UNIVERSITY OF VAASA

FACULTY OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

Arvi Lehesvuo

A SOFTWARE FRAMEWORK FOR STORING USER WORKSPACES OF DESKTOP APPLICATIONS

Master's thesis in Technology for the degree of Master of Science in Technology

submitted for inspection, Vaasa, June 30, 2012.

Supervisor	Prof. Jouni Lampinen
Instructor	M.Sc. Pasi Pelkkikangas

TABLE OF CONTENTS **page**

ABBREVIATIONS

C#	C Sharp programming language is a multi-paradigm programming language developed by Microsoft.
.NET	The .NET Framework is a software framework developed by Microsoft. .NET runs primarily on Microsoft Windows operating system.
PC	Personal Computer is any general-purpose computer whose size and capabilities makes it useful for individuals.
UML	Unified Modeling Language is a standardized general-purpose modeling language in the field of object oriented software engineering.
IoC	Inversion of Control is an object oriented programming practice whereby the object coupling is bound at run time.
MVC	Model–View–Controller is an architectural pattern used for building user interfaces.
MVP	Model–view–presenter is a derivative of the MVC pattern, also used mostly for building user interfaces.
MVVM	Model-View-ViewModel is an architectural pattern targeted in modern UI development, originated from Microsoft.
OOP	Object Oriented Programming is a programming paradigm that uses abstraction to create models based on the real world.
CAN	CAN bus for Controller Area Network is a communication standard designed to allow microcontrollers and devices to communicate with each other.
GUI	Graphical User Interface is a type of user interface that allows users to interact with electronic devices with images rather than text commands.
SDI	Single Document Interface is a method of organizing graphical user interface applications into individual windows that are handled separately by the operating system.
TDI	Tabbed Document Interface is a GUI model that allows multiple documents to be contained within a single window, using tabs as a navigational widget.
MDI	Multiple Document Interface is a GUI model where windows reside under a single parent window.
API	Application Programming Interface is a source code-based specification intended to be used as an interface by software components to communicate with each other.

LIST OF FIGURES **page**

VAASAN YLIOPISTO
Teknillinen tiedekunta
Tekijä: Arvi Lehesvuo
Diplomityön nimi: Ohjelmistokehys käyttäjän työtilojen tallentamiseen työpöytäsovelluksissa
Valvojan nimi: Jouni Lampinen
Ohjaajan nimi: Pasi Pelkkikangas
Tutkinto: Diplomi-insinööri
Koulutusohjelma: Tietotekniikan koulutusohjelma
Suunta: Ohjelmistotekniikka
Opintojen aloitusvuosi: 2007
Diplomityön valmistumisvuosi: 2012 **Sivumäärä:** 68

TIIVISTELMÄ:

Työpöytäsovellusten käyttöliittymien toteuttamiseen liittyy useita suunnitteluongelmia. Useimmat näistä ongelmista voidaan ratkaista olemassa olevan suunnittelumallin tai ohjelmistokomponentin avulla. Tämä tutkimus keskittyy yhden tällaisen työpöytäsovellusten kehittämiseen liittyvän suunnitteluongelman, käyttäjän työtilojen tallentamisen ja palauttamisen tutkimiseen.

Työn tavoitteena on ratkaista esitetty ongelma rakentamalla suunnittelumalli ohjelmistokomponentille, jota voidaan hyödyntää ongelman ratkaisemiseen tulevissa ohjelmistoprojekteissa. Tarkoituksena on myös toteuttaa tämä malli ulkoisena ohjelmistokirjastona, sekä testata sen soveltuvuutta käytännössä.

Tutkimuksen aikana kehitetty ratkaisu rakennetaan tapaustutkimuksen kautta teollisuudelle ohjelmistoalihankintaa tarjoavassa yrityksessä. Tutkimuksen tarkoituksena on rakentaa ratkaisu reaalimaailman ongelmaan, joten tutkimusmetodologiaksi valittiin konstruktiivinen tutkimusote.

Aluksi työssä käydään läpi ja pohdiskellaan asiaan liittyvää teoriaa, sekä ratkaisulle kerätään vaatimuksia tutkimalla kohdeorganisaation olemassa olevaa projektidokumentaatiota. Työssä pohditaan myös miten ratkaisu on mahdollista rakentaa ulkoisena ohjelmakirjastona. Myöhemmin kerättyä teoreettista tietoa sovelletaan käytännössä toteuttamalla ohjelmakirjasto.

Ohjelmakirjaston käytännön soveltuvuutta testataan ottamalla se käyttöön useassa erityyppisessä sovellusprojektissa. Testauksen suorittaa ryhmä kohdeorganisaatiossa työskenteleviä ohjelmistosuunnittelijoita. Myöhemmin suunnittelijoille suoritetulla kyselyllä todistetaan, että ohjelmakirjasto täyttää sille asetetut vaatimukset.

AVAINSANAT: ohjelmistotuotanto, suunnittelumalli, ohjelmistokehys, työtila, työpöytäsovellus

UNIVERSITY OF VAASA
Faculty of technology
Author: Arvi Lehesvuo
Topic of the Thesis: A Software Framework for Storing User Workspaces of Desktop Applications
Supervisor: Jouni Lampinen
Instructor: Pasi Pelkkikangas
Degree: Master of Science in Technology
Degree Programme: Degree Programme in Computer Science
Major of Subject: Software Engineering
Year of Entering the University: 2007
Year of Completing the Thesis: 2012 **Pages:** 68

ABSTRACT:

There are many design problems faced in user interface design of desktop applications. For most of the problems there is some suitable design pattern or existing software component to cope the problem, without having to spend too much design time on it. This research concentrates on one design problem repeatedly faced when designing desktop applications; storing and restoring user workspaces.

The main goal of this thesis is to solve presented design problem by constructing a design model for a software component which can be used in the upcoming application projects. The aim is also to build this design as an external software framework and to test its applicability in practice.

The solution developed during this research will be built and evaluated through a case study at an industrial software subcontractor company. Constructive research approach is used as the research method for this study, because the purpose of this thesis is to produce a practically relevant solution for an explicit problem, which is usually the baseline of a constructive study.

At first, the theory related to the subject is presented and discussed. Existing project documentation in the case organization is studied to gather the requirements for the solution. How to create the solution as a reusable software framework is also discussed. Collected theoretical knowledge is then applied in practice by building the software framework.

Practical relevance of the software framework is evaluated by deploying it to different types of application projects. The testing is performed by a group of software designers working in the case organization. A questionnaire then held for the software designers shows that the developed framework succeeds to fulfill its requirements.

KEYWORDS: software engineering, design pattern, framework, user workspace, desktop application

1. INTRODUCTION

The usage of PC software tools in part of industrial environment is constantly growing. In the past, seven segment displays or LCD-screen, later substituted by embedded touch screen devices were mainly used as user interfaces in embedded device control. Since the computers are constantly getting smaller while processing power is getting better, tablet PCs are increasing their market share also in the industrial environment.

Embedded touch screen devices usually require special implementation, because of limitations in device performance and for this reason designing software to operate on them requires special architecture, usually with reduced functionalities. Nowadays tablet devices are starting to be as powerful as normal desktop PCs and using same operating systems. This is why it is possible to design same software to be compatible in all the devices, tablets as well as laptops and desktops. This thesis focuses on applications developed for personal computers used in machinery control.

There are many design problems faced in user interface design of desktop applications. For most of the problems there is some suitable design pattern or existing software component to cope the problem, without having to spend too much design time on it. This research concentrates on one design problem repeatedly faced with designing desktop applications. Although same design problem can be found on almost any kind of desktop applications, this thesis focuses on software used for controlling or monitoring of some kind of machinery.

Usually parameterization of a device or a set of devices is not a fast task to do and it might require several workdays. Sometimes task started on one day, will continue after few days from the same situation it was left on. This creates need to make a checkpoint, from which to continue later. This kind of checkpoint can be created by storing the workspace user has created on the application. Checkpoint can be then restored to the same state on some other time. In industrial control software workspace might contain for example opened connections to different devices, parameterization files, configuration data, monitoring views, etc.

In a preliminary research, the target organization has not managed to find any free or commercial solution for resolving this design problem easily. Most of the software developed, the problem is usually solved by creating some custom solution to store whatever is required for that application. The purpose of this research is to find a way to create a reusable design to overcome this issue in upcoming application projects without spending too much project resources and work time.

The demand for this research came from actual real-world need to develop similar solutions for two applications at same time. Other application is used to parameterize variable speed AC drives and other one is used to control hydraulic device for tightening motor parts. In both of these cases there were need to continue work from the same situation after some period of time elapsed in between different process steps.

This research pursues to construct a design model to be suitable for as many software languages and platforms as possible, but the reference implementation presented mainly targets software designed for Windows operating systems using C# programming language and .NET Framework.

The solution developed during this research is built and evaluated through a case study at an industrial software subcontractor company called Wapice Ltd. While writing this thesis the author has been working as a software engineer in the target organization. Constructive research approach was chosen as the research method for this study, because the purpose of this thesis is to produce a practically relevant solution for an explicit problem, which is usually the baseline of a constructive study.

1.1. Company Introduction: Wapice Ltd

The subject for this thesis was provided by a company called Wapice Ltd. Wapice is an independent information technology service company, currently employing 190 software professionals. Wapice is specialized in solving the problems faced by industrial companies and designing solutions to meet their needs. Wapice organization structure is divided in three different segments; Embedded Systems, Industrial Systems

and Business Solutions. Provider for this thesis subject in Wapice organization was the Industrial Systems segment, since one of its tasks is to design graphical user interface solutions for industrial needs.

1.2. Professional Background Statement

The author of this thesis is a Master of Science student in Computer Science department at the Vaasa University, majoring in Computer Science. In addition to educational background he has worked slightly over six years as a software engineer in Wapice Ltd designing and developing industrial software solutions. First couple of years he has worked mainly with embedded systems, but for last four years he have been specializing in PC tool development, mainly with C# and .NET Framework. In his current work his interests include especially beneficial and profitable exploitation of object oriented software design patterns, especially with user interface design techniques.

1.3. The Scope of the Research

The main purpose of this study is to find a way to easily solve a practical problem, which is faced in most of the user interface application developed. The results of the study should provide a design for a software component that can be used in future applications and this way to reduce development time in the beginning of new projects. The goal is also to implement this design as an external software framework and test its applicability in practice.

At first, the theory related to the subject is presented, then explaining how this theory is applied in the actual design. It is expected that the reader of this research is familiar to computer programming in some level and is capable of reading basic UML class diagrams. Since the actual need for this research came from two different desktop applications both implemented using C# programming language and .NET Framework, the practical solution constructed in this thesis is implemented using these same techniques.

1.4. Research Questions

Following research questions were raised in order to achieve the targets set:

1. How to design a solution for saving user workspace of a desktop application?

This is the main research question and forms the basis for the rest of the research questions. To answer this question it is required to:

- Study existing projects and upcoming project documentation for gathering the requirements for the solution.

- Identify all the common design problems faced in these projects to be able to build a design, which provides solutions for these problems.

The answer for these questions will be given based on the research done in the target company.

2. How to build the solution as a reusable software component?

After identifying the actual problems and finding the requirements for the design, it is needed to study, if it is possible make one reusable solution, which solves problems in all the studied software projects at once. To solve this issue, it is required to:

- Recognize what kind of challenges building the solution as an external component will represent.

- Understand software architectural needs of potential projects, which shall be using the constructed solution.

- Study different GUI programming patterns to be able to design commonly suitable solution.

The answer to these questions will be given based on literature review.

3. How does the constructed solution help to improve the efficiency of GUI application development within the target organization?

The last question helps to evaluate the results of this study. The answer will be reclaimed through a questionnaire held for software designers who have tested the solution by deploying it in to their project.

2. RESEARCH METHODOLOGY

In this chapter, the research methodology used in this thesis is presented. At first, the constructive research approach is explained, and then followed by a description how it is applied in this research.

Constructive research is a research approach that can be described as managerial problem solving through the construction of models, diagrams, plans, organizations, etc. Several examples of constructive approach can be found in operations research, management accounting, and clinical medicine as well as in technical sciences (Kasanen, Lukka & Siitonen 1993: 245). The reason for selecting the constructive research approach was the need for doing practically relevant research for solving a real world problem. The constructive research approach was reported to suit well for this purpose (Kasanen *et al*. 1993; Lukka 2003).

2.1. Constructive Research Approach

The constructive research approach is a research procedure for creating innovative constructions that pursues to solve problems faced in the real world, and by this way to make a theoretical contribution on the subject where it is applied. Case studies using constructive research approach aims at creating knowledge how research problem can be solved with a new innovation, or how the new solutions is better than already existing ones. Typical for these solutions is that they are invented and developed, not discovered. (Lukka 2003: 83-84.)

Mathematical algorithms or other new mathematical entities provide examples of theoretical constructions. As well as in medicine, development of a new pharmaceuticals or a treatment can be considered as constructions (Kasanen *et al*. 1993: 245). Characteristic for constructive research is that it focuses in real-world problems felt relevant to be solved in practice. Research should produce an innovation to solve the problem while implementing an construction that can be tested in practice. It is common that researcher works very closely to the practitioners in a team-like co-

operation, where practical learning is expected to take place (Lukka 2003: 84). It is also important that prior theoretical knowledge on the subject is studied and the empirical findings on the research are reflected back to the theory (Lukka 2003: 84). The key elements of constructive research approach are illustrated in figure 1.

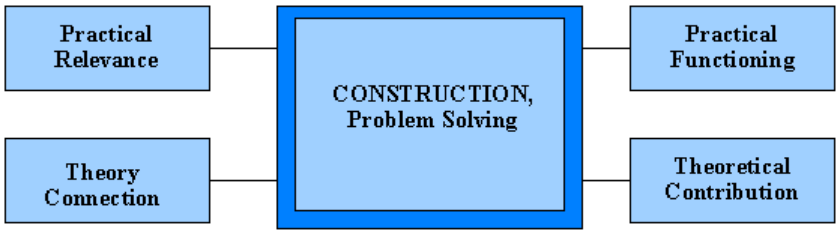

Figure 1. Elements of Constructive Research (Kasanen *et al*. 1993: 246.)

In an ideal case the constructive research produces an resolution to the real-world problem by producing a new construction with both practical and theoretical contributions (Lukka 2003: 85). A good practical solution would be probably satisfactory result for project stakeholders, however a fail on practical level can still produce significant theoretical results.

2.2. Research Process as Applied in This Research

According to Lukka (2003), there are seven crucial steps in the constructive research approach. This chapter introduces these steps followed by an explanation how the steps are applied in this research.

(1) Find a practically relevant problem, which also has potential for theoretical contribution.

(2) Examine the potential for long-term research co-operation with the target organization(s).

(3) Obtain a deep understanding of the topic area practically and theoretically.

(4) Innovate a solution idea and develop a problem solving construction with theoretical contribution potential.

(5) Implement the solution and test how it works in practice.

(6) Examine the scope of the solution's applicability.

(7) Identify and analyze the theoretical contribution.

Although the steps are presented in sequential order, they are not necessarily executed in that order, since the research process is usually highly iterative. This research was no exception, since new solution ideas were discovered during the implementation phase and the solution were partly re-designed few times during the implementation.

Following explains how the seven steps of research are applied in this study:

1. Selecting the research topic is the most important phase in every research project. The research outcome should produce a practical contribution, but also have good research potential (Lukka 2003: 86). In this study the research problem is that software developers in the case organization implement different solutions to solve similar design problem, again in several projects, producing yet another solutions that cannot be reused because its design does not fulfill requirements set by other projects. A solution for this problem would be designing a reusable software component, which could be taken into use in every similar software projects, this way saving a lot of development time during the project cycle. However this kind of readymade solutions does not yet exist.

2. The idea for the research project came from the author himself, while working in the target company as a software designer on two different projects with similar needs for user workspace storing. Case organization agreed that research should be done, because it would probably save total design time on these two projects, but also provide a readymade solution for upcoming projects. Small review group were gathered from the

organization experts to review study outcome and provide knowledge during the research process.

3. The theoretical knowledge for this project was gathered studying relevant literature on object oriented software patterns and GUI design models. There are several different ways of implementing GUI software, so obtaining adequate knowledge of most common patterns was crucial to be able to produce commonly reusable solution. Practical knowledge of projects implementation techniques were mainly gathered by from software experts in case organization. Recognizing needs of different projects was necessary for developing construction that is really useful and beneficial for upcoming projects of the organization.

4. During this step suitable techniques were selected for creating a solution idea and basic design for construction were built. According to Lukka (2003: 87) this is the critical step in the research, since if innovative construction cannot be designed there is no point of going forward with the project. In this study the construction is a software framework designed to provide certain basic functionality and this way improving efficiency of developing graphical user interface applications.

5. Next, the actual solution was implemented according to the model designed in previous phase. Also pilot software was implemented to test how functionalities of the constructed software framework worked in practice with different GUI design patterns. Also a test group of software designers in the target organization deployed the software framework to their projects.

6. During this phase of the process researcher should step back from the empirical work and analyze the results together with the target organization (Lukka 2003: 88-89). At this point the results of the final solution were analyzed. A questionnaire was made for the software designers who tested the framework to grade its functionality and usefulness. Eventually conclusions based on user questionnaire were drawn.

7. From the academic point of view it is crucial that the researcher is able to explicate the theoretical contribution of the project (Lukka 2003: 89). In this final phase the theoretical contribution of this project will be identified and analyzed.

Table 1. Relations between research steps and chapters of this study

Step in research process	Chapters where applied
1. Find a practically relevant problem, which also has potential for theoretical contribution.	Chapter 1
2. Examine the potential for long-term research cooperation with the target organization(s).	Chapter 1
3. Obtain a deep understanding of the topic area practically and theoretically.	Chapter 3, 4
4. Innovate a solution idea and develop a problem solving construction with theoretical contribution potential.	Chapter 4
5. Implement the solution and test how it works in practice.	Chapter 5
6. Examine the scope of the solution's applicability.	Chapter 5
7. Identify and analyze the theoretical contribution.	Chapter 6

3. THEORY AND BACKGROUND OF THE STUDY

There are many different programming tools and patterns available for creating programs containing graphical user interface. However the target organization has not managed to find a reusable component for storing user workspaces of desktop applications. Although this kind of ready-made component could be found through a more extensive research, the target organization decided to build their own solution to be able to tailor it when needed, according to their needs.

This study is limited to two of the most commonly used programming languages at the target organization; C# and Java. Both of the techniques mentioned are object oriented programming languages. To understand the core of the research problem it is crucial to have some basic knowledge of object oriented programming and related theory. This chapter introduces reader to basics of object oriented programming and presents some commonly used software patterns applied in this study.

3.1. Introduction to the Subject

Since this thesis mainly studies and considers issues around object oriented programming and related design patterns, this chapter introduces reader to the basics of object oriented programming. Also some of the basic patterns used and some more advanced software libraries used to construct of this study will be presented.

3.2. History of Object Oriented Programming

“Object: A thing presented to or capable of being presented to the senses.” This is one of the phrases can be found by looking it up from the dictionary. It generally means that an object can be just about anything. The word oriented in turn is usually defined as “directed towards”. With this logic, object oriented can be defined as; directing towards just about anything you can think of. No wonder that the software industry have been struggling coming up with the agreed definition “object oriented”. (Meilir 1999: 1-2.)

So what is the object oriented programming? It is commonly believed that object oriented programming (OOP) is product of 1980's, when the C language was moved into object oriented world by Bjarne Stroustrup creating C++ language. Actually earliest object oriented languages were created in the 1960's. They were called SIMULA 1 (1962) and Simula 67 (1967). These earlier languages contained most of the OOP advantages, but it was not until the 1990's when C++ made its breakthrough and OOP began to truly flourish. (Purdum 2008: 4-5.)

Next step in OOP development was 1991, when James Gosling, Mike Sheridan, and Patrick Naughton initiated project to develop new programming language for intelligent electronic devices capable of being centrally controlled. Language was originally named as Oak after oak tree outside Goslings window. Language was later renamed as Java, since they discovered that name Oak was already used by another programming language. (Purdum 2008: 4-5.)

Java was quickly noticed by the big audience and grew popularity along rapidly growing popularity of the World Wide Web. Java was shortly integrated part of various web browsers and the ability to run Java on web browsers improved web functionalities significantly, this way providing more boost to the growth of the World Wide Web itself. It is typical for Java, that applications are compiled to byte-code, which can be run on any Java Virtual Machine regardless of the computer architecture. (Purdum 2008: 4-5.)

C#, which is by many programmers said to be Microsoft's answer to Java, is a programming language developed 2001, by team lead by Anders Hejlsberg. According to the C# Language Specification, it is intended to be a simple, modern, and general-purpose, object oriented programming language. C# contains several features aiding in the construction of robust and durable applications. Its garbage collection automatically reclaims memory occupied by unused objects, so programmer does not basically have to take care of memory handling. Exception handling provides extensible approach to error

detection and recovery. Also type-safe design makes it impossible to have uninitialized variables or to perform unchecked type casts. (Ecma International, 2006)

3.3. Most Central Elements in Object Oriented Programming

To understand the ultimate reason for this research, it is crucial to understand the basics of programming and more closely the basics of object oriented programming. Since this is quite large subject and it cannot be fully covered in this thesis, only the central elements of object oriented programming will be introduced. According to Meilir (1999: 3) there are nine central concepts in object orientation; encapsulation, information hiding, state retention, object identity, messages, classes, inheritance, polymorphism and genericity. In this chapter these elements will be introduced with small examples. Also concept of class access modifiers will be introduced.

Encapsulation is one of the basic concepts in software that are almost as old as software itself. Meilir (1999: 9) defines encapsulation as grouping of related ideas into one unit, which can thereafter be referred to by a single name.

When it was noticed, that same patterns would appear multiple times inside same program, it was invented to call the same block of code from multiple points of the program by same name, which created the concept called as subroutine. Using subroutines was a good way of saving computer memory and also presented a clear way for programmers to consider and manipulate certain block of code as a single idea.

Object oriented encapsulation has an idea similar to subroutines, but is structurally more sophisticated. An object can consist of a set of attributes and operations. Attributes represents some information that the object holds. Operation in turn is a procedure or a function which can be used for modifying the object attributes or providing some other functionality. In object oriented encapsulation can be defined as packaging operations and attributes into an object type, which state can be accessed or modified through interface provided by the encapsulation. Basically each object operation or attribute,

which are visible through this interface can be modified by other objects. (Meilir, 1999: 9-12)

Information / implementation hiding is the use of encapsulation to restrict the external visibility of certain information or implementation decisions that are internal to the encapsulation structure. This means that an external observer of an object can have full knowledge of what the object can do, but no knowledge of how the object will do it, or how it is constructed internally. Encapsulated unit, an object can be basically viewed from two directions, from outside and from inside. The outside view is called public and the inside view is called private view. Implementing some object attributes or functionality as private can be called information / implementation hiding. (Meilir 1999: 12-13.)

Information / implementation hiding has two major benefits. It localizes the design decisions, which means, that local implementation changes will have minimal impact on the whole system. It also decouples the local information from its representation form, which prevents external users becoming tied to any internal information format. (Meilir 1999: 14.)

State Retention in object oriented programming refers to the matter, that objects are able to remember their internal state. When a traditional procedural module (function, subprogram, etc.) returns to its caller the module dies, leaving only its results as its legacy. When the module is called again, it has no memory of its previous existence, so it acts same as it would be called the first time. But an object is aware of its past. For example a caller of an object may pass some information to the object, which can be retrieved later. Technically this means, that object can retain its state. (Meilir 1999: 14.)

Object Identity in object oriented programming can be described as a property by which each object can be identified and thus treated as a distinct software entity. There is a unique handle attached to each object in the creation phase by an object-handle mechanism. This handle remains with the object for its entire lifetime. Two objects can

never have the same handle and therefore can be always told apart. (Meilir 1999: 15-19.)

Messages are transactions, where an object requests another object to carry out activity. This means, that a sender object gives a target object demand to apply one of its methods. In order for sender object to send a message to the target object, it must know three things; handle of target object, name of the operation it wishes to execute and the arguments required in the execution of this operation. (Meilir 1999: 19.) Sending a message is usually referred as a method call.

Like subroutines, most messages pass arguments back and forth. The structure of the message to the target object is defined by a signature of the target operation or in other words by target objects method description. The method description consists of the operation name, list of input arguments and list of output arguments. In pure object oriented environment these arguments are not data, but object handles. (Meilir 1999: 21.)

Classes are needed to be able to describe an object containing some functionality or information. Basically class can be described as a model or a stencil for creating certain kind of objects. For example if an class called Engine is examined. Every time we execute new-statement, a new instance of a class Engine, an Engine-object is born. Each instantiated object are structurally identical to every other object created by the statement new Engine. This means that all the Engine-objects contain same operations and variables as the others. When these objects were programmed, the class Engine was written. (Meilir 1999: 27.)

There are two differences between objects created with same class: Each have own identity, meaning, that they contain different handle. Each object can also be in a different state than the others meaning that object variables can contain different data. Easy way of distinguishing objects from classes is to remember that classes are the ones programmed and objects are the ones created at run-time. (Meilir 1999: 28-29.)

Each object created contains bunch of attributes and operations, which means, that it consumes certain amount of memory. Although each object has same set of operations, called methods, same amount of memory are not allocated every time an object is created. Since methods contain only procedure code, single set of physical memory can be shared by all of the objects for storing the methods. Because all of the objects variables can have different data during run-time, physical memory cannot be shared same way among the objects for storing the variable data. (Meilir 1999: 29-32.)

Inheritance in object oriented programming is a way to reuse code of existing classes to create new classes containing slightly similar behavior without duplicating whole class code. It is a mechanism which can be used to avoid this extra work and also the maintenance issues that the duplicated code would later produce.

Inheritance can be more easily explained by an example, where two classes A and B will be examined. If class B is inherited from class A, it means that the objects of class B can make use of attributes and operations that would otherwise be available only to the objects of class A. Class A can then be termed a superclass of class B and class B subclass of A. Inheritance efficiently allows building of software in a way where class to cope with most general cases are first built, and then to deal with special cases, more specialized class can be created inheriting the first class. (Meilir 1999: 33.)

Although concepts of object and instance go almost synonymously together, it is important to distinguish these two from each other. By using inheritance it is possible to instantiate a single object from multiple classes. Although in most cases subclass inherits everything that superclass has to offer it is possible to cancel out some of the superclass operations by overriding them. Inheritance is not necessarily just a relation between two classes. It is possible for multiple classes to inherit same superclass. In some programming languages like C# it is also possible for class to have an arbitrary number of superclasses, which is then called multiple inheritance. (Meilir 1999: 35-38.)

The word **Polymorphism** comes from two Greek words that mean “many” and “form”. Object oriented textbooks contain two definitions for polymorphism, both valid and both properties of polymorphism bring a great deal of power to object orientation.

1. “Polymorphism is the facility by which a single operation or attribute name may be defined upon more than one class and many take on different implementations in each of those classes.”

2. “Polymorphism is the property whereby an attribute or variable may point to (hold the handle of) objects of different classes at different times.”

(Meilir 1999: 38.)

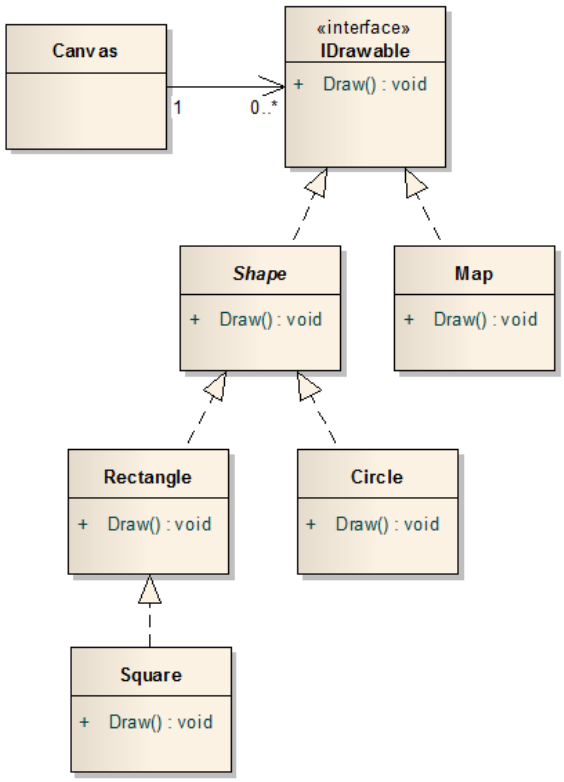

Figure 2. Polymorphism Example

Diagram in figure 2 presents an example where Canvas utilizes polymorphism by pointing to IDrawable-object list which contain shapes taking forms of Shape, Map, Rectangle, Square or Circle. When Canvas is drawn it can draw the list of IDrawable objects by calling draw-method of IDrawable-object. Although Canvas only points to list of IDrawable-objects, when the objects are drawn, a form of an object pointed by the Canvas appears to the screen, whether it is any of these objects in question.

Genericity is a construction of a class so that one or more of the classes that it uses is supplied at run-time, at the time that an object of class is instantiated (Meilir 1999: 43). Genericity is one of the most powerful means for obtaining flexibility in object oriented programming. Generic classes are very useful, when creating generic container-classes like groups, stacks, lists, trees, networks, etc. (Louhimies, 2000: 60).

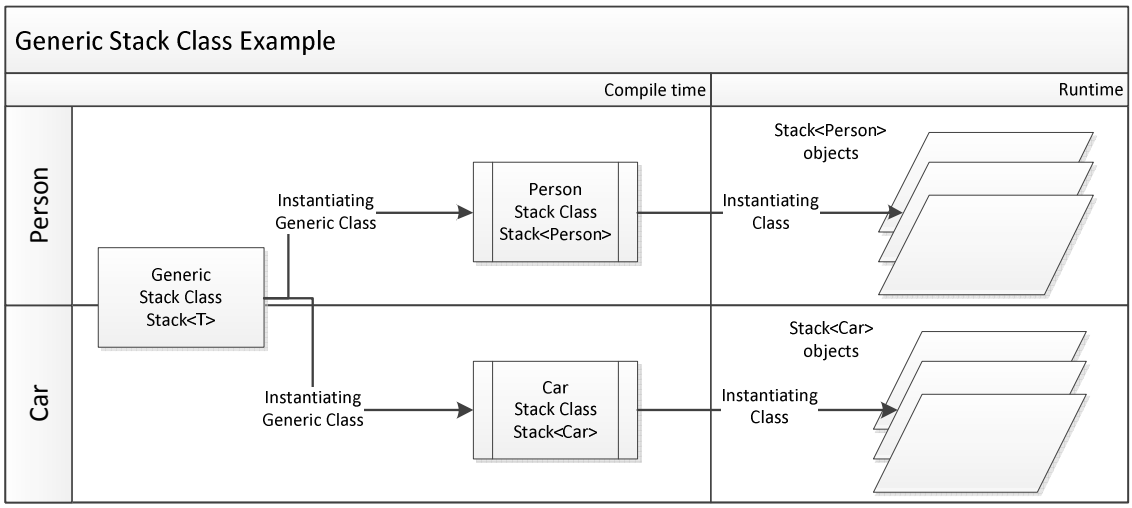

Figure 3. Example of Generic Class Usage

Figure 3 represents a stack, which contain objects defined as generic. When stack is constructed its content is defined by passing the used class to stack as a compile-time parameter. This enables using of same stack for basically any kind of objects. In the figure same stack is used for storing Car- and Person-objects. (Louhimies, 2000: 60.)

Visibility of Class Members or in other words the accessibility of classes, methods, and other members can be set by a functionality called access modifiers. Access modifiers are set by adding prefix keyword to class, attribute or method name. The functionality of these modifiers might vary little bit between some programming languages.

C# describes following modifiers; public, internal, protected and private. Public access is the most permissive access level, which means that public members can be accessed from anywhere without any restrictions. Protected members in turn are accessible only from within the class in which those are declared in, and from within any class derived from the original class declaring those members. Internal members are accessible only within the same assembly. Private members are accessible only within the class in which they are declared. In addition to these four types, internal and protected keywords can be used together. (Microsoft Corporation, 2010)

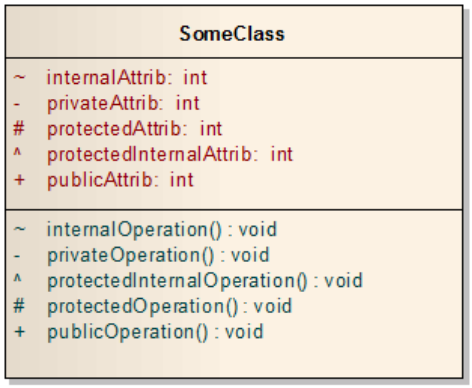

Figure 4. Different attribute and operation types presented in UML diagram

3.4. More Advanced Features of Object Oriented Programming

This chapter introduces some of the more advanced and powerful functionalities supported as a built-in feature by more advanced object oriented programming languages such as Java and C#.

Sobel & Friedman (1996) explains the concept of **Reflection** by reaching back to the study of self-awareness in artificial intelligence: “Here I am walking down the street in the rain. Since I'm starting to get drenched, I should open my umbrella.” This conceit reveals a self-awareness of behavior and state, that leads to a change in both, behavior and state. Reflection in computer science introduces these same capabilities to computer programming. Through reflection computer programs can examine themselves in order to make use of meta-level information in decisions about what to do next.

In computer programming reflection is basically a mechanism which by a programming can observe and modify its own structure and behavior at runtime (Hurlbutt, 1998). In object oriented programming languages such as Java and C#, the reflection allows examination of classes, interfaces, fields and methods at runtime without knowing their names at compile time. It also allows instantiation of new objects and invocation of methods. Reflection can also be used to adapt the program to different situations dynamically. (Oracle Corporation 1995; Microsoft Corporation 2005.)

Serialization in computer programming is a process of converting data structure or object state into a format that can be stored and resurrected later in the same or another computer environment. Cline (2011) compares serialization to Transporter on Star Trek: “it's all about taking something complicated and turning it into a flat sequence of 1s and 0s, then taking that sequence of 1s and 0s (possibly at another place, possibly at another time) and reconstructing the original complicated something.” Common use case for serialization is to store the needed data to a file, memory buffer or to transmit it across a network through a connection link. (Cline 2011.)

As one disadvantage of serialization can be mentioned the issue, that it can require breaking the opacity of an abstract data type by potentially exposing private implementation details (Miller 2003). Trivial implementations which serialize all data members may violate the encapsulation. For this reason some systems using the serialization to store data, deliberately obfuscate or encrypt the serialized data.

There are two types of serialization formats available: text & binary. Which one of these is better highly depends on the goal that is tried to be achieved (Miller 2003). Both has advantage and disadvantages as well. For example data serialized to text format is easier to interpret without any special tools, but that can be seen as disadvantage as well as advantage.

In the .NET languages, such as C#, classes can be serialized and deserialized by adding the Serializable-attribute to the class. Also Java provides automatic serialization, which requires that the serializable classes are marked by implementing the java.io.Serializable interface.

3.5. Software Design Patterns

A pattern is a way of doing something, a way of aiming towards of achieving a purpose, a technique. The idea of finding effective techniques of establishing something applies to many real-world efforts: making food, washing laundry, building house as well as implementing software. In each craft that have been practiced for a longer period of time, the people working on it start to find common and more effective methods on achieving their aims. When these methods in time are being documented, become those techniques a set of standardized patterns. (Metsker 2004: 1-2.)

In object oriented programming design pattern is a way of pursuing intent by using classes and their methods. A powerful usage of design patterns in object oriented programming is usually a good way of achieving the goal with fewer classes and cleaner code structure. Although there are tens of popular design patterns out there, this chapter only introduces patterns that contains some significance for understanding this research or patterns that will be someway applied in the construction of this research.

Interfaces and Abstract Classes are the base components commonly used in many design patterns. Generally speaking a class's interface is a collection of methods that the class allows objects of other classes to access. Interface usually represents a set of methods that will perform operations implied by their names. These operations are

typically described by code comments and other related documentation. Class's implementation of an interface lies within its methods, described in the implemented interface. (Metsker 2004: 9.)

In object oriented programming interfaces provides a way for unrelated objects to communicate with each other. Programming languages like C# and Java, which this solution is targeted to, have direct support for interfaces. This means that the interface description can be completely separated from the actual implementation. These programming languages also allow several classes to implement same interface or one class to implement several interfaces (Metsker 2004: 9.) In some older programming languages, which do not have direct interface support, can still have some features like pure virtual functions that supports the interface concept.

An abstract class with no non-abstract methods can be used fairly similar way as an interface; however it contains some differences important to understand: A class can implement multiple interfaces, but can only be a subclass of one abstract class. All the methods of an interface are effectively abstract, but an abstract class can have also non-abstract methods. An interface cannot declare or use variables, but an abstract class can. Access modifiers are not allowed in interface member declarations and all members of interface indirectly have automatically public access. An interface cannot declare constructors, but an abstract class can. (Metsker 2004: 346.)

Adapter Pattern, also known as a wrapper, is a pattern that pursues to convert an interface of a class to another interface that client expects. Adapter allows two classes to work together, that otherwise couldn't, because of incompatibility issues. Adapter pattern can be useful if some toolkit class that's designed for reuse can't be reused because its interface doesn't match the domain-specific interface that application requires. (Gamma, Helm, Johnson & Vlissides 1995: 139.)

There are two types of adapter patterns, object adapter pattern and class adapter pattern. In object adapter pattern presented in figure 5, adapter contains an instance of a class

that it wraps. With this pattern adapter makes calls to the instance of wrapped object. With class adapter pattern, adapter uses multiple polymorphic interfaces to achieve its goal. The adapter is created by implementing both, the interface that is expected and the existing interface.

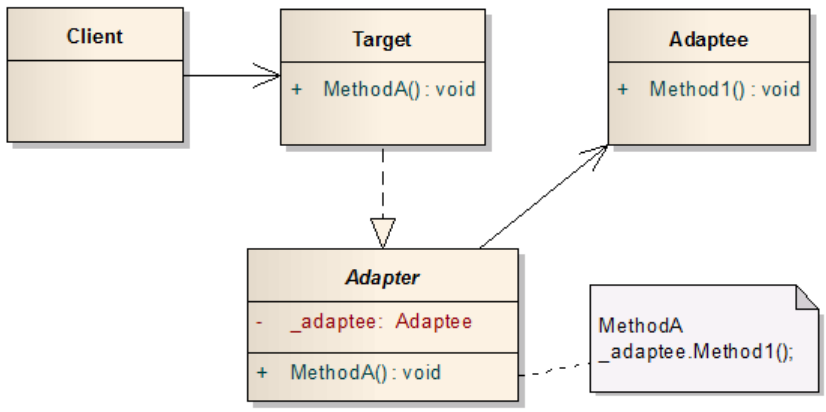

Figure 5. Structure of Object Adapter Pattern

Usage of adapter class is useful; when interface of existing class does not match the one needed. A reusable class that cooperates with unrelated classes is needed to be created. There is need to use several existing subclasses, but it is impractical to adapt their interface by implementing every one. (Gamma *et al*. 1995: 140.)

Frameworks use abstract classes to maintain relationships between objects. A software framework is often responsible of creating objects as well. Sometimes a framework class responsible of creating object might be implemented as abstract class. When both of these classes are abstract, clients using the framework have to subclass them to realize their application-specific implementations. In this case the creator class can't predict what kind of subclass to instantiate; it only knows when it should be instantiated. This creates a dilemma: the framework class must instantiate classes, but it only knows abstract classes, which cannot be instantiated. (Gamma *et al*. 1995: 107-108.)

The Factory Method Pattern offers solutions for this dilemma. It encapsulates knowledge of which subclass to create and moves knowledge out of the framework. Application subclasses redefine abstract method on creator class to return appropriate subclasses. This method can be called a factory method, since it is responsible for manufacturing required object. (Gamma *et al*. 1995: 107-108.)

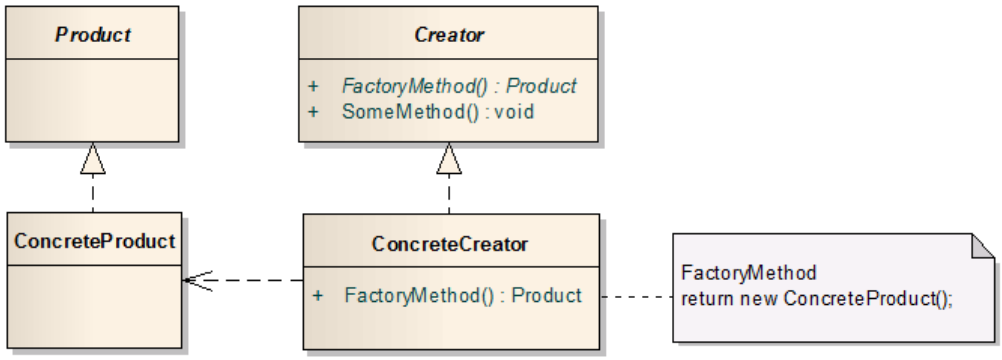

Figure 6. Structure of Factory Method Pattern

The usage of factory method pattern is needed, when a class can't anticipate the class of objects it must create or it wants its subclasses to specify the objects it creates (Gamma *et al*. 1995: 108). These are the main reasons, why this pattern is important to this research and crucial to understand when creating a design that should be implemented as an external software component.

Since the main focus of the research construction is to store and restore data, review of software pattern created for this purpose is required. One of the best patterns suitable for this task is a behavioral pattern called the Memento Pattern.

Design patterns that do not just describe model of objects or classes, but also the patterns of communication between them, can be called behavioral patterns. Behavioral patterns are concerned with algorithms and the responsibility assignment between objects. These patterns define complex control flow that can be difficult to follow at run-time. (Gamma *et al*. 1995: 221.)

When implementing undo mechanism or checkpoints to software it is sometimes necessary to record the internal state of an object. State information has to be stored somewhere so that it can be later used to restore objects back to their original state. Objects however usually encapsulate some of their internal state making it impossible to save this information externally. Exposing this state however would violate encapsulation, which can compromise the applications reliability and extensibility. (Gamma *et al*. 1995: 285.)

These problems can be overcome by applying memento pattern to the design. The memento pattern is implemented with two objects: an Originator and a Caretaker. The Originator is some object that has an internal state. The Caretaker in turn is an object that uses the Originator for some purpose and needs to be able to store its state. For example to be able to make a checkpoint from originators state, the Caretaker can ask the Originator for a memento object. To roll back to the state, it can return the memento object back to the Originator. The memento object itself is an opaque object which the caretaker cannot and should not change. (Gamma *et al*. 1995: 283-291.)

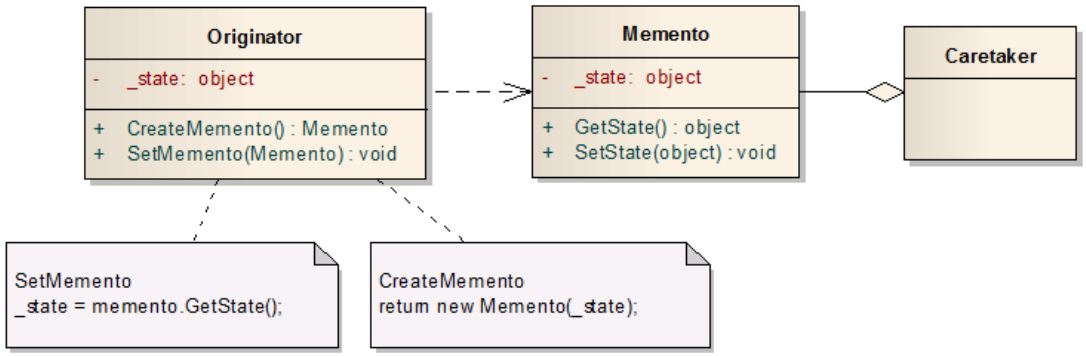

Figure 7. Structure of Memento Pattern

Implementing memento pattern places some special challenges, since implementing Memento-object needs support from programming language used. Memento has two interfaces: a wide one for the Originator and narrow one for other objects. Ideal implementation language will be able to support two levels of static protection, which is

not automatically supported in all object oriented programming languages. (Gamma *et al*. 1995: 287.)

As mentioned in the earlier, frameworks often use the factory method to be able to instantiate classes that can have application-specific implementation. However this functionality is inadequate, when there is sometimes need to create families of related or dependent objects without specifying their concrete classes. This design problem can be solved by using the **Abstract Factory Pattern**. (Gamma *et al*. 1995: 87.)

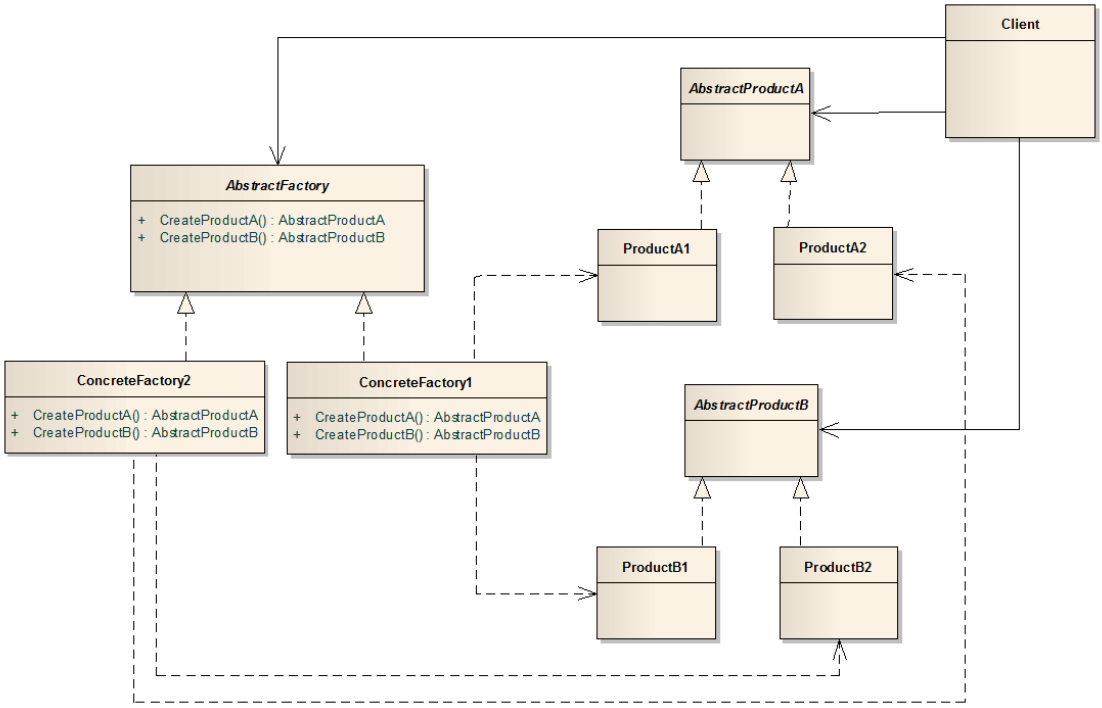

Figure 8. Structure of Abstract Factory Pattern

This pattern provides a way to encapsulate a group of individual factories that have a common theme. The client using factories does not know which concrete objects it gets from each of these factories, since it uses only the generic interfaces of their products. This pattern separates the details of implementation of a set of objects from their general usage. Figure 8 presents a base structure of the abstract factory pattern, with two

factories containing two factory methods. Number of factory classes and factory methods used in the design may vary depending on the application details.

Common way to create abstract factory pattern is to use factory method to create factory methods for each product and to create factory classes as singleton, since multiple factories for same product types is not usually needed (Gamma *et al*. 1995: 90).

Inversion of Control (IoC) is a object oriented programming practice, where the object coupling is bound at the runtime and is typically not knowable at compile time. In traditional programming, the flow of the application logic is determined by objects that are statically assigned to each other, whereas with the IoC, object interactions are being defined through abstractions. In order to be able to bind these objects to one another, the objects must own compatible abstractions. (Fowler 2005).

Implementation techniques depend on the programming language used. For example in Java IoC can be implemented through six different techniques: a factory pattern, a service locator pattern, a constructor injection, a setter injection, an interface injection or through a contextualized lookup. (Fowler 2005; Hammant 2006).

Inversion of Control is not a new term in computer science. The etymology of the phrase first came to light in the 1988, in the article by Ralph Johnson and Brian Foote:

> One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application. (Johnson & Foote 1988.)

Inversion of Control is one of the key parts, which separates software libraries from software frameworks. A library usually contains a set of functions that can be called from outside of the library. Framework in turn contains some abstract design with more built-in behavior. In order to use this behavior user needs to insert own behavior into

framework by subclassing or plugging in own classes. The frameworks code then calls user code at these points. (Fowler 2005; Gamma *et al*. 1995: 26-27.)

The binding process in the IoC is in most cases achieved through Dependency Injection pattern. The basic idea of the Dependency Injection is to have a separate object, an assembler that populates a software component the dependencies it needs to be able to do its work. The injector decides what concrete classes satisfy the requirements of the dependent object, and provides them to the dependent. Figure 9 describes an example where a MovieFinder-object is provided to MovieListener-object by an Assembler to fulfill its dependency to IMovieFinder-interface. (Fowler 2005.)

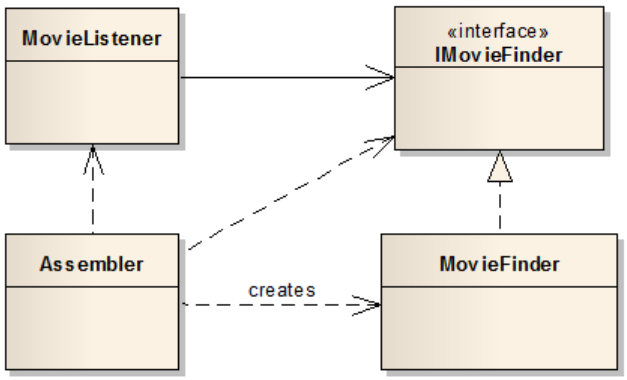

Figure 9. The Dependencies for a Dependency Injector

3.6. Graphical User Interface Programming

Most PC applications today allow real time user interaction achieved by graphical user interfaces, this way improving the usability of an application. User interfaces provide convenient access to software systems data or services, and therefore allow users to produce results quickly by learning the usage of the application.

The challenge in specifying architecture for such applications is managing to keep the functional core independent from the user interface. Functional core of the system is usually based on the functional specification and remains stable, while user interface in

turn are often subject of change and adaptation. Many systems might have to support multiple customer specific “look and feel” scenarios or user interface have to be adjusted to fit into customers business processes. This requires usage of user interface architectures that supports the adaptation of user interface parts without causing major effects on application functionality or underlying data model. (Buschmann, *et al.* 2001: 123.)

This chapter represents more closely three patterns most commonly used at the case organization projects: Model-View-Controller, Model-View-Presenter and Model-View-ViewModel patterns.

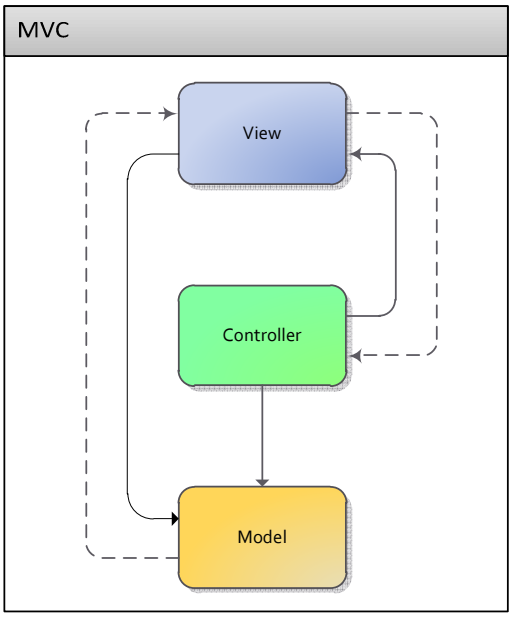

Figure 10. Structure of Model-View-Controller Pattern

In the **Model-View-Controller** (MVC) architectural pattern interactive application is divided into three components. Data and core functionality is included in the Model. Controller handles the user input and the view displays the information to the user. View and controller together create the user interface. The consistency between the user interface and the model is ensured by a change-propagation mechanism. (Buschmann, *et al.* 2001: 125.)

Model component contains the functional core of the application. It encapsulates the data and export procedures that perform application-specific processing. The procedures are called by the controllers on behalf of the user. The model also exports functions for view components to be able to acquire the data to be displayed. The change-propagation mechanism maintains the relations of dependent components within the model. All application views and some of the controllers, if needed, register their need to be informed about changes. Changes to the model state trigger the change-propagation mechanism, which is basically just a link between the model and the views and controllers. (Buschmann, *et al.* 2001: 126.)

View components present information to the user. Different views can present the information of the model in different ways. Each view defines an update mechanism, which can be called by the change-propagation mechanism. When this mechanism is called, view retrieves current data from the model and displays it on the screen. During the initialization views are associated with the model and each view creates a suitable controller for itself. Views offer functionality to controllers that will allow them to manipulate the screen, without having to affect the model. This can be useful for user triggered operations, such as scrolling. (Buschmann, *et al*. 2001: 127.)

Controller components accept user inputs as events. How these events are transferred to the controller depends on the used user interface platform. Events are then translated into requests for the model or the joined view. The behavior of the controller can be dependent on the state of the model. It can be necessary, when change of the model enables or disables for example a menu entry in the view. This interaction can be achieved by registering the controller to the change-propagation mechanism, which triggers the update procedure, when needed. (Buschmann, *et al*. 2001: 127.)

Model-View-Presenter (MVP) pattern is a variant of Model-View-Controller (MVC) pattern also used for building graphical user interfaces. In the MVP pattern the controller is replaced by a presenter. MVP is a design pattern developed to improve

automated unit testing and to help separation of the application logic from the user interface layer (Boodhoo 2006).

In MVP pattern, the model is an interface defining the data to be displayed. The view manages the controls in the user interface; it displays the data and forwards user events to the presenter. The presenter contains logic for responding to the user events and it basically acts as a middleman upon the model and the view. The presenter retrieves data from the model, manipulates it and formats it to be displayed in the view. (Microsoft Corporation 2010.)

There are two different approaches to implementing the MVP pattern, Passive View and Supervising Controller illustrated in figure 11. In Passive View, the view is updated exclusively by the presenter to reflect changes to the model. The view itself is not aware of changes in the model. In Supervising Controller, the view can interact directly with the model through a simple data-binding that can be defined without presenter intervention. The presenter updates the model and it manipulates the state of the view only in cases where more complex UI logic is required. (Microsoft Corporation 2010.)

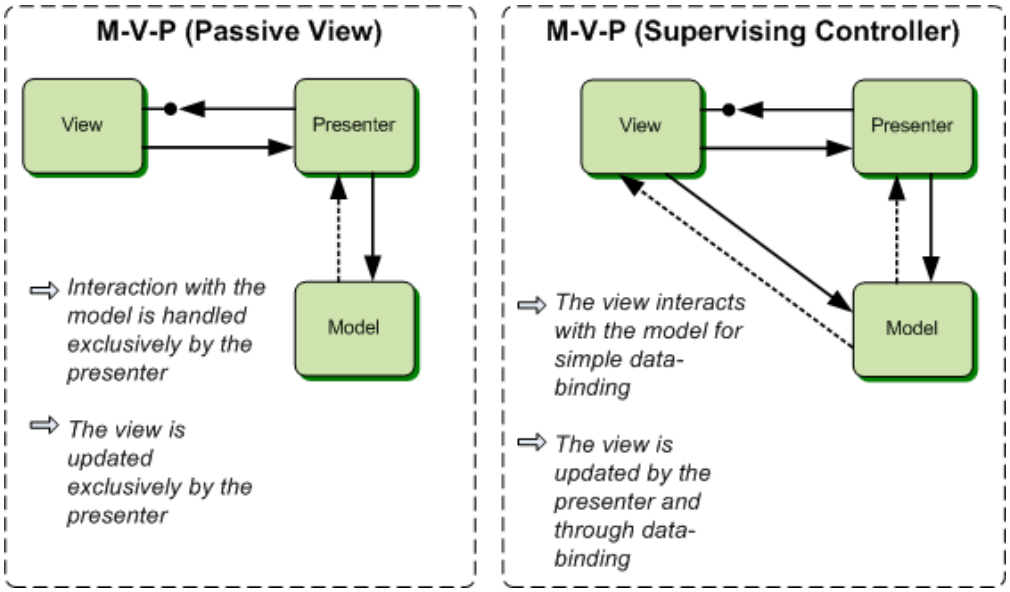

Figure 11. Passive View and Supervising Controller (Microsoft Corporation 2010.)

The **Model-View-ViewModel** (MVVM) is an architectural GUI design pattern originated by Microsoft. Pattern is targeted at modern GUI development platforms which support event-driven programming. MVVM was introduced as a standardized way to leverage core features of Windows Presentation Foundation to simplify the creation of user interfaces. (Smith 2009.)

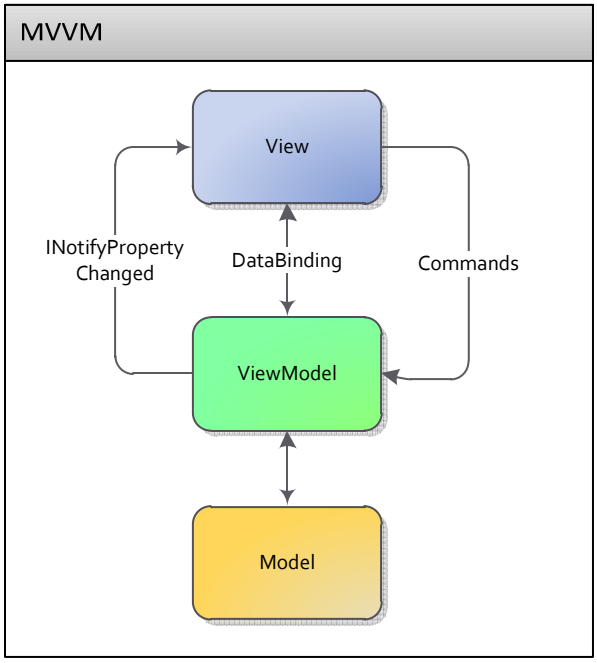

Figure 12. Structure of Model-View-ViewModel Pattern

Although MVVM is partly based on MVC pattern, a viewmodel does not need a reference to the view. Properties of the view are binded to the viewmodel, which in turn exposes data from the model objects and other state specific information to the view. The bindings between view and viewmodel are created by setting the viewmodel object as the datacontext of the view. View data is automatically updated via these data bindings if values in the viewmodel change. When user clicks a button in the view, a viewmodel command executes to perform the requested action. The viewmodel always performs all the modifications made to the model data. The view and viewmodel classes are completely unaware of each other, which makes this a highly loosely coupled design model. (Smith 2009.)

4. DEVELOPING THE SOLUTION

This chapter introduces step by step building of the solution and explains the design methods used. Also convenience of chosen techniques will be discussed.

4.1. Identifying the Solution Requirements

To be able to produce a useful construction for solving the design problem it is important to understand all the needs set by different design approaches. In this chapter the research problem will be presented step by step in smaller entities. Requirement identification is done in three steps. At first application tiers are studied in general level to recognize the actual data needed to be stored. Next different user interface types are compared to find most crucial needs for each interface type to be able to build commonly suitable solution. Finally additional requirements set by usage of different application architectures are identified.

Most of the applications created at target organization are using **Multi-Tier Architecture** model. This kind of software architecture is used to create more flexibility and reusability to developed application. By dividing an application into tiers, developers only have to modify or add a specific tier to overcome some challenge, rather than have to rewrite the entire application. To simplify the examination of the research problem it was chosen to study it from view of one of the most commonly used multi-tier models; the three tier model.

The **Three-Tier Architecture** model, which segments an application's components into three tiers of services, is commonly used as a fundamental framework for the logical design model for computer software systems. These tiers do not necessarily correspond to physical locations on various computers or devices on a network, but rather to logical layers of the application. How the parts of the system are distributed in a physical topology can change, depending on the system requirements. Although the concepts of layer and tier are often used interchangeably, a common point of view is that a layer is a

logical structure for an element of software solution, while a tier is a physical structure of the system. (Microsoft Corporation 2012.)

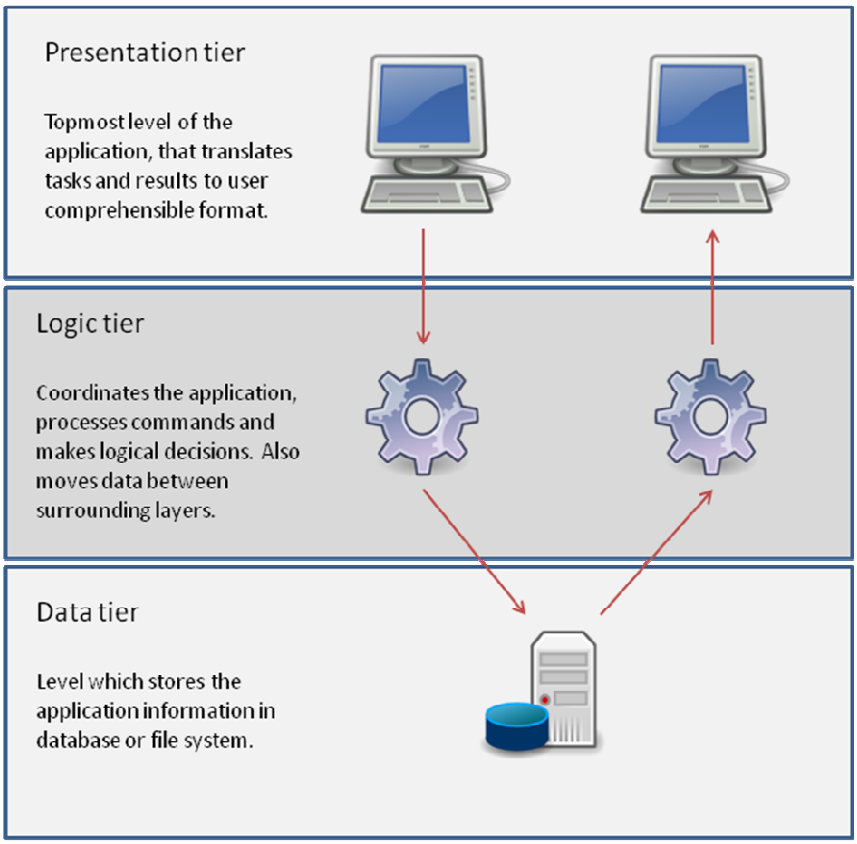

Figure 13. Example Implementation of Three Tier Architecture

As the figure 13 presents applications usually have three tiers; presentation tier, the user interface itself, data tier containing the data presented in the user interface and logic tier binding these two together.

In industrial device control the data tier can be the machine or device itself. It is also common, that there is some data storage in the data tier, where the configuration values set to the device will be stored. This data storage is normally a database or set of files stored to computers file system.

Our research mainly focuses on the logical tier, since it usually holds the state set through the user interface to interfere with the data tier. In this context the state represents communication settings set to open the communication link to the device. In the two projects studied for this research these communication links used are serial-, CAN-, and Ethernet-connections. State can also represent the parameters used to make database connection or reference to the configuration data set to user interface by user during the configuration process. This observation presents a requirement for the construction:

Requirement 1: Solution needs to be able to retrieve and restore the state to the logic tier of the application.

By studying different desktop application project developed in the target organization it was noticed that there is at least three different User Interface Types, that needs to be supported by the solution: single document interface (SDI), multiple document interface (MDI) and tabbed document interface (TDI). Some projects also contained combinations of these types.

In a single document interfaces each application window is usually represented as an individual entry in the operating system's task bar or manager. Each window also contains its own menu or toolbar, whereas in the multiple document interface model multiple nested child windows are embedded in a single parent window containing the menu or toolbar. In the tabbed document interface multiple documents are contained in a single window, using tabs as a navigation method for switching between the documents. In TDI usually only one documents is shown at once. TDI is a commonly used in web browsers, although nowadays many browsers combine TDI with SDI.

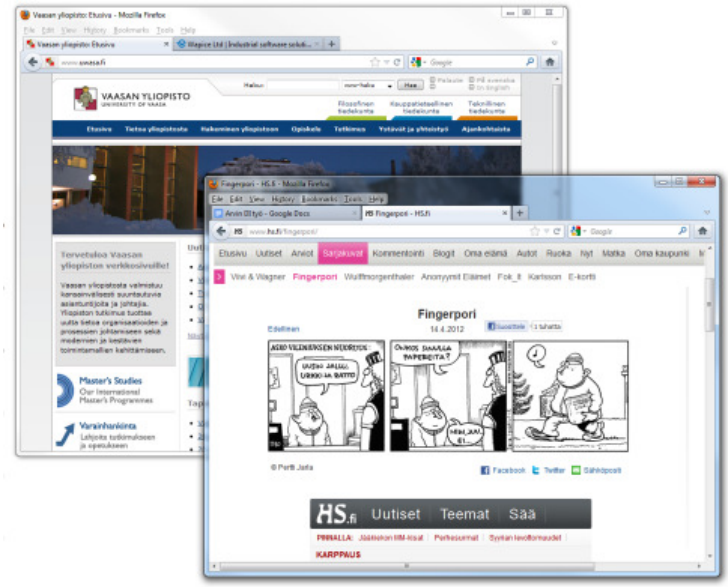

Figure 14. Firefox Web Browser Utilizing SDI & TDI

Since the constructed solution needs to be adaptable to most of the GUI projects using these different types of document interfaces it presents a new requirement for the solution:

Requirement 2: Solution needs to be able to store and restore state of interface controls, regardless of document interface type used.

In some cases applications are extended to use multiple document interface types at once. Figure 14 demonstrates a situation, where TDI in embedded to SDI. Each of the individual windows contains their own settings, like position, size and information about opened tabs, whereas each of the tabs contains its own settings. This basically means that there can be undefined amount workspaces nested to each other. This places a new requirement for the solution:

Requirement 3: Solution needs to support tree structure of workspaces.

It was also noticed that dependency injection pattern was chosen to be used in most of the new GUI application projects developed in the target organization. However the

dependency injection framework used, varied depending on the application architecture. Some of the applications even used some kind of composition layer to achieve plug-in-like architecture. This means that usage of software components providing dependency injection or plugability to the software architecture should be allowed with the solution. This presented a new challenging requirement for the solution:

Requirement 4: Solution should not prevent developer from using inversion of control.

4.2. Designing Basic Structure for the Solution

During this chapter creating a design based on the requirements identified in previous chapter will be discussed. By applying software design patterns explained in previous chapters an UML-model containing mandatory elements for the solution will be designed.

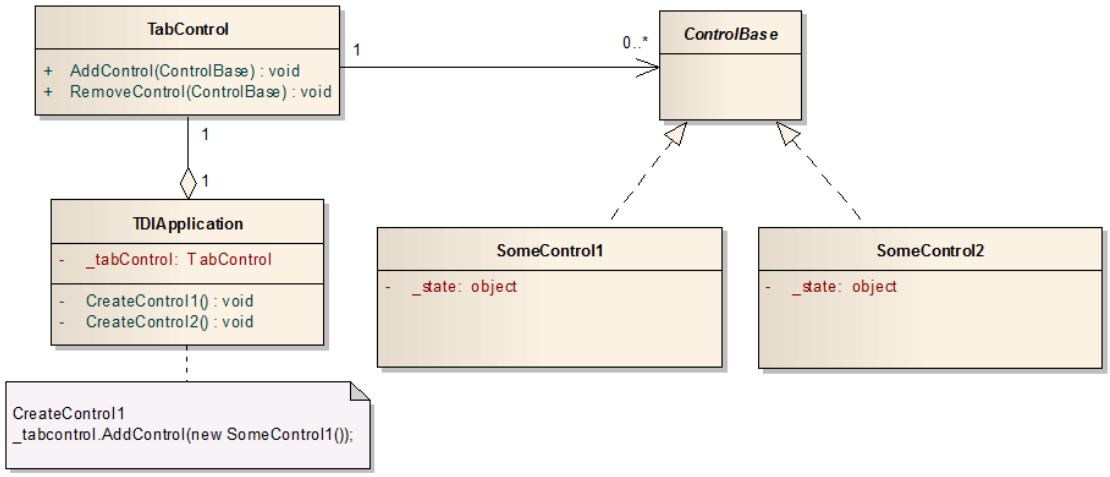

Figure 15. TDI Application Example

Building the design was started by examining a simple tabbed document interface application. Figure 15 presents an application called TDIApplication containing a tab control component TabControl with two user controls SomeControl1 and SomeControl2 inside it. Both user controls are inherited from abstract base class ControlBase. It is

quite common, that the tab control and base class for the control are provided by some third party software framework, for e.g. .NET Framework.

In this example a new user control is brought into view by instantiating a user control class and passing object to tab control, which then draws a new tab to user interface. The user workspace in this application is basically the tab control containing created user controls inside its tabs. State of the user workspace would be stored inside created user controls as a private attribute, demonstrated by _state-objects in the figure.

Since the third party GUI components cannot be usually modified, first step of constructing the design was defining basic structure for the storable workspace beside the GUI framework as presented in the figure 16. A Workspace-class was created to hold together the workspace-concept. An interface IWorkspaceObject was created to determine which user classes belong to the workspace. By inheriting IWorkspaceObject-interface programmer can then define which classes belong to the workspace.

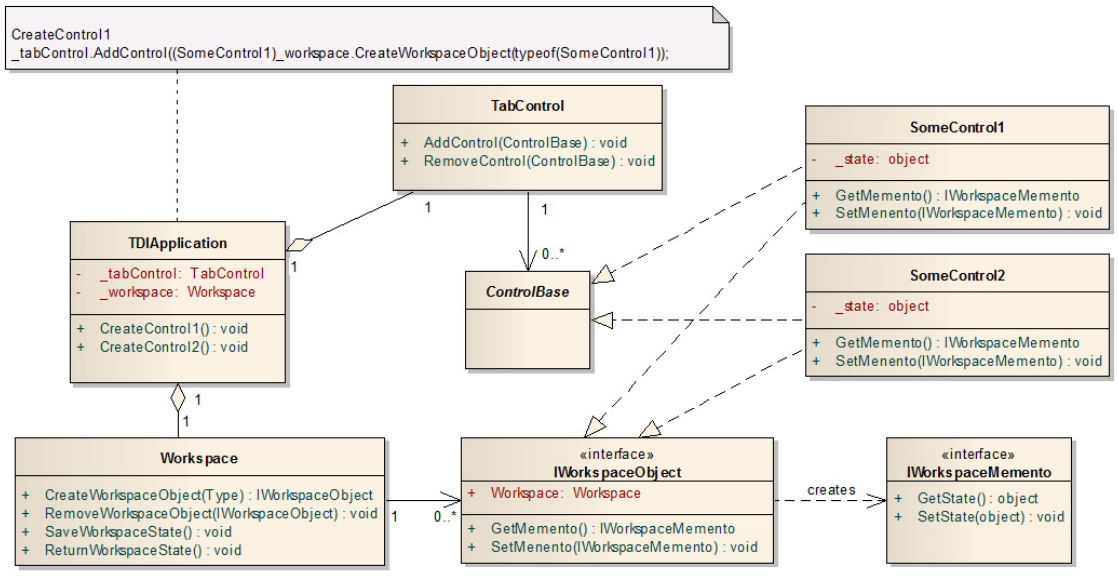

Figure 16. TDI Application Example with a Storable Workspace

Memento pattern were applied in IWorkspaceObject-interface to define a way to get an object containing controls internal state from the user controls. Memento was only defined as an interface passing the responsibility of implementing suitable memento classes to the programmer.

Factory method was applied in Workspace-class method CreateWorkspaceObject-method, which was defined for creating user controls by passing control type into the method. Since the returned object is IWorkspaceObject-type, the user is responsible of casting the object to suitable type for the tab control. However this kind of design also provides a possibility to store, not only user controls, but also any kind of other objects as workspace objects. Reference to workspace objects created through this method will be gathered inside the workspace providing a way of getting and setting the internal states of all workspace objects inside workspace.

Design provides a way of creating a workspace with multiple workspace objects, e.g. user controls inside it. Workspace internal state from all workspace objects can be retrieved and restored by the workspace by using the memento objects. So at this point the design satisfies requirements 1 and 2, but not 3 and 4.

To satisfy requirement 3, support for creating workspaces inside workspace was added. Also a WorkspaceManager-class was added to design at this point to act as main interface for creating, saving and opening workspaces. The solution API was deliberately designed so that it requires the programmer to use WorkspaceManager to create a root Workspace-object which could be later used for creating nested Workspaces to root object and so on.

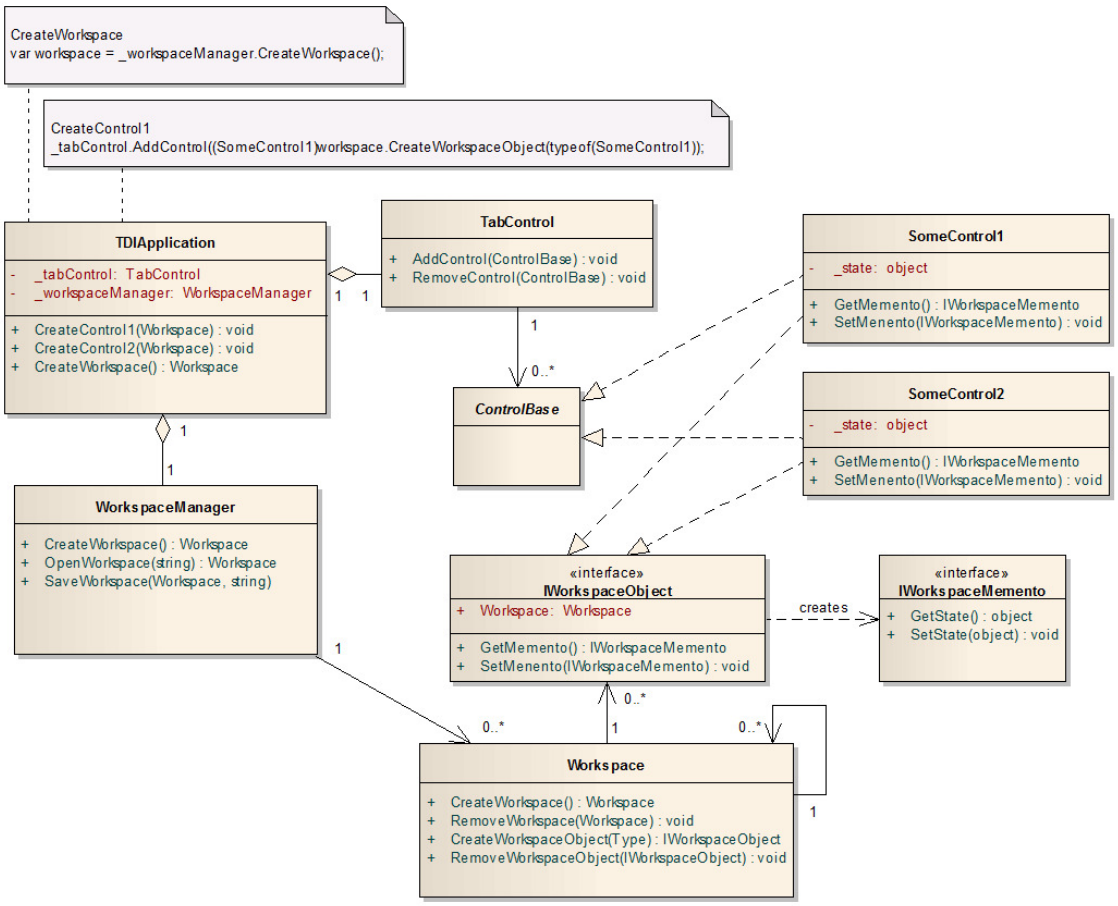

Figure 17. TDI Application Example with Storable Workspace Tree

Solution presented in figure 17 defines a basic structure for the solution and provides a design that can be used to answer the first research question: *"How to design a solution for saving user workspace of a desktop application?*" Methods for saving and opening a workspace from file were added to WorkspaceManager-class, although at this point it is still not defined how to retrieve the workspace object data encapsulated by the memento objects and store it to the computer file system. Another issue is that the solution structure is also currently tightly binded into the example application. Overcoming these design issues will be discussed in the upcoming chapters.

4.3. Building the Solution as an External Software Component

To answer the second research question: *"How to build the solution as a reusable software component?"* requires inevitable some modifications to the solution. As presented in figure 18, at this point the application parts were divided into three different areas according to its use; GUIFramework containing parts that would be adapted from some third party GUI framework, UserApplication containing parts implemented for the user application and WorkspaceFramework containing parts that belong to the workspace framework implementation.

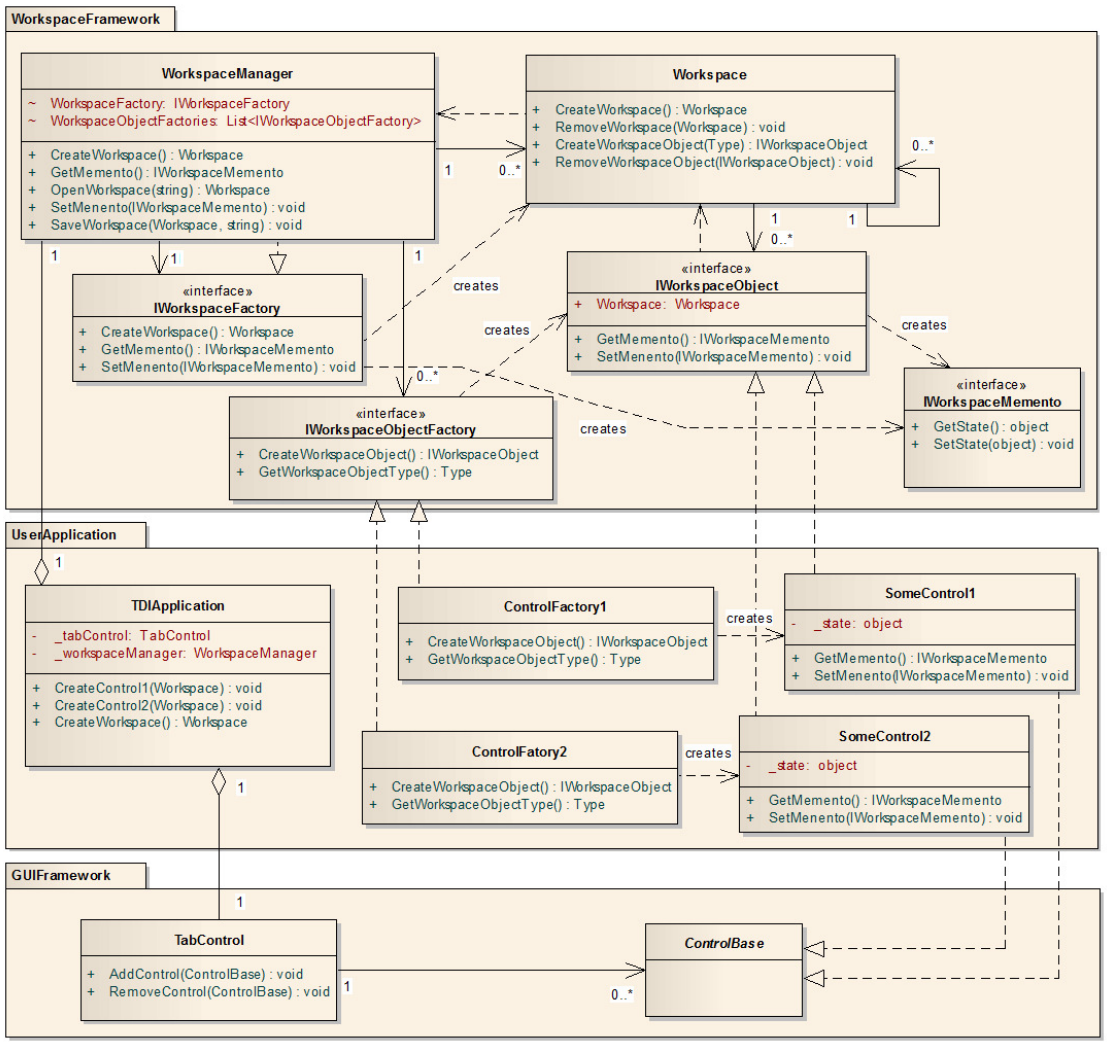

Figure 18. TDI Application Utilizing Workspace Framework

After reordering the classes Workspace become unaware of the user classes implementing IWorkspaceObject-interface and therefore cannot anymore directly instantiate those classes. This problem was solved by utilizing the adapter factory pattern by adding to the design an IWorkspaceObjectFactory-interface for creating objects inheriting IWorkspaceObject.

By forcing framework user to create factory for each framework object, also requirement 4 was solved. With this design implementing dependency injection is possible through the workspace object factories. Also an IWorkspaceFactory was added for user to be able to create own workspaces in addition to the default workspace provided by the framework. By creating own class inheriting IWorkspaceFactory, user is able to store also the workspaces internal state in addition to workspace objects states.

In order to use framework it is now required to pass list of IWorkspaceObjectFactory implementations and optionally the IWorkspaceFactory implementation when WorkspaceManager is created as presented in the C# code example below.

```
IWorkspaceFactory WSFactory = new WorkspaceFactory();
IEnumerable<IWorkspaceObjectFactory> WSOFactories =
    new List<IWorkspaceObjectFactory>()
    {
        new UserControl1Factory(),
        new UserControl2Factory()
    };
WorkspaceManager WSManager =
    new WorkspaceManager(WSFactory, WSOFactories);
```

4.4. Storing the User Data

After designing the base structure of the solution it was required to find a way for storing the user data wrapped in the memento objects inherited from the IWorkspaceMemento-interface. Although all the mementos can be gathered by the Workspace-object, it is still unable to access the actual data wrapped inside mementos, because of their encapsulation.

A mechanism chosen to resolve this problem was serialization. By implementing memento-objects as serializable it is possible to store the object to computers file

system and restore it later to same state. It is bit controversial to break the mementos encapsulation this way, but considering the purpose of this framework it can be considered legitimate.

The binary serialization was chosen to be used for the serialization mechanism, because it provides type fidelity. This means, that it will save the entire object state, not just some of the object's data. When binary serialization is used to persist an object, deserializing it will give an exact copy of the object. This is vital for the framework, because it is implemented apart from the user application and is not aware of object types used in the user application. Framework also needs to be able to restore stored user data, even if the user application has changed. Changing application radically will most likely prevent framework from restoring some of the data, but it should still be able to restore everything, that is still restorable.

Having compiler to force classes implementing IWorkspaceMemento to serializable in C# is not possible, which sets a new design rule for framework user: All classes implementing IWorkspaceMemento need to be serializable. By passing responsibility of following this implementation rule to user leaves always some possibility for error to the implementation, which is not usually desired. To ensure that IWorkspaceMementos are implemented correctly, e.g. unit tests using reflection can be generated for ensuring that all application classes inheriting IWorkspaceMemento are serializable.

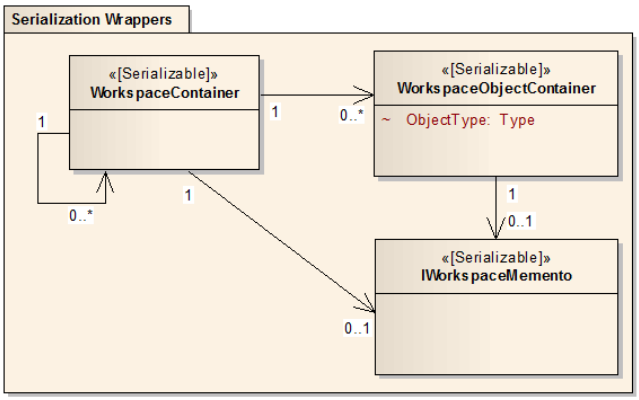

Figure 19. Serialization Wrappers of Workspace Framework

Maintaining the workspace object-structure during the serialization phase also set an additional challenge for storing the data. Since the mementos are inside nonserializable objects inherited from Workspace or IWorkspaceObject, it is impossible to store the whole workspace structure to the file system. Figure 19 presents a set of serializable wrapper-objects, containing same objects structure as the actual workspace. These wrappers were created as frameworks internal implementation to maintain the objects-structure, when data is stored.

By interpreting the wrapper-structure and using the object factories provided by the user application, framework is able to recreate the whole object-structure, when data is restored. In addition to the structure, also type of the referred IWorkspaceObject is required to be stored in WorkspaceObjectContainer-objects to be able to restore correct type of objects, when recreating the workspace. The object type was chosen to be stored as string, by its assembly qualified name, so that removing some of the stored object types from implementation does not prevent restoring still existing objects. Trying to deserialize a Type-object that cannot be resolved causes the serializer to throw an exception.

Following code describes the implementation of WorkspaceObjectWrapper-class in C#.

```
[Serializable]
internal class WorkspaceObjectContainer
{
    private string _type;

    public WorkspaceObjectContainer(Type type,
        IList<IWorkspaceMemento> mementos)
    {
        _type = type.AssemblyQualifiedName;
        Mementos = mementos;
    }

    public IList<IWorkspaceMemento> Mementos
    {
        get;
        private set;
    }

    public Type ObjectType
    {
        get { return Type.GetType(_type); }
    }

}
```

4.5. Optimizing Construction for C#

The workspace framework design as explained so far can be adapted with small adjustments to multiple different programming languages. Since the main focus was to implement the solution with C# programming language, it was decided to improve the usability of the design by improving it with help of some more powerful features existing in C#.

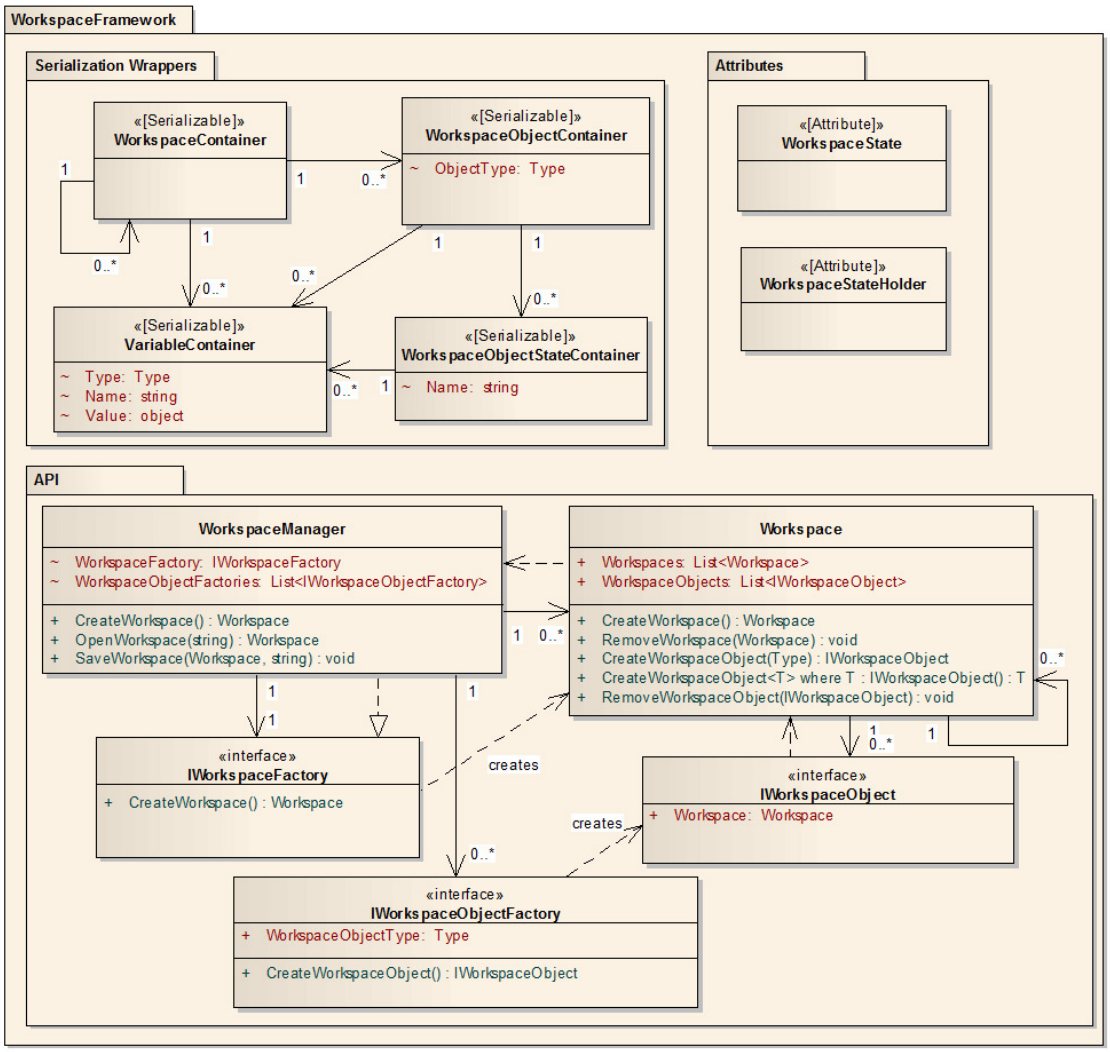

Figure 20. Workspace Framework Optimized for C#

C# provides a mechanism for defining declarative tags, called attributes, which can be placed on certain entities in source code to specify additional information. The information that attributes contain can be retrieved at run time through reflection. It is also possible to define custom attributes. These features were exploited in the solution to be able to remove IWorkspaceMemento-interface from the design completely making the API of the design more user-friendly. The optimized design of the solution is presented in figure 20, where also API and internal implementation of the solution is divided into their own namespaces.

IWorkspaceMemento-interfaces were replaced by introducing two custom attributes: WorkspaceState and WorkspaceStateHolder. This way user can implement own memento objects of any kind and mark references to objects with WorkspaceState-attribute to indicate the framework which objects needs to be stored. Only requirement is that the memento objects need to be serializable. In case there are some nonserializable data references inside state objects, a NonSerializable-attribute provided by the .NET Framework can be used to restrict those from the serialization.

At this phase an additional class WorkspaceObjectStateContainer was also added to the wrapper-structure to provide one additional level to the workspace structure. This was done to support design patterns, where workspace object does not have direct reference to the workspace object state, because there is another object in between these objects. This is a common phenomenon, when implementing GUI using e.g. MVC, MVP or MVVM pattern. The WorkspaceStateHolder-attribute was added to mark reference to this object from the workspace object.

Retrieving and restoring the data fields encapsulated by workspace objects was implemented inside the framework by using the reflection to find the custom attributes. Each field was then wrapped into VariableContainer for serialization as described in the following C# implementation.

```
/// <summary>
/// Get all fields from object containing attribute TAttribute
/// </summary>
/// <typeparam name="TAttribute">Type of the attribute</typeparam>
/// <param name="obj">Object to search from</param>
/// <returns>List of fields and values</returns>
private static IList<VariableContainer>
GetFieldValuesWithAttribute<TAttribute>(object obj)
{
    List<VariableContainer> values = new List<VariableContainer>();

    var allInfos = obj.GetType().GetFields(BindingFlags.Instance
        | BindingFlags.NonPublic | BindingFlags.Public);

    foreach (var info in allInfos)
    {
        var attributes = info.GetCustomAttributes(false);

        if (attributes.SingleOrDefault(x => x.GetType()
            == typeof(TAttribute)) == null)
            continue;

        values.Add(new VariableContainer(info.Name,
            info.GetValue(obj), info.FieldType));
    }

    return values;

}
```

It was also decided to simplify the workspace object creation by adding a new generic method to Workspace-class, making it possible for user to create an instance of IWorkpaceObject without need to manually cast the objects to correct type. Following code example demonstrates the difference between old and new way of creating IWorkspaceObject-objects.

```
// Using new method
UserControl1 UserControl1 = _workspace.CreateWorkspaceObject<UserControl1>();

// Using old method
UserControl1 UserControl1 =
    (UserControl1)_workspace.CreateWorkspaceObject(typeof(UserControl1));
```

5. EVALUATING PRACTICAL RELEVANCE OF THE DESIGN

To be able to evaluate the actual practical relevance of the design it was necessary to test it in practice. First the framework was tested by the author by implementing sample application using different GUI programming approaches. Second test were performed by software designers working in the case organization. Designers deployed the implemented software framework into their own applications and filled a user questionnaire to grade its functionality and usefulness. This chapter presents the results of these evaluation methods.

5.1. Testing Design by applying it to Different GUI Design Models

It is not recommended, but still quite common to implement a GUI application without using any specific GUI design pattern. This is why an application implemented **without design pattern** was used as a reference in first test case.

In this test case a simple application containing one user interface was created. Also the optional feature of creating own custom workspace was tested by implementing IWorkspaceFactory. User interface and the custom workspace both contained some internal state data, which was marked with WorkspaceState-attribute.

Since the original starting point for the design was a TDI application, not using any particular pattern, it was not surprise that the solution worked fine for application without using any design pattern. Both, workspace and user interface data could be stored and restored without any problems. Figure 21 represent a situation where workspace API is utilized by simple test application.

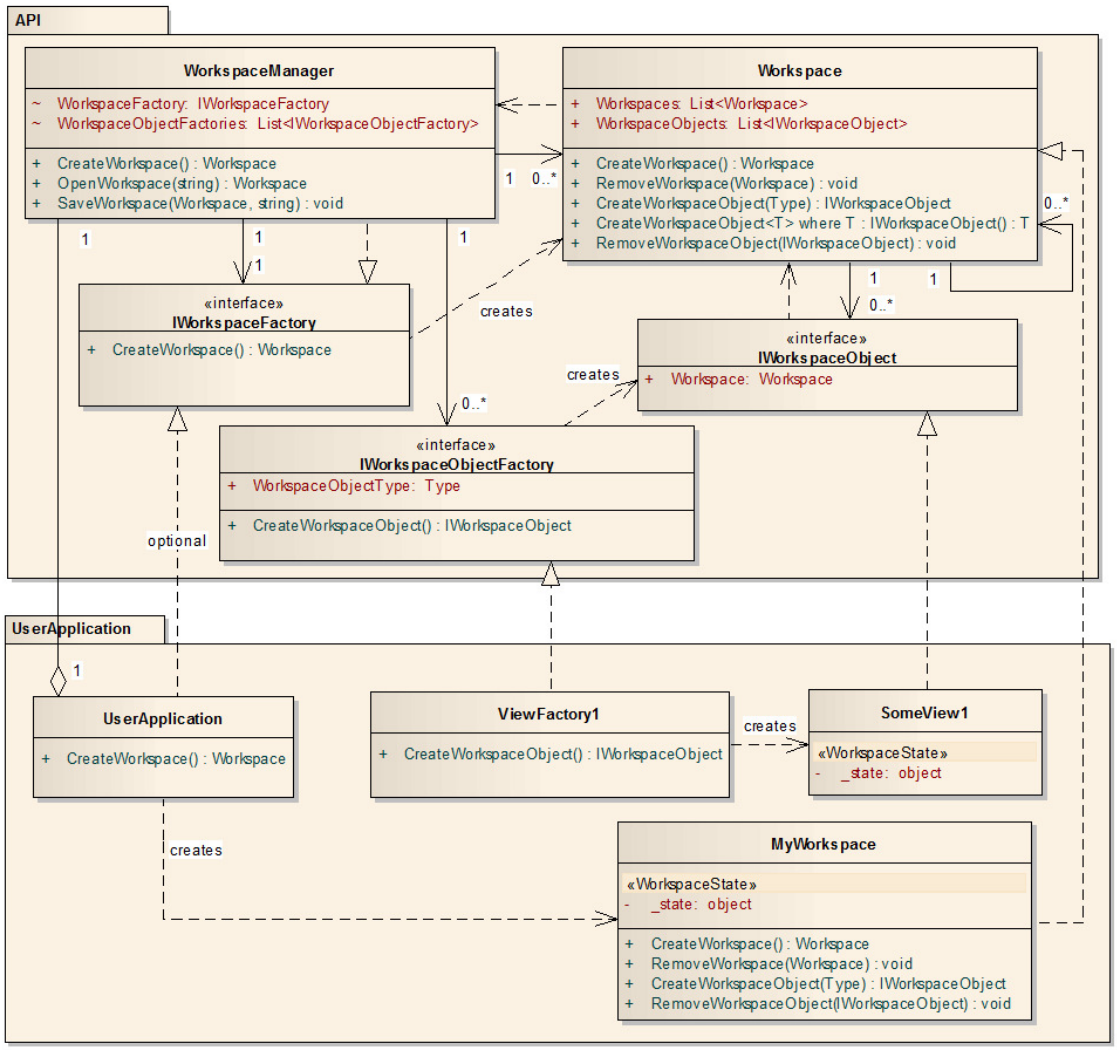

Figure 21. Example of Workspace Framework Usage without Design Pattern

Second test were performed to application using **MVVM design pattern**. In this test case similar application to first test case was created, except that the ViewModel for the view was added to the application. Although WorkspaceStateHolder-attribute was already presented earlier in this thesis, it was actually at this test phase where it was noticed that additional attribute is needed to be able to support MVVM pattern. As presented in figure 22, WorkspaceStateHolder-attribute was used to address the framework where ViewModel containing state data can be found.

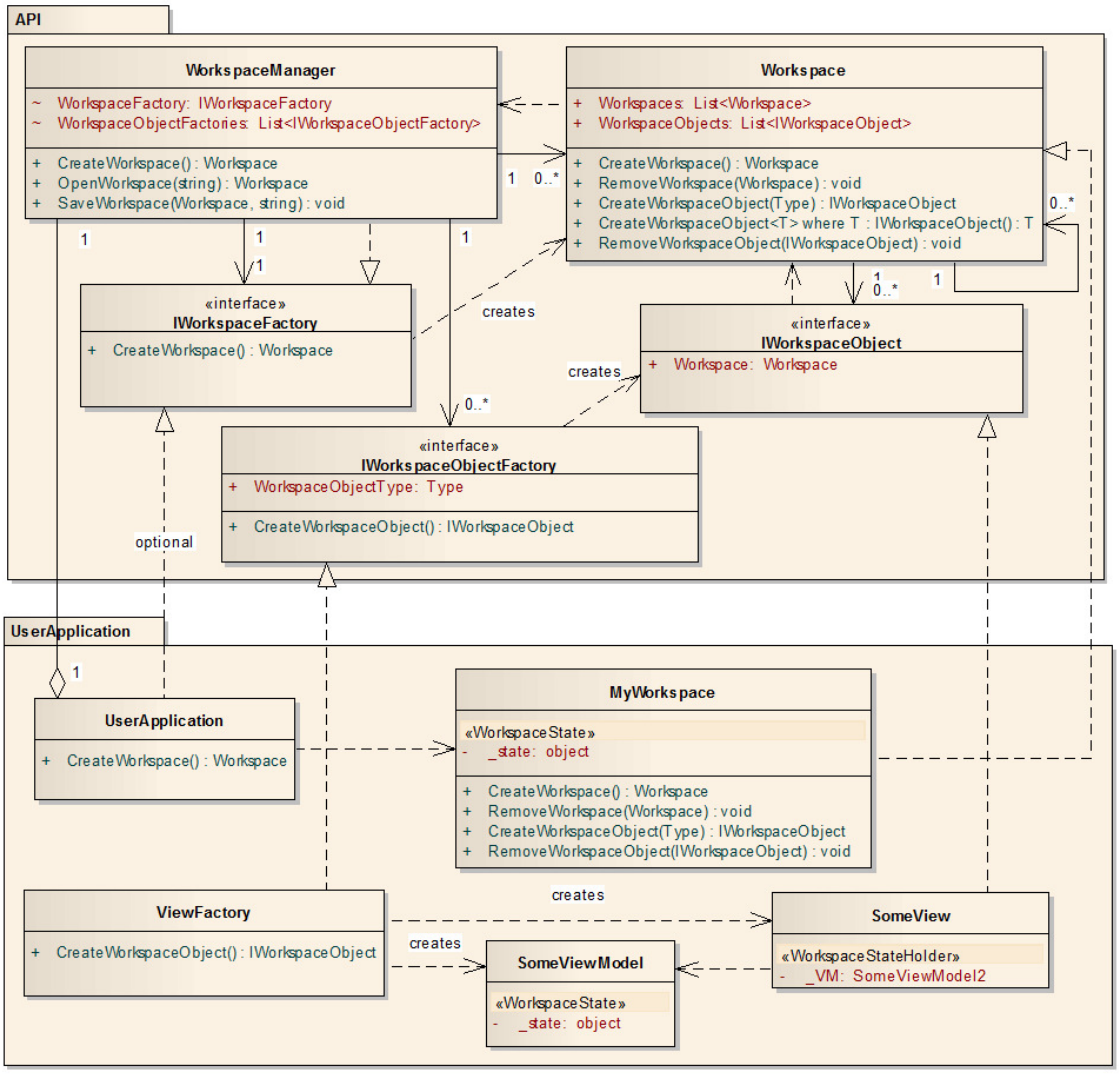

Figure 22. Example of Workspace Framework Usage with MVVM Design Pattern

Following code example presents how ViewFactory implemented to create view and viewmodel when CreateWorkspaceObject-method is called. It also associates the viewmodel to the view using constructor injection.

```
class ViewFactory : IWorkspaceObjectFactory
{
    IWorkspaceObject IWorkspaceObjectFactory.CreateWorkspaceObject()
    {
        return new SomeView(new SomeViewModel());
    }

    IWorkspaceObjectFactory.Type TypeOfWorkspaceObject
    {
        get { return typeof(SomeView); }
    }
}
```

Following code example presents how the view is implemented to expose the viewmodel to the framework by using WorkspaceStateHolder-attribute.

```
public partial class SomeView : UserControl, IWorkspaceObject
{
    [WorkspaceStateHolder]
    private SomeViewModel _VM;

    internal SomeView(ExampleViewModel vm)
    {
        this.DataContext = _VM = vm;
        _VM.Workspace = Workspace;
        InitializeComponent();
    }

    #region IWorkspaceObject Members

    public Workspace Workspace { set; private get; }

    #endregion

}
```

By utilizing MVVM pattern as presented above, the framework was able to store the created workspace and restore it back to its previous state.

Since the applications utilizing **MVC and MVP design patterns** contain three components containing relations to each other as the MVVM pattern, it is possible to apply the same design principles as presented in MVVM test case to utilize the framework functionality. By correct exploitation of WorkspaceStateHolder- and WorkspaceState-attributes the workspaces data can be stored and restored also in application using these GUI design patterns.

5.2. Analyzing Results of the User Questionnaire

Although it was originally planned to have larger test group for the framework evaluation, only three software designers took part in the testing, because it was really difficult for the designers to find time from their other work within the tight timeframe of this thesis. Although the test group was relatively small, the questionnaire still gives some directional information about usefulness of the research outcome.

Work and educational profile of the interviewees were quite similar; each had Bachelor of Engineering degree and worked as a software designer within the target organization. Interviewees work experience varied from three to ten years, but each of the designers had spent most of their work career developing mainly GUI applications. However one of the designers specified, that his experience were only focused to graphical web-applications.

As preliminary questions the interviewees were asked about their previous needs for similar workspace storage solution. Two of the designers answered that they have had need for the solution and even implemented a similar solution before. However another of the designers defined, that his solution was not implemented in a reusable way. Only the web-application designer had not had any need for this kind of solution before.

When designers were asked about their most commonly used design pattern, MVC were found as the most commonly used pattern within this group of designers. The most experienced designer in the group gave quite interesting answer to this question: *"MVC, Magic pushbutton, Big ball of mud."* (Designer 2.) With this answer he was most likely referring to the fact that there are still quite many application projects implemented without any proper architectural design and by utilizing anti-patterns.

After testing the solution, designers were questioned about the framework usage. Directly it was noticed, that none of the designers used any actual GUI design pattern in their test solution, which unfortunately means that this questionnaire does not provide any actual data about framework suitability to different GUI design patterns.

As implementation technique designers used two different .NET Framework APIs, WinForms and Windows Presentation Foundation (WPF). Two of the designers implemented MDI-application and one SDI-application.

Designers were able to deploy the framework without any bigger problems. However each of the designers agreed that most challenging part was setting it up for the first time. After getting the hang of the framework, using it got easier. One designer also

pointed out, that better examples would be needed for larger distribution of solution, when asked about easiness of framework usage.

"At first it was a bit challenging to grasp the concept, but after I started implementing functionality in the framework it became clearer how it worked. The example application did also make it easier, even if I didn't use the same GUI framework" (Designer 1.)

"Yes, when I got hang of it. Simpler code examples & tutorials would be must if this is taken into wider distribution." (Designer 2.)

The main functionality of the framework, storing and restoring application workspaces was proven to be working by each designer. Only one of the designers reported, that he had to change his application architecture to get the framework working.

"Had to move program logic out of UI layer, but that stuff shouldn't even be there if done properly (Magic pushbutton)" (Designer 2.)

However, making this change can be actually interpreted as a change towards better application architecture. Also one other designer mentioned that using framework guided him to tier the user interface layers better.

"In order to use framework it's better to separate GUI components to logical pieces, e.g. views. This helps to handle those components which will be included in the workspace. Using framework will actually help developer to build correct architectural GUI design." (Designer 3.)

As an additional benefits gained by of using the framework designers mentioned the ease of storing the application data and the fact that the framework did not require wrapping all the data together for storing.

"Saves the trouble of doing more manual serialization." (Designer 2.)

"There's no need to implement separate classes or settings to store user data since framework can also handle user data, e.g. text in text box." (Designer 3.)

None of the designers found any disadvantages from using the framework in their application. However one of the designers noted that his test application was not necessarily large enough to even find any real advantages from using the framework.

"Maybe app was bit too small to have any real advantage from this FW. Larger app next time" (Designer 2.)

As a result of this questionnaire, it can be said that the concept was proven working. Since the applications which the designers used to test the framework were relatively small, it can be suspected, that none of the testers really exploited the full potential of the framework to handle large workspace structures. To determine the real effectiveness of the framework it needs to be taken into use in larger application projects developed for real-world purposes. At least one of the designers who participated in testing had intentions to start using the framework in upcoming projects.

"While developing GUI applications there's usually always demand for saving the user workspace. Therefore this framework will provide huge help for upcoming projects while it saves lots of working hours by providing generic working solution for workspaces. In addition it will guide the architectural design of the application's GUI to better direction" (Designer 3.)

Complete list of questions and results of the questionnaire is presented in appendix 1.

6. CONCLUSIONS AND FUTURE

The final chapter of this thesis discusses about the work done and includes conclusions of the research. Also practical and theoretical significance of the work and future research possibilities are discussed.

In the beginning of the research project there was a need in the case organization for solving a desktop application programming problem of storing user workspaces. It was also decided that the problems should be solved in general way in order to utilize the solution also in similar upcoming software projects.

Solving of the research problem was stated by gathering knowledge about what kind of solution the company actually needs. At first a literature review was carried out to get acquainted with basic concepts of object oriented programming and ways of reusable software programming by studying design patterns and their usage. Also with co-operation of company employees a field study was performed to gather insights and implementation details about previous desktop application projects.

During the research phase an initial idea how to develop a solution for overcoming the design problem started to evolve. At the development phase a base design for the solution were build through an application example. By testing the solution model iteratively against different design patterns it eventually formed a software framework implementation that provided the actual construction of this research.

The design of the software framework provides an answer for two first research questions. Solution for saving user workspaces of desktop applications can be created by implementing design containing object structure for workspaces and a base interface for user controls utilizing memento pattern as explained in figure 17. The solution can be built as a reusable software component by utilizing abstract factory pattern for moving the design outside of user application as presented in figure 18.

Finally the construction was tested in the case organization and the effects of using it were evaluated. All of the designers testing the solution were able to utilize it in their own application easily. Designers also agreed that they gained clear benefits by using the solution. Some of the designers even noticed that using the solution guided their GUI architecture to better direction. One of the designers also considered using the framework in his upcoming projects.

The questionnaire held for software designers provides at least a directional answer for the third research question. It can be concluded that the solution helped to improve GUI application development efficiency by shortening the project development time and by guiding the architectural decisions of the application development to better direction.

It is however important to keep in mind that the constructed framework is no silver bullet that would solve all the user workspace related issues in every desktop application project. There can be several difficulties that were not noticed during this research and therefore cannot be solved directly by using the framework. This is why it is important that the target company continues iteration and development of the framework during upcoming software projects were it will be applied.

6.1. Practical Contribution

Construction developed during this research was built as a proprietary software component for an industrial software subcontractor company based on its project needs. A software framework was implemented to save development time in upcoming projects, this way increasing the business profit of the projects. This kind of reusable component can be also used as a selling point in gaining new software projects.

The constructed framework was adapted as a part of target organizations reusable software libraries and has been taken into use in at least one desktop application project. Due to this it can be said, that the construction passed the weak market test of constructive research approach (Kasanen *et al*. 1993: 253).

6.2. Theoretical Connection

From the academic point of view presenting the theoretical connections and the research contribution of the construction is an important part of constructive research study. In this research the construction itself has a notable practical contribution, but it also makes theoretical contribution and provides knowledge in the field of desktop application development.

The theoretical background study carried out in this research was not comprehensive enough to be seen as a new theoretical contribution. It was mainly documented to give reader sufficient base knowledge to understand why and how the actual construction was built. However there is very few publication available discussing about the problematic of handling user workspaces of desktop applications. Therefore the detailed description of building the construction is an important part of theoretical contribution of this research. It provides a reference design and an extendable concept how a software framework can be constructed for similar purposes.

This study also brings some contribution to the theory of constructive research. There are many articles in which the constructive research approach has been applied to different topics, through slightly different approaches. Such as the other articles, also this study brings up some ideas on how to carry through the constructive research process, in the field of software development.

6.3. Future Research

The constructed framework has so far been adapted to only one software project, but the future looks promising. Since the framework will be distributed as a part of organizations common software libraries it will be easily available to be taken into use in other upcoming software projects. When the framework starts to be more commonly used it will most probably set new requirements for the framework and new areas of improvement will be noticed.

During the implementation of the framework some improvement ideas were noted, but not included into design because of the tight timeframe of the research. These improvement ideas followed by a question, presented in the next paragraphs can provide some base for possible future research.

The storing of the framework data is done to file system inside the framework, by passing file paths to framework API. This limits the design only to applications, with access to file system. Could the initializing of the storage streams be moved outside the framework enabling it to use same solution also for example in rich internet applications running in web browser plug-in?

Using serialization to store the workspace data clearly breaks workspace object encapsulation. Could it be possible to obfuscate or encrypt the storage streams?

How well does the framework adapt to changes in the application structure? Should some kind of versioning be added to serialized streams, to ensure change tracking and backwards compatibility?

It is clear that, if this kind of software component becomes widely adapted in many different types of applications, it will take several years of improvement, before it will find its final solid form.

REFERENCES

Boodhoo, Jean-Paul (2006). *MSDN Magazine: Design Patterns: Model View Presenter* [online] [cited 2.4.2012]. Available from Internet: <URL:http://msdn.microsoft.com/en-us/magazine/cc188690.aspx>

Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad & Michael Stal (2001). *Pattern-Oriented Software Architecture : A System of Patterns.* West Sussex: John Wiley & Sons. 467 p. ISBN 0-471-95869-7.

Cline, Marshall (2011). *Serialization and Unserialization* [online] [cited 8.4.2012]. Available from Internet: <URL:http://www.parashift.com/c++-faq-lite/serialization.html>

Ecma International (2006). *C# Language Specification: 4th edition / June 2006* [online] [cited 1.4.2012], 2-20. Available from Internet: <URL: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf>

Fowler, Martin (2004). *Inversion of Control Containers and the Dependency Injection pattern* [online] [cited 8.4.2012]. Available from Internet: <URL:http://martinfowler.com/articles/injection.html>

Fowler, Martin (2005). *Inversion of Control* [online] [cited 8.4.2012]. Available from Internet: <URL:http://martinfowler.com/bliki/InversionOfControl.html>

Gamma, Erich, Richard Helm, Ralph Johnson & John Vlissides (1995). *Design Patterns - Elements of Reusable Object-Oriented Software.* 21. ed. Boston: Addison-Wesley. 395 p. ISBN 0-201-63361-2.

Hammant, Paul (2006). *IoC Types* [online] [cited 8.4.2012]. Available from Internet: <URL:http://docs.codehaus.org/display/PICO/IoC+Types>

Hurlbutt, Matt (1998). A *Tutorial on Behavioral Reflection and its Implementation* [online] [cited 7.4.2012]. Available from Internet: <URL:http://www2.parc.com/csl/groups/sda/projects/reflection96/docs/malenfant/ref96/ref96.html>

Johnson, Ralph & Brian Foote (1988). *Designing Reusable Classes. Journal of Object-Oriented Programming* 1:2, 22-35. Available from Internet: <URL:http://www.laputan.org/drc/drc.html>

Kasanen, Eero, Kari Lukka & Arto Siitonen (1991). Konstruktiivinen tutkimusote liiketaloustieteessä. *The Finnish Journal of Business Economics* 3, 301-329.

Kasanen, Eero, Kari Lukka & Arto Siitonen (1993). The Constructive Approach in Management Accounting Research. *Journal of Management Accounting* 5, 243-264.

Louhimies, Kai (2000). *Oliokirja*. Jyväskylä: Satku - Kauppakaari. 422 p. ISBN 951-762-720-3.

Lukka, Kari & Tero-Seppo Tuomela (1998). Testattuja ratkaisuja liikkeenjohdollisiin ongelmiin: kostruktiivinen tutkimusote. *Yritystalous* 4, 23-29.

Lukka, Kari (2003). The Constructive Research Approach. In: *Case Study Research in Logistics*, 83-101. Ed. Lauri Ojala & Olli-Pekka Hilmola. Turku: Turku School of Economics and Business Administration. ISBN 951-564-102-0.

Meilir, Page-Jones (1999). *Fundamentals of object-oriented design in UML.* New York: Addison-Wesley. 458 p. ISBN 0-201-69946-X.

Metsker, Steven John (2004). *Design Patterns in C#.* Boston: Addison-Wesley. 456 p. ISBN 0-321-12697-1.

Microsoft Corporation (2005). *MSDN Library: Reflection (C# Programming Guide)* [online] [cited 8.4.2012]. Available from Internet: <URL:http://msdn.microsoft.com/en-us/library/ms173183%28v=vs.80%29.aspx>

Microsoft Corporation (2010). *MSDN Library: Access Modifiers* [online] [cited 23.3.2012]. Available from Internet: <URL: http://msdn.microsoft.com/en-us/library/wxh6fsc7.aspx>

Microsoft Corporation (2010). *MSDN Library: Model-View-Presenter Pattern* [online] [cited 2.4.2012]. Available from Internet: <URL:http://msdn.microsoft.com/en-us/library/ff647543.aspx>

Microsoft Corporation (2012*). Dev Center - Desktop: Using a Three-Tier Architecture Model* [online] [cited 14.4.2012]. Available from Internet: <URL:http://msdn.microsoft.com/en-us/library/windows/desktop/ms685068%28v=vs.85%29.aspx>

Miller, Mark (2003). *Safe Serialization Under Mutual Suspicion* [online] [cited 8.4.2012]. Available from Internet: <URL:http://erights.org/data/serial/jhu-paper/intro.html>

Oracle Corporation (1995). *The Java Tutorials: The Reflection API* [online] [cited 8.4.2012]. Available from Internet: <URL:http://docs.oracle.com/javase/tutorial/reflect/index.html>

Purdum, Jack (2008). *Beginning C# 3.0 : An Introduction to Object Oriented Programming*. Hoboken: Wiley. 554 p. ISBN 978-0-470-26129-3.

Shalloway, Alan & James R. Trott (2004). *Design Patterns Explained : A New Perspective on Object Oriented Design*. 2. ed. Boston: Addison-Wesley. 429 p. ISBN 0-321-24714-0.

Smith, Josh (2009). *MSDN Magazine: Patterns: WPF Apps With The Model-View-ViewModel Design Pattern* [online] [cited 2.4.2012]. Available from Internet: <URL:http://msdn.microsoft.com/en-us/magazine/dd419663.aspx>

Sobel, Jonathan & Daniel Friedman (1996). *An Introduction to Reflection-Oriented Programming* [online] [cited 8.4.2012]. Available from Internet: <URL:http://www.cs.indiana.edu/~jsobel/rop.html>

APPENDIX 1.

User Questionnaire for Software Designers and Results

Profile

What is your job title?

1. Software Designer
2. Coder
3. Software Designer

What is your education?

1. BEng (IT)
2. University of Applied Sciences
3. B.Eng, Information technology, University of Applied Sciences

Years of programming experience?

1. ~4 years
2. 10
3. 6 years

Years of programming experience with GUI applications?

1. ~3 year, web GUIs
2. 10
3. 6 years

Preliminary questions

Have you had need for similar solution before?

1. No.
2. Yes, but in a C++ project.
3. Yes

Have you implemented solution for similar purpose before?

1. No.
2. Quite, but not in re-usable way.
3. Yes

What GUI design patterns do you commonly use in your applications?

1. MVC (using the ASP.NET MVC framework)
2. MVC, Magic pushbutton, Big ball of mud.
3. Tabbed Document Interface (TDI)

Questions about framework usage

Did you have any problems using / deploying the framework?

1. Only at first, before I got to know the framework better.
2. None. It complied and worked fine.
3. At first I did experience some challenges to get framework deployed. However after I got basic workspace functionality working adding workspace objects to workspace was piece of cake.

Did you use any GUI design pattern? What?

1. No, not this time.
2. None. Tried to use FW in an "already existing project".
3. Multiple Document Interface (MDI)

What programming technique did you use?

1. Winforms, as I'm most familiar with it.
2. WPF
3. Forms

What type of user interface did you use?

1. MDI
2. SDI
3. Multiple Document Interface (MDI)

Was the framework easy to use?

1. At first it was a bit challenging to grasp the concept, but after I started implementing functionality in the framework it became clearer how it worked. The example application did also make it easier, even if I didn't use the same GUI framework
2. Yes, when I got hang of it. Simpler code examples & tutorials would be must if this is taken into wider distribution.
3. Yes, once you get familiar with it.

Were you able to store and restore user workspaces in your application with the framework?

1. Yes. When I corrected a few errors in my code, it worked like a charm!
2. Yep.
3. Yes

Were you able to use the framework without any major architectural changes to your application?

1. Yes. If the application is well built with good separation of UI and logic, I don't see any major problem of using this framework.
2. Had to move program logic out of UI layer, but that stuff shouldn't even be there if done properly (Magic pushbutton)
3. Yes.

What kind of effect did the framework have on the architectural design of your application?

1. Nothing major that is worth mentioning.
2. None, app was simple enough to not have any real architecture. But this FW should be easy to apply if app is properly tiered.
3. In order to use framework it's better to separate GUI components to logical pieces, e.g. views. This helps to handle those components which will be included in the workspace. Using framework will actually help developer to build correct architectural GUI design.

Did you get any additional benefits by using the framework?

1. Not in the current use.
2. Saves the trouble of doing more manual serialization.
3. There's no need to implement separate classes or settings to store user data since framework can also handle user data, e.g. text in text box.

Did you have any disadvantages by using the framework?

1. No, not that I could realize.
2. Maybe app was bit too small to have any real advantage from this FW. Larger app next time
3. No

Do you have actual need for the framework and are you going to use it in upcoming projects?

1. Actually it could be useful in at least one project, but I'm not aware of that the user(s) would actually need it.
2. Not currently.
3. While developing GUI applications there's usually always demand for saving the user workspace. Therefore this framework will provide huge help for upcoming projects while it saves lots of working hours by providing generic working solution for workspaces. In addition it will guide the architectural design of the application's GUI to better direction