SOFTWARE ENGINEERING 1252

SOFTWARE ENGINEERING

GENERAL CONCEPT OF THE PLATFORM INDEPENDENCY MODEL

Siniša Vlajić[1], Ilija Antović*[1], Dušan Savić[1]
[1]University of Belgrade, Faculty of Organizational Sciences
*Corresponding author, e-mail: ilijaa@fon.bg.ac.rs

Abstract: *This paper presents the ways and types of interpretation of the concept of the platform independency, as well as the tools and mechanisms of its realization in real world environment. The paper introduces General Concept of the Independency (GCoI) model. The model is derived from most important platform independence mechanisms. The paper identifies four key mechanisms for achieving platform independence, and all of them are described using GCoI model. The GCoI model represents the fundamental concept that lies behind the platform independence.*

Keywords*: platform independency, General Concept of the Independency (GCoI) model, mechanisms for achieving platform independence, software engineering, platform independent software architecture*

1. INTRODUCTION

Appearance of new software architectures (Liu et al. 2011; Bass, Clements and Kazman, 2003; Gorton, 2011) and software platforms, especially in context of cloud computing approach, impose the need and challenge for identification and understanding of concepts and mechanisms which enables integration and adjustments in heterogeneous environments.

The main goal of this paper is to identify the mechanisms for achieving independency between software architectures and software platforms. While developing new platform or software architecture, engineers should be aware of all necessary elements that need to exist in order to achieve platform independence. In that sense, we will first try to define platform independent architectures and to identify main characteristics software architecture needs to poses in order to be considered as platform independent. In addition, we will try to classify the types of platform independence, and to identify mechanisms that are used for achieving each type of platform independence. By analyzing identified mechanisms, we will establish a general model for achieving platform independence. We will use platform independent architectures such as SOA (Vitvar et al. 2007; Erl, 2005), COA (Gorton, 2011), and MDA (Meghan; Object Management Group, 2003), and most important software platforms Java and .NET as examples for practical explanation of the general model for platform independence.

In this paper, we defined and explained the platform and platform independent software architectures and referred to the main problems in platform independent software development in the section2. Section 3 introduces the platform independent classification. This classification is based on noted relationships between software architecture and software platform from which we derivate different type of platform independency. The next section, describes four key mechanisms used for platform independency. Each of these mechanisms is explained by the General Concept of the Independency (GCoI) model. The GCoI model represents the fundamental concept that lies behind the platform independence and in the last section (section 5) authors point out importance of this model and emphasize future directions of research.

2. THE PLATFORM INDEPENDENCE

This section defines and explains the platform and the platform independence concepts as well as platform independent software architectures. It also discusses key concepts related to platform independence and objectives that lead to its realization. The part of this section refers to the main cause of platform independence implementation problems.

2.1. Definition of platform

The platform independence and platform as a concept appear in a various segments of the software engineering science. Consideration of the platform independence concept begins with considering the concept of platforms and their basic classification.

According to Object Management Group, a platform is defined as:
"Set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns that any subsystem that depends on the platform can use without concern for the details of how the functionality provided by the platform is implemented (Object Management Group, 2003)

A typical definition of a platform is: *A platform is a combination of hardware and software used to run software applications. A platform can be described simply as an operating system or computer architecture, or it could be the combination of both.*

There are two basic types of platform:

- A hardware platform can refer to a computer architecture or processor architecture. For example, the x86 and x86-64 CPUs make up one of the most common computer architectures for general-purpose computers.
- Software platforms can either be an operating system (Microsoft Windows, Linux, Mac) or implementation technologies, though more commonly it is a combination of both. Java and .NET are software platforms; they represent a set of technologies and programming languages, which together with specialized development environments enable the development of complex software systems. Java and .NET represent platforms that enable applications (implemented using these technologies), to work on multiple operating systems and hardware platforms.

A platform represents a subsystem or a set of subsystems that provide certain functionality. Platforms represent foundation for execution of other software systems and applications. The Figure 1. illustrates the basic platforms classification.

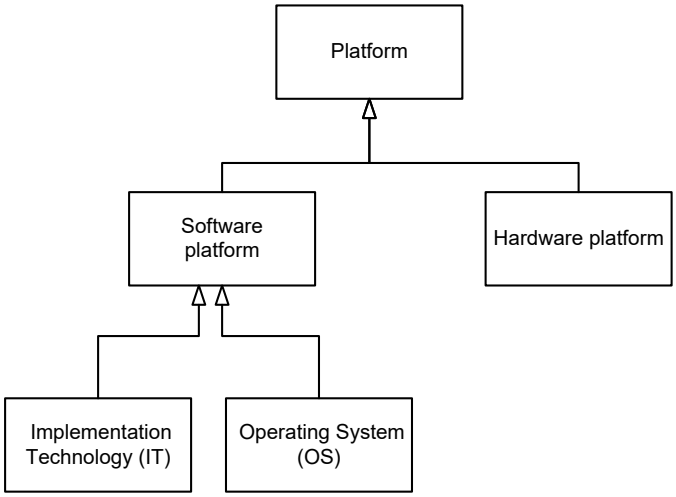

Figure 1: Basic platform classification

2.2. Platform independent software architectures: definition and characteristics

In its strict sense, a software architecture is "a description of the subsystems and components of a software system and the relationships between them." (Buschmann et al. 1996) Software architectures that will be further discussed are Model Driven Architecture – MDA (Meghan), Component Oriented Architecture – COA (Gorton, 2011) and Service-Oriented Architecture - SOA (Erl, 2005).

The platform independent software architecture does not include any specifics of implementation technologies, nor the details of target hardware platforms or operating systems on which software applications executes.

The platform independent software architecture should meet the following requirements:

- Platform independent software architecture does not contain details relating to the hardware platform on which software executes.
- Platform independent software architecture does not contain details relating to the operating system on which the software executes.
- Platform independent software architecture does not contain details relating to the implementation technology.
- Platform independent software architectures are specified with universally accepted languages for software systems specification.
- Platform independent software architectures can have large number of implementations.
- Platform independent software architectures are based on platform abstraction.

Because implementation and target platform details are excluded, the participants in the development process may focus on the key and essential aspects of software architecture and software systems. Also, it is much easier to explain the architecture to non technical people, who have interest or influence on the software systems financing and implementation.

Universally accepted notation and method makes platform independent software architectures important in understanding the software system and making the communication more comfortable.

2.3. Platform independence concept: goals and importance

The concept of platform independence is related to the concepts of portability, reusability, universality and financial viability. Portability, multiple usage and financial viability can be considered as primary objectives for realization of platform independent software architecture.

Universal and widely accepted rules have not been changed as quickly as the hardware platforms, operating systems or technologies. Universally accepted principles often go beyond the boundaries of time and space in which they have appeared. Platform independent software architectures tend not to obsolete rapidly, because they keep the knowledge and experience accumulated over years and generations, knowledge that can be reused in a number of situations. Such architectures are strong foundation for future development of software systems.

Based on the above mentioned we can conclude: The ultimate goal of platform independence realization is software applications that can run on many operating systems and hardware platforms, without modification or adaptation. Such solutions have maximum portability level.

The ultimate objectives of the software architecture platform independence realization in respect to the software platform are:

- Provide that the same software architecture can be used for various implementations using different technologies.
- Protect the results of software design process from the danger of rapid obsolescence, caused by the changes in technology, operating systems and hardware platforms area or platforms of any kind.

3. THE PLATFORM INDEPENDENCE CLASSIFICATION

While considering platform independence, it is necessary to specify the type of platform independence to which the review relates. Therefore, in this paper, we often use terms ”**which/what”** is independent **"in respect to" "what"** with the term and concept of platform independence**.**

Independence can be achieved in respect to:

- Software platform or
- Hardware platform
- Within the software platform, we distinguish:
- Operating systems
- Implementation technologies

Platform can be observed as software or hardware platform. We make distinction between implementation technologies as a platform (like Java and .Net) and operating system as a platform. The implementation technology is executed on an operating system while the operating system is executed on some hardware platform. Software system is based on software architecture, and is implemented using concrete implementation technology and it executes on some operation system.

Based on the above-mentioned relationships between software architecture and software platform, the following basic classification of platform independence is derived:

- PI1 – Independence of software architecture in respect to the implementation technology
- In this category, we have particularly considered:
  - PI11 – Independence of MDA in respect to the implementation technology
  - PI12 – Independence of COA in respect to the implementation technology
  - PI13 – Independence of SOA in respect to the implementation technology
- PI2 – Independence of implementation technology in respect to operating system
- PI3 - Independence of operating system in respect to hardware platform

4. THE PLATFORM INDEPENDENCE REALIZATION MECHANISMS

In this section, we have described four key mechanisms used for platform independency:

1. Services-based mechanism
2. Component-based mechanism
3. Model-based mechanism
4. Virtual machine-based mechanism

These mechanisms are presented below on the Figure 2.

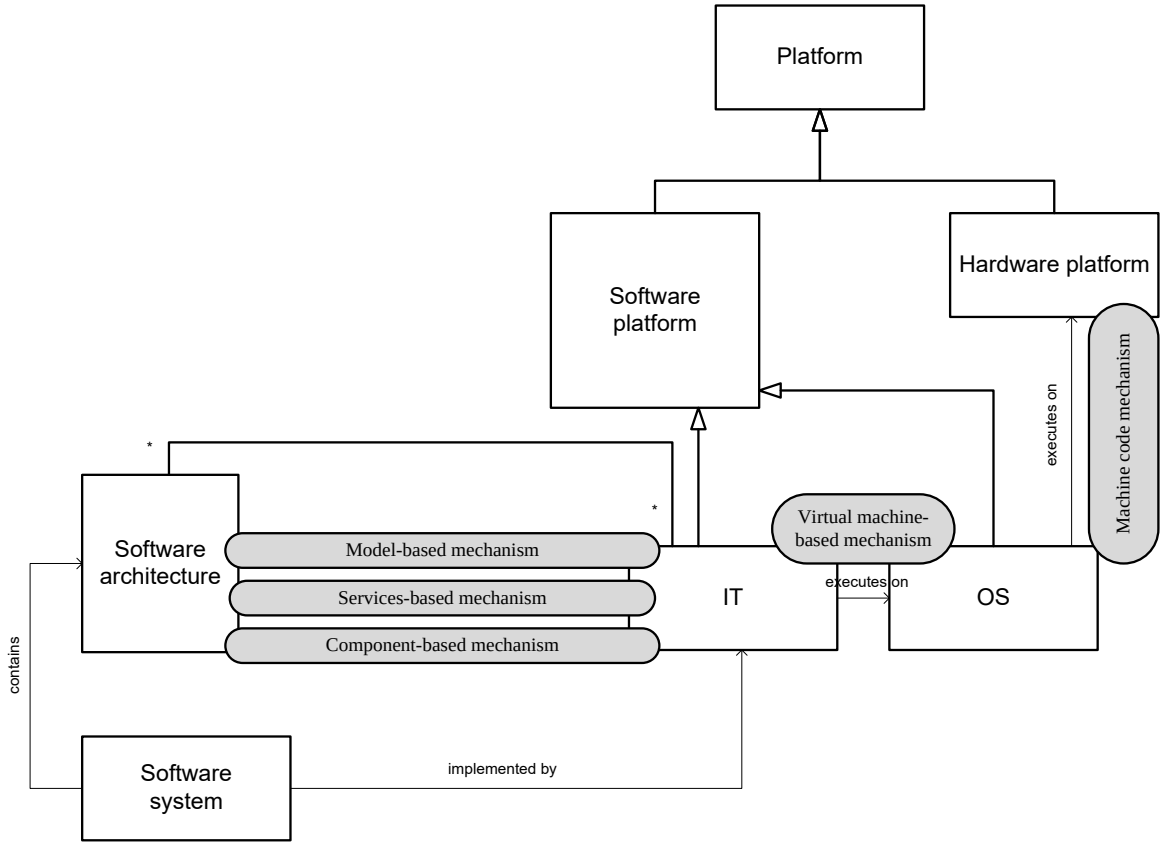

Figure 2: The relationships among software application, software architecture and platform

Model-based mechanism, services-based mechanism as well as **component-based mechanism** is used in achieving the independence of software architecture in respect to software platform.

Virtual machine-based mechanism is used in achieving the independence of the implementation technology in respect to the operating system on which the software system executes. The independence between software architecture and operating system is achieved indirectly (transitive relation) using this mechanism.

In addition to these four mechanisms that are covered in detail in the paper we have also identified the **mechanism of machine language** that enables operating system independence in respect to the hardware platform. In this paper, we have not specifically discussed about this mechanism. The independence between software architecture and hardware platform is achieved indirectly (transitive relation) using this mechanism.

Each of these mechanisms will be explained by the General Concept of the Independency (GCoI) model, which is presented on the Figure 3.

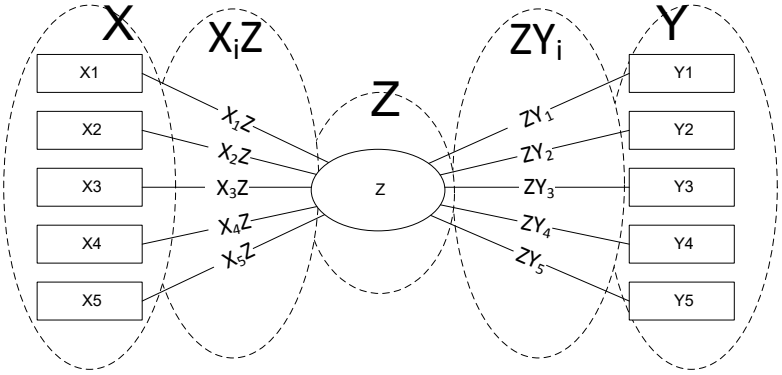

Figure 3. General Concept of the Independency (GCoI) model

The GCoI model represents the fundamental concept that lies behind the platform independence. The model consists of five elements (X, XZ, Z, ZY,Y). For each type of platform independence (PI1, PI2 and PI3), we have identified X and Y as elements between which we want to establish independence. X and Y can be observed as pair of software architecture (SOA, COA, MDA) and implementation technology (Java, .Net) in PI1, or as a pair of implementation technology (Java, .Net) and operating system (Windows2000,WindowsXP, Linux) in PI2 as well as pair of operating system (Windows2000, Linux) and hardware platform (x86 PC, AS/400) in PI3. In order to achieve independency between X and Y we have introduced independent component Z. Each element from sets X and Y should have different transformation to Z (XiZ or ZYj, i=1..n, j=1..m). If XiZ and ZYj exists, we can say that set X is independent to set Y.

Therefore, contrary to dependencies between every element from X (X1..Xn) to every element from Y (Y1 …Ym), these elements are only dependent on Z element. The GCoI model can be observed from two different aspects: as structure when we consider all the elements involved in achieving independency or as process of transformation between elements of set X and elements of set Y.

GCoI model can be applied to different mechanisms for achieving platform independence. If we consider Christopher Alexander's definition of patterns (Christopher et al. 1997), we can conclude that GCoI model obey this definition, and can be observed as a pattern. In software engineering, patterns are largely used for solving the problem of dependencies (coupling) between classes and objects. For creating maintainable software systems, it is important to reduce coupling as much as possible. The following part of this section describes mechanisms for achieving platform independence through GCoI model.

Services-based mechanism is used in achieving the independence of software components usually implemented as services (e.g. Web Services) using an implementation technology in respect to components implemented by another implementation technology.

Service-Orientated Computing (SOC) has become a main trend in software engineering that promotes the construction of applications based on the notion of services (Erl, 2005; Jonathan et al. 2008).

Web services provide a standard means of interoperating between different software applications, running on a variety of platforms and/or frameworks. A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-process format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

We have observed characteristics of web services, and ways of achieving platform independence in two different times:

1. Development time and
2. Runtime.

It is important to analyze both situations because in each of those we can notice different technologies, transformations, protocols, languages, but the same principle. This principle conform to the GCoI model.

According to General Concept of the Independency (GCoI) model, we have identified key elements of services-based mechanism, which are shown on the table below.

When we observe web service in development time, we can notice that both sides – provider agent and requestor agent can be developed using different technologies. Table 1 presents these elements.

Table 1. Services-based mechanism applied WS in development time

X - provider agent	**XZ**	**Z**	**ZY**	**Y-** requestor agent
Java	wsgen	WSDL	wsimport	Java
.Net	wsdl		wsdl	.Net
C++	wsdl2h		soapcpp2	C++
Delphi	delphi IDE		Delphi IDE wsdl importer	Delphi

Both sides need to have access to the WSDL document (Table 1, column Z). With access to the WSDL document both sides can generate programming code for components that will be used in runtime to establish communication (Table 1, column XZ – tools for generating components for provider agent, column ZY – tools for generating components for requestor agent). In this way, the XML based WSDL document stays the only component that gathers different technologies together and allows platform independent development of provider and requestor agents.

When we observe web services in runtime, platform independence is achieved using SOAP (Table 2).

Table 2. Services-based mechanism applied WS in runtime time

X - provider agent	**XZ**	**Z**	**ZY**	**Y-** requestor agent
Java	SOAP	SOAP Message	SOAP	Java
.Net				.Net
C++				C++
Delphi				Delphi

When the requestor agent calls some functionality of web service, the request is transformed into SOAP message (Table 2, column Z), and delivered to provider agent most commonly using HTTP protocol. The SOAP message is then transformed into platform specific call to provider component that will execute requested functionality, and optionally create the response and send it to requestor agent in the same manner.

Component-based mechanism is used in achieving the independence of software components usually implemented as services (e.g. CORBA) using an implementation technology in respect to components implemented by another implementation technology.

Development of software that is reliable, efficient and highly flexible, component-based software development can be employed for the complex software systems. (Jonathan et al. 2008) The Common Object Request Broker Architecture (CORBA) is a standard defined by the Object Management Group (OMG) that enables software components written in multiple computer languages and running on multiple computers to work together (i.e., it supports multiple platforms). It provides a platform-independent, language-independent architecture for writing distributed, object-oriented applications.

There are two perspectives (times) that, we have considered under the CORBA to explain component-based mechanism of the platform independency:

1. Development time and
2. Runtime.

From the development perspective, Interface Definition Language (IDL) is key concept that is used in component-based mechanism in order to achieve independence of a software system that was implemented through an implementation technology in respect to another implementation technology. According to General Concept of the Independency (GCoI) model, we have identified key elements:

- Different implementation technologies that conform to the X element on the GCoI model
- Different specification that enables transformation from IT to IDL interface (XZ elements on the GCoI model)
- IDL interface (Z elements on the GCoI model)
- Different compiler that generate code for different implementation technologies from IDL (ZY element on the GCoI model)
- Different implementation technologies that conform to the Y element on the GCoI model
- These elements are shown below on the Figure 4. and on the Table 3.

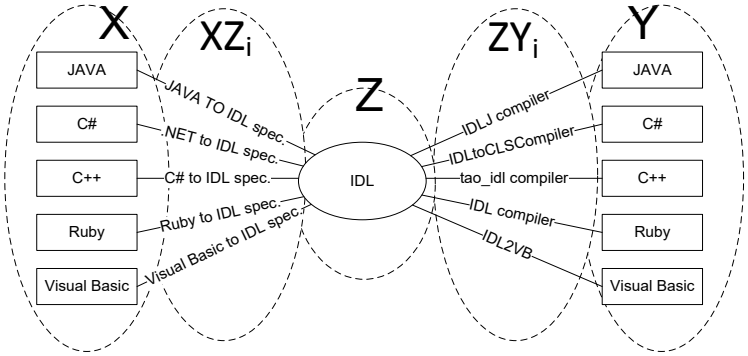

Figure 4: GCoI applied on the CORBA in development time

Table 3. GCoI applied on the CORBA in development time

X	XZ	Z	ZY	Y
Java	JAVA to IDL specification	IDL interface	IDLJ compiler	Java
C#	.NET to IDL specification	IDL interface	IDLtoCLS compiler	C#
C++	C++ to IDL specification	IDL interface	Tao_idl compiler	C++
Ruby	Ruby to IDL specification	IDL interface	IDL compiler	Ruby
Visual Basic	Ruby to IDL specification	IDL interface	IDL2VB	Visual Basic

From runtime perspective, client and server classes, generated by specific ORB programming language compiler, communicates among themselves using ORB vendor classes. ORBs communicate using IIOP protocol that enables software platform independence on both sides (client and server).Therefore the IIOP protocol is the key concept used in order to enable different software components to communicate among each other. According to General Concept of the Independency (GCoI) we have identified key elements:

- Different implementation technologies (Java, C#...) that conform to the X element on the GCoI model
- Different implementation of the CORBA standard (XZ elements on the GCoI model)
- IIOP CORBA that conform to the Z element on the GCoI model
- Different implementation of the CORBA standard (ZY elements on the GCoI model)
- Different implementation technologies (Java, C#...) that conform to the Y element on the GCoI model

These elements are shown on the Figure 5. and Table 4.

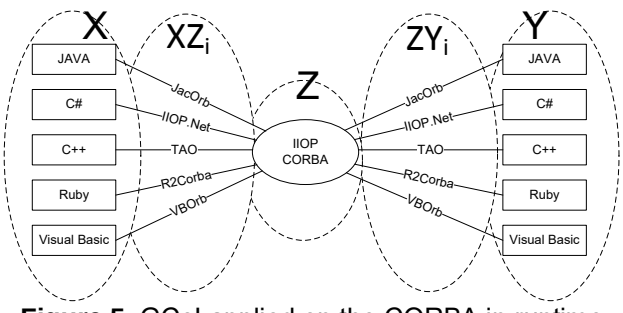

Figure 5. GCoI applied on the CORBA in runtime

Table 4. GCoI applied on the CORBA in runtime

X	XZ	Z	ZY	Y
Java	JacOrb	IIOP CORBA	JacOrb	Java
C#	IIOP.Net		IIOP.Net	C#
C++	TAO		TAO	C++
Ruby	R2Corba		R2Corba	Ruby
Visual Basic	VBOrb		VBOrb	Visual Basic

Model-based mechanism is used in achieving the independence of software architecture in respect to software platform. This mechanism is implemented under Model-Driven Architecture approach.

The concept of Model Driven Architecture (MDA) is published by OMG. It is based on creation of models and transformations between them. OMG describes different type of models and their relations but it does not specify how to create these models and which exact models and notations to use for their representation and how to transform them with one another. The top three models are created as graphical models while the last one as implementation code model.

Computation Independent Model (CIM) - CIM is a model that does not display details of IS construction but it specifies activities that are being processed in the IS. It represents business processes of the organization for which the IS will be developed. (Kardos, 2010)

Platform Independent Model (PIM) – PIM is a model, which describes IS, but hides details in usage of concrete technology. The PIM describes the behavior and the structure of the system. It does not specify operating system, programming language and hardware. PIM models are used to model the functionality and structure of the information system independently of the technological details of the platform, upon which it will be implemented.

Platform Specific Model (PSM) – PSM connects specification from PIM with details that specify what type of platform IS. will use. The PSM is responsible to specify the technical details to implement the PIM, e.g. the operating system, the programming language.

Implementation model (IM) – IM presents platform specific code. This model is usually generated from PSM but also it can be generated from PIM. It represents the deployable code that could be directly compiled and deployed without human interaction.

According to GCoI model, we have identified key elements the can be found in MDA. These elements are presented in Table 5.

Table 5 GCoI applied on the MDA

X - CIM	XZ - CIM-PIM	Z - PIM	ZY - PIM-PSM	Y - PSM
UML activity diagram	Query/View/ Transformation DSL	UML use case	UML Profile Query/View/Transformation DSL GPL	JavaEE
Business Process Model		UML activity diagram		
Data Flow Diagram		UML activity diagram		.Net
		UML use case diagram		
		UML sequence diagram		ORM
		UML class diagram		

PIM in MDA presents element used to achieve independence software architecture in respect to software platform. Different transformation from CIM to PIM can be written in DSL as well as transformation from PIM to CIM.

Virtual machine-based mechanism is used in achieving the independence of the software system, which was implemented through implementation technology in respect to the operating system on which the software system executes. The Fig.13 presents this mechanism.
This type of mechanism is slightly different from other types of mechanisms for achieving platform independence, because it is not directly related to a software system or software architecture, but indirectly through the implementation technology that is used for implementation of the software system. As mentioned earlier in this paper, implementation technologies and operating systems are different types of software platforms. Virtual machine-based mechanism enables independence of implementation technologies in respect to operating system. This mechanism is crucial mechanism because, in combination with other types of mechanisms, that enables independence for different software architectures, it enables independence related to different operating systems, and by that to different hardware platforms.

Virtual machines are used in a number of sub disciplines ranging from operating systems to programming languages and processor architectures. By freeing developers and users from traditional interface and resource constraints, VMs enhance software interoperability, system impregnability, and platform versatility. Despite their incredible complexity, computer systems exist and continue to evolve because they are designed as hierarchies with well-defined interfaces that separate levels of abstraction. The simplifying abstractions hide lower-level implementation details, thereby reducing the complexity of the design process.

Java and .NET are technologies and software platforms that both have the mechanism that realizes platform independence of software systems/applications in respect to a hardware platform and operating system. Java and .NET have a very similar approach to the implementation of the platform independence.

Java virtual machine is a virtual processor "a virtual CPU" that allows the same software to run on the multiple platforms, because software itself does not work directly on the operating system, but works through the Java virtual machine. Java applications can work on all platforms for which Java Virtual machine (JVM) exists.

Furthermore, the goal of Microsoft in the .NET development is to provide independence of .NET applications from the hardware platforms and operating systems. During compilation of .NET source code, compilers, instead of machine code produce common intermediate language (Common Intermediate Language) instructions. These instructions are translated into machine instructions or processor instructions on the machine where the application executes.

According to GCoI model, we have identified key elements:

- Different implementation technologies that conform to the X element on the GCoI model
- Different compilers that compile source code to bytecode or CLI (XZ elements on the GCoI model)
- Bytecode or CLI (Z elements on the GCoI model)
- Different interpreter which interpret compiled source code to concrete operating system (ZY element on the GCoI model)
- Different operating systems that conform to the Y element on the GCoI model

Table 6. presents the key elements of the GCoI model for Java and .Net technologies.

Table 6. Virtual machine-based mechanism applied on the Java and C#

X	XZ	Z	ZY	Y
Java	javac	Bytecode	JVM	Windows, Solaris, Linux
C#	csc	CLI	CLR	Windows 2000,Windows XP, Windows 7

The similar mechanism is identified for programming languages such as Jython, Scala, Boo and IronPython. Table 7. presents the key elements of the GCoI model applied to these programming languages.

Table 7. Virtual machine-based mechanism applied on the other programming languages

X	XZ	Z	ZY	Y
Jython	jythonc	Bytecode	JVM	Windows, Linux
Scala	scalac	Bytecode		Windows, Linux
IronPython	pyc	CLI	CLR	Windows, Linux
Boo	booc	CLI		Windows XP, Windows 7.

Table 8 shows the relationship between types of platform independence and mechanisms of its realization. Columns PI11 to PI3 indicate which type of platform independence is achieved, and rows R1 to R4 refer to a type of mechanisms used in realization.

Table 8. Relation between platform independence types and realization mechanisms

▼ *REALIZATION MECHANISMS* ▼	*PLATFORM INDEPENDENCE TYPES*				
	PI11	*PI12*	*PI13*	*PI2*	*PI3*
R1 (Services-based mechanism)			•		
R2 (Component-based mechanism)		•			
R3 (Model-based mechanism)			•		
R4(Virtual machines- based mechanism)	•				
R5(Machine language mechanism)					•

5. SUMMARY AND CONCLUSION

There are books and papers in literature related to Java's architectural support for platform independence (McGovern et al. 2003; Gong et al. 2003) as well as Web service platform-independent model (Szyperski, 1999; Alonso, 2004) and Corba platform independent model (Merle et al. 1997; Merle et al. 1996; Wang et

al. 2000). Although these papers describe platform-independent model, we could not found papers that analyze all these models of platform independence integrally.

There is no research in literature that analyzes the various mechanisms for achieving platform independence in this way, so we can say that this is the first study of this kind with the goal to not only analyze and compare the ways of achieving platform independence, but to define the general rule that enables it.

The key contribution of this paper is the introduction of the GCoI model, and identification and explanation of different mechanisms for the platform independency realization. The GCoI model represents the fundamental concept that lies behind the platform independence.

This model has multiple significances. While developing new platform or software architecture, engineers should be aware of all necessary elements that need to exist in order to achieve platform independency. In the era of cloud computing, and smart devices, the need for this kind of solutions and "adapter" mechanisms arises, and GCoI model represents the general principle for creating them.

Our practical experience, as university teachers, proved the value of this model when teaching relationships between any new software architecture and platform. When students know the elements of GCoI model, and the concept of platform independency, they can understand it much easier and faster.

REFERENCES

Alonso, G., Casati, F., Kuno, H. and Machiraju, V. (2004), *Web Services: Concepts, Architecture and Applications*. Springer

Bass, L., Clements, P., Kazman, R. (2003), *Software Architecture in Practice* (Second Edition). Addison-Wesley

Buschmann, F. (1996), Pattern-Oriented Software Architecture: A System of Patterns. *John Wiley & Sons*

Christopher, A., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., Angel, S. (1977) A Pattern Language. *Oxford University Press*, New York

Erl, T. (2005), Service-Oriented Architecture – Concepts, Technology, and Design. *Pearson Education, Inc.*

Gong, L., Ellison, G. and Dageforde, M. (2003) *Inside Java™ 2 Platform Security: Architecture, API Design, and Implementation*. Addison Wesley

Gorton, I. (2011), *Essential Software Architecture* (2nd Edition), Springer

Gorton, I. (2011), *Essential Software Architecture* (2nd Edition), Springer

Jonathan, L., Shang-Pin, M., Ying-Yan, L., Shin-Jie, L., Yao-Chiang, W. (2008), Dynamic Service Composition: a Discovery-Based Approach. *International Journal of Software Engineering and Knowledge Engineering*, 199-222

Kardos, M. (2010), Analytical method of CIM to PIM transformation in Model Driven Architecture (MDA). *Journal of Information and Organizational Sciences*; Vol 34, No 1.

Liu, Y., Liang, X., Xu, L., Staples M., and Zhu L. (2011), Composing enterprise mashup components and services using architecture integration patterns. *Journal of Systems and Software*, Elsevier, Vol. 84, No. 9, 1436-1446

McGovern, J., Tyagi, S., Stevens, M., Mathew, S. (2003), *Java Web Services Architecture.* Addison Wesley

Meghan, K., Model Driven Architecture Straight From The Masters, *The MDA Journal*

Merle, P., Gransart, C. and Geib. J.M. (1996) CorbaWeb: A Generic Object Navigator. *Proceedings of the Fifth International World-Wide Web Conference*

Merle, P., Gransart, C., Geib, J.M. (1997) Generic tools: a new way to use Corba. *In European Conference on Object-Oriented Programming*, *Workshop on CORBA: Implementation, Use, and Evaluation*

Object Management Group (2003), Ontology Definition Metamodel Request For Proposal OMG Document: ad/2003-03-40

Szyperski, C., *Component Software: Beyond Object-Oriented Programming*. Addison- Wesley, New York

Vitvar, T., Mocan, A., Kerrigan, M., Zaremba, M., Maciej, M., Moran, M., Cimpian, E. (2007). Semantically-enabled service oriented architecture: concepts, technology and application. *Service Oriented Computing and Applications*, *1*(2), 129-154. Springer

Wang, N., Schmidt, D. C. and Levine, D. (2000) Optimizing the CORBA Component Model for High-performance and Real-time Applications. *ACM/IFIP*

DEVELOPING ANDROID APPLICATION FOR LEARNING DATABASE DESIGN

Dr Saša D. Lazarević[1], Igor Stanišić[*2]
[1]Faculty of organizational sciences, University at Belgrade
[2]City Administration of the City of the Belgrade
*Corresponding author, e-mail: igor.p.stanisic@gmail.com

Abstract: *Period when mobile phones were just devices used for voice and text communication are far behind us. Thus we must to consider option to use them with different approach where they will be accepted as main source to find, discover and learn. We find basis of this idea in mobile applications which - versatile and acceptable, can bring focus to new way of studies which will then transform user's habits and give them alternative for learning. Respecting the results of previous research in the field of using mobile phones for the purpose of education, as well as habits of modern generation of students, we created the educational student application whose content complements the lectures and, if necessary, consultations. This application represents a solution for students of naturally humanities faculties which was, for the first time, in contact with software for databases. The application is based on the respective subjects which are responsible for the further understanding of study programs.*

Keywords*: Android, application, education, studies*

1. INTRODUCTION

While generation X use internet as product which slowly show in their lives creating enough space to learn and get new experience, in the next generation Y it become part of their life making them more informational educated. The period of the latest generation Z is in the progress, but we can predict that diversity in making new knowledge and experience will be bolder and cause can be internet itself – for those generations internet is product that just "exists" (Bogdanović, 2012.). From that point of view, from any new student's generation we can expect to be more and more in connection with internet and new technology, which basically lead to high end devices such as mobile phones. With that information on mind, academic society must find better respond in seek for knowledge for new generations – mobile phones can be used as tube (or channel) which will provide accurate information's to anyone on any place at any time (Milutinović, 2014.). So, if we percept application as tool for learning, we accept future of high education (as well as undergraduate or mid school learning). Of course, any educational app can't be accepting as one and only knowledge source. Instead of be self-enough, those digital learning tools must coexist with formal sources and that compound will provide unique, modern, faster, quality and more dynamic presentation of information's that represents subject of study (Bogdanovic, 2014.). Besides that, we have two additional benefits: first, classes wouldn't be crated only with instructor knowledge and creativity, and second – it will represent domestic educational system with high range of availability to modify content to meet new standards (Pasek K.H, 2015).

2. VERBAL MODEL

Logic that lay in root of the application represent practical product which will connect technical solutions with conventional information's sources. Thus, application must be simple and logically oriented for use, with main concept that will represent standards of Android operating system which users of the platform already known with integrated organization well known in books and educational literature which students use on every day basis: get correct answer fast and in clear way at the same time. Additional functionality will be created with option to take personal notes. Last but not least, application need to use less battery power (Chen, 2015, Flinn, 1999, Morley, 2015.). With all this in mind, we create map of the application where main content is away from user with maximum two clicks. Possibility to create notes is structure which is separated from others and user need just one click to use it (Figure 1). Please keep in mind that diagram shows few activities with the same name, but that is not mistake: content of fields is different.

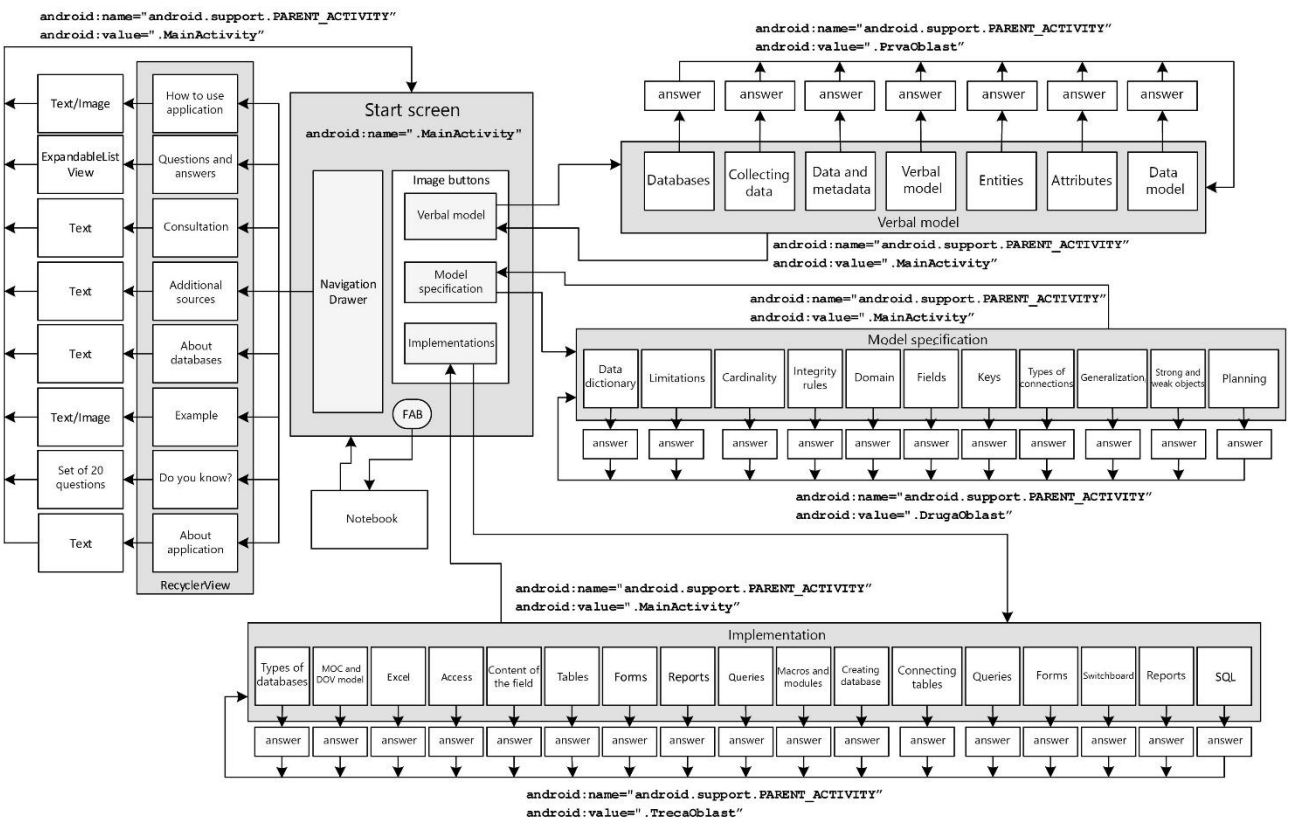

Figure 1: Diagram of the application based on verbal model

Start screen consolidate three application segments: main content which is split into three areas and which represent central part of the display, note taking part which open with click on the button at the lower right corner and additional menu position at the left top corner represented with “hamburger” icon (Figure 2).

Figure 2: example of the main application screen

At the top of any page lay unification App bar.

Main content has three areas:
1) verbal model – contains question and answers about actions which applied prior to planning and implementation, and explains what are entities, attributes, how and why the data are collected and more.

2) model specification – allows users to find explanations of all basic factors that are necessary to create a database such as aggregation, constraints, data dictionary...

3) implementation – the third area define the practical application of the previous steps, explaining database creating software as well as connections, query’s, report generation, etc. Also, students have practically described processes step by step.

Content doesn't need to be used by any fixed order. User can open, read and use part of application that is important for him in the moment. Keeping that in mind, simplicity and usability of the application is created in less than three clicks: from main category, user choose question which lead them to the answer and vice versa. So, first choose is category and the second one is the question. All answer contain text and (if there is appropriate) picture (Figure 3).

Figure 3: navigation through application. Main content is only two quests away

Part of the application which provide possibility to take notes lay behind circle Floating Button in bottom right corner of the main screen. Blank field provide user a space to create important information's which later he could copy or modify. Once entered, content stay available as note even after application is close or the phone is rebooted.

Finally, slide menu on the left side of the main application screen contain additional information's such as: how to use application, frequently asked questions, further resources, about databases, example, test and about application.

2.1 CHOOSE RIGHT ARCHITECTURE, TECHNOLOGY AND TOOLS

First step was to define right architecture, technology and tools which will be used during creating application with main purpose to satisfy concept standards. Keeping that in mind, core architecture was Linux kernel with Dalvik virtual machine (or Android runtime – ART) and program languages Extensible Markup Language (xml) and java (Ward, 2014.). Objective API's (Application program interface) in java language determine specific class which ask to implement accompaniment library for objects of that class type can be held by adequate methods. Lower API limit is 15, which means that any device with operating system 4.0.3 or newer can run the application. One of the special requests was Material design which natively run on systems version 5.0 and newer which set up the task to modify xml and java files to cover older Android versions (Annuzzi, 2014.). At the end, the tool which used was Android Studio with support and additional API 23 library's (Jackson, 2014.).

3. MODEL SPECIFICATION

First contact of students from humanity and social sciences with software used to create databases such as Excel, Access or MS SQL Server can be very complex and represent issue for normal learning and afterward using at work. Reason lay in specific logic of work and using programming languages and operations which are usually strange to those students that they didn't use earlier. Therefore, it is necessary for the purposes of understanding the basic definitions and architecture procedure, technology and applied tools, to prepare a unified solution in the form of applications that can provide answers in these areas represent them verbally and graphically.

3.1 OBJECTS-CONNECTION DIAGRAM AND THE DATA DICTIONARY

The application basically has a very simple structure, which is one characteristic that is set as a condition in its drafting – function before form. From the above we see that the home screen aggregation main categories, notebook and side menu (Figure 4).

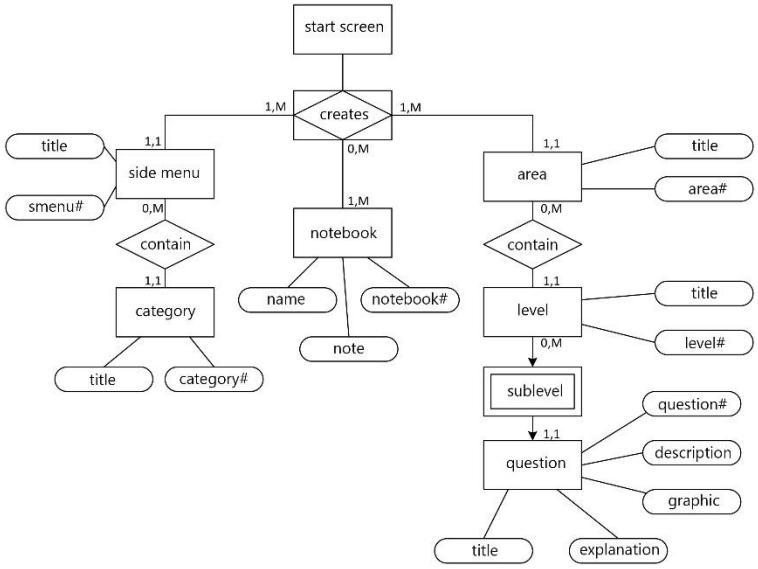

Figure 4: Diagram chart objects - connections for Android app

Start screen can contain one or more side menus, while a side menu can be found in only one home screen. Time this menu can be expanded if necessary by adding new content. Side menu does not have to but it can contain a number of different categories. On the other side, a category may belong to only one side menu. One initial screen doesn't need to contain but can have multiple notebooks and one notebook can be on one or more screens. Since the notebook represent additional functionality that is not a high priority in this case it's cardinality indicates that the same notebook can be accessed from other screens (such as the level of some areas) which isn't obligation. Finally, the area doesn't not need to have a single level, which practically makes possible to adapt the application to a new study program at any moment. Last one, each level (list of issues) should have area to which it belongs.

When we talk about the list of questions (layers), they must have at least one query (defined by entity "question"), and one question must belong to one level. Open condition of the connection between the entities "area" and "level" is defined with cardinality 0, M which giving us freedom of adapting content to other study programs and departments.

Diagram also characterized a simple data dictionary with a mere value-limits (Table 1). Answer, which represent the core of application can't be empty, so we're talking about a very elemental restriction imposed by the requirement that said the field can't be empty.

Table 1: data dictionary for Android application

			simple value limit
	attribute name	**attribute type**	**attribute category**
TABLE AREA			
attribute	NAME	CHAR (40)	IN (VERBAL MODEL, MODEL SPECIFICATION, IMPLEMENTATION)
TABLE LEVEL			
attribute	NAME	CHAR (40)	
TABLE QUESTION			
attribute	TITLE	CHAR (30)	not null
	EXPLANATION	CHAR (300)	not null
	GRAPHIC		
	DESCRIPTION	CHAR (1000)	not null
TABLE NOTEBOOK			
attribute	NAME	CHAR (40)	
	REMARK	CHAR (1000)	
TABLE SIDE MENU			
attribute	NAME	CHAR (40)	
TABLE CATEGORY			
attribute	NAME	CHAR (40)	not null

4. IMPLEMENTATION

Application does not become popular by chance – it is the result of the right decisions made at the right time. It is important to understand the role of performance, quality and robustness that must be respected in the broad market of different devices, considering that the same application can behave completely differently in the two models even if they are both on the same version of the operating system and with the same hardware specification. This is the result of platform fragmentation, as well as free modification of Android OS which is often used by device manufacturer that created them in all shapes and forms rarely updating the core version of the operating system. Also, other software corrections which can be applied on devices (such as manufacturers and operator's modes) may affect the stability and performance of other applications. Finally, it should be understood that the operating system is also the software such as the application that we want to create.

Creating an application create resources that need to be connect with the data sources. System construction (script) takes all sources (XML and java files) apply appropriate tools (for example, converts java classes in dex format) and then grouping them into a single compressed file, with APK extension. Gradle scripts are used in order for this process to be automated (Lee, 2012.). The most important part for developers is build.gradle (Module:app), which sets the lowest value of Android system on which the application can be installed, and to determinate and import the dependent libraries such as RecyclerView or FloatingActionButton.

4.1 THE LOGIC OF CREATED ENVIRONMENTS AND FORMING ELEMENTS

It has already been stated that the minimum requirement to install application is API version 15 or higher. This means that it is necessary to adjust the Material Design environment (which is native at KitKat, Lollipop and Marshmallow systems) to systems where they are not provided by default. Of course, functionality of the application isn't put behind design at any point during creation process. Rather, appearance and specific elements (such as the App bar, Floating Action Button, Recycler View) are used in a way to provide the content on the most functional way closer to user.

The first step in creating and customizing functionality of the new application environment was the adaptation of the Toolbar into an App bar (formerly the Action bar) in the way to hold additional functionality (Phillips, 2013.). In order to achieve a new look and functionality desired, it was necessary to couple the steps of:

1) it is necessary to prevent the system to display a predefined bar with selecting specific theme: basic theme of the application (which is located in styles.xml) need to be changed in the parent value which will invoke the theme without the Action bar:

<style name="AppTheme" parent="Theme.AppCompat.NoActionBar">

At the moment, the activity will show a window without Action bar.

2) define the xml file containing Toolbar: under resources> layout it is necessary to create a new XML file that we will call app_bar while Toolbar define as RootElement. After Android Studio (AS) creates the desired xml file, it will be marked as a Toolbar widget which we don't want. That is the reason why we will instead enter predefined: android.support.v7.widget.Toolbar. Another important step is the change of the content presentation, so instead of "match_parent," the android: layout_height should call "wrap_content".

3) in the layout xml file you need to add <include>: in xml file activity in which we want to show a new Toolbar you need to enter the following code:
<include
android: id = "@ + id / app_bar"
layout = "@ layout / app_bar "/>
This step must be repeated within each newly created activity because the AS will apply the theme that we selected in Step 1.

4) Toolbar initialization using findViewById within the Activity class and use setSupportActionBar () in the Toolbar: in the context of java class activity in which is projected a new Toolbar, first we define a variable private Toolbar toolbar (while respecting that it is android.support.v7.widget.Toolbar) and then in the method onCreate we bring the following:
toolbar=(Toolbar) findViewById(R.id.app_bar);
setSupportActionBar(toolbar); (This command requires that the system does not use the system defined toolbar)

When creating a variable, AS automatically creates a menu item under onOptionsItemSelected that appears in the right corner of the Toolbar in the form of three vertical points (menu). While we don't want this to show up in our application, we completely remove this method from the code.

5) adjustment of various properties of the Toolbar using the Toolbar object or through getSupportActionBar (): opening another activity that also initiated a new Toolbar, the title of the chapter will appear in the header (defined by string and AndroidManifest files) but the button which allows back to previous, parent (parent), activity will miss. Therefore, in the new java class activities under the onCreate method we need to add the following (just below setSupportActionBar (toolbar)):
getSupportActionBar().setDisplayShowHomeEnabled(true);
getSupportActionBar().setHomeButtonEnabled(true);
getSupportActionBar().setDisplayHomeAsUpEnabled(true);
Last method – setDisplayHomeAsUpEnabled (true) must have a boolean value of "true" to return the user to one level instead of at the beginning (start screen) of the application.

AppBar usually contains navigation icon ("hamburger" menu) in the top left corner, the name of the application and a filter option (if there is multiple hierarchical levels), action icons (for access to certain features of the application) and the menu icon in the upper right corner. We decide to look for more simple, cleaner look (Figure 5).

Figure 5: appearance of the native AppBar menu (left) and AppBar menu in our application (right)

4.2 LAYERS AND FUNCTIONS

Most part of user interface was created using ListView containers in RelativeLayout plan, while TableLayout was implemented in several activities. Specific solutions include Navigation Drawer fragment, Recycler View, List View Expandable and Floating Action Button.

For side navigation we chose Navigation Drawer Fragment (NDF) located on the left edge of the screen that will appear only on user request over the home screen (activities), and will represent additional informations. For NDF to be implemented, it is necessary to modify the XML and java file activity in which will be located. The first step represents use of DrawerLayout which contains two items (child) with the corresponding IDs: FrameLayout which hold the main content and NavigationDrawer, while the root XML file activity instead of LinearLayout became android.support.v4.widget.DrawerLayout. Xmlns attributes are transferred to DrawerLayout which then became holder of the LinearLayout who's actually FrameLayout with main content. Adding NavigationDrawer's is done by creating a new fragment (Java and XML files) which is initiates in the XML file of the main activity within DrawerLayout and below LinearLayout. The width of the fragment is 280dp while using android: layout_gravity = "start" we defined a fragment to be on the left side of the main screen until user action. We want to system recognize that fragment so in the last line we entered location of the created xml file. To use NDF, it is necessary to call methods within main activities Java class in order to bypass Toolbar, which is created under the Java class Navigation Drawer Fragment. It is necessary to apply the constructor ActionBarDrawerToggle that implements class DrawerLayout.DrawerListener. In other words, in this way the NDF, Toolbar and layout activities will be linked into one which, in practice, mean that when user select an icon in the toolbar it will open NDF (Lee, 2013.).

If we run the application at this point, we will have transparent fragment covering the main activity while in the Toolbar will be "hamburger" menu. Therefore, it is necessary to adjust the value of the background color in xml file fragment, while the java class NavigationDrawerFragement must have method: mDrawerToggle.syncState ().

With this step we have created the NDF, which is empty and respond to the user's reference. In the next step we need to input content.

4.3 RECYCLER VIEW (RV)

Recycler View is not a substitute for the List View. It is far more advanced solution which use Layout Manager and allows us to view content at a much more dynamic way than the List View can provide (Jackson, 2014.). The architecture shown in Figure 6 explains the work of RV. Data Model is a menu item. Adapter draws each menu item and trying to display them with a View holder who is the xml layout for each menu item, which is incorporated into the java code. View holder decides whether the item will be

permanently displayed or not. Recycler View will use View holder to display each menu item. On the other hand, the Layout Manager determines how the content will be displayed (as a list of fixed width, in parallel columns or as a group of cards in different sizes). Item Animator is responsible, as its name suggests, to animate Recycler View such as removing items from a list, move them in the order, etc. Item Decorator allows items to be grouped into different sections according to their content. RV represents a flexible solution for viewing large amounts of data on a limited area of the screen. Since the creation of content composed of images and text represent complex and demanding task for application (each line requires the creation of an independent xml file that must be inserted in the code via the Layout inflator as well as finding items by using the TextView, ImageView and the related controls findViewById) calling (inflate) the item only when recycled and which is already found represents a significant beneficial solution. The point is to avoid use of the View Holder object or method findViewById every time you want to display a longer list of items on display but

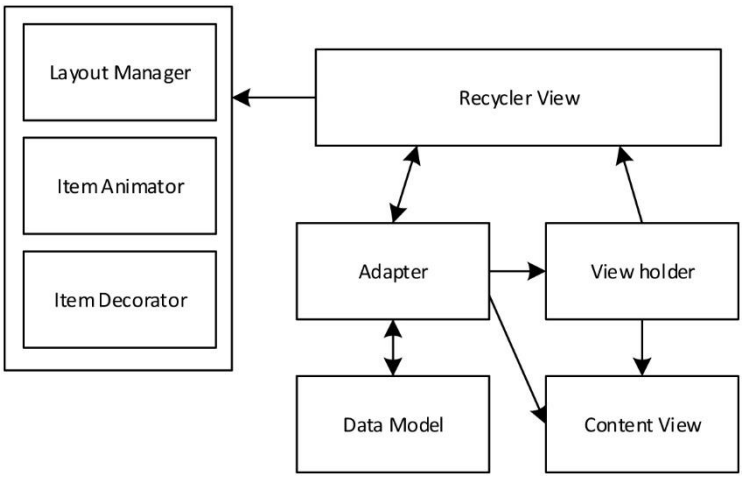

Figure 6: construction of the Recycler View

rather to allow the first item from the list to be cashed and appears only when it is needed. In fact, the limited area of the screen that is able to show, for example 5 items, when we move through the menu to display the paragraph 6, paragraph 1 shall be removed from the list. It will then move to recycler status and wait to be invited again.

The first step of the implementation is compiling a support library in Gradle script for recyclerview v7 widget while next represents its implementation in NDF xml file. It is important to enter the code below Linear (or Relative) Layout in which is placed an image that is part of the NDF. When rendering in Android Studio, review will not show any content. The reason for this is that unlike the List View, content must be initialized in Java code in the Layout Manager since it is not fixed. Before that, it is necessary in NDF java class to define Recycler View which is achieved by the following code:

recyclerView= (RecyclerView) layout.findViewById(R.id.drawer_list);

In this code, R.id.drawer_list represent id which we assigned to RV in xml file.

To populate the menu list, it is necessary to create a new java class (which we'll call Information) in which will be defined integer for the icon, and string value for the name. This data will provide adapter that represents a new java class that extends RecyclerView.Adapter which owns argument <VH> or ViewHolder which means that the latter is expected to have another class that extends a ViewHolder to represent argument. View Holder is there to describe the appearance of the item and its place in the RV and that the source, once found, automatically draws and displays in the required order. Therefore, in the construction we use parent class ViewGroup and viewType. In order for content to be developed, it is necessary to create a new xml file which will represent the appearance of a row, whose content will define the ImageView and TextView. Data from the newly created file are referenced in the java class adapters to allow the ViewHolder data calls and shows itself without the need for constant creation. Contents will be displayed by entering a few lines of code and linking with the previously created files. First you must enter the following classes in the java file adapter:

List<Information> data = Collections.emptyList();

and then in onBindViewHolder:

Information current = data.get(position);

```
holder.title.setText(current.title);
holder.icon.setImageResource(current.iconId);
```

Displaying the contents is calls by the NDF java file, creating methods:

```
public static List<Information> getData(){
List<Information> data=new ArrayList<>();
int[]icons={R.drawable.ic_lightbulb_grey600_24dp};
String[]titles={"Kako koristiti aplikaciju"};
for (int i=0; i<titles.length && i<icons.length;i++)
{
Information current=new Information();
current.iconId=icons[i];
current.title=titles[i];
data.add(current);
}
return data;}
```

Adapter initialization that displays content is created under onCreateView.

We want to connect content of RV with new activities which referred to, thus in the adapter Java class we need to create OnClickListener and in the NDF Java class method itemClicked. Because the Java class adapter has very little code, it makes it very flexible and opens up the possibility that the same adapter be used for other RV if they call the same information.

By implementing Recycler View in Navigation Drawer Fragment and creating connections with external activities that carry content, we finally fill the side menu and make it available to the user.

4.4 FLOATING ACTION BUTTON (FAB)

Floating key of activities represent the specific solution that characterizes Material environment (Jackson, 2014.). As NDF, this element is not recommended for use if there is no meaningful purpose that is usually reflected in the functionality that need to be quickly available - shortcuts to applications segments, arranging and writing messages, custom content (for copying and pasting content) etc. Therefore, we created FAB at the bottom right corner that leads directly to a notebook activity which characterized up to 100 lines of text input with up to 20 lines to display at the time (as defined in the value of its XML files). All the entered text is stored in the notebook until delete, which means that leaving the application or restart the device does not affect the stored text, which is achieved by entering the created content in SharedPreferences of corresponding java file. Entered text can be marked, copied, cropped and pasted, but because we didn’t set any specific requirement when installing, it can’t be send directly from the application.

Creating of the FAB can be do manually or using existing library. In our case, we have created a library using existing sources which is why we are in the Gradle script called and compile a dependent library CircularFloatingActionMenu: 1.0.2. Within the java class activity in which the FAB will occur, it must be initializing under the method onCreate and determine the icon that will appear (located in the drawable folder). Button background also is also called from drawable folder and represent an XML file that defines the layout when the key is pressed and in standby mode (lighter and darker background). Since FAB is removed from the screen when the notebook activity is open, this step is not necessary since there is no visual effect. Newly formed FAB must be connected with corresponding XML files, which is why necessary to implement OnClickListener after which it calls creation of tag icon.setTag("fab"); icon.setOnClickListener (this); after which in the onClick method needed to initialize event that will trigger the activity:

```
public void onClick(View v) {
if (v.getTag().equals("fab")){
startActivity(new Intent(this, Beleznica.class));}}
```

When a user clicks on FB on the home screen it will open activity notebooks. User will be return to the home page, which is the parent of this activity, after he done editing.

5. CONCLUSION

This paper presents a simple application for education of students and others who want to learn more about the possibilities of creating a database on their smartphones. Respecting the results of previous research of using mobile phones for the purpose of education and habits of modern generation of students, created the

content that complements the lectures and, where appropriate, consultations. The application is based on the respective subjects who are responsible for the further understanding of the matter that students learn. To avoid any compromise between content and form, the application is designed, created and implemented by respecting and implementing of the Google standardization of Material environment that will make users of newer mobile devices to use it in friendly, easy environment. Most of the time was spent on special modification of code and application structure. The development of application core took about four weeks, gathering materials and implementation additional fifteen days while the extra week spent for removing bugs and code optimization.

REFERENCES:

AMIDuOS, http://www.amiduos.com/, (31.08.2015.)

Annuzzi Jr. J. et al. (2014.) Advanced Android Application Development, Addison-Wesley Professional

Bogdanovic Z et al. (2014.) Evaluation of mobile assessment in a learning management system, British Journal of Educational Technology, vol 45, Issue 2, 231 – 244.

Bogdanović Z., Zrakić M. D et al. (2012.) Providing Adaptivity in Moodle LMS Courses

Chen X. et al. (2015.) Smartphone Background Activities in the Wild: Origin, Energy Drain, and Optimization, Purdue University, Mobile Enerlytics, Intel Corporation

Flinn J., Satyanarayanan M. (1999.) Energy-aware adaptation for mobile applications, 17th ACM Symposium on Operating Systems Principles (SO SP '99), Published as Operating Systems Review, 34(5):48–63

Jackson W (2014.) Android Apps for Absolute Beginners, Apress

Jackson W. (2014.) Pro Android UI, Apress

Lee W.M. (2012.) Beginning Android 4 Application Development, Wrox

Lee W.M. (2013.) Android Application Development Cookbook: 93 Recipes for Building Winning Apps, Wrox

Milutinović M et al (2014.) Ontology-Based Multimodal Language Learning, High Performance and Cloud Computing in Scientific Research and Education, Hershey, PA: IGI Global, DOI:10.4018/978-1-4666-5784-7.ch008, 195-212.

Morley B. D. et al. (2015.) "Dynamic battery saver for a mobile device", United States Patent, US 8,958,854 B1, Feb. 17

Pasek K.H et al. (2015) Putting Education in "Educational" Apps: Lessons from the Science of Learning", Psychological Science in the Public Interest, 2015, Vol. 16(1) 3–34

Phillips B. (2013.) Android Programming: The Big Nerd Ranch Guide, Big Nerd Ranch Guides

Ward B. (2014.) How Linux Works: What Every Superuser Should Know, No Starch Press

COMPARATIVE ANALYSIS OF UML MODELING TOOLS WITH FOCUS ON BUSINESS LOGIC SPECIFICATION

Vojislav S. Stanojević*[1], Saša D. Lazarević[1], Milić Ž.Miloš[1]
Faculty of Organizational Sciences,
*Corresponding author, e-mail: vojkans@fon.bg.ac.rs

Abstract: *Growing need for software products and reduced time for software development are crucial for success of a software project. Therefore, there are lot of approaches for software development in order to reduce time to market. One of the most popular is Model Driven Development. Main goals are to narrow the gap between all stakeholders in software development process and to decrease time from requirements specification to final version of software. There are lot of tools that supports UML modelling and MDD approach in order to shorten software development process. Using case study we will analyse three selected tools (Enterprise Architect, Visual Paradigm and Papyrus) in order to determine which one is best suited for MDD. For that purpose we will determine criteria for comparison and give a summary.*

Keywords*: MDD, business logic, UML, models, code generation, UML tools*

1. INTRODUCTION

If we look at economic development in recent years, even decades, the obvious conclusion is that the information systems and technologies are one of the most dynamic industries, where a lot of resources are invested. In such an environment, continuous development and progress are necessary.

In step with the progress of technologies, market requirements have become more sophisticated. The critical factor is response time needed to meet those requirements. High-quality of system architecture assures achieving this. The software architecture must support the complex requirements, frequent changes and quick response to those changes.

The aim of this research is reducing development time, and more importantly reduce response time to the frequent changes required by the client. In addition, it is preferable to avoid repetitive tasks and processes, and make them to be executed automatically. Model driven development is approach that reduces response time, complexity of requriements, business processes modellling and implementation at the end. Approach that will be used in this research is to use model as basic component and set of business rules defined for that model.This reserach should give an answer regarding possibilites of automatic code generation for elements of business logic using specific tools. For this purpose we will conduct case study survey analysing tools supporting UML models and code generation. For this purpose we selected Enterpise Architect, Visual Paradigm and Papyrus (plug-in for Eclipse) .
We will first give a short introduction to MDD, UML and OCL. Afterwards we will determine place and role of business logic in a software system and then present results from tool analysis and then give final conclusion.

2. MODEL DRIVEN DEVELOPMENT - MDD

Start of a software project can be very difficult. The path from requirements elicitation to software product is long and there is a problem with requirements validation. Often mistakes made in requirements gathering and specification are identified in late stages of software development which can have great impact on software success. One of approaches that tried to narrow the gap is Model Driven Development (MDD). Main idea in MDD is that there is no model which will be thrown away. One of key features is model transformation. This is very important because there are lot of interested parties which has different levels of technical knowledge. For instance on one side there are persons like sponsors and domain experts, with poor technical knowledge, and on the other side there are software engineers (architects, software, testers...). All of them are interested in software development from their point of view. Therefore there is a need for different type of models, but essential thing is to keep those models in tune with each other. This is not always easy but there are lot of software tools which offer help in model synchronization and their use throughout software development. Language which is widely used for purpose of software modelling is UML. Very important concept in MDD is model to model transformation. Final model is usually programme code and that is most important concept for software development.

3. UML LANGUAGE

Unified Modelling Language (UML) is language and a standard for software modelling and design. Besides structural and behaviour models UML defines models for business process modelling [5]. It is widely used in object-oriented software development methods. There different models that are used in different development phases. The main idea behind all models is to keep them simple as possible in order to make them understandable for large variety of stakeholders.

UML advantages:

- Is a standard for software development
- There is a lot of tools that supports UML models
- Shortens software development
- UML models overcame software engineers problems (Schmidt, 2006)

From our point of view most interesting models for the paper are:

- Structural models: Class diagram
- Behavioural models: Use case diagrams, State diagrams, Sequence diagrams

UML supporting tools enable model integration and model transformation, which is of big interest for this paper. For purpose of this paper it is very important to mention Object Constraint Language (OCL) and it use and integration with UML models.

4. OBJECT CONSTRAINT LANGUAGE - OCL

Object Constraint Language, widely known as OCL, is semi-formal language for describing constraints of object-oriented models (F. Barbier et al., 2001). It is very important addition to UML language and it can be considered as integral part of UML. Formal languages require big knowledge of math, which business analyst and modellers usually do not have. OCL is less formal and it is easier to write and read. It is not directly executable.

```
context Person::getCurrentSpouse():
   Person
pre:   isMarried = true
body:  mariages->select
       ( m | m.ended = false ).spouse
```

5. BUSINESS LOGIC AS PART OF SOFTWARE SYSTEM

The focus of our work is business database applications. Most of software systems of this kind have multi-tier architecture and most common one is three-tier software architecture. Larman identifies these tiers as **User interface**, **application logic** and **data storage** (Larman, 2004).

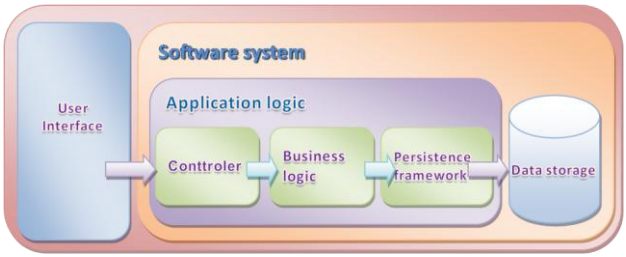

Figure 1: Three tier software system architecture

Furthermore, application logic has three logical parts: **Application logic controller**, **Business logic** (Domain objects, Services) and **persistence framework**.

In his book Vliet describes that from users point of view user interface represents whole system (H.V.Vliet, 2008). Research showed (B. Myers and M. Rosson, 1992) that 48-51% of time needed for software development goes to user interface. We have already written about models and tools, which we proposed for user interface generation in (Antovic et al. 2012). Business logic development takes around 15% and we think that this is not in accordance with significance that this tier has in software system. This was a sort of alarm because we think that business logic development does not get as much attention as it should.

We will define UML models of great interest for business logic modeling. In order to do that will will look for models used in different stages of software development.

5.1 Business logic in requirements phase

This phase of software development is marked as crucial for success of software project. This paper focuses on use case technique. There are lot of use case definitions and all of them point out interaction between actor and the system (Cockburn, 2000; Vlajic, 2011; R. Schach, 2010; Adolph, 2001; Jacobson et al.1992).

From a business logic perspective, it is very important to consider an action (transaction) types. *Ivar Jacobson* mark off four types of transactions (Jacobson et al.1992). Only one relates to a main actor and Jacobson defines it as User request action. Three remaining actions are relates to the system: System validates request and the data, system changes its internal state and the last one refers to response showed to the actor.

Vlajic gives a little different list of actions (Vlajic, 2011) :

Actor actions:

- Actor prepares input for system operation. (**APISO**)
- Actor calls system to execute system operation. (**ACSO**)
- Actor executes non-system operation. (**ANSO**)

System actions:

- System executes system operation. **(SO)**
- System shows result of system operation execution. **(OutA)**

5.2 Business logic in analysis phase

During the analysis phase, one should describe **logical structure** and **behaviour** of a software system. System sequence diagram and system operation contracts describe behaviour. Conceptual and relational model defines structure of a software system.

The use case specification is a starting point for analysis phase. System sequence diagram is made according to use case, and only two types of actions are shown ACSO and OutA. In order to create

these diagrams one must identify system operations signature. While APISO action describes name and parameters of a system operation, the IA action refers to return value and messages that has to be show to the actor.

Preconditions are conditions that have to be met in order to execute SO. They are highly related to constraints of a domain and relational model. On the other hand **post conditions** are related to conditions expressed through state of a domain object after successful SO execution

5.3 Business logic in design phase

In this stage of software development detailed architecture of software system is given. In the introduction of this paper, we wrote that the area of our research is software systems divided in three logical parts. In this phase of a software development one must take implementation technologies in to the account.

This phase of software development must give answers to questions about system operation execution. For this purpose detailed sequence diagrams for every system operation is made in this phase. In contrast, we believe that it is important to make some kind of technology independent specification. This will provide an open space of possible implementations.

6. UML MODELING TOOLS ANALYSIS USING CASE STUDY

In this section we will give an overview of tools which are marked as important. Chosen tolls are Enterprise Architect, Visual Paradigm and Papyrus plug-in for Eclipse IDE.

We will conduct a survey that will give an answer how well the tools support business logic specification and it transformation to programme code. We marked models that are important for a business logic modelling. In addition to that we should add a decision table which is related to business rule specification. Will examine how decision tables are integrated in code generation process.

During the research we have marked criteria important for analysed tools. The criteria are as follows:

- Visual view and drawing diagram utility
- Complex business rule definition support
- OCL support
- Model - code synchronization

Next we will give an overview of tools analysis. We will compare their features related to the criteria.

We analysed the tools using several use cases. Next use case model diagram created in Visual Paradigm shows all use cases used in tool analysis (figure 2).

Firstly, authors will like to point out glossary option in Visual Paradigm. One has option to define terms and even synonym for it, which can be very useful to end users. Next figure (figure 3) shows terms which are often used in case study.

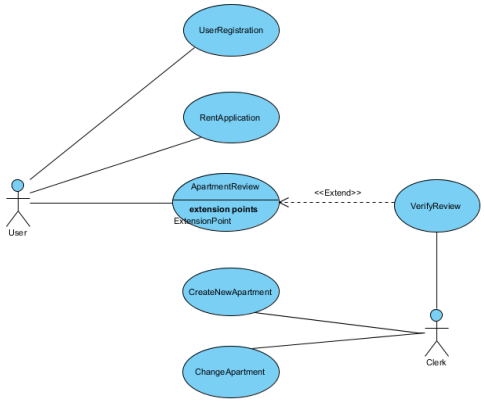

Figure 2: Use case modl diagram - Visual Paradigm

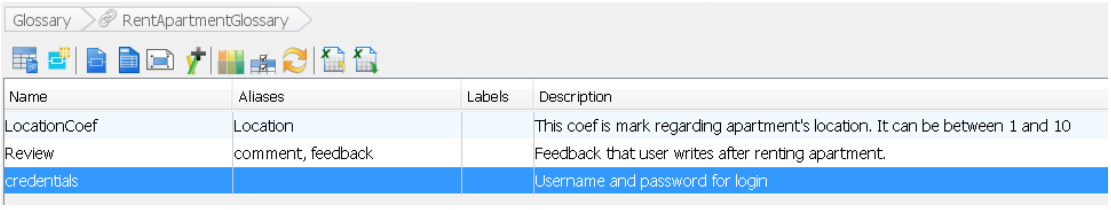

Figure 3: Term definition in Visual Paradigm glossary

The terms from the Glossary are automatically recognized in all UML diagrams. Figure 3 shows Use case definition where are terms recognized. One can see it as underlined word. This is very useful feature. Enterprise architect also enables Use case specification.

Figure 4: Use Case in Visual Paradigm

Diagram that is most often used is class diagram. All of the tools enable visual class diagram modelling. There is slight difference between the tools. It is mostly related to model constraints definition. Figure 5 presents class diagram designed in the tools. It important to address that Papyrus tool enable profiles, so one can choose its own profile instead of default one (for instance JAVA). This will enable JAVA type support for attribute definition and method signature as well.

As one can see models are almost the same and difference is present in OCL constraint support.

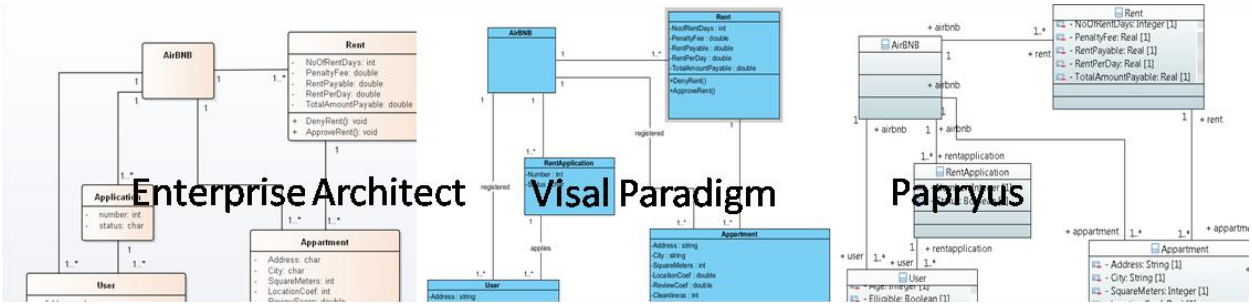

Figure 5. Class diagram in all tools

Papyrus tool has the best OCL integration. It is present in Enterprise Architect as well but with poor integration. Visual Paradigm does not have support for OCL.

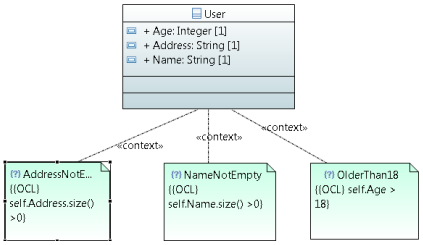

Figure 6: OCL constraints - Papyrus

When it comes to business rule specification it is best supported by Enterprise Architect. It is based on decision tables and in addition it enables code generation. At the first stage user define textual specification for business rules and afterwards one should define rules in specially designed decision table. On the other side Visual Paradigm offers same decision table functionality but authors did not find any code generation feature for it.

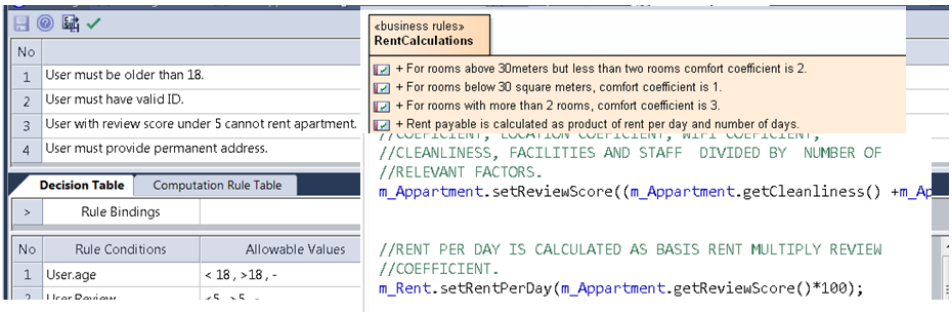

Figure 7. EA decission tables support with generated code

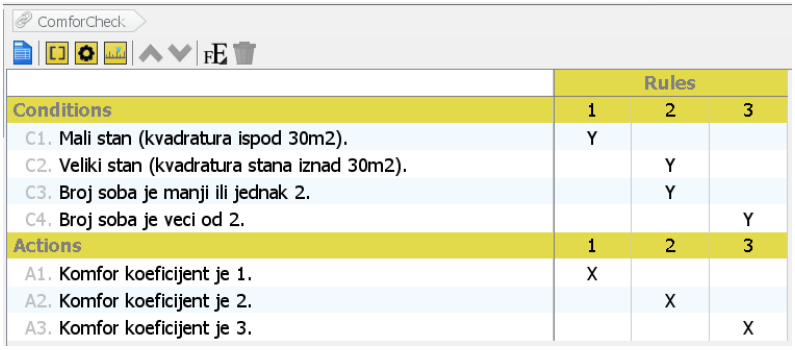

Figure 8: Visual Paradigm decision table

All tools have support for sequence diagram. Visual Paradigm enables transformation from Use Case specification to system sequence and activity diagram. It is very strange that Visual Paradigm does not support transformation from sequence diagram to programme code. It offers only reverse engineering option from code. Other tools supports code generation from sequence diagrams.

Very important feature is model-code synchronization. All tools in certain degree have model-code transformation but it is hard to expect that all features are covered, especially business rule and decision table support. For instance changes detected in Domain class code is reflected in Class diagram in all tools. All the tools support synchronization from programme code to sequence diagram.

We will give short summary for every chosen criterion.

Visual view and drawing diagram utility:

Enterprise Architect *enables modelling of every aspect of software development. It has good performance even when loading big models. Its environment is user friendly models are simply and easy to maintain.* ***Visual Paradigm*** *has very large community which is pushing forward tool development. It has good support for UML diagrams.* ***Papyrus*** *has well designed environment but it is not very easy to use. It does not have automatic align functionality so models can be difficult to read.*

Complex business rules definition:

Enterprise architect is absolute winner by this criterion. It has very well business rule definition system and good code generation utility. **Visual Paradigm** has very poor support for business rule specification. **Papyrus** does not have decision table support as two already mentioned tools but it has best OCL support.

Code generation options:

Enterprise Architect has several models to code transformations. It enables transformation to target programming platform from Class diagram, decision tables and sequence diagram. Visual paradigm

has very poor support for code generation. Only this tool does not support code generation from sequence diagram. Papyrus has most sophisticated support for code generation. It very easy to generate programme code skeleton for chosen target platform. It is also possible to add OCL constraints to UML models and include it in transformation process.

Final results are presented in table 1.

Table 1. Summary results

Criteria	Enterprise Architect	Visual Paradigm	Papyrus
Visual view and drawing diagram utility	4	**5**	3
Complex business rules definition	**5**	3	3
Code generation options	4	3	5
OCL support	2	1	**5**
Model - code synchronization	**5**	**5**	4

7. CONCLUSION

Growing need for software products and reduced time for software development are crucial for success of a software project. Therefore, there are lot of approaches for software development in order to reduce time to market. One of the most popular is Model Driven Development.

From one side MDD narrows the gap between all stakeholders in software development process and from the other side it decrease time from requirements specification to final version of software. Market offers lot of tools that supports UML modelling and MDD approach. For the authors most interesting tools that supports UML modelling are Enterprise Architect, Visual Paradigm and Papyrus.

Firstly we gave introduction to Model Driven Development approach, followed by UML and OCL. Afterwards we have presented place and role of business logic in a software system and then present results from tool analysis and then give final conclusion.

This research gives an answer regarding possibilities of automatic code generation for elements of business logic using specific tools. Using case study survey we have analysed tools supporting UML models and code generation (Enterprise Architect, Visual Paradigm and Papyrus).
We made a conclusion that all tools supports visual UML modelling but the extent of business rule specification and related code generation varies. All in all we made conclusion that all tools have some advantages and disadvantages. Visual efects are best in Visual Paradigm and Enterprise Architect are almost at the same level. When it comes to code generation papyrus is best solution.
Authors will give a slight advantage to **Enterprise Architect** tool.

8. REFERENCES

Douglas C. Schmidt, *Model-Driven Engineering*. *IEEE Computer*, 39(2), February 2006.

F. Barbier, B. Henderson-Sellers, A. L. Opdahl, and M. Gogolla, *Unified Modeling Language: Systems Analysis, Design and Development Issues, 2001*

Craig Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development,* Prentice Hall PTR Upper Saddle River, NJ, USA, ISBN:0131489062, (3rd Edition), 2004

H. V. Vliet, *Software engineering: principles and practice*, John Wiley & Sons Ltd, 3rd edition, Chichester, West Sussex, England, 2008.

B. Myers, M. Rosson, *Survey on user interface programming,* ACM: Human Factors in Computing Systems, Proceedings SIGHI, 1992

Ilija Antovic, Siniša Vlajic, Milos Milić, Dušan Savić, Vojislav Stanojević**,** *Model and software tool for automatic generation of user interface based on use case and data mode*, IET SOFTWARE, (2012), vol. 6 br. 6, str. 559-573

A. Cockburn, *Writing Effective Use Cases*, Addison Wesley Longman Publishing Co. Inc, Boston, 2000

S. Vlajić, *Projektovanje programa (skripta)*, Dr Siniša Vlajić, Beograd, 2011

Stephen R. Schach, *Object-Oriented and Classical Software engineering*, Eight edition, William C Brown Pub**,** 2010

Steve Adolph, Paul Bramble, *Patterns of Effective Use Cases*, Addison -Wesely, ISBN 0-201-72184-8, 2001

I. Jacobson, M., Christerson, P. Johnsson, and G. Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley,Wokingham, England, 1992.

A POSSIBLE APPROACHTOAUTOMATING THE DESIGN OF NoSQL DOCUMENT-ORIENTED DATABASES

Dejan Stojimirović*[1], Siniša Nešković[1], Nina Turajlić[1]
[1] Faculty of organizational sciences, University of Belgrade
*Corresponding author, e-mail: dejan.stojimirovic@fon.bg.ac.rs

Abstract:*Even though NoSQL databases are being used more and more for the development of web applications, there is no systematical approach or methodology for their development. A methodological approachtodocument-oriented NoSQL databases design is laid out. In the presented approach semantically richExtended Entity-Relationship modelsare used as conceptual data models,which are then transformed into concrete document-oriented NoSQL data models through the application of a set of rules. The paper suggests a possible approach to automating this transformation which is in accordance leading approach to software development today – Model-Driven Development (MDD).*

Keywords*:Extended Entity-Relationship model, document-oriented NoSQL databases, database design, transformation rules,automated transformation, model driven development*

1. INTRODUCTION

Nowadays so-called NoSQL (*Not only SQL*) databases are being used more and more for the development of web applications. The popularity of these databases can be attributed to the fact that, on the one hand, they are geared towards rapid and easy manipulation of large volumes of data, while, on the other hand, most existing NoSQL database management systems are open-source, thus the costs of developing web applications when using such systems are much lower. An additional advantage of NoSQL databases, over traditional relational (SQL) databases, is that the data is stored in a far more flexible manner given that they do not presume the existence of a database schema, or more precisely put, they do not require a rigidpre-defined data structure.

Even though a schema structure is not required, there is still a need for knowing how the data is structured in order to be able to manipulate it in the application.In other words, it is necessary to map the data thatis to be stored in the database onto the concepts which are available in the chosen NoSQL database type (e.g. collections, tables, documents, key-value pairs, etc.). The design phase, as is the case with traditional relational databases, should result in a database which enables easy and efficient manipulation of the stored data. Even though NoSQL databases have been in use for a number of years, no precise methodological approach to designing such databases has, thus far, been put forth. Thus, the design of these databases is usually based on the general recommendations of individual NoSQL database vendors.

Apossible methodological approach to designing NoSQLdocument-oriented databasesis presented (*Stojimirović & al, 2015*). It follows the traditional phases of database design (*Simsion& Witt, 2004*): conceptual design (i.e. defining a technology-independent specification of the data that is to be stored), logical design (i.e. translating the conceptual model into a model defined in terms of the structures of a DBMS) and physical design (i.e. the specification of the physical storage, access mechanisms, performance optimization, etc.). Extended Entity-Relationship (EER) models are used as conceptual data models, which are then transformed into concrete document-oriented data models through the application of a set of rules.

However, the manual coding of CRUD (Create, Read, Update and Delete) operations for complex data structures can be very time-consuming and is often prone to errors (even more so in view of the absence of a rigid data structure schema). This paper, expounds on the work presented in (*Stojimirović& al, 2015*)and suggests a possible approach to NoSQL document-oriented database design that is in accordance with the leading approach to software development today – Model-Driven Development (MDD). The main goal of MDD is to enable the automation of software development. In MDD models are primary software artifacts and development is automated through appropriate model transformations. Hence, model transformations are a key component of MDD as they represent a means for the automatic generation of target models from source models, with the ultimate goal of producing a concrete implementation (i.e. executable code) starting

from a conceptual model (i.e. a platform independent model – PIM). Thus, the main goal of the suggested approach to automating the design of NoSQL document-oriented databases is to eliminate the need for manually coding CRUD functionality by automatically generating the data structures and corresponding CRUD operations for a document-oriented NoSQL database, from an EER model.
The paper is organized as follows: Section 2 gives a brief overview of the current state of the art pertaining to this issue. The main concepts of Extended Entity-Relationship models and document-oriented NoSQL databases are presented in Section 3. The proposed methodological approach is outlined in Section 4. Finally, Section 5 concludes the paper and discusses future work.

2. CURRENT STATE OF THE ART

An assessment of the current state of affairs in the field of databases reveals that relational databases are prevalent (*Sadalage& Fowler,2013;Vaish, 2013; Lazarević& al, 2016*).Given that they have been in use for more than 30 years, a good deal of research effort has been dedicated to the issue of their design, resulting in a number of methodologies and approaches with clearly set rules regarding the manner in which a database schema should be designed.

While NoSQL databases do not presume the existence of a database schema, it is still necessary to structure and organize the data in a manner that facilitates its manipulation. Consequently, a number of decisions must be made when designing NoSQL databases, which are influenced, on the one hand, by user requirements, and on the other, by demands related to their scalability, performances and especially their consistency. These issues also aroseformerlywhen it came to the logical design of relational databases or mapping XML (eXtensibleMarkup Language) documents to relational databases (*Sadalage& Fowler,2013*).

The introduction of an abstract NoSQL database model, as an intermediate model between logical concepts and NoSQL database concepts, has been proposed in (*Bugiotti& al, 2013*) with the aim of simplifying the management of data in such databases. This paper presumes the existence of a semantically rich model (i.e. the extended Entity-Relationship model) as a conceptual model.

To the best of our knowledge, as of yet, there is no systematical approach or methodology for developing NoSQL databases. Even though a number of researchers have indicated the need for such an approach (e.g. *Vaish, 2013; Sadalage&Fowler ,2013*) NoSQL database development is currently based on the best practices in this field (*Lazarević& al, 2016; Katsov, 2012; Baker & al, 2011*).

3. OVERVIEW OF THE FUNDAMENTAL CONCEPTS

3.1. Extended Entity-Relationship (EER) model

Entity-Relationship (ER) models (*Chen, 1976*) are semantically rich data models which can be graphically expressed and, as such, are extensively used for database design. The original ER model was later extended in order to incorporate additional (semantically rich) concepts for more accurately modeling complex systems.Ametamodel of the Extended Entity-Relationship model is depicted in Figure 1.

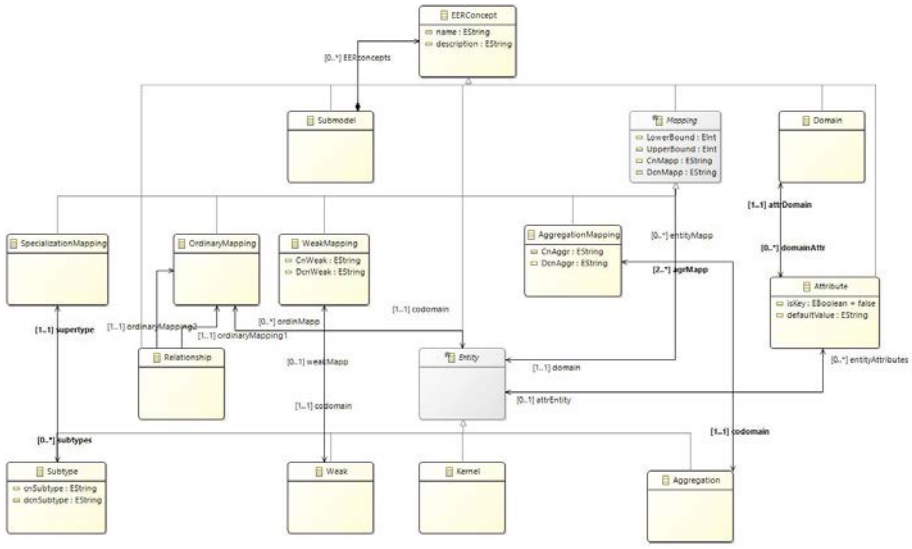

Figure 1: Metamodel of the Extended Entity-Relationship model

The *EERConcept* concept represents the core concept from which all of the other concepts are derived. The *Submodel* concept represents a concrete EER model and is the root element which encloses all of the other concrete model elements. An *Entity* is an abstract concept, representing classes of objects, which is specialized into the *Strong Entity (Kernel)*, *Weak Entity*, *Subtype* and *Aggregation* concepts. An *Aggregation*is a type composed from other entities. A *Weak Entity* is an entity whose existence depends on the existence of another entity, while a *Subtype* represents a specialization of an *Entity*.Entities are described by their *Attributes* which take their values from *Domains*. A *Mapping* represents an abstract mapping(wherein a relationship is specified by two mappings)and is characterized by two attributes: *upper*(maximum) and *lower*(minimum) bounds of the mapping cardinality. The *Mapping* concept is specialized into the *Aggregated Mapping*, *Weak Mapping*, *Specialization* and *Ordinary Mapping*.Each specialized entity is associated with a specialized type of mapping, i.e. an *Aggregation* is associated with an *Aggregated Mapping*, a *Weak Entity* with a *Weak Mapping*, and a *Subtype* with a *Specialization*.

3.2. Document-oriented NoSQL databases

As stated in (*Katsov, 2012; Vaish, 2013*) several types of NoSQL databases exist: *key-value* databases, *column-oriented* databases, *graph-oriented* databases, *document-oriented* databases and *hybrid* databases (asacombination of the other types).Given the scope of this paper, only the fundamental concepts of document-oriented NoSQL databases, adapted from (*Katsov, 2012; Chodorow, 2013*), will be elaborated.

Collection

In NoSQL databases *collections*are used for storing (i.e. physically grouping) *documents* which are to be accessed collectively (i.e. as a group). This concept corresponds to the *table* concept in relational databases but with the key distinction that the elements of a *collection* (*documents*) may have different structures (sets of fields) whereas all elements of *table* (*rows*) always have the same structure.

Document

A *document*is the main unit of data (stored in some standard format or encoding like JSON) in document-oriented NoSQL databases. The creation of a separate document for each concrete entity (i.e. a normalized structure) is recommended in the following cases (*Hamrah, 2011; MongoDB, 2016*):

- If the embedding of documents would result in a data redundancy which is not outweighed by sufficient read performance advantages;
- When representing complex M:M relationships;
- When the data set that is to be modeled is large and has a hierarchical data structure.

The elements of a document (i.e. its fields) can be either simple fields (holding a value), arrays of elements or sub-documents.

Identifier

The *_id* field is used for representing a unique attribute (i.e. an identifier) in document-oriented NoSQL databases. The value ofan *_id* field must be unique in a collection. If the user does not specify a value for this field, the value will be automatically generated. The value of the *_id* field can be of any data type, save for arrays, and it is immutable. The *_id* field must appear as the first field in a document. If it is not specified as the first field in a document, it will be relocated to the beginning of the document.

As stated in (*MongoDB, 2016*)the key issue when designing data models revolves around the structure of the documents and how the relationships between the documents will be represented.Contrary to relational databases, most NoSQL databases do not support the joining of documents in queries. Thus, when it is necessary to relate documents, one option is to store a *reference* to a document within another document (which results in a normalized database structure) while the other option is to store an entire document within another document (which results in a denormalized database structure).

Reference

Referencing is accomplished by storing the value of the *_id* field of one document (i.e. a reference to the document) in another document thereby relating the two documents. Given that NoSQL databases do not provide support for resolving references, it is necessary to execute a separate query in order to retrieve the referenced object. Thus, while references provide greater flexibility, in comparison with embedded structures, their main drawback is that they inherently necessitate the “generation” of multiple queries when retrieving a complex data structure, whereas with embedded structures the entire data structure can be obtained by a single query.

Sub-document

A sub-document is a document which is stored within another document. In contrast to references, where only the identifier of the document is stored in another document, in this case the entire structure of the document is stored (i.e. embedded) within the other document. This approach is recommended in the following cases (*Hamrah, 2011; Chodorow, 2013; MongoDB, 2016*):

- When representing weak objects;
- When representing 1:M or 1:1 relationships, and the embedded documents should always be displayed within the context of the main document.

The creation of sub-documents yields better read operation performances, in comparison with references, as all of the related data can be retrieved in a single queryand, in addition, the sub-documents can also be updated using a single write operation. However, the creation of sub-documents can lead to an increase in the size of the main document after its initial creation (for example, if an invoice, initially containing three items, is created and stored in the database, and subsequently five additional items are added) and the additional memory may entailthe relocation or fragmentation of data on the disc which is an expensive operation(*MongoDB, 2016*).

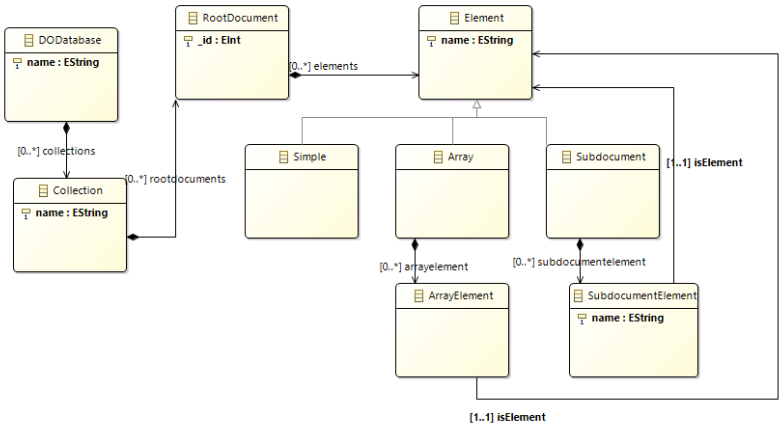

Figure 2: Document-oriented NoSQL database metamodel

4. PROPOSED APPROACH

In general, to design a database is to decide how to organize data into specific forms and how to access them (*Chen, 1976*). Database design is usually conducted in three phases (*Simsion& Witt, 2004*): conceptual, logical and physical design. The proposed approach also follows these three phases:

- Conceptual design – during which the real-world concepts, and their relationships are defined using an EER model.
- Logical design – during which the real-world concept are mapped onto NoSQL database concepts.
- Physical design – .during which additional physical characteristics of the NoSQL database are defined (e.g. indexes, etc).

Due to space constraints, and given the focus of this paper, only the conceptual and logical design phases will be further elaborated and illustrated.

4.1. Conceptual design

The conceptual design phase of the proposed approach would not significantly differ from the conceptual design of relational databases. In this phase the relevant real-world concepts are identified and their attributes and relationships are defined. The utilization of the EER model, as a semantically rich data model, is proposed for the conceptual design of NoSQL document-oriented databases.

The EER model which will be used for illustrating the proposed approach is presented in Figure 3. The *Person*, *Department*, *Course*and *Student Organization* represent strong entities. A *Person* is further specialized into either a *Professor* or a *Student*. A *Student* can be a member of multiple *Student*

Organizations. A *Professor* can belong to one and only one *Department*. The *Pay Slips* for each professor are recorded as weak entities. An *Exam* is an aggregation of the *Student*, *Course* and *Professor* entities.

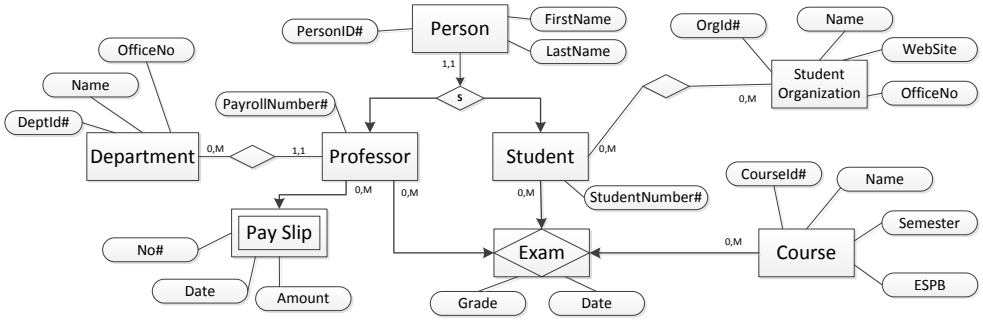

Figure 3: An example of an EER model, adapted from (*Lazarević et al, 2016*)

4.2. Logical design

In the course of the logical design phase, the concepts, which have been identified in the conceptual modelling phase, are transformed into NoSQL database concepts using a set of rules for mapping (transforming) EER concepts onto NoSQL document-oriented concepts. The proposed rules(*Stojimirović& al, 2015*), Figure 4,have been formulated in accordance with the existing recommendations and best practices in this field (*Katsov, 2012; Lazarević& al, 2016; Vaish, 2013; Baker & al, 2011*).

Rule R1: *A strong entity is transformed into either a document or a sub-document. If a strong entity is transformed into a document, then the identifier of the strong entity becomes the identifier of the corresponding database document. If the strong entity is transformed into a sub-document then the identifier of the strong entity becomes a field in the corresponding sub-document.*

Rule R2: *A subtype is transformed into a document. The identifier of the supertype becomes the identifier of the corresponding database document.*

Rule R3: *A weak entity is transformed into an array of sub-documents. The identifier of the weak entity becomes a field in the corresponding sub-document.*

Rule R4: *An aggregation is transformed into either a document or a sub-document. If the aggregation is transformed into a document, then the identifier of the document will either be the identifier of the entity which participates in the mapping with maximal cardinality of "one", or a composite identifier consisting of the identifiers of the entities participating in the mapping with a maximal cardinality "many". If the aggregation is transformed into a sub-document document, then the identifiers of the participating entities (except for the entity corresponding to the document in which the sub-document is embedded) will become fields in the sub-document.*

*Relationships with M:M cardinality are regarded as aggregated entities, thus Rule **R4** will also be applied to such aggregated entities.*

Rule R5: All entities which are transformed, by applying rules ***R1*** through ***R4***, into documents, also become collections.

Rule R6: *The attributes of an entity become the fields of the document, or sub-document, corresponding to the entity.*

Rule R7: *The identifiers of all entities onto which a given entity is mapped with a maximal cardinality of "one", become fields in the document (or sub-document) corresponding to the given entity.*

Rule R8: The *fields, obtained by applying rule **R7**, represent references.*

Figure 4: The rules for transforming EER concepts into document-oriented NoSQL concepts

The outcome of this phase is a logical model of the database structure. It should be emphasized that, as is the case with any other design, several different logical models can be derived from the same conceptual model.One approach would be to model a completely normalized structure. Given that a NoSQL *collection* can be regarded as being similar to the *table* concept in relational databases, all EER entities (i.e. strong and subtype), save for weak entities, can be transformed into separate collections. On the other hand, since the collection concept, unlike the table concept, does not prescribe the structure of documents that will be stored within the collection, it is also possible for all entity types to be stored in a single collection. Consequently, the first step is to determine which collections will exist in the database. The next step would then be to establish which entity will be stored in which collection, unless the creation of separate collections for each type of document was decided on. Finally the established transformation rules will be applied.

Rule ***R1***states that a strong entity can be transformed into either a document or a sub-document. The application of this ruleto the EER model in Figure 3, resultsin four types of documents: *Department*, *Student organization*, *Course*and *Person*. Given that these entities also have attributes, rule ***R6***is also applied, so the attributes of each entity become fields in their corresponding documents.Figure 5depicts a representation of instances of these entities, stored as concrete NoSQL documents. It should be noted that a strong entity can

also be transformed into a sub-document. This type of transformation is performed when, for optimization purposes, an entity on the “many” side of a 1:M relationship is regarded as a weak entity.

(i) the *Department* document:

```
{ "_id": 1, "Name": "Department of Information Systems", "OfficeNo": "018" }
```

(ii) the *Student organization* document:

```
{ "_id": 1, "Name": "FONIS", "OfficeNo": "026", "WebSite": www.fonis.rs }
```

(iii) the Course document:

```
{ "_id": 1, "Name": "Data structures and algorithms", "Semester": "4", "ESPB": "6"
                                                    }
```

(iv) the *Person* document:

```
{ "_id": 1, "FirstName": "Petar", "LastName": "Petrović" }
```

(v) the *Professor* document:

```
{ "_id": 1, "PayrollNumber": "7531" }
```

(vi) the *Student* document:

```
{ "_id": 1, "StudentNumber": "IT130001"}
```

Figure 5: Examples of concrete NoSQL documents

Rule ***R2***is related to the mapping of subtypes which are transformed into documents. By applying rules ***R2***and ***R6*** to the *Professor* and *Student*subtypestwo additional documents are obtained, Figure 5 (v)and (vi).

Since a weak entityis both existence dependent and identifier dependent on itsstrong entity, it will be transformed into an array of sub-documents within the document corresponding to the strong entity.Aweak entity in an EER model cannot exist without the existence of its parent entity. The same principle applies to sub-documents in document-oriented NoSQL databases.Consequently, the deletion of the parent document from the database also entails the deletion of its sub-documents, as they are contained within the parent. In the EER model (Figure 3) the *Pay Slip* entity is a weak entity of the *Professor* entity. By applying rule ***R3***it is transformed into an array of sub-documentsof the document corresponding to the *Professor* entityFigure 6.

the *Professor* document:

```
{
    "_id": 1,
    "PayrollNumber": "7531",
    "PaySlip": [
      { "No": 1, "Date": "15.03.2016.", "Amount": 35743.41 },
      { "No": 2, "Date": "31.03.2016.", "Amount": 31454.22 }
    ]
}
```

Figure 6: Aconcrete NoSQL document with a sub-document

The *Exam* aggregation of three entities: *Course*, *Student* and *Professor*in Figure 3,is transformed by applying rule ***R4.*** In this case there are two options, either the *Exam* entity will be transformed into a document, or it will be transformed into a sub-document of one of the documents corresponding to the participating entities. The choice will depend on the intended usage of the stored data.

If the first option is chosen then a concrete *Exam* documentwill be stored in the database. Its *_id* field will be a subdocument containing the identifiers of the documents corresponding to the entities participating in the aggregation (sincean *_id* field of a document cannot be an array). In the depicted example (Figure 7), the identifier of an *Exam* document is composedfrom the *Course_id*, *Student_id*, and *Professor_id* fields.

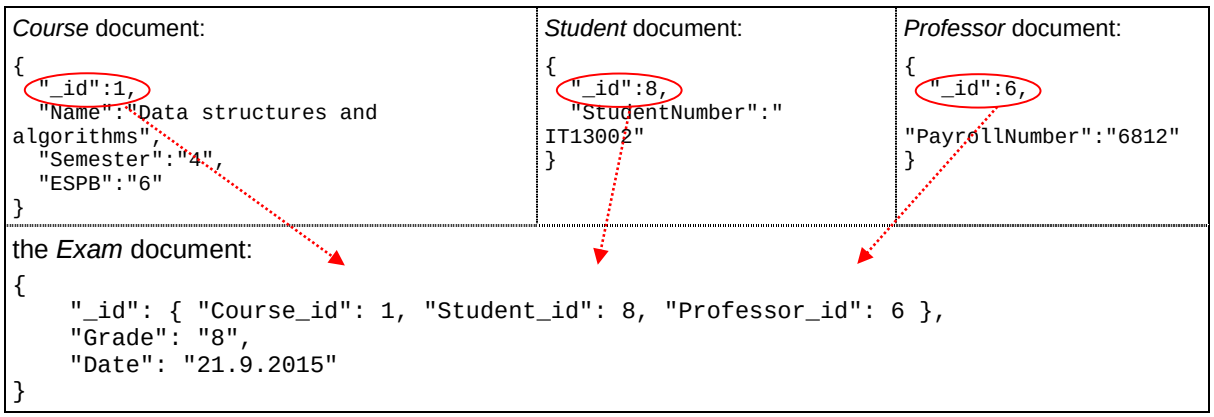

Figure 7: An example of a concrete NoSQL document generated by transforming an *Aggregation*instance

However, if the application requires that the exam results are customarily to be viewed alongside the student data, and are only occasionally to be displayed summarized per professor or course, a better approach would be to choose the second option and transform the *Exam* entity into an array of sub-documents of the *Student* entity. The identifiers of the documents corresponding to other entities participating in this aggregation (i.e. *Course* and *Professor*) will then become fields of a sub-document. A concrete example of one such document is shown in Figure 8. It should be emphasized that the *Exam* entity could be transformed into a sub-document of any of the documents (or sub-documents) corresponding to the entities participating in the aggregation. The choice is left to the designer, and will depend on the application requirements.

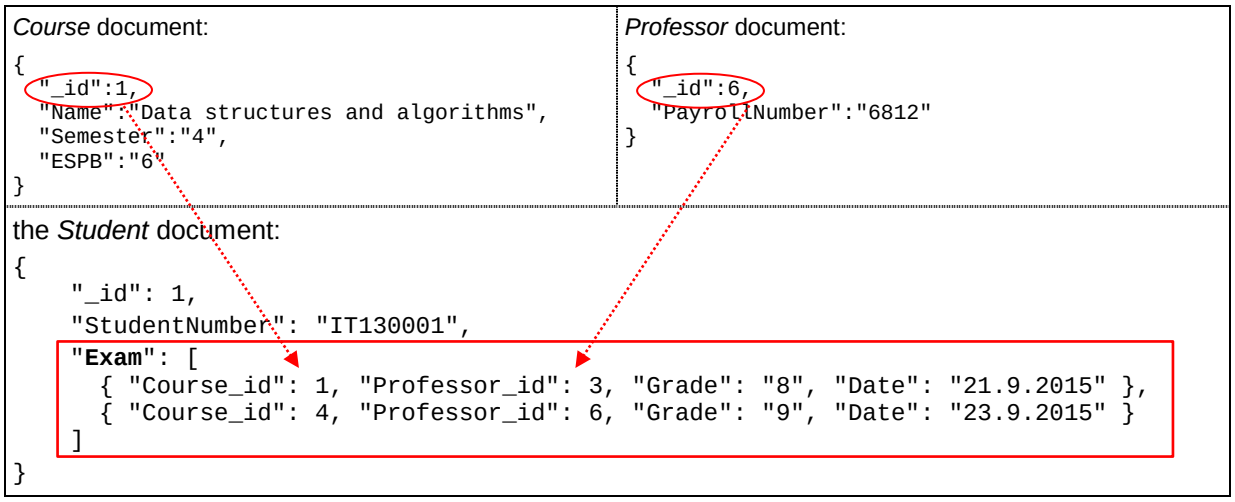

Figure 8: An example an *Aggregation* instance transformed into a concrete NoSQL sub-document

The main advantage of the first approach (i.e. a normalized structure with aggregations transformed into documents) is that such a structure offers greater flexibilityand is more easily maintained in comparison with the second approach (when aggregations are transformed into sub-documents). On the downside thefirst approach entails the execution of multiple queries in order to retrieve the relevant data.

4.3. Automating the execution of the transformation rules

The manual application of the transformation rules and coding of the necessary CRUD operations (responsible for manipulating the stored data) can, as mentioned in Section 1, be a very complex and error-prone task. The absence of a database schema makes this task even more challenging, given that it is impossible to check whether the data is actually stored in the correct format. Thus this paper suggests a means for automatically generating the code (necessary to create the data structures and corresponding CRUD operations for a document-oriented NoSQL database) from a conceptual EER model.

As defined in (*OMG, 2014*): “transformation specifications [in the context of MDD] provide the mechanisms to transform between representations and levels of abstraction or architectural layers”. In general, the transformation specification is based on a set of rules which define how the concepts of a source model (in this case an EER metamodel) are to be automatically transformed into the concepts of a target model (in this case a document-oriented NoSQL database metamodel). The transformation specification is then executed by a transformation engine which has an EER model (conforming to the EER metamodel) as its input and

generates the necessary artifacts as its output. Since document-oriented NoSQL databases, contrary to relational databases, do not possess a schema, the suggested approach would not follow the traditional EER -to-relational model transformation. It is proposed that the specified transformation could be a model-to-text transformation (M2T), wherein the source would be an EER model while the generated target would be a textual artefactrepresenting code (e.g. Java). The proposed approach suggests that three artifacts should be generated: the domain classes (for representing the in-memory object model in which the data will be stored at runtime), a generator (responsible for knowing how to transform an object into a JSON document using generated text templates storing the structure of a concrete document type) and brokers (responsible for executing the CRUD operations by invoking the generator to obtain concrete JSON documents).

Thus the first step would be to map the EER metamodel elements onto the document-oriented NoSQL database metamodel concepts. However, in order to obtain the flexibility, which the presented rules offer, the designer would need to specify, for those rules which provide alternatives, how certain concrete concepts should be interpreted e.g. how to transform aggregations, or if strong entities should be transformed into sub-documents. In general this can be accomplished in two ways: either by annotating the concrete EER model elements or by initially configuring the engine throughstatic rules providing the desired settings as XMI.
The transformation engine would read a concrete EER model and use a visitor pattern, such as topological sortor a similar algorithm, to traverse the EER model (and for each element its related elements would be visited)whereupon the necessary domain classes, brokers and JSON generator (based on text templates representing the structure of the corresponding document types) would be generated, on the basis of the specified metamodel mappings and, if prescribed, the EER model annotations. At runtime, upon the invocation of a concrete CRUD operation, the broker, responsible for the execution of the operation, would invoke the generator which would extract the information from the domain objects and populate the previously generated document template with the concrete values, thereby generating a concrete JSON document that can be stored in the document-oriented NoSQL database. The metamodels, EER model instances and transformation rules could be stored, for example, as XML (eXtensible Markup Language) documents conforming to the XMI (XML Metadata Interchange) standard, while the transformation engine could be built using e.g. Java code or, preferably an existing one could be used.

5. CONCLUSION

Contrary to relational databases, NoSQL databases do not enforce a database schema and, in addition, in relational models the data structure is decoupled from the actual data, which is not the case with the JSON documents stored in a document-oriented NoSQL database. Yet it is still necessary to design the data structures that will be stored in a NoSQL database in order to be able to correctly manipulate the stored data in an application. A possible approach to automating the design of NoSQL document-oriented databases, in accordance with the leading approach to software development today – MDD, is outlined in this paper. The aim is to automatically generate the data structures and corresponding CRUD operations for a document-oriented NoSQL database, from a conceptual model. Consequently, there would be no need for the error-prone manually coding of CRUD functionality.In the presented approach semantically rich Extended Entity-Relationship models are used as conceptual data models. The rules for transforming EER concepts into NoSQL document-oriented database concepts are presented, and a possible approach to automating these rules is outlined.

If the transformation is automated there would be no need for manually coding the CRUD functionality (which is usually very time-consuming and prone to errors). Thus the proposed approach reduces the risks of incorrect mappings, thereby ensuring consistency and reliability, while at the same time increasing productivity and lowering the costs of web application development. Moreover, the runtime performances of a web application would be accelerated, as it will not be necessary to store the object mappings separately and then check theses mappings, to determine the how to handle an object (document), each time a CRUD operation is to be executed, rather the entire manipulation logic will already be embedded in the code.

Future work would be aimed at providing a concrete implementation of the proposed approachand testing it in different real-life scenarios.

REFERENCES

Baker, J., Bond, C., Corbett, J. C., Furman, J. J., Khorlin, A., Larson, J., ...&Yushprakh, V. (2011). *Megastore: Providing Scalable, Highly Available Storage for Interactive Services*. In Proc. of Conference on Innovative Data system Research– CIDR (Asilomar, CA, USA), pp. 223-234.

Bugiotti, F., Cabibbo, L., Atzeni, P., &Torlone, R. (2013).*A Logical Approach to NoSQL Databases*.Retrieved from:http://cabibbo.dia.uniroma3.it/pub/noam.pdf

Chen, P.P.S. (1976). The entity-relationship model – toward a unified view of data.*ACM Transactions on Database Systems (TODS)*,1(1), 9-36.

Chodorow, K. (2013).*MongoDB: the definitive guide* (2nd ed.).Sebastopol: O'Reilly Media, Inc.
Hamrah, M. (2011*). Data modeling at scale: MongoDB+ mongoid, callbacks, and denormalizing data for efficiency*.Retrieved fromhttp://blog.michaelhamrah.com/2011/08/data-modeling-at-scale-mongodb-mongoid-callbacks-and-denormalizing-data-for-efficiency/
Katsov, I. (2012). *NoSQL data modeling techniques*.Retrieved from Highly Scalable Blog:https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/
Lazarević, B., Marjanović, Z., Aničić, N., Babarogić, S. (2016).*Bazepodataka* (7thed.). Beograd: Fakultetorganizacionihnauka.
MongoDB (2016).*MongoDB Reference Manual*. Retrieved from http://docs.mongodb.org/manual/
OMG (2014).*MDA Guide, Revision 2.0*. Retrieved from http://www.omg.org/cgi-bin/doc?ormsc/14-06-01
Sadalage, P.J., &Fowler M. (2012).*NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Addison-Wesley.
Simsion, G., & Witt, G. (2004).*Data modeling essentials*. San Francisco: Morgan Kaufmann Publishers.
Stojimirović, D., Nešković, S., Babarogić, S. (2015).Predlog postupka projektovanja NoSQL baza podataka zasnovanih na dokumentima. In Proc. of YU INFO 2015 (Kopaonik, Serbia), pp. 115-120.
Vaish, G. (2013).*Getting started with NoSQL*.Birmingham: Packt Publishing Ltd.