BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

LANGUAGE FOR HIGH-LEVEL DESCRIPTION OF USER INTERFACE REQUIREMENTS

VYSOKOÚROVŇOVÝ JAZYK PRO POPIS UŽIVATELSKÉHO PROSTŘEDÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. MARTIN RAŠOVSKÝ
SUPERVISOR VEDOUCÍ PRÁCE	Ing. RADIM KRČMÁŘ

BRNO 2018

Abstract

This master's thesis investigates new high-level language for description of graphical user interface. The theoretical part of this thesis studies the UI technologies and software methodologies from a side of general requirements on new language. From that, we derive general requirements specifying what it is meant to be a functional part of an UI. We also list requirements for special educational needs according to known *computer therapy design principles*. According to analyzed requirements is proposed a design of new language, including an algorithm of composition of UI components for further implementation of the language. Designed language is implemented in C# programming language and is demonstrated on a representative example. We conclude this work with outlining further extensions of the proposed language.

Abstrakt

Diplomová práce se zabývá problematikou návrhu vysokoúrovňového jazyka pro popis grafického uživatelského rozhraní. Teoretická část rozebírá současné technologie uživatelských rozhraní zejména pro stanovení požadavků na nový jazyk. Z těchto poznatků následně jsou shrnuty zásadní požadavky, které se nutně musí zakomponovat při návrhu samotného jazyka. Jsou zde zmíněny i požadavky z pohledu osob se specifickými vzdělávacími potřebami dle tzv. návrhových principů *počítačové terapie*. Následně práce dle analyzovaných požadavků navrhuje jazyk pro vysokoúrovňový popis uživatelského rozraní. Součástí návrhu jazyka je také popis algoritmu pro kompozici jednotlivých komponent definovaných v jazyce do výsledného uživatelského prostředí. Navržený jazyk je implementován v programovacím jazyce C#. Implementace je demonstrována na reprezentativních příkladech. Nakonec se práce věnuje dalším možným rozšířením jazyka.

Keywords

Graphical user interface, high-level language, usability, accessibility, touch screen design, *computer therapy*, design principles, MDE, SBVR, OCL, visual programming.

Klíčová slova

Grafické uživatelské prostředí, vysokoúrovňový jazyk, použitelnost, přístupnost, dotykové rozhraní, *počítačová terapie*, návrhové principy, MDE, SBVR, OCL, vizuální programování.

Reference

RAŠOVSKÝ, Martin. *Language for High-Level Description of User Interface Requirements.* Brno, 2018. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Radim Krčmář

Rozšířený abstrakt

Pojem grafické uživatelské prostředí[1] je v oblasti informatiky část aplikace umožnující uživateli snadnou interakci se systémem, např. pomocí různých grafických tlačítek a textových polí. Postupem času byly vyvinuty nástroje pro rychlejší a jednodušší vývoj těchno uživatelských rozhraní. Pomocí těchto nástrojů se začaly vyvíjet systémy s uživateským prostředím využívající podobné ovládací prvky a díky tomu se uživatelé nemusejí učit jak používat uživatelské prostředí v každé nové aplikaci. V současnosti je téměř většina moderních aplikací rozšířena právě o toto grafické uživatelské prostředí. Na druhou stranu pokud programátor chce aplikace inovovat do novějších technologií vzhledem k uživatelskému prostředí, tak musí typicky přepsat zdrojový kód. Daný proces následně stojí mnoho nákladů. Pokud se podíváme na příklad vývoje ASP.NET technologií, tak zde mohla být daná aplikace napsána v technologii ASP.NET Web forms. Po několika letech byla vydána další technologie ASP.NET MVC. Pokud by právě programátor chtěl využívat nejnovější technologii, musel by pak aplikaci v technologii ASP.NET Web forms přepsat do technologie ASP.NET MVC. Navíc se obě technologie liší syntaxí a architekturou. Tedy abychom dosáhli migrace technologie uživatelského prostředí, firma musí typicky investovat do vývoje zcela nové aplikace. Navíc výsledná migrovaná verze může vypadat zcela stejně jako ta původní.

Přesně daný vysvětlený problém se snaží řešit programátoři při vývoji komerční aplikace corima, vyvíjené firmou COPS Gmbh. V této firmě vznikl požadavek vytvořit tutéž aplikaci, avšak v jiné technologii uživatelského rozhraní. corima je mnohouživatelská client-server aplikace and aplikační platforma v jednom. Serverová strana i klientská strana je vyvíjena ve frameworku .NET. Serverová strana je vyvíjena v technologii .NET WCF, zatímco klientská strana je vyvíjena v .NET WPF technologii. Jelikož většina business logiky je umístěna na serveru, není problém tuto business logiku opětovně použít novým klientem. Problém je ovšem s opětovným použitím uživatelského prostředí. Proto vznikl nápad vytvořit mechanismus konverze jedné .NET uživatelské technologie do druhé. Abychom dosáhli dané konverze, bude pravděpodobně nutné vytvořit reprezentaci uživatelského prostředí nezávislou na konkrétní .NET technologii. Reprezentace uživatelského prostředí by měla být vysokoúrovňová a měla by obsahovat prostředky pro popis běžných uživatelských prostředí v aplikační platformě corima.

Přístup k návrhu vysokoúrovňového popisu uživatelského prostředí se bude odvíjet od funkce komponent uživatelského prostředí.

Výsledným cílem práce je navrhnout deklarativní vysokoúrovňový jazyk pro nezávislý popis uživatelského rozhraní. Uživatelské prostředí popsané v tomto novém jazyce bude dále vstupem do tzv. generátoru. Tento generátor je program v jazyce c#, který na základě vstupu bude generovat adekvátní výstup ve formě uživatelského prostředí v konkrétní cílové .NET technologii uživatelského prostředí. Použití generátoru bude demonstrováno na typických CRUD formulářích, které byly vysvětleny v textu práce.

Následně jsou zanalyzovány požadavky pro návrh tohoto jazyka. Tyto požadavky jsou nezávislost technologie uživatelsého rozhraní, snížení nákladů během migrace z jedné technologie uživatelského prostředí do druhé technologie uživatelského prostředí, oddělení funkce a konstrukce v daném jazyce, zohlednění různých atributů komponent uživatelského prostředí, zohlednění požadavků ohledně finančních aplikací a možnost propojení uživatelského prostředí s back-end logikou.

[1] Pro účely zkrácení textu budeme používat pojem *uživatelské prostředí*

Na základě těchto definovaných požadavků je navržen nový jazyk. Tento nový jazyk je navržen jako meta-model a to z toho důvodu, že jeho reprezentace může být jak grafická, tak textová, např. XML. V práci je uvedeno schéma tohoto meta modelu s vysvětlením jeho důležitých částí. Nyní řekneme, že daný jazyk se zkládá z následujících entit: technologie uživatelského rozhraní, komponenta reprezentující fukci, konstrukce, stránka uživatelského rozhraní, obecná vlastnost, kontrukční vlastnost. Dané entity definují stránku uživatelského rozhraní a spolu s technolgií jsou dále vstupy do generátoru. Nakonec v návrhu definujeme algoritmus pomocí něhož generátor komponuje definované konstrukce v jazyce do výsledného uživelského rozhraní.

Implementací tohoto jazyka v jazyce C# je vhodné převedení návrhu jazyka jako meta-modelu do reálného použití v praxi do jazyka C#. Jednotlivé entity jsou převedeny do zdrojového kódu a je implementován algoritmus generátoru a vhodně rozšířen pro použití v aplikační platformě corima. Je zde ukázáno, že systém je schopen generovat CRUD formuláře a propojit uživatelské prostředí s back-end logikou aplikace.

Vyhodnocení ukazuje, že byly splněny všechny body zadání včetně vytyčených konkrétních cílů během práce. V rámci vyhodnocení bylo ukázáno, že generátor je schopný vzít v úvahu různé definice uživatelského rozhraní a podle nich generovat různé realizace. Zároveň bylo ukázáno, že systém je schopný generovat uživatelské prostředí definované pro jedince se specifickými vzdělávacími potřebami.

Možné další rozšíření práce bylo shledáno v optimalizaci algoritmu pro kompozici uživatelských komponent navrženého jazyka. Problém může nastat, pokud v jazyce bude definováno příliš mnoho komponent s podobnými obecnými vlastnostmi. V tomto případě může daný algoritmus vybrat méně vhodnou konstrukci dané komponenty uživatelského rozhraní.

Language for High-Level Description of User Interface Requirements

Declaration

Hereby I declare that this masters's thesis was prepared as an original author's work under the supervision of Mr. Ing. Jiří Fiala, Ing. Krčmář Radim, and Ing. Ondřej Dvořák. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Martin Rašovský
May 23, 2018

Acknowledgements

I am using this opportunity to express my gratitude to the Ing. Ondřej Dvořák who supported me throughout the completion of this thesis.

Contents

Chapter 1

Introduction

1.1 Motivation

In information technology, a graphical user interface (GUI)[1] is a part of a software giving a user control over the application, for instance using buttons and text fields. Currently, most of the modern software applications are extended with GUI. A program with GUI includes graphical controls that user can control with mouse, keyboard or with touch screen. Over time user interface (UI) technologies have evolved to give developers the ability to create these GUI easier and faster. The developer is now able to create new applications having similar design and users do not have to relearn the interface. On the other hand, a switch to a new technology usually requires to rewrite the code. Such a re-engineering of aging system is related with a big cost. E.g., in development of web based ASP.NET[2] application, an application was initially written in ASP.NET Web forms[3]. After few years, new technology called ASP.NET MVC[4] was released. Although it shares the name with original ASP.NET, it differs in syntax and architecture. Next technology in ASP.NET was introduced ASP.NET API that came up with the use of JavaScript frameworks, e.g., AngularJS, Knockout, or React. To gain features offered by these technologies, the company must invest into rebuilding the whole software. However, the resulting UI mostly looks and feels the same, only the technology underneath changes.

Therefore the motivation of this thesis is to reduce costs of migrating UI to new technologies. To achieve that, the migration should try to keep two aspects:

- the same **function** of migrated UI,
- certain **UI attributes** of migrated UI.

1.2 Overview

During a life-cycle of a commercial software system corima, developed by COPS GmbH, a requirement for a client in new technology came up. corima is a multi-user client-server

[1]Graphical user interface is being shortened to user interface (UI)

[2]ASP.NET is a framework designed for building enterprise-class server-based web applications using .NET on Windows.

[3]ASP.NET is Web forms is one of the ASP.NET web development models and it is the oldest one.

[4]ASP.NET MVC is is one of the ASP.NET web development models. ASP.NET MVC is a framework using Model View Controller (MVC) design pattern.

application and an application platform at once. A server and a client are both developed in .NET framework. Server side is developed in .NET WCF[5] technology, while the client is developed in .NET WPF[6] technology. New client for corima is demanded in ASP.NET technology. Since a business logic is mostly placed on a server-side, the new required client can fully reuse it. However, an idea of reusing client-side source code arose. The idea was to introduce a mechanism to convert UI from one .NET technology to another. To achieve the conversion, some new technology-independent representation of UI should be established. A representation should be independent from any existing language and should describe the UI well enough for most common usages in corima.

An approach would require to separate its function from a construction in a specific technology. Thus, instead of describing UI by explicitly referring to UI constructs of given technology (e.g., JavaScript Text-box), we should concentrate on describing its function (e.g., Text Input). The use of so-called declarative language seems to be a natural choice for describing the UI. Therefore, this thesis elaborates on how to represent User interface requirements using a higher-level description. It investigates the optimal structure of such a language, it shows implied restrictions (e.g., limited developer's freedom), and it prototypes the use of this language in corima.

Hence, the goal of this thesis is to introduce a declarative language for an independent UI description. The code in that language will be further used as an input for so-called generator. For purposes of corima, the generator will be .NET library generating[7] UI in the required .NET technology. Generator is the key factor of choosing how the final UI will look like. This way we will achieve the consistency between the different .NET technologies while having the same declarative description of the UI. Therefore we could generate an application to HTML and CSS for a ASP.NET web application and XAML files for a WPF application.

1.3 Structure of the Text

In Section 1.4, we will define the goals of this thesis. In Chapter 2 of this thesis, we will describe the general rules for user interface from which will be derived the requirements on the new language. This also includes the requirements from a point of usability and accessibility, which are very important especially for individuals with specific educational needs. Here we come from the domain of *computer therapy* design principles, that offers solutions to common issues in UI design. Further in Section 2.3, we will describe current languages that are used for the UI definition and can be taken as an inspiration for the new language, where will be taken the advantages of these languages for the new proposed language. In Section 3.2, we will revisit the goals and state the specific goals that should be achieved. In the Chapter 3, there will be analyzed the problem and proposed a high-level language. In the Chapter 4, we will describe how the language was implemented in .NET and how generator was constructed. In the Chapter 5, we will show the related work, possibilities how to generate UI from other existing languages, and why their direct

[5]The Windows Communication Foundation (WCF) is a framework for building service-oriented applications (SOA). Using WCF, a developer can expose endpoints from which data can be send between a server and a client.

[6]The Windows Presentation Foundation (WPF) is a framework for building user interactive Windows applications. WPF provides a consistent programming model that separates UI from business logic.

[7]Generator represents .NET generating library.

use in corima is not cumbersome. Finally, in Chapter 7 we will conclude the thesis in the conclusion and provide how the further integration to corima should look like.

1.4 Goals

The goal of this thesis is to propose a solution to reduce costs of migration of UI from one .NET technology to another:

1. Define characteristics of typical UI,
2. Propose a language to describe characteristics of UI,
3. Propose a mechanism to make the use of described characteristics,
4. Evaluate, how the mechanism and language can help to reduce costs of migration from one UI technology to another.

Chapter 2

State of The Art

To design a high-level language, we must introduce common principles of UI first. We cover these principles from the most general to the most specific ones. All important aspects of these discussed principles should influence the language design of language accordingly.

While we need to propose a design of language keeping some attributes and its function, we need to discuss possibilities how the separation of function and construction with its attributes can be solved. One of those approaches is studies in Section 2.1.

In Section 2.2.1, we cover common approaches to achieve usability and accessibility of each UI control. We follow with the groups of common UI controls and their common constructions, including the touch screen design principles for these constructions. Further we will study the computer therapy design principles with focus on UI. Finally, we derive requirements that should be considered in new language describing UI on a general level.

Next, in Section 2.2.2, we reveal specifics of UI principles within certain business domain[1] . There are described required attributes for finance domains that should be considered in new language. Finally for UI principles there will be described how user specific groups have impact of the UI itself and how it should be considered in language too.

New language is not needed to design from scratch, therefore we will study and describe the current approaches how user interfaces are defined. For each approach there will be finally concluded what we can benefit from it and what is not suitable for our purposes and the explanation of the reason.

The mechanism should be demonstrated on some complex UI. As an example of complex UI is UI performing so called CRUD operations among some memory unit. We will provide an explanation what these CRUD operations are and what are minimal requirements that an UI performing these CRUD operations should consist of.

2.1 Separating Function and Construction

The goal of this thesis is to reduce costs on migrating UI from one technology to another. One possibility to assess this problem is to clearly separate a function and a construction (F/C) of a system (i.e., UI), and to map F to C using a rigorous engineering way. This approach is grounded in findings of Enterprise Engineering (EE)[16]. Their applicability in Software Engineering has been studied by researchers at Faculty of Information Technology at Czech Technical University (FIT CTU) in Prague. Thus, in this thesis, we refer

[1]In this text, business domain represents all business specific activities such as finance, accounting, marketing, medicine, and research.

to a paper Affordance-driven Software Assembling (ADA)[18], which overviews the concepts of software architecture aiming at reducing costs of systems by clearly separating their function from their construction. The research [18] explains, that based on so-called τ-theory (Teleology Across Ontology) [15] and β-theory (Binding Essence to Technology under Architecture) [14], software system can be assembled from certain components. However, this approach expects that components expose their properties, and that we clearly describe users with their purposes on using the system. Furthermore, the approach expects a reasonable automated, or semi-automated mapping algorithm selecting convenient components. The Figure 2.1 demonstrates this approach.

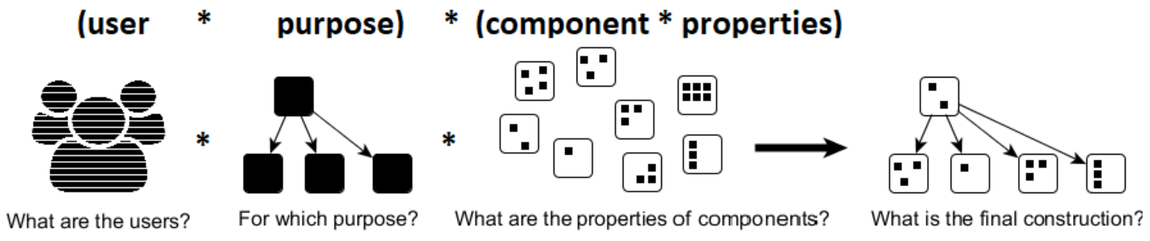

(user * purpose) * (component * properties)
What are the users?
For which purpose?
What are the properties of components?
What is the final construction?

Figure 2.1: Affordances in component-based systems [18]

Since in this thesis, we want to propose a solution of building up a system respecting limitations of challenged individuals, and we want to reduce costs on migrating systems from one technology to another, the research at FIT CTU is an important basis of our work.

2.2 UI Principles

A designer has to involve the creativity to outcome the interesting appearance of an UI. However, the UI would be worthless if it would not keep certain aspects that leads the design to be usable by the appropriate target group. No matter what kind of UI for any kind of application is designed, the UI should consists of common UI regions (called UI controls) for which all targets groups are used to. Further when a designer constructs the UI for a target group with some physical or mental disposition, a designer should take these dispositions into account and produce the design to be usable also for these target users, e.g., a designer designs a suitable sound control for blind people. All these general aspects (called principles) of a UI will be described in the further text with then focus on specific needs of mentally challenged people. We will also propose a common set of UI controls that might be considered by new language.

2.2.1 UI principles of current designs (General)

Usability and Accessibility

Generally, a user interface (UI) can be created individually according to a developer's attitude and a design feeling. A given user interface is usually not appropriate for everybody. A number of users can face troubles to understand it. Others are not able to use it at all. The UI is commonly judged by its user friendliness, or easiness to use. However, the right technical term expressing the quality of UI is known as a *usability*. *Usability* has many definitions. The ISO 9241 standard on *Ergonomics of Human System Interaction* (Part 11 1998) defines *usability* as [25]:

This part deals with the extent to which a product can be used by specified users to achieve specified goals with effectiveness (Task completion by users), efficiency (Task in time) and satisfaction (responded by user in term of experience) in a specified context of use (users, tasks, equipments and environments).

Jakob Nielsen [35] states:

Usability has multiple components and is traditionally associated with these five usability attributes: learn-ability, efficiency, memorability, errors, satisfaction.

Even though the designed system is not very usable, the common practice in companies is solved by introductory lessons or trainings to explain end-users how to deal with the new software. However, there can be also users that have some dispositions to be not able to work with presented software at all. This kind of people can be children that cannot read, seniors, somehow mentally challenged individuals, etc. Due to this facts, the designer should follow some rules during the creation of user interface to avoid the problem stated above. To address this problem, another essential term to study is *accessibility. Accessibility* is usually connected with the use of UI by people with disabilities and by the older people. ISO 9241 standard on *Ergonomics of Human System Interaction* (Part 171 2008b) defines accessibility as [26]:

The usability of a product, service, environment or facility by people with the widest range of capabilities.

By accessibility we also understand the physical ability to have “access” to the usage of a provided system. Therefore, the “accessibility” plays an important role in a system. It expresses a barrier between the system and its user.

Firstly, this text will be more focused on usability. That means that it will be more focused on general rules for well usable user interface. In further sub sections we will specify other rules for the touch screen design and mentally disabled people (accessibility).

From general point of view, the design should maximize the number of people who can:

- reach the controls (accessibility),
- find the individual controls or keys if they can’t see them (visibility),
- read the labels on the controls or keys (readability),
- physically operate controls and other input mechanisms (physically accessible),
- understand how to operate controls and other input mechanisms (intuitive),
- connect special alternative input devices,
- view the output display without triggering a seizure (compactness).

That means, there should be some standard how to create UI to have the best usability of demanded product. In the process of designing user interface, the UI is typically produced from a finite set of elements. To increase the usability, it is recommended to use well-known elements. The users are familiar with them, and they expect them to behave in a certain way. Thus, choosing this kind of elements seems to maximize the number of people capable to use them.

User interface elements are:

- input controls,
- navigational components,
- informational components,
- containers.

Input controls

Input controls allow user to interact with an application. Widely used input controls are check-boxes, radio buttons, drop-down lists, list boxes, buttons, toggles, text fields, date fields, and buttons.

Check-boxes allow the user to select one or more options from a set. It is usually best to present check-boxes in a vertical list. More than one column is acceptable as well, if the list is long enough that it might require scrolling or if comparison of terms might be necessary. See fig. 2.2 for check-boxes example.

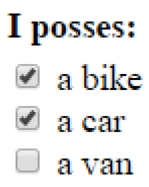


Figure 2.2: Check-boxes example

Radio buttons are used to allow users to select one item at a time.

Drop-down lists allow users to select one item from a set at a time, but are more suitable for large sets. The list is shown after clicking the drop-down list and user is able to scroll through a set and select one item.

A button indicated an action upon touch and is typically labeled using a text, an icon or both. See fig. 2.3 for a button example.

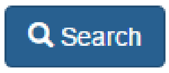


Figure 2.3: A button example

A drop-down button consists of a button that when clicked displays a drop-down list of mutually exclusive items. See fig. 2.4 for a button example.

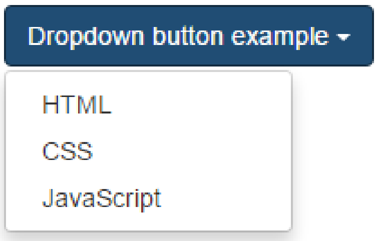


Figure 2.4: A drop-down button example

A toggle button allows the user to change a setting between two states. They are most effective when the on/off states are visually different. See fig. 2.5 for a button example.

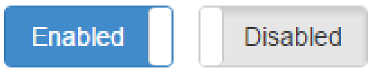


Figure 2.5: Toggle buttons example

Text fields allow users to enter text. It can allow either a single line or multiple lines of text. See fig. 2.6 for a button example.

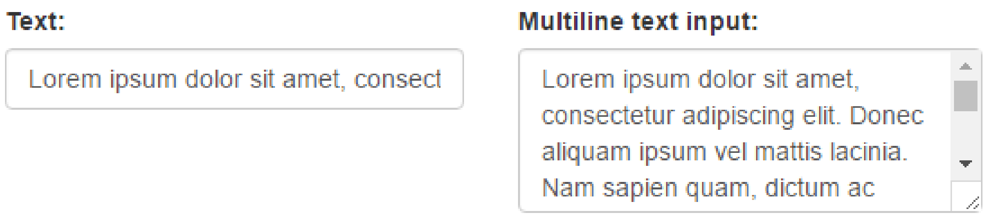

Text:
Lorem ipsum dolor sit amet, consect
Multiline text input:
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Donec aliquam ipsum vel mattis lacinia. Nam sapien quam, dictum ac

Figure 2.6: Text fields example

Navigational components

Navigational components are elements such as search fields, sliders, icons and pagination dividers. These controls allows user to navigate in the system and increase usability.

Search field allows users to enter a keyword or phrase and submit it to search the index with the intention of getting back the most relevant results. Typically search fields are single-line text boxes and are often accompanied by a search button. See fig. 2.7 for a button example.

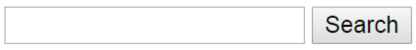


Figure 2.7: Search field example

A slider, also known as a track bar allows users to set or adjust a value. When the user changes the value, it does not change the format of the interface or other information on the screen. See fig. 2.8 for a button example.

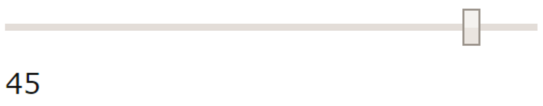


Figure 2.8: Slider example

An icon is a simplified image serving as an intuitive symbol that is used to help users to navigate the system.

Pagination divides content up between pages, and allows users to skip between pages or go in order through the content. See fig. 2.9 for a button example.

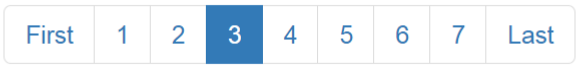


Figure 2.9: Pagination example

Informational components

Informational components are elements indicating additional information to the user for better user experience. The user can be for example informed if data is loading or if an error occurred. The informational components are: a progress bar, a tool-tip, a message box and a modal window.

A progress bar indicates where a user is as they advance through a series of steps in a process or it can indicate percentage done from the whole process. Process can be for example downloading of some file or shopping order. See fig. 2.10 for a button example.

Figure 2.10: Progress bar example

A tool-tip allows a user to see hints when they hover over an item indicating the name or purpose of the item. See fig. 2.11 for a button example.

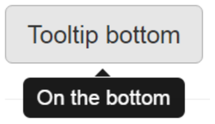
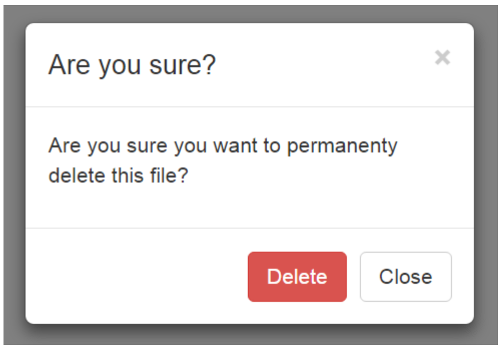


Figure 2.11: Tool-tip example

A **message box** or dialog box is a smaller window in window that provides information to users and requires performing an action.

A **modal window** is smaller window within window and requires users to interact with it in some way before they can return to the parent window. See fig. 2.12 for a button example.


Are you sure?

Are you sure you want to permanenty delete this file?

Figure 2.12: Modal window example

Containers

Containers are elements that contain any kind of information in an effective way. The only one commonly used container is an accordion.

An **accordion** is a vertically stacked list of items that utilizes show/hide functionality. When a label is clicked, it expands the section showing the content within. There can have one or more items showing at a time and may have default states that reveal one or more sections without the user clicking. See fig. 2.13 for a button example.

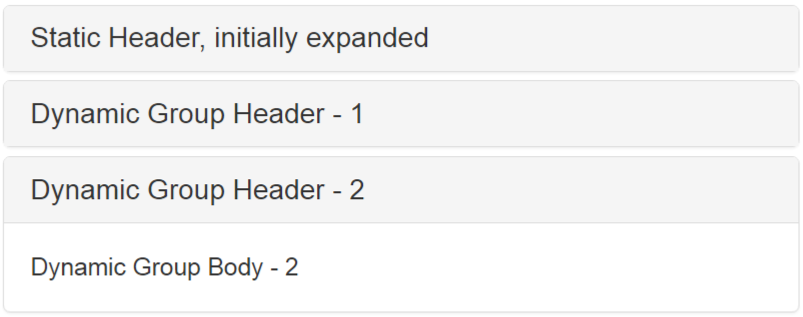


Figure 2.13: Accordion example

Touch screen design

When we focus on the touchscreen design, the well known user controls should be changed to satisfy users interacting with the system via their fingers and hands that interact with the system itself and therefore to be usable with human hands.

Nielsen creates a set of design patterns applicable for the construction of touchscreen based mobile design [36]. These problems are grouped in three main problem areas: (1) Utilizing screen space, (2) Interaction mechanisms and (3) Design at large. Furthermore, there are patterns based on experiments made for the purpose of usability heuristics for touchscreen-based mobile devices [24] [23]. The patterns use the same template like Nielsen used [36] with some modifications.

As the outcome, there are new patterns for mobile touch screen devices [24]:

- TMDP1.1 The thumb rule,
- TMDP1.2 The thumb rule #2,
- TMDP3 Explicit user control,
- TMDP4 Recognizable icons,
- TMDP5 Clean form fields,
- TMDP6 Shape of buttons.

The TMDP1.1 The thumb rule is defined as following:

Use When: Designing the interface. Placing main elements/options on the screen.
How: Place main elements within a range of a semicircle with a 2,7 inches' radius from the right-middle side of the screen.

Why: The average length of a human thumb is 2,7 inches. Considering that statistically, most of users hold the phone with their right hand and use their right thumb to interact with the device, main elements should be placed within user's reach...[24]

For purposes of this thesis there should be added another rule regarding the size of the finger. It means that user controls should have minimal size to be comfortably clicked on and to avoid clicking the elements that user doesn't want to click on. We will focus on this in following section too.

Designs for mentally challenged individuals

In-spite the fact the touch screen is much more usable than common PC for mentally challenged individuals, according to current studies, opinions and experiences, mentally challenged people can encounter some problems using touch screen devices [21]. The most common problems occur with buttons, menus, a text size, and with the touch screen devices itself. Buttons are often too small, it cannot be clicked on them when pressing too long and provides no action. Menus are constructed that there is a lot of options to select, most of them unnecessary or misleading. Text sizes are too small to read even with the corrective lenses.

The possible solution could be in keeping following set of requirements:

- Remove unneeded buttons (images, functions). Try to find negative factors on every button (images, functions). If the button (image, function) is here only due to aesthetic purpose, remove it,
- Input controls must be of a suitable size so that they can be easily pressed or even seen,
- Add voice output for available menus for blind people,
- Add alternative feedback to interface components such as scrollbars, drop down lists etc., when user interacts them,
- Add more information to the user, for example: when a system needs two clicks to perform some action, after first click it should inform user to click again on another component.

These set of rules can solve many problems in using UI by mentally challenged people, however, these studies do not cover all issues regarding usage of UI by mentally challenged people. These rules are too specific and only somehow decreases the set of problems that can occur. Due to this fact we need some study that can generally describe the solution, from which we can derive possible solution on at least the most common problems occurring in the study of mentally challenged.

Design principles: Computer Therapy (*i-CT*)

The project of the *computer therapy* represents new way of research in a field of information technology and in result it brings new attitude to therapy for mentally challenged people. The author of the project itself is Ing. Jiří Fiala [19]. In 2012 the project originated in directly in a mentally disabled care facility and is currently developed on a Faculty of

Information Technology Brno University of Technology with a support of Red Hat company [6], with the non-profit sector donated from ICT resources focused on education and therapy of mentally challenged people, and with support of experts from iSEN community [4] (SEN stands for Special Educational Needs).

The detail goal of the presented *computer therapy* is the application of suitable resources of information technologies for specification of uniformed standard for a hardware/software development. This standard should be easily used in practice, should be publicly accessible and due to its implementation a created software should be effectively usable by people with some mental disorder (special education and therapy). This should lead to therapy which has long term effect, decreases impacts of daily issues and compensates deficits, including help to decrease impact of mental disorder. The synthesis is made from several branches of study and the problematics is analyzed in its whole life cycle [19] [20] [21]. In the research of computer therapy there are proposed SW/HW design principles with the same names. These design principles with the usage of suitable methods from software engineering, e.g. MDE [37], leads to the improvement of a quality of development process directly on an end application. Final application (SW) is due to keeping the unified standard better usable on a current HW and accessible for the target group.

For purposes of this thesis the design principles of *computer therapy* can be summarized into several priorities [21]:

- **Specification of a goal:** Before a development of an application on a specific HW there should be clearly specified goal, eg. compensation of some dispositions of a person,
- **Safety:** Developers and an application should keep security and safety restrictions for a mentally challenged people. An application should support several different modes depending on the current user. Modes should be at least for a client, an assistant and a system administrator.
- **Open source:** In a development process there should be an emphasis on usage of open source technologies, together with publicly sharing the source codes. Source codes should be accessible for free to allow extending itself by another developers around the world. Also it should be free to use to be able to build with it another useful applications. A developer should add these extensions back to this application,
- **Cross-platform software:** An application should be available on several mobile platforms. The development of application should be handled as cross platform, that means to be compilable according to developer's need to all supported platforms,
- **Expandability:** An application should have clearly defined structure and should have been implemented in higher programming language with an usage of object oriented design. A development should be realized according to modern standards,
- **Configuration:** An application should take into account on individual needs of clients and support maximal adaptation to this needs. Moreover an application should support switching between configuration of several clients,
- **Usability:** Applications should not need any Internet access. Applications should be usable for its SEN purpose also including the usage outside a school environment. SEN usability gives more constrains than common usability, hence they are part of *i-CT* design principles,

- **Accessibility:** Applications should have low prices to be able to be offered to less wealthy people and non profit organizations. Further SEN accessibility also setup other constrains which are matter of proposed CT design principles.

These priorities also let to the design and development of a new framework. With this framework was also created several applications that satisfy these priorities. The new framework is called *Framework computer therapy* and is more deeply described in [27]. The therapy using presented IT resources are finally used as a permanent activity that can be offered regularly according to an individual plan. This attitude brings to the target group several possibilities of usage with new applications [19] [21]:

- **Serves as an educational tool:** Teaching of school subjects, reading, writing,
- **Compensation tool:** Becomes as a part of the person,
- **Development of intellectual abilities:** Handling the daily needs and activities of every person,
- **Free time activities:** Serves for relaxation and rest of a person, decreasing of a stress,
- **Connection with other therapies:** e.g. the usage for a music therapy.

The design principles of computer therapy for UI: usability and affordances

For the purposes and goals of this thesis we will be more focused on stated design principles of *computer therapy* from a view of priorities of usability and accessibility on a mobile touch screen platform that will be further described according to [21]. The stated standard of the *computer therapy* also states design principles with a focus on a user interface and *"human computer interaction"* that is also focused on this mobile touch screen devices. These principles are then called ***"principles of usability and affordances amplification"***. This is essential for the following design of the high-level language for the description of user interface, because it states the requirements and criteria for the design. These two principles will be described in the following subsection.

The principles of usability and affordances amplification

The principles of usability and affordances amplification are also related to general rules of Human Computer Interaction (HCI)[17], however they do not use some specific set of proposals that should be kept on specific UI component. It is due to its generality, which should be applicable to the most of UI components, which may differ in many attributes. Hence these principles tells the designer generally what should be done in the design of UI [21].

> ***Definition of problem of affordances:*** *Generally each control or other action element in user interface should suggest its usage (affordances). In the case of mentally challenged individuals, this rule should be multiplied (amplified) by the degree of intelligence deficiency or deficiency in perception abilities.*
> ***Solution for problem of affordances is following:*** *each element of user interface should be formed well enough (size, shape, color, sound response) to*

suggest its usage, even for the mentally disabled (e.g. using simplified and amplified principles of "Design of everyday things"). Furthermore, all gestures should be intuitive, simply based on common-known, real world gestures (real world of mentally challenged) [21].

The problem of usability is defined as following: *A similar situation occurs in the focus on user interface element's practical usability. A user interface element may be usable (touchable) for an intellectually capable individual, but not usable (touchable) for a mentally challenged individual with worse perception or deficiency in soft motoric functions.*
The solution for the problem of usability can be following: *Size of elements should be large enough to avoid "thick finger effect" and distances among elements should allow freer place to avoid of multiple action-button touches [21].*

The same holds for other UI component's attributes, parameters which can be matter of invisibility, unreadability, inaccessibility and unintuitivity. For the usage of this thesis we can list a set of following rules, which are also part of proposed CT design principles [21] and that should be kept for a usage of designing UI for mentally challenged people:

- shape of an element should have rounded edges,
- color of an element should be different different than its background or in the case of the same colors these colors should have enough contrast,
- text inside an element should have enough space from the edge of an element and should be centered inside an element,
- an element should have some minimal size,
- distances between elements should be given from the sizes of two elements together,
- elements should be equally positioned on the space of user interface (to avoid creating chunks).

Requirements on languages for GUI and design

If we compare the present possibilities and *computer therapy*, we can figure out that computer therapy solves problems more generally to cover all possible problems. E.g., rules like *"Remove not needed buttons"* or *"Input controls must be of a suitable size so that they can be easily pressed or even seen"* are covered in ***"principles of usability and affordances amplification"***. Due to that fact, we will use *computer therapy* as a domain from which we will propose set of requirements that should be described by following designed language for UI definition. From *computer therapy*, it can be seen that more design principles are concerned about the content of UI description with proper style - construction of each element and construction among them. However, it also proposes some semantics meaning which is part of affordances amplification principle (e.g., for certain SEN purpose there should be used proper UI component, which semantics is the closest to our purpose). In further part of this thesis we will focus more on construction rules covered in *i-CT* design principle. Construction has its functional purpose. E.g., language should be able to somehow describe the shape of the elements that will be used for the definition of UI. The analysis will be more discussed in Chapter 3 where we will describe the general solution for all possible descriptions of UI by new language, not only focused on mentally challenged people, but also for other possible target groups that can occur in the common life situations.

2.2.2 UI principles of current designs (Business-Domain)

When we come to the Business-Domains, such as Finance or medicine, UI is needed to fulfill all the Domain requirements. Domain requirement is such an attribute specific for a certain domain. For purposes of the thesis there will be studied requirements in the Finance domain. In the Finance domain sector there, are typical attributes that should be kept to keep consistence between Finance applications:

- decimal separator,
- thousand separator,
- currency format,
- negative pattern,
- date-time formats,
- number precision.

Decimal separator is the character used as the decimal separator. For instance, Great Britain and the United States are two of the few places in the world that use a period (.) to indicate the decimal place. Many other countries use a comma (,) instead. The decimal separator is also called the radix character.

Likewise, **thousand separator** is the character used to separate groups of thousands. In the U.K. and U.S. use a comma (,) to separate groups of thousands, many other countries use a period (.) instead, and some countries separate thousands groups with a thin space.

Currency format is the way of expressing monetary units. There are three possibilities how to express monetary units:

1. The currency sign. The currency sign is primarily used for graphic purposes. Alternatively, its use is also permitted in promotional publications (e.g. sales catalogs). No space after the sign. E.g. €35.
2. The ISO code. ISO code for defining currency is ISO 4217. ISO 4217 is international standard for marking the currencies as 3 character codes. These codes are defined by International Organization for Standardization (ISO). E.g. 30 EUR
3. The written name. Used when a monetary unit is referred to generally however an amount is not included. E.g. an amount in euros

Negative pattern is the way how to distinguish positive and negative numbers. There could two ways:

- minus sign before the value,
- different graphical representation of the values. For instance, positive values can be displayed in green color otherwise negative values with red color.

Date-time formats are formats that represents date and time values in Finance applications. These must follow ISO 8601, the International Standard for the representation of dates and times. ISO 8601 describes a large number of date/time formats. To reduce the scope for error and the complexity of software, it is useful to restrict the supported formats to a small number. This profile defines a few date/time formats, likely to satisfy most requirements.

Number precision is important attribute in Finance sector. In finance sector often occurs situation when user edits thousands or even millions. Then is is suitable to offer customer possibility to enter these thousands (millions) as if it would be single units.

2.2.3 UI principles of current designs (User groups)

In previous sections is described impact of user onto the UI. Derived from these facts, specific groups of users demands the specific requirements for the UI. Users differ with respect to, for instance, their preferences, capabilities, speaking different languages and level of experience. E.g., young, middle age, old people or even mentally challenged individuals or people having some physical disorder. This heterogeneity of end users should be considered in the proposed language.

2.3 Approaches to describe UI

In present, a huge set of technology-specific approaches for defining (development of) UI exists. So far, many variations of programming/markup languages (e.g., C#, C++, Java, HTML) with different widget libraries (e.g., WPF, Qt, Swing) has been developed. Automatic conversion between these technologies is not solved problem. Reasons why automatic conversion fails is mainly based on the complexity of developed UI, e.g., dependencies of form fields, validations, connection to back-end logic and technical differences between languages. Therefore instead of technology-specific approaches is needed a **higher-level description language**. This language should contain concrete descriptions, from which could be obtained concrete technology-specific implementations of user interface through associated conversions.

2.3.1 Languages for UI definition

SW methodology (MDE, OMG)

Model Driven Engineering (MDE) is a software methodology that has a goal to define software specification with the highest amount of *abstraction* and also raise amount of *automation* in an software development. MDE focuses on creating and exploiting conceptual models at different levels of *abstraction* that can be used for the description of every possible problem. Hence it increases the level of abstraction in specification of a software. Also with the usage of executable *model transformations* raise the *automation* in software development. *Model transformations* in practice means the transformation of high-level models to the lower models until the models itself are executable. For further purposes of this thesis it can be used as a template to be interpreted as UI by a specific technology, e.g. HTML interpreted by a web browser. These high-level models are represented in some model notation or language. Such language is then called *Domain Specific Language* (DSL)[33] due to the fact that model is connected with a certain domain. The representation of DSL then can be textual or visual. More information regarding DSL can be found in [22]. DSL specifies the model called *Domain Specific Model* (DSM) [28]. Finally the whole application can be thus specified by several DSMs that are specified in different DSLs. The term often related to MDE is *Model Driven Architecture (MDA)*. MDA was introduced by *Object Management Group (OMG)* and can be seen as OMG's vision on MDE. For purposes of this thesis the language for description of graphic user interface is the specific case of modeling language for a creation of a model that is usable for development according to MDE.

SBVR

Semantics of Business Vocabulary and Business Rules (SBVR) [7] is the meta-model for development of semantic models of business rules and business vocabularies. Presented rules are described in common language, however some rules are presented graphically in proper cases. Therefore SBVR offers language that describes a structure of rules that is written in a language that business people commonly use (need to point out that opposite way is more often in companies). This usable language SBVR calls *"semantic formulation"* that is not expressions or statements. *Semantic formulations* are structures which create meaning. There exists a vocabulary in SBVR that describes these meanings. In SBVR, the meaning of a sentence is communicated as facts about the semantic formulation. In formal language it means a restatement of the meaning [9]. We will describe *semantic formulation* on the following simple business rule. The rule is stated several times with the same meaning. We should also note that there can be other possible interpretations of these rules [9]:

A barred driver must not be a driver of a rental.
It is prohibited that a barred driver be a driver of a rental.
It is obligatory that no barred driver is a driver of a rental.

Description of semantic formulation of the business rule above in terms of the SBVR [9]:

The rule is meant by an obligation claim.
That obligation claim embeds a logical negation.
The negand of the logical negation is an existential quantification.
The existential quantification introduces a first variable.
The first variable ranges over the concept 'barred driver'.
The existential quantification scopes over a second existential quantification.
The atomic formulation is based on the fact type 'rental has driver'.
The atomic formulation has a role binding.
The role binding is of the fact type role 'rental' of the fact type.
The role binding binds to the second variable.
The atomic formulation has a second role binding. The second role binding is of the fact type role 'driver' of the fact type.
The second role binding binds to the first variable.

As we can see, SBVR is not used to provide a clear and short description like formal language, however, SBVR is used to provide detailed description about meaning. The description is then divided into sentences where each sentence represents a fact about the rule.

OCL

A class diagram from UML [2] is generally not able to specify all kind of information to the model. The problem occurs when we need to describe additional constraints about the objects in the model. These constrains are then directly written into model in natural language. However common practice has shown that this always leads to misinterpretation. Hence some formal languages were developed to describe these constrains in a clear formal way. One of these formal languages is called *Object Constraint Language (OCL)*. OCL [3] is formal language used for description of *expressions* and *constrains* on UML models. Therefore the OCL is a language for description of *expressions* on object-oriented models. *Expressions* then specify rules that must be kept for the modeled system or specify conditions to be hold for queries over objects in a model. These expressions also enables to set operations that can manage a change of the state of the system. Even though it is formal language it can be easily read and write due to the fact that it was designed as a business modeling language. When an expression is evaluated, it just returns a value. OCL cannot be used as standard programming language, because it cannot generate the executable code. Even though it is not programming language it is typed language, e.g. user cannot compare a String with an Integer. Each Classifier defined within a UML model represents a distinct OCL type. The example of class diagram with OCL can be seen on fig. 2.14. In addition, OCL includes a set of supplementary predefined types [3].

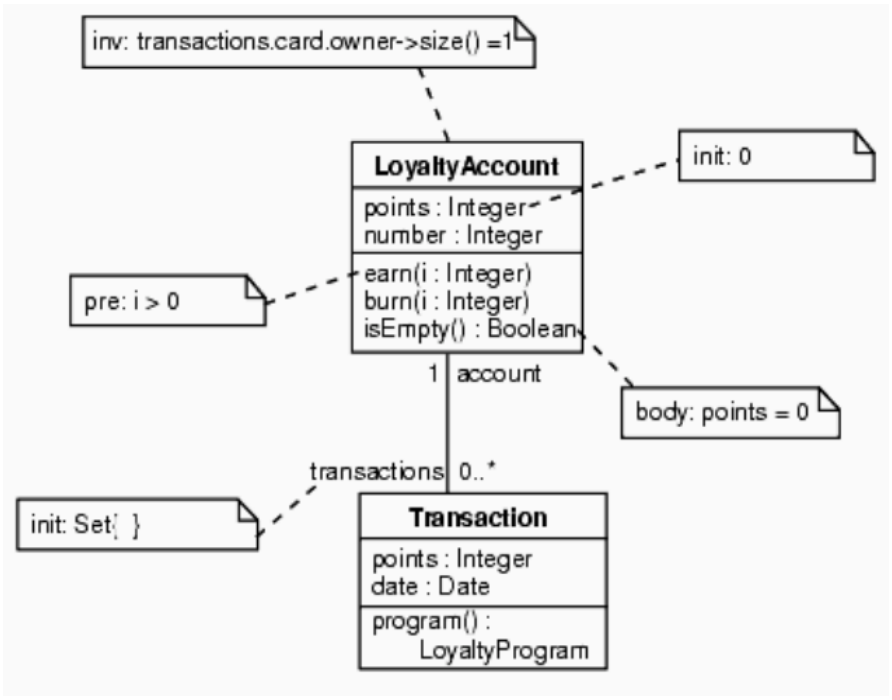


Figure 2.14: Example of class diagram with OCL

From the OCL specification we can list a set of purposes where to use OCL:

- as a query language,
- to specify invariants on classes and types in the class model,

[2]Stands for Unified Modeling Language.

- to specify type invariant for Stereotypes,
- to describe pre- and post conditions on Operations and Methods,
- to describe Guards,
- to specify target (sets) for messages and actions,
- to specify constraints on operations, and
- to specify derivation rules for attributes for any expression over a UML model.

Visual programming languages

In a recent past, for a usage of computer was necessary to educate people. Progress in information technologies brought the user graphical user interface that should be usable without learning. Moreover there were developed applications that enables people to publish their content on the Internet or Social networks without programming. However, when a person wants to develop his own software, there still exists a “barrier” due to programming language. Fortunately there were developed tools that offer user user-friendly interface for development of software. These tools are called *Visual programming languages (VPL)*. These languages are platforms that typically provides user a set of visual graphic elements, like diagrams, free-hand sketches, icons, or demonstrations of actions performed by graphical objects, from which with a support drag-drop interface can be created output software. These languages also abstract a way of other functionality like functions or conditions that must be hold in an application. Graphic elements typically serves as input and connections between them serves as the output of the application. Then run of a program is started on a start element where is given an input and then are given outputs to the other elements by its connections and continues in this order till the program reaches last elements. An example of VSL can be seen on fig. 2.15 (RapidMiner studio).

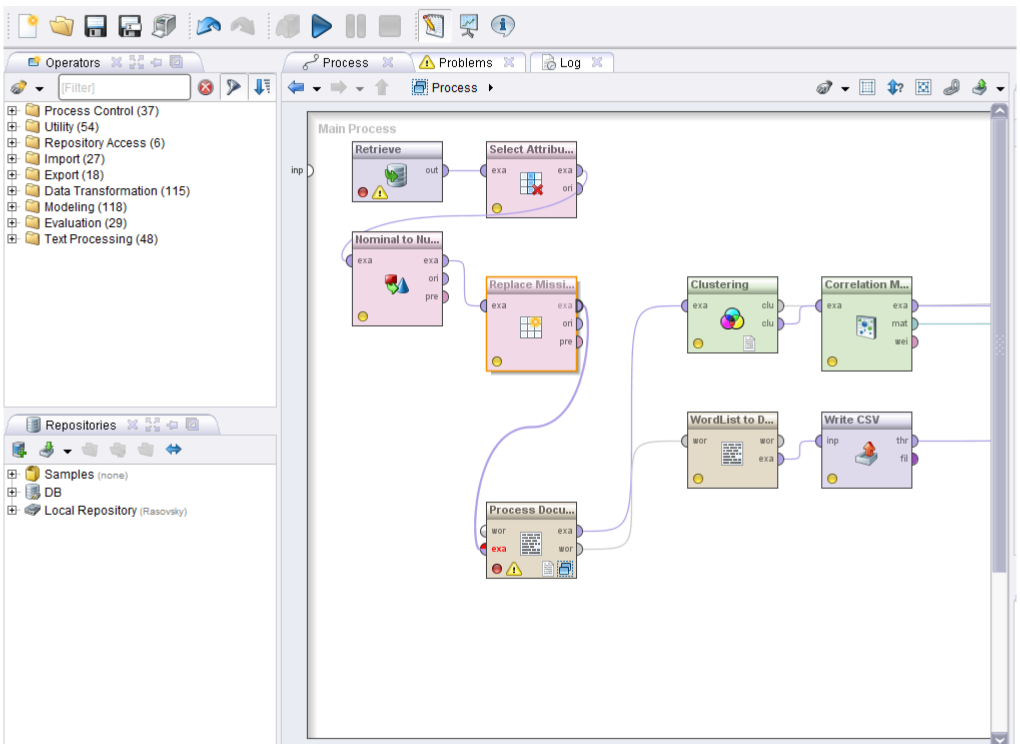


Figure 2.15: Example of Visual programming language

For the purposes of this thesis it is essential that these GUI elements are often described in some meta language that holds information about elements, their positions connections and settings. The following description of GUI 2.1 represents a shortened source code for fig. 2.15. As we can see, the source code is defined in XML [3] which is the most common language used for the description of user interface. In the next sections we will study further other languages that are based on XML and describe the user interface.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<process version="5.3.000">
  <context>
    <input/>
    <output/>
    <macros/>
  </context>
  <operator activated="true" class="process"
   compatibility="5.3.000"
   expanded="true" name="Process">
      <operator activated="true" class="k_means"
      compatibility="5.3.000" expanded="true"
      height="76" name="Clustering" width="90" x="447" y="165">
        <description>For clustering of text data</description>
        <parameter key="k" value="7"/>
        ...
      </operator>
      <operator activated="true" class="correlation_matrix"
```

[3]stands for Extensible Markup Language [2]

```
      compatibility="5.3.000"
      expanded="true" height="94" name="Correlation Matrix"
      width="90" x="581" y="165">
        <parameter key="create_weights" value="false"/>
        <parameter key="normalize_weights" value="true"/>
        <parameter key="squared_correlation" value="false"/>
      </operator>
      <operator activated="true" class="text:wordlist_to_data"
      compatibility="5.3.002"
       expanded="true" height="76" name="WordList to Data"
       width="90" x="447" y="300"/>
      <operator activated="true" class="write_csv"
      compatibility="5.3.000" expanded="true"
       height="76" name="Write CSV" width="90" x="581" y="300">
        <parameter key="csv_file"
        value="PATH/fracking-example-stemming-wordlist.csv"/>
        <parameter key="column_separator" value=","/>
        <parameter key="encoding" value="SYSTEM"/>
      </operator>
      ...
    </process>
  </operator>
</process>
```

Listing 2.1: An source file for Visual programming language

XUL

XUL (XML User Interface Language) is XML based language that is used to write applications. It is markup language implemented as XML dialect. The user interface design is defined generally as three sets of files:

- XUL files serving as content files that defines the user interface, eq. lists elements that are in applications and labels,
- the second type of files contains the other information about the design of elements in a form of CSS files and images, and,
- files containing localization strings.

A simple login prompt on fig. 2.16 has the following source Listing 2.2.

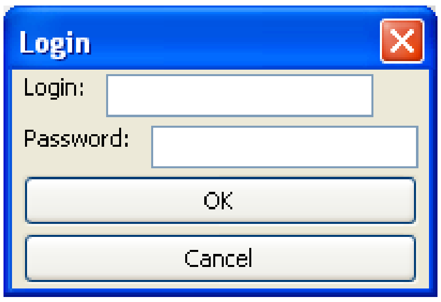


Figure 2.16: A simple login prompt in XUL

```
<vbox>
  <hbox>
    <label control="login" value="Login:"/>
    <textbox id="login"/>
  </hbox>
  <hbox>
    <label control="pass" value="Password:"/>
    <textbox id="pass"/>
  </hbox>
  <button id="ok" label="OK"/>
  <button id="cancel" label="Cancel"/>
</vbox>
```

Listing 2.2: XUL source file for the simple login prompt in XUL

Conclusion on current languages of UI definition

For a specification of high-level language for UI definition there is needed to specify the term high-level. High-level for this thesis means specification of a user interface by means of its functionality, e.g. language that describes a text box on a page is not high-level. high-level language description is for example that on a page is some element that can receive text input or generally an input of information of some type. Therefore as we have mentioned in section SW methodology it should be specific case of modeling language according to MDE. A language that is generally describing a UI and after a set of some specific steps generates exact UI in specific markup language, e.g. HTML. To conclude studied languages, SBVR is used for business modeling that is generally used for generating semantic formulation of the business rule and is not able to generate UI. OCL is able to add information about objects, however it is not good path for defining UI of the objects. On the other hand, we can benefit from the advantages of SBVR and OCL. SBVR show us how to describe a rule in a semantic sentence, that can be used in our language, because we need a way how to define the UI abstractly. OCL has it strong advantages in describing rules among objects and we can similarly use it for specifying these logical constrains on our UI elements. To go on to the visual programming, we can see that visual programming languages uses XML representations of processes that exactly describes the processes without an abstraction. The same stands for the markup languages like XUL or XAML. Therefore none of studied languages are valid for a high-level UI definition as we defined above. In the following chapter we will describe the requirements on this language and the design of the new high-level language will be proposed.

2.4 CRUD operations

In a field of computer programming, CRUD stands for create, read, update, and delete. Each of these words represents an operation typically performed on some row in relational database table. In general, these operations can be performed not only on relational databases, but also performed on any kind of persistent storage.

An acronym CRUD is also often used to describe UI conventions that enables to view, manage and destroy some persistent unit, e.g., form editing users data. The acronym CRUD is probably firstly popularized in book Managing the Data-base Environment [32]. As a minimum requirements for a user interface described as CRUD, the UI must allow to:

- Create new entries,
- View existing entries,
- Manage existing entries,
- Delete or invalidate existing entries.

For purpose of this thesis we will use the CRUD as the goal to show the further proposed mechanism will be able to generate CRUD forms.

Chapter 3

Analysis and design of new language for UI definition

In this chapter, we will analyze the requirements for the language derived from the Chapter 2. According to the requirements we will provide revisited goals that has to be accomplished within the thesis. Next we will propose a design of high-level language for description of UI. At the end of the chapter we will describe the algorithm of composition of UI components designed, where the algorithm is essential for further implementation of the generator tool.

3.1 Analysis of requirements for a new language

New proposed language is considered to be in a high-level form. High-level form can be considered as the first main requirement derived from the Section 2.3. From approaches in Section 2.3 we derived that we cannot use any of the current technology-specific languages for description of UI. Presented high-level form can be explained as an abstract language describing the UI independent from specific platform and technology. Based on MDE we require to propose a DSL having specific requirements based on discussed knowledge covered in Chapter 2. Now, we will go through the analyzed requirements from Chapter 2.

3.1.1 Independence of the user interface technology

The language should be introduced to keep independence from any existing user interface technology. This requirement is already covered by the specification of a language as a DSL. A DSL can have a textual or visual form. We will create a language as a domain diagram and describe its semantic meaning with words. Therefore high-level form strongly connects to this requirement.

3.1.2 Reduction of a cost within migration between UI technologies

To reduce cost within migration from one UI technology to another, a DSL should consist of the specification of technologies. A designed UI in new high-level language (DSL) will be possible to generate in several technologies of user interface. In next requirements, we will describe a attributes that will help to keep a consistence between the UIs generated in different technologies.

3.1.3 Separation of the function and construction

From Section 2.1, a DSL should consists of certain components having its properties and users with their purposes to the UI (system). For a DSL, it is important to define an UI with user only by purpose (function). Therefore this approach achieves the separation of the construction from the function part.

Next requirement is to propose some automatic or semi-automatic mapping algorithm selecting convenient components.

3.1.4 Attributes of UI controls (Usability, accessibility, HCI requirements and user groups)

From HCI and according to the definition of usability, we should be able to define a set of commonly used UI elements and controls in a DSL. Next requirement is responsibility (compactness) of UI design. Responsive design should respond to a different sizes of a screen. On a different devices the controls should be replaced or resized in a way that user is still able to interact with a system. Next there are other rules from HCI a DSL should consider like accessibility, visibility, readability, physically accessible ,and intuitive.

According to *computer therapy* project the UI for mentally challenged people must have specific construction of elements, like color, shape, and distances between elements must be in proper relations.

The language for UI definition should have some phrases for definition that this UI design should be touchscreen or not. If so, the rules defined in 2 should follow. There the positions of elements, recognizable icons, shape of buttons etc., has its function. That means here already came the first answer concerning the construction and its function.

Furthermore proposed language has to enable designer to describe UI for specific group of people. The set of computer therapy requirements is too small because a DSL should enable to describe UI usable by children.

To sum up, there is very big set of attributes that can describe a control or UI. Probably can occur situation when during the time there will come another attribute and the designed DSL should be extended. Moreover there must be a way how to describe some general attributes like if the UI is designed for finance domain. Therefore all these discussed possible attributes of a UI control should be generalized to general attributes in a DSL.

3.1.5 Business-domain requirements

The proposed language should contain an option to define all finance specific attributes, such as decimal separator character or currency format. This attributes will accordingly affect the construction of generated elements affecting only the business domain.

3.1.6 Connection to back-end logic

A designed DSL should consist of some mechanism how to provide an option to connect a generated UI to the back-end logic. This requirement is not studied explicitly in the 2 but is needed for corima.

3.2 Goals Revisited

In Chapter 2, we described all the important facts about the UI generation. We must state the exact goals that are revisited according to gained knowledge.

Now, it is clear we need to create high-level language in a form of DSL that will describe instances of abstract user interface. The steps to this approach will be:

1. **Goal 0:** Create a meta-model of a high-level language for describing UI including general attributes of the UI components. These attributes should consists of functional purpose of UI element, business-domain attributes, user group attributes, and any other kind of attributes,
2. **Goal 1:** Clarify the algorithm of composition of components according to its characteristics,
3. **Goal 2:** Propose a proof-of-concept of language implementation in .NET including class diagram of a proposed system,
4. **Goal 3:** Propose an implementation of components necessary for CRUD operations,
5. **Goal 4:** Propose an implementation of mapping logic of algorithm of composition.

The first two points will be described in this chapter. The next points will be described in Chapter 4, where technical details will be explained.

3.3 Proper design of UI language

Proposed language was designed as meta-model independent from any platform and technology. The reason why it was designed by a meta-modeling is because this meta-model can be then represented in any form, e.g., XML. This meta-model will be further taken and used to create domain specific language for corima purpose in .NET. Furthermore, it is designed for possibility to use this meta-model in any other DSL and in any specific technology. The scheme of the meta-model is depicted in Fig. 3.1.

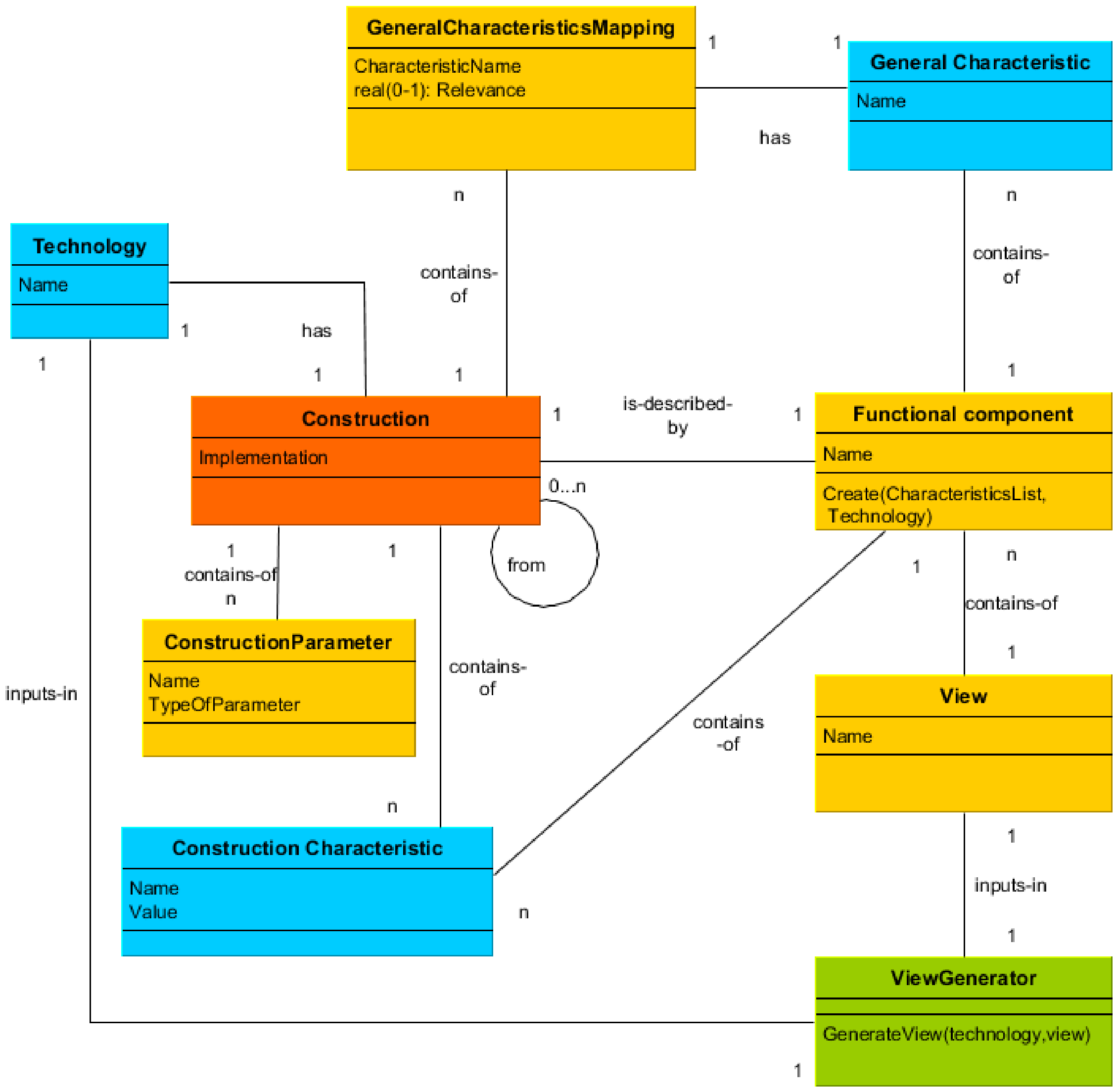


Figure 3.1: Meta model of high-level language for UI description.

As we can see in Figure 3.1, the proposed language consists of the following elements:

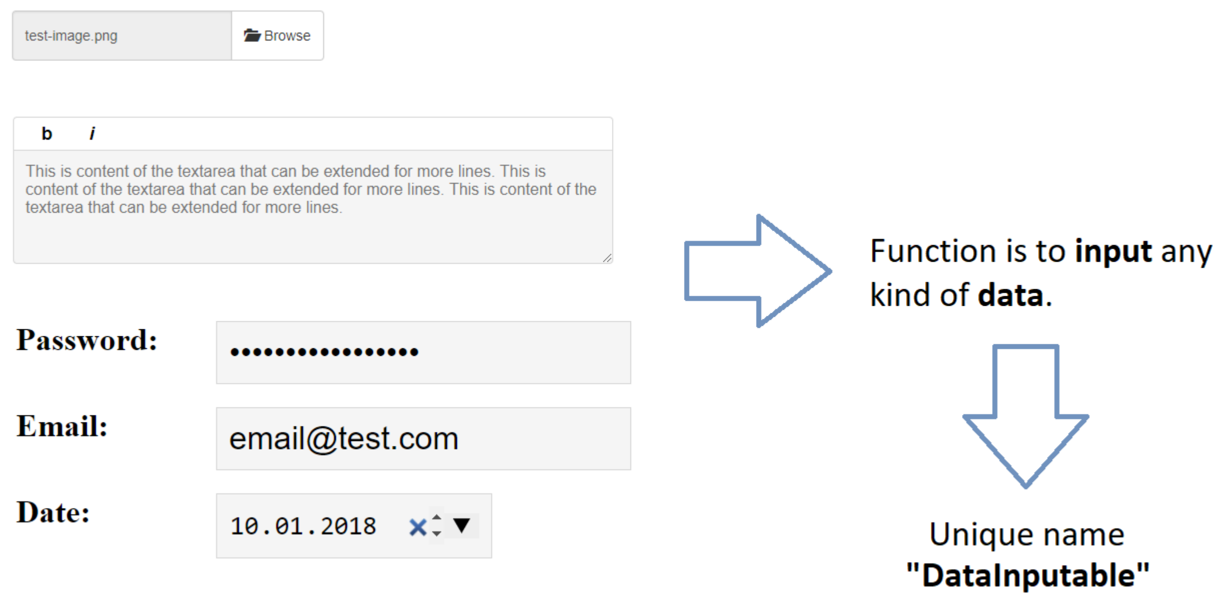
- Technology,
- Functional component,
- View,
- Construction,
- General characteristic,
- Construction characteristic,
- General characteristics mapping,
- ViewGenerator (or just a generator).

Each of listed elements has its own semantic meaning important to be well described to fully understand the language. Now, we will provide a description for each of the elements.

ViewGenerator presented in the scheme is not a part of a language. ViewGenerator represents some kind of a UI generating library. Immediate inputs to this library are **view** and **technology**. View defines UI together with context parameters, and technology specifies in which technology is UI finally generated. We can see the ViewGenerator do not have available information regarding the exact constructions that should be rendered. The ViewGenerator have only the all set of functional components having some characteristics that will be described further and according to them, it has to choose which one of the constructions is the most valid for the UI. This determination of the most valid constructions is the most important and key role in the thesis. All decisive logic and UI rendering is done by this tool and its algorithm will be described at the end of this chapter.

View as can be seen in Fig. 3.1 is abstract representation of UI consisting of a set of functional components. This view can order these components or wrap functional components to bigger groups with so called grouping functional components used for grouping of components. Affecting the view can be then managed through other functional components created in the DSL. View is not capable of specifying the exact look of the UI. Each view should have its unique name describing the purpose of the UI page. For example, view for displaying user data in grid should be called *"UserDataGridView"*.

Functional component is an abstract representation of some construction having specific function. We can derive the function of the component from the **problem of affordances**. All UI components are having some purpose and therefore we can derive from that its function. The wrappers of UI component are having wrapping function, text-boxes inputing function, tables filtering purpose, pagers paging purpose and so on. We can find the function name easily from its use. Good to point out, this function has to be named by name that not affect construction of any component because it would be recipe also for construction of this component. For instance, component with function for inputting any kind of data can have infinite set of real constructions. E.g., secured text-box, text-area for longer texts, image upload component, and responsive text-box. Thats why we can simply abstract the UI with this set of UI functional components. To specify further rendering attributes, we have to introduce new concept – **characteristics**. These characteristics then should affect the rendering logic. Each functional component will then have according to its function its unique name similar to the view. E.g., the component with function for inputting any kind of data will have unique name like *"Imputtable"*. An example of derivation of function and therefore the functional component is depicted on Figure 3.2.



test-image.png Browse

b i

This is content of the textarea that can be extended for more lines. This is content of the textarea that can be extended for more lines. This is content of the textarea that can be extended for more lines.

Password: ••••••••••••••••

Email: email@test.com

Date: 10.01.2018

Function is to **input** any kind of **data**.
Unique name **"DataInputable"**

Figure 3.2: Example of functional component and its naming. Graphic content was designed by the author in HTML and CSS.

Characteristics are abstract self explaining additional declarative information added to the functional component and **construction**. There are designed two types of Characteristics:

- **General characteristic**. General characteristics are attributes concerning any type of additional information extending the functional component/construction. These characteristics are designed to add typically information regarding user groups. On the other hand, they may contain also any other information. These general characteristics should describe whether UI is usable (principle of usability), accessible (principle of accessibility) or whether the UI is valid for mentally challenged people. These general characteristics should be given a valid self-explaining name that explains the semantic of an characteristic. For instance functional component/construction having the characteristic called *"ImageUploadable"* would suggest the functional component/construction should be somehow capable of uploading images. See the fact the language does not advise any further steps or parameters how the final construction should look like. Therefore general characteristics can be also described as static semantic information of any kind. The purpose of this general characteristics is to create a very abstract language unlimited of a set of available attributes the UI can consist of. Designer then can propose his own characteristics that suits for him. Therefore designer can create unlimited number of general characteristics and describe with them the functional component/construction that best suites for the construction. This is next key extension of the language allowing the language to be in high-level form. An illustration of derivation of general characteristic is depicted on Figure 3.3,

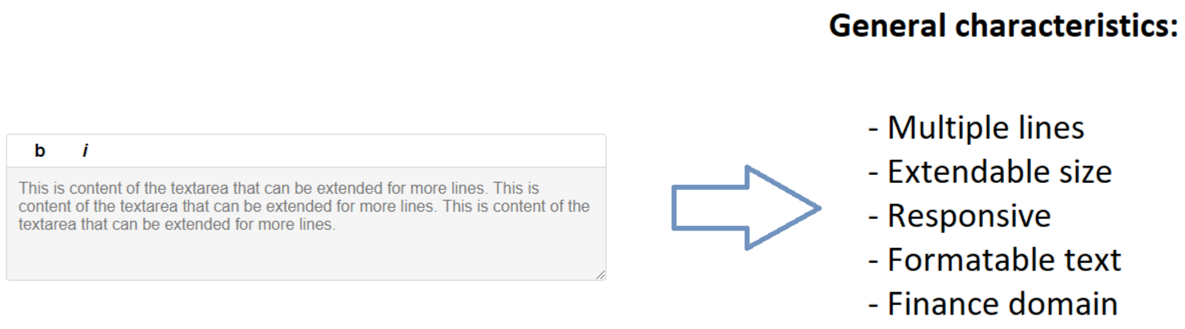

General characteristics:
b
i
This is content of the textarea that can be extended for more lines. This is content of the textarea that can be extended for more lines. This is content of the textarea that can be extended for more lines.
- Multiple lines
- Extendable size
- Responsive
- Formatable text
- Finance domain

Figure 3.3: Illustration example of derivation of general characteristics from a simple textarea. Graphic content was designed by the author in HTML and CSS.

- **Construction characteristics** are attributes having purpose to extend the functional component/construction with some parametrized information. Furthermore its semantic is to extend just its construction. That is why they are called **construction** characteristics. These information extends the final generated construction with context model. These construction characteristics has been designed e.g. to add labels to the UI components, add unique identifiers to UI components or define thousand separator character. Moreover, with these construction specific characteristic is possible to generate UI with connection to back-end logic and therefore allows the designer to create complex UIs. Construction characteristics advise how exactly UI control should look like. For instance text-box having construction characteristic describing label name as *"User name"* may be rendered as a text-box having appropriate label. Again it does not mean a label must be rendered there. This is just an advise for generator that should suggest its use. Illustration how construction characteristics affects the rendering of some functional component is depicted on Figure 3.4.

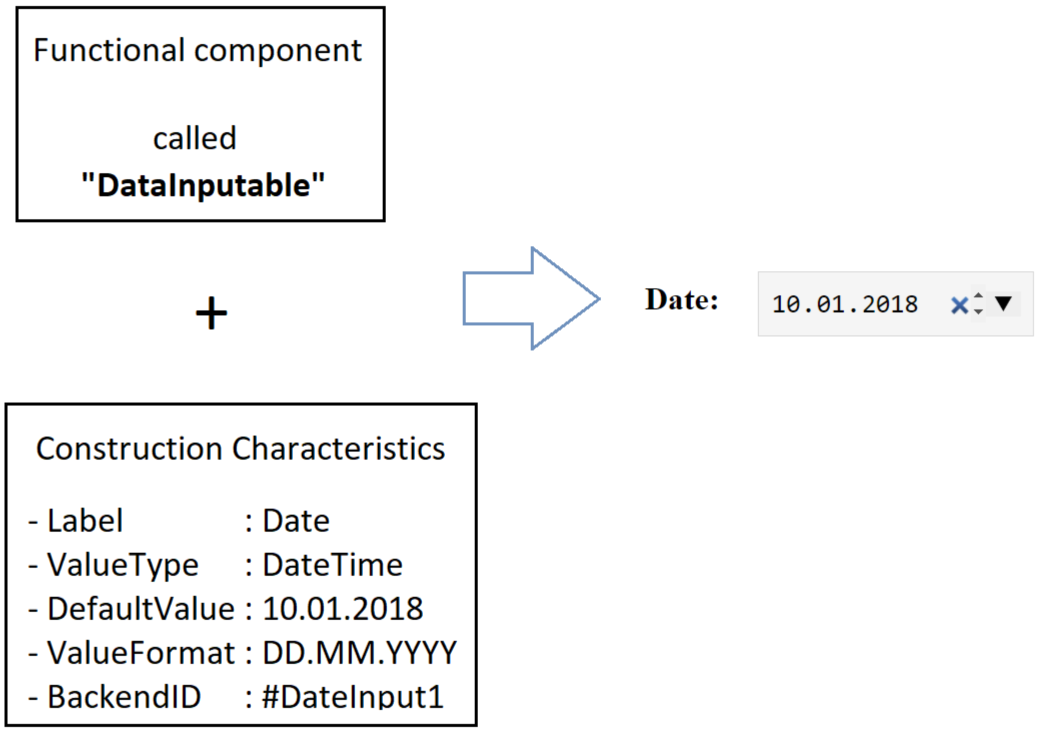

Functional component
called
"DataInputable"
+
Date:
10.01.2018
Construction Characteristics
- Label : Date
- ValueType : DateTime
- DefaultValue : 10.01.2018
- ValueFormat : DD.MM.YYYY
- BackendID : #DateInput1

Figure 3.4: Illustration how construction characteristics affects the functional component during the rendering. Graphic content was designed by the author in HTML and CSS.

Construction presented in the scheme represents a wrapper containing real construction of an UI component implemented in certain UI technology. According to exact implementation of construction are further defined its technology, function and characteristics[1]. This definition of technology, function and characteristics will be called mapping in this paper. This mapping of technologies, functions and characteristics is done by developer. Only developer knows in which specific technology is the UI component implemented. Technology could be derived automatically, however function definitely cannot be derived so easily. Even human could have sometimes problems with determining purpose of some UI elements without further clicking and interaction. Thats why according to terms like usability and affordances only developer who designed and developed the specific UI component should be capable of defining its function. The same claim is valid for the characteristics. Even more, characteristics should be defined as precise as possible. Again, only the developer who implemented the UI component can derive this set of general characteristics. For instance, the UI component may have some attributes hidden in the source code such as mandatory field in a form.

Since the **functional component** and **construction** is explained, it is needed to describe the mapping of characteristics to these components. When a construction is implemented, there must exist some of platform independent set of general characteristics and construction characteristics defined in the language. These characteristics are defined by

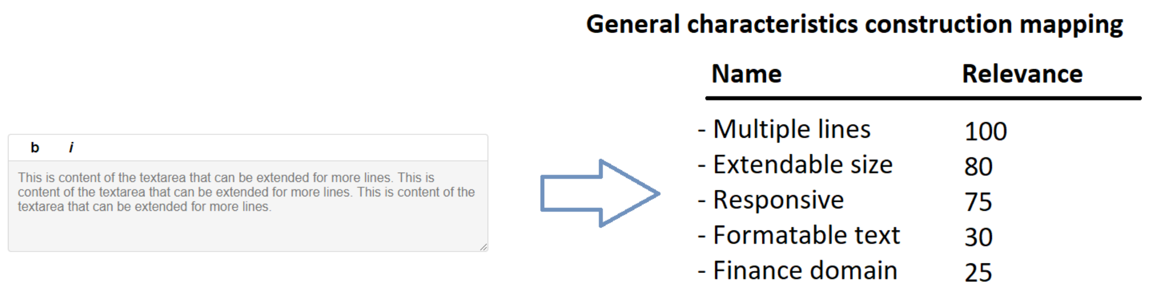
[1]Characteristics are meant generally both general and construction characteristics

developer explicitly. Each characteristic is having its unique name in the language and its semantic function. This semantic function has to be described with every new characteristic to keep consistence between UIs. When these characteristics are defined they serves as some kind of a dictionary. The set of technologies is similar to the characteristics. It is defined set of technologies with given unique name. Each technology represents unique UI technology where no duplicates are available.

Mapping of construction and general characteristics to functional component is depicted in Fig. 3.1. Firstly will be described mapping of general characteristics to functional component. A functional component is having a subset of whole set of general characteristics. This mapping represents that described functional component in a view. According to these characteristics will be also rendered. There is no connection to the implementation of construction. Designer in this state assumes there exists just some construction matching these general characteristics.

The situation is very similar with the **mapping construction characteristics** to **functional component**. A developer assigns a set of construction characteristics to a functional component, each having its value. Developer assumes these construction characteristics will be used during the rendering process, but they do not have to be used at all. This choice will be made by the other developer of specific construction if the developer will use the construction characteristic or not.

Mapping of **general characteristics** to **construction** is depicted in Fig. 3.1 as **General Characteristics Mapping**. It is a set of elements having general characteristic and relevance of concrete construction to this general characteristic. For instance, UI component implementation of form for children will have general characteristic so called *"children-user-group"*. The relevance is designed as integer from 0 to 100. The relevance can be also described as a fuzzy set, where zero means the characteristic does not suits with implementation at all and number 100 means the characteristics fully represents the implementation of UI component. For instance, UI component implementation of form for children will have general characteristic so called *"children-user-group"* with relevance 100, but characteristic so called *"older-user-group"* with relevance 0. Again, this relevance should be very strongly considered by a developer. Due to this mapping logic is possibility to design UI components for mentally challenged people. There developer will keep the rules of accessibility, usability, visibility, readability, compactness, touchscreen design and computer therapy principles and after that describe this UI component with characteristic with unique name *"mentally-challenged-user-group"*. Mapping of general characteristics to construction is depicted on Figure 3.5.



This is content of the textarea that can be extended for more lines. This is content of the textarea that can be extended for more lines. This is content of the textarea that can be extended for more lines.

General characteristics construction mapping

Name	Relevance
- Multiple lines	100
- Extendable size	80
- Responsive	75
- Formatable text	30
- Finance domain	25

Figure 3.5: Illustration how general characteristics are mapped to some construction. Graphic content was designed by the author in HTML and CSS.

Good to point out these characteristics with relevance are then given only to the constructions itself, see on the scheme. Functional components than has just its general characteristics and construction characteristics and then the generator should according to its algorithm get a most suitable construction.

3.4 Algorithm of composition of UI

Now we assume there in a language is designed a set of technologies and a set of general and construction characteristics. There is also defined a view having appropriate functional components to which are mapped the general and construction characteristics. Moreover assume there exists a big set of constructions to cover all combinations of functional components and their mapped characteristics and technologies. Let us assume we want to generate the defined view in a specific technology. **ViewGenerator** is responsible for this generating logic. ViewGenerator has as the input the view and the technology. The algorithm of generating the view in a specified technology is described in the Algorithm 1. The algorithm was separated into several smaller algorithms for better readability.

In algorithm there is firstly filtered the constructions according the given technology. Then is in cycle for each functional component executed the same logic of finding the most appropriate UI construction and then its implementation. As can be seen in the Algorithm 1, the logic also handles the situation when functional component has specified zero number of general characteristics. This algorithm for this case retrieves the first available construction. For the implementation of this algorithm there should be included in DSL some kind of the **default general characteristic** that should be included in some construction and behaved as a fall-back for these edge situations.

For the second situation when construction has specified some general characteristics there is separated Algorithm 2 which describes how we choose the most valid construction according to its characteristics. The algorithm is firstly trying to find all the constructions having all of the general characteristics specified in the functional component. However if there are none of those having all of these characteristics, there is mechanism how to choose at least most valid constructions. We choose every construction having one same general characteristic as the functional component has. By this algorithm we achieve bigger set of constructions and we need to choose only one of them. This choice is then made by relevances of general characteristics in constructions. For this purpose we need only to order the list of achieved constructions by this relevances.

Algorithm of ordering the constructions according to relevances is depicted in Algorithm 3. It is based on principle that sorts the characteristics with the highest relevances to the left. The same relevances keeps next to each other and the characteristics with lower relevances than sorts to the right. Therefore final sorted array should consist of the best constructions on the left of an array.

Once the constructions are ordered, algorithm selects the first construction from the left. Now the implementation of UI in specific technology is loaded from selected most valid construction. All construction characteristics are substituted in appropriate places in implementation of the UI. We can again see that there is not required that implementation has to use these construction characteristics. Some of the constructions might use them and rest of the constructions might not use them.

Finally the implementations in specific technology are concatenated together as whole view and returned from the ViewGenerator.

Algorithm 1: Algorithm of composition of UI in specified UI technology and view

```
Inputs : A view $V$; a technology $T$
Output: A generated UI in specific technology as plain-text
initialization;
$generatedUI \leftarrow emptyString$;
$selectedConstructions \leftarrow emptySet$;
$constructions \leftarrow getAllConstructionsInASystemByTechnology(T)$;
foreach functional component $fc_i \in V$ do
    $constructionCharacteristics \leftarrow getConstructionCharacteristics(fc_i)$;
    $filteredFunctionalConstructions \leftarrow$
      $filterConstructionsAcconrdingToFunctionalComponent(constructions, fc_i)$;
    if $fc_i$ have any general characteristics then
        Logic of choosing valid construction according to general characteristics, see
          Algorithm 2;
    else
        $selectedConstructions \leftarrow$
          $appendSet(selectedConstructions, getFirstElementInSet($
          $filteredFunctionalConstructions))$;
    $sortedSelectedConstructions \leftarrow$ sort the set $selectedConstructions$ according to
      relevances in characteristics, see Algorithm 3 ;
    $selectedConstruction \leftarrow getFirstElementInSet(sortedSelectedConstructions)$;
    $generatedConstruction \leftarrow$
      $generateUI(selectedConstruction, constructionCharacteristics)$;
    $generatedUI \leftarrow appendString(generatedUI, generatedConstruction)$;
return $generatedUI$;
```

Algorithm 2: Logic of choosing valid construction according to characteristics.

```
Inputs : All variables having Algorithm 1
Output: A set of $selectedConstructions$
$generalCharacteristics \leftarrow getGeneralCharacteristics(fc_i)$;
foreach
 $constructionByTechnologyAndFunction_i \in filteredFunctionalConstructions$ do
    $counterForValidatingCharacteristicsFromView \leftarrow 0$;
    foreach $viewFunctionalComponentCharacteristic \in generalCharacteristics$
     do
        $generalCharacteristicsOfConstruction \leftarrow$
         $getConstructionGeneralCharacteristics(constructionByTechnologyAndFunction_i)$;

        foreach $generalCharacteristicOfConstruction \in$
         $generalCharacteristicsOfConstruction$ do
            if $generalCharacteristicOfConstruction ==$
             $viewFunctionalComponentCharacteristic$ then
                $counterForValidatingCharacteristicsFromView + +$;
                break;
    if $Count(generalCharacteristics) ==$
     $counterForValidatingCharacteristicsFromView$ then
        $selectedConstructions \leftarrow$
         $appendSet(selectedConstruction, constructionByTechnologyAndFunction_i)$;

if $isEmptySet(selectedConstructions)$ then
    foreach
     $constructionByTechnologyAndFunction_i \in filteredFunctionalConstructions$
     do
        $validAtLeastForOneCharacteristic \leftarrow false$;
        foreach
         $viewFunctionalComponentCharacteristic \in generalCharacteristics$ do
            $generalCharacteristicsOfConstruction \leftarrow$
             $getConstructionGeneralCharacteristics(constructionByTechnologyAndFunction_i)$;

            foreach $generalCharacteristicOfConstruction \in$
             $generalCharacteristicsOfConstruction$ do
                if $generalCharacteristicOfConstruction ==$
                 $viewFunctionalComponentCharacteristic$ then
                    $validAtLeastForOneCharacteristic \leftarrow true$;
                    break;
            if $validAtLeastForOneCharacteristic$ then
                $selectedConstructions \leftarrow$
                 $appendSet(selectedConstruction, constructionByTechnologyAndFunction_i)$;

                break ;
return $selectedConstructions$;
```

Algorithm 3: Sorting of constructions according to relevance of characteristics.

```
Input  : A $selectedConstructions$ from Algorithm 1
Output: A set of ordered $selectedConstructions$
$orderedSelectedConstructions \leftarrow selectedConstructions$;
$orderedSelectedConstructionsIsNotOrdered \leftarrow true$;
while $selectedConstructionsIsNotOrdered$ do
    $orderedSelectedConstructionsIsNotOrdered \leftarrow false$;
    for $i \leftarrow 0$; $i < length(orderedSelectedConstructions) - 2$ ; $i++$ do
        $orderToLeftCount \leftarrow 0$;
        $orderToRightCount \leftarrow 0$;
        foreach $construction1Characteristic \in$
          $getGeneralCharacteristics(orderedSelectedConstructions_i)$ do
            foreach $construction1Characteristic \in$
              $getGeneralCharacteristics(orderedSelectedConstructions_{i+1})$ do
                $constrCharRelevance1 \leftarrow$
                  $getRelevance(construction1Characteristic)$;
                $constrCharRelevance2 \leftarrow$
                  $getRelevance(construction2Characteristic)$;
                if $constrCharRelevance1 > constrCharRelevance2$ then
                    $orderToLeftCount++$;
                if $constrCharRelevance1 < constrCharRelevance2$ then
                    $orderToRightCount++$;
        if $orderToLeftCount < orderToRightCount$ then
            $switchItemsInArrayByIndexes(i, i+1, orderedSelectedConstructions)$;
            $selectedConstructionsIsNotOrdered \leftarrow true$;
return $orderedSelectedConstructions$;
```

Chapter 4

Implementation

This chapter discusses the implementation of the given language designed in Chapter 3. Section 4.1 provides technical information about the proposed solution including the explanation of used framework and programming language. Text further describes explanations why the choice regarding a programming language is the most suitable for purposes of corima.

The Section 4.2 describes the class-diagram of a proposed solution. Each part of the class-diagram is explained. Finally, there is explained how the algorithm of composition of UI was used in source code and what changes had to be made there.

4.1 Technical information

The only one requirement for technical implementation was to be able to reuse the view logic for all .NET platforms. Therefore, the most suitable solution for corima was to propose a solution as the .NET library. The library itself should be independent from any other .NET framework and the implementations of the UI components will be loaded by the library and used from another .NET source .dll files (files with extension *.dll*).

The next variable needed to discuss is the most suitable .NET programming language for the .NET library. For implementation was chosen the C# programming language. This was chosen from several reasons. First reason is that this language is used in corima as main language and developers who involves in that has the main experience. The second reason is it is the most advanced multi-paradigm language from all .NET languages and is still under development and new features are being developed in new versions. To describe it fully C# is a multi-paradigm programming language containing strong typing, imperative, declarative, functional, generic, class-based, and component-oriented programming disciplines. It was developed by Microsoft within its .NET initiative and later approved as a standard by ECMA-334 and ISO/IEC 23270:2006. C# is one of the programming languages designed for the Common Language Infrastructure.

As an integrated development environment (IDE) was chosen Visual Studio. Visual Studio has build in a lot of programming languages including all .NET programming languages (we need) and other UI specific languages, like XML, HTML ,and CSS. Some other languages can be then included as a plug-ins. Visual Studio also contains of other features like debugging, code analysis, refactoring tools, underlining of errors, and IntelliSense[1].

[1] IntelliSense is an implementation of a code completion used in Visual Studio.

Proposed solution in .NET and programming language C# creates the .NET library. As was described before, the library should not be dependent on implementations of constructions of UI components. The reason is the implementations brings together with them dependencies on other framework libraries that can contain platform specific libraries, such as *Microsoft.Web.Mvc* containing infrastructure for ASP.NET MVC. Therefore these dependencies must be excluded from the general .NET library to the platform specific application.

4.2 Implementation of .NET library

Implementation of the new language proposed in Chapter 3 was produced in several steps when each part of the language was continuously converted into C#. This conversion had several iterations influenced by the changing of language itself and also by the improving of the current solution.

To achieve **Goal 2** there was proposed an implementation of the new language. The final proposed implementation of the new language for UI in .NET is depicted on Figure 4.1. In implementation is kept consistent naming with analysis so the semantic meanings of each classes should be at least partially clear from Chapter 3. The implementation details will be described in the following text.

The implementation will be described from bottom to top, therefore firstly will be described the simplest parts of the system up to the most sophisticated parts.

As can be seen on the Figure 4.1 *ICharacteristic* is the most simplest node on the class diagram and represents the general characteristics. The realizations of the interface *ICharacteristic* are the general characteristics. Unique name is solved by the name of the class (realization). Semantic meaning of each realization should be written as comment next to the class definition. We can see there were already defined some set of general characteristics, such as *SecureCharacteristic* class. An example of realization of ICharacteristic interface in C# can be seen on Listing 4.1.

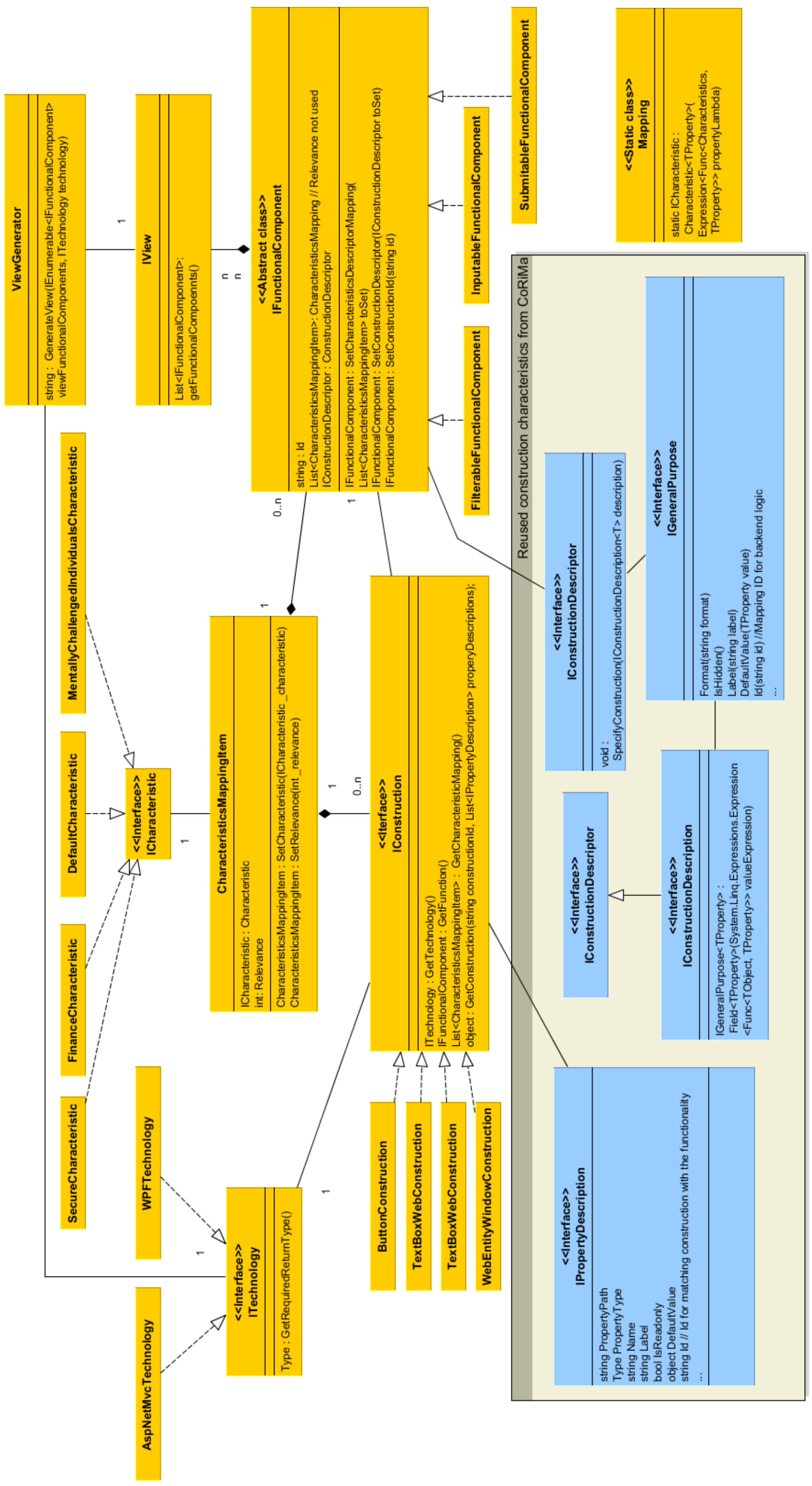

ViewGenerator
string : GenerateView(IEnumerable<IFunctionalComponent> viewFunctionalComponents, ITechnology technology)
IView
List<IFunctionalComponent>: getFunctionalCompoennts()
<<Abstract class>> IFunctionalComponent
string : Id
List<CharacteristicsMappingItem>: CharacteristicsMapping // Relevance not used
IConstructionDescriptor : ConstructionDescriptor
IFunctionalComponent : SetCharacteristicsDescriptorMapping(List<CharacteristicsMappingItem> toSet)
IFunctionalComponent : SetConstructionDescriptor(IConstructionDescriptor toSet)
IFunctionalComponent : SetConstructionId(string id)
SubmitableFunctionalComponent
InputableFunctionalComponent
FilterableFunctionalComponent
<<Static class>> Mapping
static ICharacteristic : Characteristic<TProperty>(Expression<Func<Characteristics, TProperty>> propertyLambda)
MentallyChallengedIndividualsCharacteristic
DefaultCharacteristic
FinanceCharacteristic
SecureCharacteristic
<<Interface>> ICharacteristic
CharacteristicsMappingItem
ICharacteristic : Characteristic
int: Relevance
CharacteristicsMappingItem : SetCharacteristic(ICharacteristic _characteristic)
CharacteristicsMappingItem : SetRelevance(int _relevance)
<<Interface>> IConstruction
ITechnology : GetTechnology()
IFunctionalComponent : GetFunction()
List<CharacteristicsMappingItem> : GetCharacteristicMapping()
object : GetConstruction(string constructionId, List<IPropertyDescription> properyDescriptions);
ButtonConstruction
TextBoxWebConstruction
TextBoxWebConstruction
WebEntityWindowConstruction
WPFTechnology
AspNetMvcTechnology
<<Interface>> ITechnology
Type : GetRequiredReturnType()
Reused construction characteristics from CoRiMa
<<Interface>> IConstructionDescriptor
void : SpecifyConstruction(IConstructionDescription<T> description)
<<Interface>> IGeneralPurpose
Format(string format)
IsHidden()
Label(string label)
DefaultValue(TProperty value)
Id(string id) //Mapping ID for backend logic
...
<<Interface>> IConstructionDescriptor
<<Interface>> IConstructionDescription
IGeneralPurpose<TProperty> : Field<TProperty>(System.Linq.Expressions.Expression <Func<TObject, TProperty>> valueExpression)
<<Interface>> IPropertyDescription
string PropertyPath
Type PropertyType
string Name
string Label
bool IsReadonly
object DefaultValue
string Id // Id for matching construction with the functionality
...
1
1
n
n
0..n
1
1
1
1
0..n
1
1
1

Figure 4.1: Class diagram of implementation of new language.

```
/// <summary>
/// General characteristic describing the UI that is valid for usage in
    finance domain.
/// </summary>
public class FinanceCharacteristic : ICharacteristic
{

}
```

Listing 4.1: Example of implementation of ICharacteristic interface

we can see the class itself not contain any other properties or methods. Therefore it is independent from any .NET technology and can be reused then in any other .NET framework technology.

Implementation of *ICharacteristic* called *DefaultCharacteristic* has its special meaning. It is different from all of the implementations of *ICharacteristic* because is directly bend to the system. Role of the *DefaultCharacteristic* is to specify those constructions that should behave as a fallback for not found valid constructions demanded by the developer (by other characteristics). Or will be used in situations when there are no characteristics specified at all. See implementation of *DefaultCharacteristic* on Listing 4.2.

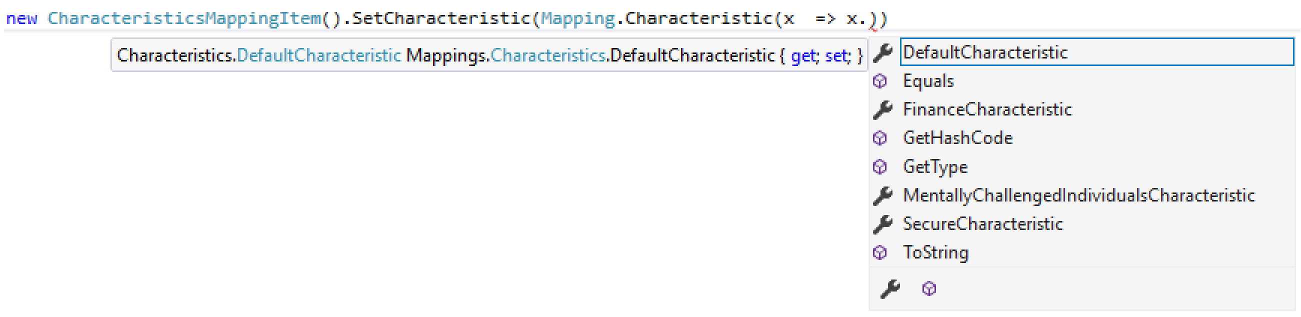
```
/// <summary>
/// General characteristic used as an fallback for any type of functional
    construction not having specified its general characteristics.
/// </summary>
public class DefaultCharacteristic : ICharact8eristic
{

}
```

Listing 4.2: Implementation of *DefaultCharacteristic*

Realization of technology is interface *ITechnology*. The interface contains only one required method and it is *GetRequiredByType()* returning the type in which UI should be generated. This type is essential for generating the UI for different types of technologies, because not all UIs is suitable to be generated just as sequence of characters (string). An example of instance class of this interface is the *AspNetMvcTechnology*.

For the purpose of mapping the general characteristics to functional component or construction was proposed the class *CharacteristicsMappingItem*. The class consists of the Relevance and Characteristic. The Relevance is of type integer and Characteristic is needed to be derived from interface *ICharacteristic*. For better setting of characteristics in method *SetCharacteristic()* there was introduced the static class called *Mapping* that enables the selection of all implementations of *Icharacteristic* interface. The use is proposed for Visual Studio IntelliSense showing all possible implementations. The example of use of Mapping class is depicted on Figure 4.2.



```
new CharacteristicsMappingItem().SetCharacteristic(Mapping.Characteristic(x  => x.))
```

Characteristics.DefaultCharacteristic Mappings.Characteristics.DefaultCharacteristic { get; set; }

Figure 4.2: The example of use of Mapping class in Visual Studio.

An implementation of Construction is depicted on Figure 4.1. It is the interface *IConstruction* containing four methods that needs to be implemented. Method *GetTechnology()* requires to return an implementation of *ITechnology* interface. Method *GetFunction()* has to return the implementation of *IFunctionalComponent*. Third method *GetCharacteristicMapping()* sets the array of implementations of *CharacteristicsMappingItem* described before. Last method is called *GetConstruction* and it returns the implementation of specific construction, where it gets as parameters *constructionId* for mapping back-end logic and *PropertyDescriptions* that represents construction characteristics from Chapter 3. The logic of the *PropertyDescriptions* has been reused from corima and is extracted in class-diagram appropriately. The reason why it was reused is that implementation of these construction characteristics is not part of the assignment and should be also studied well in State of the Art and then also considered in analysis. The whole problem would then too much extend the thesis. Moreover the problem of construction characteristics is already solved by corima, so we can just reuse it for purposes of this thesis. The reused code from corima is commented appropriately. An example of implementation of *IConstruction* interface is *ButtonConstruction*. The construction of UI generated for CRUD operations from **Goal 3** will be described in the end of this chapter.

An abstract class called *IFunctionalComponent* is node of diagram representing the functional component. An abstract class in C# programming language means it can have implemented in class some methods and some of them can contain only their declarations. It has properties *Id*, *CharacteristicsMapping* ,and *ConstructionDescriptor*. *Id* is in abstract class for connection of back-end logic to the control. *CharacteristicsMapping* is the same as for Construction, hovewer there is further not used Relevance in the mapping. The *ConstructionDescriptor* describes the construction characteristics for a given functional component. *ConstructionDescriptor* is class reused from corima and we will describe it no more. For each of these attributes there are already defined setters for simplest initiation of the new instance of class derived from *IFunctionalComponent* abstract class.

Now when all components of the scheme is presented, is needed to express the implementation specifics of *ViewGenerator*. *ViewGenerator* was implemented according to Algorithm 1 and slightly changed to suit for C#.

Firstly in source code are retrieved all classes derived from *IConstruction* interface. This extraction of classes is achieved by C# reflection[2] technology. Then are appropriate parameters assigned to the Algorithm 1. The main change in the source code in comparism with Algorithm 2 is that the source code is extended for situations when functional component

[2]Reflection C# provides objects (of type Type) that describe assemblies, modules and types.

has not specified general characteristics. In this case there are preferred the constructions having the *DefaultCharacteristic* with relevance bigger than 0.

4.3 Implementation of constructions for CRUD operations

To achieve **Goal 3** we will show the implementation of UI components for CRUD operations. There was implemented UI component for viewing forms called *WebEntityWindow*. This form can manage instances of defined classes. These classes can consist of several attributes and these attributes can have several data types. Therefore the UI component for viewing forms then consists of several sub UI componets for editing the different data types. These data types for implemented UI components are:

- string,
- int,
- DateTime,
- bool,
- double.

The *WebEntityWindow* uses then the *IPropertyDescription*s for further specification of each attribute of the managed class. It uses PropertyType to determine what type of attribute is needed to be rendered and according to that what construction should be used. *DefaultValue* is used for getting default value for each attribute's construction. Last important information is how *constructionId* is used in this construction. This *constructionId* is finally also rendered as *id= „constructionId value“* into the UI constrol so developer then can handle the construction with javascript and connect to the back-end logic. In Appendices are included examples of generated views with implementation of *WebEntityWindow*.

Chapter 5

Related work

Automatic generating of complex user interfaces is currently solved by various approaches. These approaches can be divided into aspect based approaches, generation approaches, model based approaches, and inspection based approaches according to their attributes. Each of these approaches has its own advantages suitable for development of specific types of applications. On the other hand, all of this approaches may fail in situations like dynamic changing of UI during the runtime (e.g. validation of forms) or adaptation to user.

One of the simplest solution to develop/design UI is use of visual editors and widget builders, e.g., XAML Designer from Microsoft [1], Qt GUI Designer [5] or Swing GUI builder [8]. Once such tool sets are used to design first version of UI for specific technology-based language and platform, it is very difficult to maintain this UI with these tools. Moreover these tools provide limited set of controls and functionality than it is possible to design in target UI language. Also builders are not able to adapt to the further UI changes in the source code, therefore editor may be disabled for all further changes [29]. So some kind of refactoring, wrapping UIs to functions to be reused or maintained is not possible for these tools. Maintenance is one of the most important requirements to finance applications.

Next systems using widget based builders are systems that consists of form-based UI for accessing data in relational databases. For instance, examples of these systems can be SQL Server Management Studio (SSMS), Oracle SQL Developer, Microsoft Access [13] and Oracle Forms. These systems also consists of semi-automatic generation of the tables. These systems work very effectively for their use, however they are not implemented to generate more complex UIs and also in several technology based languages. They do not contain support for custom templates, context adaptation for disabled users, they contain limited set of components and also UI is being generated for specific platform.

Model based approach, Model-driven development (MDD) [41] is an approach using model as the source of information and the resulting code is being generated from the model using given transformation rules. Variant of this MDD, Model-based user interface development (MBUILD) has then advantage in no replication of information, however it is applied only to basic use-cases.

Further investigation on model-based approach was done by Stephanidis C. [39]. Work provides an information regarding self-adaptation techniques of UI in web platform. They show the differences between adaptivity terms and adaptability. Adaptability is here referred to self adaptation based on knowledge before rendering of the UI. Next, Adaptivity refers to self-adaptations based on knowledge gained during the use of UI. In [39] is proposed project to show adaptivity features. To sum up, project is able to adapt to people with disabilities or adapt to interests of the user. This adaptation is done by context knowledge gained from questionnaires or other system resources. Self adaptation of UI during the runtime is not goal of this thesis, therefore this approach is not valid for our purposes.

MDD is further studied by Sottet et al. [38]. Their work provides information regarding MDD approaches to model-code and model-model transformations. (Semi)automatic UI generation preserving usability is described. Transformation mappings has been defined that keep usability properties. Authors of the work state ergonomic and usability attributes defined by mappings are very often inconsistent and the solution should contain compromises. Finally they also showed their solution on a home heater control. Unfortunately, the work has some disadvantages. For instance, system not allows parametrization of UI controls, modification or positioning. Even more, the presented system is not compatible with traditional development approaches (C++, JavaEE).

To fill the gap between HCI design and software engineering Lyuten [30] applies MDD based approach on a task-centered approach. Concur Task Tree (CTT) notation is used in this paper to design tasks in an environment context-aware manner. However, similar to [38] there is not possible to connect with traditional development approaches.

Calvary et al. [10] propose an unifying reference platform for developing multi-context UIs. The context is divided into environment, user and platform context. There is also introduced the plastic UI supporting multiple contexts of use while preserving usability as context-adaption occurs. However, this approach is too complex for common UIs and is difficult to be used by real systems.

Clerckx et al. used MBUILD model transformations [12]. In [12] occurred inconsistencies for more complex cases of UI. These inconsistencies occurred between the source and derived models. They show in [12] that these inconsistencies created in source models should be back reflected in abstract models too.

UI developed with MDD often struggles from other issues. [34] shows situations when MDD suffers during adaptation and evolution management. MDD can generate common UIs, however when it comes to small modification of UI it is easier in target source code than in model itself [11]. Therefore developer need to add the information to the source code manually and this become very impractical. Next, using domain specific languages (DSL) for the UI definition, these DSLs often do not provide type safety and are edited manually in plain-text as XML. This attitude leads often to errors.

Macik et al. [31] describe their user interface platform (UIP) for machine generation of context sensitive UIs. Their inputs for the generation of UI are abstract UI (AUI) defined in their domain specific language and context model. AUI is defined as hierarchical composite structure describing UI independent from platform. The structure describes what the UI should consists of (input, output and action triggers). In AUI there is no description about the construction of the individual components and the layout of the UI. AUI can be defined manually, by visual editor or generated through code inspection of the persistence model of data oriented applications. Next, context model is defined according to ability based design provided by [40]. The UI generator outputs concrete user interfaces (CUI). These CUI are finally send to platform-specific applications interpreting CUI for the user using native UI elements. Problem is that whole system/framework is based on Java Persistent API (JPA). The back-end logic is then connected to Java because of data mapping. Moreover the UIP clients are platform based and do not allow web based clients. For our purposes we need general solution that can generate UIs in different .NET technologies where backend logic can be written in any .NET technology.

Chapter 6

Evaluation

In this chapter we discuss the implemented solution with respect to the assignment and the defined goals derived from the assignment. Next, we discuss the reasons why none of the related work was not used and a new approach was proposed and implemented in corima. Finally, in this chapter, we present the testing of implemented language and we show how the implemented solution responds to different definition of the UI.

6.1 Evaluation with respect to the assignment

To demonstrate, how the assignment was accomplished, we need to state what has been discussed in the thesis. Now, we review each point of an assignment separately, and we clarify the way we approached that.

1. Study existing languages for system specification and to define a set of annotations for common software user interface (UI) components, we discussed that in Chapter 2 and we defined the set of common UI components there,

2. Study the computer therapy design principles with focus on UI, we studied that in Chapter 2 and extended how different user groups can be handled by a language (Section 2.2.1),

3. Design a language for high-level description of UI requirements, we designed a language in Chapter 3,

4. Implement a tool for automatic generation of required UI from defined and designed descriptions, we implemented a tool in Chapter 4 and demonstrated the use on generation of UI for CRUD operations,

5. Demonstrate the use of the designed language and implement a tool with focus on description of UI requirements for people with disabilities, we demonstrate the use of language in Chapter 3 and we took into account the requirements for people with disabilities so the language contains of general characteristics that are able to obtain descriptions designed for people with disabilities, furthermore now in this chapter we will test the implemented tool to show the generated user interfaces also for these people with disabilities,

6. Evaluate the solution and suggest possible future enhancements, we will evaluate that in this chapter and give suggestions for possible future enhancements.

Therefore, up to the last point the assignment the thesis is already accomplished. Now, we need to evaluate the thesis according to defined goals derived from the assignment.

1. The goal to create a meta-model of a high-level language for describing UI including general attributes of the UI components, we introduced in Chapter 3 (**Goal 0**),
2. **Goal 1** is accomplished at the end of Chapter 3. The algorithm of composition of UI components is explained with possible enhancements,
3. **Goal 2** is achieved in Chapter 4 where is also included class diagram of a proposed system. On the diagram there is also distinguished between reused corima models and new designed modules,
4. **Goal 3** is achieved by implementation of *WebEntityWindow* that is UI component for ASP.NET MVC that creates a form having multiple possible types of fields and is able to manage them,
5. **I**mplementation of constructions for CRUD operations are described in Chapter 4 in Section 4.3 (**Goal 4**).

6.2 Evaluation with respect to the Related work

The Related work chapter described us possible related solutions to the problem of generating UI. In this section we will discuss the advantages and disadvantages of these related works and why these solutions are not suitable for corima purposes.

Visual editors and **widget builders** are tools that best suits for definition of complex UIs containing some interaction with customer and a lot of specific features. Because these features are difficult to define declaratively, these tools have its purpose. Also these tools are used to generate UI in specific technology. For purposes of corima, it is needed to define UIs declaratively and in several UI technologies. Also they lack with maintenance of UI source code. Therefore these tools cannot be used for corima purposes.

Next related works and their disabilities according to corima were described in Chapter 5.

The most suitable found solution would be the last described one by **Macik et al.** [31]. The problem with this solution is that back-end side of the application has to be written in Java Persistent API. This would need to be able to somehow change also for some .NET variant of API. The second problem is the UI definition would need to be extended by an existing corima code (e.g., *IConstructionDescriptor*, and *IPropertyDescription* that limits the usage of this tool. Therefore finally was the best way to propose clean direct solution just for corima.

6.3 Evaluation of implemented language and results

The evaluation of proposed language will be conceived as a list of language requirements and their implementation. In each implementation, actual functionality in the UI generator will be demonstrated. In addition, for each implementation, possible deficiencies and extensions will be discussed.

6.3.1 Evaluation of high-level form of the proposed language

An requirement of language in high-level form is accomplished due to the fact we proposed a general domain model (Figure 3.1) that is handled as a DSL in Chapter 3. This DSL can be further taken and used in any technology, e.g, in XML. We implemented for corima solution of this DSL in .NET technology in C# programming language. Even though it is implemented in C# the DSL has just declarative purpose and is strongly independent from the constructions of controls. These constructions just use the implemented DSL in C# not affecting it with its specific technology, see 4.1.

6.3.2 Evaluation of independence of UI technology

Once the DSL implemented in C# is used we can see from 4.1 the view is totally independent from the technology. The technology is just a parameter to the generator process (ViewGenerator from the class diagram) taken. Therefore there cannot be a way to create there these dependencies.

6.3.3 Evaluation of separation of function and construction

From the 4.1 can be seen the two nodes. These are *IConstruction* and *IFunctionalComponent*. Therefore their roles are seperated. Each *IConstruction* can have one function by which is described, but the *IFunctionalComponent* can have infinite possible *IConstruction*. The mapping algorithm in *ViewGenerator* compose those elements together.

6.3.4 Evaluation of attributes of UI controls and business domain requirements

As a solution for a general attributes in UI there was introduced *IGeneralCharacteristic* in DSL. *IGeneralCharacteristic* can express any kind of additive information to the UI, therefore it perfectly matches the needs. To demonstrate the system can generate different UIs according to different characteristics we created two different constructions of components for CRUD operations. One construction suitable for finance domain and the second suitable for mentally challenged people.

The source code defining these two views differs only in the characteristics. The source codes of the views are shown on Listing 6.1 and Listing 6.2. The only difference in the source codes is the definition of *characteristics*. However, the generated UIs from these views differs a lot. The generated UIs are depicted on Figure 6.1 and Figure 6.2. The UI for mentally challenged people was implemented with respect to specified rules in Chapter 2. We can clearly see these two generated views suits for defined general characteristics and therefore the requirement is accomplished.

From business domain was introduced an requirement for specification of some construction attributes. This requirement was accomplished by reusing source code from corima. To demonstrate the usage, see Listing 6.3. This *IConstructionDescriptor* can describe the fields in CRUD form, e.g., labels.

```
var technology = new AspNetMvcTechnology();
List<IFunctionalComponent> view = new List<IFunctionalComponent>(){};

var characteristics = new List<CharacteristicsMappingItem>(){
    new CharacteristicsMappingItem().SetCharacteristic(
        Mapping.Characteristic(x => x.FinanceCharacteristic))
};
view.Add(new SubmitableFunctionalComponent()
    .SetCharacteristicsDescriptorMapping(characteristics)
    .SetConstructionDescriptor(new UserFormConstructionDescriptor())
    .SetConstructionId("PresenationSubmittableId"));

string viewContent = (string)ViewGenerator.GenerateView(view,
    technology);
```

Listing 6.1: Implementation of view for finance domain

```
        var technology = new AspNetMvcTechnology();
        List<IFunctionalComponent> view = new List<IFunctionalComponent>() { };

        var characteristics = new List<CharacteristicsMappingItem>(){
            new CharacteristicsMappingItem().SetCharacteristic(
                Mapping.Characteristic(x =>
                    x.MentallyChallengedIndividualsCharacteristic))
        };
        view.Add(new SubmitableFunctionalComponent()
            .SetCharacteristicsDescriptorMapping(characteristics)
            .SetConstructionDescriptor(new UserFormConstructionDescriptor())
            .SetConstructionId("PresenationSubmittableId"));

        string viewContent = (string)ViewGenerator.GenerateView(view,
            technology);
```

Listing 6.2: Implementation of view for mentally challenged people

```
public class UserFormConstructionDescriptor : IConstructionDescriptor<TmpUser>{
        public UserFormConstructionDescriptor(){}
        public void SpecifyConstruction(IConstructionDescription<TmpUser>
            description)
        {
            description.Field(x => x.Name)
                      .Label("Name").Id("sss");

            description.Field(x => x.Surname)
                      .Label("Surname");

            description.Field(x => x.DateOfBirth)
                      .Label("Enter date of birth");

            description.Field(x => x.Salary)
                      .Label("Salary");

            description.Field(x => x.Mature)
                      .Label("Is mature?");
        }
    }
```

Listing 6.3: Implementation of *IConstructionDescriptor*

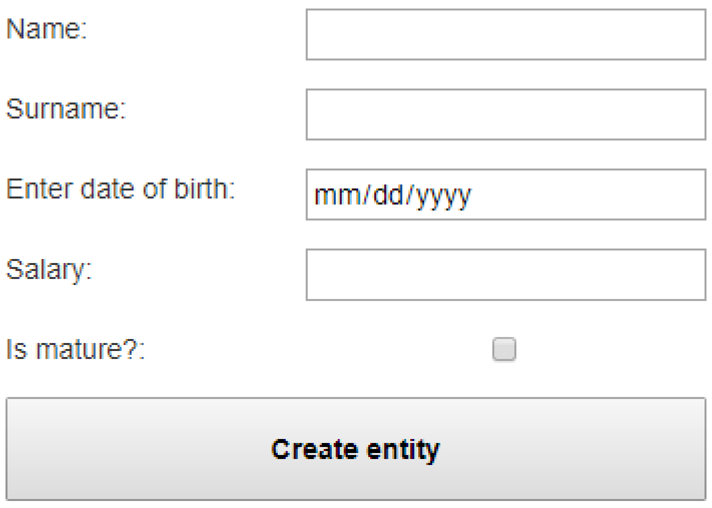


Figure 6.1: An generated UI suitable for finance domain.

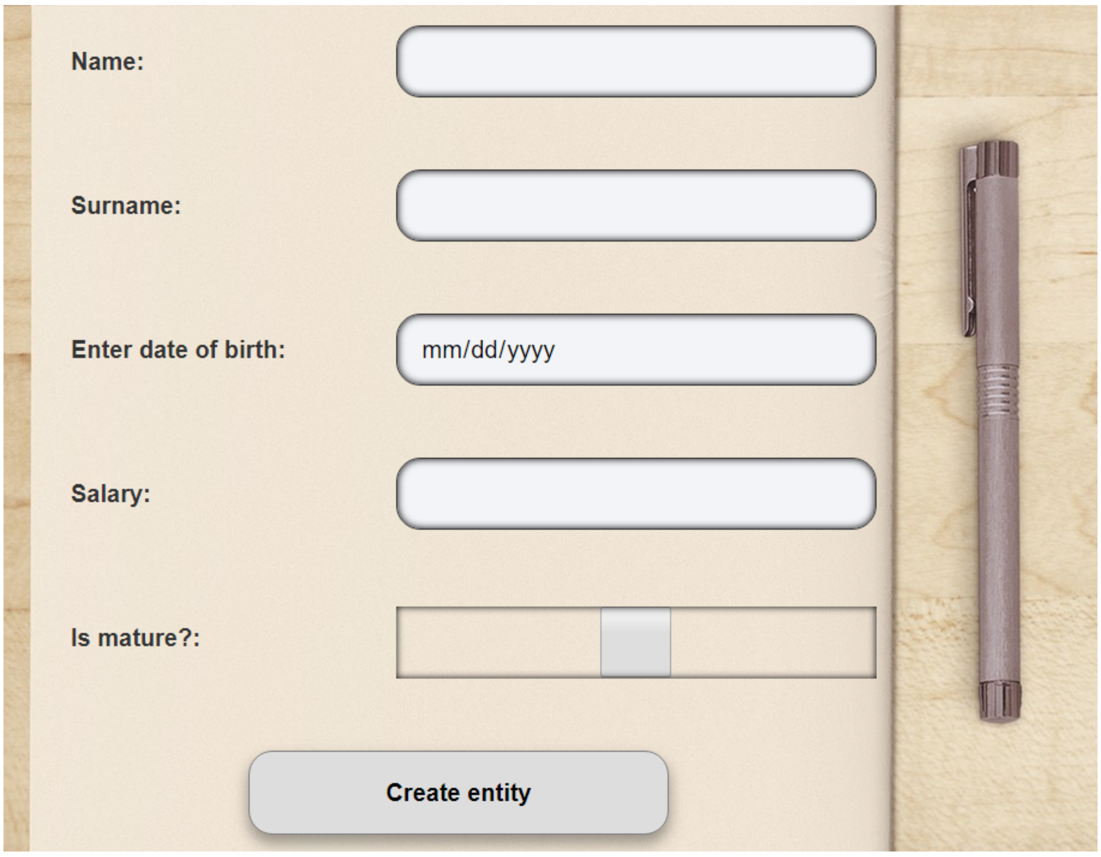


Figure 6.2: An UI better suitable for mentally challenged people generated by the implemented tool according to specified rules in Chapter 2 (Background image from pixabay.com).

6.3.5 Algorithm of composition of UI

The testing was managed in a way where a several general characteristics were assigned to the views and it was observed how the *ViewGenerator* handled the views and which constructions were selected during the generation process. The demonstration of the generated views with different general characteristics was depicted on Figure 6.1 and Figure 6.2.

The possible improvement is in the algorithm of selection of the constructions according to its general characteristics. The problem is when there will exist a huge set of construction having very similar general characteristics with almost the same relevances. In current solution the algorithm choose from very similar construction the one with the first name in the alphabetic order.

6.3.6 Reduction of cost within the migration

The reduction of cost within migration can be seen when we have one definition of view and just change the technology in which the UI should be generated. Therefore we can reuse this UI definition in any future technology and back-end logic too. The only work for a new technology will be to implement in new technology exactly same constructions as in previous technology. This process leads to reduce the implementation time of constructions, because previous construction are having a lot of general characteristics describing the construction so the developer can better understand for which purpose the construction should be and what properties should consist of. Also the constructions very probably will be implemented as build from smaller constructions to build bigger one as is depicted on Figure 3.1. Hence, the generator very probably will reduce cost of migration from one UI technology in .NET to another. The exact reduced cost will be calculated further when the generator will be more used in corima.

Chapter 7

Conclusion

In this Master's thesis, we studied possibilities of separation of function and construction (F/C) and graphical user interface according to HCI, touchscreen design and mentally challenged people needs. Further in the thesis were studied software methodologies, like model driven engineering, that is used for the definition of the new model (language definition). Languages like SBVR or OCL are then described to benefit from their strong points. Further there are studied the typical finance domain attributes important for corima.

We analyzed the set of requirements on any language. When we studied all existing languages we analyzed there has to be proposed a new language. According to the requirements we designed a new language as DSL having the possibility to express the user interface as a functional components having certain general attributes called characteristics in the language. The designed DSL is also having the possibility to include the construction of these functional components in specific technologies and have the certain construction attributes (called construction characteristics in a DSL) according to studied finance domain attributes.

At the end of analysis, we define the algorithm that composes the defined user interface in new language into real user interface in specific UI technology. This algorithm is introduced in pseudo-code and further described for better understanding.

The designed language was taken and used as a base for an implementation of a language in .NET and programming language C#. Including the implementation was introduced a generator tool using defined algorithm of composition of UI components. Generator has as inputs the technology (in which UI should be generated) and the defined view. The process of generation of UI from available implemented constructions is automated by the generator.

In evaluation we states all of the points of assignment were accomplished. Further we show how the language responds on different definition of UI with different set of characteristics. Therefore the way of separation of function and construction results in expected results. This way we have achieved the expected results for the generation of UI for mentally challenged people and finance sector too. Finally we state the real reduced cost within migration is not calculated, however there are several reasons why the cost should be reduced and the cost will be definitely calculated when the migration will be finished in corima.

Possible enhancements are defined for the algorithm of composition of UI components. The enhancement is about the optimization of the algorithm when there will be defined a huge set of constructions in a system. There constructions having very similar construction characteristics can be selected better. The whole designed language and implemented solution is currently being integrated in corima and is running in real environment.

Bibliography

[1] Creating a UI by using XAML Designer in Visual Studio. [Online; 20.5.2018]. Retrieved from: https://docs.microsoft.com/en-us/visualstudio/designers/creating-a-ui-by-using-xaml-designer-in-visual-studio

[2] Extensible Markup Language (XML) 1.0 (Fifth Edition). [Online; 8.1.2017]. Retrieved from: https://www.w3.org/TR/REC-xml/

[3] Object Constraint Language™ (OCL™). [Online; 8.1.2017]. Retrieved from: http://www.omg.org/spec/OCL/

[4] Project I-SEN (open community of parrents, pedagogues, therapists and IT experts). [Online; 8.1.2017]. Retrieved from: http://www.i-sen.cz

[5] Qt GUI Designer. [Online; 20.5.2018]. Retrieved from: https://doc.qt.io/archives/2.3/designer.html

[6] Red Hat. [Online; 8.1.2017]. Retrieved from: https://www.redhat.com/en

[7] Semantics Of Business Vocabulary And Rules™ (SBVR™). [Online; 8.1.2017]. Retrieved from: http://www.omg.org/spec/SBVR/

[8] Swing gui builder (2013). [Online; 20.5.2018]. Retrieved from: https://netbeans.org/features/java/swing.html

[9] Baisley, D.; Hall, J.; Chapin, D.: Semantic Formulations in SBVR. [Online; 8.1.2017]. Retrieved from: https://www.w3.org/2004/12/rules-ws/paper/67/

[10] Calvary, G.; Coutaz, J.; Thevenin, D.; et al.: A Unifying Reference Framework for multi-target user interfaces. *Interacting with Computers*. vol. 15, no. 3. 2003: pp. 289–308. doi:10.1016/S0953-5438(03)00010-9. /oup/backfile/content_public/journal/iwc/15/3/10.1016_s0953-5438(03)00010-9/3/iwc15-0289.pdf. Retrieved from: http://dx.doi.org/10.1016/S0953-5438(03)00010-9

[11] Cerny, T.; Donahoo, M. J.; Song, E.: Towards Effective Adaptive User Interfaces Design. In *Proceedings of the 2013 Research in Adaptive and Convergent Systems*. RACS '13. New York, NY, USA: ACM. 2013. ISBN 978-1-4503-2348-2. pp. 373–380. doi:10.1145/2513228.2513278. Retrieved from: http://doi.acm.org/10.1145/2513228.2513278

[12] Clerckx, T.; Luyten, K.; Coninx, K.: The Mapping Problem Back and Forth: Customizing Dynamic Models While Preserving Consistency. In *Proceedings of the 3rd Annual Conference on Task Models and Diagrams*. TAMODIA '04. New York, NY, USA: ACM. 2004. ISBN 1-59593-000-0. pp. 33–42. doi:10.1145/1045446.1045455.
Retrieved from: http://doi.acm.org/10.1145/1045446.1045455

[13] Conrad, J.; Viescas, J.: *Microsoft Access 2010 Inside Out*. Microsoft Press. first edition. 2010. ISBN 0735626855, 9780735626850.

[14] Dietz, J.; Hoogervorst, J.: Theories in Enterprise Engineering Memorandum - BETA. 2014.
Retrieved from: http://www.ciaonetwork.org/uploads/eewc2014/EE-theories

[15] Dietz, J.; Hoogervorst, J.: Theories in Enterprise Engineering Memorandum - TAO. 2014.
Retrieved from: http://www.ciaonetwork.org/uploads/eewc2014/EE-theories

[16] Dietz, J. L. G.: *Enterprise Ontology: Theory and Methodology*. Berlin, Heidelberg: Springer-Verlag. 2006. ISBN 3540291695.

[17] Dix, A.; Finlay, J. E.; Abowd, G. D.; et al.: *Human-Computer Interaction (3rd Edition)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc.. 2003. ISBN 0130461091.

[18] Dvorak, O.; Pergl, R.; Kroha, P.: Affordance-driven Software Assembling. *Enterprise Engineering Working Conference*. 2018. doi:inprintpaper.

[19] Fiala, J.; Kočí, R.: Počítačová terapie jako koncept nové formy terapie pro osoby s mentálním postižením: teorie i praxe. *Journal of Technology and Information Education*. vol. 6, no. 1. 2014: pp. 89–103. ISSN 1803-537X.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=10718

[20] Fiala, J.; Kočí, R.: *Computer as Therapy in role of alternative and augmentative communication*. 2015. 34–42 pp.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php.cs?id=10737

[21] Fiala, J.; Zendulka, J.: Mentally challenged as design principles and models for their applications. *Applied Computer Science*. vol. 12, no. 4. 2016: pp. 28–48. ISSN 1895-3735.
Retrieved from: http://www.fit.vutbr.cz/research/view_pub.php?id=11129

[22] Fowler, M.: *Domain Specific Languages*. Addison-Wesley Professional. first edition. 2010. ISBN 0321712943, 9780321712943.

[23] Inostroza, R.; Rusu, C.; Roncagliolo, S.; et al.: *Usability Heuristics Validation through Empirical Evidences: A Touchscreen-Based Mobile Devices Proposal*. Nov 2012. 60-68 pp.. doi:10.1109/SCCC.2012.15.

[24] Inostroza, R.; Rusu, C.; Roncaliolo, S.; et al.: *Design Patterns for Touchscreen-based Mobile Devices: Users Above All!* ChileCHI '13. New York, NY, USA: ACM. 2013. ISBN 978-1-4503-2200-3. 50–51 pp.. doi:10.1145/2535597.2535616.
Retrieved from: http://doi.acm.org/10.1145/2535597.2535616

[25] ISO 9241-11:1998: Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: Guidance on usability. Standard. March 1998.

[26] ISO 9241-171:2008: Ergonomics of human-system interaction – Part 171: Guidance on software accessibility. Standard. 2008.

[27] Kalina, J.: *Vývoj i-CT frameworku a jeho aplikace pro komunikaci typu ANO/NE*. Master's Thesis. Brno: Vysoké učení technické v Brně. Fakulta informačních technologií. The address of the publisher. 2016.
Retrieved from: http://hdl.handle.net/11012/61917

[28] Kelly, S.; Tolvanen, J.: *Domain-Specific Modeling - Enabling Full Code Generation*. Wiley. 2008. ISBN 978-0-470-03666-2.
Retrieved from:
http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470036664.html

[29] Kennard, R.; Leaney, J.: Towards a general purpose architecture for UI generation. *Journal of Systems and Software*. vol. 83, no. 10. 2010: pp. 1896 – 1906. ISSN 0164-1212. doi:https://doi.org/10.1016/j.jss.2010.05.079.
Retrieved from:
http://www.sciencedirect.com/science/article/pii/S0164121210001597

[30] Kris, L.; Chris, V.; Jan, V. d. B.; et al.: Context-sensitive User Interfaces for Ambient Environments: Design, Development and Deployment. In *Mobile Computing and Ambient Intelligence: The Challenge of Multimedia*, edited by N. Davies; T. Kirste; H. Schumann. number 05181 in Dagstuhl Seminar Proceedings. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany. 2005. ISSN 1862-4405.
Retrieved from: http://drops.dagstuhl.de/opus/volltexte/2005/377

[31] Macik, M.; Cerny, T.; Slavik, P.: Context-sensitive, cross-platform user interface generation. *Journal on Multimodal User Interfaces*. vol. 8, no. 2. Jun 2014: pp. 217–229. ISSN 1783-8738. doi:10.1007/s12193-013-0141-0.
Retrieved from: https://doi.org/10.1007/s12193-013-0141-0

[32] Martin, J.: *Managing the Data Base Environment*. Upper Saddle River, NJ, USA: Prentice Hall PTR. first edition. 1983. ISBN 0135505828.

[33] Mernik, M.; Heering, J.; Sloane, A. M.: When and How to Develop Domain-specific Languages. *ACM Comput. Surv.*. vol. 37, no. 4. December 2005: pp. 316–344. ISSN 0360-0300. doi:10.1145/1118890.1118892.
Retrieved from:
http://doi.acm.org.ezproxy.lib.vutbr.cz/10.1145/1118890.1118892

[34] Morin, B.; Barais, O.; Jezequel, J. M.; et al.: Models@ Run.time to Support Dynamic Adaptation. *Computer*. vol. 42, no. 10. Oct 2009: pp. 44–51. ISSN 0018-9162. doi:10.1109/MC.2009.327.

[35] Nielsen, J.: *Usability Engineering*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.. 1993. ISBN 0125184050.

[36] Nilsson, E. G.: Design Patterns for User Interface for Mobile Applications. *Adv. Eng. Softw.*. vol. 40, no. 12. December 2009: pp. 1318–1328. ISSN 0965-9978. doi:10.1016/j.advengsoft.2009.01.017.
Retrieved from: http://dx.doi.org/10.1016/j.advengsoft.2009.01.017

[37] Rodrigues da Silva, A.: Model-driven Engineering. *Comput. Lang. Syst. Struct.*. vol. 43, no. C. October 2015: pp. 139–155. ISSN 1477-8424. doi:10.1016/j.cl.2015.06.001.
Retrieved from: http://dx.doi.org/10.1016/j.cl.2015.06.001

[38] Sottet, J.-S.; Calvary, G.; Coutaz, J.; et al.: A Model-Driven Engineering Approach for the Usability of Plastic User Interfaces. In *Engineering Interactive Systems*, edited by J. Gulliksen; M. B. Harning; P. Palanque; G. C. van der Veer; J. Wesson. Berlin, Heidelberg: Springer Berlin Heidelberg. 2008. ISBN 978-3-540-92698-6. pp. 140–157.

[39] Stephanidis, C.: Adaptive Techniques for Universal Access. *User Modeling and User-Adapted Interaction*. vol. 11, no. 1. Mar 2001: pp. 159–179. ISSN 1573-1391. doi:10.1023/A:1011144232235.
Retrieved from: https://doi.org/10.1023/A:1011144232235

[40] Wobbrock, J. O.; Kane, S. K.; Gajos, K. Z.; et al.: Ability-Based Design: Concept, Principles and Examples. *ACM Trans. Access. Comput.*. vol. 3, no. 3. April 2011: pp. 9:1–9:27. ISSN 1936-7228. doi:10.1145/1952383.1952384.
Retrieved from: http://doi.acm.org/10.1145/1952383.1952384

[41] Černý, T.; Song, E.: Model-driven Rich Form Generation. vol. 15. 07 2012: pp. 2695–2714.

Appendices

List of Appendices

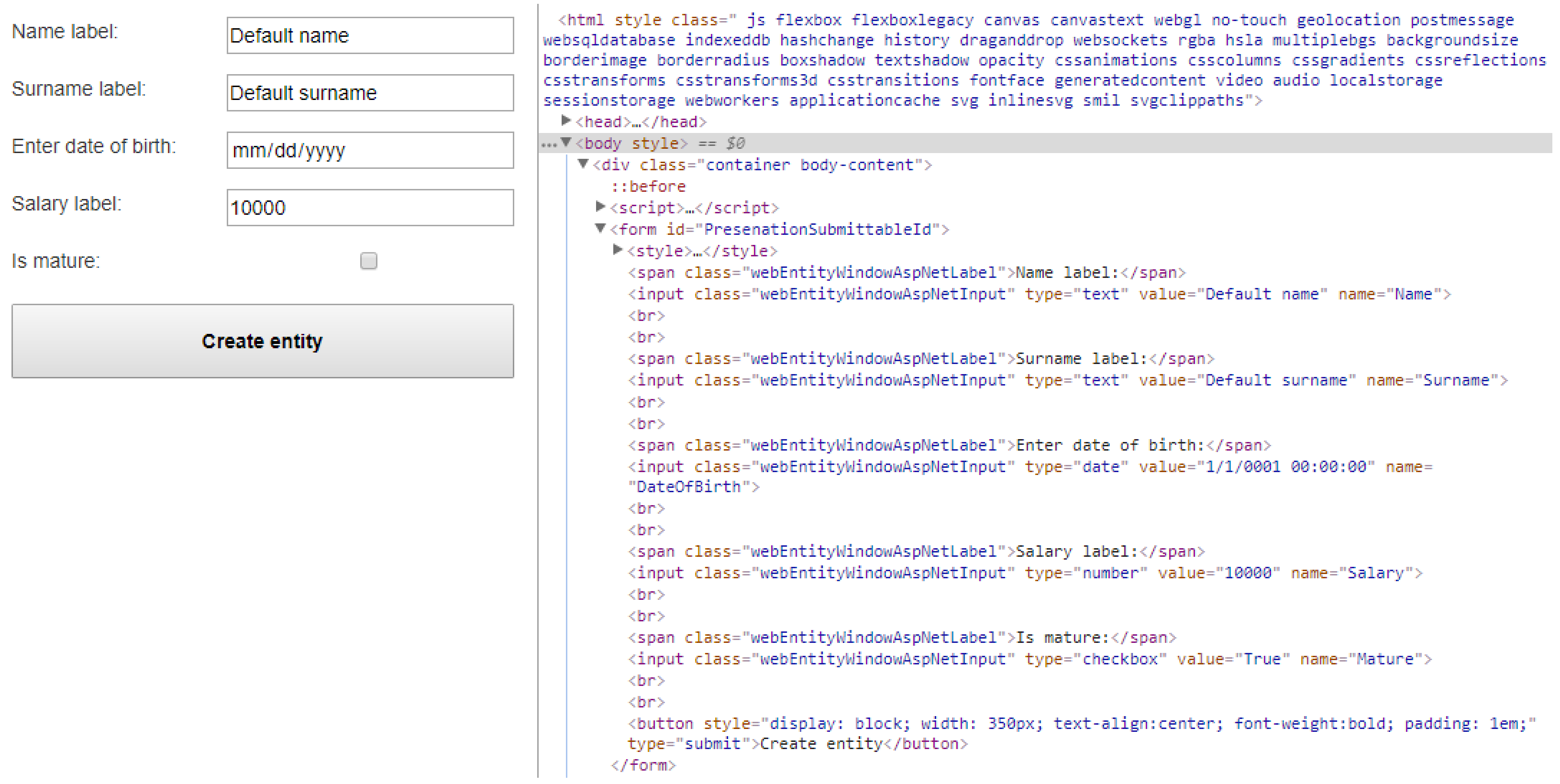
Appendix A

CD contents

As a part of the thesis are also attached contents of implemented application on an enclosed storage media. Source files of the implemented solution are placed into the folder `src`. There is also stored the file *readme.txt*, where is described a way of usage of proposed implementation. There are also included predefined *.sln* files for opening the project directly in Visual Studio.

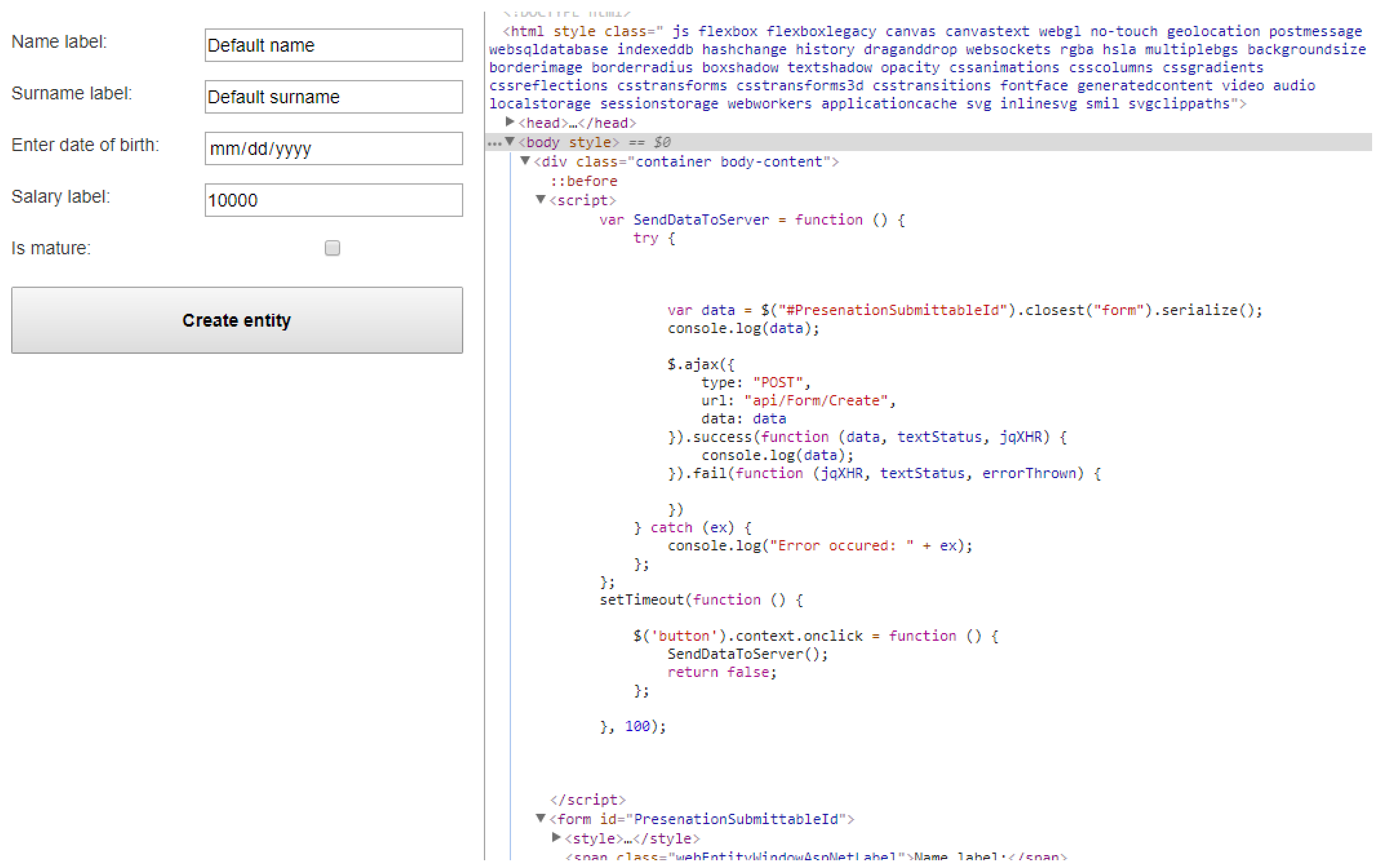
Appendix B

Figures



```
<html style class=" js flexbox flexboxlegacy canvas canvastext webgl no-touch geolocation postmessage
websqldatabase indexeddb hashchange history draganddrop websockets rgba hsla multiplebgs backgroundsize
borderimage borderradius boxshadow textshadow opacity cssanimations csscolumns cssgradients cssreflections
csstransforms csstransforms3d csstransitions fontface generatedcontent video audio localstorage
sessionstorage webworkers applicationcache svg inlinesvg smil svgclippaths">
 ▶<head>…</head>
…▼<body style> == $0
  ▼<div class="container body-content">
     ::before
   ▶<script>…</script>
   ▼<form id="PresenationSubmittableId">
     ▶<style>…</style>
      <span class="webEntityWindowAspNetLabel">Name label:</span>
      <input class="webEntityWindowAspNetInput" type="text" value="Default name" name="Name">
      <br>
      <br>
      <span class="webEntityWindowAspNetLabel">Surname label:</span>
      <input class="webEntityWindowAspNetInput" type="text" value="Default surname" name="Surname">
      <br>
      <br>
      <span class="webEntityWindowAspNetLabel">Enter date of birth:</span>
      <input class="webEntityWindowAspNetInput" type="date" value="1/1/0001 00:00:00" name=
      "DateOfBirth">
      <br>
      <br>
      <span class="webEntityWindowAspNetLabel">Salary label:</span>
      <input class="webEntityWindowAspNetInput" type="number" value="10000" name="Salary">
      <br>
      <br>
      <span class="webEntityWindowAspNetLabel">Is mature:</span>
      <input class="webEntityWindowAspNetInput" type="checkbox" value="True" name="Mature">
      <br>
      <br>
      <button style="display: block; width: 350px; text-align:center; font-weight:bold; padding: 1em;"
      type="submit">Create entity</button>
    </form>
```

Figure B.1: Generated view using *WebEntityWindow* construction. See different inputs for different data types, default values, and defined id in the generated source code.



Name label: Default name

Surname label: Default surname

Enter date of birth: mm/dd/yyyy

Salary label: 10000

Is mature:

```
<!DOCTYPE html>
 <html style class=" js flexbox flexboxlegacy canvas canvastext webgl no-touch geolocation postmessage
websqldatabase indexeddb hashchange history draganddrop websockets rgba hsla multiplebgs backgroundsize
borderimage borderradius boxshadow textshadow opacity cssanimations csscolumns cssgradients
cssreflections csstransforms csstransforms3d csstransitions fontface generatedcontent video audio
localstorage sessionstorage webworkers applicationcache svg inlinesvg smil svgclippaths">
  ▶<head>…</head>
…▼<body style> == $0
   ▼<div class="container body-content">
       ::before
     ▼<script>
            var SendDataToServer = function () {
                try {


                    var data = $("#PresenationSubmittableId").closest("form").serialize();
                    console.log(data);

                    $.ajax({
                        type: "POST",
                        url: "api/Form/Create",
                        data: data
                    }).success(function (data, textStatus, jqXHR) {
                        console.log(data);
                    }).fail(function (jqXHR, textStatus, errorThrown) {

                    })
                } catch (ex) {
                    console.log("Error occured: " + ex);
                };
            };
            setTimeout(function () {

                $('button').context.onclick = function () {
                    SendDataToServer();
                    return false;
                };

            }, 100);


       </script>
     ▼<form id="PresenationSubmittableId">
       ▶<style>…</style>
        <span class="webEntityWindowAspNetLabel">Name label:</span>
```

Figure B.2: Generated view using *WebEntityWindow* construction with focus on connection to back-end with JavaScript.