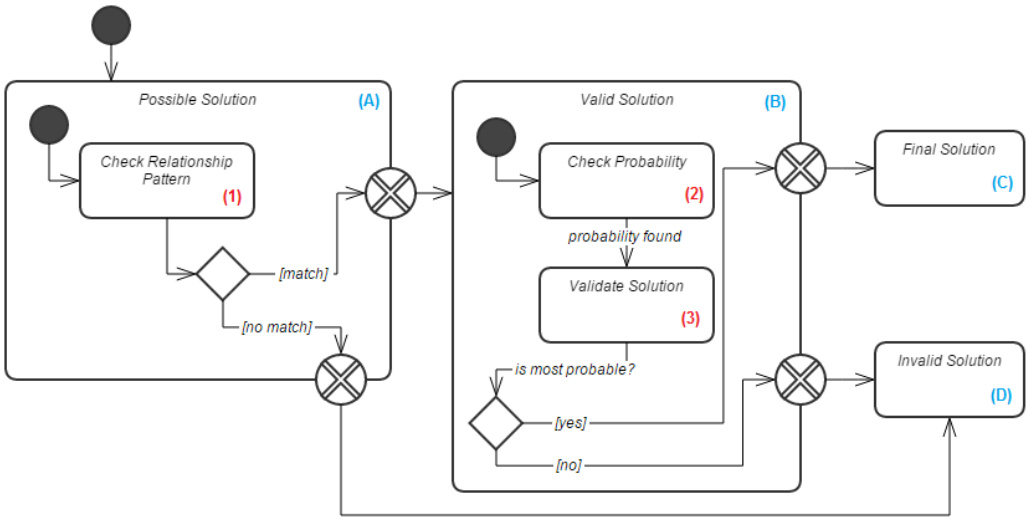


Figure 3.3: UML state machine diagram of the association process between an entity and a standard class

Since different patterns can overlap, it is likely that for each standard class more than one valid entity is identified. One asset that can be used to find the final solution is the identifier of the entity, its name. It is known that programmers tend to name their classes according to the subject they represent. The *MODUS* approach exploits this fact, setting the next step of the analysis (see Figure 3.3 (B)) as an estimation of the compatibility between the entity and the standard class name. Every standard class is linked to a dictionary, which contains a set of synonyms, each with the probability to match the name of the standard class in question. The name of the valid entity is looked up in the dictionary, being compared to each record (2). The resulting probability is adjusted depending on the degree of similarity with the synonym and the probability found in the document (3). The most probable solution is associated with the standard class (C), and all other solutions are discarded (D).

3.4 The Generic User Interface Deduction

In order to create a user interface, it is required to define it in terms of content and navigation. A browser-based UI is usually composed of a set of views, single pages of the application, which can be considered as the user interface content. In its turn, the navigation assembles all of the existing transitions between each of the application's views.

The *MODUS* approach exploits the fact that user interfaces belonging to the same domain are similar. It specifies UI generalizations that can be applied to all interfaces belonging to a certain domain. The outline of the interface can thus be originated from the business logic model and the assumptions taken from the application domain. It is necessary to find a way to organize this information. A clever way was unifying all the generic data of both navigation and content in a singular interface model, which represents a typical user interface belonging to a certain domain. This model will be provided to the user, reducing the development effort required.

To avoid having to prescribe a new form of modeling from scratch, an existing model was adapted to the context. An appropriate choice is the UML state diagram, which portrays the evolution of a runtime system, by displaying the transitions between its various states. In this perspective, the changes in states can be compared to the transition between the pages of a browser-based web application. The states can stipulate the content that needs to be assumed by the interface in a specific moment of the execution. However the Interaction Flow Modeling Language (IFML) model can also be a suited representation. This model is focused on the front-end software of an application, portraying its behavior, the interaction

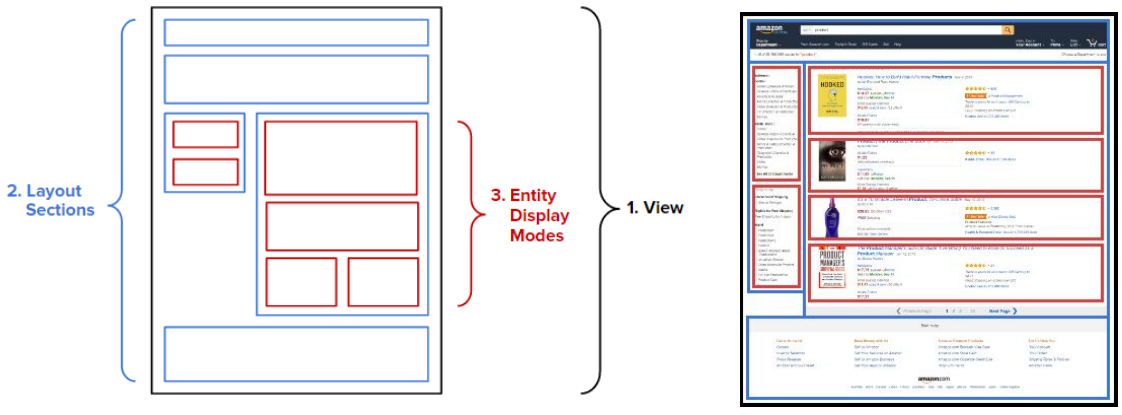
with the user and even the displayed content. Due to time restrictions it was decided to use the most acquainted approach by the author, selecting the UML state diagram as the base of the interface model. Despite the IFML model being less expressive for the specific purposes of the approach, adapting to a content and navigation map would be, in principle, a relatively feasible process.

3.5 The User Interface Definition

The *MODUS* approach stipulates that each view (1), as presented in Figure 3.4, is composed by a layout (2), which defines its structure, and a grouping of different entities (3), which determine its content. The layout specifies the combination of sections that make out the view, wherein a section is considered a view fragment common to other views, such as a *header*, a *footer*, among others. The grouping of entities is implemented in the form of display modes, portraying the various ways of presenting an entity in the application's user interface. Some common examples are:

- complete representation, which states all (or almost all) the information about an entity;
- miniature representation, which presents some information about an entity;
- hyperlink representation, which represents an interface hyperlink of an entity;

If a display mode is not associated with any entity, it describes a content independent from any of the entities present in the business logic model, such as a *slideshow* of a page.



(a) View Definition (b) Example: Amazon

Figure 3.4: Simplified representation of a View in *MODUS*

During the content and the navigation map deduction process, the layout and content of each view are determined. The creation of these UI intermediate components is based on templates. More specifically, they are originated from the parsing of templates, according to the front end resources and the application domain. The template approach for the generation process is suited for the field of user interfaces, in which aesthetic concepts are often changed, undergoing a constant evolution. Furthermore, a creation based on templates allows the establishment of a generation engine which is independent of the obtained results. The addition, deletion or manipulation of the templates does not affect the implementation of the process. The designer can even be allowed to import his own personal templates to further personalize the interface. The variety of templates, with all possible combinations that can be applied, increases the diversity of the solutions that are generated, thus having a positive impact on the flexibility of the *MODUS* approach.

3.6 The Final User Interface Generation

The generic user interface deduction process establishes the outline of the user interface views. It intends to determine the views that compose the UI as well as the navigation between each of the views. Then, the user interface definition process implements the defined intermediate elements that are tangible, in other words, the layout sections and the entity display modes. By complementing the schema of the views obtained from the interface model with these components, one can start the generation process of the final user interface.

As stated before, a browser-based application is made out of different views. One can conclude that the FUI should contain a set static views to demonstrate the overall look & fell of the application. However, it is necessary to take into account that the web pages of an application tend to be generated or updated in response to an interaction with the user. At programming levels, the views are created from the manipulation of different UI fragments, being those in most cases the partials, which are view fragments, and the layouts. Therefore, the FUI should also contain these fragments, in order to better integrate with the software production.

3.7 Conclusion

This chapter was dedicated to exposing the architecture of the *MODUS* approach. The architecture can be divided in three main units, the Data Interpreter, the UI Resources Con-

figurator and the UI Components Creator, grouping five different stages:

- the front end resources configuration, which sets up the look and feel of the application;
- the business logic model analysis, which associates the class diagram data of the application to its domain;
- the generic user interface deduction, which defines the generic outline of the UI
- the user interface definition, which implements the base components of the UI;
- the final user interface generation, which creates the FUI;

All stages were conceived to produce a systematic approach, in order to stipulate the foundations for the implementation of a *MODUS* prototype. The front end resources configuration resorts to mappings from the style definitions to the design updates associated with a front end framework. The business logic model analysis is based on an inference mechanism based on both relationship patterns of standard classes and the estimation of the probability of matching their identifiers. The generic user interface deduction will stipulate a new interface model, unifying all information about typical user interfaces of a certain domain. The user interface definition will originate the intermediate UI components from the parsing of templates. Finally, the final user interface generation defines the FUI as a set of views, to simulate the interaction with the user, and a grouping of partials and layouts, to be to generate the interface at programming levels.

Chapter 4
The *MODUS* Prototype

The presentation of the *MODUS* methodology premises serves as the foundation for establishing the main lines of the supporting tool, known as the *MODUS* prototype. The tool should be understood as a reference implementation. In practice, the *MODUS* prototype will be used to demonstrate the feasibility of the proposed approach.

This chapter is dedicated to presenting the architecture of the *MODUS* tool prototype. It defines a concrete implementation for the conceptual stages presented in the Chapter 3. Beyond the introduction of the prototype's architecture, this chapter intends to validate all the previously formulated decisions about the conceptual process, selecting an implementation to fulfill the *MODUS* approach. The prototype architecture will be composed by the following stages: the *ecore* model extraction, the standard classes identification, the CSS framework manipulation, the content and navigation map interpretation, the UI intermediate components creation, the FUI generation.

4.1 The Prototype Architecture

Figure 4.1 summarizes the prototype architecture for the implementation of the *MODUS* approach. The architecture is composed of 6 intermediate stages. In Figure 4.1 every stage is represented by numerically labeled arrows, displaying the occurring flow of information. The input data is labeled alphabetically, in uppercase if provided by the user (specific to a particular application) and in lowercase if provided by the tool (generic across applications).

The first stage is **The Ecore Model Extraction** (Figure 4.1 - 1), presented in Section 4.2. In this stage, the business logic model is validated, and the data about the entities is extracted.

The second stage is **The Standard Classes Identification** (Figure 4.1 - 2), presented in Section 4.3. In this stage, the entities' data is interpreted with the help of a) relationship patterns and b) semantic knowledge databases, in order to identify relevant entities of the application domain.

The third stage is **The CSS Framework Manipulation** (Figure 4.1 - 3), presented in Section 4.4. In this stage, the front end resource configurations are applied to the front end framework of the user interface to be generated.

The fourth stage is **The Content and Navigation Map Interpretation** (Figure 4.1 - 4),

presented in Section 4.5. In this stage, an outline of the user interface is drawn from c) an abstract interface model for the domain initially set.

The fifth stage is **The UI Intermediate Components Creation** (Figure 4.1 - 5), presented in Section 4.6. In this stage, intermediate components of the interface are implemented, namely the display modes and layout sections, through the use of d) templates.

The last stage is **The FUI Generation** (Figure 4.1 - 6), presented in Section 4.7. In this stage, generates the final user interface.

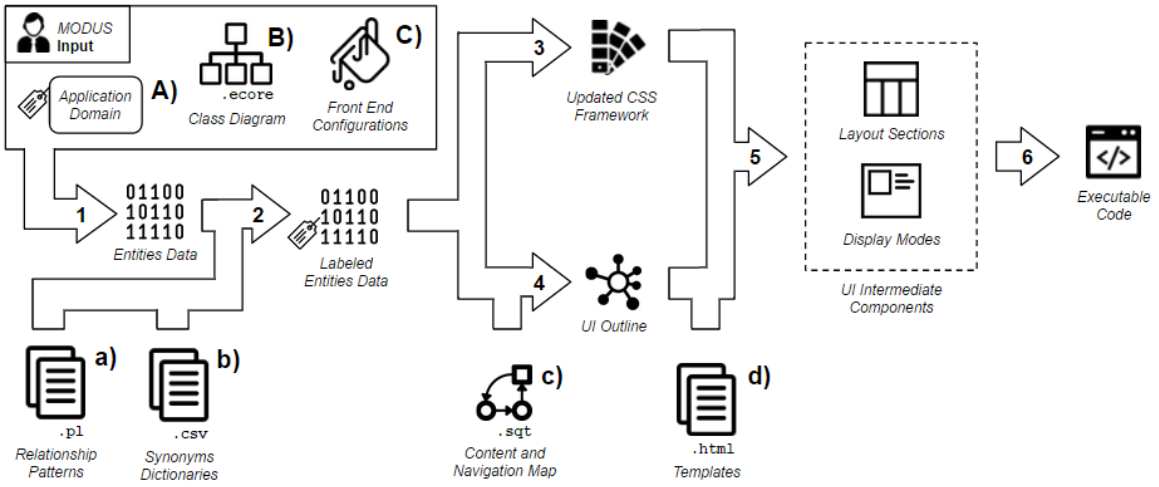


Figure 4.1: Overview of the Prototype Architecture

4.2 The *Ecore* Model Extraction

The first stage of execution of the *MODUS* approach is the interpretation of the business logic model. An example of a UML class diagram of the *eCommerce* domain is presented in Figure 4.2. In this model, an ”Article” is defined by its ”Section” and may be carried out on the specific types of ”Clothing”, ”Shoe” and ”Accessory”. In its turn, ”Clothing” can be stated as a ”Dress”, ”Skirt”, ”Top” or ”Pants”. A ”Customer” has a ”WishList” and a ”Cart”, both groupings of products. He can perform ”Order”’s associated with a ”Cart” and one of its ”Address”’es. A ”Customer” can also make ”Review”’s about an ”Article”. Finally, the ”Customer” has a ”Contact” information to specify some personal details. This example will be used throughout the presentation of the *MODUS* architecture for demonstration purposes.

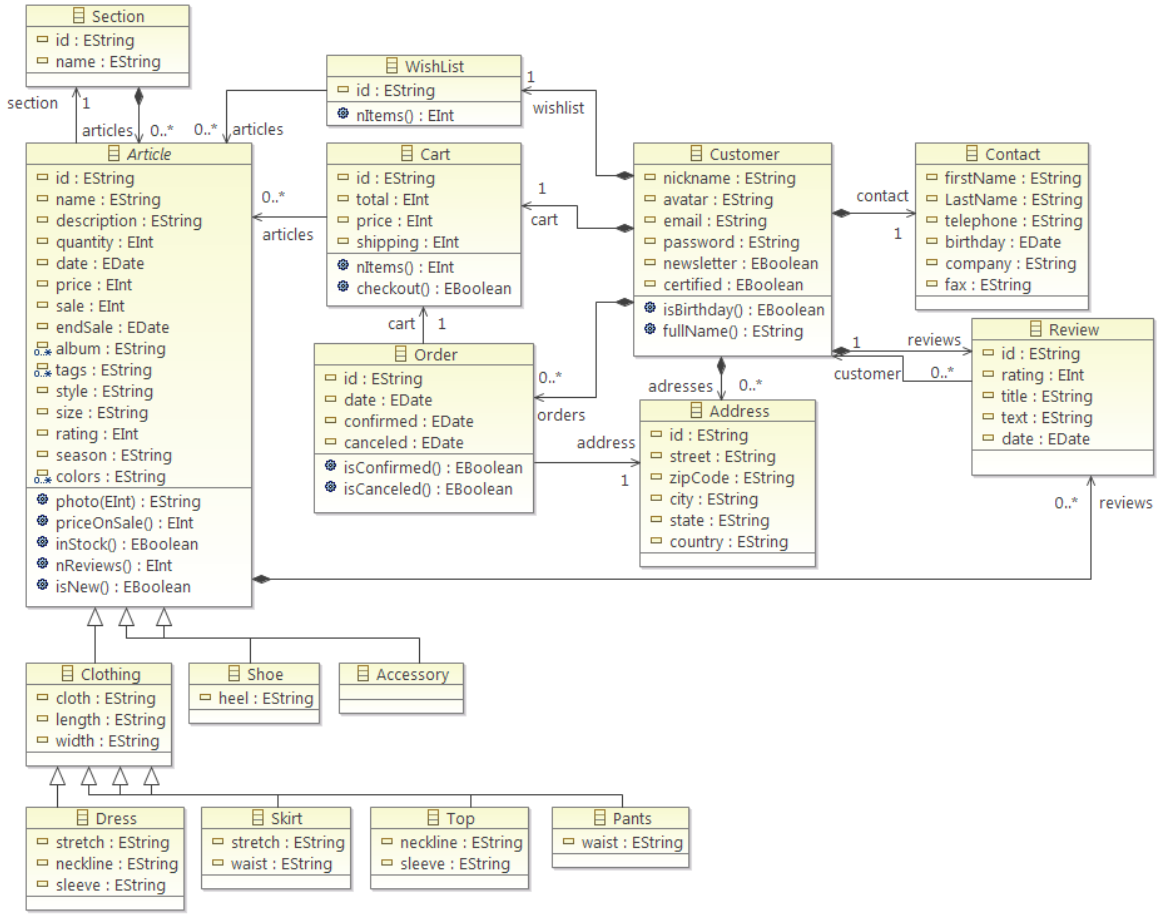


Figure 4.2: Example of a UML class diagram of a *eCommerce* web application

In order to extract the relevant information from the class diagram, it is necessary to determine the interchange format expressing the UML model. The first choice in mind is the XML Metadata Interchange (XMI) format. XMI is considered a standard for exchanging metadata information in EXtensible Markup Language (XML), used mostly to represent UML diagrams. Another choice is resorting to *ecore*, a meta-model directed towards the description of models [Steinberg et al., 2009]. The *ecore* format is serialized with XMI, representing the canonical form of the *ecore* model itself [Steinberg et al., 2009]. Using the *ecore* format allows one to work upon a known standard. Furthermore, *ecore* is the standard format for *Eclipse EMF*. Considering this analysis, it was decided to work upon the *ecore* format for representing class diagrams, in order to facilitate the implementation process.

As the *MODUS* prototype is an *Eclipse* plugin, it can exploit one of the Integrated Development Environment (IDE) tools to create the business logic model. The *ecore* format ben-

efits from one particular visual diagram editor, the *Ecore Tools* [Foundation, 2014], which is an appropriate candidate to support *MODUS*. The *Ecore Tools* plugin has many advantageous features, being one of the most relevant its validation feature. This asset verifies the existence of errors in the file, aiming to help the user create a correct model. However, it is necessary to filter some inconsistencies specific to the *MODUS* generation process. Examples are: an empty model, duplicate identifiers, references to non-existing elements, among others. Some auxiliary validations were added to the extraction mechanism, as described in the Figure 4.3.

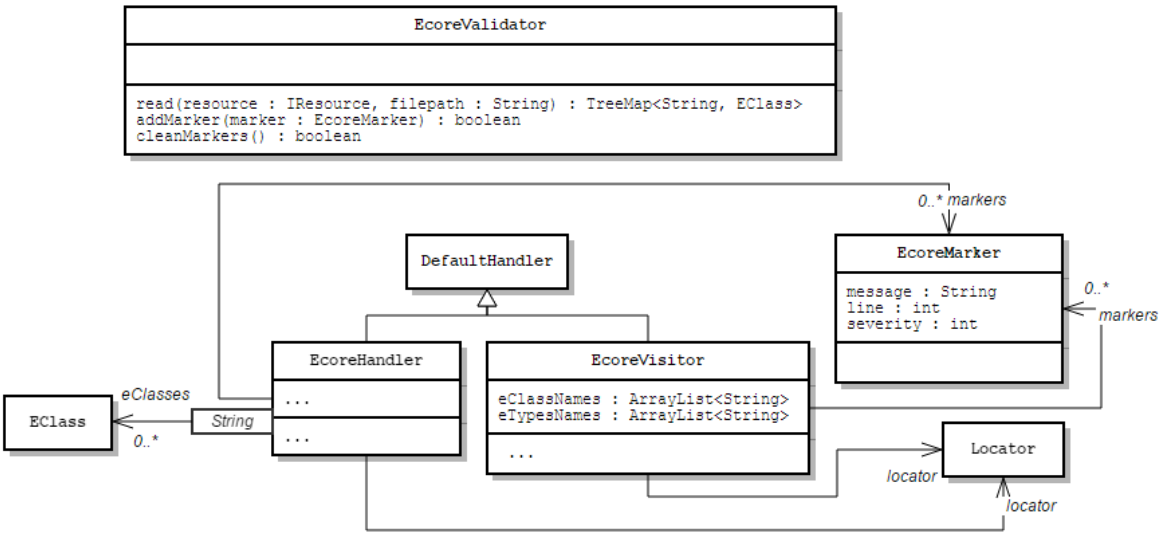


Figure 4.3: UML class diagram for the ecore model extraction

The main idea, is to read the *ecore* file and obtain all relevant information about the model's entities. In particular in the form of instances of *EClass*, a *MODUS* prototype *Java* implementation for the class element of the class diagram. This process is divided into two main steps. First, the document is visited to extract the class identifiers and data type elements. Despite some basic validations being applied, the main goal is to obtain basic information, relevant for further verifications and data storage. Second, the document is deeply checked and parsed to extrapolate the relevant knowledge in the model. If errors are detected, the *IResource* object referencing the model in the *Eclipse* editor is injected with validation markers. The file would display in detail the errors, so the user is able to correct the diagram accordingly. At the end of this stage, all the relevant information on the class diagram was extracted and validated according to the *MODUS* process, being the entities data stored for further manipulation.

4.3 The Standard Classes Identification

The process of associating entities and standard classes is composed of two steps as displayed in Figure 3.3. The first step is the search for relationship patterns of the standard classes. In this stage, the data present in the class diagram is converted to a *Prolog* ontology. Choosing a logical programming language allows one to easily infer on the knowledge contained in the extrapolated ontology. This way, the information about the entities can be expressed in facts, as they represent true statements, and can be represented as predicates about the diagram. Furthermore, the patterns can be described as rules, so they can be consulted to calculate whether or not an entity matches the standard class. This inference process is based on the approach presented in [Couto et al., 2012].

The knowledge base is composed of two types of facts, represented in Listing 4.1. The fact from the first line states the existence of an entity called `E` (`Entity`). The fact from the second line states the existence of a relationship between two entities, being: `R` (`Relation`) the name of the relationship; `F` (`From`) the starting entity; `T` (`To`) the entity of arrival; `L` (`Lower Bound`) the lower bound and, to conclude, `U` (`Upper Bound`) the upper bound of the cardinality of `T` in the relationship. The example shown in Listing 4.2, based from the UML model on Figure 4.2, specifies that there are two entities, ”Customer” and ”Address”, in which the customer can have multiple addresses.

Listing 4.1: Types of facts of the *Prolog* ontology

```
entity(E).
relation(R,F,T,L,U).
```

Listing 4.2: Examples for facts of the *Prolog* ontology (based on 4.2)

```
entity(customer).
entity(address).
relation(addresses,customer,address,0,n).
```

During the inference process, a query verifies if a certain entity meets a relationship pattern. This amounts to checking the truth value of the rule representing the pattern in question. For example, Listing 4.3 describes the rule for the pattern of the standard class `Address` of the `eCommerce` domain. It is defined that a class `U` (`User`) must have addresses and that a class `O`(`Order`) must have at least one address, being `A` (`Address`) an entity of the ontology. In the model of Figure 4.2 the entity ”Customer” has many ”Addresses” and the

entity ”Order” has one ”Address”. The entity ”Address” appears to match the relationship pattern of the standard class `Address`. When interrogations for all patterns are conducted, the next step of the analysis may start.

Listing 4.3: An example of a relationship pattern in *Prolog*

```
ecommerce_address_pattern(A):-
  entity(A),                % Address is an entity
  relation(_R,U,A,_M,_N),   % User has Address
  has_at_least(O,A,1),      % Order has at least 1 Address
  different(A,U), different(A,O), different(O,U).
```

Since interrogating long *Prolog* queries in *Java* may reduce significantly the performance in terms of time, it was decided to implement two variations of the inference process: the Pure Inference Algorithm and the Mixed Inference Algorithm.

Figure 4.4: Brief description of the Pure Inference Algorithm

The Pure Inference Algorithm resorts only to *Prolog* queries for the first step of the analysis, as portrayed in Figure 4.4. Here, the relationship pattern is entirely represented in a rule, contained in a *Prolog* file, which makes this algorithm highly maintainable. Updating a pattern consists only in modifying the apropriate file.

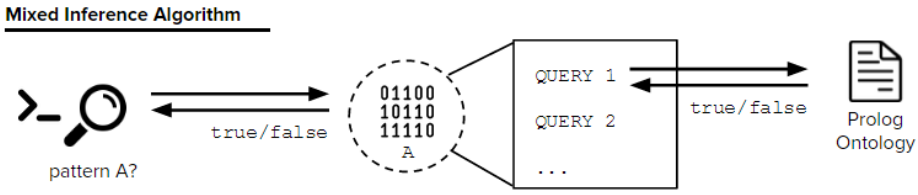


Figure 4.5: Brief description of the Mixed Inference Algorithm

In the Mixed Inference Algorithm, the patterns were statically implemented in *Java* code, as portrayed in Figure 4.5. Each rule was decomposed into small *Prolog* queries used to interrogate the knowledge base and obtain intermediate results. Each query is shorter, thus faster

to execute. The *Java* code can exit the pattern quicker when an intermediate interrogation returns an invalid value. Overall, this decomposition accelerates the inference process.

For the *MODUS* prototype it were established two inference engines for the standard class association. However the methodology could be supplemented with new algorithms that could meet future needs of the tool. It is necessary to take into account that each of these engines should be applied to an application domain. Depending on the inference method, its inclusion may or may not hinder the goal of having a generic and reusable implementation. For example, in the case of the Mixed Inference Algorithm the patterns are hard coded. Not to mention that applications domains can vary over time or due to cultural and social context changes. A deployment solution to solve this issue is required. If it is a recurring problem in the field of software engineering, one of the options would be to use design patterns [Gamma et al., 1995]. By analyzing the problem it is possible to conclude the following premises. First, these inference algorithms can be considered as a family of objects following the same pattern for calculating the association. Second, the specification of the concrete classes that solve the association is not relevant for the prototype user. Acknowledging these two premises, the Abstract Factory Design Pattern seems to be an appropriate choice for structuring the implementation. This pattern was adapted to the problem's context, so new algorithms and inference sub-processes for each application domain could be easily created, as portrayed in Figure 4.6.

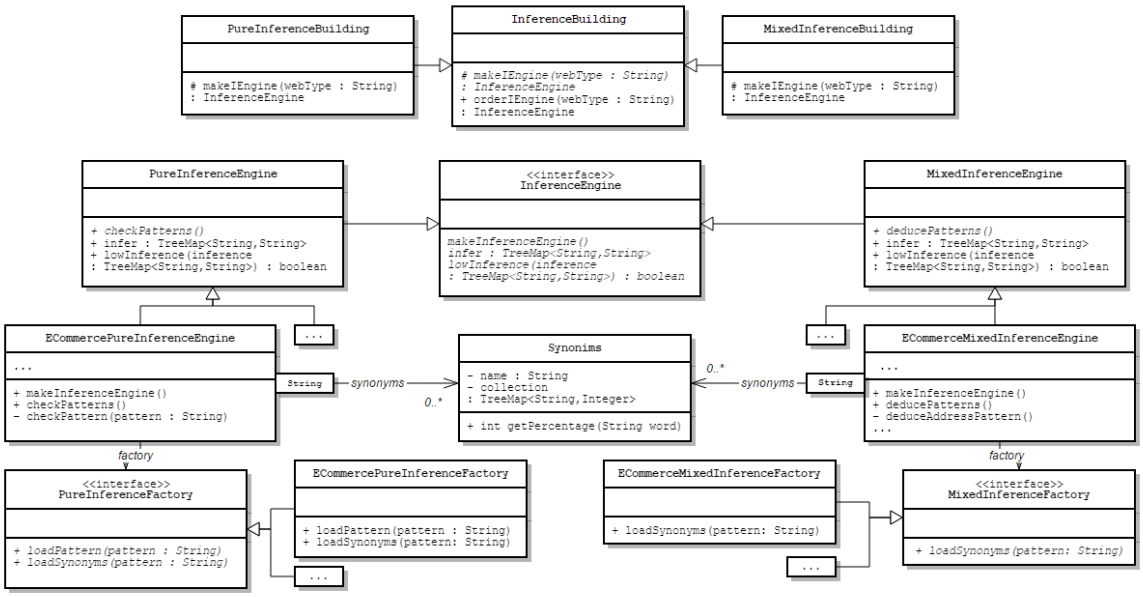


Figure 4.6: UML class diagram for the standard class association process

The objective is to build different inference engines that follow the same implementation structure. The inference engines will infer about all standard classes by interpreting all *EClass* classes obtained from the *ecore* extraction. These engines are built with the embodiments of `InferenceBuilding` abstract class. Each engine stipulates how to infer in the knowledge database, following the associated algorithm. For example the `PureInferenceEngine` class implements the Pure Inference Algorithm. The inference engines are complemented with factories which are generally responsible of either loading the relationship patterns or the synonym dictionaries. Each of the engines is then specified for an application domain. For example, the `ECommerceMixedInferenceEngine` implements the Mixed Inference Algorithm for the *eCommerce* domain, thus hard coding every *eCommerce* pattern in its implementation. The `Synonyms` class is used to store and manipulate the synonyms, and their probability. An instance of this class is obtained by reading the appropriate dictionary, in the forms of an XML file. This class is used in all inference engines, to realize the second step of the analysis as shown in Figure 3.3. Every engine is complemented with a `lowInference` method, which updates the inference result if a low match of association is met. If this is the case, the analysis is executed once again, skipping its first stage.

Resuming the example of Figure 4.2, the name ”address” is the perfect synonym for the standard class `Address`, thus having a 100% probability match. Since it is the most probable entity, it is associated with the standard class. Once the standard class association is finished, it is necessary to identify the entity's standard attributes. This identification process is based on the second step of the analysis, taking this time into account their respective types. Once the desired inference engine has finished the association process, the entities data are labeled according the standard classes and their attributes.

4.4 The CSS Framework Manipulation

To implement the CSS Framework Manipulation it was decided to focus on the CSS preprocessor files which are often incorporated into the front end frameworks. CSS preprocessor files such as *Sass* [Hampton Catlin and Eppstein, 2006] files, are used to facilitate the production of CSS code, through the use of variables, nested structures, operations, among others. The manipulation of those elements will be the base of the implementation of the design of the interface.

As stated in Section 3.2, each resource configuration can be set to a specific value, which

is the key of the style definition mappings registered with the front end frameworks. The value assigned for the style definition matches a document in the XML format, determining the according CSS preprocessor updates. In order to be able to define these actions, the style definition file was structured following the represented in Listing 4.4.

Listing 4.4: DTD of the style definition document

```
<!DOCTYPE style-override [
 <!ELEMENT style-override (action-set)+>
 <!ELEMENT action-set (file,(replace-var|replace-inc|append)+)>
 <!ELEMENT replace-var (variable,value)>
 <!ELEMENT replace-inc (include,variable,value)>
 <!ELEMENT append (selector,(line)+)>
 <!ELEMENT variable (#PCDATA)>
 <!ELEMENT value (#PCDATA)>
 <!ELEMENT include (#PCDATA)>
 <!ELEMENT selector (#PCDATA)>
 <!ELEMENT line (#PCDATA)>
]>
```

The Document Type Definition (DTD) states that the `style-override` node is the root of the document, containing only a non empty list of `action-set` nodes. An `action-set` defines the set of actions to be applied to a preprocessor file. Thus, it contains the `file` node, the path of the file relative to the framework main folder, and a group of at least one action. An action can be determined by:

- a `replace-var` node, identifying the action of replacing the value of a variable; it is composed by a `variable` and a `value`, exposing respectively the name of the variable and its new value.
- a `replace-inc` node, identifying the action of replacing the value of an include; it is composed by an `include`, a `variable` and a `value`, being the `include` node the name of the include associated with the `variable`.
- an `append` node, identifying the action of appending new content to a selector; it is composed by the `selector`, the CSS selector, and a non empty set `line`, representing the CSS code lines that will make out the appended content.

An example is demonstrated in Listing 4.5, which represents the style definition for the ”squared” value of the border radius. This style definition means that every visual component of the application will have no border radius, therefore appearing to be squared. As can be observed, it contains only one `action-set` associated to the preprocessor file ”_variables.scss”. In this example the border radius related variables will be updated with the value ”0”.

Listing 4.5: Style definition file for the ”squared” value of the overall radius

```
<style-override>
  <action-set>
    <file>sass/bootstrap/_variables.scss</file>
    <replace>
      <variable>$border-radius-base</variable> <value>0</value>
    </replace>
    <replace>
      <variable>$border-radius-large</variable> <value>0</value>
    </replace>
    <replace>
      <variable>$border-radius-small</variable> <value>0</value>
    </replace>
  </action-set>
</style-override>
```

The interpretation process of the style definition resorts to the `ActionSetReader` class, displayed in Figure 4.7 to create the necessary groupings, from preprocessor files to their list of `ActionSets`. An `ActionSet` is generated by parsing the XML file with the `ActionSetHandler` class, extending the default handler for SAX2 events. The `ActionSet` and composing classes match the DTD structure presented in 4.4. The classes implementing the abstraction `Action`, in other words, all actions of the style definition, contain an `exec` methods, which will be applied the portrayed update to a textual content. This method will be dispatched for all `Actions` when the groupings obtained from all files are orderly merged. When the interpretation process is concluded, all the CSS preprocessor files have been updated, the main file preprocessor file is compiled, generating the final CSS code that will define the design of the user interface.

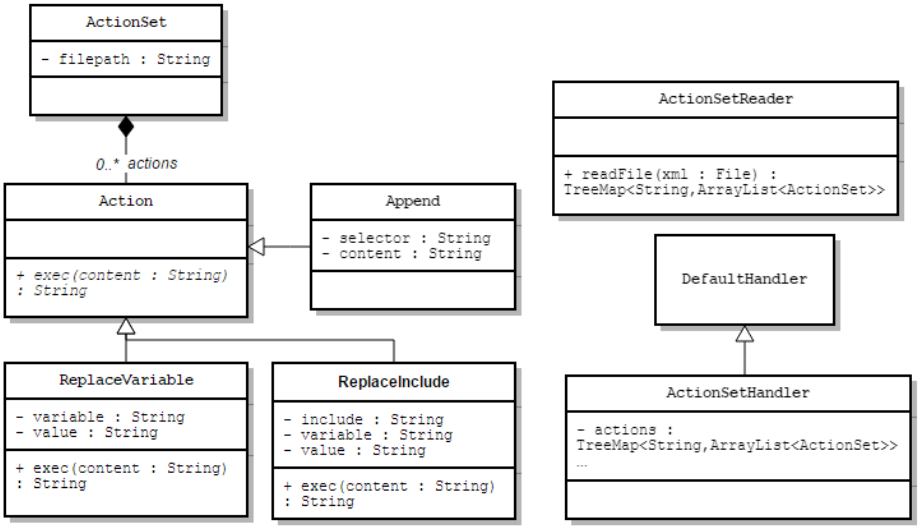


Figure 4.7: UML class diagram for the style definition interpretation process

4.5 The Content and Navigation Map Interpretation

The interface model was established as a content and navigation map, a variation of UML state machine. In the content and navigation map, the views are defined as a composition of partials. The assembling of every partial will build up to be the final content. Partials either group other partials or contain the interface representation of one or more entities. An entity interface representation, commonly know in the *MODUS* approach as a *entity display mode*, is meant to draw on the UI a concrete rendering of the entity's content, in terms of variables, operations and references. The navigation would be represented as transitions between each of the views. Due to the specifications of this methodology the state machine was adapted as follows:

State Represents either a page, a partial, a section or an entity display mode. The types associated with this component are distinguished by their identifiers or their location in the diagram. If the state belongs to a region of another state, it represents a section of the page, as shown in Figure 4.8 2). The partial identifiers are initiated with the character `'_'` and the display modes identifiers are delimited by `'<'` and `'>'`, as shown in Figure 4.8, respectively in 3) and 4). The remaining states identify views, as shown in Figure 4.8 1). The states have attributes to represent auxiliary information, as depicted in the Table 4.1.

Final State Indicates the exit of a transition due to a specific condition, as shown in Figure 4.8 5). Meaning that, the transition will only be applied if the state in which the transition is sourced does not match one of the identifiers present in the condition.

Initial State Indicates the starting view of the user interface, as shown in Figure 4.8 6). Usually, this is the first view the user interacts with during the execution of the application.

Transition Indicates a transition, replacement or composition, depending on the target element, as shown in of Figure 4.8 7). A transition to a page represents a page change. A transition to a partial represents the content replacement of the source section. Finally, a transition to a display mode represents a content composition.

Condition States the identifiers involved in a transition, as shown in Figure 4.8 8). They are only used to complement the Final State.

Name	**State**	**Description**	**Operands**
link	partial/view	It indicates the link's relative path	path
order	partial/view	It indicates the order of assembly	identifier, position
repeat	partial/view	It indicates the number of repetitions	identifier, number
redirect	partial/view	It indicates the navigation of a certain element	identifier, CSS selector
name	view	It indicates the view's name	name
template	view	It indicates the view's template	template name
display	display mode	It indicates the name of the display mode	name
type	display mode	It indicates the type of the display mode	type

Table 4.1: Description of the attributes in the Content and Navigation Map

An example of the content and navigation map is portrayed in Figure 4.8. This map depicts that the system will have as a starting point, the `homepage` view. The `homepage` view, despite the default `body` section assumed by the all states, has only one other section: the `#sidebar` section. The `_index_sidebar_category` contained in the previous partial, is made out of a repetition of `_show_sidebar_category`, which includes the `sidebar`

display mode of `Category` standard class. This combination of states is frequently used to describe a list of a partial or an entity in a certain display mode. The `homepage` view is composed of a list of `Product` elements in the `min` display mode. Each of these elements forwards to the `product` view, only if the parent state is not `_index_min_product`.

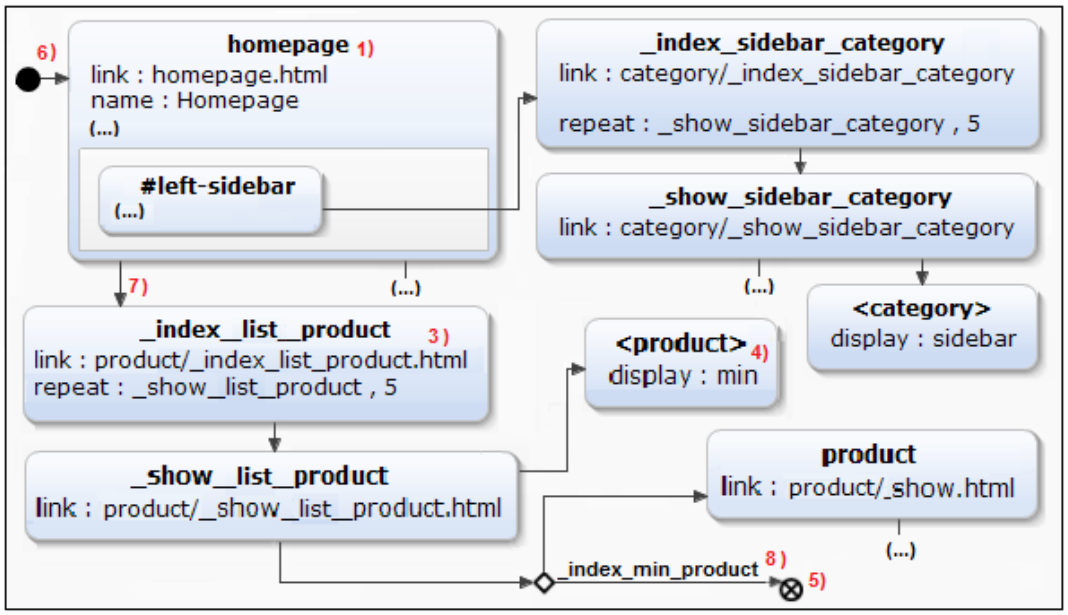


Figure 4.8: Example of the content and navigation map (fragment)

Since the content and navigation map is mostly based on generalizations about the user interface of an application domain, it was decided to allow the the *MODUS* user, in other words the developer, to personalize these assumptions. He can modify the model through the use of the *Yakindu Statechart Tools* [itemis AG, 2014], a graphical editor for state machine diagrams provided by the *Eclipse* IDE. Thus, the content and navigation map interpretation is divided in two stages: the map update and the UI assumptions extraction. The map update is a write and read process, being briefly described by the sequence diagram of Figure 4.9. In summary, while the process is reading the XML file, it applies the necessary updates to adjust to the business logic model. The first step is validating the file, in order to assure the correctness of the user interface assumptions taken from the content and navigation map. Each node of the file is manipulated twice, in two separate cycles. The first time, the node is updated to match the business logic model. For example, if a state relates to a standard entity that does not exist in the business model, it is removed. In this cycle the node dependencies are also calculated. The second time, the nodes are diagnosed to check if they are or not a

dependency of a deleted node. If it is the case, the node is removed. At the end, only a valid and adjusted version of the map remains, which the developer may edit. Then the assumptions extraction process parses the map, in order to extract all the relevant information. This step is similar to the algorithms represented in Figures 4.7 and 4.3. All information about the content and navigation map is stored, grouping all instructions for the generation of the user interface components.

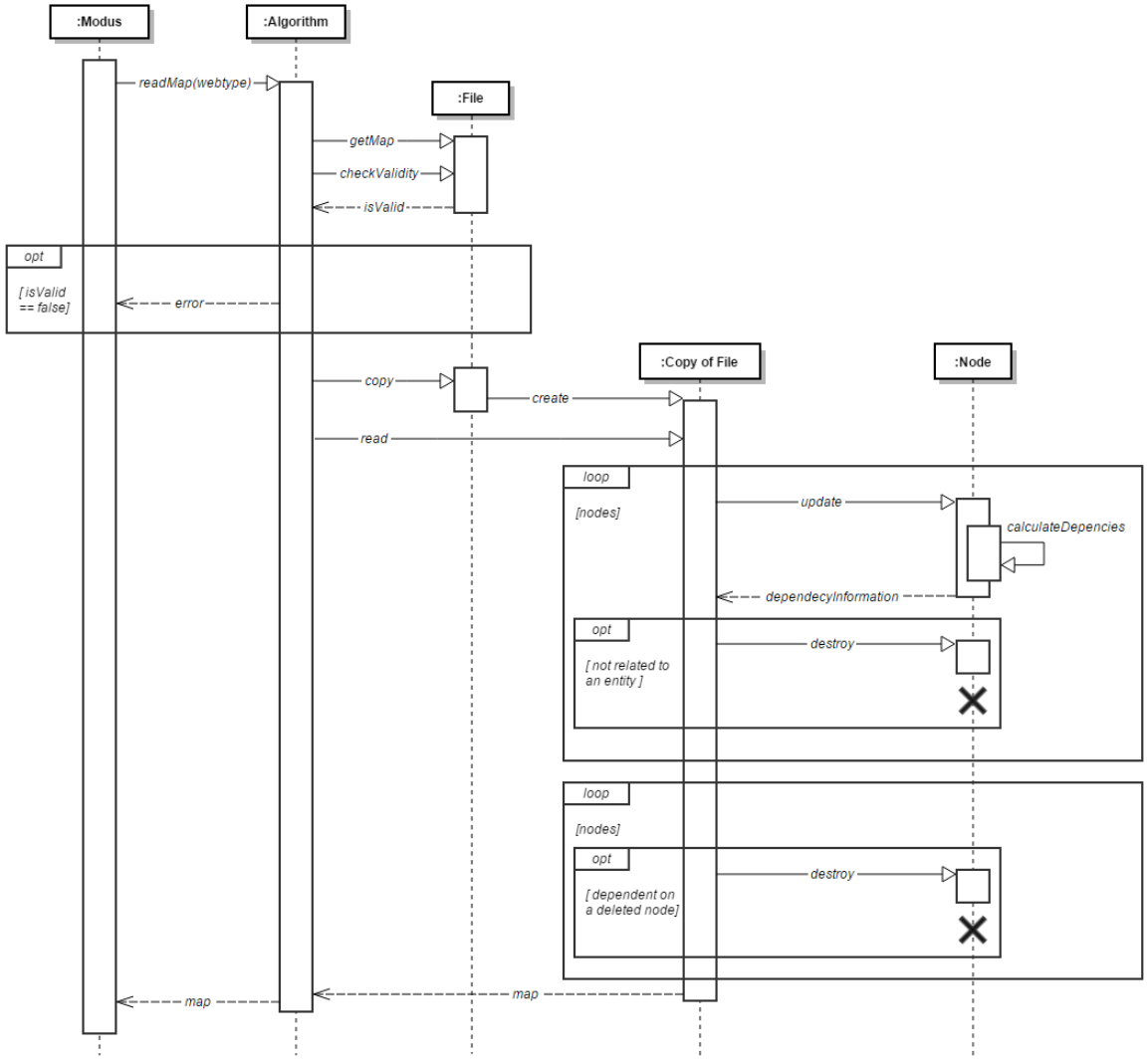


Figure 4.9: Sequence diagram of the content and navigation map update

4.6 The UI Intermediate Components Creation

Every intermediate component must be generated from the parsing of their according template. The templates for both display modes and layout sections are expressed in the final user interface technology, meaning HTML. The use of HTML as a GUI implementation markup language is an appropriate choice to represent browser-based interface content. Being HTML a standard, the templates are more maintainable at long term, being easier to construct or update. Also, being HTML the technology of the FUI, the tool user can obtain intermediate components more approximated to the final result, allowing more accurate modifications during the prototyping process.

The content of the template is located inside the `body` Document Object Model (DOM) element of the HTML file. Nevertheless the `head` tag can be used to refer *stylesheets* or other library dependencies, to display the user a functional and accurate demonstration. Despite the templates being defined as standard HTML files, they contain specific data attributes to support the creation of the output. These attributes indicate actions to perform or encapsulate elements of the business logic model, interface and navigation map. However, during the final stage of the generation process, they are removed, so as not to pollute the final user interface. The data attributes that define the template for the layout section and the display mode are, respectively, presented in Table 4.2 and in Table 4.3.

Name	**Value**	**Description**
`section`	String	CSS selector that identifies the layout section.
`element`	String	CSS selector that identifies an element of the layout section. A section element groups hyperlinks to other views.
`link`	”replace”	Indicates a hyperlink to an existing navigation to a view.
`link`	”sample”	Indicates a hyperlink template for a new navigation to a view.
`linker`	Boolean	Points to the hyperlink node. The boolean value indicates whether it is possible or not to replace its content.
`dependency`	String	Indicates that the node is dependent on a standard class.

Table 4.2: Data attributes that define the layout sections templates

Name	Value	Description
`type`	”class”	A ”class” type is used to encapsulate the display mode of an entity.
`type`	”attribute”	An ”attribute” type contains the representation of an attribute.
`type`	”reference”	A ”reference” type points to a display mode of another entity.
`type`	”container”	A ”container” type indicates the node which will contain repetitions of the display mode.
`type`	”action”	A ”action” indicates that an action will be performed on the node.
`name`	String	The name of the class diagram component.
`mode`	String	The name of the display mode.
`container`	Boolean	Points to the direct parent DOM element of the repetitions.
`dependency`	String	Indicates that the DOM element is dependent on a standard class.
`visible`	Boolean	Indicates the visibility of the class diagram component.
`action`	”fill”	Indicates that the DOM element will contain all the class components missing from the display mode.
`content`	List	Indicates the type or types of class components related to an action. The list can be composed by the following values can be either ”attribute”, ”operation” or ”reference”

Table 4.3: Data attributes that define the display modes templates

An example of a template for a `bottombar` layout section is expressed in Listing 4.6. In this template the `nav` tag contains the representation for the layout section. The template is composed only by simple DOM elements.

An example of a template for a a `Category` display mode of the *eCommerce* domain is portrayed in Listing 4.7. The display mode ”breadcrumbs” is a list element, including only ”name” attribute. The ordered list is a `data-modus-container`, thus responsible to contain multiple ”breadcrumbs” representations.

Listing 4.6: Example of a template for a `bottombar` layout section

```
<nav id="bottombar" class="navbar navbar-default"
    data-modus-section="#bottombar">
  <div class="container">
    <div class="collapse navbar-collapse">
      <p class="navbar-text">
        <small>Copywright Universidade do Minho 2014-2015</small>
      </p>
    </div>
  </div>
</nav>
```

Listing 4.7: Example of a template for a `Category` display mode

```
<ol class="breadcrumb" data-modus-type="container">
  <li><a id="breadcrumb_home" title="Homepage">Homepage</a></li>
  <li data-modus-type="class" data-modus-name="category"
    data-modus-mode="breadcrumbs">
    <span data-modus-type="attribute"
      data-modus-name="name">Category Name</span>
  </li>
</ol>
```

The creation process, for both layout sections and display modes, is similar to content and navigation map update process presented in Figure 4.9. The templates are searched by their stipulated name in the interface model. Following the content and navigation map presented in Figure 4.8, there is a display mode `list` for the `Product` standard class, in this case the ”Article” entity. The appropriate template is shown Figure 4.10a. However, if no name was stipulated a default or generic template is selected. The file is then validated and copied, so not to override the original template. During the reading process, the information is extracted and retained by the tool. The file is updated according the business logic, thus dependencies to non existing entities are removed, all template actions are executed, among others. Note that the front end resources were previously configured, thus the design of the template was set up. Since the ”Article” entity contains all the standard attributes featured in the template, and since the template itself does not have any `action` *MODUS* attribute, the ”list” template was updated as presented in Figure 4.10b. However, the modifications applied are not final as, for example, no nodes are removed, only hidden. This allows the tool to interpret the

same document more than once, since the user should be able to manipulate the content as he wishes, as shown in Figure 4.10c. After the user has personalized the template, the final modifications are applied, removing unnecessary template elements that will not be part of the user interface. At the end, all display modes and layout sections are stored in HTML temporary files to serve as the base of the generation of the FUI.

(a) Original template for the `Product` class in display mode `list` of the `eCommerce` domain

(b) Template adapted by the prototype

(c) Template modified by the developer

Figure 4.10: Evolution of a display mode in the *MODUS* prototype

4.7 The FUI Generation

The final user interface can be generated, being this process the last step of the generation of the *MODUS* prototype. As stated in the methodology, the FUI is made out of the views, layouts and partials. The main objective is to materialize each of these components in HTML files.

The first components to be created are the layouts. Each layout will group only its visible sections, as formulated in the creation of the of the UI intermediate components. The layout will be built following a previously established order of sections: `topbar`, `header`, `left-sidebar`, `body`, `right-sidebar`, `footer`, `bottombar`. From these sections, the view's content can only be further inserted in the `left-sidebar`, `right-sidebar` and `body`. They can be considered as empty containers for display modes.

Before starting to elaborate the partials, it is necessary to update the display modes representations. This is a recursive process, where the information about the references of entity display modes is completed. Note that infinite recursion or incoherent processes are properly managed during the generation. Once this stage is concluded, the partials containing only display modes are created, followed by the remaining partials. In this process, the repetitions will assure the insertion of the correct number of the same component, whether it is a display mode or a partial, respecting the stipulated multiplicity. In case the component is associated with a transition to a view, it is surrounded by a HTML hyperlink.

At this point of the generation, the views can be created. First, the HTML file is created with all CSS stylesheets, to set the visual style, and *Javascript* library dependencies, to set front end dynamism, related to the selected front end framework. Second, the file is complemented with the layout associated with the view. Finally, for every section the content is defined, by grouping all the appropriate partials.

4.8 Conclusion

This chapter focused on describing the implementation of a prototype for the *MODUS* approach. The idea was to develop each of the conceptual stages of the methodology presented in Chapter 3, thus demonstrating the viability of the approach.

The *Ecore* Model Extraction defined the interchange format for the UML model as the *ecore* format.

The Standard Classes Identification explored complex implementations in the inference process, through the use of logic programming and software engineering design patterns.

The CSS Framework Manipulation established a generic process based on XML descriptions to stipulate diverse degrees of specification for the front end resources.

The Content and Navigation Map Interpretation defined the variations of the UML state machine necessary to portray a user interface model containing information of both content and navigation.

The UI Intermediate Components Creation stipulated the templates as HTML files complemented with specific data attributes to complement the generation process.

The FUI Generation determined the creation of the final user interface as an iterative process, in which all intermediate interface components are combined to generate FUI.

Most of all this chapter demonstrated, with success, that it is possible to develop a reference implementation for the *MODUS* approach. The existence of a functional prototype is

the first step for corroborating the viability of the methodology. The prototype can thus be further used to conduct detailed tests on the viability of the *MODUS* approach.

Chapter 5

Operating the *MODUS* Prototype

Throughout chapters 3 and 4, the *MODUS* conceptual methodology and its reference implementation were presented. All the necessary premises were formulated to develop the *MODUS* prototype as a plugin for the *Eclipse* IDE. This chapter introduces the *MODUS* prototype, by demonstrating step by step how to generate the user interface from a UML class diagram, namely the case study example presented in Figure 4.2.

5.1 Starting the *MODUS* Tool

To start the execution of the *MODUS* tool, it is required to have a *ecore tools project*, which will be considered as the *MODUS* Project. This project is used to store the tool input, in particular the business logic model (see Figure 4.2), and the generated user interface, as well as some intermediate results obtained by the tool.

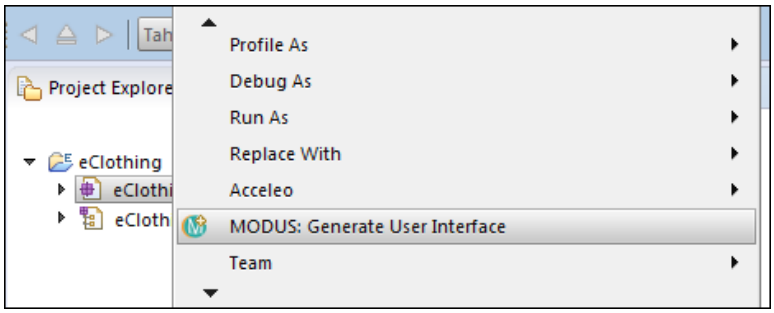


Figure 5.1: *Eclipse* print screen - Starting the *MODUS* tool

Once the class diagram is imported (or built) in the *ecore tools project*, one can select in the context menu associated with the *ecore* format, the option to generate the appropriate UI with the *MODUS* plugin, as shown in Figure 5.1. The selection triggers the reading and validation of the file. If the validation fails, the IDE opens the document with the respective error messages. Figure 5.2 displays an example where the cardinality of the ”articles” relation between ”Cart” and ”Article” was modified to have a lower bound of 4 and an upper bound of 3. As can be observed in the line numbers column of the IDE, the appropriate line is highlighted with an error marker. In the *Problems* panel, the error is developed extensively to explain in detail the warning to the user.

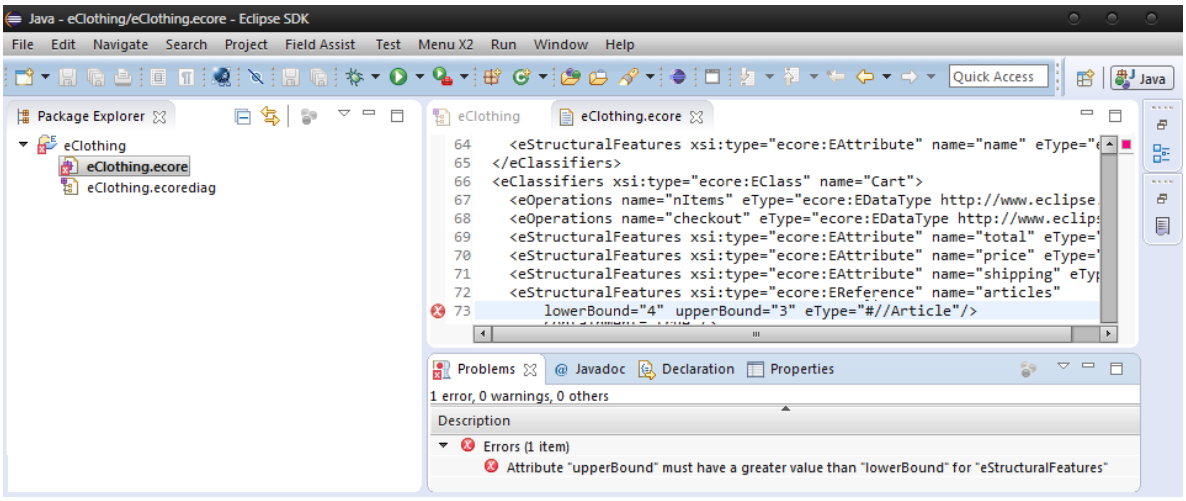
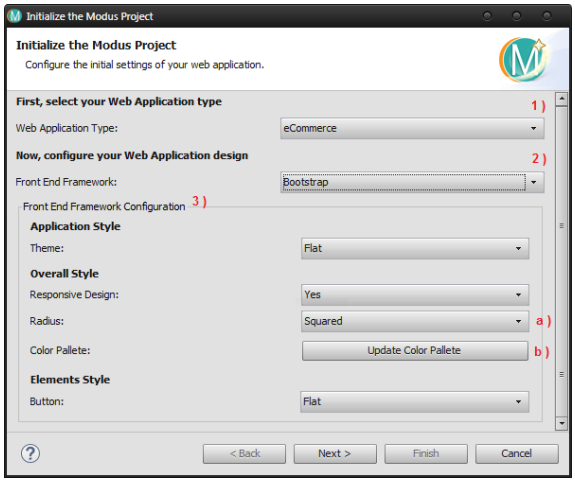


Figure 5.2: *Eclipse* print screen - Error marking in *ecore* file

5.2 Initializing the *MODUS* Project

The first contact with the plugin's user interface is dedicated to the setup of the *MODUS* Project, as exposed in Figure 5.3a. In this menu the users states the tool's input (besides the business logic model): the application domain and the front end resources.



(a) Main Setup

(b) Color Setup

Figure 5.3: The *MODUS* tool : Project Setup

The application domain can be chosen from a list of previously determined domains, as shown in Figure 5.3a 1). The *MODUS* user can select, from a default set 2) , the CSS frame-

work that will complement the generated user interface. If no framework is stipulated, the FUI will be generated as if it was a user interface mockup. This feature allows to obtain a simple outline of the UI, helping the designer to conceptualize the user interface in the first stages of creation. An example is portrayed in Figure 5.4. Otherwise, the fieldset 3) is enabled, allowing the setup of the front end framework. In this case the *Bootstrap* Framework was selected to complement the user interface. An example of resource configuration is demonstrated in a), where the style definition of the overall radius was set to ”squared”.

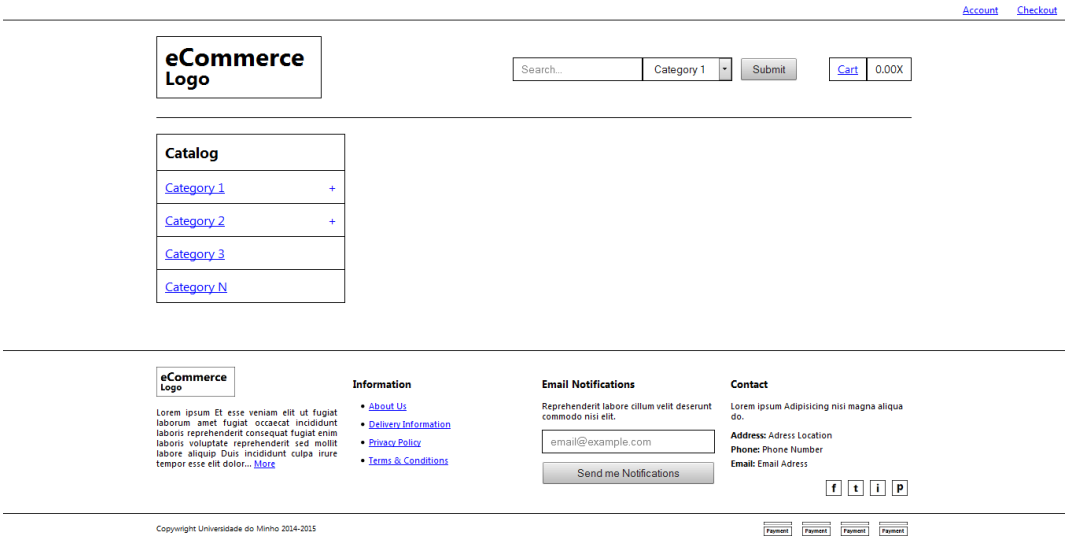


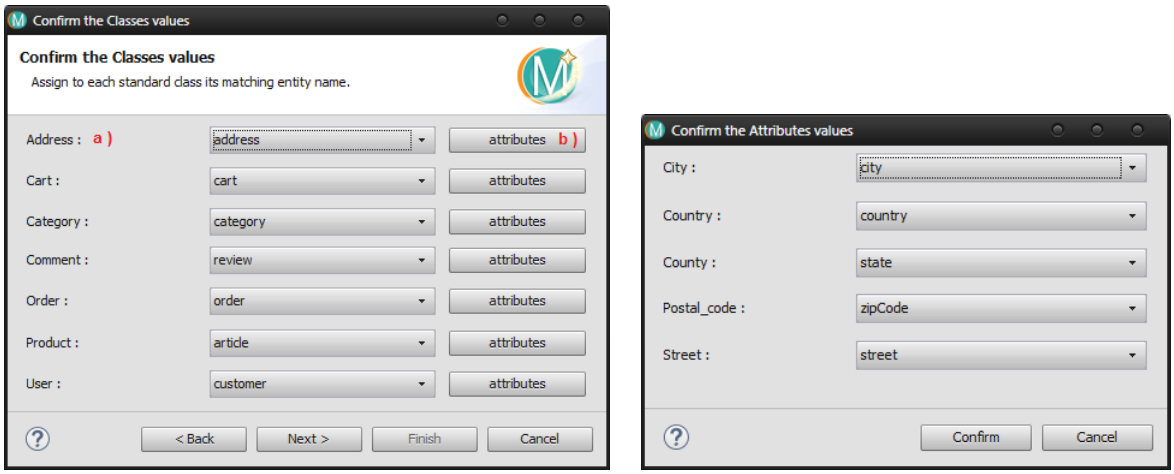
Figure 5.4: Preview of a Mockup generated using *MODUS* (incomplete)

For the CSS framework manipulation it was decided to add a specific configuration definition for the FUI colors. The button ”Update Color Palette” (see b) in Figure 5.3a), displays an auxiliary window to set up the main colors of the application, displayed in Figure 5.3b. Section 1) of the interface, allows the user to set the hexadecimal value for each color, either manually or with the GUI selector. Undefined colors, meaning colors with no valid hexadecimal value, are ignored in the process. In Figure 5.3b the colors were set according to the desired user interface color palette[1]. Meanwhile section 2) presents a search form to obtain sets of color palettes from the *COLOURlovers* [IV, 2004] web service. This search will help the user discover hue patterns that can match the web application style. Once the search is finished, the result is displayed to the user in the fieldset b), each line representing a color palette. The user can then copy the values for one or more color definitions.

[1] A combination of different colors.

5.3 Confirming the Standard Classes Association

Figure 5.5a presents the interface dedicated to the association of the entities of the business model with the standard classes of the respective application domain. The result calculated by the tool is presented to the user in a list of combo box fields set up with the appropriate values. As can be observed, for the model 4.2, the `Address` standard class is associated with the ”Address” entity in a). If the standard class was not associated, the according field would display ”I don’t want to assign anything”. The combo boxes contain the set of entities interpreted from the class diagram plus the empty value, allowing the user to modify the result if intended. The ”attributes” buttons such as b), open a similar window dedicated to the standard class attributes, as shown in Figure 5.5b. Note that both standard classes and attributes are identified and estimated at runtime according to the application domain.



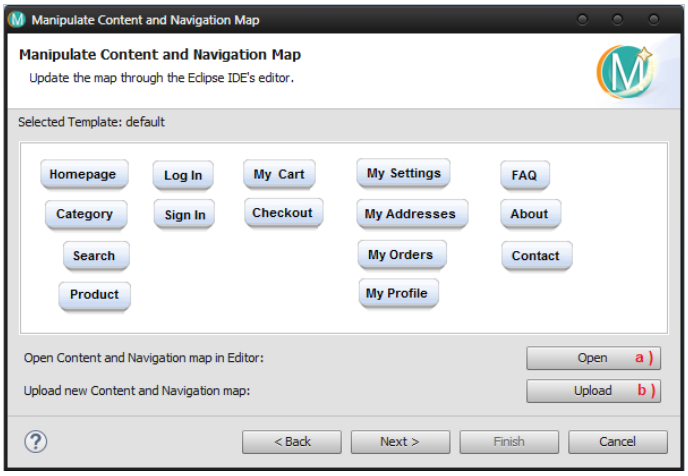
(a) Standard Classes Association (b) Standard Class Attributes Association

Figure 5.5: The *MODUS* tool : Domain Association

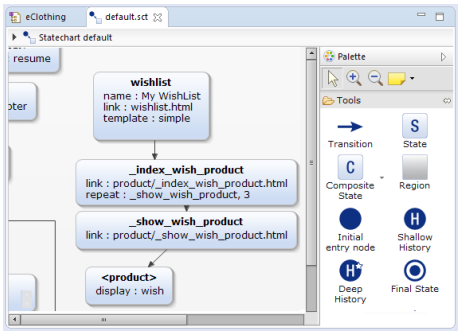
5.4 Manipulating the Content and Navigation Map

The *MODUS* tool looks up for the appropriate content and navigation map associated with the application domain, and does a basic interpretation. Then the user is able to manipulate it to better suit the user interface to be built. These adjustments are performed at

the interface portrayed in Figure 5.6a. First of all, this window of the *MODUS* tool presents the name of the template, as well as a diagram to summarize its content. The diagram lists the views that will be generated at the end of the process according to the content and navigation map, namely: `Homepage`, `Category`, `Search`, `Product`, `Log In`, `Sign In`, `My Cart`, `Checkout`, `My Settings`, `My Addresses`, `My Orders`, `My Profile`, `FAQ`, `About`, `Contact`. This helps the user to have a brief understanding of the content and navigation map used for the generation of the UI. If desired, the user can change the interface model using the Update button a), which triggers the opening of the graphical editor provided by the IDE, as shown in Figure 5.6b. As can be observed, the map was modified to include a "My Wishlist" view, to present the entity "WishList" entity. The user can also import a different content and navigation map from the plugin list, through the Upload button b).



(a) Map Setup



(b) *Eclipse* cropped print screen : Map

Figure 5.6: The *MODUS* tool : Content and Navigation Map Manipulation

5.5 Managing Display Modes

When the plugin interprets the interface model, it generates the required set of display modes, and other intermediate components, to fulfill the generation of the final user interface. Figure 5.7 exposes the interface dedicated to the entities display modes.

The drop down 1) located at the top, allows the developer to select an entity. In the presented example, the "Address" entity is selected. This selection triggers the listing 2) on the

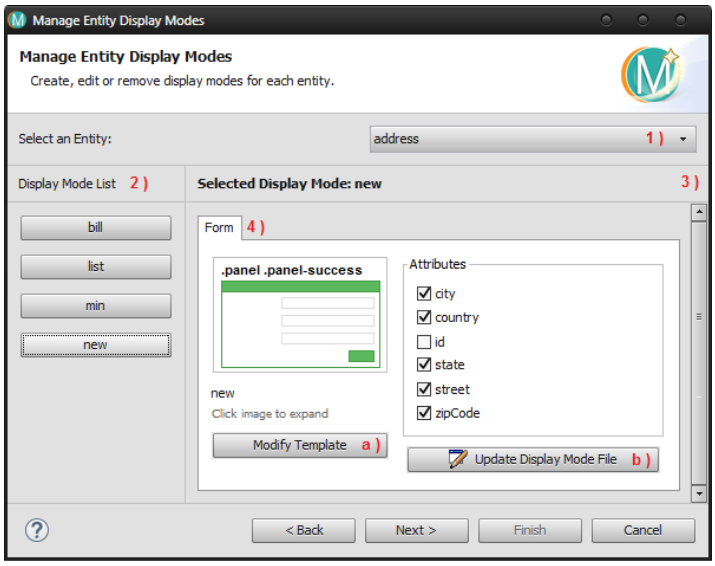


Figure 5.7: The *MODUS* tool : Display Modes Management

left side of the interface, of the display modes associated with the entity. Each mode is listed as a button to enable its selection. Choosing a mode, fills the right side of the interface 3) with the respective information. The tabs 4) provide access to the types of representation available for the selected display mode, which can be either exhibition (`Show`), which displays the entity in the application according to a given mode, or form (`Form`), which portraits the mode's form in the application. In the example of Figure 5.7, the display mode "new" of the entity "Address" is composed only by the type `Form`.

The Modify Template button a) lets the user associate a new template, either by selecting it from a predefined list or by loading it from and external source. In addition to the template information, the allowed entity's components, whether attributes, operations or references, are displayed. Each component is accompanied with a checkbox field, to allow the user to modify its visibility. The reference components have also a combo box field listing the display modes that can be associated with it. In the example of Figure 5.7 operations are not allowed and references are non existing, thus not being present in the list.

The Update Display Mode File button, allows the edition of the mode, in the HTML editor of the IDE. A preview of the display mode is shown in Figure 5.8. The "new" display mode portrays a creation form for the "Address" entity. As can be observed, only the visible elements of the entity (see Figure 5.7) are instantiated, namely the attributes "street", "zipCode", "city", "state" and "country". Each of these attributes is translated as a pair label and input field in the display mode.

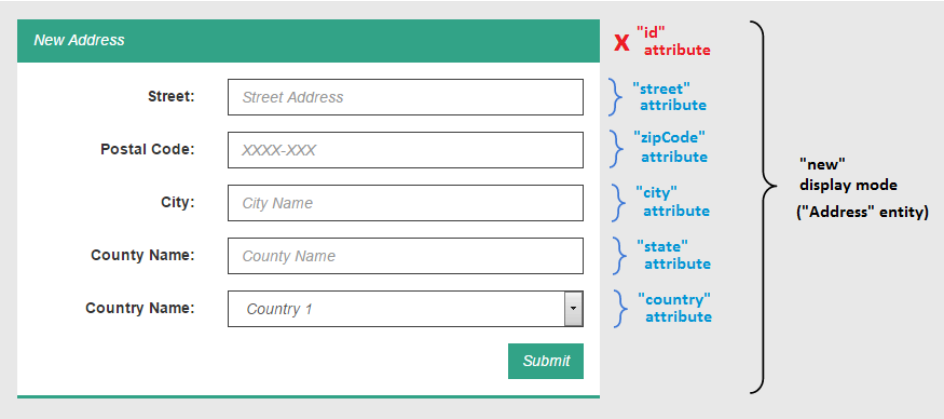


Figure 5.8: Preview of the display mode ”new” for the entity ”Address”

5.6 Managing Layout Sections

As previously mentioned, the *MODUS* tool contains a window for the management of the layout sections, as shown in the preview of Figure 5.9. This window is highly similar to the display modes interface in terms of structure and interaction with the user. In listing 2), the sections are associated with a checkbox field, to define its visibility in the layout. As can be observed, for the template ”default”, the right sidebar was omitted.

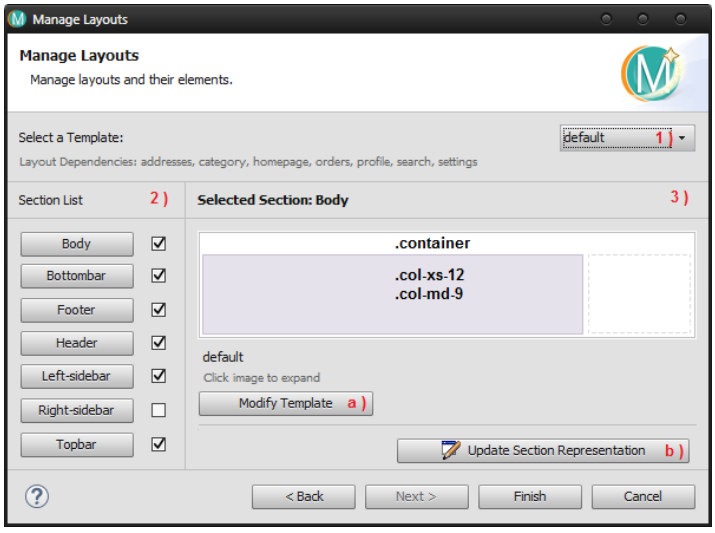


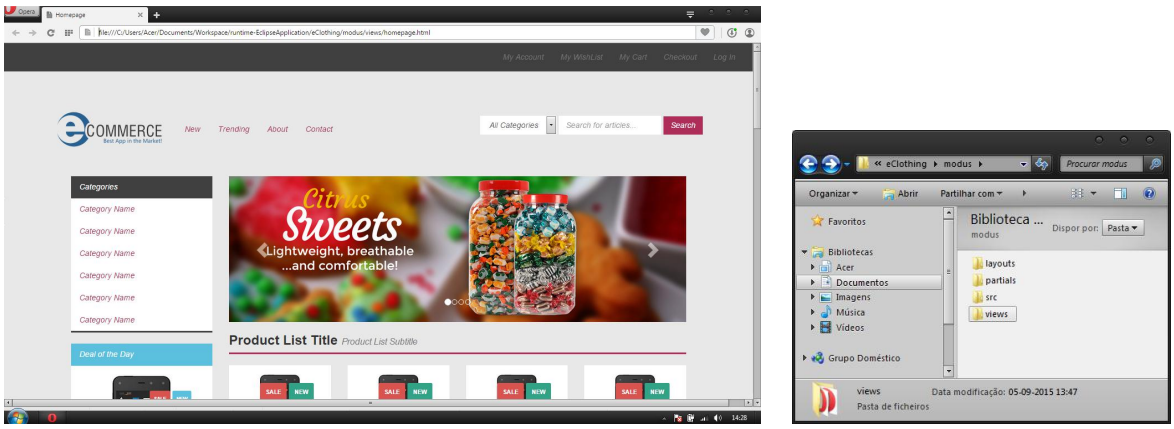
Figure 5.9: The *MODUS* tool : Layout Sections Management

5.7 Opening the Final User Interface

After the configuration of the layouts is complete, the final user interface is built. A ”modus” folder is created in the root of the *ecore tools project* and is then opened by the file manager of the Operative System. This folder, as displayed in Figure 5.10b, will store the final user interface composed by:

- ”layouts” folder, containing all HTML layout files.
- ”partials” folder, containing all HTML partials files.
- ”views” folder, containing all HTML partials files.
- ”src” folder, containing the front end framework files.

Furthermore, the starting view of the content and navigation map is opened in the browser, as pictured in Figure 5.10a. The *MODUS* user can then observe the interface, testing the simulated user interface interaction. For example, while observing the `Homepage` view (see Figure 5.10a) it can be noted that the design of the UI was adapted to the front end configurations defined initially.

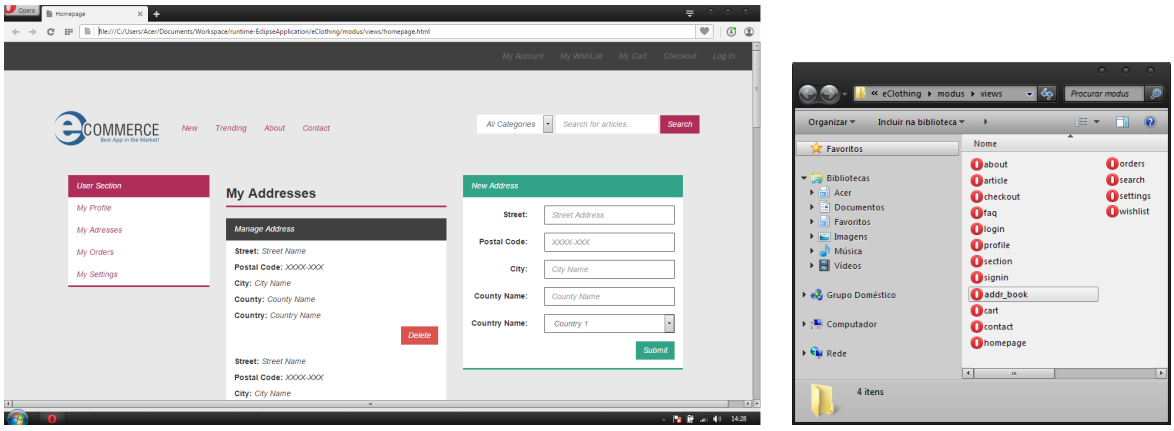


(a) Browser Result (b) Folder Result

Figure 5.10: The *MODUS* tool : Final User Interface

By opening the ”views” folder, as shown in Figure 5.11b, one can confirm the views determined by the content and navigation map, including the view ”My WishList” added to the interface model. In particular, the view `My Addresses` (see Figure 5.11a) will be used to demonstrate the impact of the intermediary results obtained in the *MODUS* prototype. First

of all, this view exists due to the `Address` standard class associated to the ”Address” entity. The ”User Section” element, a common component of `User` related views, has a `My Addresses` hyperlink. Moreover, the view features a listing and a form for the ”Address” entity. In terms of the views’ content, the creation form matches the ”new” display mode of the ”Address” entity as previously previewed in Figure 5.8. In terms of structure, the view does not feature the `right sidebar` section, which is consistent with the modifications applied to this view layout, in this case the ”default” layout. A ”My WishList” hyperlink was also included in the `topbar` section.



(a) Browser Result (`My Addresses` view) (b) Folder Result (”views”)

Figure 5.11: The *MODUS* tool : Final User Interface

5.8 Conclusion

This chapter was dedicated to presenting the operation mode of the *MODUS* prototype. It presented, with a demonstrative example, the achievement of the appropriate user interface, by explaining every stage of the generation process. Therefore, it was shown how to manipulate each intermediate result obtained by the prototype.

Chapter 6
Testing the *MODUS* prototype

This chapter is dedicated to the validation of the *MODUS* process. Three types of analysis were performed. The first analysis focused on validating the main assumptions of the proposal. In particular, it intended to demonstrate that the domain shapes the architectural diagram of an application. The second analysis focused on simulating the generation of the user interface in real life conditions, by testing the approach among a group of different people. The main goal of these two phases was to establish the viability of the *MODUS* approach. The third analysis focused on the evaluation of the user interface generation process. It studied and compared different versions of the same user interface created with different levels of automation, in order to estimate the gains in development costs and to analyze the obtained results.

6.1 The Assumptions Validation

One of the main assumptions of the thesis approach is that different applications of the same domain tend to follow navigational, structural and content patterns. More specifically, one can identify elements within the class diagrams that are strictly related to the application domain. To demonstrate and corroborate that assumption a survey was performed where a group of people had to develop UML models for a given domain. This survey was later on used to validate the *MODUS* approach, as seen in Section .

6.1.1 The Study

At the beginning of the development of the *MODUS* approach it was decided to focus the attention on one application domain, the *eCommerce* domain. Two main reasons led to selecting *eCommerce*. First, it is a domain with a reasonably complex level in terms of content and functionality. This makes it possible to demonstrate that the approach is applicable to challenging applications and allowing the collection of more interesting results. Second, it is a relatively well-known domain to web application developers. This should allow the collection of more accurate results in investigations and inquiries to be made.

Considering that, in the *MODUS* process, the main idea is to bind the application domain to the class diagram, it was decided to conduct a survey about *eCommerce* class diagrams.

The data collected helped diagnose how people perceive this particular web application domain, contributing to the development of the approach.

The main objective of the survey was then to identify how web application developers conceptualize a default model of the *eCommerce* domain. The survey required that the participants create a class diagram of a generic *eCommerce* application. It was intended to gather data about the class diagrams' content, in terms of classes and their relationships, between different representations of the same application. The main survey was composed of two different surveys, whose results were later combined for further analysis. The first survey was meant to study the standard classes and relationship patterns. The second survey was meant to identify the common attributes present in standard classes.

6.1.2 Setup of the Study

The study was conducted among a group of 10 participants. The surveys were conducted either online or face to face with the participants, being the results delivered either by email or paper. Participants did not contact each other during the study, and responses were developed independently. However the attainment of the answers was not isolated from any source of external information, such as the Internet.

The attendees were classified according to different characteristics: age range, degree of experience with UML, academic and professional background.

- In terms of **Age Range**: 40% were aged between 21 and 23 years old and 60% were aged between 24 and 26 years old.
- In terms of **UML's Degree of Formation/Experience**: 50% had formal education in UML, 10% had knowledge of UML and 40% had no knowledge of UML.
- In terms of **Academical Background**: 20% had a Bachelor Degree in Computer Science with no Master Degree, 50% had both Bachelor and Master Degree in Computer Engineering, 20% had a Bachelor Degree in Computer Science and a Master Degree in Computer Engineering and 10% had a Master Degree in Industrial Electronics and Computer Engineering.
- In terms of **Professional Background**: 70% had professional background and 30% had no professional background.

The first survey required the creation of a class diagram consisting of only classes and their relationships. The second survey was created as a questionnaire, see Appendix A.1, for the identification of both attribute names and types for a set of presented classes. To complement both surveys, the author developed class diagrams to represent *eCommerce* applications known as standard trading applications, namely "Amazon"[1] and "LightInTheBox"[2]. This would help to compare the assumptions taken upon a generic and standard application for the presented domain.

6.1.3 Study Results

By analyzing the content of the various models, it was possible to recognize certain recurring patterns regarding the application domain. The occurrence frequency was first calculated for each pattern identified. Only results achieving an occurrence frequency higher than or equal to 40% were considered. After filtering the entities and grouping them in a single class, a set of standard of classes was defined:

`User` - a person registered in the application, which uses/intends to use it;

`Product` - any good or service that can be traded in the application;

`Order` - the transaction of acquiring one or more products in the application;

`Shopping Cart` - a collection of products, which the user intends to purchase;

`Category` - a specific division of products;

`Comment` - a written statement, criticism or opinion;

`Address` - description of a user location;

The percentage of use of each standard class is presented in Table 6.1. Note that almost 30% of the standard classes in the models have a frequency of 100%, and above 85% of the standard classes in the models have a frequency higher than 50%. The results do corroborate the existence of recurring generic classes for a certain domain, as assumed in the *MODUS* approach.

[1] http://www.amazon.com - last visited on April 6, 2015
[2] http://www.lightinthebox.com - last visited on April 6, 2015

Standard Class	**Frequency (%)**
`User`	100
`Product`	100
`Order`	90
`Shopping Cart`	80
`Category`	70
`Comment`	60
`Address`	40

Table 6.1: Percentage of use of each main entity in the modelings

Studying the models also allowed to detect a vast set of relationships patterns, common relations between standard classes of the same domain, for each class. The relationship patterns are presented in the form of a modified UML class diagram. In this version of the model, regular expression operators are allowed on the cardinality of the relationships to display alternative values in the same relation. Once again it can be observed that there are common relationship patterns for the same domain.

All relationship patterns between standard classes are listed in Appendix A.2. As an example, the `Address Pattern` depicted in Figure 6.1 can be described as follows: an `Order` has one `Address`; a `User` has one or has at least one `Address`.

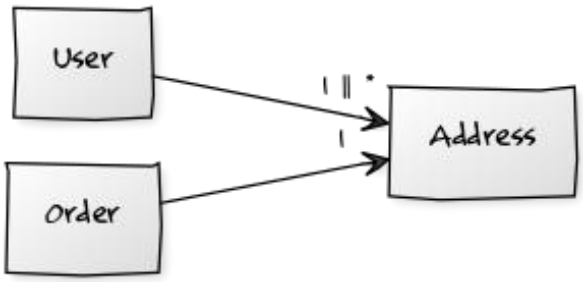


Figure 6.1: Representation of the `Address Pattern`

Once the standard classes were identified, the second survey was carried out to identify for each standard class a set of common attributes, matched by name and type, as presented in Table 6.2.

Standard Class	**Standard Attribute**	**Type**	**Frequency(%)**
Category	Name	String	92
Comment	Date	String	50
Comment	Name	String	92
Comment	Rating	String	42
Order	Total	Integer/Float	75
Product	Name	String	92
Product	Description	String	75
Product	Price	Integer/Float	83
Product	Quantity	Integer	42
Shopping Cart	Total	Integer/Float	50
User	Birthday	Date	50
User	Email	String	75
User	Name	String	58
User	Password	String	67
User	Username	String	83
Address	Street	String	100
Address	Country	String	75
Address	City	String	75
Address	Zip Code	String	67
Address	County	String	42

Table 6.2: Percentage of use of each attribute of the Order entity

6.2 The Survey Tests

The survey was reused to further prove the viability the *MODUS* approach. This study involved testing the generation process on each of the models obtained by complementing the information about the classes with the data about the attributes. Each participant generated, from the class diagram, the desired user interface using the *MODUS* prototype. Then, the

attendees answered two different questionnaires about the FUI produced: a questionnaire, to qualify the efficiency of the approach and an additional questionnaire, to quantify the usability of the user interface generated with *MODUS*.

6.2.1 The Efficiency Results

The first questionnaire intended to test the efficiency of the *MODUS* approach. It essentially intended to examine if the user interface was produced according the provided UML business logic model. This was achieved by quantifying the degree of satisfaction of the participants with the results. The questionnaire was made of 5 items and was answered on a scale from 1 (strongly disagree) to 5 (strongly agree), or ”N/A” (no answer). The results are displayed in Table 6.3, reaching an average result of 4.4 in 5.

Questionnaire Item	**Answer**	**”N/A”**
The UI achieved the user expectations considering the provided UML model	4.5	0%
The UI allowed to identify issues and inconsistencies in the provided UML model	4.6	20%
The UI efficiently simulated a typical user interface for the defined application domain	4.5	0%
The UI is comprehensive in terms of content according to the provided UML model	4.5	0%
The UI is comprehensive in terms of navigation according to the provided UML model	4.4	0%

Table 6.3: Average Answers to Efficiency Questionnaire

Rounding up, the participants tended to completely agree that the produced user interface met the requirements matching the UML model, whether in terms of user expectations, UI content and navigation. They also tended to completely agree that the user interface efficiently portrays a common UI for a specific domain. Thus proving the validity of the user interface generation process based on a singular architectural model of the business logic and the definition of the application domain. Moreover, 80% of the participants tended to completely agree that, the *MODUS* prototype allowed the validation of the UML model itself.

The *MODUS* prototype can be used as a complementary tool for the software developers. This would help to diagnose potential problems of the solution for the application's implementation, and thus would accelerate the development process.

6.2.2 The Usability Results

Usability was tested with a Post-Study System Usability Questionnaire (PSSUQ) [Lewis, 2002], applied in the UI produced for each model. A PSSUQ is composed of 19 different items, which are answered using 1 (strongly disagree) to 5 (strongly agree) or "N/A" (no answer). The results, presented in Table 6.4, estimated an average of 4.15 (4 if rounded) in 5, which indicates that developers tend to agree to a usable interface. Complementing this data with the efficiency test, one can deduce that the developers are positively satisfied with the *MODUS* approach.

6.2.3 Threats to Validity

In the course of the survey, factors may have influenced the validity of the results. First, each inquiry was not run simultaneously. Although the participants did not interact with each other while answering the survey, there was no surveillance regarding posterior and anterior contacts. The study could have been better monitored and contained, in order to ensure pristine results. A greater control of the interaction and a limitation of external sources, would have been necessary. Second, the survey was conducted with a representative sample of software developers. A larger number of participants could have provided more accurate results. Furthermore, testing the prototype among UI experts could have derived more severe results.

PSSUQ Item	**Answer**	**”N/A”**
Overall, I am satisfied with how easy it is to use this system	4.9	0%
It was simple to use this system	4.5	0%
I could effectively complete the tasks and scenarios using this system	4.4	0%
I was able to complete the tasks and scenarios quickly using this system	4.5	0%
I was able to efficiently complete the tasks and scenarios using this system	4.4	0%
I felt comfortable using this system	4.6	0%
It was easy to learn to use this system	4.6	20%
I believe I could become productive quickly using this system	4.5	0%
The system gave error messages that clearly told me how to fix problems	3.7	30%
Whenever I made a mistake using the system, I could recover easily and quickly	4.5	40%
The information (such as on-line help, on-screen messages, and other documentation) provided with this system was clear	4.4	20%
It was easy to find the information I needed	4.3	0%
The information provided for the system was easy to understand	4.4	0%
The information was effective in helping me complete the tasks and scenarios	4.6	10%
The organization of information on the system screens was clear	4.2	0%
The interface of this system was pleasant	4.4	0%
I liked using the interface of this system	4.3	0%
This system has all the functions and capabilities I expect it to have	4.1	0%
Overall, I am satisfied with this system	4.2	0%

Table 6.4: Average Answers to the PSSUQ Items applied to the Survey

6.2.4 Identifying the Levels of Automation

The third analysis conducted for the validation of the MODUS approach was focused on a case study. Using the model presented in Figure 4.2, the goal was to test the quality of the output produced using different levels of automation:

- **manual**, a user interface fully coded by the developer;
- **fully automated**, a user interface generated by the prototype without any developer intervention
- **partially automated**, a user interface generated by the prototype with some developer intervention;

This analysis intended to compare the three perspectives balancing the pros and cons of each one, drawing conclusions about the level of automation. For the partially automated solution, meaning the desired final user interface, usability tests were performed.

All versions of the case study user interface were produced by the author, which has strong design skills in the development of browser-based user interfaces and experience in working with *Bootstrap*. Furthermore the author is, obviously, accustomed to using the *MODUS* prototype to generate user interfaces. Note that these factors influence the development costs estimated for the case study, especially in terms of time spent and produced code.

6.3 The Case Study Analysis

6.3.1 Identifying the Levels of Automation

The third analysis conducted for the validation of the MODUS approach was focused on a case study. Using the model presented in Figure 4.2, the goal was to test the quality of the output produced using different levels of automation:

- **manual**, a user interface fully coded by the developer;
- **fully automated**, a user interface generated by the prototype without any developer intervention
- **partially automated**, a user interface generated by the prototype with some developer intervention;

This analysis intended to compare the three perspectives balancing the pros and cons of each one, drawing conclusions about the level of automation. For the partially automated solution, meaning the desired final user interface, usability tests were performed.

All versions of the case study user interface were produced by the author, which has strong design skills in the development of browser-based user interfaces and experience in working with *Bootstrap*. Furthermore the author is, obviously, accustomed to using the *MODUS* prototype to generate user interfaces. Note that these factors influence the development costs estimated for the case study, especially in terms of time spent and produced code.

6.3.2 Understanding the Class Diagram

The class diagram of the case study was presented in Figure 4.2. This model represents the intended implementation of the web application ”eClothing” for the online retail of clothing for women. The application was created so it could follow common patterns found in the *eCommerce* domain, but also so it could specify some particular features of the *Fashion* sub-domain. There was an interest in creating a challenging example, in terms of modeling and expected result. The expected result should be a modern site with a ”flat” visual style, following the themes available on website templates' marketplaces such as ”Themeforest”[3]. In terms of its content and functionality, the user interface was based on ”Amazon” and ”LightInTheBox”.

The model itself was designed to be fairly complex, demonstrating inheritance between classes, a wide range of attributes, a strong presence of operations, among others. In addition, it was structured attempting to hinder the standard element recognition processes, at both class and attribute levels. For example, introducing elements that follow the same relationship patterns, mixing attributes and operations, having elements with similar names, and so on. The inclusion of non standard classes also allowed to test the creation of brand new elements of the prototype, from views to display modes. Overall, the case study class diagram allowed the analysis of the extent to which the interface can be complete, and to estimate the development efforts required to create the desired UI.

6.3.3 The ”Manual” User Interface Generation

The ”manual” user interface represented the expected final user interface. It was the basis for the comparison with the interfaces generated by the *MODUS* prototype in terms

[3]http://themeforest.net - last visited on April 6, 2015

of development effort. Furthermore, it was the point of reference for testing the levels of resemblance between the desired FUI and either the fully or partially automated interfaces.

For a fast development and easy maintenance it was decided to implement the desired user interface with the front end framework *Bootstrap*. The main front end specifications defining its design were listed as follows:

- ”flat” theme for the user interface, including all of its elements, from forms to widgets;
- responsive design for the user interface;
- squared borders for the overall design of the user interface;

The *Bootstrap* main website allows the download of a custom version of the framework, where the developer is able to set a number of different resource configurations. Thus, two versions of the framework have been used to implement the ”manual” interface: the default version, a pristine version of the framework, and a custom version, personalized according to the front end specifications of the case study. Working with these two different releases of the framework helped to study, in greater detail, the gains in design customization. Table 6.6 portrays the number of CSS code lines used to configure *Bootstrap*, in order to obtain the desired design for the UI. Note that only the CSS code lines that influenced the visible design of the application's interface were considered. As Table 6.6 demonstrates, the custom version of *Bootstrap* requires less amount of CSS code lines, however the improvement barely reaches 16%. Still, establishing the design from scratch allows the developer or designer to have more control over the CSS styles, specially if the user interface requires a very specific appearance. Besides, the creator of the user interface has a greater feeling of creative freedom.

Framework	**Number of CSS Code Lines**
Pure *Bootstrap*	758
Custom *Bootstrap*	637
Total Gain in terms of CSS Code Lines	121

Table 6.5: Number of CSS code lines complementary of the *Bootstrap* Framework

To build the overall user interface, from the front end framework manipulation to the creation of the views it was necessary to spend approximately 40 hours. The prototyping of

the user interface was not taken into account since it would be applied to the FUI generated from all degrees of automation. Due to the freedom of creativity, the user interface was iteratively changed to obtain the desired result. The problem being that these changes need to be singularly updated throughout the views. Some inconsistencies may even end up not being resolved, leaving and incoherent UI. To develop the desired user interface 16 views were developed, 15 of which are standard in the *MODUS* generation process. Table 6.6 presents the number of HTML code lines required to develop each view, reaching a total of 7385 lines. Note that the HTML code was written according to the *Bootstrap* framework, thus influencing the total number of lines. Using a different framework or even no framework at all may interfere with the amount of HTML code lines needed to produce the required result.

User Interface Page	**Number of HTML Code Lines**
`About`	307
`Cart`	332
`Category`	1018
`Checkout`	446
`Contact`	255
`FAQ`	239
`Homepage`	1150
`Log In`	256
`My Addresses`	317
`My Profile`	266
`My Settings`	341
`Orders`	303
`Product`	607
`Search`	1025
`Sign In`	252
My WishList	271
All	7385

Table 6.6: Number of HTML code lines of the user interface

6.3.4 The ”Fully Automated” User Interface Generation

The ”fully automated” user interface demonstrated the absolute default UI that can be obtained from the *MODUS* prototype without any interaction with the developer. This interface is to be compared, in terms of visual aspect (design, structure, content), with the result achieved with a smaller degree of automation. This comparison allows the study of the level of detail that can be achieved trough iterative manipulation. Note that the generation of this user interface was almost immediate, only requiring to set up the front end framework as *Bootstrap*. No other modification was applied in any of the stages of the generation process. Therefore, no relevant data was collected, besides the final user interface itself.

6.3.5 The ”Partially Automated” User Interface Generation

The ”partially automated” user interface was achieved by balancing the automated result with the developer interaction. The goal was to recreate, as much as possible, the expected interface using the *MODUS* prototype. Overall, the generation of the final user interface required less than 30 minutes of development time, which is 1% of the time spent in the creation of the ”manual” version. It appears that there are significant gains in the production time of the user interface. Nonetheless, to study the development efforts in detail it was necessary to analyze the generation process step by step.

Despite the reduced number of style definitions provided by prototype, the manipulation of the CSS framework allowed to set up all desired front end configurations. Only a small amount of CSS code (less than 5%) was added to complement the framework. One can consider that there is almost a 95% gain in terms of CSS development costs. If few style definitions allow the prototype to produce these great results, adding new definitions would certainly have a meaningful impact on the level of detail of the generated user interfaces.

To set the correct structure of the expected user interface, in terms of views and their contents, some modifications in the content and navigation map were required. These modifications are identified in Table 6.7, and can be classified as: ”create”, creating a new component; ”update”, updating an existing component; ”remove”, removing and existing component. In this context a component is an element of the content and navigation map, being either a view, partial, display mode or navigation. A total of 25 modifications were applied, which, taking into account the complexity of the user interface model, can be considered a relatively small number. Furthermore, the shared components will ensure an uniform upgrade of the map, ensuring its consistency.

User Interface View	Number of Actions	Action	Interface Model Component
`Account`	1	create	Display Mode
`Checkout`	1	create	Display Mode
`Checkout`	3	create	Navigation
`Checkout`	1	create	View
`Checkout`	1	update	Display Mode
`Checkout`	1	update	Partial
`Checkout`	1	update	View
WishList	1	create	Display Mode
WishList	2	create	Partial
WishList	1	create	View
WishList	3	create	Navigation
`Homepage`	1	create	Display Mode
`Homepage`	1	create	Partial
`Homepage`	2	create	Navigation
`Homepage`	1	remove	Navigation
`Homepage`	1	update	View
`Search`	1	update	Partial
`Product`	1	remove	Navigation
`Product`	1	update	View
All	25		

Table 6.7: Content and Navigation Map Updates

To achieve the visual outlook of the final user interface, it was necessary to apply changes to the display modes and layout sections. In order to update each of the UI intermediate components, the developer used mostly HTML templates with the appropriate CSS classes associated with the selected front end framework. The modifications for the display modes and layout sections are respectively presented in Table 6.8, listing each entity/standard class display mode updates, and Table 6.9, listing the layouts sections updates. In general there are

three reasons that have led to the need for these changes:

- the original template may differ from the desired output;
- the original template may limit the insertion of some class elements, meaning attributes, operations or references;
- the original template was generic, requiring more thorough changes;

Entity Class	**Display Mode**	**Number of Code Lines**
`Customer`	simple	1
`Contact`	panel	1
`Contact`	complete	25
`Article`	wish	15
`Article`	sort	1
`Article`	filter	11
`Article`	row	5
`Article`	complete	12
`Article`	daily deal	5
`Article`	best sellers	3
No Standard Class	app team	15
No Standard Class	app brands	10
No Standard Class	contact info	2
No Standard Class	checkout register	6
No Standard Class	promotion grid	5
No Standard Class	shipping	5
No Standard Class	user nav	15
No Standard Class	tag grid	7
All		144

Table 6.8: Number of updated code lines in display modes

Section	Number of Code Lines
`Bottombar`	3
`Header`	12
`Topbar`	8
All	23

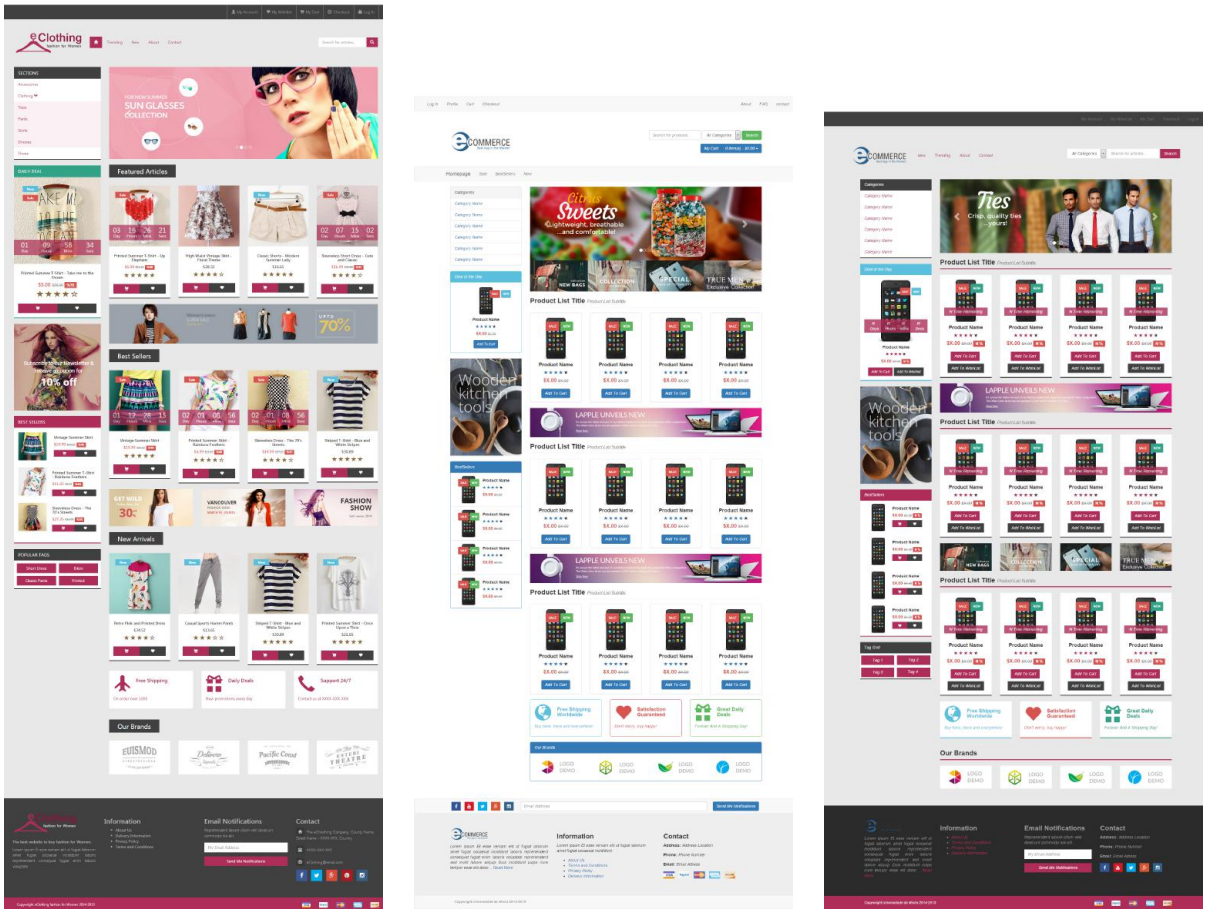
Table 6.9: Number of updated code lines in display modes

As previously discussed the intermediate components are limited to the existing templates, which limits the designer control over the user interface. Still, the tool is expansible and allows the developer to upload external templates during the generation. Above all, there is a huge gain in development time. In fact, the adjustment of the templates allowed a quicker creation process. It required only a total of 167 HTML line updates, which is 2% of the HTML code lines (2% from display modes and almost 0% from layout sections) that had to be written in the ”manual” user interface version. The preview in the IDE editor of the current state of the intermediate component allows the developer to easily identify changes, thus also accelerating the generation process. At last, as for the content and navigation map, every change will be applied to all dependent elements.

6.3.6 Comparing the Case Study User Interfaces

Summing up, there was a gain of 99% on development time, 98% and 95% of development efforts, respectively in terms of produced HTML and CSS code lines. It is however necessary to keep in mind that, in the case study, a lot of different factors influence the results: from the framework and templates used to the developer skills and habits. Overall, one can easily conclude that the *MODUS* approach does facilitate and accelerate the development of user interfaces for the domain in question.

After identifying the gains in terms of development costs, the potential of the generation process was analyzed. In other words, it was necessary to investigate how advanced the user interface generated using the prototype can be. All three versions of each user interface's view are compared side by side. The `Homepage` view, pictured in Figure 6.2, was used as the basis for the comparison of the case study interfaces. Appendix A.3 contains all comparisons for each of the 15 views generated for the user interface.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 6.2: User interfaces generated for the case study : `Homepage` View

Let’s start the analysis by comparing the ”fully automated” and the ”partially automated” versions. The first aspect that immediately catches the attention is the design. The ”fully automated” user interface almost looks like a website template provided by the *Bootstrap* framework. The style definitions applied in the ”partially automated” UI had a huge impact on the visual aspect of the output created. Then one can notice that the order of the elements of the user interface has changed. For example, one of the *promotional banners* was replaced by the *promotional grid of images*. Only 6 modifications to the content and navigation map were needed to create the desired structure for this view. Finally, the view’s content was changed: the layout is different and many elements were amended. Still, only a total of 73 code line updates of the UI intermediate components were required to produce the expected content of this view. One can observe that these versions share similarities as a whole, nevertheless the ”partially automated” version has some unique and detailed features, particular

to its design. By comparing the ”partially automated”’ with the ”manual” user interfaces one can notice that the versions are highly similar in terms of look & feel, structure and content. Actually, the main difference in the two UI was the images used.

6.4 Conclusion

The chapter was centered around testing the viability of the *MODUS* prototype. At first, a survey was run among a set of participants for the domain in the study, to demonstrate the viability of the approach. The results corroborated with many assumptions taken in the methodology. By resorting to PSSUQ tests, it was also evidenced that a positively usable UI can be obtained using *MODUS*. Furthermore, an additional inquiry related to the prototype results was conducted, contributing to validate the efficiency of the approach. A case study was then used to evaluate the generation process, by comparing the same user interface generated with different levels of automation: with no automation, fully automated and partially automated. Overall, it was proven that the approach could create with significant precision the expected user interface, with a very small amount of development effort. A PSSUQ test was also applied to the partially automated user interface, demonstrating a positively usable user interface, this time in more contained conditions of testing.

Chapter 7
Conclusion

This chapter discusses the work carried in this dissertation. It presents a general overview of the *MODUS* approach, from the outline process to the implementation of a supporting prototype. Then, it describes the future work, to be developed in order to further complete or improve the approach.

7.1 Discussion

In nowadays technology, the User Interface is essential for the success of any application, being the bridge of interaction with the user. And yet the implementation of an interface implies heavy development costs for software developers. Model-based Tools are designed to ensure the quality of the user interface, while reducing production time, through the introduction of automation in their generation process. These tools usually resort to many detailed user interface models, limiting the acceptation of the model-based approach. Furthermore, the use of automation tend to have a negative impact on the user interface produced and, more specifically, a weak integration with the software development process.

The main goal of this dissertation was to develop a model-based tool for the generation of Web User Interfaces, exploring an automatic creation process directly based on structural models of the business logic: the *MOdel-based Developed User Systems* approach, known as *MODUS*. In this approach, an high level of automation was assured by assumptions taken from the identification of the application domain. Thus, exploring the fact that user interfaces of applications belonging to the same domain tend to follow patterns, in terms of content, structure and navigation. The degree of automation can however be balanced with the developer intervention, in order to create more appealing and flexible user interfaces. To further increase the appearance of the user interface, the generation process integrates the configuration of front end resources. These unique features presented by *MODUS* intend to differentiate it from the several approaches presented in the thesis's state of the art, on Chapter 2.

The manipulation of a CSS Framework enables the developer to easily configure the visual appearance of the user interface. Each setting is represented as a style definition, from the style to be applied to the code updates to be executed in the front-end framework to

enforce the style.

The identification of standard classes enables the recognition of relevant entities of the application domain, in the forms of classes and their attributes present in the class diagram. The identification process is based on checking relationship patterns of those classes, complemented by semantic matching of their names using dictionaries of synonyms. In the inference process the relationship patterns were implemented as either independent *Prolog* patterns or hard coded *Java* patterns. Resorting only to *Prolog* patterns allows an easy update of the inference mechanism, assuring a quicker integration of new patterns from the same or other application domains. Resorting to *Java* patterns proved a faster inference mechanism but difficult to maintain over time.

The establishment of a content and navigation map supports the definition of a typical user interface for a certain application domain, unifying this information in a single user interface model. This map was implemented as a modified version of the UML state machine diagram, to allow to express both content and navigation. As it is provided by the tool, it greatly reduces the development time of the outline of the user interface. Still, resorting to the *Yakindu StateChart Tools*, revealed in the long term to be an inefficient solution.

The creation of the user interface intermediate components originates, from domain based templates, all pieces that will be composed to create the final user interface. A template based creation mechanism establishes a generic implementation process for every application domain and front-end framework selected. This mechanism also proves to be easily maintainable and extensible, being a good solution for the field of user interfaces. However, to assure the flexibility of the generated results, it must support a vast and varied set of templates.

The generation of the final user interface assembles the intermediate components in order to create the desired interface. It iteratively interprets each component to create each element of the final user interface, namely the views, partials and layouts. The production of the views allows to simulate the user interface as a whole. By isolating the partials and layouts from the views, it improves integration between software back-end and front-end implementations.

MODUS was implemented as a plugin for the *Eclipse* IDE, being developed in the *Java* programming language. The development of a functional reference implementation for *MODUS* has demonstrated the viability of the approach. Besides being a proof of concept, the prototype was also essential for testing the approach with real life users. A survey was applied to a set of participants, composed by two distinct type of inquiries: usability tests to complement the previous analysis and efficiency tests to validate the *MODUS* approach itself. The usability of the produced user interface was proved positive through the use of a Post-Study

System Usability Questionnaire. Finally, the efficiency questionnaire returned an overall satisfactory outcome. The prototype was also tested with an extensive case study to analyze the generation the user interface from an architectural model and the domain of an application.

While developing the *MODUS* prototype, the author wrote the scientific article "MODUS: uma metodologia de prototipagem de interfaces baseadas em modelos" [Marina Machado, 2015] for the INFORUM 2015 symposium in Portugal. The article presents the approach as a suitable tool for the fast development of user interfaces. Furthermore, it was essential to promote the *MODUS* potential as a model-based tool by its unique features.

7.2 Future Work

The implementation of the *MODUS* approach demonstrated a great potential for the model-based generation of user interfaces. The MODUS prototype can, however, benefit from the implementation of some improvements. By studying the results obtained from the prototype one can decide to either upgrade or add new features in order to overall perfect the tool. It is hoped that, in the future, the *MODUS* tool might be published as a fully functional *Eclipse* plugin, available to the open source software community.

The first step to improve the tool would be integrating more application domains. To ease this process, the tool should focus on an inference engine based solely on *Prolog* patterns. One needs to investigate a solution to accelerate the engine deduction process. The data about the application domains should be collected on a large scale to assure more detailed information about the standard classes. This might be achieved by either a survey applied to a large group of people or by elaborating an engine for automatic data collection for a given set of sites for a domain.

Furthermore, the *MODUS* tool could integrate more CSS frameworks, like *Foundation* [Foundation, 1998], *UIKit* [YOOtheme, 2007] among others. Above all, it should define more style definitions, in terms of settings and values. This would greatly increase the level of detail in which one can configure the design of the interface to be generated. To fulfill this improvement it would be required to develop or update the files provided by the tool, such as the HTML templates for the intermediate components. In a future in which the tool is complete and stable, it would be interesting to introduce the concept of sub-domain in certain application domains, to produce even more accurate results.

The integration of front-end frameworks tends to have issues in the generation of user interface forms, due to their unique HTML templates and CSS classes. The automated gen-

eration of forms needs to be improved to suit every front-end framework used. To do so, one could incorporate one or more template defining the structure of a form associated with each framework. In terms of templates in general, one could also implement a mechanism to import valid community made templates, to increment the diversity of the tool. This could only be achieved if *MODUS* would be associated to some site featuring a fan community, providing an API to access the template files.

The content and navigation map could be further improved, to solve some issues previously presented. The first solution would be dividing the content and navigation maps per view, in order to reduce the size of the user interface model. Yet, this would decrease the level of element reuse of the model, and would also hinder the definition of the navigation between views. The second solution would be replacing the format of the state machine model by a IFML diagram. In view of the studies conducted on IFML during this dissertation, the user interface model would become shorter and more readable. Since, in this context, the IFML diagram is less expressive, there would be loss of information. Some complementary data could contained in another file, however this would diverge from the concept of a singular interface model.

The *MODUS* approach was meant to focus on the development of the user interface for web applications, not converging to any specific technology in terms on either front and back-end implementation. Nevertheless the approach was developed to easily be prepared to integrate business logic, in particular front-end business logic. The approach could optionally resort to front-end technologies such as *AngularJS*, which tend to be compatible with the CSS frameworks used in the generation process.

Appendices

A.1 The Survey Standard Attributes Questionnaire

Listing 1: The Survey Standard Attributes Questionnaire

```
For each of the presented classes, indicate the name and type
of attributes
that usually tends to exist for a generic eCommerce web application.
Please answer as follows:

CLASS NAME

"attribute name" - "attribute type"
...

Do not worry if the attributes names are written in either
Portuguese or English.
Please save the questionnaire file as follows: survey_"name".txt,
eg "survey_marina.txt".
I thank you for your participation :).
-------------------------------

USER

PRODUCT

CART

ORDER

ADDRESS

CATEGORY

COMMENT
```

A.2 Survey Standard Classes Relationship Patterns

In definition the `Product Pattern` depicted in Figure 1 can be described as follows: a `Product` has many `Comments`; a `Product` has one or at least one `Category`; a `Category` has many `Products`; a `Shopping Cart` has many or at least one `Product`.

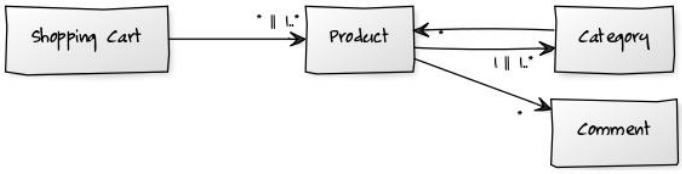


Figure 1: Representation of the `Product Pattern`

The `Order` standard class has two different patterns, depicted in Figure 2. In definition the first `Order Pattern` can be described as follows: a `User` has many `Orders`; a `Order` has one `Shopping Cart`. In definition the second `Order Pattern` can be described as follows: a `User` has many `Orders`; a `Order` has at least one `Product`.

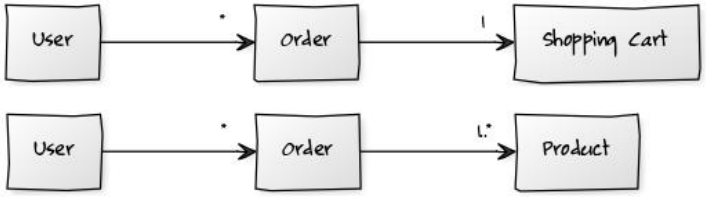


Figure 2: Representation of the `Order Patterns`

In definition the `Category Pattern` depicted in Figure 3 can be described as follows: a `Category` has at least one `Product`. A `Product` has one or at least one `Category`.

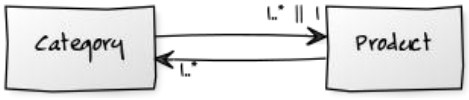


Figure 3: Representation of the `Category Pattern`

The `Shopping Cart` standard class has two different patterns, depicted in Figure 4. In definition the first `Shopping Cart Pattern` can be described as follows: a `User` has

one `Shopping Cart`, which has many `Products`. In definition the second `Shopping Cart Pattern` can be described as follows: a user has many shopping carts, which have at least one product.

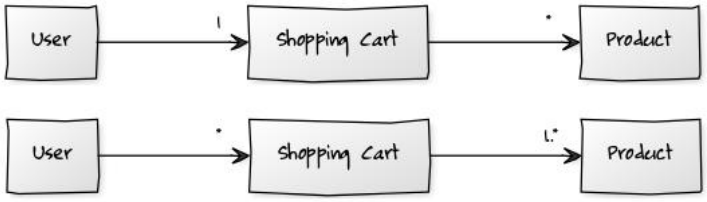


Figure 4: Representation of the `Shopping Cart Patterns`

In definition the `Comment Pattern` depicted in Figure 5 can be described as follows: a `Comment` has one `User`; a `Product` and a `User` have many `Comments`.

Figure 5: Representation of the `Comment Pattern`

In definition the `User Pattern` depicted in Figure 6 can be described as follows: a `User` has many **Orders**; a **Comment** has one `User`, and a `User` has many `Comments`; a `User` as one or at least one `Address`; a `User` has one or many `Shopping Carts`.

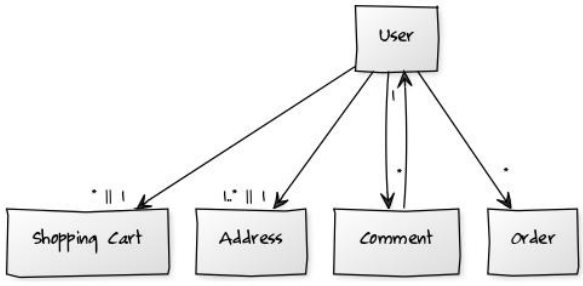


Figure 6: Representation of the `User Pattern`

In definition the `Address Pattern` depicted in Figure 7 can be described as follows: an `Order` has one `Address`; a `User` has one or has at least one `Address`.

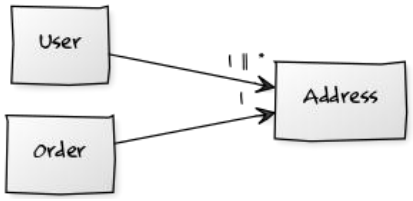
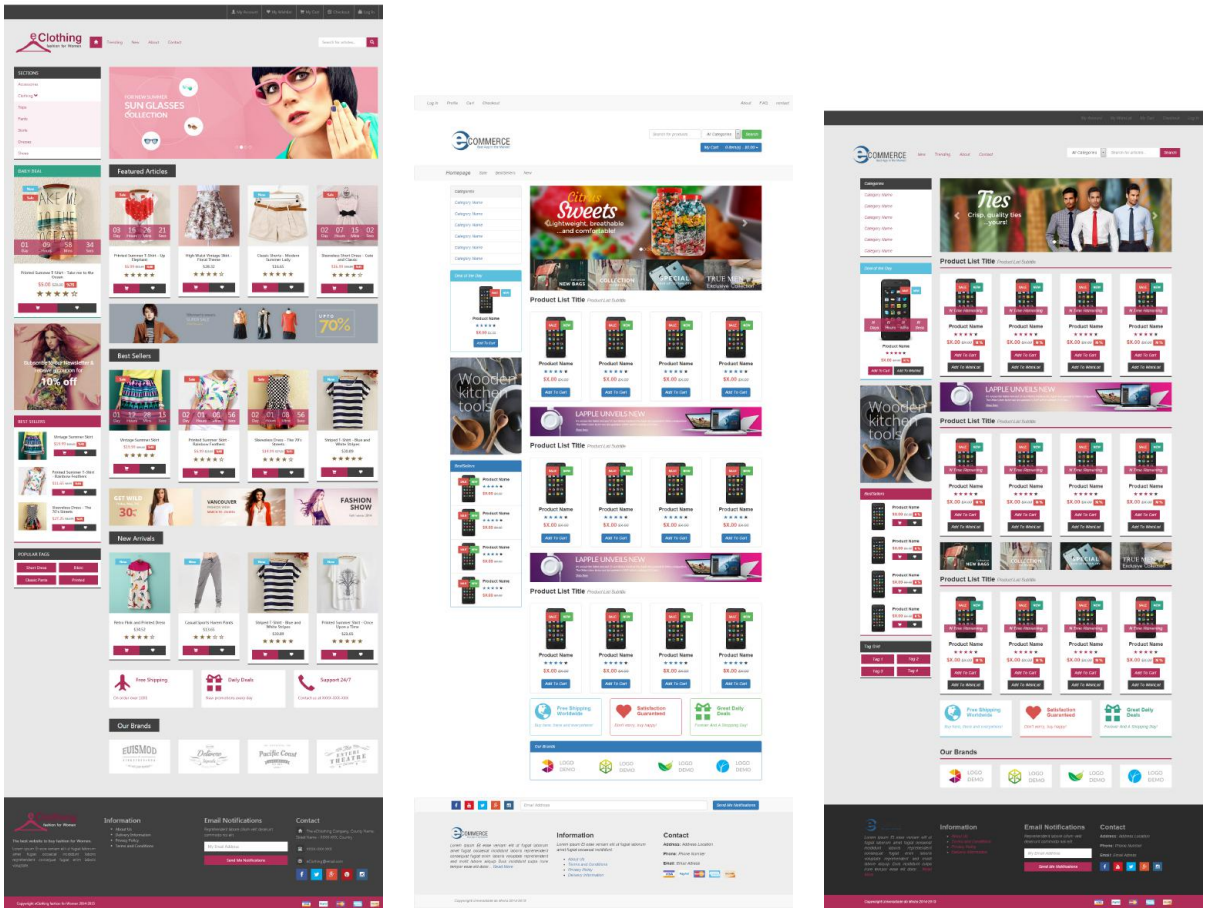


Figure 7: Representation of the `Address Pattern`

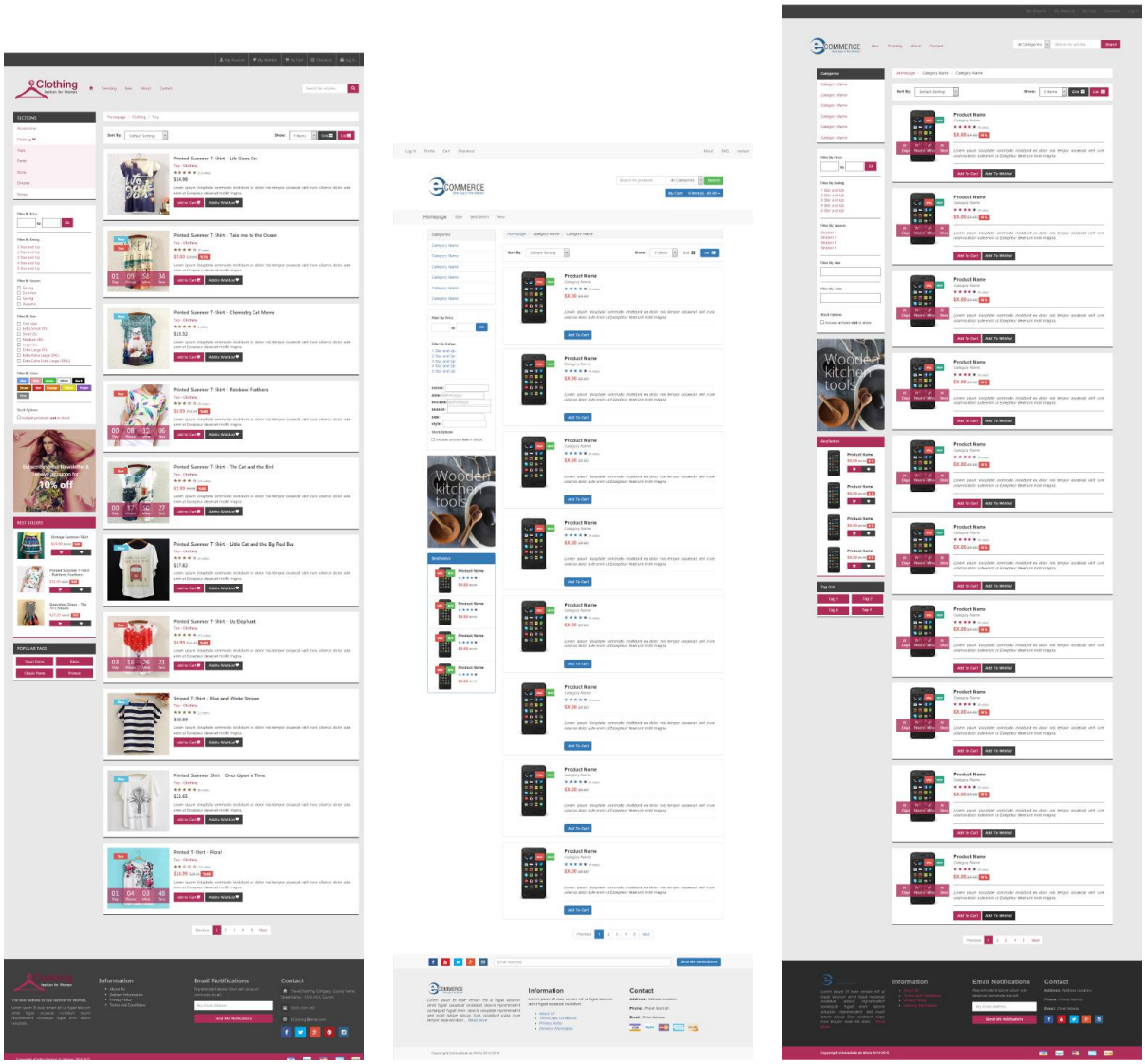
A.3 User Interfaces Generated for the Case Study



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 8: User interfaces generated for the case study : `Homepage` View

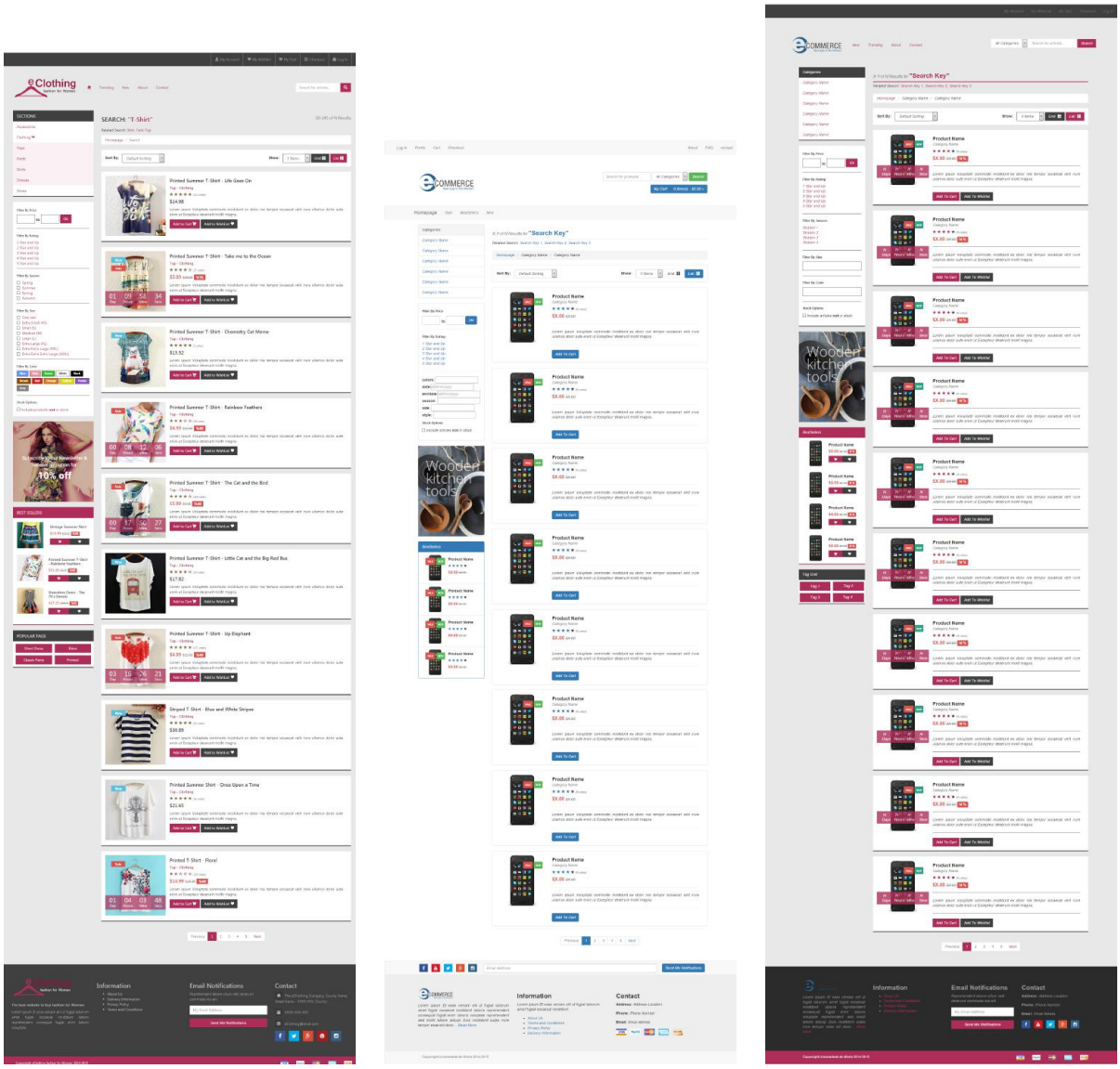
Figure 8 displays the `Homepage` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 9: User interfaces generated for the case study : `Category` View

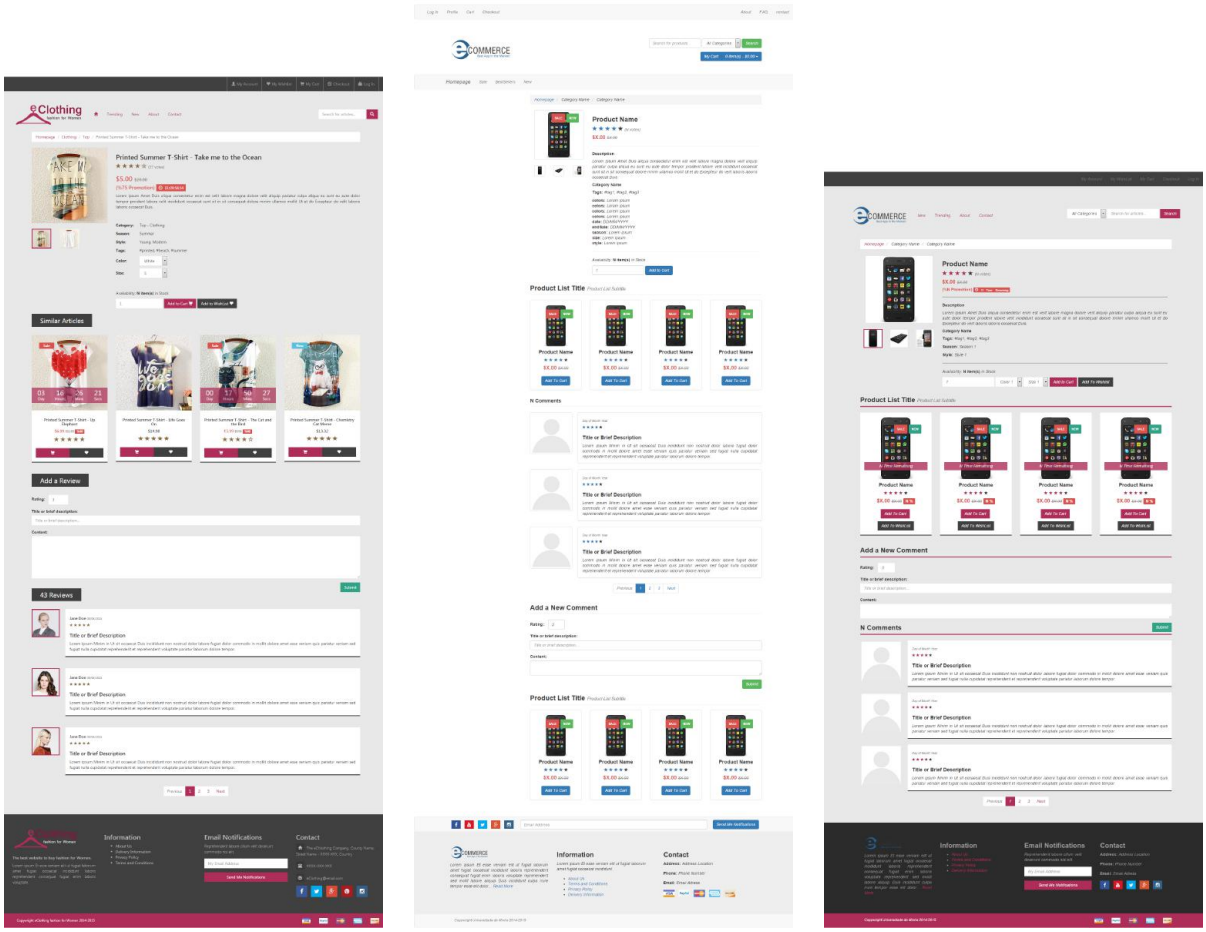
Figure 9 displays the `Category` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 10: User interfaces generated for the case study : `Search` View

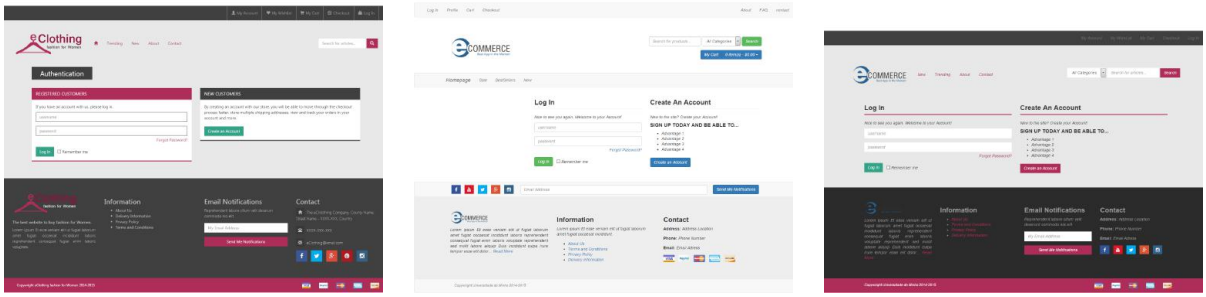
Figure 10 displays the `Search` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 11: User interfaces generated for the case study : `Product` View

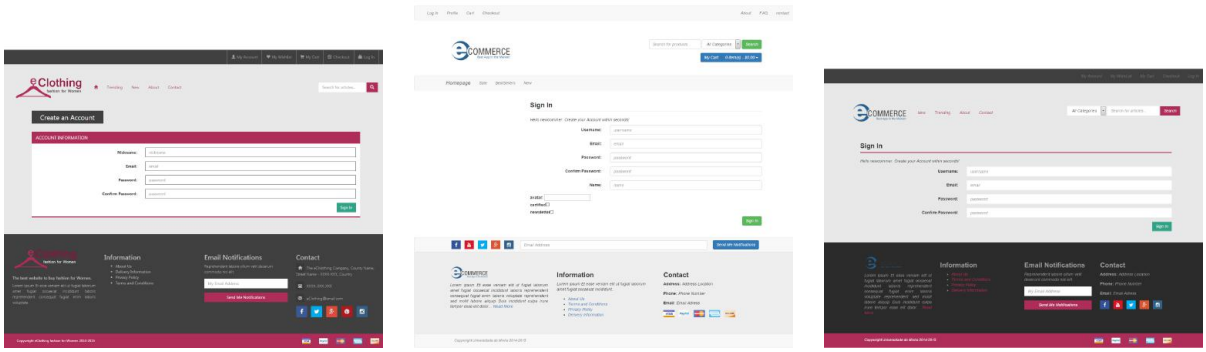
Figure 11 displays the `Product` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 12: User interfaces generated for the case study : `Log In` View

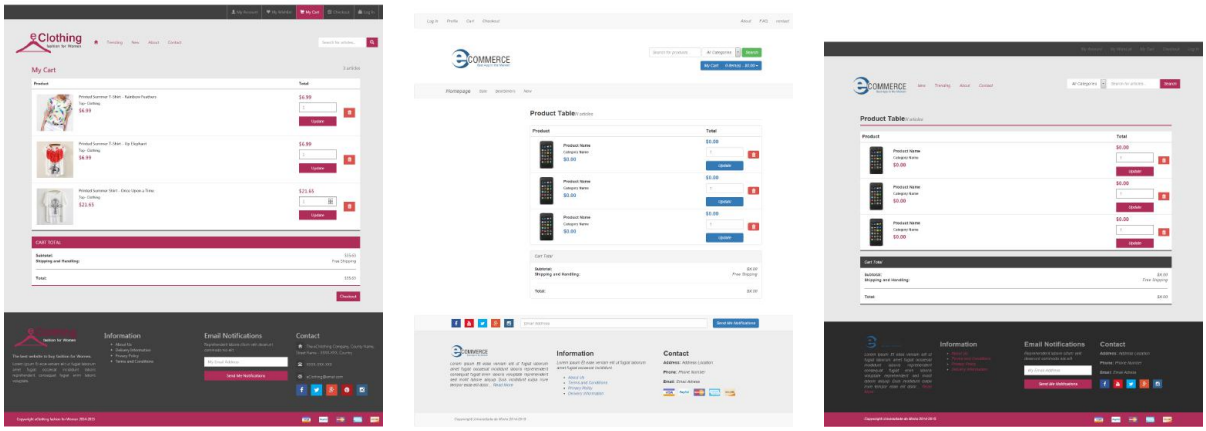
Figure 12 displays the `Log In` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 13: User interfaces generated for the case study : `Sign In` View

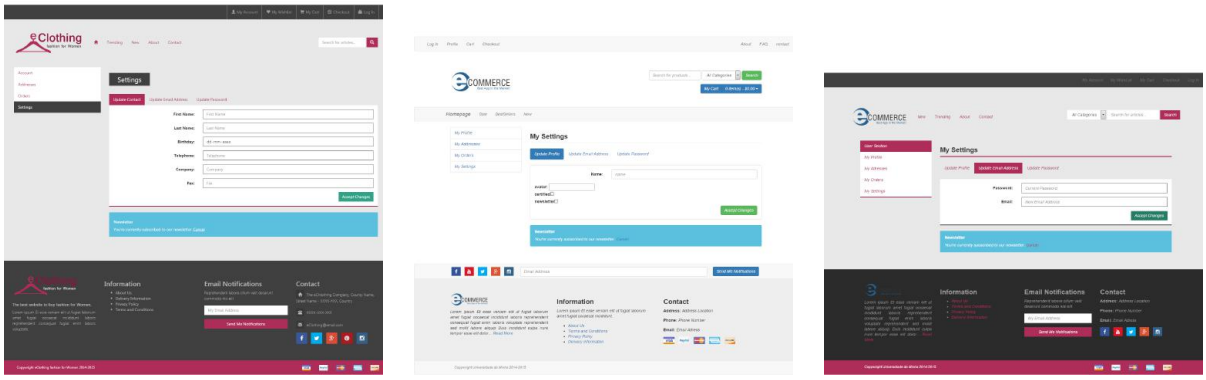
Figure 13 displays the `Sign In` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 14: User interfaces generated for the case study : `My Cart` View

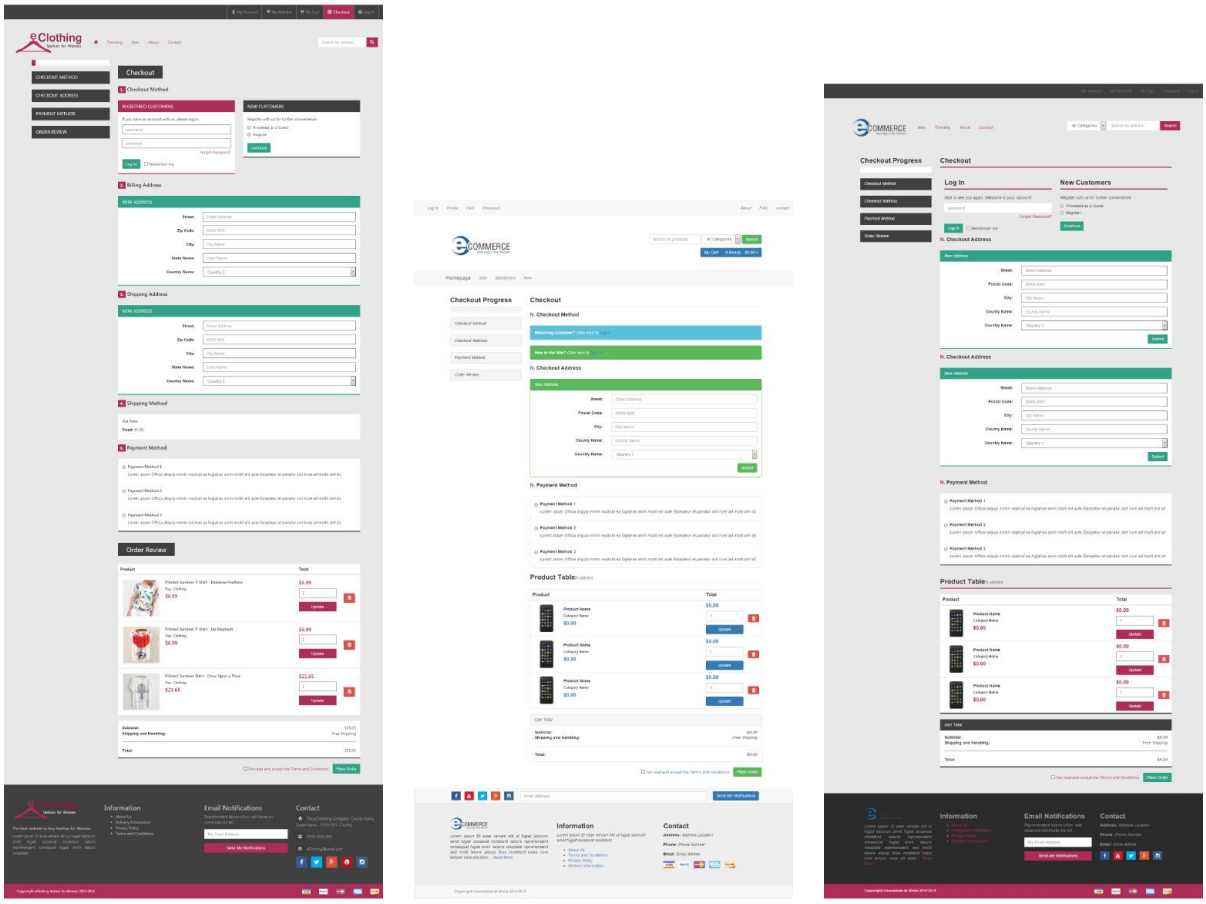
Figure 14 displays the `My Cart` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 15: User interfaces generated for the case study : `My Settings` View

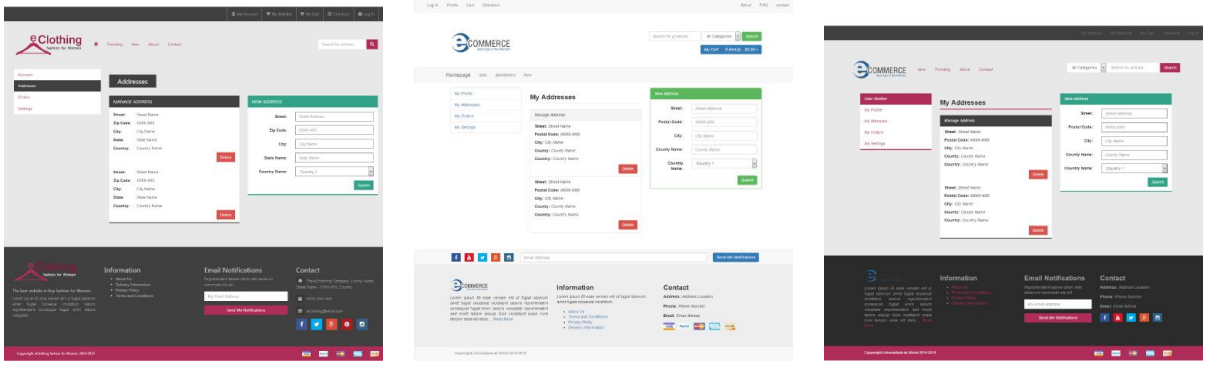
Figure 15 displays the `My Settings` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 16: User interfaces generated for the case study : `Checkout` View

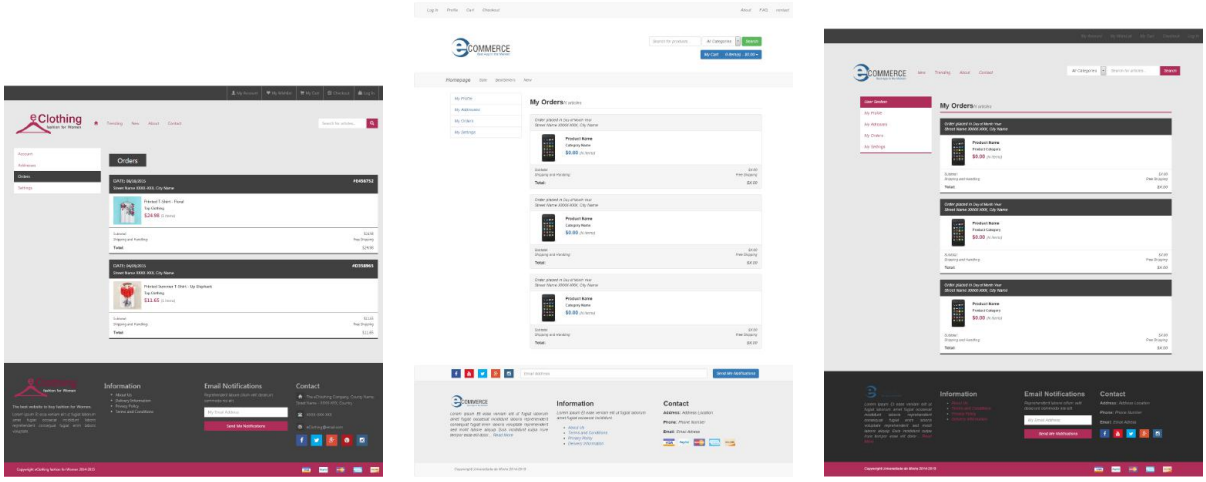
Figure 16 displays the `Checkout` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 17: User interfaces generated for the case study : `My Addresses` View

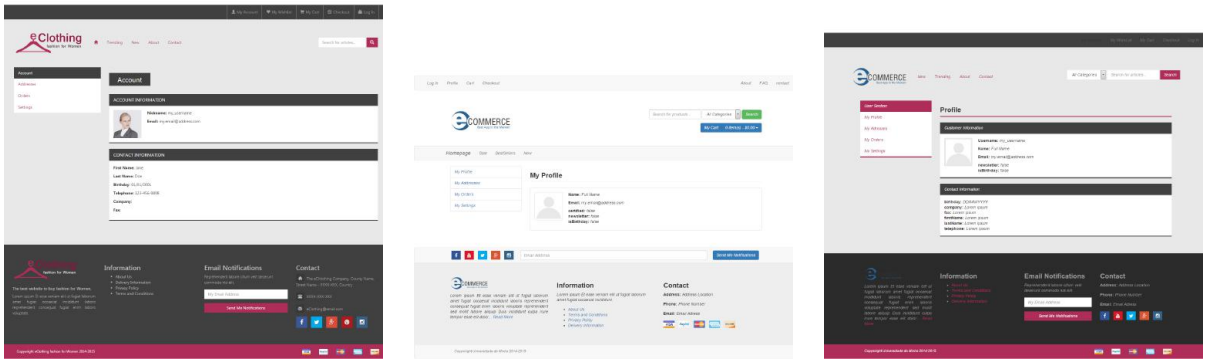
Figure 17 displays the `My Addresses` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 18: User interfaces generated for the case study : `My Orders` View

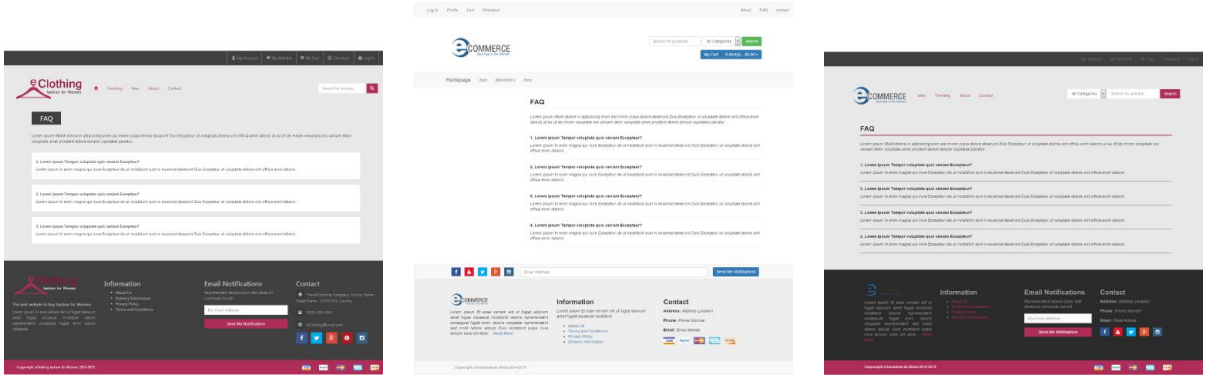
Figure 18 displays the `My Orders` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 19: User interfaces generated for the case study : `My Profile` View

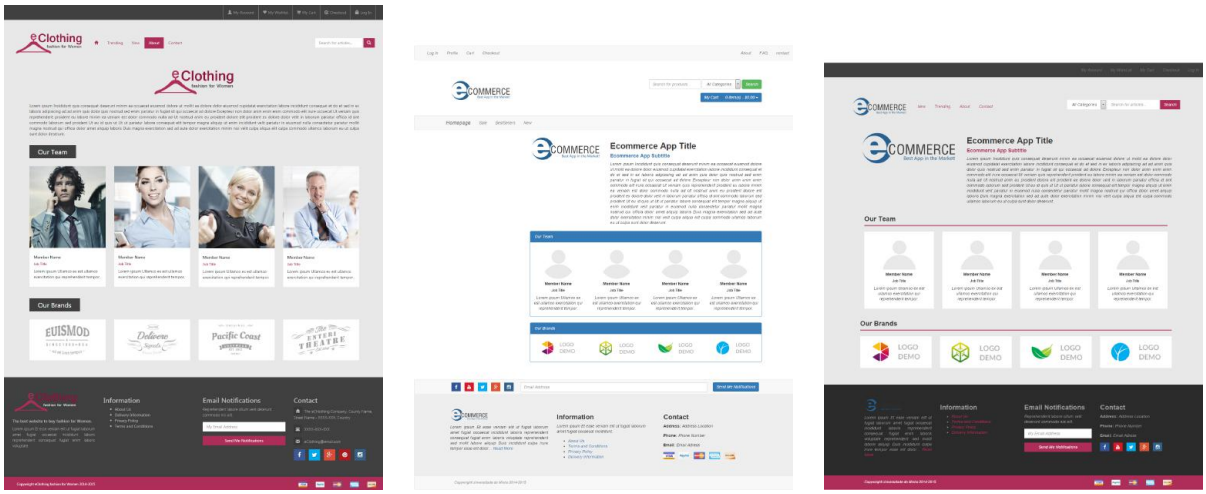
Figure 19 displays the `My Profile` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 20: User interfaces generated for the case study : `FAQ` View

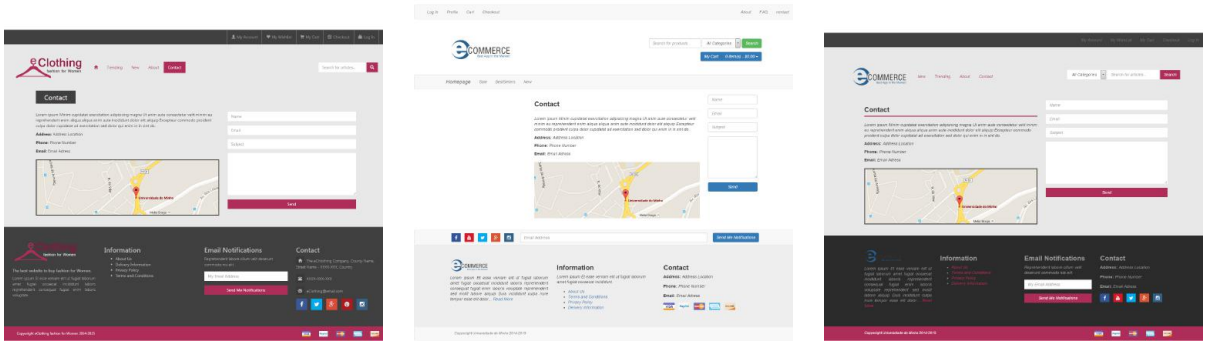
Figure 20 displays the `FAQ` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 21: User interfaces generated for the case study : `About` View

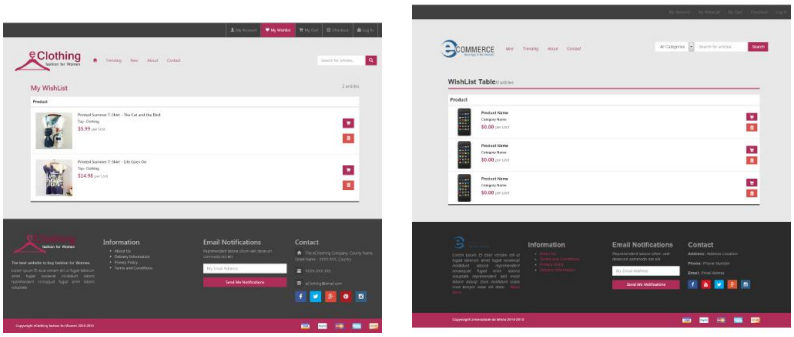
Figure 21 displays the `About` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”fully automated” (c) ”partially automated”

Figure 22: User interfaces generated for the case study : `Contact` View

Figure 22 displays the `Contact` view for, respectively the ”manual”, ”fully automated” and ”partially automated” generated user interfaces for the case study.



(a) ”manual” (b) ”partially automated”

Figure 23: User interfaces generated for the case study : `My Wishlist` View

Figure 23 displays the `My Wishlist` view for, respectively the ”manual” and ”partially automated” generated user interfaces for the case study.

Bibliography

Silvia Berti, Francesco Correani, Giulio Mori, Fabio Paternò, and Carmen Santoro. Teresa: A transformation-based environment for designing and developing multi-device interfaces. In *CHI '04 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '04, pages 793–794, New York, NY, USA, 2004. ACM. ISBN 1-58113-703-6. doi: 10.1145/985921.985939. URL http://doi.acm.org/10.1145/985921.985939.

D. G. Bobrow, S. Mittal, and M. J. Stefik. Expert systems: Perils and promise. *Commun. ACM*, 29(9):880–894, September 1986. ISSN 0001-0782. doi: 10.1145/6592.6597. URL http://doi.acm.org/10.1145/6592.6597.

Gaëlle Calvary, Joëlle Coutaz, David Thevenin, Quentin Limbourg, Laurent Bouillon, and Jean Vanderdonckt. A unifying reference framework for multi-target user interfaces. *INTERACTING WITH COMPUTERS*, 15:289–308, 2003.

T. Cerny, V. Chalupa, and M.J. Donahoo. Impact of user interface generation on maintenance. In *Computer Science and Automation Engineering (CSAE), 2012 IEEE International Conference on*, volume 2, pages 621–625, May 2012. doi: 10.1109/CSAE.2012.6272847.

Rui Couto, Antonio Nestor Ribeiro, and José Creissac Campos. Mapit: A model based pattern recovery tool. In *Model-Based Methodologies for Pervasive and Embedded Software, 8th International Workshop, MOMPES 2012, Essen, Germany, September 4, 2012. Revised Papers*, pages 19–37, 2012. doi: 10.1007/978-3-642-38209-3_2. URL http://dx.doi.org/10.1007/978-3-642-38209-3_2.

Paulo Pinheiro Da Silva. User interface declarative models and development environments: A survey. In *Proceedings of the 7th International Conference on Design, Specification, and Verification of Interactive Systems*, DSV-IS'00, pages 207–226, Berlin, Heidelberg, 2001. Springer-Verlag. ISBN 3-540-41663-3. URL http://dl.acm.org/citation.cfm?id=1756227.1756245.

Alan M. Davis. Operational prototyping: A new development approach. *IEEE Softw.*, 9 (5):70–78, September 1992. ISSN 0740-7459. doi: 10.1109/52.156899. URL http://dx.doi.org/10.1109/52.156899.

Dennis J. M. J. de Baar, James D. Foley, and Kevin E. Mullet. Coupling application design and user interface design. In *Proceedings of the SIGCHI Conference on Human Fac-*

tors in Computing Systems, CHI '92, pages 259–266, New York, NY, USA, 1992. ACM. ISBN 0-89791-513-5. doi: 10.1145/142750.142806. URL http://doi.acm.org/10.1145/142750.142806.

Clint Eccher. *Professional Web Design: Techniques and Templates (CSS & XHTML)*. Charles River Media, Inc., Rockland, MA, USA, 3rd edition, 2008. ISBN 1584505672, 9781584505679.

The Eclipse Foundation. Eclipse, 2001. URL https://eclipse.org/.

The Eclipse Foundation. Ecore tools, 2014. URL http://www.eclipse.org/ecoretools/.

ZURB Foundation. Foundation, 1998. URL http://foundation.zurb.com.

Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003. ISBN 0321193687.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. ISBN 0-201-63361-2.

B. Hailpern and P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Syst. J.*, 45(3):451–461, July 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0451. URL http://dx.doi.org/10.1147/sj.453.0451.

Natalie Weizenbaum Hampton Catlin and Chris Eppstein. Sass, 2006. URL http://sass-lang.com.

itemis AG. Yakindu statecharts tools, 2014. URL http://statecharts.org.

Darius A. Monsef IV. Colourlovers, 2004. URL http://www.colourlovers.com/.

Christian Janssen, Anette Weisbecker, and Jürgen Ziegler. Generating user interfaces from data models and dialogue net specifications. In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, INTERCHI '93, pages 418–423, Amsterdam, The Netherlands, The Netherlands, 1993. IOS Press. ISBN 90-5199-133-9. URL http://dl.acm.org/citation.cfm?id=164632.164964.

Jonathan Katz, Martin Capeletto, and Hernan Berroja Albiz. Layoutit, 2015. URL http://www.layoutit.com.

Richard Kennard and Robert Steele. Application of software mining to automatic user interface generation. In *New Trends in Software Methodologies, Tools and Techniques - Proceedings of the Seventh SoMeT 2008, October 15-17, 2008, Sharjah, United Arab Emirates*, pages 244–254, 2008. doi: 10.3233/978-1-58603-916-5-244. URL http://dx.doi.org/10.3233/978-1-58603-916-5-244.

P. Kidwell. The mythical man-month: Essays on software engineering. *IEEE Ann. Hist. Comput.*, 18(4):71–, October 1996. ISSN 1058-6180. doi: 10.1109/MAHC.1996.539925. URL http://dx.doi.org/10.1109/MAHC.1996.539925.

James R. Lewis. Psychometric evaluation of the pssuq using data from five years of usability studies. *International Journal of Human-Computer Interaction*, pages 463–488, 2002.

Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. Usixml: a language supporting multi-path development of user interfaces. pages 11–13. Springer-Verlag, 2004.

Macromedia. Adobe dreamweaver, 2012. URL http://www.adobe.com/.

José Creissac Campos Marina Machado, Rui Couto. Modus: uma metodologia de prototipagem de interfaces baseada em modelos. Proceedings of the 7th National Informatics Symposium, September 2015.

Patrick Mcneil. *The Web Designer's Idea Book: The Ultimate Guide To Themes, Trends & Styles In Website Design*. How-To Primers, 2008. ISBN 1600610641, 9781600610646.

Patrick McNeil. *The Web Designer's Idea Book, Vol. 2: More of the Best Themes, Trends and Styles in Website Design*. How to Books, United Kingdom, 1st edition, 2010. ISBN 160061972X, 9781600619724.

Gerrit Meixner and Gaelle Calvary. Introduction to model-based user interfaces. W3C note, W3C, January 2014. http://www.w3.org/TR/2014/NOTE-mbui-intro-20140107/.

Gerrit Meixner, Fabio Paternò, and Jean Vanderdonckt. Past, present, and future of model-based user interface development. *i-com*, 10(3):2–11, 2011. doi: 10.1524/icom.2011.0026. URL http://dx.doi.org/10.1524/icom.2011.0026.

B. Meyer. On formalism in specifications. *IEEE Softw.*, 2(1):6–26, January 1985. ISSN 0740-7459. doi: 10.1109/MS.1985.229776. URL http://dx.doi.org/10.1109/MS.1985.229776.

Sanjay Mittal, Clive L. Dym, and Mahesh Morjaria. Pride: An expert system for the design of paper handling systems. *Computer*, 19(7):102–114, July 1986. ISSN 0018-9162. doi: 10.1109/MC.1986.1663284. URL http://dx.doi.org/10.1109/MC.1986.1663284.

Parastoo Mohagheghi and Vegard Dehlen. Where is the proof? - a review of experiences from applying mde in industry. In *Proceedings of the 4th European Conference on Model Driven Architecture: Foundations and Applications*, ECMDA-FA '08, pages 432–443, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-69095-5. doi: 10.1007/978-3-540-69100-6_31. URL http://dx.doi.org/10.1007/978-3-540-69100-6_31.

Pedro J. Molina. A review to model-based user interface development technology. In *MBUI 2004, Making model-based user interface design practical: usable and open methods and tools, Proceedings of the First International Workshop on Making model-based user interface design practical: usable and open methods and tools, Funchal, Madeira, Portugal, January 13, 2004*, 2004. URL http://SunSITE.Informatik.RWTH-Aachen.de/Publications/CEUR-WS/Vol-103/molina-moreno.pdf.

Brad Myers. Challenges of hci design and implementation. *interactions*, 1(1):73–83, January 1994a. ISSN 1072-5520. doi: 10.1145/174800.174808. URL http://doi.acm.org/10.1145/174800.174808.

Brad A. Myers. State of the art in user interface software tools. Technical report, Pittsburgh, PA, USA, 1992.

Brad A. Myers. User interface software tools. Technical report, Pittsburgh, PA, USA, 1994b.

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.

Brad A. Myers, Scott E. Hudson, and Randy F. Pausch. Past, present, and future of user interface software tools. *ACM Trans. Comput.-Hum. Interact.*, 7(1):3–28, 2000. doi: 10.1145/344949.344959. URL http://doi.acm.org/10.1145/344949.344959.

Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993. ISBN 0125184050.

Oracle. *Lesson: Using the NetBeans GUI Builder*. 2011. URL http://download.oracle.com/javase/tutorial/javabeans/nb/.

L. Osborne, J. Brummond, M. Zarean R. Hart, and S. Conger. Clarus: Concept of operations. Technical report, USA, 2005.

Mark Otto. Bootstrap, 2011. URL http://getbootstrap.com.

Fabio Paterno. *Model-Based Design and Evaluation of Interactive Applications*. Springer-Verlag, London, UK, UK, 1st edition, 1999. ISBN 1852331550.

Fabio Paterno, Carmen Santoro, and Lucio Davide Spano. L.d.: Maria: A universal, declarative, multiple abstraction-level language for service-oriented applications in ubiquitous environments. *ACM Trans. Comput.-Hum. Interact*, 2009.

Angel R. Puerta, Henrik Eriksson, John H. Gennari, and Mark A. Musen. Beyond data models for automated user interface generation. In *Proceedings of the Conference on People and Computers IX*, HCI '94, pages 353–366, New York, NY, USA, 1994. Cambridge University Press. ISBN 0-521-48557-6. URL http://dl.acm.org/citation.cfm?id=211382.211411.

Jorg Rech and Christian Bunse. *Model-Driven Software Development: Integrating Quality Assurance*. Information Science Reference - Imprint of: IGI Publishing, Hershey, PA, 2008. ISBN 160566006X.

IT Resources. Outsystems, 2011. URL http://www.outsystems.com/itresources/.

D. S. H. ROSENTHAL. A simple x11 client program, or, how hard can it really be to write "hello world". pages 229—233, 1987.

Ana Isabel Sampaio and José Creissac Campos. Towards a framework for adaptive web applications. In *HCI International 2014 - Posters' Extended Abstracts - International*

Conference, HCI International 2014, Heraklion, Crete, Greece, June 22-27, 2014. Proceedings, Part I, pages 240–245, 2014. doi: 10.1007/978-3-319-07857-1_43. URL http://dx.doi.org/10.1007/978-3-319-07857-1_43.

Egbert Schlungbaum. Model-based user interface software tools current state of declarative models. Technical report, GRAPHICS, VISUALIZATION AND USABILITY CENTRE, GEORGIA INSTITUTE OF TECHNOLOGY, GVU TECH REPORT, 1996.

Egbert Schlungbaum and Thomas Elwert. Automatic user interface generation from declarative models. In *Computer-Aided Design of User Interfaces I, Proceedings of the Second International Workshop on Computer-Aided Design of User Interfaces, June 5-7, 1996, Namur, Belgium*, pages 3–18, 1996.

Integranova Software Solutions. Integranova, 2005. URL http://www.integranova.com/.

Herbert Stachowiak. *Allgemeine Modelltheorie*. Springer, Wien[u.a.], 1973.

Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006. ISBN 0470025700.

David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

Pedro Szekely, Ping Luo, and Robert Neches. Beyond interface builders: Model-based interface tools. In *Proceedings of the INTERCHI '93 Conference on Human Factors in Computing Systems*, INTERCHI '93, pages 383–390, Amsterdam, The Netherlands, The Netherlands, 1993. IOS Press. ISBN 90-5199-133-9. URL http://dl.acm.org/citation.cfm?id=164632.164954.

Rails Core Team. Ruby on rails, 2005. URL http://rubyonrails.org/core/.

Allen B. Tucker, editor. *The Computer Science and Engineering Handbook*. CRC Press, 1997. ISBN 0-8493-2909-4.

Jean Vanderdonckt. A mda-compliant environment for developing user interfaces of information systems. In *Proceedings of the 17th International Conference on Advanced Information Systems Engineering*, CAiSE'05, pages 16–31, Berlin, Heidelberg, 2005. Springer-

Verlag. ISBN 3-540-26095-1, 978-3-540-26095-0. doi: 10.1007/11431855_2. URL http://dx.doi.org/10.1007/11431855_2.

Jean M. Vanderdonckt and François Bodart. Encapsulating knowledge for intelligent automatic interaction objects selection. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, CHI '93, pages 424–429, New York, NY, USA, 1993. ACM. ISBN 0-89791-575-5. doi: 10.1145/169059.169340. URL http://doi.acm.org/10.1145/169059.169340.

YOOtheme. Uikit, 2007. URL http://getuikit.com.

NB: place here information about funding, FCT project, etc in which the work is framed. Leave empty otherwise.