RADBOUD UNIVERSITY NIJMEGEN

Creating Adaptable and Adaptive User Interface Implementations in Model Driven Developed Software

Master Thesis

Information Sciences

Radboud University

Comeniuslaan 4

6525 HP Nijmegen

The Netherlands

Faculty of Science

Huygens building

Heyendaalseweg 135

6525 AJ Nijmegen

The Netherlands

Author
Nikolaos Makris

Internal Supervisor RU
Prof.dr. M.C.J.D. van Eekelen

External Supervisor CGI
Edwin Hendriks MSc

Abstract

This Master thesis research attempts to explore the ways of implementing dynamic user interfaces in the context of software generated by MDD (Model Driven Development), using SPADE as the representative of this approach. The ways of achieving the dynamic part are split between adaptivity and adaptability, two notions which are explained, along with the implementation challenges that they pose. Based on these two concepts, the implementation efforts are split between two paths: adaptability is approached via creation of a custom DSL language which is an extension of SMART (used by SPADE), whereas adaptability is realized via intelliGUI, an AngularJS based front-end solution which is based on use cases execution. The two approaches are analyzed, presenting the challenges and the drawbacks that were encountered. Furthermore, results are evaluated with regards to the ways they achieve the target of creating dynamic user interfaces, facilitating users and their task execution. Finally, a discussion about the research process is presented, along with propositions for future work on the subject.

Contents

Acknowledgments

In the course of completing this research, from inception to realization, several people have assisted and supported this goal. First of all, I would like to thank my university supervisor Marko van Eekelen, for all the guidance, assistance and efficiency in unraveling complex situations. In addition, I would like to thank my CGI internal supervisor, Edwin Hendriks, who was the initiator of this fascinating project and a great guide and helper throughout the whole process. I cannot stress enough how grateful am I for the internship opportunity that I was given.

Special thanks go to Joeri Arendsen, my fellow colleague, for the numerous hours spent together trying to cooperate and help each other in reaching our research goals; it was a fun ride! Furthermore, CGI colleague Marcus Klimstra deserves acknowledgment for all the high quality help and information provision regarding the implementation part and the technical details. Last, but not least, all my family and friends deserve a token of gratitude; everyone gave a small hand and supported me from the beginning towards the very end of this journey.

Disclaimer

This thesis has been a result of a research internship done at software company CGI in Arnhem. It is a part of a generic exploratory effort in Dynamic User Interfaces for the SPADE tool, which was led by Edwin Hendriks and two MSc Students; me and Joeri Arendsen, who is the creator of the other part of the research [20]. Since the whole process of the research was collaboration between us, there is expected to be an overlap between the two theses in some theoretical topics. However, there is clarity in the differentiation of scope, results, and the part that they are linked. Joeri's thesis is mainly focused on design solutions, whereas the current thesis is focused on the implementation side and the challenges posed. As an evidence of the above, the two researches are having a specific linkage which complements each other, and that is the use cases of the UI adaptation; these are defined on the design level by Joeri's research, and then taken and implemented on the current research[1].

The source code that is presented in parts of this thesis is available at
https://github.com/nmakrisru/intelligui

[1] See Chapter 3: Implementing solutions

1 Introduction

In the introduction chapter, the theoretical notions and foundations of this research will be explained. This research revolves around two generic domains of Computer Science:

1. Model Driven Development
2. User Interfaces Adaptation (Adaptability and Adaptivity)

The approach of Model Driven Development will be introduced along with an implementation of such a framework, called SPADE. Then, the basic principles of User Interface adaptation will be presented. Finally, after pairing the two notions and explaining the UI adaptation part in the MDD software context, there will be a first explanation of the notions of adaptivity and adaptability and the differences between them, resulting in the research domain and questions of this thesis.

1.1 The model – driven software development approach

Within the rich and mesmerizing world of computer science and software generation, SPADE introduces an approach which attempts to simplify the software development process by

a) Having an end-result orientation and

b) Following the principles of model–driven development.

But what is model-driven development and what are the principles of it?

Model-driven development (MDD) is the term used to describe software development approaches where abstract models of software systems are being created, which are then transformed in a systematic way to actual, concrete implementations [9]. MDD is focused on isolating – in the maximum possible level – the definition and design of software on a higher level from the low-level domain of implementation (i.e. programming language, frameworks, and platforms). Afterwards, the automatic transformation of these abstract high-level models into running software systems is being realized by predefined and systematic modules [10].

It is a quite common phenomenon that, while amidst of a development process, any design/architectural/implementation decisions taken are arbitrarily perceived, applied and associated with the process per se. This means that, many times, all these decisions (which can be in many forms – from a design document or a manual, to a simple comment in the code) are not **explicitly** stated, consolidated and shared, introducing the "reinventing the wheel" problem in the software development world.

What MDD does is that, as an approach, it intends to raise the level of abstraction on which developers evolve and create software, with the purpose of introducing simplicity and formalization – via common terminology and artifacts used - in the different tasks that define on how a software development process is being undertaken; structures and common vocabularies are being imposed in order to tackle the aforementioned problems [11].

Simply put, MDD is about:

a. discretizing the software development process and isolate the complexities that arise in the way
b. solidifying implementation variables (definition/design decisions, architectures, testing data/scripts, “best practices” in code) into reusable components
c. establishing common terminology for knowledge sharing and reusability

Separating the model from the low-level implementation processes and frameworks, gives the possibility to a model to independently describe the solution to the problem that it is tackling in a high level and coherent way. With this method, the model becomes easy to understand, not only for the developer, but also for the end user. Furthermore, explicit definition of everything that revolves around the actual implementation leads to knowledge sharing, reusability and scalability; things that can prove hugely beneficial when the size of projects increase exponentially (something that is quite often nowadays especially in large companies who undertake large-scale projects). Finally, it is safe to conclude that feedback loops and testing work much better in this context, and they also seem to align smoothly with the scrum framework and agile way of working which tend to dominate the IT world.

1.2 SPADE –MDD platform

It is time now to present an actual application which carries out the MDD approach; the idea behind SPADE will be shown, the philosophy of this approach towards software development and there will also be an attempt to point out how exactly this approach deals with the User Interfaces of the resulting products.

SPADE stands for Smart Process Acceleration Development Environment, which contains in a nutshell the main notions that describe it; smart and quick development of software. It consists of an application framework that takes as an input business requirements which are result-oriented, defined directly by the client. The system then has the ability to automatically deduce the required business processes, automatically implement a complete Java based software system solution, compile and test it, finally deliver a working, platform independent piece of software ready for the end user.

Using SPADE to create software is arguably an innovative and unprecedented approach, since this end result oriented philosophy is the core of this development process. The development cycle (high-level) is:

1. The client defines desired end result – product
2. With the assistance of a SPADE expert, these are translated into SMART requirements, a high-level pseydo-like language (see 1.2.1)
3. These requirements are interpreted as high level models which, in turn, are translated into java classes and methods via the SPADE generator
4. Classes are assembled into executable jar files (platform independent), testing is done within this process
5. A fully functional software product (with generic and predefined UI) is generated

1.2.1 SMART requirements

It was stated that SPADE, as a system, requires an end-result oriented input; this is a set of rules, formed in a metasyntax (pseudo-language) based language called SMART notation[2]. The business requirements that the user/customer defines for the desired system are written in exactly this type of notation. As stated before while defining the system, these requirements have to be interpreted in the form of the desired **end result** - meaning that the requirements should specify **what** the end result is and not **how** it is achieved. To get the end result in a SMART way (and therefore give the desired input for the SPADE engine) you need to answer the following questions[3]:

1. **What** is the (part of the) end result?
2. **When** will the (part of the) end result be achieved?
3. **Where** does the information needed come from?

You can see an example of the SMART notation below, regarding a definition of the requirement that when a customer has ordered products from a product list, an invoice has to be automatically created (see next page):

```
Process 'Order products' with subject #('Order': ORDER)

// ============== individual results ==============

The following applies:
    "Customer has ordered products"
    and
    "Customer has an invoice"  // as specified in [messages.pa]

// ============== SMART details about the individual results ==============

"Customer has ordered products" =
    One ORDER exists in ORDERS with:
        date        = currentDate()
        CUSTOMER    = CUSTOMER
        ORDER_LINES = "order lines"

"order lines" =
    Several ORDER_LINE exist in ORDER_LINES with:
        PRODUCT = input from CUSTOMER
        number  = input from CUSTOMER
```

In the above (simple) example, the result is broken down to individual and human-readable results, and then these are being analyzed in their components and respective attributes. There are two logical requirements for the underlying process, which are defined on top:

`"Customer has ordered products"` and

`"Customer has an invoice"`

Each of these two logical statements are then analyzed in a “programming” way; variables and loops are used (it can get more complicated in other examples!) in order to make the statements plausible for the SPADE generator to implement them.

[2] SMART notation language specification_MANUAL – CGI Intranet, available on demand
[3] SPADE training (powerpoint presentation) – CGI Intranet, available on demand

SMART requirements were used in the process for the research purposes of the thesis and we will come back to them in Chapter 3, when we will be using SMART notation for implementation of UI adaptation techniques.

1.2.2 SPADE functionality and criticism

When SMART requirements are set and properly defined, aligning with the wishes of the user/stakeholder, SPADE takes care of the implementation, testing and deployment of these requirements into a working product, automating many steps from the normal software development process (Figure 1):

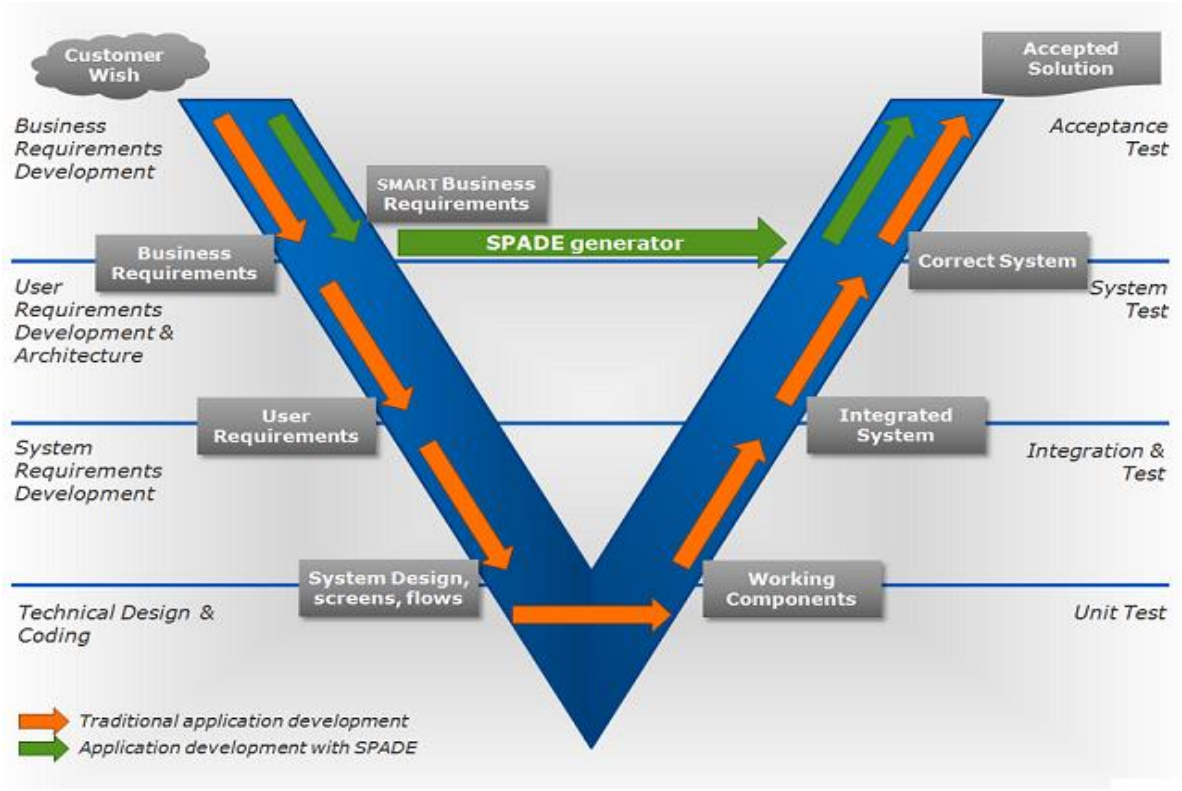

Figure 1: Regular development process vs SPADE development process (source: CGI presentation)

As long as requirements are properly and strictly defined, SPADE generator takes the lead into the development process; the requirements are translated into system specific requirements and concrete Java classes, methods, variables etc. These are resulted in modularized working components and testing processes are initiated. Then, the components are integrated and assembled into a working system, which passes through the user and acceptance tests to reach the conclusion. In essence, human factor is only involved in defining test cases, as well as in the user acceptance testing[4].

Describing how SPADE generator works does bring in front the biggest advantage of it; speed. It bypasses some of the rudimentary software development steps and automates processes of low-level coding, integration, testing and deployment. In addition, by minimizing the human intervention in the process, it is less prone to errors and a smoother workflow is being realized.

[4] SPADE training (powerpoint presentation) – CGI Internal document, available on demand

SPADE, as stated by its creators, is mostly suitable for:

- ✓ workflow oriented,
- ✓ case management and
- ✓ output generating systems (generating output such as documents, mails, messages – example in 1.2.1 where an invoice is generated).

This implicitly states one of SPADE's weaknesses, which is that it is not suitable for more complex, non linear workflow-based systems; embedded systems are also a no-go. Furthermore, the target platforms where SPADE can fit are limited to Java-based platforms (this is bound to change in the future) .Last but not least, SPADE suffers from the MDD drawbacks, with one of them being the single unified user interface which has no flexibility and adaptation to users, processes or data – and this problem is the core foundation of the research issues this thesis attempts to deal with (see Chapters 1.4 and 1.6).

Before addressing the User Interface generation and issues in MDD software, the following chapter will give an idea of the background and the basic foundations of dynamic UI development.

1.3 Towards dynamic user interfaces

1.3.1 Building dynamic user interfaces

Before discussing on the presence – or not – of dynamic UI's in the context of the MDD approach, an introduction to the world of user interface development will be given, along with challenges and evolutions into dynamic UI's.

User Interface (UI) development is an area of increasing interest, focused on accomplishing the aforementioned task. Developing UI used to be a unified process, along with the rest of the software product or the application; but then, after the growth in the size and complexity of software products and applications (especially enterprise focused), it became an independent professional and research domain. This "isolation" of the UI development makes perfect sense, since it is now a major part of the software development process; according to [1], roughly 48% of application code and 50% of the development time are dedicated to UI development and implementation.

Up until *recently*, before the UI development area drew significant attention, UI's were mostly developed in a static way; in reality, even until now it is quite common for the developer to create a single and unified UI that is addressed to all users [2]. The developer evaluates and measures the target group of a specific application and subsequently tries to create a UI that fits the specific needs and profiles of those respective users. Then, the result is a static and user-generic interface, with the same appearance and functionality in every instantiation. The main reason of this approach is (as someone would expect) the costly process of UI creation both in terms of development and maintenance, as stated before.

However, things have changed; the appearance of ubiquitous (or pervasive) computing and context aware systems [3], not only has made a whole new set of devices available, but also highly increased the range and variety of possible users that interact with it. Now, a software product that wants to

become dynamic and coherent with regards to the user has to take under consideration three factors [5]:

1. **User range**: Diversity of users, different backgrounds, uneven technology familiarity levels, personal preferences
2. **Device range**: Screen size, performance, interaction mechanisms
3. **Environment diversity**: Location based services (GPS), accelerometers, movement sensors.

Creating software and applications that can accumulate information from the aforementioned factors and tailor results and outputs to the user's desire is a challenge and an ongoing process currently in Computer Science; by isolating the scope to UIs, the area of Adaptive User Interfaces (AUI) has emerged.

Creating an AUI (which has been given different labels through the years, such as adaptive interfaces, dynamic interfaces, modeling systems, user-adaptive systems [4]) is, in principle, a complicated and extended process. The goal – or, more accurately, the challenge - is to create a UI that can adjust itself accordingly based on the user's needs and preferences, taking also into account the context of use, which is the device that runs the program, as well as the environment variables that affect the usage of it at that certain point. This whole process is – as someone would expect - far from trivial; analyzing and demonstrating all three factors and the multiple ways they can be perceived in order to achieve a dynamic UI is complex, time-consuming - as it will be demonstrated in the process – and out of scope of this thesis; consequently, and for the purposes of our research, user adaptation will be the main focus.

1.3.2 AUI basic principles and methods

The pursuit of interfaces that adapt to each user's needs in order to offer enhanced user experience has been a challenge throughout many years for software developers and computer science researchers. This research area includes methods that range from explicit requests for input from the user – giving him ability to choose from a spectrum of possible combinations the desired interface – to machine learning and data mining techniques that extract the information on-the-fly, providing the user with the desired interface based on experience.

A simple approach to adapt UI's and make them easier to use are GUI builders such as Swing builder [6]. Creating a UI on generation can provide multiple options in initiation of UI's in various platforms and devices. However, this approach lacks flexibility; consequential extended differentiations on the requirements of the user (or the interface designer) are difficult to be implemented through the building tool.

Another approach of achieving adaptation in UI is presented in [7]. In this approach (which is focused on the web applications domain), the user is not explicitly questioned about their preferences or expectations; instead, an implicit data mining method is taking place, exporting knowledge of the user's behavior from web log files which store information of the visits that were made to the website. Thus, the website can tailor its interface structure and appearance based on the individual user. An implementation was created for this approach, called AdAgent, which would create list of recommended links for the user based on his logged history.

One conceptually similar approach is the "Smart menus" feature that was first present at Windows 2000 and is mentioned in [4]. The idea is that the menu's most infrequent choices from the user will be hidden after a certain point from the default menu – they will only return if user selects these options with enough frequency so they can qualify again as frequent choices. The goal is to always swiftly provide the user with the menu choices he will most probably use (Figure 2)

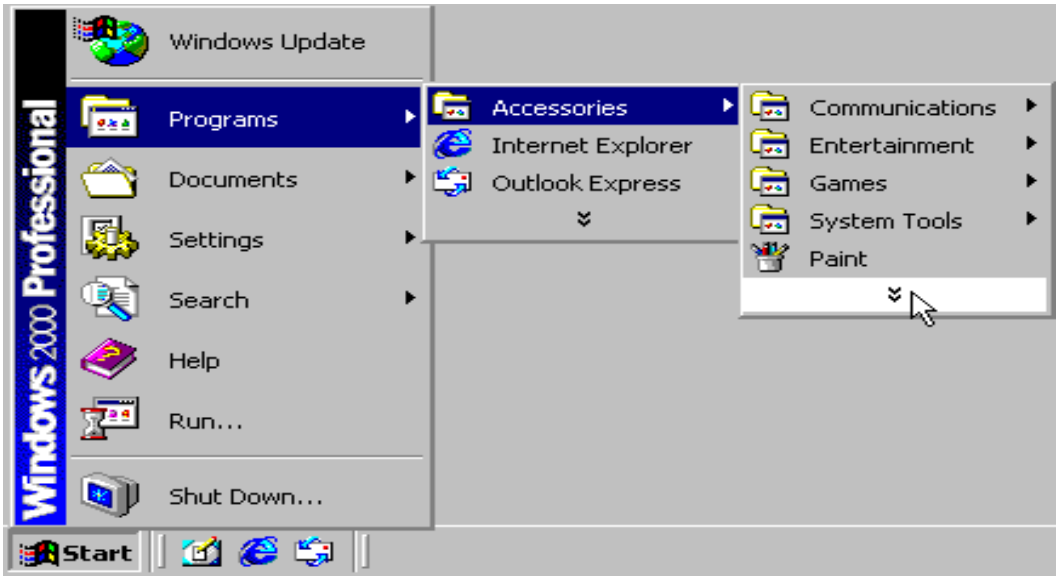

Figure 2: The smart menu that was first introduced on Windows 2000.

The MDD approach has –unsurprisingly- also touched the field of the user interface development, introducing more modularized and sophisticated ways of implementing interfaces. One is presented in [12], where the transformation process from high-level models to code that deploys an adaptive UI is extensively explained; there is also a presentation of a framework which generates ubiquitous User Interfaces in an automated way, putting focus on usability. Despite the obvious advantages that exist in a theoretical level, this approach is encountering many issues of applicability in practice; it is explained that this approach is not compatible with standard software development platforms such as Java EE.

1.4 Adaptability vs Adaptivity

In the next part, it is important to pinpoint the main segregation of ways of achieving dynamic user interfaces; adaptability and adaptivity. The analysis and explanation of the differences between them is vital, as they constitute critical concepts of this research,

The problem of adapting user interfaces is far from being trivial. Creating a mechanism that can ultimately produce a unique UI for every single user that exists out there seems like a utopia at this moment. All pieces of the puzzle must be formed and fit well with each other in order to achieve 100% user-centered adaption; from extraction of user data and log files, to adaptive mechanisms (pre and post-runtime) that process the input data in order to modify the UI in such a way that it would fit each individual user's knowledge, habits and needs. Considering the complexity of the problem, the

need of breaking it down to multiple subparts emerged; as a result, the introduction (and distinction) of terms adaptivity and adaptability was introduced.

Adaptivity and adaptability are terms used to describe the "how" an intelligent mechanism can achieve the goal of tailoring a UI to a specific user. There are two main categorizations of these terms in the bibliography, each one having a different interpretation of the term (although, there seems to be some correlation).

The first way of describing and comparing the terms is given at [8], where the Unified User Interface methodology (U2ID) is introduced. According to this paper, only a single unified user interface is created, with multiple alternative components for different user categories. In this context, **adaptability** is the multiple instantiation of this user interface with combinations of components and attributes, in order to adjust the UI to a specific group of users (Figure 3).

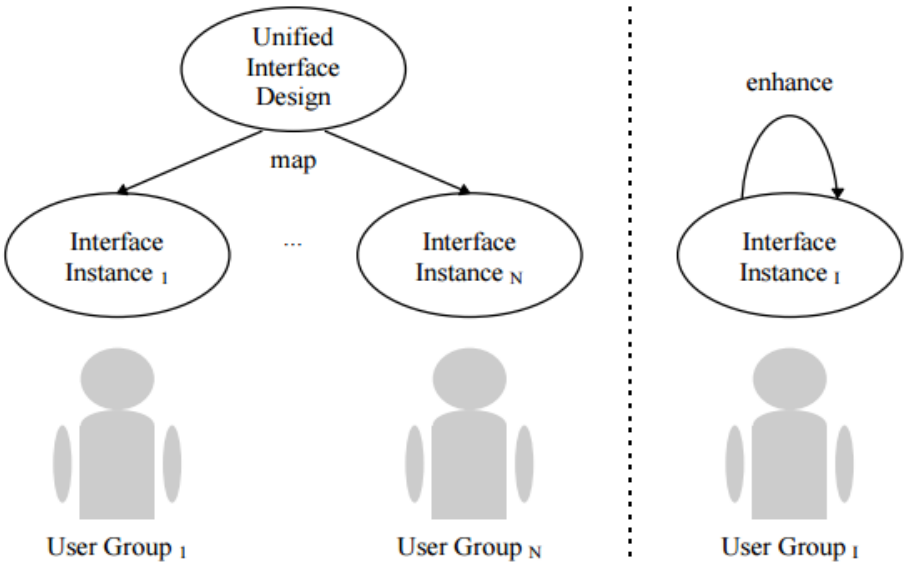

Figure 3: Adaptability and Adaptivity(Source: [8])

The left side of the above figure shows that multiple instances of user interfaces are defined (each one having its own characteristics) and then, based on predetermined rules, they are mapped to the respective user groups.

On the other hand, **adaptivity** is the constant amplification of the user interface that occurs during *runtime* (based on feedback methods), and is depicted on the right side of the above figure. There are no multiple predefined user interfaces; only one – the user interface which interacts with the actual user – which is being transformed constantly. Simply put, adaptability changes the UI at (or even prior to) instantiation time, whereas adaptivity amends the UI during runtime, while the user is interacting with it.

There is, however, a second way of describing these two ways of adapting UI's. As stated in [23], there is a distinction between adaptivity and adaptability in the way these two tactics draw user information to create the dynamic behavior; adaptivity receives this user information in an implicit way (via some form of learning or inference), whereas adaptability asks the user in an implicit and direct way to determine the appearance and layout of the UI. Therefore, in adaptability a user is actively and consciously involved in the adaptation process of the UI (for example, being asked

questions from the system about the preferred layout), while in adaptivity the system collects user information based on his activity.

Carefully examining the above different distinctions it seems that, in the end, there is a converging of notions and they can be described based on the procedure which a system will use to adapt interfaces. The first distinction simply claims that the difference is in the "when" the adaptation happens:

adaptability → pre runtime

adaptivity → runtime

In the second distinction, the difference is in the user participation:

adaptability → explicit

adaptivity → implicit

It is non-trivial to conclude that adapting the UI at runtime in most of the occasions will preferably not involve the user in this process; the user will preferably be involved prior runtime, asked by the system regarding his preferred customizations that can be applied to the UI and possibly will not be bothered at runtime with adapting the interface, focusing on his tasks.

Therefore, the following matching of notions can be derived:

Adaptability → pre runtime / explicit

Adaptivity → runtime / implicit

Overall, these concepts and their translations are quite important and serve as a guide throughout the research paths of this thesis. As it is quite evident in the bibliography [13][14], adaptivity and adaptability are the two sides of the UI adaptation coin; they are basically two system properties that "complement" each other in the process of creating dynamic user interfaces. Therefore, extensive analysis of these two types of adaptation has to be made in order to explore the implementation methods (and challenges) that surface into the model-driven software development domain.

1.5 The need for dynamic UI's in MDD software

So, after examining and presenting the principles of the MDD approach, along with dynamic user interfaces, there is an apparent question that comes to mind: when creating software under the MDD approach, how is the interface being treated? What kind (if any) methods are there in handling the interface creation? Is adaptation taken into account and incorporated in the core of the development process?

Let's take the example of the SPADE engine as an example of MDD developed software. See the image below (Figure 4):

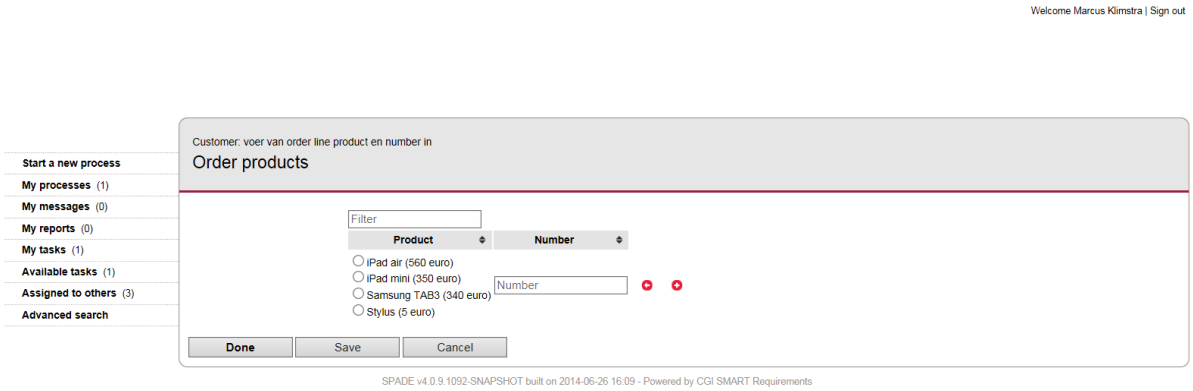

Figure 4: Process 'Order products' (SPADE generated)

This is a SPADE output (called "process" in the tool's context) which allows the user to order products based on a list of items stored on a database. In this example, the variation of items is small and the radio button list seems to fulfil the user's requirements.

What if:

a) the item list was big (e.g. 1000), how would that affect user experience? Would the user be happy with a huge radio button list that would take up his screen space and require multiple mouse scrolls in order to browse products?

b) user wants to choose multiple items at once? Radio button list is mutually exclusive on its elements.

c) the user wants to modify the board (switch columns, drag & drop components etc.)? Apart from filtering and sorting, there is no more flexibility on his side; the columns are as is.

Look at another example of a different process (Figure 5):

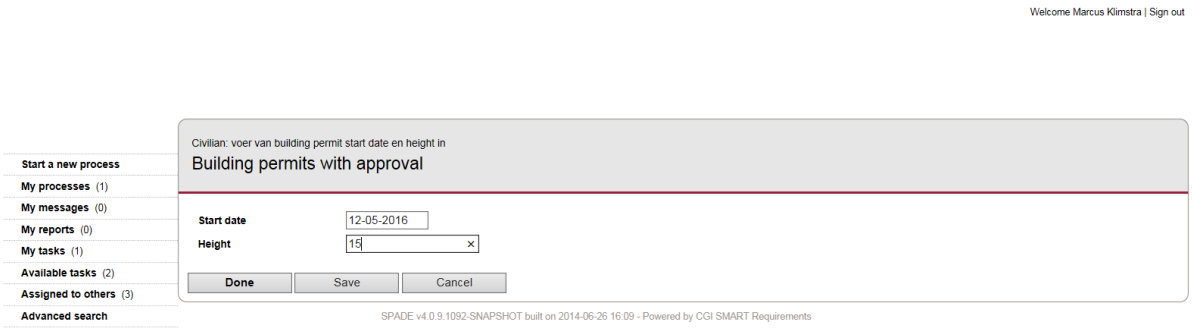

Figure 5: Process 'Build permits with approval' (SPADE generated)

This is yet another (albeit simplistic) process, in which the client simply chooses a date and the height of a building to be approved. The interface backbone and the visual platform are **identical** to the previous process; there seems to be no new/added elements. The clue here is that there exists a unified user interface approach from the SPADE generator, which is applied in the various kinds of processes that are being executed.

The above example is exactly the problem that exists generally in MDD: it generally does not incorporate UI adaptation techniques or approaches. But why this is the case?

Well, simply because MDD in general does not **care** about adaptation on user interface level! The underlying models that derive the code which compile and create the application are strictly focused in the functionality, persistence and business part of delivery. These abstract models are created in order to make software which simply works in efficiency and based on the requirements, without digging deep into user interface and providing flexibility for the user and/or the environment under which they will be executed. Therefore, the unified user interface approach (clearly depicted in the SPADE example) is prevailing in MDD.

1.6 Research goals

Based on the above background in MDD, SPADE and UI adaptation, the definition of research questions was done on the premises of clarity, solidity, and focus area distinction between me and Arendsen[20]. Generally, the distinction was done in a design/implementation basis; as the title discloses, the current research focuses on the implementation level.

More specifically, the main research question is:

-How to manipulate the GUI-components of Model-driven Development generated software in order to achieve User Interface adaptation?

This can be broken down to:

-How to manipulate GUI components towards adaptivity in Model Driven Development generated software?

-How to manipulate GUI components towards adaptability in Model Driven Development generated software?

2 Implementation challenges in Adaptivity and Adaptability

In this chapter, the actual challenges of implementing adaptivity and adaptability in MDD software will be presented and analyzed. These challenges have been identified after multiple sessions with SPADE domain experts and colleagues, and they are describing the main concepts that require specific attention when trying to produce adaptability and adaptivity in UI based on an MDD context. They have also been verified as valid in the later stage of the solution implementation which will be extensively analyzed in Chapters 3 and 4.

The challenges that arise in reaching adaptation have been classified in under three main categories:

1. User experience intervention aspects
2. Implementation Complexity
3. Interface plasticity

These three main notions describe in essence the main answers on why it is quite challenging to create dynamic UI's in the MDD world (such as SPADE, which is the case in point). A nice visualization that can make these three concepts more tangible and also present their associations is the following overlapping circles (Figure 6, idea first perceived by thesis supervisor Marko van Eekelen):

Figure 6: The three implementation challenges with the overlaps

According to the above figure, the challenges are presented as circles which are all intersecting against each other. In principle, the solution domain should be located in the trisection in the middle, or else the other (non-participating) circles will impose their problems to the result. In mathematical terms, the solution must be:

$$Solution = UserExperienceIntervention \cap ImplementationComplexity \cap UIPlasticity$$

Now, let us take a look in each of the areas and assert them against the implementation approach to be taken.

2.1 User experience intervention

The first factor that can affect the implementation path relates to the level of intervention of the solution in the user's interface, and how that would affect his experience with it. The significance of this parameter is quite evident also in the bibliography, where it is stated that a UI with multiple adaptive components can cause unwanted interruptions to the user, and, in addition, automatic and frequent adaptivity of interface may end up causing a feeling of loss of control [15].

This intervention can take place in different ways, such as:

1) **Multiple adaptive components**: when the majority of the components that assemble the UI are changing (irrelevant of the sophistication and excellence of the underlying intelligence in the adaptation decisions), they can mess up with the user's familiarity with the interface and take him out of his comfort zone.
   Putting it into perspective, this indeed can spiral out of control and cause confusion, stress and even frustration. It is really nice and elegant to have dropdown lists that add filter options when data input is large, tables that sort themselves out based on input type, or pop-up windows that advise on next step and/or preselect actions for you; however, if they **all** exist together and there are minimum standard UI components that can act as "reference points" for the user, this can easily escalate into a bad experience for him and bring the adaptation effort to a failure.
2) **Simultaneous adaptation of components**: implementing behavior that changes the UI in an ad-hoc and disorderly mode can result in a complete change of UI that will - at least! - confuse the user. Think of the simplest example: filling a form in a web page application, which can consist of:
   - textfields
   - datepickers
   - radio buttons
   - dropdown lists
   - dynamic tables
   - labels
   - checkboxes
   - graphs
   - togglers
   - etc.

One can only imagine the chaos that might be caused when a dynamic UI implementation, based on some logic, might give an order for all of these components to change real-time. Confusion, surprise and frustration of the end-user are a guaranteed result.

3) **User prompting**: Prompting the user for confirmation when filing in a password is a smart choice; also when he is about to overwrite a saved item, or replace a UI component with a new one of his choice, or submitting a form etc. It is really important to point out that this selection of UI components that will trigger prompting/confirmation has to be done in a very careful and user-centered way. The line between facilitating and irritating pop-up windows is very thin and has to be taken under consideration while implementing adaptation in runtime scenarios. After discussions and feedback from CGI colleagues, it was decided that:
    - **a.** Prompting will be solely used in components connected with underlying critical value (i.e. error messages, save confirmation).
    - **b.** The amount of UI components on-screen and interdependencies between them will be taken into account. A typical example would be a form completion; since multiple fields with important data need be filled, no prompting will be given to the user until he presses the "Submit" button. On the other hand, isolated occurrences of similar components in different pages/context may adhere to the prompting/confirmation rule.

In general terms, the degree on which the adaptation will intervene and interact with the user is a crucial factor that has to be considered in the implementation choices. Manipulating UI components at runtime (adaptivity) or decide upon them based on predefined set of rules (adaptability) has to be done carefully in order not to mess up with the user experience. The required result has to be that the user is facilitated from sublime and easily comprehensive changes, which affect positively his productivity and satisfaction, while having a minimum impact in the output and efficiency of his tasks.

2.2 Complexity of implementation

While discussing about software development and implementation of design, one of the first and most important things that come to mind is the complexity of the solution that will be created. How difficult will this be to implement? How much time will it take to develop? Will it be coherent with what was perceived and intended at the first place?

This kind of issues grow bigger when dealing with POC (proof of concept) solutions, where basically there is no prior knowledge on the topic and the complexity factors; this is evident in the topic of discussion, the effort on achieving UI adaptation in MDD generated software.

No knowledge of the derived software in advance!

One of the most prominent challenges that arise in this occasion is the fact that, when implementing the solution for an adaptable (or adaptive) UI, the software behind this interface is **not** known!

This paradox – term brought up in the CGI sessions – of SPADE is expected; this is an engine which generates software based on input from a requester/client. When attempting to create the user interface, as well as the adaptation rules and techniques, there simply exists no prior knowledge of the underlying software that will be presented in the terms of this interface, apart from the generic category of software that SPADE generates (as stated in chapter 1.2.2). See the figure below (Figure 7):

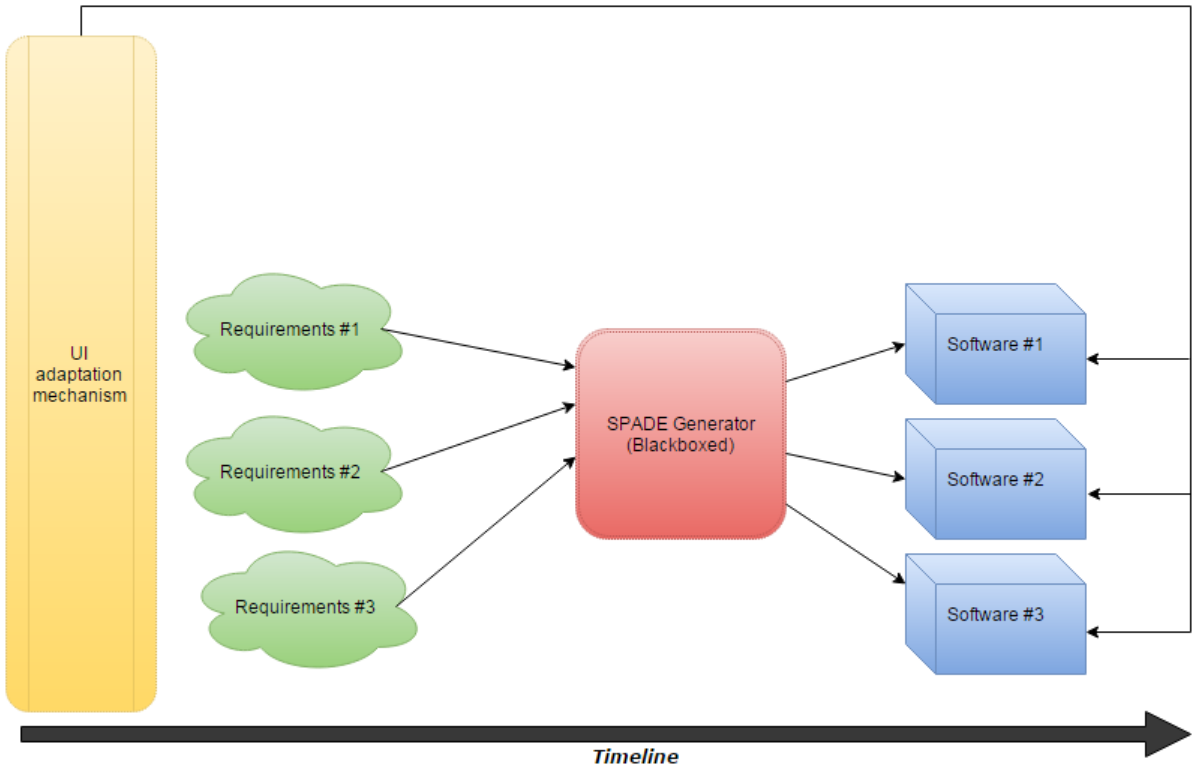

Figure 7: SPADE timeline and the paradox of UI generation

As you can derive from the figure above, the adaptation mechanism is established prior to user's requirements and the generation of the executables. It is also evident that this is a predefined state; there is no workaround in getting this knowledge of software in advance.

Because of this special nature of SPADE engine, the implementation of the UI that can achieve adaptation in all kinds of input is a demanding task by default. Apart from that, there are also challenges that occur in the different adaptation paths (adaptivity and adaptability, as described in the research questions declaration).

1) Adaptability (pre-runtime)

Adaptability, as described before, is the ability to adapt the user interface pre-runtime and based on predefined reasoning. This can be described in the implementation domain as a set of **rules** that have to be defined and explained in a programming way; these have to be:

- clearly defined
- upfront delimited
- tested for non-breaking compilation and/or runtime
- triggered under a well-specified set of conditions

As discussed with the CGI developers, implementing this decision/rules engine directly in a front-end Javascript based implementation would be a difficult task due to the problem dimensions. A middle layer between the backend (SPADE generated software) and the frontend (User Interface) part, where these rules would 'sit' independently and drive the UI based on the input would help greatly in the

solution. This approach was eventually taken and will be discussed further ahead in the implementation part (Chapter 3).

2) Adaptivity (runtime)

In the process of achieving UI adaptation, the part of real-time, dynamic changing of the UI (adaptivity) is arguably the most complex part implementation-wise. In practice, the UI mechanism has to:

- Understand the **data** and the **context** at any given time
  - **data**
    - Type → Float, Int, Double etc.
    - Structure → Objects, Lists, Hashes etc.
    - Batch / Streaming
  - **context**
    - Screen
    - Sensors
    - Location
- Evaluate them against the implemented logic of decision making
- Adapt the UI and its components in an effective, smooth and non task-blocking way

Analyzing the above points into actions, it would require months of implementation within small scrum teams in order to achieve a fully working UI mechanism that takes care of the above points. This was evident after the initial sessions; while at the beginning there was a desire to go all the way through of the above points and try to implement a context and data aware UI, it was eventually decided merely to focus at the **understanding the data** part. The main goal was to create a POC (proof of concept) based on this initial level of UI adaptation, which would serve as a basis for further extension. In Chapter 3 the implementation process, choices and results are explained.

2.3 Plasticity of Interface

The third and equally important challenge that we come across is the actual result that the adaptation will bring to the User Interface. These results, and the effect that they have, can be measured by the notion of plasticity.

Plasticity is a term that is interpreted in the bibliography in many (slightly) different variations, although the root remains stable: it is the term used to describe the capacity of a user interface to withstand variations of the context of use and adapt to changes of the interactive space in order to preserve usability [16]. While in principle a generic term, usability is defined as a set of properties and their corresponding domain of values. Therefore, in the context of a plastic UI, the goal is to maintain these properties within their values while achieving adaptation [17].

In simple words, and as perceived under our research domain context, we want the user interface to be

1) effective,
2) usable,
3) self-learning,
4) fluid,
5) user-facilitating,

while adapting and changing based on the context. It should ideally resemble to the properties of plastic or brain: learn, change and adapt to context changes with the purpose of improvement.

In terms of complexity, it is obviously a non-trivial task; aligning with generic notions of effectiveness and usability while implementing and generating lines of coding is always a difficult task; this jump from abstract to concrete is the main challenge of this type of problems and is something to be always taken into account during the course of this research.

2.4 Keeping the balance

After presenting the challenges regarding the implementation of UI adaptation in MDD software, the main question is how do we deal with them in the coding and development process, and where should we emphasize. Let's revisit the figure of page 16 (Figure 6) and present it in a slightly different way (Figure 7):

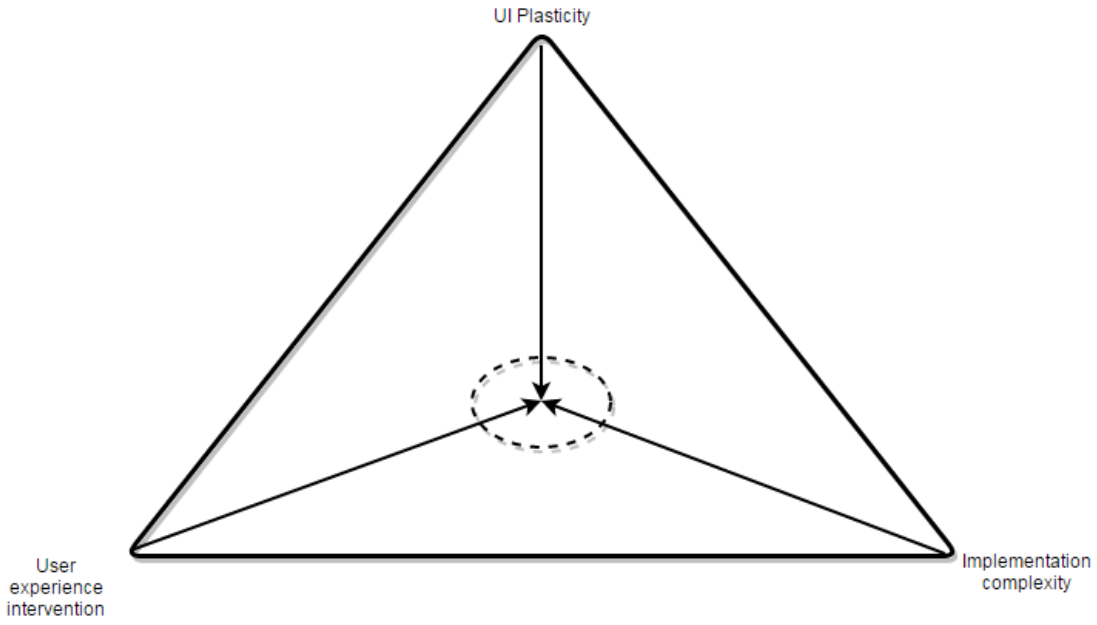

Figure 7: The three challenges triangle and the equally-weighted balance point

We see now the challenges as the corners of an equilateral triangle, and three arrows converging right in the middle of the triangle. This is the **balance** point, and this is what is the desired result is. There need to be a balance between the three challenges and the weights that we will assign to them during the implementation process are equal. This quite simply means that all three of them will be equally

taken into account while implementing solutions for adaptivity and adaptability. The goal is a UI adaptation mechanism which will:

1. Respect the user and facilitate his interaction with the system in a smooth, effortless and unbothered way
2. Will be easy to comprehend, break down, design and implement in a reasonable amount of time and
3. Achieves plasticity, by keeping UI usability and functional effectiveness in acceptable standards throughout the adaptation process.

3 Implementation - design choices and results

In this chapter, the implementation part will be analyzed, which is probably the core part of this research. The effort of implementing the adaptation to SPADE generated software UI is in the spotlight here, whilst tackling the challenges already presented in Chapter 2. Both adaptivity and adaptability implementation paths will be discussed:

- Design choices (usage of Domain Specific Language in adaptability, AngularJS based front-end rules in adaptivity)
- Implementation principles (UI ribbon, use cases) and the reasoning behind them
- Technologies that were used
  - Custom DSL in SMART – Adaptability
  - Javascript / HTML / CSS on an AngularJS framework with custom jQuery libraries – Adaptivity
- Implementation Results

3.1 Adaptability – DSL in SMART

As previously discussed (Chapter 1), adaptability concerts the pre-runtime adaptation of the UI based on a set of specific rules. While dealing with the implementation part, a decision was to be made for the design principles of the way these rules would be interpreted in software. The result should be able to:

1) cope with all the multiple variations of underlying software (generated by SPADE)
2) have efficient adaptation mechanisms that understand the data input and adapt the UI components in the analogous way
3) minimize complexity in implementing and usability as much as possible

In the context of this domain and based on the above requirements, the decision was made to follow the approach of the implementation of rules under a Domain Specific Language (DSL). Domain specific languages are – as the title states – tailored and specified for a particular application domain; by providing notations and terminology which are customized against the specific area of interest, they offer significant benefits on the expressiveness and the ease of use compared to the general purpose programming languages (Java, C, C++ etc.) [18]. The ease of use and tailoring to the specific domain for which it is designed, renders a DSL a very productive and effective way of developing software.

Theoretically, in the adaptability implementation problem, using a DSL satisfies all the above requirements. Due to the UI adaptation specialty, by using a DSL the whole spectrum of underlying software and data input can be covered in a proficient way; this, most notably, can be done in a much easier, quick and efficient way than using a generic language. All the ‘dirty’ work of mapping and creating the low-level code is promised to be done by the DSL itself. In addition, this simplicity and straightforwardness of a DSL can make it usable from more, less specialized people (like stakeholders and clients) who want to get involved in the UI development process. In that scenario, SPADE will be

available to a broader target group, with fewer requirements when it comes to coding and development skills.

For all the above reasons, the usage of a DSL language was decided for achieving adaptation. The next question is **which** DSL language should be used in the terms of the specified problem. The literature states that it's a usually challenging task to develop a DSL language from scratch, given the fact that this would require extended development and coding skills, as well as thorough knowledge of the application domain [18]. There has been extensive research for a DSL that is designed specifically for User Interface Adaptation and could serve as a tool for the implementation part. The results were some dedicated implementations of DSL for adapting UI such as [8] and [19], but all of them were isolated experimental efforts. The solutions found were lacking in:

- Documentation
- User community
- Maintenance / support
- compatibility (old implementations)

As a result, the choices were limited and would require quite some time and dedicated effort to overcome the above difficulties. After some unsuccessful tryouts with some of the languages and consultation with the CGI colleagues, it was decided to use the SMART notation in order to define a **new** DSL language that will be used in creating rules for achieving UI adaptation. The reasons for this decision were the following:

1. **Familiarity**: The SMART notation was already familiar to the thesis writer during the MSc Information Sciences completion, because SPADE software generator was a part of two courses from the study curriculum (Architectural System Design and System Development Management). This prior familiarization would save quite some time in the learning curve and makes implementation faster (especially in the context of the graduation internship were results need be produced in a timely manner).
2. **Business domain correlation**: SMART language was designed for building software requirements that are turned into working software from the SPADE engine. Although not directly related and built for the application domain of UI adaptation, they are associated with the underlying software that the UI is based on; this can be a very useful feature for continuity and usability. It would be the ideal scenario if it is possible not only to translate business requirements but also UI customizations within a single language, providing an easy and fully automatic end-to-end software implementation.
3. **Support / Documentation**: SMART notation is an in-house CGI project; therefore all the support from colleagues as well as documentation can be instantly available, thus facilitating the process of implementation.

Overall, using SMART notation is considered to be quicker, easier and closer to the business domain behind the processes that derive from the SPADE engine. Therefore, it was used as the language to define the rules that will apply adaptability in the UI of SPADE generated software.

3.1.1 Development

During the development process, it was decided to follow a use-case based implementation path, meaning that the definition of use cases and their subsequent implementation was chosen as a tactical approach. This choice of rules would be affected from factors such as:

- Clarity: clearly defined and well stated
- Generalization : applicable to all types of SPADE generated software, not process specific
- Simplicity : non complex and decision sequence based rules

Generally, it was decided to start "simple and quick": small rules, globally applied, which are easy to implement and functional test can be done quickly[5].

Generic implementation decisions

- In SMART notation, the term **process** is used to define a set of specific actions that reach a certain predefined goal; therefore, a combination of processes results in working software (.jar files). In the terms of the customized DSL language, a process is the whole **set of rules** which triggers based on conditions and adapts the UI accordingly. These rules are represented by the goal declaration part, where they are identified (as 'goals'). For example, look at the below declaration structure:

```
Process 'intelliGUI rules'

The following applies:
    " Mapping between generic and concrete UI components"

    and

    "Apply choice component rules"

    and

    "Apply map component rules"
```

  In this example, the process 'intelliGUI[6] rules' is the whole adaptability process, which is realized only when the below three end goals-rules (component mapping, choice and mapping rules) are fulfilled.

- As a part of the engineering process, it was decided to create mappings between abstract and concrete UI components while defining rules. The rationale was to be coherent with the MDD principles of SPADE, with regards to this connection between abstract and concrete via models. A structure of mapping in the user interface level was already in place, thus making it easier to connect components with their abstract (or "generic") categories. A simple example of these mappings is the following :

[5] For further analysis of design decisions and use cases, consult thesis from Arendsen [20]
[6] intelliGUI is the name of the project, as perceived by Edwin Hendriks and CGI colleagues

ChoiceInputComponent	OutputComponent
Radio button list	Data graph
Dropdown list	Pie chart
Checkbox list	Google map

In the above example, there is a clear distinction between the ChoiceInputComponents (where user is required for a selection input) and the OutputComponent (solely display of information on screen).

1) Rule "Radio vs Dropdown list"

The first rule decided to implement is the choice between a radio button list and a dropdown list UI component. The decision would be done based on the data input; specifically, the factor that will affect the decision mechanism is the number of dimensions and entries of the data that are inputted and displayed.

Requirement: Loaded data are univariate (one variable)

Condition: entry number < 7 → Radio button list

entry number > 7 → Dropdown list

Here is the code snippet where the choice is happening:

```
"Apply choice rules" =           //dropdown vs radiobutton list

    one BAR exists in BARS with:

        BAR.TASK = FOO.TASK     //Map abstract (FOO) to concrete (BAR) components

        type = if (FOO.ELEMENTS.count >= 7) and (FOO.ELEMENTS.len == 1)

               then 'dropdown'

               else 'radio'
```

2)Rule "Map generation"

In this rule, the end goal was to create a condition that, when a table would contain values of various coordinates (longitude and latitude), an activation of a google map component will be triggered, putting markers to the respective places where the column pairs (tuples) would point to.

Requirement: Presence of a table data structure with > 3 columns

Condition: two consequtive columns with tag "geo", type double

Below the code snippet where the map creation occurs:

```
"Google maps for ValueOutputComponents that have decimal columns" =

TEMP_MAP = FOO.ELEMENTS[traits contains 'geo']
```

```
if (count(TEMP_MAP) >= 2) and (table_exists == TRUE)  /* 2 columns or more */
    then
        one MAP exists in BARS with:   //map component creation
            BAR.TASK = FOO.TASK
            type     = 'google maps'
           markers  = "elements of {ELEMENTS} to tuples" //add markers
```

With the above SMART code, when a table contains two columns with data tag “geo”, a google map component is activated and populated with the markers based on the tuple values.

3.1.2 Impediment

While working on creating the rules on the SMART language, there was a constant implementation struggle; while the code was created in a logical way and adhered to the SMART principles (it was “passing” from the SMART compiler), there were two main issues faced:

1. Basic programming functionalities were missing from the SMART notation, such as possibility to loop over dynamic conditions, recursive calls and creation of non-sequential programming logics.
2. Many of the classes or objects that were declared in the SMART code while implementing the rules were not incorporated in the SPADE generator engine, thus it would require multiple Java implementations in order to fill in these gaps.

Generally speaking, at some point there was the realization that SMART is **not** optimized for creating and describing other than business processes and linear workflows. After quite some time spent, and with the help of the CGI experts, it was indeed proven that there are certain limitations of this DSL language in the way that it is implemented. For the purposes of this research, in order to create more adaptation rules that will also be **functional** (and not “theoretically” functional), implementation on a non-DSL level of the classes and objects missing would be required. This, in turn, would be very costly both in time and effort, definitely extending more than the time limits of the graduation internship. As a result, it was decided to focus on the adaptivity part (see next Chapter) and propose the extension of SMART language as a part of future work that will bring all the benefits of a DSL acting as a UI adaptation mechanism (see also Chapter 5).

3.2 Adaptivity – intelliGUI

After the hindrance that occurred into creating the DSL language on implementing rules for adaptability in the UI, the next step was to explore the second path of the UI adaptation processes, adaptivity. In this approach, there was the idea to implement adaptivity based on the design decisions that were made (and extensively refined during constant feedback cycles) during the CGI sessions. The two main concepts around which the implementation revolves are:

1) The implementation of the UI ribbon (by design and proposal from Arendsen[20]

2) The implementation of (some of) the use cases, same as adaptability

3.2.1 Technologies – Tools used

In the process of implementing adaptation, the challenges that have been discussed in Chapter 2 are also present here. Implementing the adaptation rules should take into account:

- efficiency in implementation (achieve plasticity)
- keep complexity at a minimal
- enhance user experience and facilitate task execution

It was decided to start up with the most common and standard approach in front-end development, the combination of Javascript and HTML; **Javascript** would deal with the implementation of the logic, while **HTML** would take care of the visual representation of the UI components. There was also a requirement for a framework that would help the code be modular, DRY (Don't Repeat Yourself) and reusable; in this way, the goal was to deal with implementation complexity and produce a clean, discretized, reusable and testable implementation.

The framework that was chosen is **AngularJS**. It is a structural framework for building web apps, based on Javascript, which uses HTML language as a template of the web-based UI and lets us extend its syntax in order to express your application in an extensively custom made level (in this case, customized UI components) [21]. There are several benefits that made AngularJS the choice of the adaptive UI implementation:

1. Model – View –Whatever (MVW or MV*): This approach, although has a somehow non-conventional naming, simply means that AngularJS is a very flexible framework that allows the developer to choose on which level his view and his model (data and logic) will be connected; putting them together and/or separating those two is the developer's decision, and AngularJS takes care of the rest. AngularJS is flexible and this is a huge advantage in implementing a research project when requirements and goals vary during the process.
2. Two-way data binding: Listening changes in objects and variables is much more efficient and easy in AngularJS. When a change occurs in the model, the view side is updated automatically from the framework; no need in setting up listeners and getters/setters, things are made easier.
3. Custom directives: With this powerful feature, create custom UI components that fit you specific requirements can be created once and reused at will; they can be based on premade AngularJS components, or even user-defined. This is very helpful in the sense of adding intelligent UI components in the web application.

Concluding, AngularJS is a powerful, fully supported, flexible and secure framework that can aid in creating a powerful and easy to implement web application that will incorporate the rules for adaptivity in UI.

Apart from the framework, **jQuery** was also used, which is a Javascript library suitable for DOM[7] manipulation. This library has a more UI-manipulation oriented approach and this is evident by many custom open-source solutions that are available online. Some of these were used in the implementation process, also contributing to lowering complexity and also achieving plasticity.

[7] Document Object Model (DOM): an object which is build from HTML and is a representation of the web page (interface)

Regarding the format used for transferring and testing data, the **JSON** (Javascript Object Notation) format was used because:

1. It is compatible with Javascript (JSON data are translated into Javascript objects
2. SPADE uses JSON for sending data in the server – client channel, thus making our implementation more realistic and reliable
3. Complies with the REST API, also used in SPADE
4. Lightweight
5. Simple syntax (i.e. compared to XML)

Furthermore, there was usage of **CSS** (Cascading Style Sheets) along with **Bootstrap** templates in order to edit and customize rendering and appearance of UI components on-screen.

Last, but not least, it is noteworthy to point out the IDE that was used for the UI development process. This was **Brackets**, an open-source text editor, specifically designed for HTML, CSS and Javascript (Figure 8):

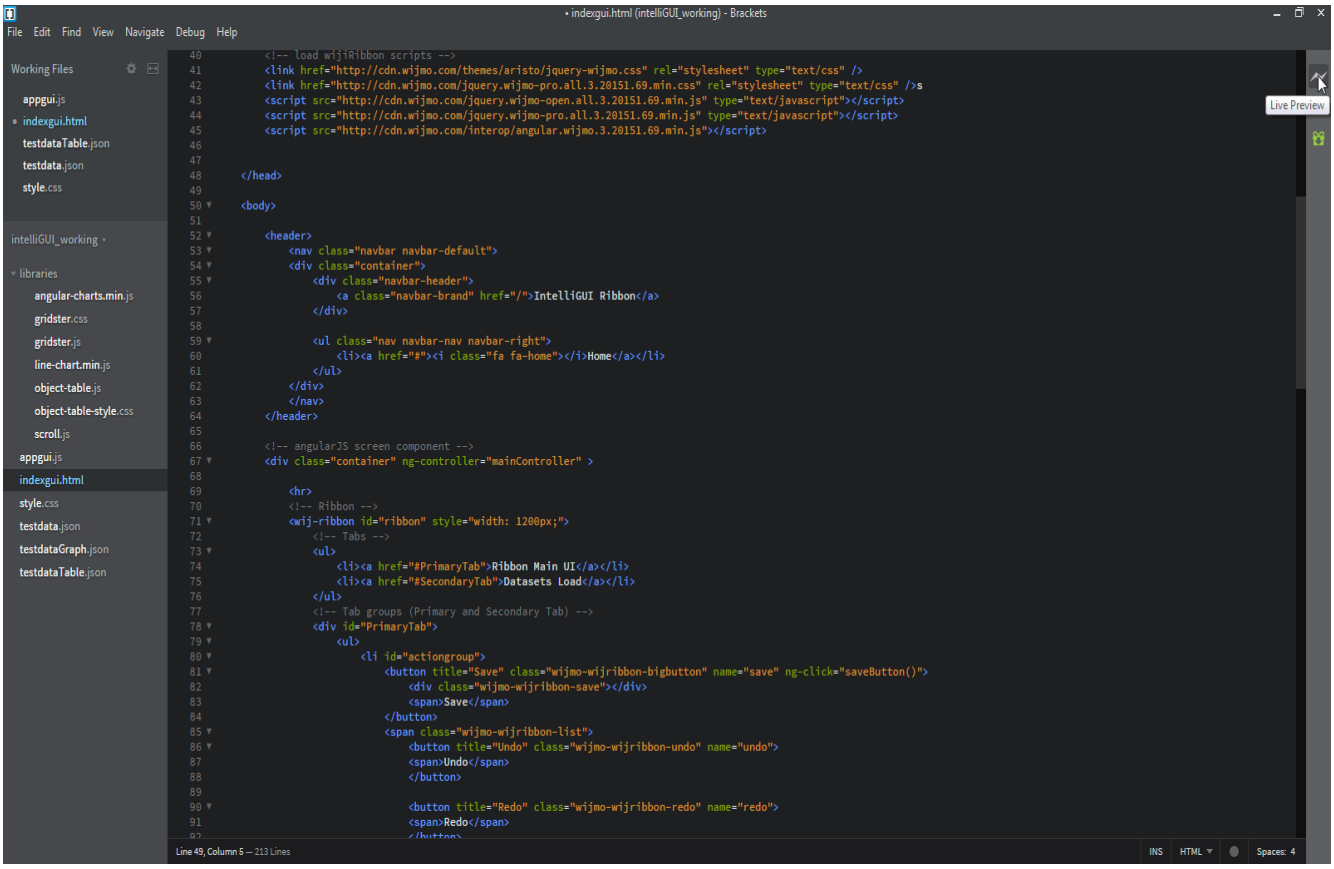

Figure 8: Brackets working environment with Live Preview highlighted

Brackets comes with the project structure on the left side of the screen and all the standard IDE benefits – syntax highlighting, automatic indentation, code hints, debugger etc. The most notable thing though is the Live Preview (seen on Figure 7 on the top right side), which is a static built-in server that simulates the backend logic. With Live Preview activated, a browser window opens, and any changes made on the code are instantly updated on the browser; a feature which greatly facilitates and accelerates the development process.

3.2.2 Implementation – Ribbon and Use cases

As mentioned in the beginning of the adaptivity implementation chapter, the two main reference points around which the implementation revolves are:

- the UI Ribbon and
- the set of predetermined use cases

The implementation of the **UI Ribbon** was a result of the tactical decision to give to the user the power to adjust the interface at will, based on his preferences. This decision has had the collateral effect of slightly deviating from the "implicit" notion of adaptivity (as defined in Chapter 1.6), which means that deliberately a user gets the power to adjust the UI, rather than forcing changes in the background at runtime. This was mainly the result of the feedback that we got, which stated that SPADE users (business analysts, clients with analytical skills and domain knowledge) are a specialized user group who have an opinion about how their interface should look like in order to enhance their experience and productivity. Therefore, in order not to tamper with the user experience (one of the corners from the challenge triangle), the ribbon was implemented as a solution.

See below a screenshot of how the ribbon looks like after the implementation phase (Figure 9):

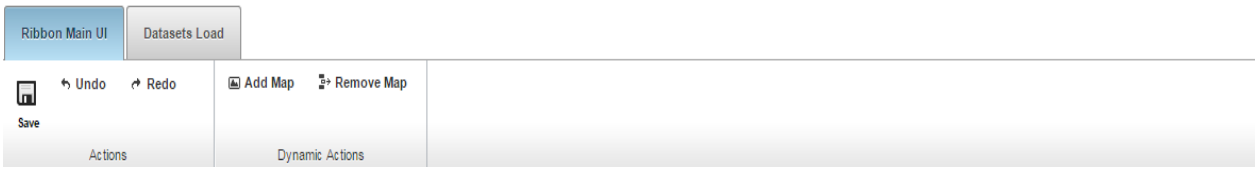

Figure 9: The UI Ribbon

The concept is similar to the Ribbon used for applications such as Microsoft Office; it serves as a static reference point for the user, a point where adjustments regarding the interface can be done at anytime and in all levels of a random task execution. That is where some of the adaptativity mechanisms are triggered from. From an implementation perspective, the ribbon is based on a jQuery custom library called Wijmo [22], where custom AngularJS code is incorporated to add the functionality that is desirable in terms of UI manipulation.

The decision of implementing based on **use cases** was credited to the same arguments used for the adaptability part; clear, well defined requirements, which adhere to user needs and are simple to interpret. In addition, this implementation of use case demonstrates the continuation and linage between the current implementation-centered research and the solution design level research by Arendsen [20].

3.2.2.1 Edit order of UI components at runtime – drag & drop (5.1.2)

In this use case implementation, the user is given the ability to drag & drop UI components, reordering them based on his own personal needs. This functionality is toggled by a checkbox button, and the ordering of the components can be saved.

Requirements: None

Definition: In detail, the user can:

1. Toggle on/off drag & drop mode via a checkbox so that no accidental reordering of components can be done in case the user does a miss click (thus preventing frustration and task execution interruption) – Figure 10:

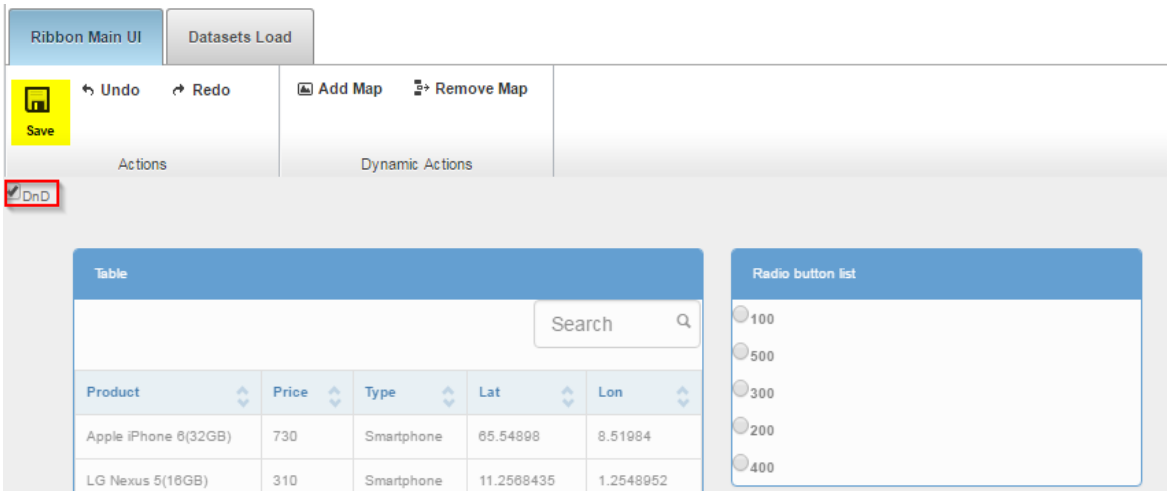

Figure 10: Drag & drop checkbox and the Save Button

When drag & drop is activated, the cursor changes to a pointer when hovered over the UI component's header. This is made to indicate to the user the area that servers as the handler of the component.

2. Reorder components via drag & drop. While dragging and hovering around, a placeholder shifts positions indicating the exact position that the component will "land", assisting out the user in the reordering process.

3. Save the layout. When the user is finished, he can save the layout for reloading it on another session. This is achieved by the **Save** button in the Ribbon (Figure 10). In this implementation, as it is a proof of concept, the layout is not fed in a backend; nevertheless, it is shaped as a JSON object and provided as a popup window (Figure 11), which is exactly the data form that is compatible with SPADE.

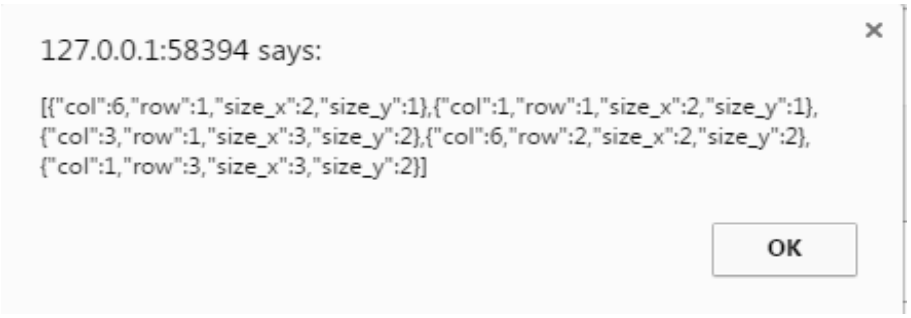

Figure 11: Saving layout in JSON format

For the implementation of the above use case, the jQuery plugin **gridster**[23] was used, which facilitates the creation of a grid –like interface where UI containers are defined. Custom jQuery was created to administer the drag & drop functionality, as is evident by the example below where the enabling and disabling of drag & drop is done:

```
// checkbox toggling on/off drag and drop functionality
$('#checkbox').change(function() {
    if ($(this).is(':checked')) {
```

```
            gridster.enable();
            // hover pointer activated
            $(".panel-heading").css({"cursor":"pointer"});
        } else {
            gridster.disable();
            $(".panel-heading").css({"cursor":"auto"});
        }
});
```

Saving of the layout is quite trivial; just the help of Angular $scope and the connection between view and model is taken care of by the framework. See below the code snippet where the function is defined:

```
//Save button functionality
$scope.saveButton = function() {
    var widgetPositions = JSON.stringify(gridster.serialize());
    alert(widgetPositions);
    console.log(widgetPositions);
};
```

Note that, when saved, the JSON file that stores the representation is not fed into a database, meaning that the front-end for this proof of concept implementation is not connected to a back-end (see Chapter 4.2 for details); instead, an alert box is appearing which displays the JSON file.

3.2.2.2 Dropdown List vs Radio Button List

The second use case is (as in adaptability) the choice between either a dropdown or a radio button list component, based on the condition of the data that are to be displayed in it. The data have to be univariate. The condition is:

- data with lower than 7 entries are displayed in a radio button list
- data with higher than 7 entries are displayed in a dropdown list

Data fetching is made via http requests (with $http AngularJS function). It is noteworthy to say that the two-way data binding functionality (described in 3.2.1) is applied in this situation. See the HTML code snippet below:

```
<div class="panel-heading">Dropdown list</div>
    <select convert-to-number class="form-control">
    <option ng-repeat="listitem in numberData">{{listitem}}</option>
    </select>
```

In this case, with the ng-repeat custom AngularJS directive we iterate over the numberData variable (our data) and populate the UI component → {{listitem}}. When a change occurs in the variable (e.g. a new dataset is loaded) the view will **also** change the displayed data, without additional coding needed.

3.2.2.3 Google map component (5.1.5)

This use case includes addition of a Google maps UI component at runtime, based on a specific action/ set of actions from the user while interacting with the UI.

Requirements: Data loaded in a table view. Two consecutives columns must include:

1) Numerical values with decimal digits and

2) The decimals have to be more than four

Definition: There exists a table with data loaded by the user, where there are two consecutive columns where latitude and longitude of specific coordinates are displayed – say, for example, a set of products along with their tracking status). The user is able to select the specific rows of the products he would like to track down (Figure 12):

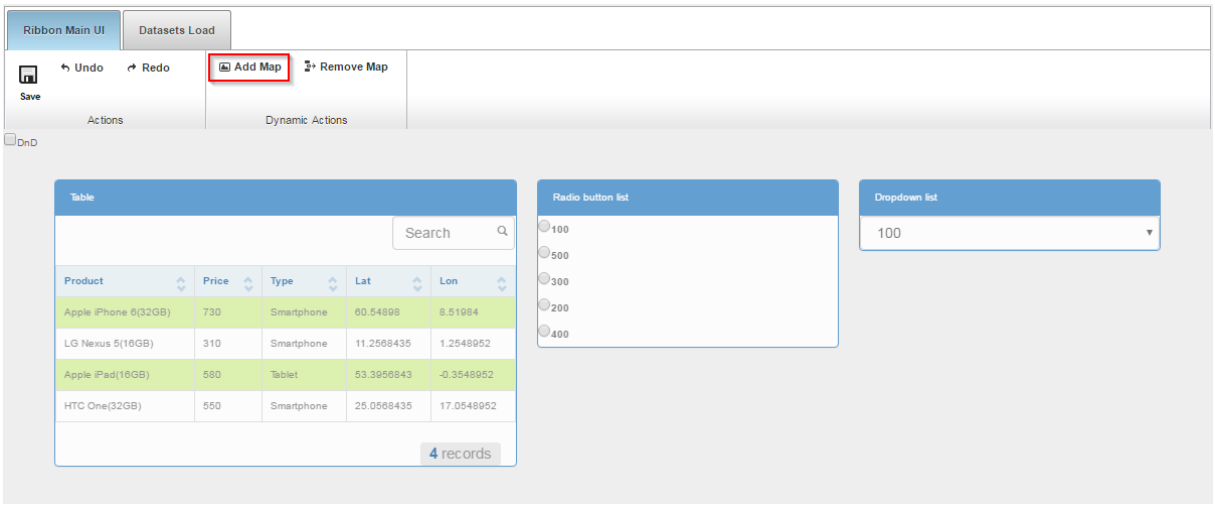

Figure 12: User selection of rows to display and the "Add Map" button

Once there are products selected, by clicking in the ribbon button "Add Map" (highlighted in red above), a Google Maps UI component appears with the positions of the products pinpointed with markers (Figure 13):

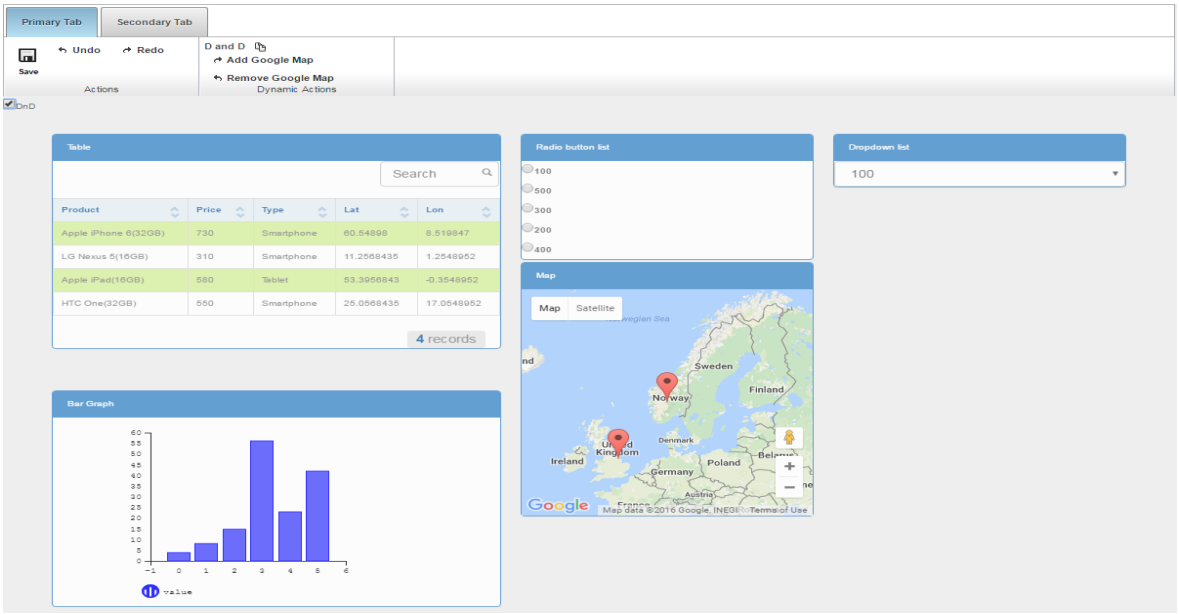

Figure 13: Google Maps with markers based on the latitude and longitude data selected

There is also an error handling alert to improve user experience; when a user presses the adding a map button without prior selecting locations to display, an error message is displayed as shown in Figure 14:

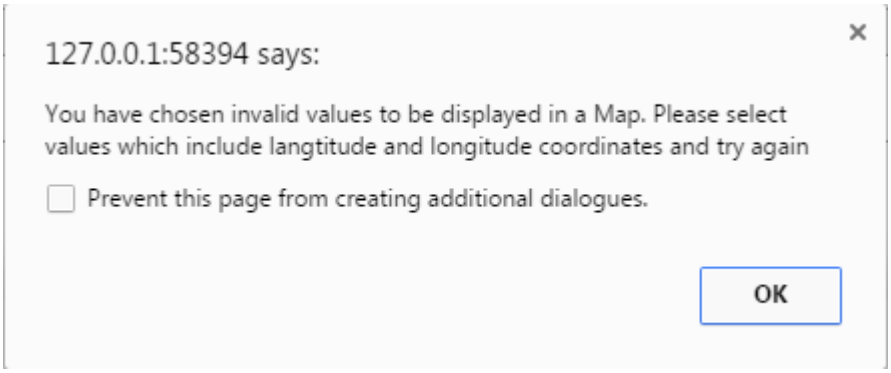

Figure 14: Alerting on "Add Map" button

The message informs the user on missing proper location data and asks him explicitly to select the data with the right attributes to be displayed. In code, this validation is done (as explained before) by checking whether the decimals of the selected values are more than four. See the code snippet below:

```
// Condition for map: >= 4 decimals
var validateMap = 0;
    for (i=0; i<$scope.selectedValues.length; i++) {
        if (get_decimals($scope.selectedValues[i].lat)>=4 &&
            get_decimals($scope.selectedValues[i].lon)>=4) {
            validateMap++;
        }
    };
```

As you can see, the check sees the decimal number of the numbers selected (via the get_decimals function) and, if gets true, increments the validateMap counter which is used later for the creation of the markers.

These are the main use cases that have been implemented in the adaptivity part. All together they form a totally dynamic UI, with the UI components and the underlying intelligence behind them, resulting in an MVP (minimum viable product) that was accepted by the CGI supervisor and the other colleagues working on this. In fact, there was designed a demonstration video to explain the benefits that this adaptive UI implementation brings on to SPADE; this acknowledgment of the research effort was a great reward and a validation that the results were in the right direction and produced value both in research and business domains. We will discuss these results, amongst others, in the next chapter.

4 Evaluating the results

In this chapter, the evaluation of the implementations efforts in both adaptability and adaptivity will be presented and discussed extensively. Furthermore, all impediments and/or deviations from the implementation plan will be explained. Purpose is to expose the value that was derived from the research paths that were taken, implicitly making justifications over the choices of the aforementioned paths.

For readability and consistency reasons, we will divide this chapter in the two different implementation paths: adaptivity (intelliGUI) and adaptability (DSL in SPADE).

4.1 Adaptability (DSL in SPADE – SMART notation)

On the path towards UI adaptation, the adaptability part includes the pre-runtime part of defining the dynamic behavior of the interface. This behavior is translated as designing, defining and implementing specific set of rules which will decide upon the things that will change in the UI. As for the implementation, the decision of creating a custom DSL language based on SMART notation (the language used to input requirements into SPADE) was taken, for all the reasons that have been already described in Chapter 3.1.

As it was discussed before, the creation of the DSL language and the consecutive building of rules on top of it to manipulate and adapt the GUI have imposed a **middle-layer mechanism** between the high level ideas and the low level implementation in Java code. The profound benefits of developing adaptability rules in SMART are:

- easy to understand (closer to human language than generic programming languages, thus clients can learn to read and even develop quicker)
- easy to use
- once extended, tailored to the user adaptation domain
- easily pluggable to SPADE (SMART is already used for business requirements specification)

In general, this middle-layer mechanism is attempting to bridge the gap between high level design ideas of a system which adapts its interface and the low level, "dirty" implementation in a generic language.

With the result of the set of rules written in SMART language which are described in Chapter 3, many of the above beneficial points are confirmed. SMART notation is easy to use, because it is closer to the human language, and especially close to the business domain of SPADE. When following the generic rule of mapping from abstract to concrete UI components, the implementation is promising to be quite straightforward:

1. define the rule
2. define the UI components affected
3. map abstract → concrete components
4. set variables
5. set the rule (if…condition then do …action)

During the implementation process of these rules and after quite some effort of solidifying and defining the above steps, creating a new rule was quite straightforward, proving the introduced advantage of simplicity. Eventually, when a new rule that will adapt the UI needs to be realized, the only thing required is a single SMART requirements piece of (high level) code that will define the rule in an easy and quick way.

Everything else is taken care of SMART and the SPADE transcompiler; creating the Java classes, calling the functions, creating variables, compiling and deploying (Figure 15). Implementation in low-level is **not** our concern anymore. A well-defined set of UI rules in a SMART file is enough to kick-off adaptability and teach the system on how to change the UI to improve productivity and user experience. The gap described before can be bridged.

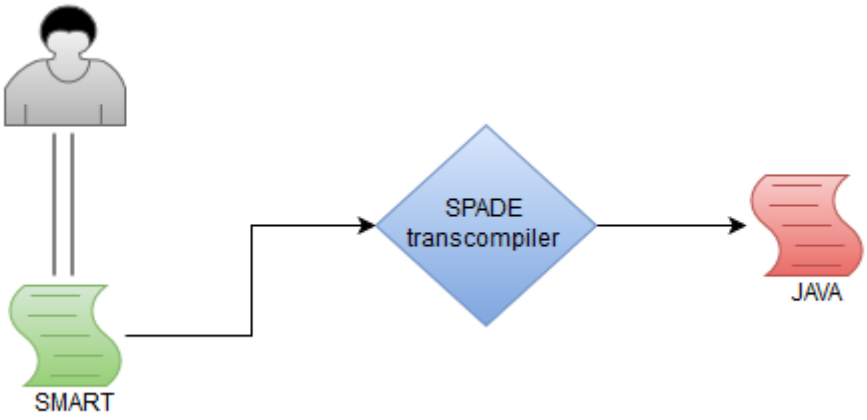

Figure 15: From SMART to low-level JAVA

However, as it has been already mentioned in previous chapters, the adaptability implementation part was halted at a certain point. This happened mainly because SMART language turned out to be of limited capabilities when it comes to implementing UI adaptation functionalities. After a specific point during the tryouts, it was proven that it would take quite some effort and time to extend the language and improve it in order to support the UI rules that would be created. It was discussed with colleagues within CGI and with the internal supervisor that the time and effort would exceed the boundaries of this research, thus concluding the implementation efforts for adaptability at a fragmentary result.

In spite of the impediment, and evaluating the solution as a whole, it is quite obvious that using a DSL language to describe UI rules is a solid solution in the model-driven development domain. Especially by taking the example of SPADE, using its own notation to create this language can prove (once fully implemented) a tactical solution in this field. With this approach, the easiness and simplicity of use that a DSL language has is exploited to make things dynamic; adaptation rules can be added and removed in an ad-hoc basis. The extensibility and the plug-in functionality that were discussed before are also a part of the benefits.

It is beyond question that, extending the SPADE engine in order to be able to understand UI rules written in SMART would take quite the effort, both in manpower and in time. The amount if this effort is not precisely estimated, something expected due to the nature of this research. However, it is safe to say after quite some time of experimentation with the language and the tool itself that once fully extended and implemented, the usage of SMART as a DSL language for defining adaptability rules for SPADE generated software is a solid solution.

4.2 Adaptivity (intelliGUI)

While the adaptability part was a case where the implementation process has reached a halt quicker than expected (although it is valuated to be a tactical solution once fully implemented), in the case of the implementation of adaptivity things were different, and more results were produced.

In the adaptivity part, the main purpose was to adapt the interface via a well-structured mechanism at runtime and in an implicit (for the user) way. This "implicit" philosophy was abandoned as a tactical solution, since changing things without human initiation would not adhere with the knowledge level of the SPADE users target group. This means that users of SPADE generated software are aware of their respective business domain and, consequently, have explicit knowledge on what they want to see on the screen and in which way. Therefore, it made more sense for us to give more control to the user when it comes to changing the UI based on his needs; this choice was followed in the majority of the adaptation scenarios and use cases that were applied during the implementation process.

The result of the efforts (as already seen in Chapter 3.2.2.) is a new, independent and fully functional front-end interface which incorporates the predefined set of dynamic behaviors that result to adaptivity (Figure 16).

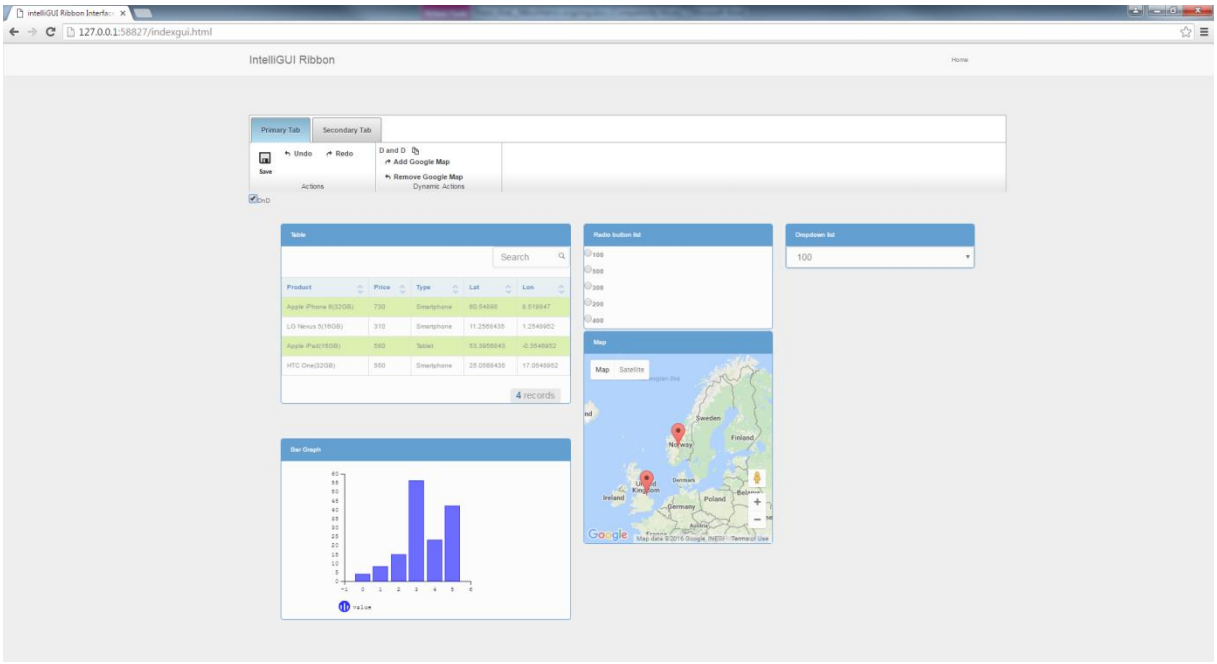

Figure 16: intelliGUI interface – implementation of adaptivity

The interface depicted in the figure above is an integrated solution that realizes the scenarios that were presented in a thorough way in the previous chapter; add/remove map, drag & drop, customizable table etc. All of the use cases are there and, in conjunction with the ribbon implementation (seen on top), a complete user interface is in place with adaptive components.

The creation and usage of this interface is based on input from CGI colleagues and expert SPADE users. It adheres with the most common representation of data and processes that derive from generated software, thus making it a quite representative result of the expected SPADE front-end interface (albeit simplified, since it is still in a proof of concept stage).

It is also critical to note that the data that are being displayed in the components are dummy data, meaning that the interface is not connected to a back-end. That said, 1)the dummy data used are

identical as the data used by SPADE processes (JSON files) and 2)fetching of data is done with a RESTful way (via http requests), in the same way that SPADE is retrieving data. On a first look, it is a bit bizarre to fetch local dummy data via http requests, but this was implemented this way exactly for the reason of simulating the way the data would be retrieved if it were for a production environment. Based on those points, it is made clear that with minimal effort the interface can be plugged in and connected with SPADE.

In an overall view, the resulted interface satisfies the adaptation expectations and is fully functional. The main requirements that derived during our intra-team standup sessions are satisfied, adherence with design principles is realized and the implementation choices with regards to implementation paths (e.g. REST calls) and technology choices (i.e. AngularJS) promise pluggability and extensibility.

There is, however, a small hitch in this implementation of adaptivity. Although it works quite smoothly and the result in screen is what is generally desired (in the context of the use cases), it is a fact that the coding part is not as "clean" and structured as it could be. This means that, in essence, there was not a clear philosophy and/ implementation principle that was predefined and followed through the process, but rather an ad-hoc implementation of rules and adaptation behaviors that were put together. The most prominent example of this absence of this global structure is the fact that there was an intention to implement a **rules engine** in Javascript; a centralized decision mechanism that would take the user and context data as input, choose the rule applicable and apply it to generate the desired output. This centralized mechanism was not implemented; instead, each rule had its own implementation mechanism, which was attached to the respective UI components. The two main reasons that this unstructured implementation was followed are:

1. The research nature of the project. Since this is a research regarding adaptation in MDD generated software, it was not clear from the start what the desired result would be and what is the feasibility of the various designs followed. Therefore, the tryouts that were developed from the start slowly shaped the current existing solution. It was only after a significant amount of time (3 months after the research process started) that the idea of a structured and centralized rule engine implementation came up; by that time, intelliGUI was almost complete and a major and extensive refactoring would be required.
2. The time limitations. Refactoring and/or creating a rules engine from scratch would take much time and effort which was not available for the current thesis purposes.

Besides the aforementioned drawbacks in the solution structure, the actual result of adaptivity is remarkably functional and stable. It incorporates the adaptation behavior that was desired from the design phase and the result is a smooth a dynamic UI that is flexible with regards to the software that lies beneath it (backend). The usage of technologies such as the AngularJS framework provides the benefits of implementation simplicity, as well as scalability. It is also significant to note that this result interface was such well perceived by the CGI supervisors that it was used as a testament of success towards management for extending the intelliGUI project; a demo-video of the main functionalities of the solution was used as a proof of success of the project, which is more than rewarding for the efforts and amount of time put into this whole process.

5 Conclusion and future work

In the last chapter of this thesis, a general conclusion will be given with subsequent suggestions for future work to extend and build on the results.

Looking back to the origins of this research, the main goal was to deal with the inflexibility of the UI of Model driven developed software (in this case, SPADE software outputs). The joint venture with Joeri Arendsen was split in two parts; he focused on the design level of UI adaptation and the definition of the use cases, while I focused on the implementation part and the challenges imposed. After identifying the ways of implementing adaptation between adaptivity and adaptability, the research questions were generated:

- How to manipulate GUI components towards adaptability in Model Driven Development generated software?
- How to manipulate GUI components towards adaptivity in Model Driven Development generated software?

The answer towards **adaptability** was the introduction of a custom DSL language, defined with the usage of SPADE's SMART notation, which serves as a middle-layer and a link between high-level adaptation rules and low-level implementation. Creating the rules in this way, just by defining them in a closer-to-human language, makes adaptability quicker to implement and scalable; it would only take a couple of lines to add a new rule that a user would like. It is also obvious that by following this approach, no external libraries or dependencies would be required for deployment and execution; this process would be part of the native SPADE compiler. Efficiency, speed and flexibility are the results of using this custom-made DSL in SMART notation.

It is a fact that this approach would require some significant time and effort to be dedicated in order to be fully implemented and realized. SMART notation is **not** designed to represent UI components, and to overcome this obstacle, extensive addition of functionality and underlying logic has to be added by a developer (or more). Nevertheless, once this is done, the DSL approach really seems to be a structural and efficient solution to manipulate the UI. It solves the issue of complexity and extensibility, facilitating the effort of making user interfaces that are specifically tailored to each and every user.

On the other big part of this research, the adaptivity part, implementation results were richer and more extensive that in the adaptability level. From the moment the DSL language in SMART was deemed too big of a project to tackle in the context of this thesis, intelliGUI took center stage in an attempt to achieve adaptation at runtime via the use cases defined. The result, a full working prototype, satisfying the majority of the use cases, is a proof of achieving flexible and adaptive UI which visualizes software that is not known on implementation time. With the data serving as a main source of information, the intelliGUI is manipulating UI at runtime both implicitly and explicitly, with respect to user experience. Usage of technologies and frameworks such as Javascript, AngulaJS and jQuery provides the ability to not perplex the development process. Instead, with small chunks of code and a relatively quick pace (implementation time $\leq$ 2 months), a well functioning prototype with respect to all implementation challenges (as defined in Chapter 2) was created.

There is a point of attention with regards to this implementation that can be a point of future work, and it concerns improving/refactoring an implementation to a better structured solution. As it was

explained in the previous chapter (Chapter 4.2), the solution for adaptivity does not fit in a centralized solution as a decision mechanism, but rather than a combination of spread decision mechanisms around the Javascript code. That being said, a build-up (or refactoring based on the developer's judgment) would make sense to serve as a next step in the future. The most straightforward path that could be followed in this sense is that of a rules engine, which would modify a solution into a standardized piece of code with clear definition of functionality and usability.

In the end of the day, implementing adaptability is facilitated by the use of a DSL language which, once implemented on top of SMART notation, provides a robust structural solution for achieving adaptation. On the other hand, adaptivity is already implemented in the form of intelliGUI and provides a working solution in UI adaptation; however, the solution hinges on further improvement and/or refactoring to reach levels that can qualify it as a tactical and robust result.

The results of both adaptivity and adaptability can be summarized in the following table, where a parameter comparison is done:

Adaptability	Adaptivity
DSL in SMART notation	Javascript/AngularJS
Meet implementation challenges (Chapter 2) after SMART extension	Meet implementation challenges (Chapter 2)
Structural solution	Non-structural solution, needs refactoring
Not full implementation	Full proof-of-concept implementation (intelliGUI)

As a closing note, it is noteworthy to mention that this research topic is a so broad that in order to achieve a thorough and exhaustive research result, it would probably qualify as a PhD project; there is great abundance in the aspects and ways you can think of adapting a UI for software deriving by MDD. One can start thinking about adapting based on factors other than input data; context of use, device, user personalization etc. This project is mainly intended to open up the discussion in this area, provide a set of possible solutions along with evidence of feasibility (i.e. implementation results); in that context, it has been generally perceived and considered as successful.

6 Academic References

[1]R. Kennard and J. Leaney, 'Towards a general purpose architecture for UI generation', Journal of Systems and Software, vol. 83, no. 10, pp. 1896-1906, 2010.

[2]J. Oh and N. Moon, 'Towards a cultural user interface generation principles', Multimedia Tools and Applications, vol. 63, no. 1, pp. 195-216, 2012.

[3]M. Baldauf, S. Dustdar and F. Rosenberg, 'A survey on context-aware systems', International Journal of Ad Hoc and Ubiquitous Computing, vol. 2, no. 4, p. 263, 2007.

[4] Jameson, Anthony. Adaptive interfaces and agents. Human-Computer Interaction: Design Issues, Solutions, and Applications, 105, 2009.

[5]G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, L. Bouillon and J. Vanderdonckt, 'A Unifying Reference Framework for multi-target user interfaces', Interacting with Computers, vol. 15, no. 3, pp. 289-308, 2003.

[7] Morzy, Tadeusz, et al. Implementing Adaptive User Interface for Web Applications. In: Intelligent Information Processing and Web Mining. Springer Berlin Heidelberg, p. 97-104, 2003.

[8] C. Stephanidis, A. Paramythis, C. Karagiannidis, and A. Savidis, "Supporting interface adaptation: the AVANTI Web-Browser," in Proc. of the 3rd ERCIM Workshop on User Interfaces for All, Obernai, France, 1997.

[9] R. France and B. Rumpe, "Model-driven development of complex software: A research roadmap," in 2007 Future of Software Engineering, 2007, pp. 37-54.

[10] S. J. Mellor, T. Clark, and T. Futagami, "Model-driven development: guest editors' introduction," IEEE software, vol. 20, pp. 14-18, 2003.

[11] B. Hailpern and P. Tarr, "Model-driven development: The good, the bad, and the ugly", IBM Syst. J., vol. 45, no. 3, pp. 451-461, 2006.

[12] J. S. Sottet, G. Calvary, J. Coutaz, and J. M. Favre, "A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces," Lecture notes in computer science., pp. 140-157, 2008.

[13] D. Thevenin and J. Coutaz, "Adaptation and plasticity of user interfaces," in Workshop on Adaptive Design of Interactive Multimedia Presentations for Mobile Users, 1999, pp. 7-10.

[14] J. Fink, A. Kobsa, and A. Nill, "User-oriented adaptivity and adaptability in the AVANTI project," in Designing for the Web: empirical studies, 1996.

[15] S. Kumar and A. Sekmen, "Single robot–Multiple human interaction via intelligent user interfaces," Knowledge-Based Systems, vol. 21, pp. 458-465, 2008.

[16] G. Calvary, J. Coutaz, D. Thevenin, Q. Limbourg, N. Souchon, L. Bouillon, et al., "Plasticity of user interfaces: A revisited reference framework," in In Task Models and Diagrams for User Interface Design, 2002.

[17] L. Balme, A. Demeure, N. Barralon, J. Coutaz, and G. Calvary, "Cameleon-rt: A software architecture reference model for distributed, migratable, and plastic user interfaces," in Ambient intelligence, ed: Springer, 2004, pp. 291-302.

[18] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," ACM computing surveys (CSUR), vol. 37, pp. 316-344, 2005.

[19] A. Behring, A. Petter, and M. Mühlhäuser, "A Domain Specific Language for Multi User Interface Development," in Modellierung, 2010, pp. 335-350.

[23] G. D. Abowd, J. Coutaz, and L. Nigay, "Structuring the Space of Interactive System Properties," Engineering for Human-Computer Interaction, vol. 18, pp. 113-129, 1992.

7 Non-Academic References

[6] Netbeans.org, "NetBeans IDE - Swing GUI Builder (Matisse) Features", 2016. [Online]. Available: https://netbeans.org/features/java/swing.html. [Accessed: 08- Feb- 2016].

[20] J. Arendsen, "Exploring the Design Space for Dynamic Interfaces," Institute for Computing and Information Sciences, 2015.

[21] "AngularJS", Docs.angularjs.org, 2016. [Online]. Available: https://docs.angularjs.org/guide/introduction. [Accessed: 20- Jun- 2016].

[22] "Ribbon", Wijmo.com, 2016. [Online]. Available: http://wijmo.com/docs/wijmo/Ribbon.html. [Accessed: 21- Jun- 2016].