



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

ASSIGNMENT OF MASTER'S THESIS

Title: Evolvability of UI technologies
Student: Bc. Václav Mareš
Supervisor: Mgr. Ondřej Dvořák
Study Programme: Informatics
Study Branch: Web and Software Engineering
Department: Department of Software Engineering
Validity: Until the end of summer semester 2019/20

Instructions

Nowadays, the pace of technological progress accelerates. The companies are under a big market pressure to evolve their solutions to latest technologies quickly. Such an upgrade is usually related to a big investment and is often very difficult to achieve. One of the technologies which struggle with that phenomenon are those used to develop User Interfaces (UI). Map common Web and Desktop UI technologies in the past years, clarify concepts of their architectures and evaluate their limitations when upgrading software from one technology to another.

1. Analyze common architecture patterns in UI technologies
2. Review latest trends in so-called evolvable architectures
3. Implement an explanatory application in few technologies of choice and demonstrate their limitation when upgrading the application from one technology to another
4. Clarify architecture concepts which limits the the upgrade of UI efficiently
5. Summarize and evaluate the results reached

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague January 1, 2019



**FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE**

Master's thesis

Evolvability of UI technologies

Bc. Václav Mareš

Department of Software Engineering

Supervisor: Mgr. Ondřej Dvořák

May 8, 2019

Acknowledgements

I would like to thank my supervisor for guiding me on the path of writing this thesis and his valuable input. I would also like to thank professors H. Mannaert and J. Verelst for their help with NS theory and their ideas on my thesis. I also express many thanks to my girlfriend and family for their never ending support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on May 8, 2019

.....

Czech Technical University in Prague

Faculty of Information Technology

© 2019 Václav Mareš. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Mareš, Václav. *Evolvability of UI technologies*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2019.

Abstrakt

Tato diplomová práce se zaměřuje na evolvabilitu technologií uživatelských rozhraní. V textu jsou popsány jednotlivé architektury takových systémů a jejich principy. Jsou představeny dvě metodologie, které se zaměřují na koncept evolvability. Text obsahuje rozvahu nad možnými přístupy k přechodu mezi technologiemi grafických uživatelských rozhraní. Dále obsahuje přehled dvou .NET technologií, zařazení jejich architektur a zhodnocení jejich evolvability. Uveden je také příklad aplikace a její převedení z jedné technologie do druhé. Výstupem této práce je přehled aspektů, které hrají roli při změně grafického uživatelského rozhraní.

Klíčová slova evolvabilita, grafické uživatelské rozhraní, normalizované systémy, evolvabilní architektury, GUI architektury

Abstract

This master's thesis looks at the evolvability of graphical user interface technologies. The text describes different architectures of these systems and their principles. It presents two different methodologies that focus on the concept of evolvability. It contains reasoning about approaches to transitioning from one graphical user interface technology to another. Overview of two .NET technologies their architecture categorization and evolvability evaluation. And an example application transition between the two technologies. The product of this thesis is an overview of all the aspects that play a role in a graphical user interface migration.

Keywords evolvability, graphical user interface, normalized systems, evolutionary architectures, GUI architectures

Contents

Introduction	1
Motivation	1
Goals	2
Structure of the Thesis	2
1 State-of-the-art	3
1.1 Paradigms	3
1.2 Methodology	21
2 Goals revisited	29
3 Analysis	31
3.1 Frameworks	31
3.2 Transition approaches	45
3.3 Testing	49
3.4 Summary	50
4 Case study	51
4.1 Introduction	51
4.2 Implementation	54
4.3 Summary	61
5 Related Work	63
Evaluation	65
Conclusion	67
Bibliography	69
A Acronyms	73

B Contents of enclosed SD card

75

List of Figures

1.1	Observer pattern	4
1.2	Composite pattern	5
1.3	Chain of Responsibility pattern	6
1.4	Example GUI	6
1.5	Presentation patterns	7
1.6	Record Set	9
1.7	MVC pattern	11
1.8	Passive View variation of MVP	15
1.9	MVP/MVC pattern for Web	16
1.10	Presentation Model	17
1.11	MVVM Pattern	19
1.12	MVI Pattern	20
1.13	Example of Evolutionary architecture's fitness function fit	26
3.1	Example of Add/Remove User Control	37
3.2	Abstraction layer placement	48
3.3	Abstraction layer usage	48
4.1	Car Dealership app use cases	52
4.2	Login screen	55
4.3	Select branch screen	55
4.4	Main screen	56
4.5	Set data dialog	56
4.6	Migrated Set dialog	59

List of Listings

3.1	Simple form example	33
3.2	Event handler function	34
3.3	Simple binding example	35
3.4	Complex binding example	36
3.5	XAML example	40
3.6	Code behind file ExpenseItHome.xaml.cs	42
3.7	Interoperability example	43
4.1	Set Dialog implementation	57
4.2	Log of action messages	58
4.3	XAML describing Set Dialog view	60
4.4	XAML describing Set Dialog view with MVVM	60
4.5	Set Dialog MVVM implementation	61

Introduction

Motivation

The world of software engineering today is an ever faster moving colossus of different frameworks and approaches to all sorts of problems. It seems like a new option to choose from for one's project pops up nearly every week e.g., see the dynamics of JavaScript [1]. If we have learnt to respect Moore's Law [2] for hardware that exponential increase of transistors enables in a few years things impossible to imagine at current point. It is in our best interest to accept the idea of Ray Kurzweil the so-called Law of Accelerating Returns [3], which claims that technological change is advancing exponentially and that future in the 100 years of 21st century should be counted more as 20,000 years of progress at present rate. If we accept this law it quickly becomes daunting for me and every other developer. We have too many options to choose from and even the option we choose will soon become obsolete. How should we deal with this?

First of all, I will narrow the scope, a lot. My interest lies in Graphical User Interface (GUI) technologies, they are affected by the Law of Accelerating Returns same way as any other technology. Every application that lives for long enough time will encounter the need for a change in the presentation layer. The past few decades changed the paradigm of GUI several times not to mention hundreds of technologies available. One of the big pressures to change GUI, but not the only one, is a move to cloud and with it related switch to web based GUIs; however, there are many other reasons as well.

With this narrowed scope I am still not attempting to set a silver bullet answer for such a complicated question. These changes to GUI are complicated and costly, yet they are seen necessary for many companies and projects. What I aim to do is to clarify concepts of GUI architectures, evaluate their limitations

when upgrading from one to another, and reach some tips to follow in order to make the transition easier.

Goals

The following list of goals serves as a template for this thesis and guides the structure of it. I will come to defining more exact goals after the State-of-the-Art in chapter 1.

1. Analyze common architecture patterns in UI technologies
2. Review latest trends in so-called evolvable architectures
3. Implement an explanatory application in few technologies of choice and demonstrate their limitation when upgrading the application from one technology to another
4. Clarify architecture concepts which limits the upgrade of UI efficiently
5. Summarize and evaluate the results reached

Structure of the Thesis

As just mentioned the first chapter is dedicated to the State-of-the-Art. Here, I have a look at GUI paradigms, the basics of how they work, and their benefits and drawbacks. I also present the methodologies that I use for evaluating the upgradeability of GUIs, their point of view at systems, and principles of evaluation.

The second chapter, is dedicated to revisiting thesis' goals. Here, based on the knowledge from the first chapter I set the list of exact goals I want to achieve or provide answer to.

In the third chapter, I choose two GUI technologies from the .NET environment. Analyse them using the GUI paradigms and evaluate them with the help of the methodologies described earlier.

The fourth chapter, is a presentation of a case study. An example of application implemented and transitioned between the above mentioned technologies.

Fifth chapter presents related work to this thesis and adds some context.

Closing the thesis are two sections Evaluation, where I provide answer to all my goals from third chapter, and Conclusion, where I summarize the whole thesis and provide some ideas for future work.

State-of-the-art

1.1 Paradigms

Graphical user interfaces or GUIs have become a must for almost every software application and fill vast majority of our screens. As such the requirements for GUI have changed over time and multiple different approaches and architectures have been described to solve problems encountered on this journey. Here, I take a look at some of the most known architectural patterns, a bit of their history, and their reasons for existence. I also compare them, and summarize their pros and cons, in order to build knowledge base which helps me to classify specific frameworks mentioned later in this thesis.

1.1.1 Architectural and Design Patterns

To avoid confusion I would like to describe the difference between architectural and design patterns that I am using in this thesis.

The design patterns are widely know from the book *Design Patterns, elements of reusable Object-Oriented software* by the Gang of Four [4]. It was defined as follows:

Design patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.

This means that design patterns are an abstract way of solving a recurring problem. We can use design patterns on different levels of abstraction as well as well as on small classes or modules of big system.

The architectural patterns have a broader scope. They describe organization on the highest levels of abstraction. They as well serve to solve problems, but also just to keep a mental picture of a system or subsystem.

Architectural patterns often, if not always, use many instances of design patterns. This can be seen in the analysis chapter 3 where I am referring to them quite a bit. It can also be seen in the *Design patterns* book, where the Model View Controller (MVC) architectural pattern is used as an example of many different design patterns in collaboration [4, p. 4]. The opposite is not true, design patterns do not use architectural patterns in their description.

There are many design patterns used in the architectures I describe in this chapter. Here I am presenting few that I will reference later, but more can be found in the already mentioned book *Design Patterns* [4].

1.1.1.1 Observer pattern

The *Observer pattern* [4, p. 293], also known as *publish-subscribe pattern*, describes how to establish a relationship between a subject and its observers, see figure 1.1.

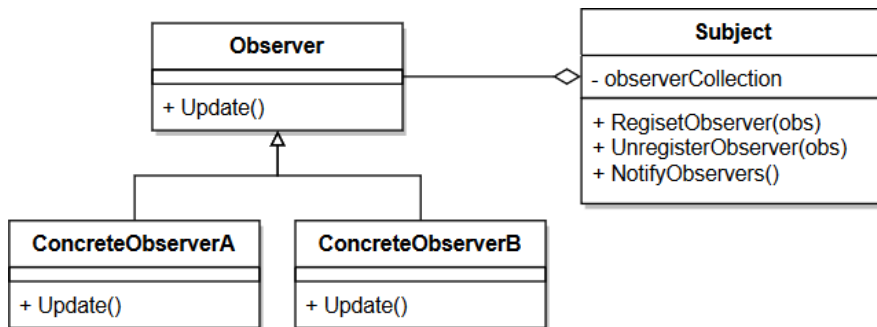


Figure 1.1: Observer pattern

Observers are notified whenever the subject undergoes a change in a state. The observers then retrieve the subject's state and carry on their business. The subject does not need to know who the observers are nor how many there are. This pattern is useful to announce change in a loosely coupled way without assumptions about the observers.

1.1.1.1.1 Data Binding One of the use-cases for the *Observer pattern* is *Data Binding*. Generally this term means connecting data of one entity to data of another entity. For GUI purposes it refers to the link between data inside elements that are able to be rendered on screen and the source of the data be it a business logic object or data transfer object. This binding can be supported directly by a framework or not. It can also be unidirectional

propagating changes from GUI elements to the data objects or bidirectional. It all depends on implementation details. In all these cases the mechanism to realize the link is usually the *Observer pattern*.

1.1.1.2 Composite pattern

The *Composite pattern* [4, p. 163] is an abstraction in order to treat individual objects and compositions of objects uniformly, see figure 1.2.

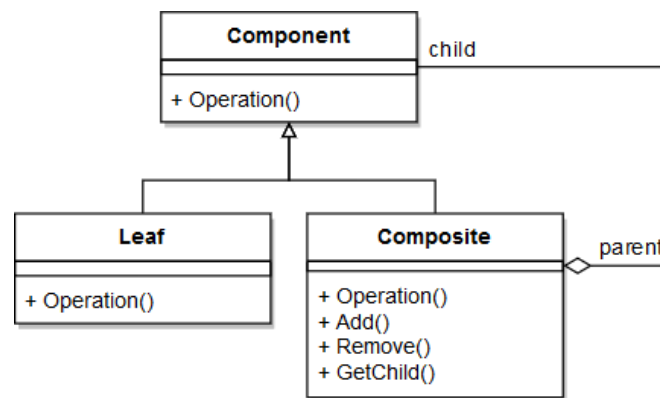


Figure 1.2: Composite pattern

The key to this pattern is the abstraction class that represents both primitives and their containers allowing to use their common functionality. This allows us to manipulate hierarchies of objects without special treatment viewing them all as **Components**.

1.1.1.3 Chain of Responsibility pattern

The *Chain of Responsibility pattern* [4, p. 223] describes a way to avoid coupling between sender of a request and its receiver.

The core idea is that a request is made and there is a **Handler** that has the option to react on it. It also has a reference to its successor so if it sees fit it can forward the request. The specific handlers are derived from a common class, see figure 1.3.

This is very useful in GUI when we register user's click action and we can let different entities react on this input sequentially. This is also known as event routing.

1.1.2 Case study

For the purpose of my analysis, I talk about a very simple application that could be used by a car dealership. Imagine an information system that is used

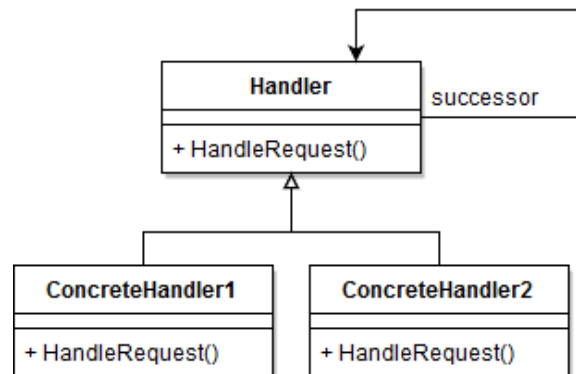


Figure 1.3: Chain of Responsibility pattern

by each branch of our dealership to monitor sales. In this system there is a dialog where we can see three edit boxes. A target number of cars to sell in a given month. This is set by the dealership headquarters. An actual number of sold cars, and the variance number calculated by the application. The system colors the variance number red if it is more than 10% below the target number and green if it is 10% or more above the set target. See figure 1.4.

The screenshot shows a window titled "Car Sales" with standard window controls (minimize, maximize, close). Inside the window, there are three labels on the left: "Target", "Actual", and "Variance". Each label is followed by a text input field. The "Target" field contains the number "30". The "Actual" field contains the number "20". The "Variance" field contains the number "10", which is displayed in red text. Below these fields is a grey button labeled "Save".

Figure 1.4: Example GUI

This simple dialog of an application is described from the points of view of the different architectures and patterns. The sections below are by no means a complete and exhaustive analysis of each pattern. They are an overview of the common principles of their variants and flavors. For some of those patterns it is enormously difficult to get the grasp of what they are supposed to present in their pure form. There are dozens of sources describing Model View Controller (MVC) pattern, yet they often do not present the same principles and ideas. Some of them I would not even consider an adaptation of MVC at all. Where

the footing was loose I used the work of Martin Fowler [5] as a reference.

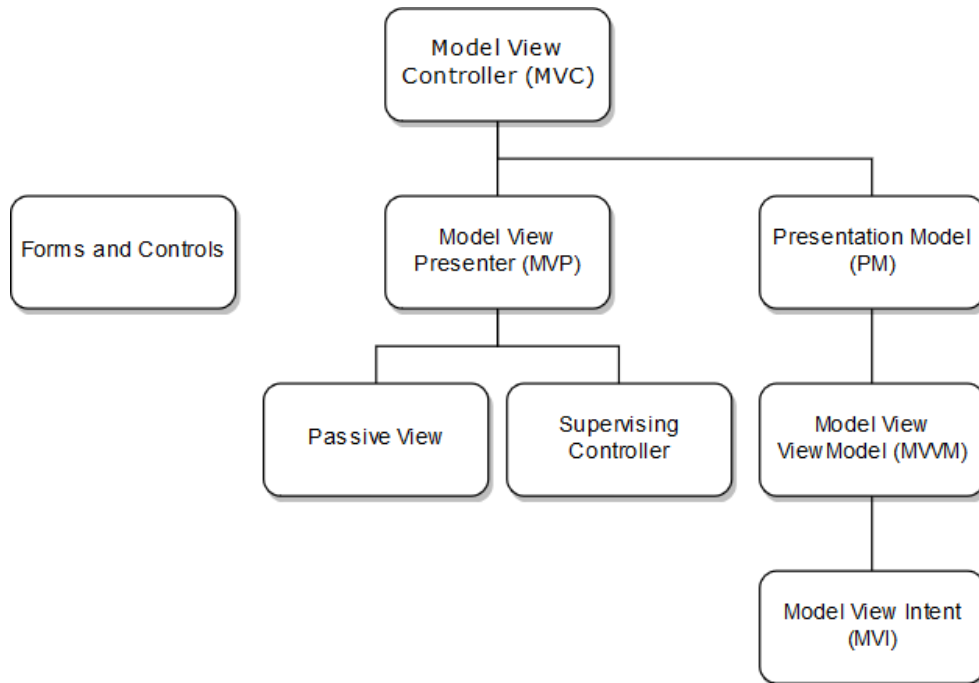


Figure 1.5: Presentation patterns

Before I dive into individual architectures I would like to provide an overview in the form of figure 1.5. The approaches are organized loosely by chronological order MVC being the oldest and MVI the latest addition. The links between are a take on showing their line of evolution as you will read later it is not exactly easy to always pinpoint the origins.

1.1.3 Forms and Controls

Starting with the simple and straight forward approach to GUI most encouraged by the server-client development in times of Visual Basic and Delphi, so think 90's, is Forms and Controls. This approach does not actually have a coined name, but I am sticking with what Martin Fowler came up *Forms and Controls*[5].

The basic building blocks of this approach are custom made forms out of generic reusable controls. A control is an element of the GUI to give some examples a TextBox, Button, Label and so on. Most of the GUI frameworks come with a bunch of premade controls that could be used to populate a specific form. If the provided controls are not enough there is still the option to implement our own control, but even in this case think about the control as a generic and reusable element for several forms or even applications.

The form fulfills two main roles:

- Screen layout - the arrangement of the controls and the hierarchical structure between them.
- Form logic - behaviour that is difficult to get out of controls alone, usually some form state or shared metadata.

Most of the GUI frameworks come with a handy graphical editor that allows the developer to drag and drop controls on a precise place in a form. This is pleasing WYSIWYG¹ experience, but it has its drawbacks as you can imagine, especially today with strong demand for responsive design. The controls display data, in case of the example GUI data about car sales, but data always comes from somewhere. For a car dealership application the data are most likely from an SQL database, but that is definitely not the only copy of the data involved here so lets have a look. There are three copies of our data:

- Record State - This is the data directly in SQL database. The database may be shared and visible to multiple users and application simultaneously.
- Session State - This is an in-memory copy of the Record State data stored in a Record Set, figure 1.6. Server-client environments usually support this with tools to make things easy (ADO.NET for example). The Session state data are private for the running application, which can make changes to it as it pleases. To publish the data to Record set a save or commit is needed and a subsequent merge of the data. I am not going any deeper into this problem as it is far from GUI and it is a chapter on its own.
- Screen State - Last copy of the data is in the GUI elements. This data is being displayed on the screen, that is where the name come from. For every GUI it is very important and interesting how the Session state and Screen state are synchronized.

1.1.3.1 Screen and Session states synchronization

The easiest way for synchronization between Screen state and Session state is *Data Binding*. The idea is that any change to the data in form or controls is propagated to the underlying Record Set and any change to the set means

¹What you see is what you get. Approach used not only for designing user interfaces. The author can see the result of his work directly. For example MS Office Word uses this approach with documents.

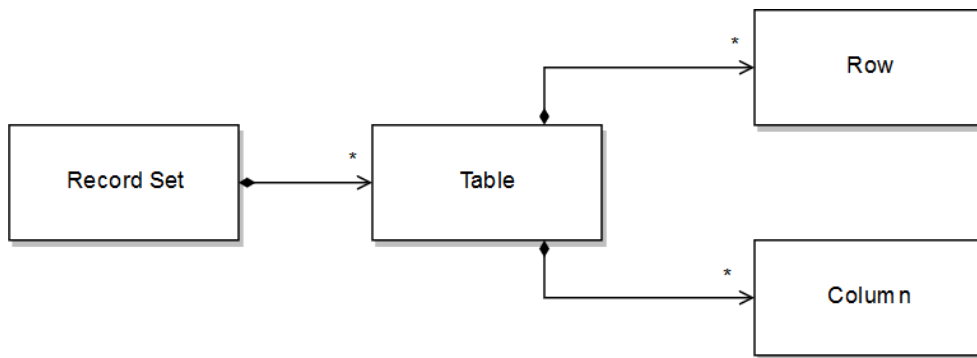


Figure 1.6: Record Set

a direct update of the Screen state. So a user action modifying the edit box **Actual** updates the correct cell in the Record Set table.

There are two things to keep an eye on with *Data Bindings*. One is a cycle of updates, when a change to the control propagates to Record Set, which changes the control, which updates the Record Set... To break this loop we can set the binding so that is not strictly bi-directional. We populate the screen when it is opened and any change done to the controls propagates to Session state. It is unusual for the Record Set to be updated directly as the screen is opened so we can omit the update the other way.

This behaviour is usually covered by the frameworks supporting this GUI approach. Setting control's property binds it to a specific column of a table from a record set. In reality this means setting the column name in a property editor for a given control.

The other issue with *Data Binding* is inherited from the fact that it binds to the Record Set. The variance is calculated by the GUI and is not part of any table. Most of the time there is some logic that won't fit into controls and is inherent to the application. In such cases the logic's place is in the form which is application specific. In order to make this work we need the generic text box of **Actual** to call some specific routine in the application form.

There is not just one solution to this problem, but perhaps one of the most used is events. Each control is equipped with a list of events that can be raised and to which anyone subscribe to and react. Essentially this means using the *Observer pattern* and letting the form observe its controls. Each framework solving the problem this way provides some mechanism how to invoke a routine for a raised event and also a place where that routine should be implemented.

Once the routine has control it can do what it needs. It can do some logic needed to populate some fields it can pull additional data from Record Set all sorts of interactions. It is also important to say that this mechanism can work alone without *Data Binding*. It just means implementing every single interaction via the handlers to events including initial loading and final saving of the Screen state to Session state on clicking a save button for example.

1.1.3.2 Example

Lets walk through a scenario assuming *Data Binding* is in place. A user opens our Car Sales form. As the form is being initialized it subscribes its own event handler method `OnActualTextChanged` to the event raised by `Actual` control when the text changes. Also it subscribes to other events, but lets keep this example simple. When the user sets the value for `Actual` the edit box raises an event for the change of text. Through the mechanism of the framework the registered handler is executed. This method gets the values from the `Target` and `Actual` fields does the subtraction and fills in the `Variance` field. It also decides on the color of the text displayed.

1.1.3.3 Summary

The Forms and Controls approach is the simplest to grasp and very straight forward. The developer writes application specific forms out of generic controls. The form defines the layout and structure of the GUI it also observes its controls and can react to interesting events raised by them. Simple data edits are usually handled by *Data Binding* complex changes on the other hand are implemented by event handlers in the form.

1.1.4 Model View Controller

This GUI model is probably the most referenced one from all mentioned here. It is also the most misrepresented one. The main reason for this is that MVC needs to be adapted for GUIs of today and every author refers to their own flavor by the same abbreviation.

The origin of the MVC pattern comes from Smalltalk-80 it is in fact one of the first attempts to do any sort of GUI architecture [6]. I am not going into all the details of monochromatic graphical system created in the later 70's, but many of the concepts first introduced here are still well used today and that is what I want to focus on.

In the core of MVC is the great idea of *Separated Presentation*. Introducing the concept of isolated domain objects that model our real world as the business logic objects, and presentation objects that are solely for the use of the GUI elements on screen. The domain objects should be completely

independent of any presentation, they should be able to support multiple presentation possibly even concurrently. This approach was heavily connected to the Unix culture allowing for one underlying program that could have GUI and command-line interface as well.

In MVC the domain objects are referred to as Model. Model is completely ignorant of any GUI. The MVC is also assuming actual domain model objects not a record set. This simply reflects the fact that unlike Forms and Controls, that were intended to manipulate records in a database, MVC was initially intended for Smalltalk a purely object oriented environment.

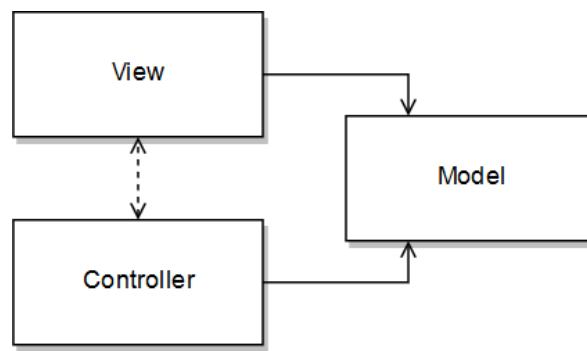


Figure 1.7: MVC pattern

The presentation part of MVC is constructed out of two elements: View and Controller. The Controller's job is to react to user input and figure out what to do. The View's job is to present the Model's data to the user. View and Controller have a direct reference to the Model. They also have reference to each other, but this connection is purposefully used as little as possible

I should mention that there are many Controller-View pairs. Each control on screen has its pair, and the screen itself has a pair too. So the first step in reacting to user input is deciding what controller should be executing.

Similar to other environments Smalltalk MVC expects developers want to reuse GUI controls, in this context it means reusing the general Controller-View class pairs and plugging in application specific behaviour. There would also be a higher level View representing the whole screen and describing the layout of the of the lower level controls, in par with form from Forms and Controls. Unlike the form, however, there are no events raised by controls and no event handlers in the higher level View. All information is conveyed through the Model.

1.1.4.1 Example

Once again lets have a look at our simple Car Sales dialog and how MVC would work with it. On the screen initialization we would have Controller-View pairs created for each of the fields present and one more for the enclosing window. We would have a Model consisting of our domain objects, values of **Target**, **Actual** and **Variance**. The developer decides what Controllers and Views do register as observers to their relevant object of interest. This is mostly implicit in this simple example. When a user changes the value of **Actual** the controller handles the user input and passes the value to the Model. As the value of the **Actual** object is changed in notifies its observers to give them chance to react. The **Actual** View updates its value so that on the screen the user sees what he/she typed. As the **Actual** domain object was changed the Model recalculated the value of **Variance** object and this object notifies its observers resulting in the **Variance** View getting updated.

There are some wrinkles to the sequence I described like what about the **Variance** color? I will get to that in a moment.

1.1.4.2 Flow vs. Observer Synchronization

MVC works quite a bit differently than the Forms and Controls approach there is no interactions from View or Controller to any other View or Controller, no events, no entity handling the application visual logic. When the **Actual** Controller changes the value in the Model it does not update its View directly it lets the *Observer pattern* take over. These are what M. Fowler calls *Flow Synchronization* and *Observer Synchronization*.

Flow Synchronization means the element that is changing directly updates all those who need to be updated. This is a heavy handed approach for a rich user interfaces. The consequences are even more apparent if we take *Data Binding* out of the system. Without it every interaction of Session state and Screen state data has to be done manually by the developer. Typically this means on opening a screen, hitting save button and other interesting point in the application flow.

Observer Synchronization makes this easier there is no form that checks everything and polices that the dependencies of a screen, but as a consequence it makes Controllers completely oblivious to any other widget needs. This is very useful especially in GUIs where are multiple screens showing the same data, like graphs and tables. Imagine dealing with forms synchronization that would need to check what other forms are open to propagate changes. So in this case the *Observer pattern* is like a blessing, when it is not a blessing at all is when you want to read the code and find out what is going on. The inherent obfuscation of the *Observer pattern* functionality means that what is

really going on can be only seen during a debug time. This definitely needs some getting used to.

I promised I would return to the **Variance** color property and here I am. I would also like to take a step back and look at the **Variance** value as well. I admit I skipped a little bit that in MVC we have the value of **Variance** in the Model and it makes perfect sense. The **Variance** is a value that is viable without any presentation in place it does not need to be in the data source be it an SQL database or not, we can always calculate it. For the color, however, MVC does not have a neat place. It does not fit into a domain logic. What can be argued as fitting into the domain logic are the rules for the color the 10% above and below **Target** value. The mapping from the intervals to colors definitely not domain logic; it is view logic.

This problem was not unknown to the Smalltalk engineers. Here I admit that the Model won't be pure domain objects and domain logic and infiltrate the necessary view logic requirements to the Model. This is definitely not ideal, but it is quite easy and straight forward to do. The downside is Model with mixed responsibilities. To deal with this problem properly we will need to shift the architecture a little bit.

For the synchronization we have a choice. Either mimic what Forms and Controls does. Register the screen View as an observer for the **Variance** value and set its color and behave like the enclosing authoritative entity. This adds another *Observer pattern* obfuscated behaviour and it can get pretty messy with bigger GUI. We could also derive another Controller-View pair that can handle color and hook on directly. This View would have internal mapping for the colors based on the boundaries that could be described in Model. This can get out of hand as well with sub-classing for all sorts of controls. Also it heavily depends on how well is certain framework developed and how much it allows for easy sub-classing. For Smalltalk it is really easy.

Lastly, there is an option to create another Model intended for the screen. A place where the visual logic could be. Any methods that are the same as in the domain Model would be delegated to it, but it can add methods that are fulfilling the needs of the GUI, like our **Variance** color. This was popularized by the Smalltalk framework VisualWorks and became known as *Application Model*. I will once again borrow a term from Mr. Fowler and use the term Presentation Model (PM), which is more abstract and I dare to say that *Application Model* is just adaptation of Presentation Model.

The Presentation Model solves the problem of visual logic place very nicely. It also adds another benefit. It allows to keep view state. The information about our interaction with the Model, not the state of the Model. Behaviour

like enabling save button only when something changed etc.

1.1.4.3 Summary

The origin of MVC comes from Smalltalk-80 and can be credited for the idea of *Separated Presentation*. This means that we have isolated presentation layer, Controller-View pair, and domain, the Model. The Controls have each their own pair, the Controller handles user input and View presents information. The communication is done through the Model as much as possible. Lastly we have the great contribution of *Observer Synchronization*, the use of *Observer pattern* to indirectly update controls.

1.1.5 Model View Presenter

The term Model View Presenter (MVP) comes from 90's when it appeared in a paper by M. Potel of IBM [7]. To describe MVP principles it is best to think about what we already know. MVP is an approach trying to lift the best out of both Forms and Controls, and Model View Controller architectures. Taking the direct approach of reusable widgets out of Forms and Controls and combining it with the *Separated Presentation* and isolated domain model of MVC. It tops it off with one more requirement GUI testing.

The paper on MVP describes View as a structure of widgets, like controls on a form, removing all the pairing. We do not use Controllers in the sense we had in MVC. All the interaction to user input is handled by a Presenter that decides what to do. Yes technically the View has the initial entry point for user actions, but they just delegate the control to Presenter. Potel describes scenario when Presenter interacts with model using commands and selections – this is useful idea as it enhances testability and allows for undo/redo functionality. As the Model is updated by the Presenter the View is updated using the *Observer Synchronization* where possible. If there are actions that are too complex the Presenter gets involved and sets the View directly. This is what become known as *Supervising Controller*.

Here I feel the need to explain why is the naming so confusing. Mr. Potel did a good job of defining the term Presenter and keeps it clean in his paper. Later adaptations unfortunately not so much. So you can find Controller when describing MVP pattern and it means Presenter. There is a solid case for calling it Controller as it handles user input. I try to do my best to keep the terminology separated, but some terms like *Supervising Controller* or even some frameworks like ASP.NET MVC do not share this strict differentiation.

1.1.5.1 Passive View

Removing all *Data Binding*, *Observer Synchronization* out of the View-Model relation, we get *Passive View*. The view is just plane structure of widgets with no logic and no way to reach data on its own. The Presenter is completely in charge of everything. It handles user inputs, modifies Model and loads data into View. If the center point of MVC was Model, here it is the Presenter, figure 1.8.

Having all logic and control in Presenter allows for simple View and a simple interface between those two. The benefit of this is that View can be replaced for testing with any test double, like a View Stab for example.

1.1.5.2 Model View Controller/Presenter Web adaptation

The MVC was not really developed when internet was around; hence, there is adaptation needed if we want to use it with web applications. One of the most well known usages are Java Server Pages Model 2 – MVC [8] and ASP.NET MVC [9]. The reality is those architectures are essentially MVP, usually *Passive View*, with another Front Controller that decides what server side Controller to reach. The Frameworks also add routing and filtering and all sorts of other functionality, some of them provide data binding of different variety.

1.1.5.3 Example

Looking at MVP (*Supervising Controller*), startup looks similar to Forms and Controls we have Presenter subscribing its handlers to events of widgets. When a user updates the text in **Actual** field, event is raised, handled by Presenter and the Model value is updated. Model recalculates **Variance** value as well. At this point the *Observer pattern* kicks in and View is updated. The last part of setting the color for **Variance** is done by Presenter, it gets the category of **Variance** and sets the color accordingly.

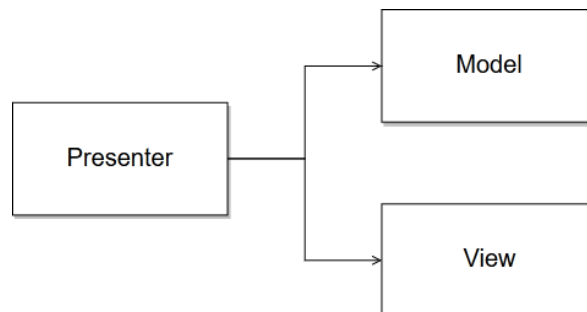


Figure 1.8: Passive View variation of MVP

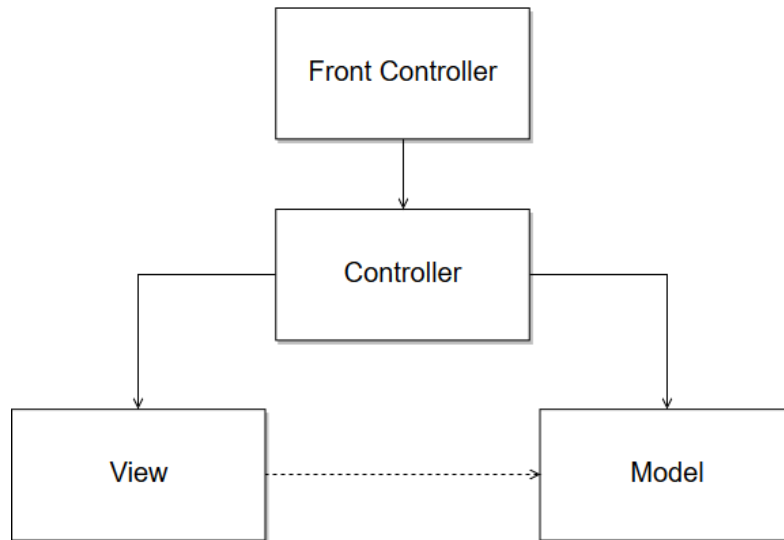


Figure 1.9: MVP/MVC pattern for Web

1.1.5.4 Summary

The Presenter is the pivotal point of this pattern. It handles user input and conveys it to Model. It deals with complex GUI settings or in case of *Passive View* is in charge of setting data to View completely. This allows for reusable widgets placed in the View, it also allows for testing.

Comparing MVP to the previously described architectures:

- Forms and Controls – With MVP we have the *Observer Synchronization* and even though the we can access widgets directly it should not be the first approach to use.
- MVC – Instead of Controller-View pairs we have widgets that pass interactions to Presenter. It is also important to say that usually there is one Presenter per form and not per widgets.

1.1.6 Presentation Model

As we have seen with MVC and MVP rich GUIs bring some problems to presentation layer. Two of the bigger issues are where to put view logic and

where to put View's state caused by user interaction. Modifying widgets directly encourages writing presentation logic into the View. The Presentation Model (PM), presented by M. Fowler [10], strives to remedy this. It aspires to be an abstraction of the view. Coordinating with the Model of domain layer and either handling the state of view completely or at least synchronizing very often.

The PM is essentially a self-contained class representing all any GUI framework would need to know or use in order to render controls. Multiple views can utilize a single Presentation Model, but each View should refer to a single one. Composition is possible and a Presentation model may contain several child PMs, but each control will again refer to a specific one.

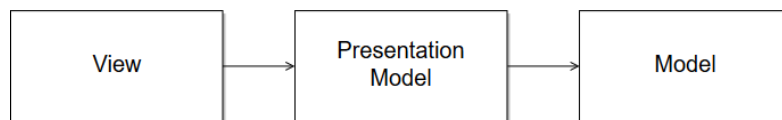


Figure 1.10: Presentation Model

To do this PM will have data fields for all information for the view and that means not just the contents of controls, but also information about their visibility, if they are enabled or highlighted etc. It does not mean that the PM has this fields for every control, if the property is never used it can be omitted, but if it is needed it is present in the PM.

The drawback of Presentation Model comes with tight synchronization. Suddenly there is a need for synchronization not just on the level of screens or components, but lower – field or key level synchronization. This opens possibility for fine-grained synchronization, M. Fowler discourages from it as it brings a lot of complication, especially when things do not work as intended. I would say it depends on the nature of specific project, but coarse-grained synchronization in the form of syncing whole stat of View with Presentation Model is definitely simpler.

Than there is the question of where to put synchronization code. Choosing Presentation Model means we can test the synchronization, which should already be a pretty simple code (coarse-grained sync for sure), but we drag a reference to the GUI framework into PM, which we have to keep in mind. On the other hand we can choose the View, this is a natural place for it as the PM can be oblivious to the View completely. If we ever feel the need to write tests for anything in the View objects it might signal, that we need to rethink how this synchronization works and what codes lives where.

1.1.6.1 Example

Returning to our simple Car Sales dialog. On startup a Presentation model would be created as a layer between View and Model. It loads data from Model **Target**, **Actual** and **Variance** value and probably also the boundaries for setting color in percentages. It decides on the status of **Variance** and provide a property **Variance** color. When a user changes value for **Actual** it reacts to it updating its **Actual** value, recalculating **Variance** and updating its property for **Variance** color. View then observers these changes and updates itself. Note that to this point no changes have been propagated to Model. If the user for example left the field of **Actual** empty, the PM could be extended to have a property for the save button and set it to be disabled in such case. Only on valid value for **Actual** and click on save button we synchronize Presentation Model and Model.

1.1.6.2 Summary

Presentation model steps in to provide a place for visual logic and View state. Widgets do not observe domain Model instead they observe Presentation Model. It allows for rich and complex GUIs. On the other hand, it calls for tighter synchronization with View making heavy use of *Observer patter*, which could be alleviated by frameworks.

1.1.7 Model View ViewModel

Mode View ViewModel (MVVM) architecture first appeared in a Microsoft blog post by J. Gossman in 2005 [11]. To be blunt it is in its core an implementation of Presentation Model. It was a model directly developed for .NET use by Windows Presentation Foundation (WPF) and then used Silverlight. Even though the abstract idea is identical to Fowler's PM it brings more to the table. The View which is declaratively described using a modified XML, Extensible Application Markup Language (XAML), which sets the visual appearance. It is expected that this work is done by a designer not necessarily a developer.

MVVM also builds on a strong *Data Binding* between View and ViewModel. This is handled by the framework and the problem of tight synchronization is solved under the hood for anyone using this technology. MVVM also encourages use of commands in ViewModel that are triggered by GUI events, this as was already mentioned above is great for re-usability and testing.

MVVM grew outside of .NET ecosystem and since MVVM is linked to Microsoft's implementation, the idea is also refered to as Model View Binder. Java has its implementation, ZK framework[12], granted it does not uses Mi-

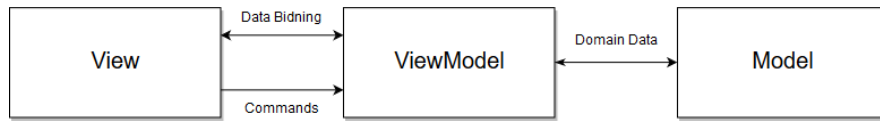


Figure 1.11: MVVM Pattern

Microsoft's XAML, but ZK User Interface Markup Language (ZUML). Similar fairly popular implementation is in JavaScript – KnockoutJS.

Model View ViewModel does not bring anything completely new in terms of architecture it is more of an extended implementation of the Presentation Model.

1.1.8 Model View Intent

Latest evolution on the MVVM pattern. It was first specified by André Medeiros in his JavaScript framework Cycle.js [13]. It was readily adopted by Android developers and Kotlin environment, where it is solving some cumbersome problems in mobile GUIs.

The issues are mutability of ViewModel which gets misused by developers. Coupling between View and ViewModel in the form of tight synchronization and finally asynchronous events that are present in web and mobile applications more than on desktop.

To answer this problems MVI is building on the concept of *Reactive programming* from functional programming. The term is spread to reactive applications and reactive frameworks. Adding ideas of states and with it related immutability and unidirectional flow. With the help of these terms I will try to explain how MVI is supposed to work. The core principle can be described in terms of mathematical formula as follows:

$$view(model(intent()))$$

User acts on the GUI and exhibits intents. These intents are processed by model and based on them view is rendered with the results. Model is where all the magic happens so let's look closer with the help of figure 1.12.

User actions are listened to by the View and passed to the Model as intents. Note that the Model here is for the purposes of GUI, meaning ViewModel or Presenter in previous patterns as if the confusion was not sufficient already. This time the Model consumes intents; acts based on them it may work with

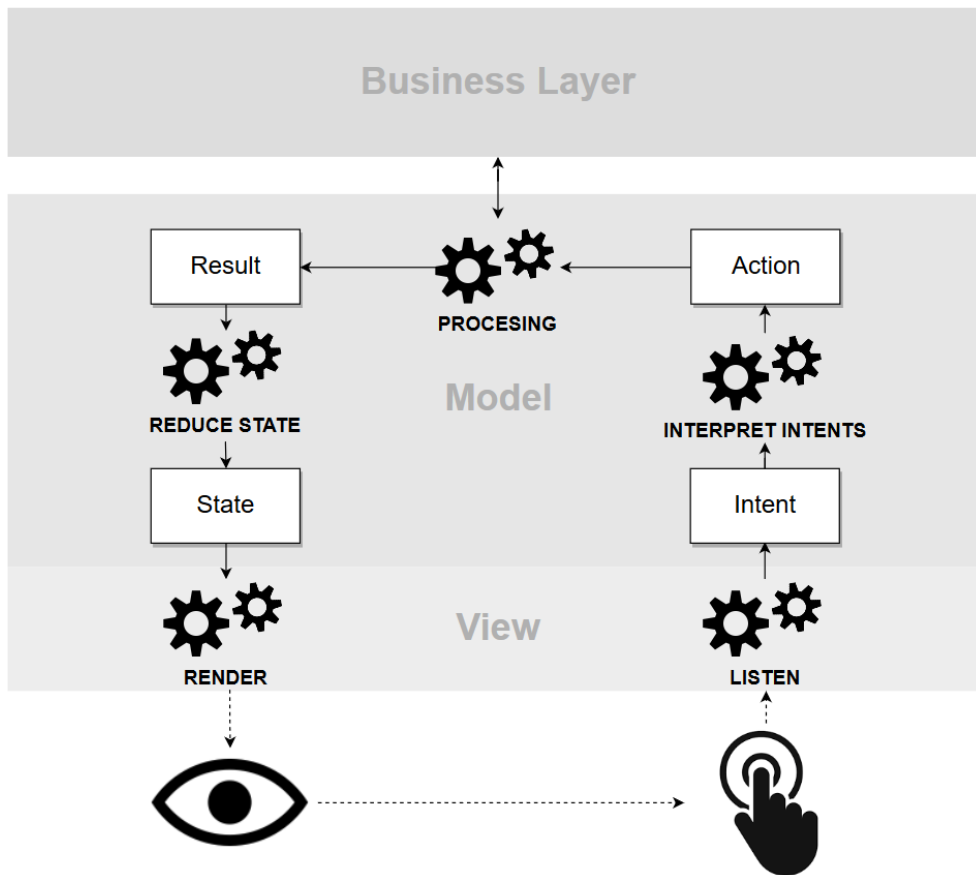


Figure 1.12: MVI Pattern

business layer of the application. As an intermediate product it creates result caused by user intent. Lastly it goes through state reduction. In this step it combines current State of the GUI and the results and produces new instance of State. This state is passed to View and it is a complete description for rendering. The loop is closed when View renders this State and we wait for the next user's action. The flow of information goes only this way, there are also no side effects, the only place where information is held and exchanged is during processing with business layer.

This is a pretty complex setup, but it allows for multiple asynchronous actions ultimately affecting the View, without problems. This is achieved thanks to the immutable State, that is created from the state reduction step. The problem of tight coupling of View and Model is minimized as the contracts are very simple.

The View is only capable of rendering State and producing Intents. The Model

is ultimately capable of consuming Intents and producing a new State. The connection here is realized just by *Observer pattern*.

1.1.8.1 Example

Bringing MVI to our case study of Car dealership dialog. On start up the complex Model is created. It loads necessary data from business layer and creates the initial State. The View is also initialized and registered as observer to Model's State. Start up is done when View renders this initial State.

Now as the user updates value of **Actual** Intent is produced and observed by Model. The flow is started and the Intent is to update value of **Actual**. This is interpreted and Actions has to be taken to: update the value of **Actual** in business layer and get the **Variance** a check for range is done, color is decided. This set of changes is our Result and it gets merged with the current State during state reduction. Finally we get new State with all the right properties and View's *Observer pattern* kicks in and rendering happens.

1.1.8.2 Summary

Overall this is a complex setup for a presentation layer, but it addresses problems in web and mobile GUIs, that are hard to solve in other architectural patterns. The fact that this approach works with asynchronous actions and asynchronous streams where data trickle by pieces makes it very powerful. Ultimately developer has to decide if the trade-off for this complexity is worth it.

1.2 Methodology

In order to evaluate GUI architectures and technologies from the evolvability point of view, I need some framing and theory to define what characteristics of the systems are interesting. I present two views; Normalized System Theory [14] and Evolutionary Architectures [15].

1.2.1 Normalized Systems Theory

The Normalized Systems Theory (NST) is an effort to design and engineer software systems that are proven to be evolvable. It started from observation of the software engineering landscape and realization that many projects do not reach their goals, don't meet deadlines, and/or go over budget. All that while the pressure is rising to be more agile and nimble supporting business, while adopting latest technologies, which leads to often implementing similar functionality over and over again. Manny Lehman's law of Increasing Complexity captures this reality stating that:

“As an evolving program is continually changed, its complexity, reflecting deteriorating structure, increases unless work is done to maintain or reduce it.”

—Manny Lehman, 1980

This law implies that software systems are growing in size and complexity while also decreasing in readability, maintainability as new features and requests are added to them. This leads to a complex architectures, dropping quality higher cost of operation until the whole system breaks and/or stops being profitable. This is inline with many industry best practices to fight the *bit rot* of applications and work on constant improving.

Normalized System Theory assumes that software architectures should be able to evolve over time and accommodate change. The NST defines rules that has to be followed in order to avoid combinatorial explosions in the impacts of changes to a software system. From the view point of NST the dream of constructing information systems based upon rational principles becomes possible:

“The user will expect families of routines to be constructed on rational principles so that families fit together as building blocks. In short, he should be able safely to regard components as black boxes.”

—Douglas McIlroy, 1968

This promise of McIlroy would allow for reuse and evolution of modules that would be the building blocks of our systems. Building a system would essentially mean picking desired modules, upgrading a system would mean a simple switch of system’s module for another. In reality we have to combat ever increasing complexity of our systems when implementing and even when we reach and master said complexity change to the system always occurs. The need is to master not static but dynamic evolvable modularity assembling a system. Ultimately NST aims to map one-to-one requirements to constructs thus promoting isolation and reuse in software systems, bringing the McIlroy’s dream to reality.

1.2.1.1 Systems theoretic stability

The NST builds on knowledge from other fields of engineering. Concepts like layered abstraction, *black boxes*, and hierarchic modularity allowed us to build

planes, space rocketry and micro processors. The starting point of NST is the systems theoretic stability. This system property means that a bounded input function results in bounded output values for an infinite time. In the world of software engineering this translates to the demand that a bounded set of changes results in a bounded amount of impacts to the system, even for an infinite time.

The infinite time assumes unlimited evolution of a system. That in turn means unlimited growth of a system increasing the number of primitives and dependencies between them up to infinity. They become unbounded. The demand for system stability says that bounded input, request for a change, has to result in bounded output. This forces the conclusion that the change cannot depend on the size of system and only on the nature of the change itself. Any impact on the system that is caused by the size of the system is called *combinatorial effect* and it is the root cause of instability from the evolvability point of view. This thought chain results in statement that any bounded change to a system results to a bounded impact that is independent of the size of the system and the point in time when applied.

1.2.1.2 Combinatorial Effect

The so called *combinatorial effect* is an unwanted impact on a system caused by a change that was not directly related and should not cause this impact. To be more explicit the *combinatorial effects* are undesired and sometimes hidden couplings and dependencies between modules, parts or primitives of a system increasing with it size. They are the consequences of integration of task, action and data entities and as current software constructs and methodologies do not pay much attention to them they are omnipresent.

1.2.1.3 Normalized Design Theorems

In the NST book [14] the authors present four principles to support anticipated changes and avoid most of *combinatorial effects*. These principles are independent of programming and modeling languages.

1.2.1.3.1 Separation of Concerns This theorem expresses the need to isolate tasks. Meaning that each function should only be implementing a single task and therefore be impacted by a single change driver. This in reality means avoiding duplication of code and implementing single purpose functions. Essentially bringing up the submodular tasks to the modular level. This theorem has many real world manifestations in the form of integration bus, multi-tier architectures, and external workflows to say the least.

1.2.1.3.2 Data Version Transparency Data Version transparency implies that in a way that is resilient to a change to data elements. This means that change to a data in the form of adding new values that were not previously needed does not effect currently implemented components and functions. This theorem voices the need for encapsulation in order to avoid *combinatorial effect*. In the NST book this is expressed as *Stamp coupling* passing data structures between modules instead of each parameter separately, called *Data coupling*.

1.2.1.3.3 Action Version Transparency This theorem is concerned with the upgradeability of task implementation – processing functions. The fact that there is a new version of task implementation must not break the system and not only that calling the new version should be seamless, without any additional changes, therefore, avoiding *combinatorial effects*. This calls for encapsulation of action entities and sharing common interface. This is seen in practice with object polymorphism, wrapping and Interface Definition Languages (IDL) such as Microsoft’s COM for example.

1.2.1.3.4 Separation of States Separation of States calls for state keeping for every action or step in a workflow. This results in an asynchronous and stateful workflow, where each task is atomic and returns action state that guides the steps of the workflow. This is once again combating the problem of *combinatorial effects* that emerge from synchronous calling pipelines that are natural to object-oriented systems. In order to realize this theorem it becomes apparent that workflow has to be separated in its own entity as well.

1.2.1.4 Summary

Authors of NST laid down a very solid foundation for their effort to reach for the McIlroy’s dream. Following this theory we know that to build an evolvable architecture we need hierarchic modularity and ideally zero *combinatorial effects*. This means identifying our change drivers and isolating them to their own entities. NST provides the theorems to follow in order to avoid many *combinatorial effects* and it is very clear that current programming principles are not enough to do so. However, all the principles are in line with well known heuristics of today, which certainly suggest the industry is noticing the problems.

1.2.2 Evolutionary Architectures

Evolutionary Architecture (EA) is a term coined in the book *Building Evolutionary Architectures, support constant change* by Neal Ford, Rebecca Parsons and Patrick Kua in 2017 [15]. The idea was first sparked at O’Reilly hosted

Software Architecture Conference. Here a lot of the speakers talked about microservices and the disruption it caused.

Thanks to progress in DevOps, Continuous Integration and Delivery, and containers like Docker a shift in the big complicated software systems became possible. Deployments could be made small and rapid. Microservices took over the architectures and instead of splitting systems by the physical layers, split by functions were possible. This also changed the notion of “Architectural changes are hard” and allowed architecture that is designed to accommodate change. It should hold true that to replacing one microservice for another should be as easy as switching Lego bricks. By definition [15, p.6]:

An evolutionary architecture supports guided, incremental change across multiple dimensions.

In order to fulfill this definition Evolutionary architectures the authors propose several useful characteristics. There are also described principles directing us in the way towards those characteristics. All of this is based on heuristics distilled from the industry proven by the successful projects and ensured by experts.

1.2.2.1 Characteristics

1.2.2.1.1 Modularity and Coupling To limit breaking changes it greatly helps to lock functionality into models that are standalone. The other important part is coupling that needs to be kept in check. The least evolvable architecture is the *Big Ball of Mud* where everything is one huge module connected to almost every other entity in that big ball. We can see that things improved with layer architectures, but with microservices and container isolation it can be finally truly exploited.

Evolutionary architectures show high modularity with very limited coupling to promote ease of change.

1.2.2.1.2 Organization around business As already mentioned microservices changed the how systems are deployed. Each a small deployment designed as a service offering functionality for the rest of the system. So modules of Evolutionary architectures are inspired by business needs not technical ones.

1.2.2.1.3 Experimenting Evolutionary Architectures allow for things like A/B testing and Canary releases. Simply by exchanging or orchestrating modules to allow for different outcomes. This allows for gradual replacement of

functionality and eventually it removes speculation out of backlog issues and allows for testing hypotheses in the real world.

1.2.2.2 Principles



Figure 1.13: Example of Evolutionary architecture’s fitness function fit

1.2.2.2.1 Fitness Function Fitness functions is a term borrowed from evolutionary computation techniques like genetic algorithms, but it is a concept very useful for evolutionary architectures. The idea is that each system has a list of “-ilities” that are essential for it. Usability, security, accessibility, traceability, fault tolerance, low latency, testability and many many more. The authors separate these into different categories, but the important message is that we as architects and developers should pay attention to them. Identify them as soon as possible, rate them based on how important they are for a given project, example in figure 1.13, and implement gatekeepers into production pipelines. This only extends the Continuous Delivery principles with additional checks that are placed on systems and modules. This could be for example requirements for code coverage over 90% and results from static code analysis meeting a certain threshold. Load testing passing the requirement that all web requests are served under 10 seconds even when network latency is present. GDPR² compliance showing logs of how personal data are

²The General Data Protection Regulation is a regulation in EU law on data protection and privacy for all individuals within the European Union.

handled and stored, and so on. These observations make it possible to keep an eye on the state of the architecture and make informed decisions to future changes.

1.2.2.2.2 Bring the pain forward This principle is also not entirely new. This is based on the idea of technical debt that does not behave linearly, but instead as projects grow it increases exponentially. Solution to this problem gave us Continuous Integration since integration was/is one of the headaches of development process. Steps in development that are complicated, time consuming and are therefore not done very often need to be automated where possible. Things that need close attention database migrations, code refactoring should be done as soon as possible. This allows for the rapid builds and deployments and only thanks to this principle the fitness functions are possible. The authors advice to identify these issues and remove the pain early before interest accumulates.

1.2.2.2.3 Last Responsible Moment I would consider this principle as an extension to the well know YAGNI (You ain't gonna need it) heuristic. In traditional architectures many subsystems, technology stacks and tools are chosen very early or even before coding entirely. The authors weight the cost of incorrect early decision against delayed decision benefiting from additional information gained during the time difference and argue for the later. Of course this decision to delay has its own price a potential re-work, that can be soften by some abstraction, but here YAGNI strikes again. The benefit is that this cost ought to be significantly smaller than for example inappropriate messaging system, which could slow down the development in many other areas and eventually be marked as tech debt and finally replaced much later in the life of the project. With this in mind a natural question presents itself. When is the last responsible moment for a certain decision? Here the fitness function provides some help. Decisions that have bigger impact on the whole system or are of significant importance should be made earlier. The core of the idea is to wait as much as possible, but don't stall.

1.2.2.3 Conway's Law

In order to bring Evolutionary Architectures into the real world we need to create microservices. I would argue that we could talk even about modules, but the book mentions microservices so I respect that. But in order to make that happen we cannot have a company divided along the knowledge expertise. This is voiced strongly throughout the book and presented by the Conway's Law [16]:

“Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations.”

—Melvin E. Conway, 1967

The lesson learned here is to inverse this law. If we want to build standalone independent microservices we need to disperse the experience and build teams around projects and business functionalities. This aligns with agile approach to build teams of diverse members and makes a lot of sense given the context of Continuous delivery etc. mentioned above.

1.2.2.4 Summary

Evolutionary Architectures build on the advances of DevOps and Continuous Delivery. It advises architects not to dwell on static diagrams of the current architecture and accept the fact that one of the core building block of a successful architecture is its evolution and openness to change. The authors also remind us of the following fact. Architecture is abstract until operationalized. Meaning that we can't judge architecture as a diagram and actually not even after first implemented. To say an architecture is successful the system has to go through several upgrades and maybe even some breakthrough in some premises that were used to build it in the first place.

1.2.3 Summary

The two presented views on evolvability of architectures and systems approach the issue from different directions. The Normalized systems theory draws on parallels from different engineering areas. Prepares a solid theoretical foundation and then proceeds to set proven theorems that has to be followed in order to achieve evolvable software product. It build from the bottom up talking about small structures, classes and single functions in its examples.

The Evolutionary Architectures comes from top down perspective. It is based in knowledge gained through experience and time proven heuristics. It also draws on the largest concepts of enterprise architectures and the transformation done here with microservices. It presents characteristics that should be present in an evolvable architecture and provides principles on how to achieve them.

Even though these methodologies come from completely different angles they reach similar conclusion. And as of my understanding they are applicable simultaneously.

Goals revisited

Before I revisit my list of goals from the introduction of this thesis I would like to summarize few realizations resulting from the paradigms and methodologies overview.

As stated in section describing Forms and Controls, section 1.1.3, there are three data states we can talk about, but for the purpose of GUI architectures we can ignore the record state and just focus on screen and session states. As I did during the whole overview as no other architectural model even mentions this data state. This choice is respected in my following analysis as well and I won't talk about data persistence at all.

Overall in the Paradigms section, 1.1, several GUI architectures were presented. There are many different adaptations and implementations in various programming languages and frameworks. In order to evaluate any concrete implementation one have to look at what are the founding principles and best practices for chosen technology and if these rules are adhered to. If the implementation is unorderedly there is no real chance to review its potential for evolvability. So I take this requirement as a prerequisite for my analysis.

Since both mentioned methodologies present very similar ideas just from different angles I will use their common principles for evaluation of technologies I'll choose for analysis. The important principles from my point of view are:

- Modularity of the GUI system, its submodules and concepts
- Separation of Concerns for objects
- Minimal combinatorial effects caused by transition changes

2. GOALS REVISITED

I do not aspire to calculate the exact number of classes need to be refactored or give a formula for an estimate of man hours. The goal of my work is to explore evolvability of GUIs and if possible provide some advice on what to look for and what to avoid.

Revised all of these points I think I can revisit my list of goals and this time be more specific about each point.

- G1** Describe common GUI architectural patterns
- G2** Present trends and methodologies focusing on evolvability of architectures and choose principles for analysis
- G3** Choose two GUI framework/technologies and categorize their architectures
- G4** Evaluate chosen technologies based on NST and EA
- G5** Reason about approaches to convert presentation layer of an application
- G6** Implement example application and upgrade its GUI layer (use chosen technologies)
- G7** Specify which concepts are costly or limits the transition between chosen technologies
- G8** Summarize knowledge needed to ease transition between GUI technologies

Analysis

In this chapter, I choose two GUI technologies, categorize their architectures, review their principles, and concepts, evaluate their evolvability, and lastly reason about approaches of transitioning an application using one to using the other.

I am following the advice of the authors of NST, professors Herwig Mannaert and Jan Verelst. I am not trying to analyse as many technologies as I can. I am much more concerned about what steps should be taken if one considers evolving an application between technologies. Analysing a technology is a costly endeavour in the scope that I am facing. For those reasons I work with sample of two technologies. If one would like to expand this number the approach to the analysis and evaluation can be closely followed.

Now, without further ado let's have a look at the specific frameworks I have chosen to work with.

3.1 Frameworks

My choices of frameworks for the analysis is Windows Forms (WinForms) and Windows Presentation Foundation (WPF). This decision is very subjective. Part of the reasoning is that I do have some experience with these frameworks. Following that, I am currently member of a team that is faced with a business project migration between these technologies. The application is a computer assisted design system to model and analyse structural statics and it used all over the world. Both of these frameworks are mature and established within the .NET ecosystem and intended for desktop applications.

Some of you might consider these, well over decade old technologies, dead; but I would like to oppose that opinion. Google Trends shows both topics are still

searched for worldwide about half as much as were their peak search volume for each topic [17]. Quick search through GitHub shows several very active and highly rated repositories [18, 19] extending or consuming these technologies. Not to mention products like DevExpress [20] and Telerik [21] that set their business models on extending these Microsoft's frameworks and produce new versions for both. So I conclude that WPF and WinForms are alive and used a lot, and I can move towards their analysis.

I want to be very clear that I am not reviewing the myriad of different libraries, extensions, and frameworks built on top of what Microsoft provides as their GUI technologies. I limit myself to the documentation, principles and code samples provided by Microsoft, mostly what can be found on their documentation website [22, 23].

3.1.1 Windows Forms

WinForms [22] is the original GUI framework developed by Microsoft for .NET applications released together with the first version of .NET in 2002. It was not completely new concept at that time either. It builds on previous Microsoft Foundation Class Library written for C++. To give an idea of the time it was the time of Windows XP, spreading internet and nearly all applications were desktop applications. Even though it is now some 17 years old framework, it is by no means not a dead platform. WinForms still have support present today, like high DPI scaling coming in with .NET Framework 4.8. WinForms are also supported in the latest version of .NET Core 3.0 [24]. These decisions might be indicators that the framework is present in a lot of business critical application that companies invested a lot of time and money into and Microsoft does not want to let their corporate customers down. Speculations aside, let's have a look at the framework itself.

3.1.2 Intended use

As the name of the framework suggests, the original architectural model was Forms and Controls, section 1.1.3. So there are indeed forms and the documentation describes them as visual surface on which you display information to the user. There are also controls displaying data to user and raising events in case of interaction. There are many controls provided out of the box, but there is the option of custom controls as well. The events raised by controls are handled by event handler functions. Where is this event handling function and what it should do? Those are some very interesting questions. Let's look at one of the simplest form examples provided in the documentation³.

³I did remove several lines for brevity that is why you can see the triple dots. Nothing of importance was lost and I will remove lines from other code listings as well.


```
...
using System.Windows.Forms;

namespace FormWithButton
{
    public class Form1 : Form
    {
        public Button button1;
        public Form1()
        {
            button1 = new Button();
            button1.Size = new Size(40, 40);
            button1.Location = new Point(30, 30);
            button1.Text = "Click me";
            this.Controls.Add(button1);
            button1.Click += new EventHandler(button1_Click);
        }
        private void button1_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello World");
        }
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.Run(new Form1());
        }
    }
}
```

Listing 3.1: Simple form example

Modifying GUI elements

Even in this tiny example, listing 3.1, we can see several clues to how the framework is meant to be used. In the constructor of the `Form1` we see it sets up its control, `button1`, and fills its properties. It also hooks the `button1_Click` event handler function to the event of `button1` being clicked. Lastly we can read that this function is showing a message box dialog directly. Maybe you can say I am overprojecting here and the message box invocation is just to demonstrate some action, but in fact there are many more examples shown, where the code in event handler function modifies other elements directly or manipulates Session state data. For example here in figure 3.2, that is described in the documentation as follows:

This example uses the `Parent` property and the `Find` method of `Control` to set properties on the parent control of a `Button` and its `Form`. The example

3. ANALYSIS

assumes that a Button control named button1 is located within a GroupBox control. The example also assumes that the Click event of the Button control is connected to the event handler method defined in the example.

```
...
private void button1_Click(object sender,
    System.EventArgs e)
{
    // Get the control the Button control is located in.
    // In this case a GroupBox.
    Control control = button1.Parent;
    // Set the text and backcolor of the parent control.
    control.Text = "My Groupbox";
    control.BackColor = Color.Blue;
    // Get the form that the Button control is contained
    // within.
    Form myForm = button1.FindForm();
    // Set the text and color of the form containing
    // the Button.
    myForm.Text = "The Form of My Control";
    myForm.BackColor = Color.Red;
}
...
```

Listing 3.2: Event handler function

What does this all mean? The Forms and Controls approach is adhered to. The `Form1` is the all knowing entity that manipulates its controls and its logic dictates what the controls do and how they behave.

In another example, listing 3.3, we can see that there is *Data Binding* present. WinForms actually have two types of binding; simple one property binding and complex so-called *Data Source* binding. In listing 3.3 we can see the simple binding that can lead to a `DataSet` which represents data from database and the methods are best prepared for that scenario, but it can represent any data source as well.

The other kind of binding expects a `DataSource` an enumerable of objects. This binding is prepared for grids, lists, and combo boxes. In listing 3.4 the `listbox1` binds to `fonts` a collection of objects and displays their property `Name`. Again it works very well with representing data tables and the documentation mentions the intended use with ADO.NET framework for connection to SQL databases.

```
...
protected void BindControls()
{
    /* Create two Binding objects for the first two TextBox
    controls. The data-bound property for both controls
    is the Text property. The data source is a DataSet
    (ds). The data member is the "TableName.ColumnName"
    string. */

    text1.DataBindings.Add(new Binding("Text", ds,
        "customers.custName"));
    text2.DataBindings.Add(new Binding("Text", ds,
        "customers.custID"));
}
...
```

Listing 3.3: Simple binding example

Extensibility

This is still very much in line with the Forms and Controls architecture. Last topic I want to go briefly over is extensibility of controls in WinForms and how it is with code sharing and behavior inheriting.

The top most concept of WinForms classes created by its users is Component. Component is any class that either does business logic or background work, it does not have a GUI representation. This often means whole libraries and dependencies that are called from forms. Controls are Components that do have visual representation and can be rendered. This also means that Forms are Controls in addition to the more obvious ones like edit boxes, buttons, etc. Microsoft provides several traditional Controls with the framework itself. To add new options for Controls developers have two options. Either define a Custom Control with brand new visuals and possibly enhanced capabilities over the provided Controls, or create a composite control called User Control. User Control is built out of already made Controls and the visuals cannot be redefined here, what is possible is to introduced more complex behavior. For example the add/remove control that can be found in many applications even today, see figure 3.1.

There are of course many more implementation details and multiple approaches to achieve goals within the framework after all it is almost two decades old. This is reflected in the current state of its documentation, different fragments of original approaches are peaking between new layers and functionality that was added in later versions of .NET. It is impossible for me to uncover how the framework was originally released, but the origins have big ramifications

3. ANALYSIS

```
...
private TextBox textBox1;
private Button button1;
private ListBox listBox1;

private BindingSource binding1;
void Form1_Load(object sender, EventArgs e)
{
    listBox1 = new ListBox();
    textBox1 = new TextBox();
    binding1 = new BindingSource();
    ...
    MyFontList fonts = new MyFontList();

    for (int i = 0; i < FontFamily.Families.Length; i++)
    {
        if (FontFamily.Families[i]
            .IsStyleAvailable(FontStyle.Regular))
        {
            fonts.Add(new Font(FontFamily.Families[i], 11.0F,
                               FontStyle.Regular));
        }
    }

    binding1.DataSource = fonts;
    listBox1.DataSource = binding1;
    listBox1.DisplayMember = "Name";
}
...
```

Listing 3.4: Complex binding example

for the current state. I did my best to present what I consider relevant for the evaluation which follows.

3.1.2.1 Evaluation

Let's have a critical look at WinForms using the lens of NST and EA methodologies. Starting with Separation of Concerns; I dislike the `Form` class. It might be even worthy of the anti-pattern label “God class”. Listing its most obvious responsibilities:

- **Visual representation** – This might not be obvious at first sight, since Visual Studio provides form designer where developers can drag and drop controls, set their properties and even generate event handlers. In the end it is one class and one object that during initialization handles all

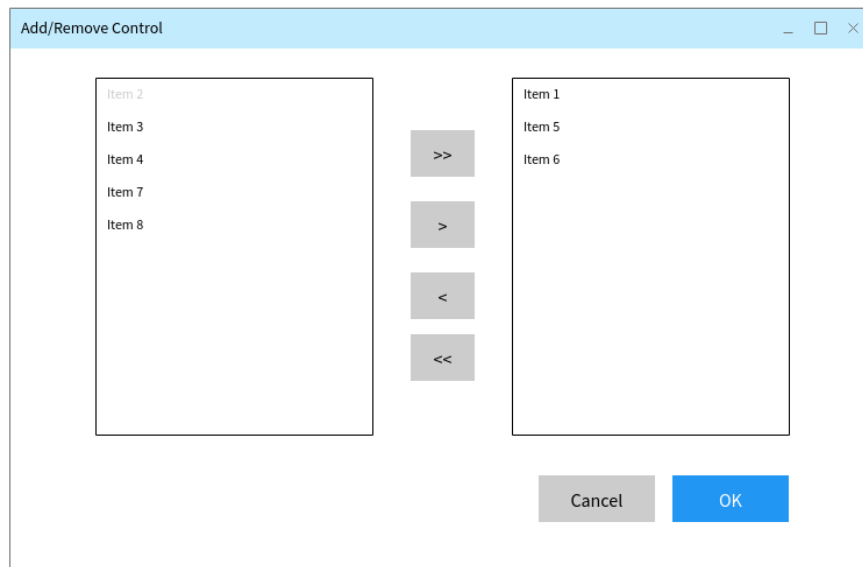


Figure 3.1: Example of Add/Remove User Control

this complexity. The **Form** takes care of the layout of controls and their properties population.

- **Data Binding** – In case of simple binding, the **Binding** object is created during **Form** initialization and the **Form** knows what data binds to what control's properties. If we consider complex binding, it is basically the same with the option that the **DataSource** might be instantiated during the form load or other startup event.
- **User input** – Controls raise events and those can be received and handled, but again it is in the realm of the form. Form designer from Visual Studio generates these event handlers directly to the **Form** class leading developers to write the event handling logic there. This can expand to calling into business Components and down the rabbit hole. Potentially this design can lead to freezing the application waiting for some back-end response.

This is a staggering number and scope of responsibilities gathered in one class. With so many reasons to change and the tight coupling present between components and methods fulfilling the different requirements, which are in the **Form**, the *Combinatorial effects* are present all over the place. Change to any control means change to the initialization of the form, its data binding and event handlers as well. This propagates to every form that is using the control. This ripple effect is also present from the other side. If data changes all the different bindings and assignments to controls have to change too.

3. ANALYSIS

There is also no notion of view state. Only the controls hold data that they are displaying, which again mixes responsibilities and also hinders any possibility to share the state across multiple views.

Lastly there is the modularity of the architecture or should I say lack of it. Any time we would like to reuse an existing form and add something to it, or we would like to adjust few controls the framework leads us to creating a new form entirely. Composition exists only on the level of whole Controls and even tweaking an existing control leads to creating a completely new custom one.

3.1.2.2 Summary

Windows Forms was intended as implementation of the *Form and Controls* architectural pattern and it was built with the needs of its time in mind. It is very simple to start a project in it and develop in it relatively small application. However, the system is fundamentally flawed, there are concepts that will create ripple effects in reaction to changes. Worst of all is the concept and role of the `Form` class, containing too many different change drivers.

I have to be critical to myself here. If anyone considers writing an application with WinForms today, the community will suggest much better approaches, MVP being the most voiced one with suggestions on how to isolate and limit certain areas that I marked as sources of *Combinatorial Effects*. I admit that is true. If one would start with WinForms now, even though I would suggest some other technology entirely, if at all possible; but the goal of transitioning and evolving a long existing application is much more in line with the presented model of WinForms. I already made some hard to overcome simplifications that would most likely not hold in the real world, like very strict following of the suggested use of a framework with minimal deviation. So I dare to expect an application that was written long ago with the standard of its time.

3.1.3 Windows Presentation Foundation

The Windows Presentation Foundation (WPF) [23] previously known as Avalon was first released in 2006 with .NET Framework version 3.0. It is an extensive GUI framework for building desktop applications with rich contents, controls, dynamic layouts, data binding, and more. It uses Extensible Application Markup Language (XAML) as a declarative way to describe its visuals, but XAML does not depends on WPF neither the other way around. Currently WPF project is hosted on GitHub pages [25] and is open source under the MIT license.

3.1.3.1 Intended use

The WPF framework was presented as new take on user interface, but not a direct replacement of WinForms. Initially it was a way to provide support for rich content such as animations, media, documents etc. It is also much more modular than WinForms allowing for placing controls into other controls and over all easier extensibility.

It evolved through the years to be very versatile and supporting modern GUI approaches. We also have to take into consideration that WPF was and still is trying to be very compatible with WinForms allowing similar concepts and code practices as WinForms. Big impact on the use of WPF is that it was presented with the MVVM paradigm as suggested approach and the community followed using it this way.

The big change is the possibility to define GUI with XAML. I'll try to explain the basic concepts with listing 3.5. I am using the example from Microsoft's documentation the first and very basic application [26].

The idea is a declarative approach to defining user interface going even as far as to have a graphical designer person doing this part of development. Microsoft supports this approach with standalone product Blend, where one can fully engage with the design part of GUI. XAML is a markup language based on XML so it describes a tree like structure. `Page` hosts `Grid`, cells of `Grid` have `ListBox` and `Button`.

XAML is not enforced by WPF, GUI elements can be defined procedurally, but XAML looks much cleaner and this opinion seems to be reflected by the community around WPF as well.

On the technical side XAML describes what is called Logical Tree in WPF. This is the description of higher level elements that create the GUI it is also used for Dependency Properties, Static and Dynamic Resources, and Data Binding. I will explain these terms a bit later. There is a support for dynamic layouts as with the `Grid` in the example. You can see the row's `Height` parameter being set to `Auto`, making layouts very easy.

There is also Visual Tree a super set of the Logical Tree with all the different elements that are being rendered. So a single node `Button` from Logical tree is expanded to a subtree with `Border`, `ContainerPresenter`, and a `TextBlock`. The Visual Tree is utilized when rendering objects, layouts, and for Routed Events that travel across it.

How a Control will render depends on its template usually a XAML file, where the complete description of its visuals is stated. This template can be

3. ANALYSIS

```
<Page x:Class="ExpenseIt.ExpenseReportPage"
      xmlns="http://schemas.microsoft.com/winfx..."
      ...
      mc:Ignorable="d"
      d:DesignHeight="300" d:DesignWidth="300"
      Title="ExpenseIt - View Expense">

  <Grid Margin="10,0,10,10">
    <Grid.ColumnDefinitions>
      <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Grid.RowDefinitions>
      <RowDefinition Height="Auto"/>
      <RowDefinition />
      <RowDefinition Height="Auto"/>
    </Grid.RowDefinitions>

    <!-- People list -->
    <Border Grid.Column="0" Grid.Row="0" Height="35" Padding="5"
            Background="#4E87D4">
      <Label VerticalAlignment="Center"
             Foreground="White">Names</Label>
    </Border>
    <ListBox Name="peopleListBox" Grid.Column="0" Grid.Row="1">
      <ListBoxItem>Mike</ListBoxItem>
      <ListBoxItem>Lisa</ListBoxItem>
      <ListBoxItem>John</ListBoxItem>
      <ListBoxItem>Mary</ListBoxItem>
    </ListBox>

    <!-- View report button -->
    <Button Grid.Column="1" Grid.Row="3" Margin="0,10,0,0"
            Width="125" Height="25" HorizontalAlignment="Right"
            Click="Button_Click">View</Button>
  </Grid>
</Page>
```

Listing 3.5: XAML example

overridden to change visuals of any given Control. Mirroring the concepts from WinForms there is also a User Control created by combining existing Controls and a Custom Control deriving from the `Control` class describing a completely new element.

I mentioned few concepts that are new with WPF so I'll briefly explain them.

- **Dependency Property** – An object property that is managed by WPF framework. It is registered with its metadata and allows for new functionalities that were not possible with ordinary object property such as: styling, data binding, using dynamic resources, and animation.
- **Resources** – Objects can be defined as resources making them available for reuse in other XAML files. Static resources causes a single lookup where Dynamic resource creates a link to the value and update on change. Availability scope depends on the declaration point and spans only the subtree. Resource dictionaries, separate files, are a common practice that allows easy reuse of defined objects.
- **Data Binding** – The concept is identical to the one presented in this thesis. It is a way to link data of GUI controls to data in objects behind GUI. XAML supports multiple modes, most used are one-way and two-way.
- **Routed Events** – This feature allows for Chain of Responsibility succession of event handlers to react to an event. The chain progresses along the Visual Tree and can be bubbling (up) or tunneling (down) depending on its direction. This allows for reacting only to events that interest certain control and pass the others.
- **Commands** – A semantic level approach to actions that application should execute. It allows for removing logic code from event handlers and sharing these objects encapsulating the actions across GUI. There is a little more complexity for `RoutedCommands`, but the are the most useful in terms of reuse and isolation.

These are some interesting additions to the GUI toolbox that can be leveraged to modularize, reuse, and extend the out of the box WPF experience. There is one more term I would like to present and that is *code behind*.

Essentially all classes generated when adding new XAML files have the extension `.xaml.cs` are *code behind* classes, see listing 3.6. These are partial classes that are also described by the XAML files and one can write logic and business code here into them and their event handlers, but the best practice

3. ANALYSIS

```
using System;
...
namespace ExpenseIt
{
    /// <summary>
    /// Interaction logic for ExpenseItHome.xaml
    /// </summary>
    public partial class ExpenseItHome : Page
    {
        public ExpenseItHome()
        {
            InitializeComponent();
        }

        private void Button_Click(object sender,
            RoutedEventArgs e)
        {
            // View Expense Report
            ExpenseReportPage expenseReportPage = new
                ExpenseReportPage();
            this.NavigationService.Navigate(expenseReportPage);
        }
    }
}
```

Listing 3.6: Code behind file ExpenseItHome.xaml.cs

is to leave them bare and empty, and use commands instead of other objects that encapsulate just logic.

So there is a familiarity for WinForms developers moving to WPF, that can be used in the transition period, but I do not think it should be the end of the process. In order to benefit from MVVM we need to create the objects of ViewModels that XAML View can bind to and link to their data and commands. The possibility is there and thanks to the multiple new concepts of WPF it is feasible to do so.

Similarly to WinForms, WPF is an established framework with several years and versions of improvements under its belt. Additional features and concepts pile up with time. I definitely did not aimed to write a deep dive into the framework, that would be out of scope of this thesis. I presented the most interesting ideas and functionalities that would be helpful to analyze. Before I do an evaluation of WPF I want to have a quick look on WinForms, WPF interoperability.

3.1.3.2 Interoperability

Microsoft documentation has an extensive section on the topic of WinForms and WPF interoperability describing different scenarios. The most valuable

```
<Window x:Class="HostingWfInWpf.Window1"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
    presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:wf="clr-namespace:System.Windows.Forms;
    assembly=System.Windows.Forms"
  Title="HostingWfInWpf"
>

  <Grid>
    <WindowsFormsHost>
      <wf:MaskedTextBox x:Name="mtbDate" Mask="00/00/0000"/>
    </WindowsFormsHost>
  </Grid>

</Window>
```

Listing 3.7: Interoperability example

information there is that both directions are supported even though there are some inherent limitations. There is a very simple approach of hosting WinForms forms inside a WPF host control that can be described using just XAML. Example of such control can be seen in figure 3.7.

3.1.3.3 Evaluation

Before I start looking at principles of NST and EA, some things are just staring at me. WPF as a framework is much more complex, with some conceptual hoops to jump through. It seems that the framework is much more capable, but is it better in terms of ripple effect in reaction to change and over all evolvability?

Regarding principle Separation of Concerns the state is much better with XAML describing visualization and layout, and code behind doing the rest. The user input is intercepted by Controls and thanks to the concept of Routed Events that traverse the tree structure. This isolation of elements allows Components to focus only on events relevant to them. The action following the input could still be implemented into the event handlers, but if I allow the influence of MVVM there should be a ViewModel objects with the relevant commands. This makes a big difference as we can now bind to the commands directly from the Controls even in XAML. This allows for complete separation of business logic, but costs some extra effort.

Where I can see a weak point is in the relation of Model and ViewModel that was not much discussed here. The idea of MVVM is that the ViewModel is a wholesome description for the GUI to render. Some of this might be very fragile depending on the exact implementation of passing data between Model and ViewModel, but WPF does not describe this part. Nonetheless this relationship can be a source of *combinatorial effects* if multiple ViewModels feed from a shared Model, change to the Model can have unbounded impact.

The binding to data is also extended allowing not only multiple modes, but also a wide variety of targets. The most notable is the Dependency Property that allows for all the dynamic links. This results in evaluating these properties just-in-time even based on multiple inputs, that are not tightly coupled. This contributes to lowering the *combinatorial effects* since instead of propagating and copying data there is a link to a central dictionary. Limiting the possible number of object relationships from n^2 to just n .

Last topic of my interest is modularity of the system. Looking at the View side of the framework XAML is working very well in this direction. It allows for resource dictionaries, separate styles, and each Control to be defined independently. On the side of business logic it can be direct and simple written in code behind, loosing the modularity and reusability, but embracing the View-Model concept that is supported with Commands and wide available Data Binding all the logic of the application can be made reusable and wrapped in appropriate objects.

3.1.3.4 Summary

It might look from the Evaluation above that WPF is clearly much better in terms of functionality and also in terms of evolvability. I would tend to agree that WPF with the MVVM architecture seems like a really good approach to GUI in general, but I would also like to add that the devil is in the detail. I tried to describe the most notable concepts and make some arguments for what they mean in terms of my scope. The point to take from this analysis is the approach. I would advise to be much more thorough in the case of real project analysis that should be done on a specific code base.

3.1.4 Summary

I gave a basic overview of the two technologies I will be using in my exploration of the transition process. I am aware that they are rather brief trying to capture the concepts and not exactly the implementation details. One of the important realizations that came from the analysis is that frameworks can be leading developers towards a certain GUI architectural pattern, but they can be adapted to others as well.

This means that for an existing project that should undergo migration to another framework one should analyse the code base on multiple levels. Namely: chosen architectural pattern, framework's support for said pattern, framework's functionalities, and adherence to the implicit implementation rules that come from these aspects. I think these levels should be looked at on both ends of the transition, meaning starting state and intended state. Adding to that in real world project we need to pose the following question. Is the code base fragmented and full of edge cases and ad-hoc solutions, or is it systematic? This is one more overarching quality that needs to be reviewed before we even consider transition at all. All of these angles are necessary in order to understand what does the transition from one GUI technology to another actually mean for a given project.

3.2 Transition approaches

Suppose we have an application and its GUI is implemented using a certain framework. If we are tasked to transition this application to another GUI framework the path is not instantly clear. One of the prerequisites to even start thinking about this path is an analysis of the initial state and the intended state. This would be similar to the one described above, most likely more detailed and in depth. However, even if we have descriptions of the states of the application there is a non-trivial question of *How do we get to the intended state?*

I divide the options to two very different approaches one being a Rewrite of at least the GUI layer of said application. This means a new project and the new version is not deployed until done. The other approach is Incremental. A transformation from the initial state to the intended state in steps. The application remains operational in each one of those steps.

In the following sections I will reason and argument about the benefits and drawbacks of these paths. I want to provide some clues as to how they should be used and what tools might be useful on that path.

3.2.1 Rewrite

This approach in scope of this thesis means developing a new presentation layer using the new technology. Even though, this might seem like a fresh start it is definitely not. The requirements on functionality, appearance, and behaviour are transferred to the new GUI. Essentially we need to reverse engineer the current presentation layer gather at least the conceptual model and build a new presentation layer. As for estimating the time and cost of this process there are solutions that deal with the reverse engineering [27],

3. ANALYSIS

the other direction is a usual process for every software project and can be estimated.

This approach leads to extraction of the concepts from a developed GUI. This act in and of itself can be very beneficial not only to the transition, but to the project itself. But it is also non-trivial and has its own challenges. I would argue that without this extraction and critical look at the conceptual model the development is just blindly copying what was already implemented, potentially leading to big problems of GUI that does not work as the initial one.

The idea of rewriting an application, or its part, that is in need of an update or severe maintenance is not new. There are real world examples that we can point to. One of which is the case of Netscape 6.0. Netscape was in its time the most dominant web browser on the market. When the company was developing the version 5.0 of its dominant browser the code proved very difficult to deal with and a decision to scrap version 5.0 was made. Instead the company decided to rewrite its aging code base and it took three long years. During these years the company actually lost its independence and was overtaken by AOL. When the Netscape 6.0 was finally released in November 2000, the product was rushed and not fully developed. By that time the competition in form of Internet Explorer already flooded the market. This marked the end of Netscape era. Last version of the Netscape browser was released in 2008 and the company is now under the Verizon brand.

These are also not unique consequences of such decisions to rewrite an application. Similar stories can be found with Microsoft's Word in 1991, that effort was abandoned, and many others. I can personally add at least one experience of such ambitions to rewrite a product. It was a desktop application. The project had a dedicated team and the original application was still being maintained and also developed with new features. The reason for that was no to stay frozen in place until the rewrite is done. So there was a race to implement all features of the old application, plus all the new features, plus challenges of new environment in the same time that the old application added only new features. After six long years of this marathon and about ten million Czech crowns spent, the project was halted and the team was dismissed.

My conclusion for this approach is that one has to be very careful if this is the chosen path forward. Rewriting implicitly means great time delay, expenses that might be hard to estimate, usually a broad scope, even if only part of a system is being rewritten like GUI, and lastly great risk that it might fail at any point. If we would want to relate to the agile approach of software development today we cannot be much further apart. I am not saying rewriting is to be completely avoided, it depends on context, tools available, etc. But personally

I want to avoid it if I can and there are other smarter people than me like Joel Spolsky, co-founder and CEO of StackOverflow, who think alike [28].

3.2.2 Incremental

This approach means gradual changes of the application, or in scope of this thesis the presentation layer. We have an initial state A and an intended state B of an application. The path is a series of small steps altering the current state until will reach B . The key principle here is that the applications remains operational at each step and is potentially also deployed.

$$A \rightarrow A' \rightarrow A'' \rightarrow A''' \rightarrow \dots \rightarrow B$$

Once again I can refer to the ideas of Martin Fowler as he calls this approach the *Strangler Application* [29]. There are real world examples that show success using this approach and promote the reduce risks that it provides. This paper, *An Agile Approach to a Legacy System* [30], gives an insight into a transition of legacy financial application InkBlot.

What changes exactly does each step introduce? This is a very good question and it is not easy to answer, but I will try. One can expect introduction of some isolation layers like the *Anti-Corruption Layer pattern* [31] or generally an abstraction layer. I will elaborate on this topic little more later. In the InkBlot example there is also mentioned the idea of delivering the core functionality first. It is building on the 80/20 rule saying that the most used twenty percent of features satisfies eighty percent of users. As this approach draws from the Agile approach to software development I will make on more parallel. Each step has to either get us directly closer to the intended state. This can mean that some part of GUI is migrated to the new technology or some part of the architecture is transformed to the new paradigm.

Or the step is enabling us to move closer to the intended state in the next step. By this I want to describe work on restructuring the code and opening new options. These preparations actually make the transition even possible. Another important part is testing, functional and integration testing, preparation of these also fits into these steps.

Coming back to the abstraction layer I mentioned. There are several case studies [32], where we can see the usage of this pattern in order to allow the transition and be able to deploy in each step. The core idea is that the abstraction layer is developed between a consumer and a component providing certain functionality, figure 3.2.

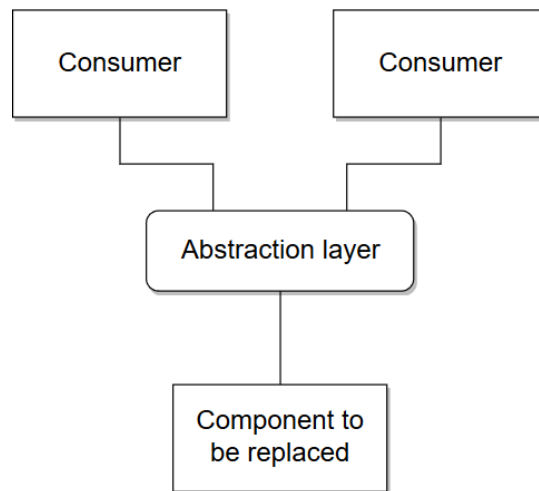


Figure 3.2: Abstraction layer placement

The step of creating the abstraction layer is beneficial on its own as it can point to what is used and needed from the consumer perspective. Thanks to the existence of the abstraction layer we can gradually replace the the old component with the new one, figure 3.3. When all the functionality is used from the new component we can remove the old one and potentially also the abstraction layer. This depends, if it imposes penalties on the application performance, and ultimately is a decision for the project's team.

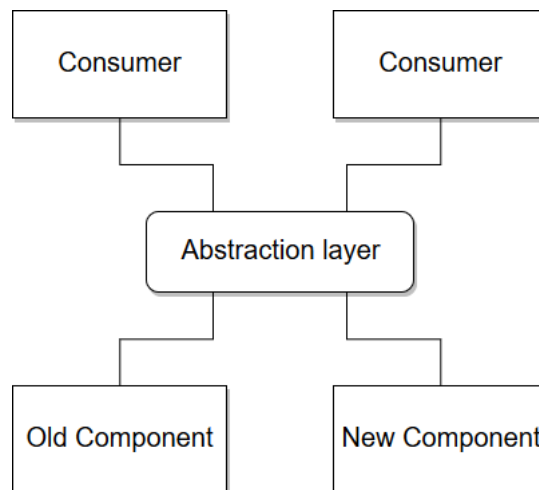


Figure 3.3: Abstraction layer usage

I intentionally do not mention how broad the abstraction layer is and what granularity the consumer and components are. This can be chosen during the transition and depends on technical details and other context.

Stepping back a bit the problem I am describing here looks like combating *technical debt*. And I would argue that indeed the root problem is that the old technology has become *technical debt* and we are now paying interest on it during the migration. There are many other great source on how to deal with and prevent *technical debt*, like the book *Refactoring* [33].

3.2.3 Summary

I presented two approaches of transforming an application from one technology to another. Both are used in real world projects even though it seems the incremental approach is more popular with the rise of agile in software engineering. It is hard to make convincing conclusions from the descriptions I provided. For me the incremental approach makes much more sense and limits the risks inherent to a migration between environments. You don't have to take my word for it.

The Standish Group International has done a research on this topic. They focused on some 100 applications that stood before decision to either rewrite, buy a package to deliver needed functionality, or modernize current software. The research looks at cost, risk, and reward of these options. Their conclusion is that the modernization is pre-optimized and limits the number of decisions needed to achieve the goal. These information come from the report that The Standish Group International publicly provided [34].

3.3 Testing

Testing plays an important part in developing any application. The fact that we are discussing revising an application, or part of an application, does not change that. On the contrary, it may pose additional requirements on the testing as it does on the development, as I described earlier. Further more as I scoped this thesis on the topic of GUIs there are inherent challenges to testing user interfaces.

To verify that application after transition behaves as before it we need some system tests or acceptance tests. Preparing these tests can uncover problems and inconsistencies that need reaction on their own. This depends on the principle behind the transition, but questions like *Should we reproduce this buggy behaviour?* might appear. Answers to these questions can shape the additional requirements on the intended state.

The other layer of complexity is testing of GUIs. It is not a simple task. One can deploy complicated setups of Selenium [35], Ranorax [36], or TestComplete [37]. All these solutions aim to automate GUI testing in some way. The test cases usually need constant support and updates to keep up with the change

of the visual appearance. In case of the Rewrite approach these are basically not usable at all.

Aiming at testing the functionality with unit testing is possible. But unit testing the GUI itself is considered not worth it [38]. So instead unit test the business logic. And track the actions that are triggered by GUI elements in form of messages for example. These practices seems like a much better idea.

Of course there is always manual testing and evaluating user feedback. This should be always part of the transition process. Utilizing the previously described approaches might prove difficult with the Rewrite approach as I do not expect modifications to the initial application. For the Incremental approach the situation is very different and I would argue that not only the intended state should pass these tests. Also the incremental steps should be covered. This is only supported by the notion that the application is deployable in every step of the transition.

3.4 Summary

Closing the analysis chapter I want to reiterate its contents. I reviewed my chosen technologies and did their evaluation with respect to limited evolvability. I did explore the approaches that are available for transition between technologies and made some comments on their characteristics. I expressed that the Incremental approach is much more relevant to me. It seems to be getting more attention in the industry. It also ties nicely with the Evolutionary Architectures that expand the ideas of DevOps. I gave a short insight into the importance of testing. There is definitively a need to at least test the initial and intended states.

Case study

This chapter is focused on a case study. In the previous chapters I was exploring the different approaches and prerequisites that I deemed necessary to realize a transition of an application. Here, I show an example of such process with a small demonstrative project. This case study by no means the focus of this thesis. In fact it is just a small part of it demonstrating few of the above mentioned concepts.

4.1 Introduction

I want to reuse the little GUI example that I presented at the beginning of this thesis in section 1.1.2. For this reason I follow through with the presented domain. A fragment of Car Dealership information system. Of course I needed to expand it a little so that the example application is not just a single dialog window. On the other hand bare in mind that the application is intentionally very naive. The quality of the solution for the domain is also just demonstrative. The point of the case study is to show the evaluation, the planning and the steps of the incremental approach of a transition.

4.1.1 Domain

Previously I described the single dialog show in figure 1.4. The idea is that there are three values **Target**, **Actual**, and **Variance**. The **Target** is set for each month by the dealership headquarters, the **Actual** is set by an employee of a branch of said dealership, and the **Variance** is calculated and colored by the system.

Now, to the expanding the story of Car Dealership. Imagine There are multiple branches in different locations. So the system keeps information for each branch location. The dealership also offers multiple models of its cars. For

4. CASE STUDY

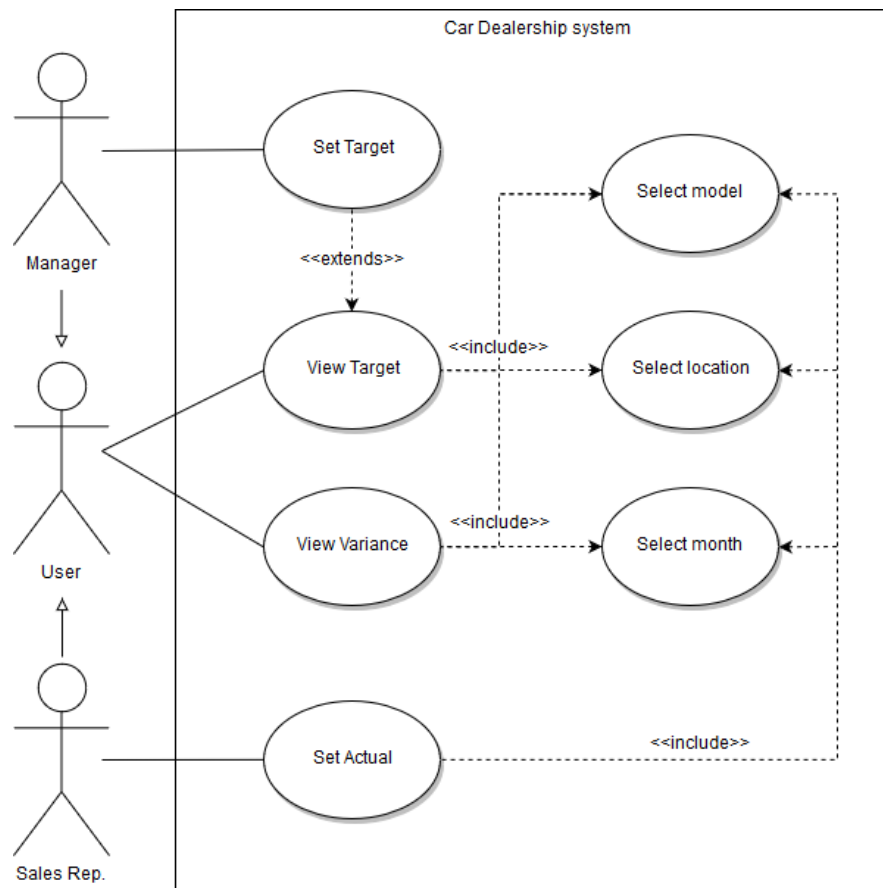


Figure 4.1: Car Dealership app use cases

each of this type the system keeps the three values. Finally there is a Manager in the headquarters and a Sales representative working for a branch. The Manager is the person who sets the **Target** value every month for each branch and model. The Sales Representative is the person who inputs the **Actual** value for each model for his branch.

This all boils down to a couple of use cases that we can see at figure 4.1. Both users want to view **Target** and **Variance** values. These use cases include the need to select branch location, month in a year, and car model. The Manager also wants to modify the **Target**. This use case extends the common view **Target** one. Lastly the Sales Representative needs to set the **Actual** value. This last use case shares the needs of selection of location, month, and model.

4.1.2 Technologies

The case study is a desktop application written in C# running on Windows. As previously advertised I am using WinForms and WPF in this case study. Aligning with my analysis of these frameworks I honor the suggested architectures Forms and Controls, and Model View ViewModel respectively. I do not claim the application is programmed expertly. But it is implemented systematically and true to the practices of the framework and the architecture.

The case study disregards any data layer or persistence as it is not important for the example. The data source is still needed though and is mocked sufficiently enough to show the GUI is working as expected.

4.1.3 Transition approach

In the summary at the end of the analysis chapter, section 3.4, I expressed I am favoring the Incremental approach for the transition process. That holds true in this case study. This opens a few questions that I need to answer like: What is the initial state? What is the intended state? I can also ponder about steps that are necessary to reach the intended state. And I think I can foresee some that will be likely on the path of the transition.

In this example transition process I do not want to change any functionality nor implement new requirements. The only goal is to realize the migration from WinForms to WPF. This is most likely an artificial setup. One of the benefits of the Incremental approach is the ability to satisfy needs of the users of potentially old and neglected application. So in a real world transition the Incremental process could be much more dynamic and agile.

Initial State

The Car Dealership application fulfills all the use cases depicted in figure 4.1. The framework used for implementation is WinForms with the Forms and Controls architecture.

Intended State

The Car Dealership application fulfills all the use cases depicted in figure 4.1. The original dialog from figure 1.4 is implemented using WPF and MVVM.

Expected Steps

As I mentioned earlier I see two kinds of steps in the Incremental process. Enabling steps and direct steps. Regarding direct steps there is the need for migrating controls from WinForms to WPF. This can be as granular as a

control per step. Using the interoperability of WinForms and WPF to keep the application working at each step. I also need to change the underlying paradigm to the MVVM for the dialog. This need some enabler steps to add necessary supporting code. Further enabling steps should cover tests to verify that the requested behaviour of the GUI remains present.

None of this is binding. I do not commit to any order of these expected steps. It only serves me as a mind map to list the hurdles that I have to overcome no matter the implementation of the transition steps.

4.1.4 Testing

In the scope of this case study I am not diving into a full testing suite. I don't even carry out manual testing of initial and intended states. What I am doing is tracking the actions of the application in a log. Each user interaction with GUI starts a chain of messages. These chains are my checks to verify that the functionality stays the same across my incremental steps. It is not a perfect system it lies on the fact that I log actions at the point when they happen consistently. And I do try to do so.

4.2 Implementation

Now after I presented the case study concept, its domain, transition approach, and testing, I show the process of transitioning in greater detail. For this purpose I created a git repository that can be found on the SD card attached to this thesis. In that repository there are tagged commits referring to the initial and intended states. All the commits in between are considered steps, respecting the scoping of this case study. In the following sections, I present the initial implementation of the example application and go through each step.

4.2.1 Initial state

The initial implementation of the case study application follows the initial state description that was presented in the Introduction. The role of the user is selected on the first screen, figure 4.2. Than the user can select the car dealership branch for which he wants to see data, figure 4.3. Next screen is the main screen, where data for each model of a selected branch location are shown, 4.4. With the click of on the button **Set** user can open the *Set Dialog* that was used when describing GUI architectures, figure 4.5.

Focusing on the *Set Dialog* that I transform in the following steps. It is implemented as a Windows Form. It gets its relevant data during construction passed from the parent form. On the save button click it sets its `_saved`

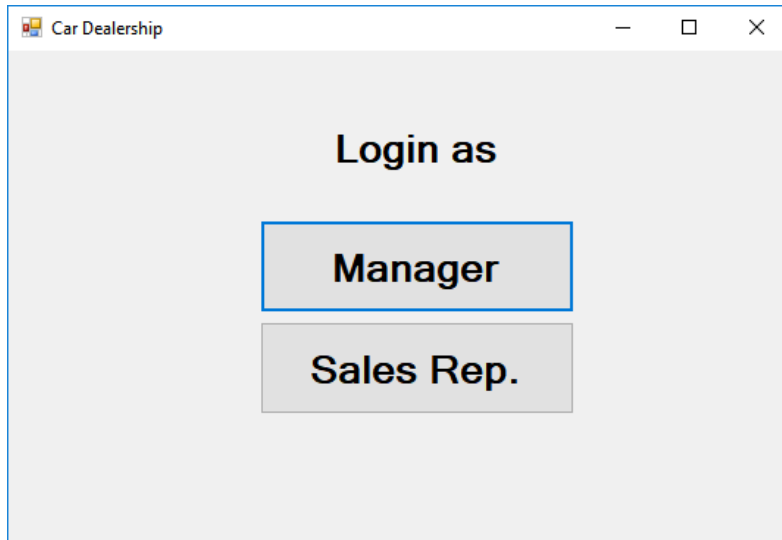


Figure 4.2: Login screen

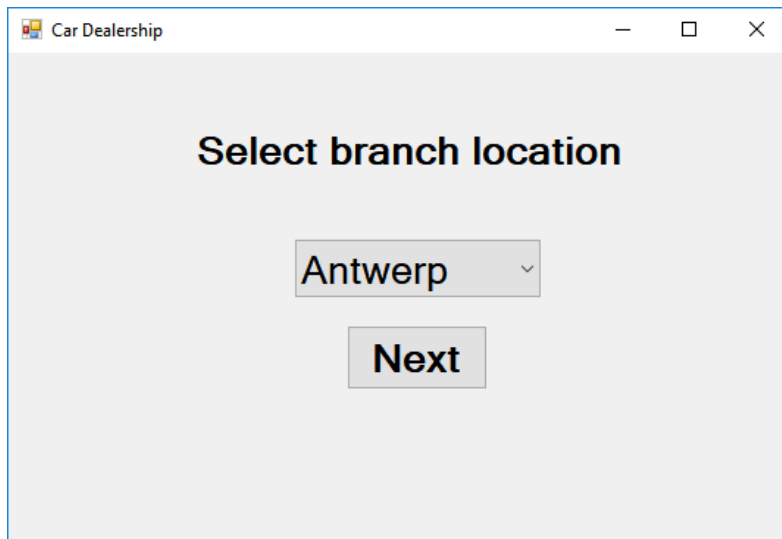
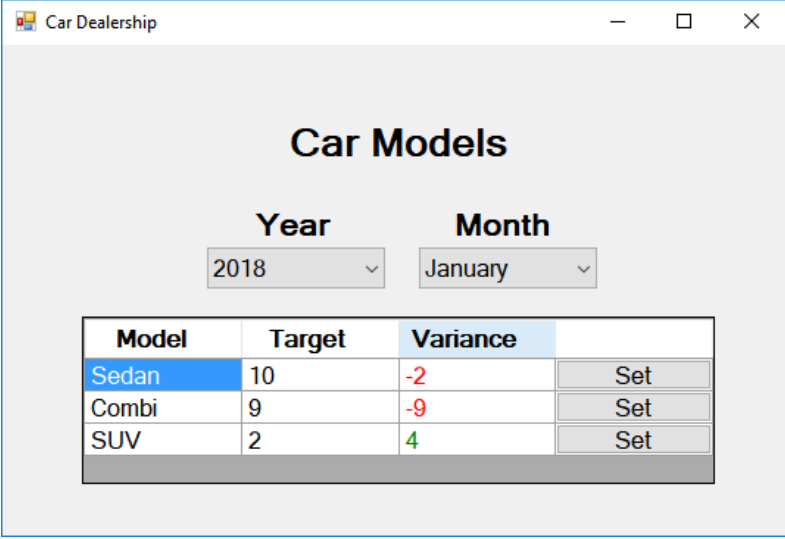


Figure 4.3: Select branch screen

4. CASE STUDY



Model	Target	Variance	
Sedan	10	-2	Set
Combi	9	-9	Set
SUV	2	4	Set

Figure 4.4: Main screen

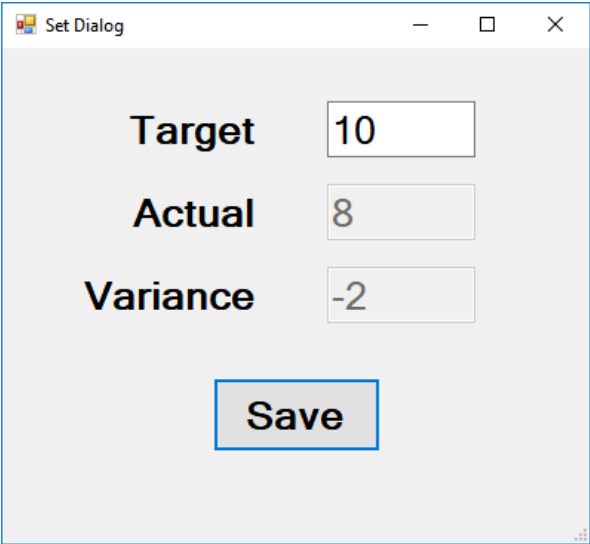


Figure 4.5: Set data dialog

```
public partial class SetForm : Form
{
    private CarDealership.Role _userRole;
    private bool _saved = false;
    private (int Target, int Actual) _originalData;

    internal SetForm( (int Target, int Actual) data,
        CarDealership.Role userRole )
    {
        InitializeComponent();
        _userRole = userRole;
        _originalData = data;
        target_tbx.Text = data.Target.ToString();
        actual_tbx.Text = data.Actual.ToString();
        variance_tbx.Text = (data.Actual -
            data.Target).ToString();
    }

    private void SetForm_Load( object sender,
        System.EventArgs e )
    {
        if( _userRole == CarDealership.Role.Manager )
            target_tbx.Enabled = true;
        else if( _userRole == CarDealership.Role.Sales )
            actual_tbx.Enabled = true;
    }

    private void save_Click( object sender,
        System.EventArgs e )
    {
        _saved = true;
        Close();
    }

    internal (int Target, int Actual) GetData()
    {
        if( !_saved )
            return _originalData;

        var Target = int.Parse( target_tbx.Text );
        var Actual = int.Parse( actual_tbx.Text );
        return (Target, Actual);
    }
}
```

Listing 4.1: Set Dialog implementation

4. CASE STUDY

variable to true. This variable is checked in method `GetData` to figure out if the new or the original data should be returned. Here you can see the whole implementation of the *Set Dialog*, see listing 4.1.

The implementation is not complex as the dialog itself is not complex. But even here there is some logic like what text boxes has to be enabled and how to return data after the dialog is closed.

4.2.2 Step 1 – Log messages

The first step is an enabling step. I manually added messages that describe user actions on GUI elements. I implemented messages to internal functions in order to see the chain of system reactions that follows. You can see A log of these messages in listing 4.2. The intended messages are produced by the application. This log serves me as an example for the application behaviour and I my goal is not to mutate it.

```
Application start
  CarDealership app initialized
  Antwerp location selected
  January month selected
Click on Manager button
  Manager role selected
Click on Next button Grid layout set up
  Grid data loaded
  Grid data updated
Click on Set button of the first row SetDialog Target
  value changed
  SetDialog Actual value changed
  SetDialog for model Sedan opened
Input value 8 as new Target value
  SetDialog Target value changed
Click on Save button SetDialog values saved
  SetDialog closed
  Grid data updated
```

Listing 4.2: Log of action messages

4.2.3 Step 2 – WPF project

Another enabler step. This is one is purely technical I had to make changes to the `CarDealership.csproj` file in order to use WPF elements.

4.2.4 Step 3 – Dialog migration to WPF

In this step I switched the framework used for *Set Dialog*. I created a XAML file where the WPF controls are described, but I kept the logic as is. This

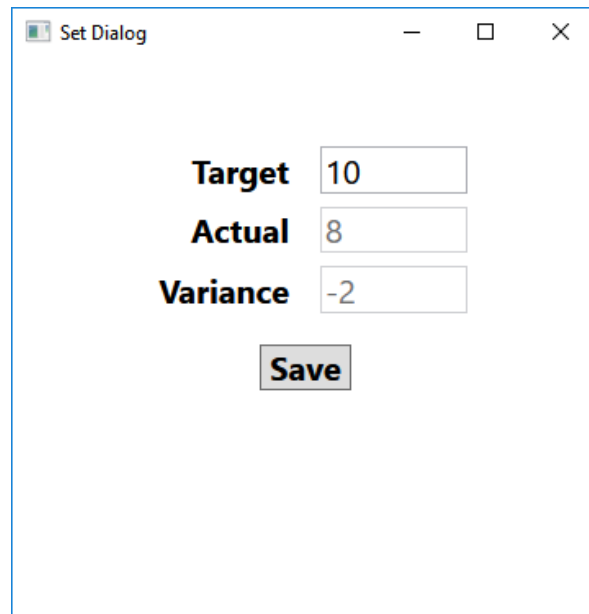


Figure 4.6: Migrated Set dialog

results in a Form and Control architecture with WPF. I had to make some changes to the names of a few control properties such as `Enabled` changed to `IsEnabled` in WPF, but these are very minor. You can see that the XAML does not use binding or commands preferred by MVVM rather it still describes event handles, see listing 4.3. You can also see that the visual appearance changed slightly thanks to WPF, figure 4.6.

4.2.5 Step 4 – Relay command

Here is another technical step in preparation to move from Forms and Controls architecture to MVVM. I added a `RelayCommand` class that I later use to implement commands that are bound to the GUI elements in XAML.

4.2.6 Step 5 – Dialog migration to MVVM

Last step, reaching the intended state for this case study is the switch of the paradigm for the *Set Dialog*. This meant adding a new class `SetFormViewModel`, which as the name suggests is a `ViewModel` for the *Set Dialog*. This step moved the logic from the Form to the `ViewModel`. This affected the XAML, that now binds to the properties of the `ViewModel`, see listing 4.4. It also meant that the code behind is essentially empty, see listing 4.5. It also changed slightly how the parent Form handles the dialog, that can be seen in the attached git repository. Lastly the log of messages produced by the application in reaction to user input stayed identical to the one listed in here 4.2.

4. CASE STUDY

```
<Window>
  <StackPanel>
    <Grid>
      ...
      <TextBox x:Name="target_tbx" Grid.Column="2" Grid.Row="0"
        IsEnabled="false"
        TextChanged="target_tbx_TextChanged"/>
      <TextBox x:Name="actual_tbx" Grid.Column="2" Grid.Row="1"
        IsEnabled="false"
        TextChanged="actual_tbx_TextChanged"/>
      <TextBox x:Name="variance_tbx" Grid.Column="2"
        Grid.Row="2" IsEnabled="false" />
    </Grid>
    <StackPanel Orientation="Horizontal"
      HorizontalAlignment="Center">
      <Button x:Name="Save" Content="Save" Padding="5,0"
        Margin="0,15" Click="save_Click"/>
    </StackPanel>
  </StackPanel>
</Window>
```

Listing 4.3: XAML describing Set Dialog view

```
<Window>
  <StackPanel>
    <Grid>
      ...
      <TextBox x:Name="target_tbx" Grid.Column="2"
        Grid.Row="0" IsEnabled="{Binding TargetEnabled}"
        Text="{Binding Target}" />
      <TextBox x:Name="actual_tbx" Grid.Column="2"
        Grid.Row="1" IsEnabled="{Binding ActualEnabled}"
        Text="{Binding Actual}" />
      <TextBox x:Name="variance_tbx" Grid.Column="2"
        Grid.Row="2" IsEnabled="{Binding VarianceEnabled}"
        Text="{Binding Variance}" />
    </Grid>
    <StackPanel Orientation="Horizontal"
      HorizontalAlignment="Center">
      <Button x:Name="Save" Content="Save" Padding="5,0"
        Margin="0,15" Click="save_Click" Command="{Binding
          SaveCommand}" />
    </StackPanel>
  </StackPanel>
</Window>
```

Listing 4.4: XAML describing Set Dialog view with MVVM

```
public partial class SetFormWpf : Window
{

    internal SetFormWpf( SetFormViewModel viewModel )
    {
        InitializeComponent();
        DataContext = viewModel;

    }

    private void save_Click( object sender,
        System.EventArgs e )
    {
        Close();
    }
}
```

Listing 4.5: Set Dialog MVVM implementation

4.3 Summary

I presented a case study on a small project. Showing the preparation and the process of migration itself. I utilized all the topics mentioned during this thesis. I focused only on properly migrating one single dialog. Even that proved to be challenging when considering all the aspects of a migration. I implemented some elementary testing in the form of logging action messages. Overall I consider this case study as a reasonable walk through the process of migration from WinForms to WPF.

The more challenging parts of the case study were definitely the categorization of the frameworks and their analysis. I mentioned it previously, that for real world projects I feel the need to be much more careful and thorough. Another more difficult step is the switch of the paradigm. I can see many pitfalls in it. There is the need to understand both frameworks and paradigm and devise a reasonable hybrid during the transition. Lastly I see testing and verifying functionality as a difficult task. I did not utilize any advanced techniques for testing. There will be problems as the GUI changes especially with the visual aspect even if the new implementation strides to be as similar as possible.

Related Work

Putting this thesis into context of other work is not exactly easy. I would argue there are at least two levels how to look at it. Directly related work. Meaning any article or research on the topic of migrating GUI across technologies and what approaches should be taken. As far as I know there is no work done directly on this topic.

With no direct comparison I can point out the loosely related work. There are several areas that I consider related in this way. The specific migration from WinForms to WPF and their interoperability. There is a book *WPF recipes in C# 2008: a problem-solution approach*. In this book there is a chapter dedicated to WinForms and WPF relation [39, ch. 13]. The chapter is loosely related as it does not delve into the wider process of migration. It only looks at the technical aspects of converting from WinForms to WPF and interoperability. The authors write specifically that they do not aspect mass migrations or rewrites from WinForms to WPF.

Another area is general migration of the GUI layer. Even here I struggled to find reasonable number of references, but here is the most interesting. The article *Automated reverse engineering of Java graphical user interfaces for web migration* [40] describes how to reverse engineer existing desktop GUI and migrate it to Web. It is also narrowly scoped. The environment presented in this article is Java environment: Java Swing and Aspect J. The authors also provide a case study of their approach. This work is related in the topic of migrating Java GUI elements. It describes similar problems as this thesis and springs from very similar motivation.

Another related area is work around the concept of evolvability. Specifically Normalized Systems and Evolutionary Architectures. Both topics are described in the section 1.2 Methodology. They are one of the corner stones of this thesis as they are used to evaluate frameworks and guide the direction

5. RELATED WORK

of the GUI transition. The NST defines *combinatorial effect* which it aims to eliminate in order to make systems more evolvable. It also describes four principles to achieve the elimination. These are: Separation of Concerns, Data Version Transparency Action Version Transparency, and Separation of States.

The NST is implemented in the form of NSX tools [41]. These tools are capable of generating and regenerating systems that are guaranteed to be evolvable based on NST. In a sense this is an automated rewrite of an application.

Evolutionary Architectures is the second approach to the topic of evolvability. It attacks the problem from the higher system level. It looks at concepts such as microservices, expanding the DevOps approach, and evaluating the progress with system fitness function.

Evaluation

Concluding this thesis I return to my goals that I formulated in section 2 Goals revisited. The following is a list of each goal and its resolution:

G1 Describe common GUI architectural patterns

I fulfilled this goal by presenting and analysing the various paradigms in section 1.1. I did my best to differentiate and name the most common architectural patterns that are publicly known. The categorization wasn't an easy task as there are conflicting sources on this topic.

G2 Present trends and methodologies focusing on evolvability of architectures and choose principles for analysis

This goal is satisfied mainly by section 1.2 Methodology. In this section I describe the two main approaches to looking at the term evolvability. Those are Normalized System Theory and Evolutionary Architectures. In this section I also summarized that they aim for the same goals so there is a common ground. In the section 2 Goals revisited I specify the principles for following analysis based on the common ideals of the mentioned methodologies.

G3 Choose two GUI framework/technologies and categorize their architectures

In the second chapter in section 3.1 Frameworks I talk about choosing two frameworks of interest to me. These are Windows Forms and Windows Presentation Foundation. In the following sections I do an overview of both and categorize them. I used the architectural patterns that I presented to fulfill the goal G1. I have to say that one cannot categorize just frameworks. Frameworks are flexible to fit many paradigms. I did my best to select the most used paradigm to each framework in my analysis.

G4 Evaluate chosen technologies based on NST and EA

This goal is reached by two sections 3.1.2.1 and 3.1.3.3 of the Analysis chapter. The idea of doing a full in depth analysis based on Normalized System Theory and Evolutionary Architectures is just enormous. It had to be scoped down in order to reach something useful in context of this thesis. For this reason I used the principles for my analysis, established to satisfy goal G2, in the evaluation. Even this provided many interesting observations and sparked some ideas for future work.

G5 Reason about approaches to convert presentation layer of an application

Section 3.2 Transition approaches is fulfilling this goal. I presented two very different paths for transitioning an application. One is a rewrite approach the other is an incremental one. I described their benefits and drawbacks as I see them and gave some examples that I know of.

G6 Implement example application and upgrade its GUI layer (use chosen technologies)

This goal is satisfied by the chapter 4 Case study. In this chapter I first present the domain of the example application and go over pre-requisites for a GUI upgrade. I first implemented the application using WinForms and transitioned into WPF. I limited myself to upgrading just part of the GUI as the case study aims to provide overview of the process in the first place.

G7 Specify which concepts are costly or limits the transition between chosen technologies

This goal is achieved in the summary of my case study, section 4.3. I point out the difficult parts of the process of transitioning in my example project. The summary is also answering the more general view of transitioning between any two GUI technologies.

G8 Summarize knowledge needed to ease transition between GUI technologies

The topic of knowledge to ease transition is spread across this whole thesis. Summaries of all the sections combined could serve as this knowledge overview, but I repeat it shortly here. The main points are that before migration we should know the following for initial and intended states. What are the paradigms, frameworks capabilities, their interoperability, testing systems, and what is the chosen transition approach. These are the main aspects from my point of view.

Conclusion

Following the above list of all resolved goals I want to share some ideas and observations that I gathered during my work on this thesis.

First of all an overview. I looked at evolvability of UI technologies in a broader sense. Presenting the different paradigms of GUI systems. Diving into methodologies that focus around the concept of evolvability. And finally I worked this knowledge into some examples on specific technologies. I discovered that there are many preparations and considerations that are related to migrating an application between GUI technologies. There are questions that needs answers to make the process successful or even possible. Questions like: What is my initial and intended state? What are the paradigms I use? What capabilities my frameworks have? And many others. I tried to provide a way to find an answer for these questions.

Second, is the topic of the GUI paradigms. I want to express once again that there is no clear consensus on their implementation nor their definition. It can be difficult to communicate the ideas behind them. That is why I did an overview of them and tried to make clear what I am describing. I also observe a trend in the paradigms in term of complexity which is rising. And makes understanding of these architectures more and more difficult.

Third, I had to limit myself in the depth I use the NST and EA methodologies. I see a big potential in comparing these approaches and their possible symbiosis. During evaluating my selected frameworks I ran into the need of scale for evolvability. Currently there is no way to objectively compare if one approach is more evolvable than other.

Next, I have to point out the many levels of scoping I did. I limited myself in terms of evaluation using common topics of described methodologies. I chose to work with two desktop frameworks both using the .NET environment. My

CONCLUSION

case study is limited to a small naive application. I can see each of these limits being removed as an expansion of this thesis in a future work.

Lastly, the most valuable output of this thesis in my eyes is the overview of all the aspects that play a role in a GUI transition.

Bibliography

- [1] Wittern, E.; Suter, P.; et al. A Look at the Dynamics of the JavaScript Package Ecosystem. In *Proc. 13th Working Conference on Mining Software Repositories (MSR)*, IEEE, 2016, pp. 351–361. Available from: <https://ieeexplore.ieee.org/document/7832914>
- [2] Schaller, R. R. Moore’s law: past, present and future. *Spectrum, IEEE*, volume 34, no. 6, 1997: pp. 52–59.
- [3] The Law of Accelerating Returns, [online]. [cit. 2019-03-16]. Available from: <http://www.kurzweilai.net/the-law-of-accelerating-returns>
- [4] Errich, G.; Richard, H.; et al. *Design Patterns, Elements of Reusable Object-Oriented Software*. 75 Arlington Street, Suite 300, Boston: Addison-Wesley Longman Publishing Co., Inc., 1995, ISBN 0201633612.
- [5] Fowler, M. GUI Architectures, [online]. [cit. 2019-02-28]. Available from: <https://www.martinfowler.com/eaDev/uiArchs.html>
- [6] Steve Burbeck, P. Applications Programming in Smalltalk-80(TM): How to use Model-View-Controller (MVC), [online]. [cit. 2019-03-01]. Available from: <https://web.archive.org/web/20120729161926/http://st-www.cs.illinois.edu/users/smarch/st-docs/mvc.html>
- [7] MVP: Model-View-Presenter The Taligent Programming Model for C++ and Java, [online]. [cit. 2019-03-01]. Available from: <http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>
- [8] Model 1 and Model 2 (MVC) Architecture, [online]. [cit. 2019-03-01]. Available from: <https://www.javatpoint.com/model-1-and-model-2-mvc-architecture>

BIBLIOGRAPHY

- [9] Overview of ASP.NET Core MVC, [online]. [cit. 2019-03-01]. Available from: <https://docs.microsoft.com/en-us/aspnet/core/mvc/overview?view=aspnetcore-2.2>
- [10] Presentation Model, [online]. [cit. 2019-03-01]. Available from: <https://www.martinfowler.com/eaaDev/PresentationModel.html>
- [11] Introduction to Model/View/ViewModel pattern for building WPF apps, [online]. [cit. 2019-03-02]. Available from: <https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewviewmodel-pattern-for-building-wpf-apps/>
- [12] ZK, [online]. [cit. 2019-03-01]. Available from: <https://www.zkoss.org/>
- [13] Model-View-Intent, [online]. [cit. 2019-03-19]. Available from: <https://cycle.js.org/model-view-intent.html>
- [14] Mannaert, H.; Verelst, J.; et al. *Normalized Systems Theory*. Heerveldstraat 4a, 3510 Kermt: NSI Press powered by Koppa, 2016.
- [15] Ford, N.; Parsons, R.; et al. *Building Evolutionary Architectures*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, Inc., 2017.
- [16] Conway, M. E. How Do Committees Invent? *Datamation*, April 1968: pp. 28–31. Available from: <http://www.melconway.com/Home/pdf/committees.pdf>
- [17] Google Trends - WPF, WinForms, [online]. [cit. 2019-03-29]. Available from: <https://trends.google.com/trends/explore?date=all&q=%2Fm%2F05q185,%2Fm%2F03f8ny>
- [18] GitHub - Topic: WinForms, [online]. [cit. 2019-03-29]. Available from: <https://github.com/topics/winforms>
- [19] GitHub - Topic: WPF, [online]. [cit. 2019-03-29]. Available from: <https://github.com/topics/wpf>
- [20] DevExpress, [online]. [cit. 2019-03-29]. Available from: <https://www.devexpress.com/#ui>
- [21] Telerik, [online]. [cit. 2019-03-29]. Available from: <https://www.telerik.com/>
- [22] Windows Forms, [online]. [cit. 2019-03-29]. Available from: <https://docs.microsoft.com/en-us/dotnet/framework/winforms>
- [23] Windows Presentation Foundation, [online]. [cit. 2019-03-29]. Available from: <https://docs.microsoft.com/en-us/dotnet/framework/wpf>

- [24] .NET Core 3 and Support for Windows Desktop Applications, [online]. [cit. 2019-03-29]. Available from: <https://devblogs.microsoft.com/dotnet/net-core-3-and-support-for-windows-desktop-applications/>
- [25] Windows Presentation Foundation (WPF), [online]. [cit. 2019-04-04]. Available from: <https://github.com/dotnet/wpf>
- [26] Walkthrough: My first WPF desktop application, [online]. [cit. 2019-04-04]. Available from: <https://docs.microsoft.com/en-us/dotnet/framework/wpf/getting-started/walkthrough-my-first-wpf-desktop-application>
- [27] Golian, C. *Migration of relational databases using CodiScent's Projective Technologies*. Bachelor's thesis, Czech Technical University in Prague, 2015.
- [28] Things You Should Never Do, Part I, [online]. [cit. 2019-06-04]. Available from: <https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/>
- [29] StranglerApplication, [online]. [cit. 2019-11-04]. Available from: <https://martinfowler.com/bliki/StranglerApplication.html>
- [30] Stevenson, C.; Pols, A. An Agile Approach to a Legacy System. In *XP 2004: Extreme Programming and Agile Processes in Software Engineering (EDS)*, Springer, 2004, pp. 123–129. Available from: <http://cdn.pols.co.uk/papers/agile-approach-to-legacy-systems.pdf>
- [31] Anti-Corruption Layer pattern, [online]. [cit. 2019-11-04]. Available from: <https://docs.microsoft.com/en-us/azure/architecture/patterns/anti-corruption-layer>
- [32] Strangler Applications, [online]. [cit. 2019-15-04]. Available from: <https://paulhammant.com/2013/07/14/legacy-application-strangulation-case-studies/>
- [33] Fowler, M. *Refactoring*. Addison-Wesley Signature Series (Fowler), Boston, MA: Addison-Wesley, second edition, 2018, ISBN 978-0-13-475759-9.
- [34] Modernization, [online]. [cit. 2019-15-04]. Available from: https://www.standishgroup.com/sample_research_files/Modernization.pdf
- [35] Selenium - Web Browser Automation, [online]. [cit. 2019-20-04]. Available from: <https://www.seleniumhq.org/>

- [36] Ranorex - TEst Automation for GUI Testing, [online]. [cit. 2019-20-04]. Available from: <https://www.ranorex.com/>
- [37] TextComplete - Automated UI Testing Tools, [online]. [cit. 2019-20-04]. Available from: <https://smartbear.com/product/testcomplete/overview/>
- [38] Hamill, P. *Unit test frameworks - a language-independent overview*. O'Reilly, 2005.
- [39] Noble, S.; Bourton, S.; et al. *WPF recipes in C# 2008: a problem-solution approach*. Apress, 2008.
- [40] Samir, H.; Kamel, A. Automated reverse engineering of Java graphical user interfaces for web migration. *2007 ITI 5th International Conference on Information and Communications Technology*, 2007, doi: 10.1109/itict.2007.4475638.
- [41] NSX: Normalized Systems, [online]. [cit. 2019-29-04]. Available from: <https://normalizedsystems.org/>
- [42] Esposito, D.; Saltarello, A. *Microsoft .NET - Architecting Applications for the Enterprise, 2nd Edition*. Redmond, Washington 98052-6399: Microsoft Press, 2015.
- [43] Fowler, M. *Patterns of Enterprise Application Architecture*. 75 Arlington Street, Suite 300, Boston: Addison-Wesley Longman Publishing Co., Inc., 2011.
- [44] Krasner, G. E.; Pope, S. T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, volume 1, no. 3, 1988: pp. 26–49. Available from: <https://www.lri.fr/~mbl/ENS/FONDIHM/2013/papers/Krasner-JOOP88.pdf>
- [45] Reactive Apps with Model-View-Intent, [online]. [cit. 2019-03-20]. Available from: <http://hannedorfmann.com/android/mosby3-mvi-1>

Acronyms

EA – Evolutionary Architecture

GUI – Graphical user interface

MVC – Model View Controller

MVI – Model View Intent

MVP – Model View Presenter

MVVM – Model View ViewModel

NST – Normalized System Theory

PM – Presentation Model

WinForms – Windows Forms

WPF – Windows Presentation Foundation

WYSIWYG – What you see is what you get

XAML – Extensible Application Markup Language

XML – Extensible markup language

Contents of enclosed SD card

	readme.txt	the file with SD card contents description
	exe	the directory with executables
	data	data for the case study app
	initial_state	initial state exe of case study app
	intended_state	intended state exe of case study app
	src	the directory of source codes
	git_repository	case study git repository
	thesis	the directory of L ^A T _E X source codes of the thesis
	text	the thesis text directory
	thesis.pdf	the thesis text in PDF format