Approaches of the Modern Software Development

Oliver Moravcik, Tomas Skripcak, Daniel Petrik and Peter Schreiber

***Abstract*— this article is aimed at the software development process of modern applications. The first part of this article starts with the general classification of information systems based on user interaction characteristics. After an insight into methodologies, methods, design patterns and tools, which are part of modern software development, a life cycle is presented. The second part is devoted to implementing the details of the presented modern trends within a real world application. Lastly, selected drawbacks with proposed solutions are presented. The main goal of this article is to provide an overview of the current modern trends in software development and to point out problems which could be uncovered during the adaptation phase of these disciplines.**

***Index Terms*—composite application, design patterns, ORM.**

I. Introduction

The domain of information technology belongs to one of the most rapidly developing areas in the world. Nowadays, software companies, which intend to be successful within the software development market, must keep their knowledge bases up to date. The task of picking up the right methodologies, techniques and tools is critical when we are talking about delivering high-quality and maintainable software products whilst still keeping time and money costs within reasonable limits. Current modern trends in the development of software applications could help companies in their business but they have to be used correctly and there is the fact that some of them could have a negative impact on the attributes of the resulting software (e.g. performance).

A. Application Posture

The term Application Posture [1] was introduced by Alan Cooper and it basically refers to the way how end users interact with software applications. This characteristic is really fundamental, mainly because it indicates how important the software is for its users. According to Alan Cooper and Robert Reimann there is the following classification of software systems [1]:

1) Sovereign – An application that takes the user's full attention, such as Outlook or Word.
2) Transient – Application in the periphery of the user's attention, calling the user for short moments, such as (for most people) a calculator.
3) Daemonic – Alerting systems.
4) Parasitic – Support interaction mode for both sovereign and transient applications, such as chat.

From a business perspective, sovereign information systems are the most interesting field for the development process. These applications are planned to be used by many users for a long term period, this is why they have to be designed to not only work well now, but also in the future.

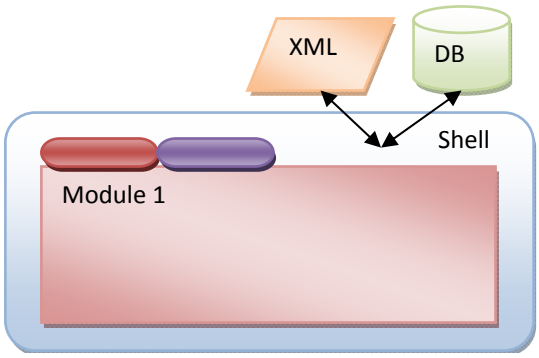

Fig. 1. Composite Application with two data sources

Designing and building applications in a monolithic style can lead to an application that is very difficult and inefficient to maintain. On the other hand there is another class of system developed according to the composite approach. A composite application is created from a group of loosely coupled, semi-independent modules which are easily integrated to a coherent solution called a "shell" [16]. A graphical mock up of such an application is shown in Fig. 1.

B. Cloud Computing

Cloud computing is one of the newest technology words which is used nowadays in many business propagation materials. The definition of cloud gives us an overall idea about how the software based on cloud architecture should behave. [29] describes cloud computing as a term that covers any system which involves delivering hosted services over the internet. Matthieu Hug provides a more precious technological definition of cloud computing:

An emerging computing paradigm where data and services reside in massively scalable data centres and can be ubiquitously accessed from any connected devices over the internet [30].

According to [31] there are three types of cloud computing models which define what types of services cloud offers:

1) *Software as a Service (SaaS):* application running on top of the cloud infrastructure is offered to the end user. This means that the customer can access an underlying infrastructure only via application functions.

Manuscript received September 20, 2011, revised September 21, 2011.

Oliver Moravcik and Peter Schreiber are with the Institute of Applied Informatics, Automation and Mathematics, Slovak University of Technology, Trnava, SK 917 24, Slovakia (e-mail: oliver.moravcik@stuba.sk; peter.schreiber@stuba.sk).

Tomas Skripcak is with the Institute of Applied Informatics, Automation and Mathematics, Slovak University of Technology, Trnava, SK 917 24, Slovakia (e-mail: tomas.skripcak@stuba.sk) and with the Department of Information Technology, Helmholtz-Zentrum Dresden-Rossendorf, Dresden 01328, Germany (email: t.skripcak@hzdr.de)

Daniel Petrik is with MMS SOFTEC Ltd., Trnava, SK 917 01, Slovakia (e-mail: petrik@mms-softec.sk).

2) *Platform as a Service (PaaS):* users are normally allowed to deploy developed or purchased software applications (build on top of supported languages and tools). Control of an infrastructure is restricted similarly as in SaaS.
3) *Infrastructure as a Service (IaaS*): users (in this case they will probably be developers) obtain a full featured platform, which is ready for the development of new software applications with access to all infrastructures which cloud provides.

When we are talking about features which form advantages (and motivation) of building software with cloud foundation, we often end up with following list:
1) *Reliability.*
2) *Scalability.*
3) *Security.*
4) *Maintainability.*
5) *Performance.*

However as it is described in [28] the biggest disadvantages known from the birth of cloud services are: privacy and ownership; these remain unchanged. There is a bias that cloud computing is always connected with public available services which usually lead to the conclusion that it is not suited for companies (privacy issues). As long as we live in an information era it is quite common that information itself will form the largest fortune for businesses. It is unimaginable for such a company to consign their valuable and critical data to a third party organization. In theory [31] we classify cloud computing according to the deployment models in order to reduce privacy and ownership issues:
1) *Private*: hardware and infrastructure is running in the company. No privacy and ownership issues, but the price of the solution is increasing rapidly.
2) *Community*: hardware and the infrastructure are shared between several companies (which share some concerns). A compromise solution is where we can reduce privacy and ownership issues and keep the price at a maintainable level.
3) *Public*: most known cloud solutions can only be used in scenarios where data and information are not critical.
4) *Hybrid*: is a composition between two or more cloud systems, where each of them is deployed in one of the previously mentioned ways and integrated via a standard or proprietary technology.

From a software development perspective, *PaaS* is the biggest advantage of cloud computing. Having one standard which supports software development companies from the process of design through implementation to deployment and maintenance seems like a perfect working environment. [29] labelled this as the "*Dream of Platform Independency*". This is only half of the truth. For the end user the application is truly platform independent, because all infrastructure specific tasks are using grid resources and everything else is distributed to the user over the internet. The infrastructure resources which are offered to the developers via an *API* (Application Interface) are specific for each of the grid providers. It is not possible to take a running system from one grid and move it to a different grid provider without changes. Infrastructure offered by providers consists mainly from large building blocks developed for specific languages and technologies. Currently the only way how the software developers can use a different infrastructure together is by the deployment of a hybrid cloud system, which is not the cheapest solution at all.

Cloud computing is an interesting solution which could finally lead to better, more economic and maintainable software. There are many areas where advantages of cloud overlay its trade off, but it is still only one option for software development. It is hard to believe that cloud could offer enough freedom for young start up projects, which need to have full freedom at an infrastructure level in order to develop new ideas.

C. Modern Methodologies in Software Development

Small and midsized software development companies are often fighting with the need of having high quality methodologies for software's developing process and the possibility of being agile enough to quickly react on the changing requirements from the users plus to reduce the time needed for the iteration cycle in order to produce prototypes of a system and to provide customers with an opportunity to get an insight of the resulting application.

An ideal solution for the previously mentioned problem is to simply compose a basis of trends, disciplines, methods and tools in such a way which will be suitable for the current software project. Below is an overview of the most followed approaches for software design:

1) Model-Driven Architecture (MDA)

Most modern information systems are developed according to an object oriented paradigm. MDA was initially introduced by an Object Management Group (OMG) and provides an approach for capturing system-specifications via the usage of formal models. In MDA, platform-independent models (PMIs) are initially expressed in a platform-independent modelling language, thus as a Unified Modelling Language (UML). The platform-independent model is subsequently translated to a platform-specific model (PSM) by mapping the PIM to some implementation language or platform (e.g. C#) using formal rules [21].

2) Agile Software Development

The idea of agility was firstly used by Kent Beck and transformed into a methodology called *Extreme Programming* (XP). This methodology is described as being easy, effective, low risk, flexible, predictable, scientific and a funny way of software development [2]. The core of agile software development is to use light, but sufficient rules of project behaviour and the use of human and communication oriented rules [3]. In the world of agile development everything is focused on processes in order to deliver a good product, this is why the use of a method or process is always depended on the current project's needs.

3) X-Driven Design/Development (XDD)

The effort of developing better applications does not only present a problem for the use of the latest technologies, one must also have an insight into the stakeholder's domain problem in order to develop a good product. Eric Evans summarized some well known facts about domain modelling

in an object oriented world in his book Domain-Driven Design [6]. Independent from DDD, Richard Pawson introduced an idea of Naked Objects in his dissertation thesis [19]. It basically tries to reduce the development of the software application to the creation of a complex domain model with metadata information, which is used to generate all other modules including User Interface (UI) in runtime. Systems developed on top of the Naked Objects paradigm are presented in a special kind of UI known as Object-Oriented User Interface (OOUI) [19]. As a response to the need of testing the Domain Model early in the software's development life cycle, together with progress of dynamically typed languages, Test-Driven Development was introduced mostly to build on top of the developers Unit Testing. According to [18] a unit test is described as a piece of code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterwards. If the assumptions turn out to be wrong, the unit test has failed. A unit is a method or a function. There is also one modification of TDD which has spread around the software developer's world called Behaviour-Driven Development (BDD), as described in [10]. The main purpose for implementing BDD was due to complexity and a wide range of TDD. BDD is trying to specify a good convention in the process of test writing and execution. Another aspect is that according to BDD only functional user requirements are covered by tests. This means that tests practically become functional user requirement specifications which can be read, but also modified easily. These practices go hand in hand with an agile methodology for software development.

4) Aspect Oriented Programming (AOP)

Aspect Oriented Programming is aimed at cross-cutting problems in object oriented applications, which could not be modelled within the object oriented paradigm [11]. An example of such a problem is functionality (e.g. logging application messages) which should be applicable on different types of classes (Business Entities, Infrastructure Services, etc.). This type of functionality is encapsulated into a routine called Aspect which makes software more reusable and maintainable. Nowadays there are AOP frameworks, for most software programming languages, which have become mainstream in application programming. These frameworks could be taken as an enhancement of the object oriented paradigm.

5) Data Access Strategy

Relational Database Management Systems (RDMS) were and still are standard preferred solutions in the market of corporate information systems, mainly because they are mature enough and their wide field of usability is a determining factor for many types of software systems. They build on top of relational algebra, which give them a solid mathematical background. Modern RDMS were also able to adapt themselves to the needs of the developers (e.g. there is no problem to work with documents in eXtensible Markup Language XML). In order to allow fluent usage of RDMS in today's modern object objected applications, we need to create a software bridge between the object and the relational area. Object Relational Mapping (ORM) [12] is the technology which is normally used to accomplish this goal. ORM frameworks are able to generate a connection between your application and RDMS in a semiautomatic/automatic way. Best of all is that by incorporating an ORM layer into the system design we can use it as a generic way to access the data from any supported data storage. An example of the layered business application with ORM's persistence layer is shown in Fig. 2.

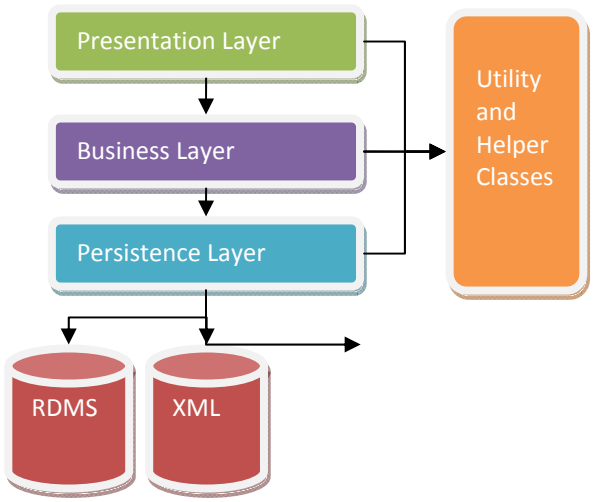

Fig. 2. Standard three layered architecture of a business system [12]

Of course we know about other data storage paradigms as Object Oriented Database Managements Systems (OODMS) or Document Oriented Database Management Systems (DODMS). According to [26] from the beginning of 2009 we could identify movement in the usage of the data storage strategy. Especially in the field of scalable web applications a demand for a schema-free database system arises. The world knows this initiative under the name NoSQL (usually translated as not only sql) or Next Generation Databases. Common features of NoSQL databases are enumerated bellow:

1) *Flexible.*
2) *Scalable.*
3) *Distributed.*
4) *Schema-free.*

It is important to say that nowadays the NoSQL approach is mainly used in the development of modern web applications running on top of cloud services. It will be interesting to see how this field of informatics will evolve in the future, however for now when we are developing an OO application with an RDMS system as a storage background, ORM simple wins the battle.

D. Architectonic Design Patterns

Design patterns are standardized solutions for solving typical and not elemental problems in object oriented programming. The special class of design patterns is used for an overview of the whole architecture for an object oriented system. These patterns are called architectonic. Current trends are mainly based on an idea of eliminating dependencies between various components of a software system. One of the most successful architectonic design patterns nowadays is the Model-View-Controller (MVC).

The idea of MVC is quite old. It was developed at Xerox PARC in 1978/79 by Trygve Reenskaug, but it gained its position in the world of mainstream software development only a few years ago. The basic idea was that the model will be an abstraction of the domain model, the view will contain a user presentable interface and the controller will coordinate the capabilities of several views making it a comprehensive tool [19]. Variations of the MVC pattern, like Model-View-Presenter (MVP) and Model-View-View Model (MVVM – in some literature you can also find View Model under the name Presentation Model) are commonly used in modern software in order to fully utilize the underlying framework of software applications. In Fig. 3, there is an example of how MVC, MVP and MVVM patterns behave.

We would like to point out another pattern, which was introduced by Rinat Abdullin and is called Command and Query Responsibility Segregation (CQRS) [1]. It is still in the process of transforming into its final form, but provides us with some interesting ideas. First of all, CQRS differentiates between commands, with a purpose to modify data in a storage system and to query the produced readable information for users in order to make responsiveness applications. Read operations are much more often needed in comparison with operations for data manipulation. Secondly, usage of CQRS entails that the domain model and especially domain objects are not presented to end users in their base form. The domain model in CQRS is only used as an abstraction of the domain problem, which contains all business logic for data modifications and also business events. However, for presentational purposes Data Transfer Objects (DTO) are used as a lightweight wrapper when presenting readable data to the user interface [1]. CQRS was designed in order to benefit from cloud computing architecture of the system. One characteristic of the cloud system is that resources like computing power, storage services, etc. are outsourced. It gives us an opportunity to be more scalable and more cost efficient, however it is still up to developer to access these outsourced services in an efficient manner.

E. Loosely Coupled System

Systems that consist of many modules and infrastructure code which depends on each other are known as close coupled systems. Direct dependences are a sign of the bad design in modern object oriented applications. The inversion of control is technique where the control of the specific tasks is not strictly dependent on the application itself but rather allows the user (which is in this context usually a developer) to plug-in task functionality by oneself. There are two design patterns which take advantage of inversion regarding the control idea:

1) *Dependency Injector (DI)* – provides a structured way for the organisation of dependencies so one never has to request the dependency directly in order to use it, but the dependency is provided by default (e.g. via the interface in the constructor or as a class property).
2) *Service Locator* – hides the dependency into the locator and so in order to use it we simple have to ask the locator for it.

DI has a much wider effect and introduces new complexity into application development. The objectives of DI are summarized in [27] :

1) *Object composition.*
2) *Lifetime management.*
3) *Interception.*

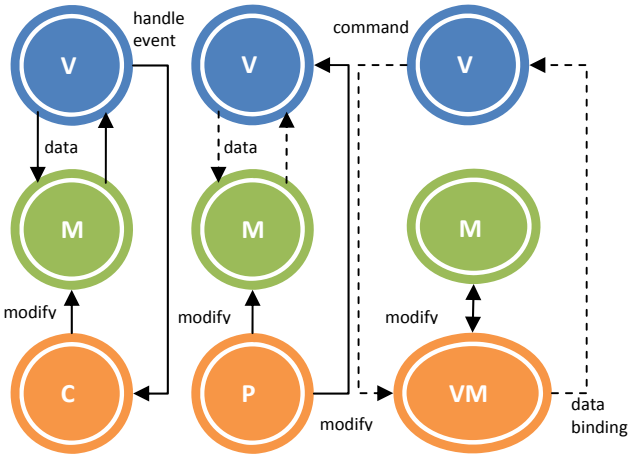

Fig. 3. Behavior of MVC, MVP and MVVM patterns

According to [22] the benefits of using the Dependency Injection technique are as follows:

1) Late binding – services can be exchanged easily.
2) Extensibility – there is a possibility of extending and reusing components without a predefined plan.
3) Parallel development – development of a different part of the system which depends on each other (large projects).
4) Maintainability – responsibilities are clearly defined.
5) Testability – everything can be covered with unit tests.

It is up to developer to choose which pattern he will use in order to decouple a developed information system. Dependency injection gives him a lot of power but could potentially lead to code which is harder to debug. On the other hand a service locator is quite straightforward but each module which is using it and has to have a dependency on the locator. For more deeper information about the implementation and usage of the inversion of control please refer to [22][27].

F. Domain Framework Languages

Optimisation of the internal business processes is one of the aspects necessary for a successful company. We can see a trend that each branch of industry is trying to formalize and store workflows and processes in order to evaluate and optimize them. Having a formal process description is also valuable when it comes to information and automation of specific tasks in a company's information systems. Nowadays it is possible to work with domain experts and create a graphical description for processes which have to be implemented in software applications. We can go even further and use this graphical description as an execution language. In the list below we enumerate some of the technologies used for this purpose:

1) *UML (Unified Modelling Language):* as a standard modelling language used in software development, it

can also be used for a description of processes and connect the generated implementation code with a UML model.

2) *BPML (Business Process Management Language), BPEL (Business Process Execution Language):* a mix of these three technologies allows us to create a visual representation of processes for a company in the way that non IT people can understand, so they can contribute during the creation of a formal workflow. This standard is used by companies like IBM, Oracle within their products. BPEL gives us an opportunity to transfer a visual representation of the process into a running implementation [33].
3) *Windows Workflow Foundation:* in technology used for design and execution of workflows on top of the Microsoft .NET platform. It was radically rewritten and extended in version 4.0 and forms a direct competition to the BPML + BPEL group.

All techniques mentioned above count on the graphical design of workflows. The biggest problem with them is that in order to develop an application with a standardized workflow, developers have to follow a specific methodology and they are also restricted on usage of the concrete stack of development tools directly connected to them. As an opposite to this rigid style of development the *DSL* (*Domain Specific Languages*) [34] differs, which should provide us with a lightweight way for expressing domain experts' rules and processes in a textual way which is readable for them but are also executable in the information system. The domain specific language is defined as:

A computer language that is targeted to a particular kind of problem, rather than a general purpose language that is aimed at any kind of software problem [34].

Many people think that in order to use DSL they have to develop their own language and write a compiler for it (this type of DSL is called *external DSL*). This can of course be done and it also offers us a lot of flexibility, however we could use existing languages like C# or java (which do not have enough syntax flexibility) for the purpose of creating a simple DSL language (*internal DSL*). *Fluent Interfaces* is the name of the design guideline which teaches us how to organize our domain logic and infrastructure application code in order to form an internal DSL language. By following the rules like: object methods that always return affected objects, adding support for generics and modern constructs which form functional languages (lambda expressions) we can reach a point when constructs in standard programming languages are easily readable for non IT people and so we can share knowledge with them and even let domain experts express their domain processes in computer executable language (be aware that such a code should be always checked and approved by the developer).

G. Continuous Integration

The term integration is used for describing an activity for combined software components into a system as a complex unit [15]. Continuous Integration is a software development practice where members of a team integrate their work frequently. Usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible [7]. Integration is also related with Version Control, which is one of the aspects in the Software Configuration Management [13]. Having Version Control within the system is necessarily when we want to develop a high quality software application. There are two main classes of the Version Control system:

1) *Centralized* – This is built on top of client-server architecture. The server plays the role of source code, is primary a repository and each client has to communicate with the server in order to perform some action such as: commit/check in, check out, view history, revert changes etc. A typical workflow consists of the following actions: check status, update if necessary, resolve conflicts, perform changes and at the end commit changes.
2) *Decentralized* – Nowadays the decentralized version controls solutions and has become more popular than the centralized version. These systems are built on top of peer to peer architecture, where each peer contains a repository and changes are distributed from peer to peer as patches. The advantage of this approach is that many operations do not need access to the network and that is why they have better performance in comparison with the centralized control systems. The typical workflow in a decentralized version control system allows each developer to only perform an update operation in order to obtain the most current version of code. His part of an implementation is newly committed but instead of that it is packed into a so called *patch*. Then it is up to the one developer authority to update all the patches from active members and integrate everything together.

II. The Implementation and Elimination of Drawbacks

In the real world, there is always a risk involved when the decision to adopt a new technology, methodology or paradigm is made. Each member of the team usually has to change their way of thinking about problems and this can take some time. The implementation of new tools could also involve some problems, e.g. performance (because we usually create another level of abstraction) and the errors which were not visible during the change preparation time will be exposed in the development process.

A. Usage and Limitation of ORM

The main reason for the usage of ORM technology was in shielding the software developer form any interaction with the database level of an application. The developer can be fully focused on the domain model, which contains all domain logic and does not have to bother with SQL (Structured Query Language) commands and relational database schema [20], [23]. This is the reason why the developer is much closer to the world of the end user and can express ideas in a language called Ubiquitous Language which is [6]) understandable for both of them.

There are many ORM frameworks on the market, so preliminary research and tests are necessary before choosing the product which will be used in the software project. For

the purpose of a persistence strategy, we have chosen ORM framework as described in [5]. In the following list, there are outlined features that are important for ORM, when used as a persistence mechanism for a composite system, which is developed according to modern trends [20]:

1) Support for many database systems and persistence storages (at least MS SQL and Oracle).
2) Schema creation and schema update – Ability to create database schema and update it if necessary from a class model or domain object. It enabled us to have one object oriented domain model modelled in CASE (Computer Aides Software Engineering) tool. It meets MDA but is also an agile way of thinking about software development.
3) Automatic mapping relations between objects (1:1, 1:N, M:N) to the database structure.
4) Support for transactions – Unit of Work design pattern (for more information please refer to [8])
5) Superior query mechanism (ideally strongly typed) – Query should not be typed as a string. Enables the compiler to check queries (in our case, we required the implementation of LINQ [14] technology).
6) Automatic notification when a change in an attribute value occurred – It enables rich UI experience in a Composite Application.

During the development process we have discovered some limitations and problems which were involved with the application of ORM into the real life development cycle. They are listed in the following list [20]:

1) Processing large amounts of data – Modification of any object's attributes require the reading of the object in memory. This can lower performance, e.g. when the aggregation of a root domain object with many associated objects takes place or with a more deeper hierarchy (object in tree) is deleted, all objects waiting for deletion should first be read from the database to the memory and the deletion process takes place afterwards. We solved this issue by analysis of the application of bottlenecks and by defining the *cascade delete* rule in the relational database system.
2) The impossibility to join objects, which do not have relations between them, in the query. The only working solution that we discovered is the definition of a database view. It can be wrapped into the object and used in the query afterwards.
3) The problem with querying calculated attributes – In some cases calculated attributes of the domain object are needed as a part of the query expression. In order to gain a value of the calculated attribute the whole object has to be read from the database, which has a significant negative impact on the query performance. The solution is to define the procedure which will be calculating the value on the relational database system but the negative side of this is that we will have duplicated business logic. This is recommended only in bottlenecks of the application.

B. Modern User Interface Composition

Actual "rich" business applications typically feature multiple screens, rich, flexible user interaction, data visualization and role-determined behaviour. The application's expected lifetime is measured in years and that it will change in response to new, unforeseen requirements. This application may start as small and over time evolve into a composite client [20].

We have based our project on the guidance [16] for building the next generation application in WPF (Windows Presentation Foundation) technology provided by Microsoft. This contains a set of standards, design patterns and libraries which help to solve common problems in composite application development. The most important is probably the implementation of the architectonic design pattern called MVVM. We have slightly modified it to meet all our needs:

1) View

XAML (eXtensible Application Markup Language) is used for the purpose of the View implementation. We decided to use "in view" constructing of UI and composition technique like templating, styling and data binding with automatic change notification. WPF provides us with all the necessary foundations. View should contain only functions directly connected with the application UI.

2) View Model

View Model class have access to all data and actions which the user can perform through UI. It transforms the data in for the purpose of displaying it to the UI. We implement the View Model's class as simple and reusable as possible. In the context of a composite application VM is created with the help of the DI approach and it automatically runs the initialization of the View component which is provided to the end user to work with immediately.

3) Model

Model is defined and generated from UML (Unified Modelling Language) Class diagrams [24], [25]. It contains all business logic and is extended with checking validation and business rules. It also serves as a basis for ORM database generation and manipulation.

Model-View-View Model design pattern is not only applicable for the architecture of a software system. Some of the new software's frameworks use this approach successfully in order to obtain a higher configurability of its components. As an example we can state WPF framework for building a user interface. In the terms of WPF every control or component which will be displayed on the screen is, by default, look less. When we need to define a graphical shape of the user control, we create an element called a control template. The control templates are defined by a framework of course, but there is still a possibility to override these settings and to modify them to meet our needs. The other purpose of the control template is for specifying the place where the data will be presented on the control. Most *LOB (Line of Business)* applications are data driven and binding declarative binding of data to UI and also changes from UI back to data, are another aspect of the modern UI composition. In WPF the data template can be used to define the UI element in which the provided data will be presented. Converters can also be used to shape the data. Finally the style is applied on the user control to create a delightful graphical feeling. In the context of an object oriented model

we would say that composition is the feature which is used in the modern UI paradigm, where the user control is directly composed from other base controls and this process can be done recursively until the final stage is reached.

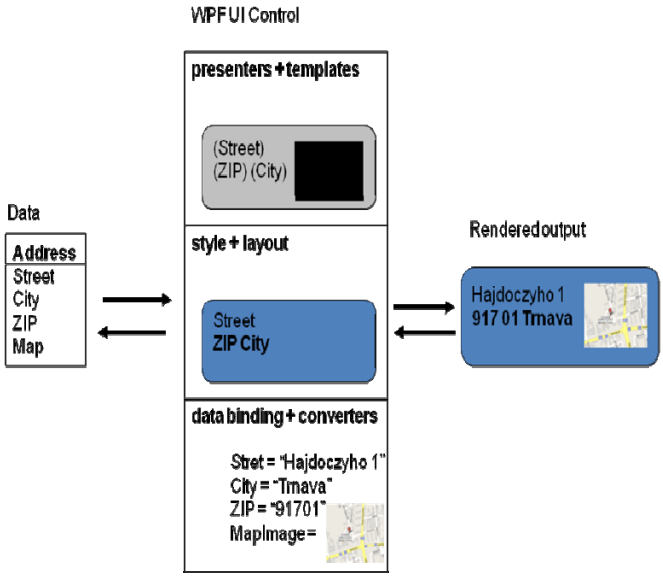

Fig. 4. Visual output composition [20].

Fig. 4, shows how all this could work together in order to provide a consistent and responsive user interface. This way of thinking about UI could be applied to different platforms, e.g. mobile, web and client. WPF is oriented on rich desktop applications but it is a big step forward. When we are considering the web, which is actually much less interactive (in the context of web applications), it has its own user interface description language (HTML – Hypertext Markup Language) and styles written as CSS (Cascading Style Sheet) rules have been used for defining graphical visuals for many years, desktop UI development, on the other hand, has been very limited until present times.

C. Business Rules and Validation

One of the problems with a rich composite application is that users except detailed validation and business rules to be checked on the client's side. Most of the frameworks including WPF offer features which are able to provide such a type of user experience. However the disadvantage is that it often leads to duplication of the business rules and a validation code (special WPF validation classes plus validation and business rules included in the domain model). This tends to make everything less maintainable. In order to make the process more flexible we designed our domain model with a core set of business rules which have to be valid no matter what. This helps us use the domain model out of the field of the main information system (e.g. supporting importing/exporting applications). Validation rules themselves, which are not specific for the business entity but differ depending on the view or application where it is used, are stored in special metadata classes (without any WPF specific constructs). This validation is loaded dynamically during the construction of user interface. Usually each amount of metadata is relevant for a specific purpose but nothing holds us from using it in different cases (if it meets our needs). Some of the frameworks on the market (e.g. ORM frameworks used in the context of a domain model [5]) offer each entity a validation, but we want to make it clear that in practice a business entity is in a valid state that mostly depends on usage of an entity. This means that it is never a good idea to enclose full featured validation into domain model.

D. Convention over Configuration

Convention over configuration is a useful paradigm and used in agile application development and is focused on the problem of application infrastructure complexity. In the conventional software company, developers are trying to implement infrastructure components (which support the application domain model) in a most generic way. The original idea is that the component is developed only once and that it can be used in different projects. This means that before we use such a generic component, we have to perform specific configurations for each project. Our empirical experiences (and also experiences of others experts in industry, e.g. [32]) shows us that it is very unlikely that we will share an application infrastructure between different systems. So we can get rid of configuration complexity by implementing the idea of convention over configuration. In order to do this, it is necessary to determine a specific set of rules, which will be respected during the development. These rules cover, e.g. naming conventions (for classes, modules, features etc…), workflows, patterns and structures. The advantage is clear and easy to maintain an application's infrastructure code, which does not suffer due to overdeveloped complexity because we restricted it for the purpose of one application.

E. Managed Languages and Performance

In object oriented applications written on top of a software framework there is a component responsible for memory management called a "*garbage collector*" (GC). Its main goal is to delete unused objects from the memory heap in order to prevent performance problems. However there are situations when dependency between objects does not allow GS to dispose of objects from memory. We had also discovered this problem, when we were dealing with the implementation of MVVM patterns in the composite WPF application. We found out that some View-Model components exist in memory more than once, which has a negative impact on performance. We examined this behaviour and it seems that it occurred only when a data object from the View Model is presented by the user control (technical speaking property DataContext of any user control keeps reference to the data object), but has not implemented the "INotifyPropertyChanged" interface (.NET interface for propagating change in data immediately to the UI) [20].

According to [9,17] WPF uses the "ValueChanged" event, which involves calling the "PropertyDescriptor.AddValueChanged" method on the "PropertyDescriptor" object that corresponds to property. Unfortunately, this action causes that the Common Language Runtime (CLR) which also keeps a reference to the "PropertyDescriptor" object in a global table.

The diagnosis of memory leaking seems to be simple especially when a memory profiler is used. This tool can help

to find out "paths to a GC root", which can be used to identify a problem object and its references [20].

III. Open Issues

Development of modern information systems is a complex process and many tasks are still very difficult to accomplish. For example there is not a well defined standard on executing parallel operations with an asynchronous user interface.

Another issue involves complex testing (involving testing components for other vendors) of an application. Currently an ideal solution for this task is the utilization of the UI testing tool.

IV. Conclusion

The aim of this article is to provide an overview of modern disciplines during the development process. It outlined some problems which are likely to come along during learning and the adoption of these techniques and solutions which are offered in the context of a real world information system. Modern trends in the application development industry have of course a positive effect and try to simplify the routine tasks of the developer. We have to make it clear that all progress needs time so that software companies can adopt and use it correctly.

References

[1] R. Abdullin, "*CQRS Starting Page*" [Online 2009], [cit. 2011-01-20]. Available on the Internet <:http://abdullin.com/cqrs/.>.

[2] K. Back, "Extrémní Programování," Grada, 2002, Praha, p. 158, ISBN: 80-247-0300-9.

[3] A. Cockburn, "*Agile Software Development*," Addison-Wesley, 2001, pp. 8-9, ISBN: 978-0-201-69969-2.

[4] A. Cooper, R. M. *Reimann "About Face 2.0*," Wiley, 2003, Indiana, p. 504, ISBN: 978-0-764-52641-1.

[5] DevExpress, "*eXpress Persistance Objects*" [Online 2006], [cit. 2011-01-07]. Available on the Internet <:http://www.devexpress.com/Products/NET/ORM/.>.

[6] E. Evans, "*Doman-Driven Design: Taskling Complexity in the Heart of Software*," Addison Wesley, 2003, New York, p. 560, ISBN: 0-321-12521-5.

[7] M. Fowler, "*Continous Integration*" [Online 2006], [cit. 2011-01-22]. Available on the Internet <:http://www.martinfowler.com/articles/continuousIntegration.html.>.

[8] M. Fowler, "*Unit of Work*," [Online 2002], [cit. 2011-01-09]. Available on the Internet <:http://martinfowler.com/eaaCatalog/unitOfWork.html.>.

[9] J. Goldberg, "WPF *Performance and .NET Framework Client Profile*" [Online 2005], [cit. 2010-11-29]. Available on the Internet <:http://blogs.msdn.com/b/jgoldb/archive/2008/02/04/finding-memory-leaks-in-wpf-based-applications.aspx.>.

[10] C. Kaner, J. Bach, "*Introduction to BDD*" [Online 2006], [cit. 2011-01-15]. Available on the Internet <:http://dannorth.net/introducing-bdd/.>.

[11] G. Kiczales, J. Lamping. A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, J. Irwin, "*Aspect-Oriented Programming*," ECOOP, 1997, Finland, p. 25.

[12] P. H. Kuaté, T. Harris, C. Bauer, G King, "*NHibernate in Action*," Manning, 2009, p. 400, ISBN: 978-1-932394-92-4.

[13] W. Lewis, G. Veerapillai, "*Software Testing and Continuous Quality Improvement*", second ed, Auerbach publications, p. 534, 2005, ISBN: 0-8493-2524-2.

[14] F. Marguerie, S. Eichert, J. Wooley, "*LINQ In Action*," Manning, 2008, p. 576, ISBN: 1-933988-16-9.

[15] S. McConnell, "*Dokonalý kód*," Computer Press, a.s.,Brno, 2006, pp. 694, ISBN: 80-251-0849-X.

[16] Microsoft Patterns & Practices Team, "*Prism 4.0*," [Online 2010], [cit. 2011-01-13]. Available on the Internet <:http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3453ab2b-2067-41e4-b087-312d8385cf1b&displaylang=en.>.

[17] Microsoft Support, "*Memory leak in WPF,"* [Online 2005], [cit. 2010-11-29]. Available on the Internet <:http://support.microsoft.com/kb/938416/sk.>.

[18] R. Osheroves, "*The Art of Unit Testing*," Manning, 2009, p. 320, ISBN: 1933988274.

[19] R. Pawson, "*Naked Objects*," University of Dublin, Trinity College, 2004,Dublin, p. 223.

[20] D. Petrík, O. Moravčík, "*Modern Trends in the Development of Software Applications*," International Workshop Innovation Information Technologies – Theory and Practice, 2010, Dresden, pp. 3-5, ISBN: 978-3-941405-10-3.

[21] J. D. Poole., "*Model-Driven Architecture: Vision, Standards And Emerging Technologies*," ECOOP Workshop on Metamodelling and Adaptive Object Models , 2001, p. 15.

[22] M. Seemann, "*Dependency Injection in .NET*," Manning, 2009, p. 375, ISBN: 978-1-935-18250-4.

[23] Tanuška, P., Važan, P., Schreiber, P., "*The Partial Proposal of Data Warehouse Testing Task*." In: Proceedings of the ISCCC 2009 International Symposium on Computing, Communication and Control, Singapore October 9-11, pp. 242 – 246, published by International Association of Computer Science and Information Technology, ISBN 978-9-8108-3815-7.

[24] Jedlicka, M., Moravcik, O., Schreiber, P., Tanuska, P., "*Reliability Assessment Using UML Models."* DAAAM International Scientific Book 2009, Vienna 2009, pp. 53-60. ISBN 978-3-901509-69-8, ISSN 1726-9687

[25] Jedlicka, M., Moravcik, O., Schreiber, P., Tanuska, P., "*UML Support for Reliability Evaluation*". In: Proceedings of the ISCCC 2009 International Symposium on Computing, Communication and Control, Singapore October 9-11, pp. 257 260, published by International Association of Computer Science and Information Technology, ISBN 978-9-8108-3815-7f

[26] NoSQL, "*Non-Relational Universe*" [Online 2009], [cit. 2011-09-07]. Available on the Internet <:http://nosql-database.org/.> .

[27] Fowler, M., "*Inversion of Control Containers and the Dependency Injection pattern*", [Online 2004], [cit. 2011-08-08], Available on the Internet <:http://martinfowler.com/articles/injection.html>.

[28] H. Erdogmus, "*Cloud Computing: Does Nirvana Hide behind Nebula?,*" IEEE Computer Society, Los Alamitos, CA, USA, 2009, Available on the internet <:http://www.computer.org/portal/web/csdl/doi/10.1109/MS.2009.31>.

[29] Search Cloud Computing, "*Definition of Cloud Computing*", [Online 2008], [cit. 20-11-09-08]. Available on the Internet<:http://searchcloudcomputing.techtarget.com/definition/cloud-computing>.

[30] Hug, M., "*Will Cloud-based Multi-Enterprise Information Systems Replace Extranets*", [Online 2008], [cit. 20011-09-07], Available on the Internet <:http://www.infoq.com/articles/will-meis-replace-extranets>.

[31] Mell, P., Grance ,T., "*The NIST Definition of Cloud Computing, Recommendations of the National Institute of Standards and Technology*", [Online 2011], [cit. 2011-09-08]. Available on the Internet <:http://csrc.nist.gov/publications/drafts/800-145/Draft-SP-800-145_cloud-definition.pdf.> .

[32] Eisenberg, R. "*BlueSpire*" [Online 2011], [cit. 2011-09-05,] Available on the Internet <:http://www.bluespire.com/>.

[33] Object Management Group, "*Business Process Management Initiative*", [Online 2011], [cit. 2011-08-29], Available on the Internet <:http://www.bpmn.org/>.

[34] Rahien, A., "DSL *in Boo: domain-specific languages in .NET*", Manning, 2010, p. 352, ISBN:978-1-933988-60-3

Oliver Moravcik was born in Male Levare/Slovakia in 1952, studied Automation at Technische Hochschule Ilmenau/Germany, in 1976 gained his master's degree and completed his Doctoral study at the same university in 1982 in the field of Automation and Computer Science. Since 1982 he was contracted as a senior lecturer and in 1990 became an associated professor of automation and informatics at the Slovak University of Technology in Bratislava /Slovakia. In 1998 he became a full professor of automation and informatics at the same university and from 2006 became the Dean of the Faculty of Materials

Science and Technology of the Slovak University of Technology in Bratislava/Slovakia. During 1986 – 1988 he was a visiting professor at the University of Applied Sciences Koethen/Ger,many and during 1990-1992 at the University of Applied Sciences Darmstadt/Germany. The spheres of his interest are software engineering and intelligent control methods. He published more than 120 articles in international journals and in the proceedings of international conferences. He participated or leaded the solution of more than 20 grant projects and/or industry contracts.
Professor Moravcik is at present a member of AC SEFI in Brussels, IFAC and also a member of the International Informatisation Academy Moscow/Russia.

Daniel Petrik was born in Ilava/Slovakia in 1968. He received his diploma degree in the field of Manufacturing Systems and Robotics from Slovak University of Technology in Bratislava in 1991. After finishing his studies he worked at the Slovak University of Technology in the field of process automation and information (1991-1996). In 1994 he started his part time job in MMS Softec Ltd (Trnava/ Slovakia) as a junior software developer. In November 1996 he left the Slovak University of Technology and became a software architect in MMS Softec Ltd (Trnava/Slovakia). In MMS Softec Ltd. he designed the architecture of the following information systems: easy, e-VEGA, Proman NG®, participated on the design of the system Proman W®.
He is publishing papers in national and international conferences and journals. MSc Daniel Petrik's research area of interest is software systems design, testing and its automation.

Tomas Skripcak was born in Trnava Slovakia in 1986. He received his Diploma degree, in the field of applied informatics and automation in industry, from the Slovak University of Technology in Bratislava in 2010.

During his studies (2005-2011), he worked as a software developer in MMS Softec Ltd. (Trnava, Slovakia), where he was responsible for design, development, documentation and testing of information systems based on .NET technology. In 2008, he obtained the dean's award for excellent bachelor thesis (Design and the implementation of a helpdesk). During February - May 2010, he was an Erasmus student at KaHo Sint-Lieven (Gent, Belgium), where he worked on a Diploma project (Generation of medical knowledge out of data) for Agfa HealthCare (Gent, Belgium). At present, he is a PhD student at the Slovak University of Technology in the field of process automation and information. Since February 2011, he has been situated in Germany at Helmholtz-Zentrum Dresden-Rossendorf. Here, he is dealing with the problem of natural human machine interaction in virtual reality applications. He is publishing papers in national and international conferences and journals.

Ing. Skripcak's research area of interest includes software systems design and testing, natural user interface design and novel ways of human-computer interaction.

Peter Schreiber was born in Bratislava, Slovakia, in 1960. He received his diploma degree in the field of Control Theory from the Slovak University of Technology in 1984 and a PhD. degree in the field of Automation from the same university in 1992. He became an associated professor of the Slovak University of Technology for Applied Informatics and Automation in the year 2000.

He has worked as assistant lecturer and lecturer in the Slovak University of Technology. In the years 1995 – 1996 he was a lecturer in Koethen in Germany. The spheres of his interest are software systems development and intelligent control methods. He published more than 100 articles in international journals and in the proceedings of international conferences. He participated or leaded the solution of more than 10 grant projects in the areas of software development and control systems.

Assoc. prof. Schreiber, PhD. is a member of the international organisations IFAC (International Federation of Automatic Control) and the International Union of Machine Builders.