Implications of Domain-driven Design in Complex Software Value Estimation and Maintenance using DSL Platform

Nikola Vlahovic

***Abstract*—**The introduction of the domain-driven design (DDD) as an alternative approach to software development had the promise of achieving several benefits in the process of creating complex domain-specific business applications. Due to the focus of this approach to the core of the application functionality, improved collaboration with domain experts and conceptual modeling benefits, it has attracted a reasonable amount of attention from the programming community in the past decade. Aforementioned benefits have also been able to create unique set of programing environments and languages that also move the boundaries of efficiency of code execution and application maintenance.

In this paper we will present and analyze one such tool, namely, DSL platform. DSL platform is a service that allows for the design, creation and maintenance of business applications. The goal of this paper is to analyze the implications of using the DDD through the DSL platform on several important aspects of software management. Primarily we will focus on the estimation of complex software system value and software refactoring and maintenance effort based on the models proposed by Groot et al.

We will show that for complex software systems consisting of a number of different components, programming paradigms and database systems can highly benefit from this approach. Some of the most important benefits pertain to lowering of the cost of software maintenance and transcending the properties of reliable business applications and databases developed using legacy systems to current systems using the underlying domain model.

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

I. Introduction

THERE is a reasonably limited number of papers is current scientific literature pertaining to different particularities of software development management and practices, such as software pricing model practices or adoption of novel software development approaches. Only in recent years overviews of some of these aspects of software management have been studied and new models have been proposed. At the same time practitioners are developing and presenting new frameworks and technologies as well as new approaches to software development altogether. Only a limited number of these developments enter the mainstream adoption by software or even non-software companies.

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).

One such phenomenon is software development approach called domain driven design. This approach tries to offer solutions to bridging the gap between business experts and software experts that is main drawback in traditional approaches that decreases the success rate of many software projects. 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.

Domain driven design is therefore dedicated in improving the development and maintenance efficiency in complex business systems. While offering great benefits for this type of software systems and additional improvements in various aspects of software management, it still faces significant obstacles to adoption.

In this paper we will explain and present the main concepts of domain driven design as an adequate software development approach for complex business systems. Implementing these benefits will be given through description of one particular implementation of the approach, a software development tool called DSL Platform. Benefits can be critically assessed through different software management issues, and in this paper we will concentrate on estimation of software asset value and maintenance of these assets.

Goal of this paper is to critically investigate possible benefits of adopting domain driven design in software management, with particular emphasis on maintenance during production phase of software assets. Inevitably these considerations will reflect on the value of software asset, so a validated approach to estimation of software assets is called for. Here we will build on a proposed model of software valuation proposed by [2].

The structure of this paper is as follows: In Section II we will describe the main features of domain driven design, its advantages and disadvantages as well as the implementation of its concepts in a tool called DSL Platform. In Section III we will take a closer look at some of the most important software management issues that can be affected by the domain driven design approach, such as the software maintenance issues, software risk managements and software value assessments approaches. In Section IV we will discuss the possible impacts of applying domain driven design within the software development process for complex coupled heterogeneous systems throughout the software process life cycle and 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 with a few guidelines outlining the main advances DDD and DSL Platform can provide for companies using complex heterogeneous software systems.

II. Domain driven design and DSL Platform

In this section we discuss the domain driven design as a type of software development approach, position this approach in a wider context and based on our analysis describe a tool that implements these features in most consistent manner.

Domain driven design (DDD) is a software development approach that rather than analytically organize the software development effort and use conceptual, modeling, programming and implementation tools, it tries 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]:

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 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].

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. As it is platform-independent it is an encompassing approach to highly coupled systems that use different, even inconsistent, technologies and platforms as well as development methodologies or practices.

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. 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.

Unlike UML that takes on a number of perspectives of the model, DDD tries to describe the model by describing its

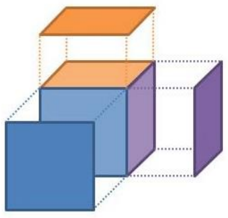

Fig. 1 model and perspectives of the model

domain as a whole and complete model (Figure 1). In this way, model itself represents the system being developed. Consequentially, programming code is the representation of the model. 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 not 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, and 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 in 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].

Implementing key features of the DDD using object oriented design can be used to create a unified platform for development and evolution of complex software systems. One such tool is DSL Platform which we will describe in the rest of this Section.

DSL Platform is a service that helps in designing, building and maintaining business applications. It allows for the automation of business application development process. The platform uses the 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 2).

Once declarative specification is defined, any of the supported compilers can use this specification to build code or

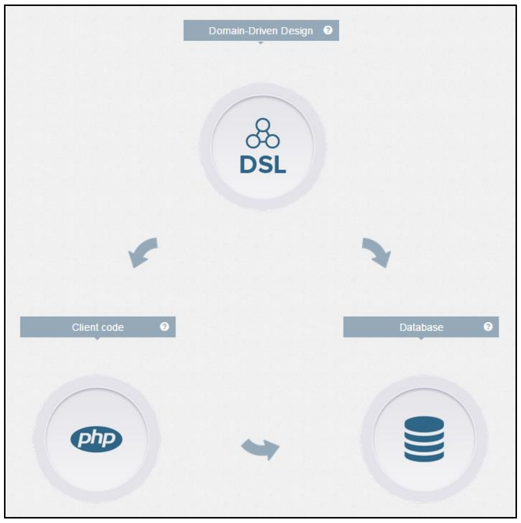

Fig. 2 DSL Platform concept

maintain databases. 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 2). 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. Similarly the maintenance or even migration of data to the underlying database system is also highly automated.

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. *Tackling miscommunication*

In each software project there is a number of different stakeholders that 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. *Improving efficiency of source code and automation*

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.

III. ESTIMATION OF SOFTWARE VALUE

In this Section we discuss the requirement and motivation for precise estimation of software value and describe one of novel concepts to strategically determining the value of software assets.

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.

In software industry this is the case with software assets. Majority of 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.

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.

All of the above 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.

A. 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 3).

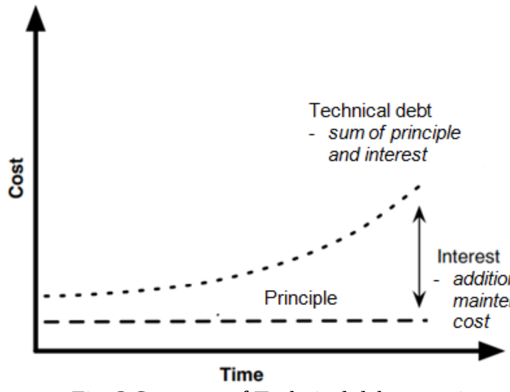

Fig. 3 Structure of Technical debt over time

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 4 we can see that future benefits from software system operating at the ideal level of quality yielding significant savings.

In order to include technical debt in the estimation of

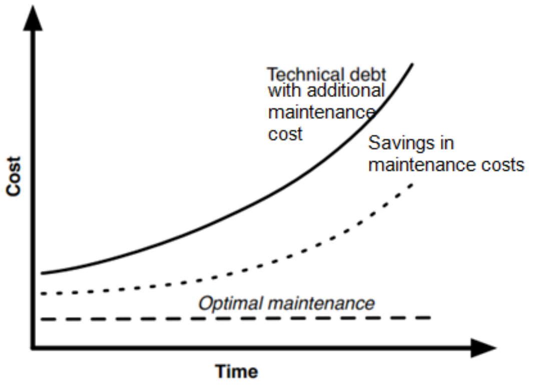

Fig. 4 Benefits from maintaining software system at the ideal level of quality

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 5).

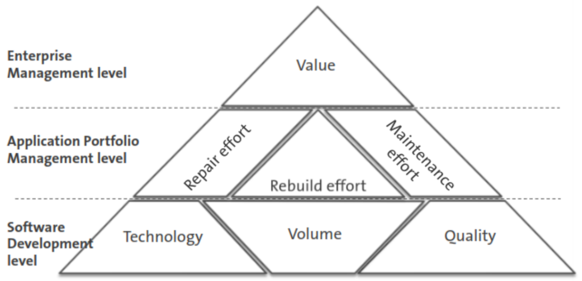
Enterprise Management level
Value
Application Portfolio Management level
Repair effort
Rebuild effort
Maintenance effort
Software Development level
Technology
Volume
Quality

Fig. 5 Software Valuation Pyramid (Groot et al, 2012)

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.

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

B. 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].

IV. Analysis of software management practices and domain driven design

A. Relating Software Asset Value Estimation and Software Development Approach

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 choses software approach to software development life cycle (SDLC).

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.

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.

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.

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. Investigating DDD Adoption Limitations in Software Management Practices

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 6.

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. 6 SWOT analysis of DDD approach to complex business software systems

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 6 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.

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.

V. Conclusion

In this paper we have presented one of more recent approaches to software development called domain driven design (DDD). We assessed its implications on software management process through impact on software value estimation and changes in maintenance efficiency. As this approach is still to see its wider adoption in practice we first took a look at its main characteristics and, building on current research, position it according to recent classifications. By comparison with other approaches we classified DDD to an intermediary group between structured approaches and behavioral approaches. In fact, DDD seems to have been the missing link since the intermediary class only recognized a class of methodologies based on agile software development concerned with small and medium projects. DDD completes the classification as it is intended for complex heterogeneous software systems.

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 level of quality of software can be greatly improved during development phase through better communication and moving focus from technical to business arena. During the production phase of software system higher quality of code optimizes maintenance cost in comparison to suboptimal software system quality. All of this is reflected through software asset value. We have shown building on software valuation models presented by Groot et al (2012) how the changes DDD provides impact all of the three proposed models of software valuation.

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 limitation to adoption of DDD in mainstream software industry and software departments of large companies that should be taken into account when communicating research information to business users and management.

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.