Domain-driven Design: Overview of Performance Benchmarks and Feasibility Assessment for DSL Platform

Nikola Vlahovic

***Abstract*—**Domain-driven design (DDD) is a software development approach that focuses on the development of a complete model of the software system domain. The model itself can then be used to generate system components. Unlike traditional software development approaches focus of DDD is on the process experts and, in the case of business applications, on business experts' knowledge about business processes. In order to bring the development process closer to business experts that are usually non-IT experts or programmers new software tool or platform is required. One such platform is DSL Platform.

DSL Platform is an infrastructure that can be used to develop and maintain critical complex software systems that supports DDD approach in higher extent than other available software solutions.

In this paper we will examine and analyze available benchmarks of the DSL Platform in comparison to leading software development tools, methodologies and techniques. The results will show the benefits and advantages in implementation of this tool both for the development of complex software systems and even more importantly the maintenance of existing complex software systems.

***Keywords*—** Software development, Software value, Software maintenance, Domain-driven design, Software engineering, Software refactoring, Legacy systems.

I. Introduction

COMPLEX software systems are software systems that for their proper operation rely on a number of different, usually, incompatible technologies, that are usually the results of prolonged software system life cycle, high scale of transactions or they perform critical core business tasks. Prolonged software system life cycle may lead to using and relining on legacy technologies and technologies that are no longer supported by their developers. On the other hand once critical core business processes risk is well covered by current software system, top management becomes reluctant to make changes if it is not absolutely necessary i.e. if the risk of discontinuation of business process does not become immediate threat for the company. Finally, over time the maintenance of this type of system becomes very expensive and inefficient. Current software development methodologies and software approaches cannot cope with this type of systems in an efficient way. This is why there is a constant need for the development of novel software development methodologies and approaches. Practitioners are developing and presenting new frameworks and technologies as well as new approaches to software development altogether, while only a limited number of these developments enter the mainstream adoption by software or even non-software companies. In this way a software approach called Domain driven design has been developed. This approach presents properties that have the capabilities to cope with coupled heterogeneous software systems while improving maintenance efficiency in current dynamic business and technological environment. At the same time it tries to offer solutions for bridging the gap between business experts and software experts that is main drawback in traditional approaches that additionally decreases the efficiency in maintenance of complex software systems.

Agile methodologies are more successful in coping with this gap for reasonably limited and small-scale software systems. When it comes to complex business systems only approaches with traditional core principles are available, mostly with increased inefficiency and additional development and maintenance costs [19].

Domain driven design has fostered new tools that are available to broader community of practitioners. One such tool based on DDD is DSL Platform.

In this paper we will analyze the main properties of DSL Platform as a DDD based tool and compare it to other available tools in terms of features and capabilities as well as appropriateness to different software system development and maintenance. The analysis will include both technical aspects as well as economical aspects of application implications. Also available independently conducted benchmarks will be present and analyzed in order to compare DSL Platform to other available software developing and maintenance tools. Based on the comparison results we will propose enhancements to currently available classifications and models that can improve the understanding of available software development methodologies and also improve models of assessing economic metrics for software systems, primarily estimation of software asset value and maintenance costs.

Goal of this paper is to estimate performance level of DDD based tool DSL Platform and identify its position in current classifications of software development approaches. After that

N. Vlahovic is the associate professor at the Informatics Department of the Faculty of Economics and Business, University of Zagreb in Croatia. Trg. J.F. Kennedyja 6, 10000 Zagreb, Croatia (phone: +385-1-238 3220; fax: +385-1-233 5633; e-mail: nvlahovic@ efzg.hr).

we will be able to extrapolate its influence on software production costs and software value estimation using currently most comprehensive estimation models.

The structure of the rest of this paper is as follows: In Section II domain driven design with its key features will be described. Currently available classification of software approaches will be considered to identify DDD's position and role in this classification. Also advantages and disadvantages will be presented in this Section. In Section III DSL Platform will be presented. It is a software development tool that is based on the DDD principle as well as some of the most efficient principles, techniques and methodologies available in software development. The second part of this Section available benchmarks will be presented and analyzed. First two Sections deal with technical aspects of software development and maintenance, while the rest of the paper will move focus to economic and social aspects of software development and maintenance efforts. Section IV will present most important software management issues that are determined by the technical aspects of software development and management tools. Here software assets will be explained along with their properties. Next software development effort will be defined as well as maintenance tasks. After that we will parent some approaches to software value estimation that takes into account all of the properties and other requirements into account (such as legislation and International Accounting Standards…). In Section V we will compare, analyze and discuss presented information and estimate possible impacts of domain driven design within the software development process for complex coupled heterogeneous systems, if this approach is fully integrated into business process throughout the software process life cycle. Here we will present a SWOT analysis that will be used to extrapolate the benefits and issues that the management should be aware when considering introduction of domain driven design. Finally in Section V. conclusions will be given and an outline for future work.

II. Overview of Domain driven design

Domain driven design (DDD) is a software development approach. Unlike most of other software approaches that analytically organize the software development effort and use conceptual, modeling, programming and implementation tools, domain driven design is focused on the software model itself. DDD strives to make a complete model of the problem domain moving the focus of the development effort away from tools, techniques and methodologies used.

In the most general terms software development approaches can be divided into two diametrically contrasted classes and one intermediary class that draws on some of the concepts from either of the two main classes [1]. This classification of software approaches is given in Figure 1.

1) Class of structured approaches. This is a group of software development methodologies that are based on a process that recognizes distinct phases of the software development process. These phases usually align with particular stages of the software development life cycle (SDLC). Depending on the particular methodology each phase can be associated with a stage in SDLC either, planning, creating, testing or deploying of the software system. Some methodologies can have several phases associated with one stage of the SDLC, and others can have one phase spanning over or overlapping with two stages of the SDLC. The main characteristic of methodologies in this group is that each phase needs to be completed with some final result, a software artifact, before next phase of the process can begin. Some of the most common methodologies that belong to this group are waterfall software development model, prototyping, incremental development, iterative incremental development, Boehm's spiral model, etc. but also object oriented approaches.
2) Class of behavioral approaches. This group of methodologies relies on the soft systems approach that takes a more relaxed definition of development process. Behavioral approaches take a holistic view of the organizational systems and social nature of software systems (both in development and deployment stages). This is why these methodologies promote participation of system users and customers during the creation phases of the system. Also the development process may return to earlier phases as required by the current perspective of the software system and even different development activities may overlap. Along with soft systems approach we can find characteristics of the behavioral approach in agent

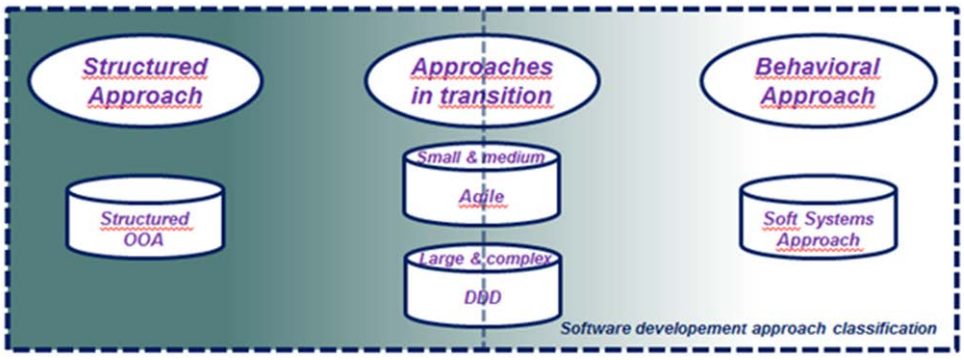


Fig. 1 classification of software approaches

based software engineering [3], [4] as well as in the behavior-driven design [5].

3) Intermediary and transitional approaches. This class of approaches to software development shares some of the characteristics with the structured approaches and some of the characteristics with the behavioral approaches. These methodologies represent the synthesis of traditional rigid structure and softer humanist elements of the behavioral approaches. Agile methodologies represent the most typical example of a transitional approach due to their strive to capture the human aspects of organization for all stakeholders involved, especially during the analysis and planning stages, while still retaining structure in design and implementations stages [6], [1], [19].

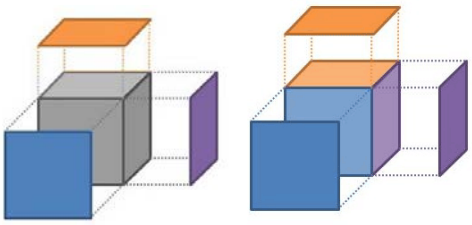
Domain driven design (DDD) as a somewhat recent novel software development approach tries to change the traditional focus from the project methodologies and tools towards the core of the problem at hand. DDD goes even beyond a particular technology or methodology, or even a framework. It is a way of thinking and a set of priorities aimed at accelerating software projects that have to deal with complicated domains [7]. As such it is very close to behavioral approaches, but as it strongly relies on hierarchies of priorities and concepts typical for structured approaches, it can be regarded as a transitional approach to software development.

Still, unlike agile methodologies that are focused on a limited, small to medium sized software projects, DDD is primarily concerned with complex and coupled software systems. Due to its platform-independency, it is an encompassing approach to highly coupled systems that use different, even inconsistent, technologies and platforms as well as development methodologies or practices. This types of systems cannot be successfully developed using original agile principles. Usually these types of coupled complex software systems are developed using more traditional structured approaches, simply in order to be able to manage the complexity of the system, tolerating the inefficiencies of most of the other aspects of the software life cycle management process.

This is why DDD is an appropriate candidate to cover this type of software systems, since it improves the efficiency of the development and maintenance of coupled complex software systems using principles similar to agile development, while at the same time enables the development team to successfully manage all of the steps of the development and maintenance effort as if structured approach is used. This is why DDD is defined as an intermediary approach in Figure 1, that completes the given classification.

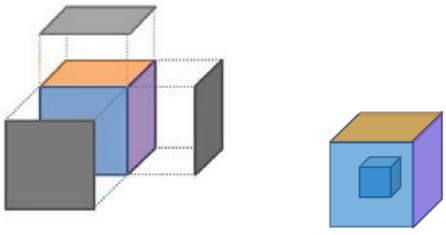
In order to understand how DDD can connect all of the varieties of concepts into a consistent and unified one, we will take a look at how previous methodologies and frameworks represent software projects. Most of them treat a software project as an entity that has to be described using a number of different perspectives. Since there are a lot of different stakeholders involved in the development of any software project, a variety of perspectives is used to promote better communication and understanding between stakeholders. In practice Unified Modelling Language (UML) is mostly used for static and dynamic representation of these perspectives. Before the beginning of software development, introductory and preparatory phases are conducted where all of these perspectives are defined (as seen in Figure 2.a).

At this stage UML covers all of the relevant views of the software system, its surroundings and dependencies using three groups of dedicated diagrams, structure diagrams, behavioral diagrams and interaction diagrams [8]. Inevitably, different perspectives may not be entirely compatible and this may present a challenge for the development team in continuation with the development of the project. After this additional effort in reconciling differences and incompatibilities between different models, product development begins and finally working version of the software system is produced (Figure 2.b).



a) b)

Fig. 2 model and perspectives of the model a) in the preparatory stage where only models exist and (b) during development of software system where a working software system is produced



a) b)

Fig. 3 DDD model and software product a) in the preparatory stage where model is programmed and (b) during development where a working software system is instantiated from the developed model

Unlike UML that takes on a number of perspectives of the model, DDD tries to describe the model by describing its domain as a whole and complete model (Figure 3.a). In this way, model itself represents the system being developed and there are no variations dependent on the perspective. In this way there are no compatibility issues with different view point over the entire software systems. Preparatory stages of the software system development are much shorter since no *a posteriori* negotiation is required. Consequence of this approach to development of the software system model is that programming code is the representation of the model. During the development stage instance of the created model is generated that represents a run-time version of the system that can readily be put into production (Figure 3.b).

As we can see the importance of the programming code is emphasized in DDD. This is why it is crucial to understand the requirements and features that are essential for the programming language that is used to model final software system. Inappropriate, platform-dependent technical programming code would cause lock-out effect for diversity of technologies, platforms, methodologies as well as a number of stakeholders, especially business experts with no programming skills. In order to avoid these lock-out effects specific requirements are expected from the team communication facilities.

Firstly, a domain specific language (DSL) is required to describe the model of the software project. Secondly, a ubiquitous language for team communication should be used and evolved during the development of the project. Consistent communication between business domain experts and developers expressing their views of the system in terms of model concepts will evolve into a ubiquitous language. The team understanding of software artefacts will express itself in the source code of the system as it represents the model of the system (through DSL). Any change in the model will change the model and these changes are clearly visible to all of the project participants, both business experts and developers [9]. DDD is an ongoing process of expressing ubiquitous domain language in code [10].

In the following Section we will present a DDD based tool called DSL Platform in order to explain the implementation of DDD principles with special focus on the formalization of a language used for the creation of a domain specific language.

III. Characteristics of DSL Platform and Benchmarks

DSL Platform is one of the most comprehensive implementations that are based on DDD and that provide tools for modeling domain specific applications as intended by the DDD software approach. In this way DSL Platform represents a unified platform for development and evolution of complex software systems. It can be considered a service that helps in designing, building and maintaining business applications while providing the development team with tools to create their own ubiquitous language through the development of DDD model while automating various steps in the business application development process and maintenance processes of the developed business application.

Essentially, platform uses specific business model as input and outputs finished components for corresponding business software system. Since DSL platform draws on the strengths of the DDD approach, business model is described in understandable language for both business experts and development team while this description is also a formal specification of the system (Figure 4). Declarative specification of a software system is defined using industry standard concepts and terminology for client domain. This results in understandable documentation which is also a formal specification of the system. Supported compilers use that specification to build code or web pages and maintain or migrate database depending on the current state of the project. Once software solution is built various features become available for automatic maintenance of the model. Developers can focus on important parts, such as specific features and user experience while more technical and manual, time consuming tasks are taken care of by the platform functionalities. Unlike with other comparable tools, advanced features, such as event sourcing or OLAP analytics are available with a fewer lines of code.

True value of DDD approach becomes apparent during the maintenance and evolution of the system. Any changes made to the business model are automatically translated by the platform into Client code or Databases (as shown in Figure 4). This functionality alleviates programmers' efforts and moves focus of their work to specific functionalities and user experience rather than code optimization, refactoring or similar technical tasks.

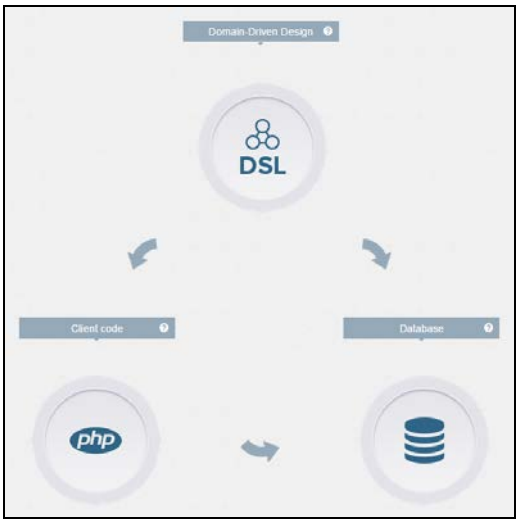


Fig. 4 DSL Platform concept

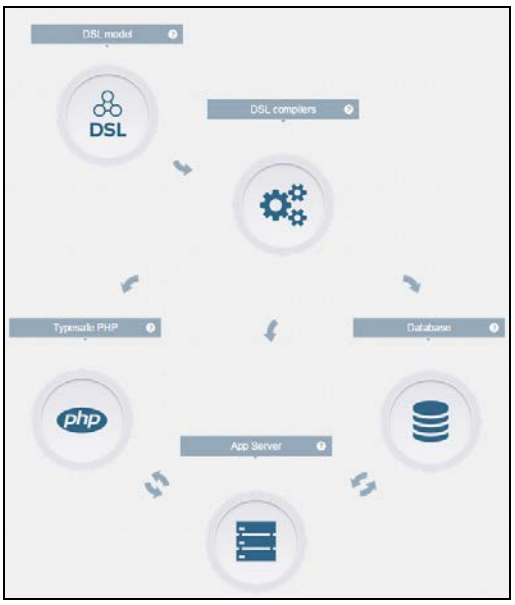


Fig. 5 Modeling business domain with resulting typesafe code, database and application server

Modeling a business domain using DDD within the DSL Platform is conducted using language specialized for such task (Figure 5). Properties of the language are created in such a way that even domain experts can read such descriptions. On the other hand this specialized language has all the technical properties that allow it to conduct efficient serialization of the model, parsing and other features that ensure model consistency. It is a functional specification for compilers, so that the described model can be checked for errors. Type safety is integrated into targets, even when they don't support it. Resulting program classes have type safety embedded within them (Figure 5). In this way programming errors are caught as early as possible. Another important part of the software system are databases (Figure 5). Their specification also originates in the described domain model, and is also checked by the compiler. DSL Platform uses object-relational database in an advanced way. As it relies on the results of the domain model there is no need for ORM tools because object-relational impedance mismatch doesn't exists at the stage of creation or migration of database. Finally, compiled code and database are stored on a server in order to synchronize any further changes to the system as domain model may be changed iteratively during the evolution stages of the system (Figure 5). Stateless application server can be added as required. Reporting, data analytics, event sourcing or any other custom feature can be consumed through various communication protocols, requests or electronic exchange standards such as JSON, XML or Protobuf.

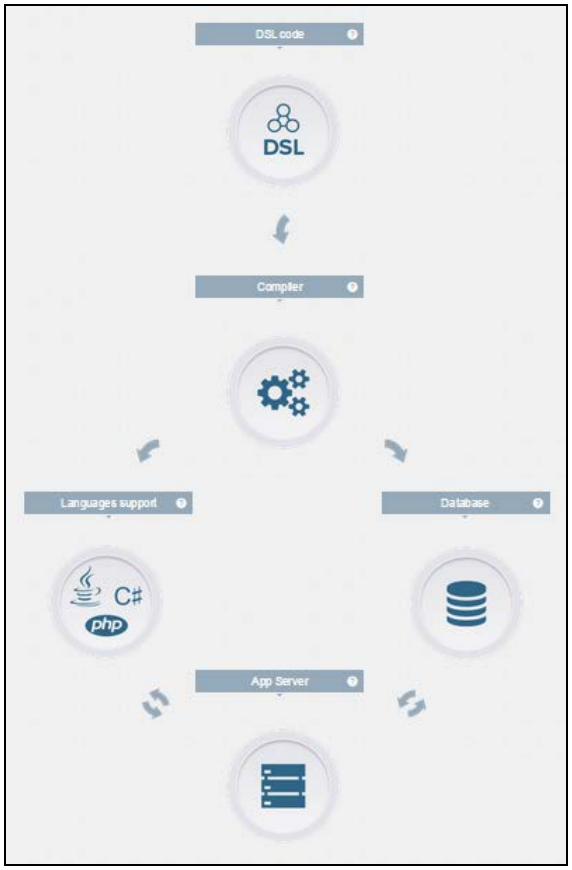


Fig. 6 Additional benefits from code reuse, scalability and compatibility of the platform towards other technologies

Finally, having described the tasks and components of domain specification development we can take a look at potential benefits and additional functionalities that can greatly improve efficiency of the software development and maintenance efforts (Figure 6).

There are five different elements that provide additional benefits:

1) DSL domain model. Once developed domain model can be reused in different technologies without friction. There is no need to re-write same model for different technologies (which is often the case in coupled complex systems).
2) Extensible compilers will take care of converting DSL domain model to various languages using best practices. This helps with maintaining high level of quality since extensive programming experience is provided in compiled libraries.
3) Programming language support. Support for various languages allow the creation of parallel variants of programming code based on the same model: .NET/JVM for the backend, dynamic languages for the frontend or Java for Android, etc…
4) Databases. Automatically maintained stored procedures optimize data access for best performance. LINQ conversions to database functions and expressions at compile time are available, as well as cache invalidation from messaging system. Combining multiple database request in a single call by using reports and similar concepts makes access and communication with the database more streamlined and efficient.
5) App server. Extensive knowledge implemented as various patterns and concepts provide additional benefits. Instead of mixing generated and hand written code, available compiled libraries can be consumed as standard REST-like API or pass-through to backend services.

As we can see two main challenges that can be effectively solved using DSL Platform and underlying DDD approach is the elimination of miscommunication between clients and contractors or even among developers within developer teams. The other is the elimination of non-creative and repetitive work done by developers by automating repetitive tasks of the development process.

A. Team communication

Teams are formed for each software project. Also for each software project there are additional stakeholders that all need to communicate with each other. They need to communicate their views, ideas and concepts between themselves. Due to different backgrounds (business backgrounds or engineering backgrounds) as well as different perspectives of the project sometimes this communication can be misinterpreted. Due to high volume of interactions between different groups of stakeholders development process may misinterpret customer needs, and finally end up with a product that does not fulfill contractors' expectations. This is why DSL platform uses a

specific language dedicated to describing business problem domains. Having a model discussed and represented using the unified language with unified meanings and understanding of concepts, team communication is significantly improved, resulting in a software that meets user need better. Documentation that is generated in this manner better specifies the software project, promotes consensus among team members and has overall higher quality. DSL Platform takes the documentation even one step further, since the documentation itself represents a full formal system specification that can be readily used for rapid prototype system validation.

B. *Source code automations and efficiency*

The formal specification of the business system can be used as a solid basis for improvement of code generation and manipulation. Dedicated compiler of DSL Platform can use this formal description of functional specifications to create any of the components for the finalized business software system. These can be libraries targeted for a particular programming language or framework or database artifacts for any relational or object-oriented database system. During the creation of the software artifacts, due to formal specifications, additional improvements of code can be automatized creating faster and more reliant execution of system tasks as well as creating more maintainable source code for the project. Finally a number of database maintenance and administration tasks can be performed using DDD model and then implementing them by simply migrating changes into a particular database system.

C. *Relevant benchmarks*

In order to evaluate performance of DSL Platform a number of benchmarks are conducted. For the purpose of this paper we will concentrate on testing the efficiency of serialization of data and data access using DSL Platform in terms of volume i.e. size of running code and time i.e. time required for the serialization process to create data access objects and remove them i.e. deserialize them.

Due to the characteristics of DSL Platform that we described earlier, it is inevitable that the architecture of this tool is multilayered. This may imply that the serialization will be more time consuming than other comparable tools. DSL Platform tries to compensate its unfavorable architecture by applying innovative algorithms with the goal of improving its serialization efficiency.

For the purpose of this test, test platform with following characteristics was used:

Operating system: Linux
Java Virtual Machine: Oracle Corporation 1.7.0_76
CPU Cores: 4

Test focused on encoding/decoding cycle-free data structure. Limitations of this test apply since different technologies and tools cope with serialization in different ways and some of them have additional capabilities that are convenient during this task The test was based on generating cycle free tree data structure where multiple object reference serializes that object multiple times; there are no manual optimizations but the scheme is known in advance.

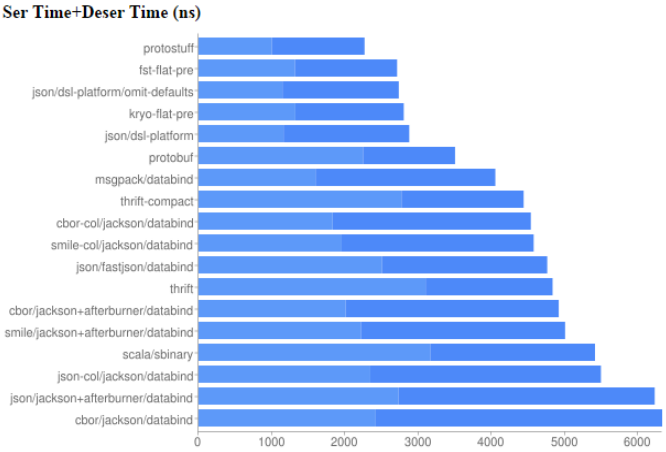


Fig. 7 Benchmark: Serialization test in nanoseconds

As we can see in Figure 7, DSL Platform is at the top of the results with low time consumption and increased efficiency in both serialization and deserialization tasks in comparison to other programming architectures. The size of code is somewhat larger in comparison to other relevant technologies and this is due to multilayered architecture of DSL Platform. Nonetheless, the size is still in medium tier of the results. Only best results are shown in Figure 8.

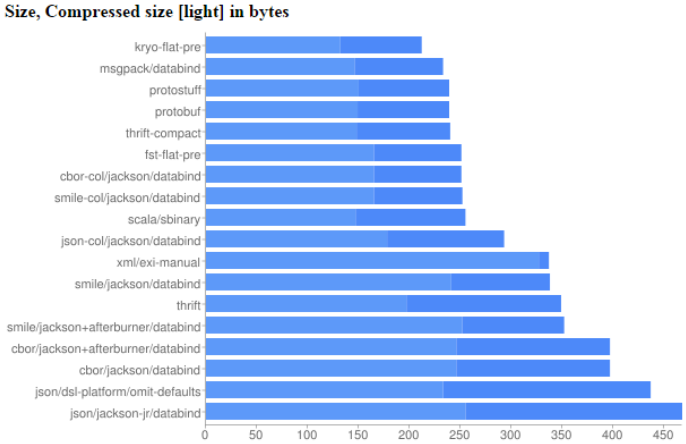


Fig. 8 Benchmark: Serialization test in compressed size in bytes

The best values obtained through test for serialization were 1145 ns for serializations, 1588 ns for deserialization, total time 2733 and compressed size in bytes 437.

For the repeated test using text format based data with inline scheme only XML/JSON serializers were compared. DSL Platform performs as the fastest serilizator (Figure 9). The size of code is also favorable (Figure 10).

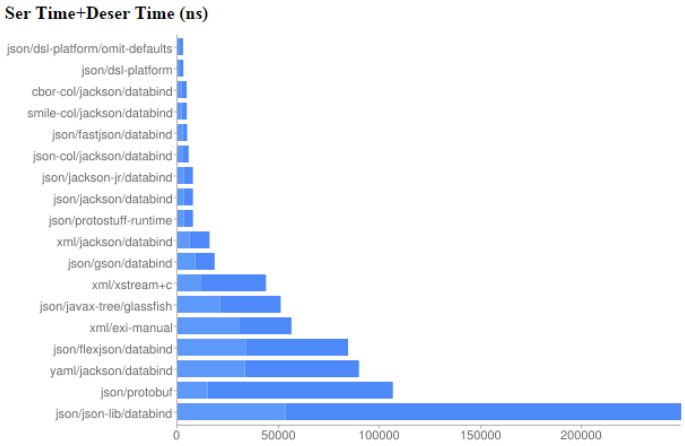


Fig. 9 Benchmark: XML/JSON test in nanoseconds

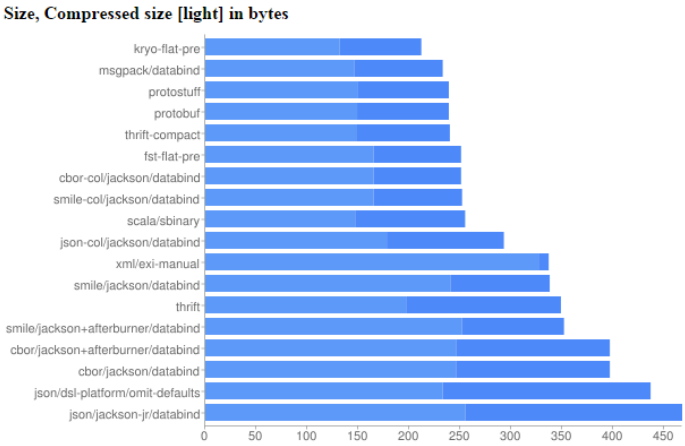


Fig. 10 Benchmark: XML/JSON test in compressed size in bytes

Overall we can conclude that DSL Platform performance is comparable to best available serialization tools while in its own class of tools (XML/JSON Serializators) it creates best results and has the best performance.

IV. Overview of Relevant Software Management Issues

Technical aspects of software development are main perquisite for the development of a software system. Efficient algorithms, technologies and methodologies such as we described in the first part of this paper also influence the efficiency and quality of the final systems. But nevertheless, non-technical aspects of software development play as important and sometimes even more important role in software development. These aspects determine whether the software system will be created, how long it will be maintained, how well it will perform its intendent tasks and also how profitable the system will be for the interested parties and stakeholders.

This is why this Section will be used to overview most critical non-technical aspects of software development. These aspects are concerned with managing the software project, economic constraints and decision making. Before we review these issues, we need to define what are the specific properties of software from economic point of view. Here we will see that the final cost of software system is more complex issues that the technical feasibility since the quality of software determines its cost but also its values as an asses and most importantly for coupled complex systems – maintenance procedure and costs.

A. *Software asset*

Software as an asset has some of the properties that differentiate it from any other asset, tangible or not [11]:

1) Indestructibility. Using software over time does not degrade its quality notwithstanding the length of usage or number of uses. Consequently this property reinforces the internal quality of software asset and its durability, so that the change in its value is solely determined by external factors. In this respect software value may deteriorate over time [13], especially with the technological advancements that change the working environment of the software.
2) Transmutability. Personalization, customization, modification and other altering practices of existing software systems are easily achieved which results in cost-effective production of software variants. This is particularly important for customer segmentation and price discrimination market targeting strategies [12].
3) Reproducibility. Since high-quality copies of the original software can be produced at low cost may authors agree that the marginal cost of production is almost zero [14]. Structure of production cost for software products contains primarily fixed cost for the software provider. Production of each additional unit does not significantly increase the total cost. In this respect the potential reproducibility deliver to software assets also significantly improves its value.

Along with this features software assets may take advantage of different economics phenomena that can also influence the estimation of its value. We will mention just a few examples. The network effect that the use of final product or services may produce in the targeted market segment can create lock-in effects promoting customer loyalty and stabile customer base. The wider the customer base the more valuable software asset becomes according to Metcalf's law. Consequently the value of customer product and services that are based on that software asset increases proportionally. Distribution of software using corresponsive Internet services reduces or even eradicates the costs of logistic and inventory. Internet services also may transform software products into services. Many desktop applications now are available as online services (SaaS) that allow for more effective pricing strategies through pricing discrimination.

B. *Software development issues*

Software assets are obtained through the process of software development. Some of the software assets are internally developed software systems that are used either to offer services on the customer markets or to sell the software itself on the customer market. Sometimes retail software either generic or tailor-made is used with the same purpose. Either way developments issues will be reflected in the final product.

Most generally development issues can be divided into several groups of issues: technical issues, process issues, people issues, project issues and holistic issues. Technical issues primarily include problems of complexity, conformity, changeability and invisibility. Some of the most important issues relate to refinement of user requirements when deciding and defining what is supposed to be developed. Also design of user interface may pose a challenge as it is not an engineering discipline but more of a creative non-systematic process. Process issues include the decision on using agile or structured approaches to software development. As we already mentioned for different types of project different approaches may be applicable, but the team management need to decide what approach will be used. People issues include communication problems and adequate levels of competency in the dynamic technological landscape. Project issues are concerned with different estimations of software, such as software value which will be discussed in more detail in the remainder of this Section. Finally, holistic issues refer to all other issues that relate to software system but that are determined during the development stage, such as quality issues.

C. Software maintenance issues

In the focus of software maintenance issues is software continuation or discontinuation of software maintenance and evolution. The decision is iteratively re-estimated periodically. Each decision during the maintenance can influence the current value of the software system, either improving it or, more usually, decreasing it. The approach to estimating the value of the software system is crucial during these decision making processes. This is why we will take more detail look at estimating software value in the following Section.

D. Estimating software value

In strategic management one of the most important basis for decision making is the assessment of economic value assets. Even more importance for appropriate decision making is the precision in assessing the economic value of intangible assets as their value may be harder to realistically judge.

All of the described features of software assets should be taken into account during the estimation of software value.

Currently, software value estimation in practice is based on three possible approaches [15]: (1) cost-based; (2) demand-driven or value-based and (3) competition-oriented.

The cost-based approach is widely used as it is covered by the International Accounting Standard 38 – Intangible Assets (IAS 38). Main purpose of IAS is to standardize financial reports for all countries that accept the standard in order to make their financial statements comparable, basic accounting principles are adopted. For asset measurement this means that there is a preference for underestimating the asset value rather than overestimate it. This is why most of the value estimates are based on historical value which is usually lower than current value, or market value, especially for intangible assets.

Computer software is treated as an Intangible asset as it is a non-monetary asset, without physical substance and identifiable. Standard defines that its value is initially measured with cost, subsequently measured at cost or using revaluation model. Also, it takes into account future economic benefits that the asset may yield. Even though these benefits may significantly influence the value of software assets, they are usually overlooked in practice, so that during the estimation of software asset only production costs is taken into account. Even production cost does not necessarily translate into software value, since during the development of software a number of software functionalities may be developed that never make it into the final product [2], or increase in project costs that do not directly increase the value of software being developed (i.e. expensive overheads, accommodation and travel costs for team members, etc.). Poor project management practices are not taken into account during current estimation approaches as well as the quality level of software asset. All these elements may lead to overestimation of software assets which in turn is contrary to basic accounting principles.

Accounting value used for financial reporting, therefore, does not reflect the true potential of software assets, honoring the specific properties that we described earlier, for the purpose of strategic decision making. Using accounting value will either underestimate or overestimate capitalization on the balance sheet or inevitably misrepresent due diligence before possible acquisitions. Strategic decision making requires better estimation of the potential of software assets that takes into account specific properties and potential software assets offer.

This is why new approaches are developed in order to make the estimation of software value more reliable. In the remainder of this Section we will present an estimation model based on the notion of technical debt and interest as described by Groot et al.

E. Software Valuation based on Technical Debt and Technical Interest

Technical debt is a type of opportunity cost defined as a set of quality issues or problems in software that will cost the organization that owns the software greater expanses if they are not resolved [16]. Furthermore, there are two major components of technical debt [18]:

1) principle, as cost to repair a software system in order to achieve ideal level of quality and
2) interest, as additional maintenance cost due to the lack of quality.

Technical debt increases over time if the quality issues of software are not resolved due to maintenance costs that increase as additional effort to negotiate quality issues is called for [17]. According to financial economics principle of technical debt is a cost that increases over time by the rate of interest (Figure 11).

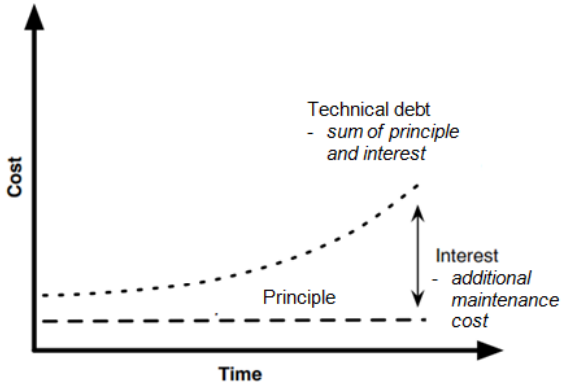


Fig. 11 Structure of Technical debt over time

Due to this increase of technical debt over time, it is feasible to pay the initial cost to repair software system and bring it to the ideal level of quality. At this level lower maintenance cost are required for the operation of the system in the future. In Figure 12 we can see that future benefits from software system operating at the ideal level of quality yielding significant savings.

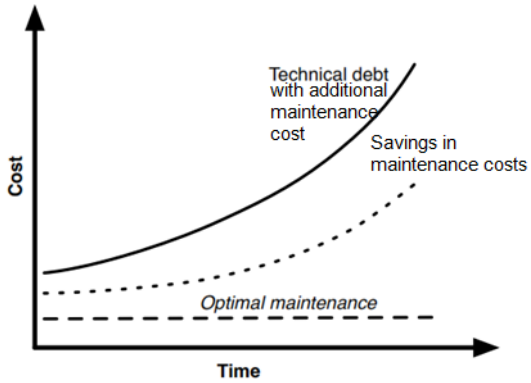


Fig. 12 Benefits from maintaining software system at the ideal level of quality

In order to include technical debt in the estimation of software value [2] have proposed a layered Software Valuation Pyramid model. This model relies on SIG Maintainability model (SIG) to determine the software development level and conclude the ideal level of software quality. On top of development level estimates they propose metrics that help estimate the operational costs of developed software systems with three key measures: rebuild effort, repair effort and maintenance effort (Figure 13).

Rebuild effort (RbE) is defined as technology-neutral measure of technical volume, based on the technology used and volume of produced source lines of code (SLOC). Repair effort (RpE) is equal to the technical debt of the software system which is primarily determined by the quality of software development process. This means that only a part of the software system needs to be rebuilt and this part is referred to as the rework fraction (RF). Maintenance effort (ME) is the yearly effort estimated to be required for regular maintenance of the system, including bug fixes and small enhancements.

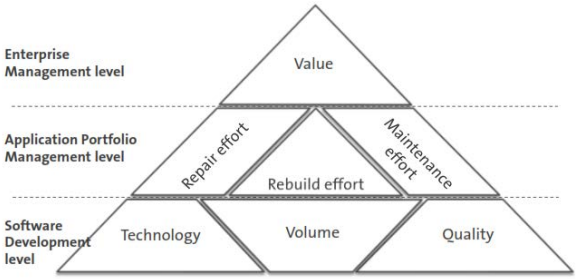


Fig. 13 Software Valuation Pyramid (Groot et al, 2012)

Based on the above defined metrics [2] propose tree different models of estimating software asset value.

F. Software Asset Estimation Models

For the purpose of this paper we will consider three models of estimating production value of software assets, which will be bases of analyzing impact of DDD approach to software asset development. All of the models are based on the assumptions that (1) there is a known level of software asset quality based on SIC metrics described earlier and (2) there is an ideal level of quality for software asset at hand that is higher than the current level of quality as previous empirical studies suggested. Even if the ideal level of quality is lower than the current level of quality these models of value estimations may apply.

First model is based on Repair effort (RbE). According to this model estimated value V is equal to rebuild effort discounted by the repair effort (RpE) required to bring the quality of software asset to ideal level.

Second model is based on the Rework fraction (RF). If bringing software system to ideal level requires the replacement of complete component or set of components that the estimated value of the system V is equal to the value of the part of the system that does not require any improvements (i.e. the value of the fraction that ought not to be reworked).

Third model is based on Technical interest. Here rebuild value (RV) is discounted by the value of technical interest during the working lifespan of the software system. Technical interest is the increase of maintenance cost that occurs if the system is running in its current level of quality. The amount of additional maintenance cost is given in Figure 4 as dotted line, representing the possible increase of present value of software system if it were upgraded to its ideal level of quality before its introduction into production phase.

For further details refer to the paper [2].

V. Disucssion

A. Impact of DSL Platform on Software value and Maintenance costs

As we can see in the proposed models of estimating value of software assets, all of them heavily rely on the costs that the exploitation of software asset incurs. Therefore, we may infer that software assets that are not used tend to lose their value, since there are no maintenance costs except storage costs. The

value of these assets decreases until it reaches the value of acquisition as defined in IAS 38.

For software assets that are activated and operational in the production system, estimation of its value can be executed using described models. The main determinant of the estimation level will be related to the quality of software development approach. This is inevitable as the Rebuild effort (RbE) relies not only on the volume of the system (i.e. SLOC) but also the characteristics of the technology used. The technological measure includes the properties of software development environments, programming languages and practices, as well as project management principles and software approaches which results in corresponding level of software quality.

On the other hand Repair effort (RpE) takes into account the maintenance costs that heavily rely on the chosen software approach to software development life cycle (SDLC)[21].

All of the three models benefit from the efficient software approach as the estimated value of software asset increases. If software approach allows for higher technological coefficient the final RbV will be higher resulting in higher value estimates.

In the first model lowering the Repair effort estimate also increases the value of the value estimate. Since RpE is equal to technical debt we can see that more efficient software approach such as DDD results in increased value estimates of software asset.

In the second model lowering the Rework fraction RF increased the value estimate. This means that if more optimized source code is used smaller part of it will have to be reworked in order to increase its quality [22].

Finally, in the third model it is even suggested that if more efficient software development approach is adopted in later stages of software development life cycle (SDLC) it may partially improve software value of the system, as the technical interest will be discounting the rebuild value RV at a lower rate.

All of the described models can be applied to complex software systems that are composed of various development frameworks, programing paradigms and languages, database frameworks and technologies. Interconnecting this type of complex systems generates substantial additional development and maintenance costs.

If these connections can be negotiated from a single centralized programing concept represented by a unified model of the complete system the effort required to maintain the system would decrease. This is why the approach to complex software system using domain driven design may effectively influence the value of complex systems and software assets. This influence can be observed during the early development stages, but also during later stages i.e. during the production stage and maintenance of the system.

As we described earlier, DDD is focused on describing the domain. For complex systems (such as business software systems) this means that only business processes have to be described without the concern with technical details.

Business experts can communicate their understanding of business processes to system development teams using a unified ubiquitous language that also represents the formal specifications of the system. In the end, model represents the business domain at hand, with no regard to what part of the complex system it refers to (particular functionalities, external systems and data sources or databases).

Further tools that draw on DDD approach can use this formal descriptions and using compilers dedicated to particular properties of the model create system components in a flexible and yet automated way, producing optimized and maintainable source code resulting with increased software quality.

Particularly, tool DSL Platform contains a number of compilers that translate the source code of the DDD model into different segments of coupled complex heterogeneous software systems, building on top of various frameworks, languages, libraries and platforms. In this way it synchronizes the complete systems and migrates data between database and the model and vice versa. Workload for the development team is alleviated so that team members can spend more time on designing the domain model itself in cooperation with business experts.

Benefits from moving the focus of the development team form technical issues to business logic, as well as the improvement of the communication between team members improves the quality of software systems developed. Additional saving obtained through lower maintenance cost and increased quality of source code through better performance of execution and improved manageability of code can significantly improve the value of complex business software systems. However, DDD does not seem to be widely spread and accepted in practice.

B. Implementation obstacles and limitations for Software Management

The disadvantage of introducing DDD in software development is the additional effort required to adopt this software development approach. As software system grows alternative software development approaches usually tend to increase maintenance cost and decrease quality of code and the system gradually degrades. With software system growth DDD establishes better management over the complexity of system with little degradation of system quality making initial entry cost feasible. Also, additional effort and time is needed to create a substantial model of the business domain before positive effects on the development process become apparent.

In order to verify the findings in this paper, several interviews were conducted with various team members from two software development companies and two financial institutions that develop their own software solutions. Based on the responses gathered during interviews SWOT analysis was conducted. Results are given in Figure 14.

The advantages were concluded based on the evidence described in this paper while the disadvantages needed further

assessment and data collection obtained through interviews. Interviews were largely used to identify weaknesses and threats of adoption DDD approach for development and maintenance of complex business systems.

As we can see in Figure 14 strengths refer to core advantages of DDD with high emphasis on software management issues and especially business management aspects of software management, such as focus on business logic, unifying business domain for all team members regardless of their background and benefits in software quality and, particularly important for in-house development, increased software asset value.

On the other hand weaknesses of adopting DDD pertain to initial cost of adopting this approach as well as the risk of overestimating final system complexity as DDD is highly cost inefficient for simple software system.

The most important weakness is the current state top management awareness which represent the main limitation to wider adoption of this approach. The highest benefits can be achieved in large-scale non-software companies that develop in-house software solutions, such as financial institutions and banks, where the focus of core business is not on software development. These are also the companies where awareness and understanding of potential benefits seems to be at a comparatively low level as well as the priority in managing software development approaches. The main obstacle preventing the higher acceptance of the domain driven design in practice is the lack of understanding the benefits of DDD and potential tools it provides by top level management. As the bottom-line in risk management is to prevent potential risks, additional adjustments of value estimations of software systems does not justify adoption of DDD in companies that were interviewed. Additionally, successful adoption requires business domain experts to adjust to the domain specific language which is characterized by high level of isolation and encapsulation which is more familiar to software experts.

SWOT matrix	advantages	disadvantages
Internal	STRENGHTS • better team communication • focus on business logic • automation of particular development & maintenance tasks • unified domain model • increased level of quality • increased software value	WEAKNESESS • high entry costs • cost inefficiency for simple software systems • top management resistance • high level of isolation and encapsulation in domain model may present a challenge for business domain experts
External	OPPORUNITIES • improved estimation of value for developed software assets • reduction of maintenance costs during production phase of software system • prolonged lifespan of software systems • sustaining business logic of legacy systems	THREATS • incentive to maintain legacy technologies and programming languages while maintaining high software value • as changes in domain model are reflected in system components risk of human error increases

Fig. 14 SWOT analysis of DDD approach to complex business software systems

External elements of the SWOT analysis describe the potentials of adopting DDD where positive potentials represent opportunities to be gained. As we can see in Figure 6 improved valuations of software assets can be achieved and in turn promote better strategic decision making. Also, reduction of maintenance cost during production phase improves internal rate of return on investment while at the same time extending the lifespan of software asset. Equally important is the potential of preserving business logic in legacy systems which would be otherwise either lost after the discontinuation of legacy systems or retained through expensive process of reengineering.

Prolonged lifespan may also lead to one of two most important threats in adopting DDD. This is the incentive to maintain legacy systems that rely on old technologies, programing languages, paradigms or frameworks while maintaining high software asset value which may expose the company to additional risks such as self-exclusion from trends in software developments and increase of inefficiency resulting in loss of competitive advantages. Additional threat that can be detected is the possible increase of the importance of human error factors since the software model is directly related to the system itself, so that any change is readily implemented in software components in the production phase.

VI. Conclusion

Domain driven design is a more recent approach to software development that fundamentally changes key aspects in software development by changing the nature of the relationship between a model and the final product. It moves the focus from methodologies, tools and project management to the core of software system being developed and the expert domain it will engage with. This radical perspective of domain driven design while it offers substantial benefits, stayed out of mainstream implementations in practice and also limited sources in academic and scientific literature is available.

In this paper we have presented key determinants of domain driven design (DDD) and assessed its implications on software management process through impact on software value estimation and changes in maintenance efficiency. While overviewing recent classification of software approaches, we have detected a missing approach aimed at fast and efficient development and maintenance of coupled complex software systems. It is author's opinion that DDD is the missing approach best suited for this type of software projects.

In order to support this thesis further comparison with behavioral approaches was conducted and analyzed. Recognizing the new role of modelling we have confirmed two major points that DDD copes well with. Firstly, it improves team communication while increasing the speed of modelling and developing system prototype since the domain code itself is model but also a representation of final software system. Consequently and secondly this allows for the implementation of number of automation tasks in developing and maintaining final software system. This is one of major reasons why DDD

approach improves the efficiency and reduces cost of development of coupled complex systems which development heavily depends on knowledge of multiple domain experts and also provides similar benefits during the maintenance stage of the software system. In order to better understand the main principles of DDD a broader introduction to a specific implementation of this software approach was given. Resulting tool is therefore considered an unique platform that encompasses tools, methodologies and technologies. This tool is called DSL Platform and is primarily based on a domain specific language development. Once domain is modeled transition of the model into working code is provided by the platform and its serialization capabilities. Recent benchmarks, presented in this paper show high performance of this tool that are comparable to other leading serialzators, and even the best XML/JSON serializer available.

As technical properties of DSL Platform exhibit such positive properties non-technical circumstances should be also evaluated As this approach is still to see its wider adoption in practice.

For the purpose of this paper we took two main benefits from DDD describing their practical implementations through an existing tool DSL Platform. We estimated the impact of these features on two major issues in software management – software value estimation and maintenance cost effectiveness. We have shown that due to specific properties of software products, and from the economic point of view - software assets, level of quality of software can be greatly improved. Various development issues may benefit from the implementation of DDD approach, either technical issues, process issues, people issues, project issues or more general class of holistic issues during software development. Similar effects can be observed during the maintenance stage of software development production, where greater focus is on the decision process whether to continue maintenance or discontinue the system use.

In this paper we have observed that all of the changes in the software development life cycle somehow reflect on the value of software system at hand. This is why we suggest that software value estimation plays an important role in assessing benefits from adoption of DDD tools such as DSL Platform. We have therefore presented a software asset value estimation models and analyzed how the estimation value changes under influence of DDD approach.

Finally we have conducted interviews with information officers and managers in software companies and banks to obtain data and create a SWOT analysis of adopting DDD in companies that manage in-house complex heterogeneous software assets. The analysis showed that main obstacle for adoption of DDD is lack of understanding the economic benefits by the top management.

This is an important confirmation of current limitations to adoption of DDD in mainstream software industry and software departments of large companies that should be taken into account. In further studies top management should be taken into consideration while assessing the applicability of DDD approach in practice. Additional benchmarks and research results should be made available to top managers in order to provide them with adequate information while making decision on adopting DDD approach for their software development needs.

ACKNOWLEDGMENT

I would like to thank Rikard Pavelic and company Nova Generacija Softvera d.o.o. for their cooperation during research of this topic, donating free access to DSL Platform (http://dsl-platform.com) for the purpose of evaluation and invaluable information that improved the quality of the research results presented in this paper.

REFERENCES

[1] N. Mavetra and J. Kroeze, "Guiding Principles for Developing Adaptive Software Products" in *Communications of IBIMA*, vol. 2010, IBIMA Publishing, 2010, pp. 1 – 15.

[2] J. de Groot, A. Nugroho, T. Back and J. Visser, "What is the value of your software?" in *Proceedings of the Third International Workshop on Managing Technical Debt (MTD), 5th June 2012*, Zurich: IEEE, 2012, pp. 37–44.

[3] N. R. Jennings, "On Agent-based Software Engineering" in *Artificial Intelligence, vol. 117*, Elsevier Science, B.V., 2000, pp. 277 – 296.

[4] D. Sharma, W. Ma, D. Tran and M. Anderson, "A Novel Approach to Programming: Agent Based Software Engineering" in *Knowslege-based Intelligent Information and Engineering Systems, Lecture Notes in Computer Science*, vol. 4253, Berlin: Springer Verlag, 2006, pp. 1184 – 1191.

[5] D. North, "Behavior Modification: The evolution of behavior-driven development", in *Better Software*, vol.-issue 2006-03, Techwell Corp.

[6] R. Brown, S. Nerur and C. Slinkman, "The philosophical Shifts in Software Development" in *Proceedings in the 10th Americas Conference on Information Systems*, New York, August 2004, pp. 4136 – 4143.

[7] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*, Addison-Wesley, 2004.

[8] G. Booch, J. Rumbaugh and I. Jacobson, *The Unified Modelling Language User Guide, 2nd Ed.*, Addison-Wesley, 2005.

[9] J. S. Cuadrado and J. G. Molina, "Building Domain-Specific Languages for Model-Driven Development" in *IEEE Software, vol. 24, Issue No. 5*. IEEE Computer Society, September/October 2007, pp. 48 – 55.

[10] R. J. Wirfs-Brock, "Driven to… Discovering Your Design Values" in *IEEE Software, vol. 24, Issue No. 1*. IEEE Computer Society, January/February 2007, pp. 9 – 11.

[11] S. Y. Choi, D. O. Stahl and A. B. Whinston, *The economics of electronic commerce: the essential of doing business in the electronic marketplace*. Indianapolis: Macmillan, 1997.

[12] S. Lehmann and P. Buxmann, "Pricing Strategies of Software Vendors" in *Business & Information Systems Engineering*, vol. 6, Heidelsberg: Springer Verlag, 2009, pp. 452 – 462.

[13] J. Zhang and A. Seidmann, "The optimal software licencing policy under quality uncertainty", in *The Proceedings of the 5th international conference on electronic commerce*, New York: ACM Press, 2003, pp. 276–286.

[14] S. Royer, *Strategic Management and Online Selling: Creating competitive advantage with intangible web goods*, New York: Routlege, 2005.

[15] C. Homburg and H. Krohmer, *Marketing Managment: Strategy – Instruments – Implementation – Governance, 2nd Ed. (in German)*, Wiesbaden: Gebler, 2006.

[16] W. Cunningham, "The WyCash portfolio management system," *ACM SIGPLAN OOPS Messenger*, vol. 4, no. 2, 1993., pp. 29–30.

[17] A. Nugroho, J. Visser, and T. Kuipers, "An empirical model of technical debt and interest," in *Proceeding of the 2nd International Workshop on Managing Technical Debt.*, ACM, 2011, pp. 1–8.

[18] B. Curtis, J. Sappidi and A. Szynkarski, "Estimating the Size, Cost, and Types of Technical Debt", in *The Proceedings of the International Workshop on Managing Technical Debt*, 2012, Zurich, Switzerland.
[19] F. Dumitriu, D. Oprea, and G. Mesnita, "Issues and Strategy for Agile Global Software Development Adoption", in *Recent researches in Applied Economics*, WSEAS, 2011, pp. 37-42.
[20] A. Spiteri, "Modeling UML software design patterns using fundamental modeling concepts (FMC)", in ECC'08 Proceedings of the 2nd conference on European computing conference, WSEAS, 2008., pp 192-197.
[21] A. Bassam Al-Badareen, Z. Muda, M. A. Jabar, J. Din, S. Turaev, "Software quality evaluation through maintenance processes", in: *Proceedings of the European conference of systems, and European conference of circuits technology and devices, and European conference of communications, and European conference on Computer science*, WSEAS, 2010, pp. 131-134.
[22] Lj. Lazic, A. Kolasinac and D. Avdic. "The software quality economics model for software project optimization", WSEAS Transactions on Computers, Vol. 8, Issue. 1 (January 2009), WSEAS Press, 2009, pp.21-47.