2014 47th Hawaii International Conference on System Science

Dual Application Model for Agile Software Engineering

Ashley Aitken
School of Information Systems
Curtin University of Technology
A.Aitken@Curtin.Edu.Au

Abstract

There are problems with Traditional Software Engineering and with Agile Software Development. A new approach called Agile Software Engineering that combines the best of both is proposed and an aspect of this approach described. The Dual Application Model involves the development of a logical software application focused on capturing the functional requirements and a physical software application focused on transforming the logical application to meet the non-functional requirements. It has advantages and disadvantages like any approach to software development but meets two of the principles proposed for Agile Software Engineering. Frameworks and tools are proposed to support the Dual Application Model but are not essential to the approach. The approach provides an almost complete separation of concerns between defining and specifying (in code) the domain solution / software problem for which the domain experts are primarily responsible and designing and implementing the software solution to meet the non-functional requirements for which the software developers are primarily responsible.

1. Introduction

Agile Software Development (ASD) has gone mainstream at 35% and dominates Traditional Software Engineering (TSE) at 21% and Waterfall Software Development (WSD) at 13%, according to a global developer survey, as reported in [1]. This doesn't mean that everything in TSE (or even WSD) was wrong and everything in ASD is right. In fact, there is a risk in such a dramatic move from TSE to ASD of throwing the baby out with the bath water. It is important to transfer to ASD (perhaps with some reinterpretation) those concepts and practices from TSE (and even WSD) that have been found to be effective and not to just discount them because they are traditional (just as new concepts and practices shouldn't be adopted just because they are new).

The overall objective of this research programme is to develop a new approach to software development that combines the best of TSE with the best of ASD (and even WSD). This paper represents a first step in this research. It discusses the context within which this new approach was first developed, explains the new approach, including some of the theoretical and practical advantages and disadvantages, and discusses areas of further research (e.g. data supporting the efficacy of the approach) and development to support the new approach (e.g. development tools, class libraries, and frameworks).

Software models, like all models, are defined by their scope – the breadth of their representation, their perspective – the particular view of the software, and their level of abstraction – the amount of detail or complexity is included in the model. There are also many different representations for models (e.g. physical, textual, graphical, mathematical, and computational models), each with their different characteristics, advantages and disadvantages.

Software development is all about modeling. Software models include textual models (e.g. user stories, use-cases and requirements documents), graphical models (e.g. class diagrams, sequence diagrams), and other models (e.g. mathematical models). An important software model that is often overlooked, whilst right in front of everyone's eyes, is the source code. The source (and executable) code for an application is another model, it just happens to be one that can be executed (i.e. can be run directly or interpreted) and the primary goal of software development (i.e. to build a software application).

Software models can represent the (mostly) static relationships between entities within the software, somewhat analogous to a photograph. Examples of models of static aspects are the UML Class Diagrams and Deployment Diagrams [2]. Software models can also represent the dynamics within and between entities within the software, somewhat analogous to a movie. Examples of models of the dynamics are the UML Sequence and Interaction Diagrams.

Software development is problem solving. As a result, all software development, no matter which approach or methodology is used, goes through the same cycle of requirements, analysis, design and implementation, independent of what form each takes or what models are (or are not) constructed. WSD

978-1-4799-2504-9/14 $31.00 © 2014 IEEE
DOI 10.1109/HICSS.2014.588

aims to do this only once, whereas best practice for TSE and ASD aims for highly iterative and incremental development. Any source code that is written has some design, analysis and requirements that directs its development and provides context, even if it's all done within developers' heads.

The next section of this paper discusses some of the problems with Traditional Software Engineering and Agile Software Development and a solution based on the best of both called Agile Software Engineering. The following section introduces the Dual Application Model as an approach to software development that is compatible with Agile Software Engineering. The paper finishes with further discussion of this approach, including related work, answers to common questions, possible tools or frameworks, and future research directions.

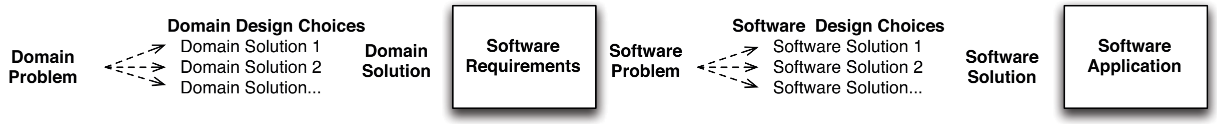

Figure 1. Domain Problem-Solution versus Software Problem-Solution

2. Agile Software Engineering

Traditional Software Engineering (TSE) is defined here as the best practice software engineering that was undertaken before agile software development came along. It was planned, highly iterative and incremental, and employed many models of the software system being developed, as exemplified by the the Rational Unified Process [3]. Agile Software Development (ASD) is defined here by the Agile Manifesto [4] and tends to be highly iterative and incremental but adaptive rather than planned and generally embraces less formal processes and fewer persistent models (e.g. primarily the code) and formalities than TSE [5].

2.1. Analysis versus Design

Analysis is the process of understanding the domain and the software problem itself. Here the word "problem" is used in the sense of the task to be undertaken, e.g. to develop a software system to do specified information processing and/or storage in a particular domain meeting specified constraints. Analysis involves modeling the problem independent of any possible final solution, in particular independent of any non-functional requirements and any decisions made about the solution. Analysis or logical models are models of the problem (not just a general model of the problem domain).

Design is the process of determining and specifying a solution to the problem with specific implementation technology that meets the non-functional requirements. In essence, if there were no non-functional requirements or implementation constraints there would be no need for design. In software development, the analysis model is in effect a solution meeting the functional requirements but not the non-functional requirements or implementation constraints. The design or physical models are models of a solution modified to meet the non-functional requirements using the chosen implementation technologies [6].

Software development is initially driven by a domain problem (e.g. in a business or technical area). However, solving the domain problem is, almost by definition, the task of domain experts. It is not the role of software developers to determine how a business should solve its business problems, even information processing problems, or to find the solution to some technical business problem (although it seems, at least in business, this is often the case).

For any domain problem there is a domain solution that is chosen by the domain experts from a set of possible domain designs to meet the functional and non-functional requirements within their domain. The task for software developers is to take this domain solution on as a software development problem with its own non-functional requirements and find the most appropriate software solution from a number of possible software designs. The domain solution is in essence the functional requirements and analysis or logical models, a complete specification of what the software system should do.

Whilst is possible to use automated tools to create different representations of the logical models (e.g. using CASE tools) and to create different representations of the physical models separately, it is not generally possible to use tools to automate (at least, not completely) the transformation from logical models to physical models or vice-versa. This is because this is where creative design happens, tradeoffs are made, solutions to meet non-functional requirements are determined, and there is not yet any way to fully automate these decisions (or the search through the solution space).

In summary, software analysis is about understanding the functional requirements as presented or developed in logical model(s) with the help of the domain expert(s). For software this can be done, for example, with user stories, use-cases or requirements documents and various other analysis models. Software design is then about modifying the logical model(s) to form physical models and to ensure the results meet the non-functional requirements with the chosen implementation technologies.

For software development the non-functional requirements can be characteristics like speed, reliability, ease-of-use, scalability, and persistence. As a result of these different non-functional requirements we have different roles and expertise within software development like algorithm designer, UI designer, persistence designer, and the designer of the software architecture (upfront or incrementally as the system is developed). Software analysis is about understanding and modeling (if these are not already provided) the domain solution, and software design is about finding a software realisation of the domain solution that meets the non-functional requirements.

The benefits of the analysis / logical models are: 1) that they focus purely on the software problem / domain solution, 2) they are independent of any decisions related to the software solution (e.g. architecture or technology decisions), and 3) they are independent of any non-functional requirements (e.g. they do not have to concern themselves with meeting non-functional requirements) . Inevitably, there must be a logical model of the problem, even if it is only partial and/or temporarily maintained in the heads of the software developers.

2.2. The Problems with Traditional Software Engineering and Agile Software Development

One of the main strengths of TSE was, in the author's opinion, its multi-model approach to software development. Having models that give different perspectives on the software at different stages in the lifecycle allowed developers to focus on particular aspects of the software development task (e.g. the requirements, the analysis, the design, and the implementation). As engineers in other disciplines also use multiple models (e.g. of buildings and roads) there is strong reason to think of this as a key aspect of an engineering approach to software development.

The biggest practical problem with traditional software engineering relevant to this paper was not the waterfall approach (as many agile software developers informally argue) since best practice traditional software engineering was highly incremental and iterative. The biggest problem was the fact that the various models developed (e.g. requirements, analysis, design, and code) quickly became out-of-sync with each other, particularly when developers changed the code without updating the other models.

The requirements (combining functional and non-functional requirements) and analysis (only functional requirements) models were also most often a collection of static and dynamic textual, graphical, and sometime mathematical models. The static textual and graphical models could quickly become very complex and difficult to understand, and the dynamic textual and graphical models had trouble capturing the full dynamical nature of the domain solution. These models were also very hard to verify and validate, because of their form or the fact that they were partial and incomplete.

One of the main strengths of Agile Software Development (ASD) is the incremental development of the software, without detailed overall plans, or even detailed requirements specifications. The software development work is often done one (vertical) piece at a time to create incremental value for the users and stakeholders. The quickest way to achieve this ASD suggest is by going directly to the source code model with minimal and often only transient forms of other modeling. ASD employs, generally speaking, even less than "just-in-time" modeling. Most ASD models (except the source code) are considered only temporary artifacts and often only for momentary communication between developers.

Of course, ASD is a very broad church [A, so it is difficult to claim it has a specific problem (since there will often be a form(s) of ASD that don't have that problem). That said, the problem that is most relevant to this research and commonly found in ASD is the use of the source code model as the primary artifact and form of documentation for everything related to the software development. The source code does not only include the functional source code but also unit testing, test harness, and other supporting code and comments within the code. It seems reasonable to suggest that it is not optimal to include all documentation and alternative models, including logical models, within the physical model code (i.e. the final solution).

The ASD code does capture most (if not all) of the physical aspects of the solution. For example, implicit in the code is the architecture of the final application, the physical design of each API, and the optimized algorithms and data structures for each module. However, by their very nature these are solution specific and implementation specific. They are after all the source code that meets the functional and non-functional requirements in the chosen implementation technologies. The problem is that the logical model is generally not recorded because there is no easy mapping between logical and physical models, and thus lost when the focus is only on the physical source code.

Some snippets of the logical model may be saved as comments in code or separate documentation (e.g. a user story, a diagram here, or a photo of a whiteboard sequence diagram there). However, this separate documentation suffers from the same problems that plague traditional documentation, i.e. that it can be come out-of-date and out-of-sync very quickly if the solution changes. Most significantly though, if other developers, e.g. those allocated the job of maintaining the source code, want to get a real understanding for the problem domain and domain solution they have to attempt to see it within the code through all the modifications made to meet the non-functional requirement and the technology chosen for the implementation. The goal of the physical source code is to meet the non-functional requirements rather than to clearly express the logical model.

2.3. Agile Software Engineering

Agile Software Engineering (ASE) aims to combine the best of TSE and ASD and overcome the problems with each. It adopts the highly iterative and incremental approach of TSE and ASD, and most often the adaptive approach of ASD rather than the planned approach of TSE. ASE adopts the multi-model view of TSE against the limited and primarily physical modeling approach of ASD. Its approach to the problem of out-of-sync models is primarily to eliminate all permanent models of the software except the source code model (or what can be represented in the source code model). ASE's approach to the out-of-sync models problem is to encourage the maintenance of multiple models and support tools or automated generation of models separately within the logical or physical spaces so that they can be kept in sync.

This paper focuses on the problems mentioned above, i.e. in TSE of the requirements, analysis, and design models becoming out-of-sync and the lack of analysis or logical models in ASD. How can a development approach have multiple models, with not just physical models, that do not (or at least should not) become out-of-sync? This paper also does not provide a complete description of ASE, it is still a work in progress, but it suggests here a couple of the general principles of ASE in line with these identified problems:

> ***Principle of Appropriate Models:*** *This principle states that the most appropriate place to work on logical aspects of a problem is in logical models of the problem and the most appropriate place to work on physical aspects of a software solution are in the physical models.*

This may seem obvious but it is not what is generally done in ASD. As mentioned, ASD tends to work primarily on the source code model, i.e. a physical model for the system. Any logical models are either implicit in the physical model or temporary and transient models discarded once the physical model is produced.

> ***Principle of Only Forward Engineering:*** *This principle, a corollary to the previous one, states that it is inappropriate to reverse engineer or work back from physical models back to logical models. Doing so is, generally, a violation of the previous principle even if possible.*

Of course, the latter doesn't mean you cannot reverse engineer if/when it is possible and necessary, for example if you receive code without any logical models. However, the aim is to not encourage developers to start with the traditional source code (i.e. a physical model) and then try to work back to logical models if needed since this is often very difficult, if not impossible, to do completely.

3. The Dual Application Model

The goal of ASE may seem difficult to attain – a combination of TSE and ASD – that at the very least meets the principles specified above. In this section, an approach to software development called the Dual Application Model (DAM) is proposed that, at least partially, provides a step in that direction.

Before the Dual Application Model is explained in detail it should be said that it only relates to specific aspects of the approach to software development (in particular, which models are developed). For all other aspects of software development the approach assumes the best current practice in software development. For example, highly iterative development with Scrum [7] or a Kanban [8] approach to work management, the use of unit testing and acceptance test driven development. Although the paper discusses the development of two models it is definitely not suggesting that these will be developed in a waterfall fashion, each may be built up incrementally during each iteration just as the traditional source code model is developed in ASD, or that these will necessarily be the only models developed.

3.1. Overview of the Dual Application Model

The DAM proposes that software developers develop two applications instead of just the one that they will deploy. The first application, called the Logical Application (LA), implements a runnable logical model of the domain solution, / software problem. The second application, called the Physical Application (PA), implements the physical model of the desired software system, which is actually in fact the desired software system.

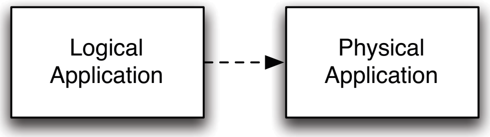

Figure 2. Logical and Physical Application

This DAM approach may sound crazy. Why should developers do twice the work to achieve the same outcome? The assumptions (that needs to be proven) are 1) that there will be some extra work but it won't be twice the work (as will be explained), and 2) that this approach has other advantages like higher quality and overall productivity that outweigh this extra work (whilst also being in line with the principles of Agile Software Engineering). It is proposed that the DAM is a way to achieve ASE without discarding the multiple models of TSE (i.e. without throwing the baby out with the bath water) and, it is claimed, will achieve better productivity and higher quality, primarily because of the separation of concerns.

In essence, this paper is suggesting that developers swap the traditional logical models (i.e. traditional declarative requirements specifications and analysis models) for a runnable LA, the development of which can be guided by user stories or use-cases). Recall the problems with declarative requirements and analysis models, is that they are often incomplete, inconsistent, difficult to validate, get out-of-sync with development. The LA should go along way to solving some of these problems – software abhors inconsistency, software can be exercised and tested to verify and determine validity, and incompleteness stands out in software, particularly when it is exercised and tested.

3.2. Logical Application (LA)

The LA is in essence a runnable logical model of the domain solution / software problem. Unlike traditional requirements or analysis models that were often incomplete, inconsistent, and piece-meal (i.e. only representing portions of the logical model), the LA can be a complete and detailed model of the software problem so far. It should also be a "natural" model of the problem since it is aiming to clearly capture the logical model. It does, by definition, lack any details pertaining to the physical implementation.

The goal of the LA is to capture the desired software system independent of any non-functional requirements or implementation technologies. The LA is a direct, explicit, and executable model of the domain solution. Having an executable logical model means the users can try out the logical model, that logical unit tests may be written, as well as logical system tests. These tests may then be mapped to physical unit and system tests.

Of course, any runnable software application must be implemented in some programming language and have some design, it is not possible to escape the "physicalness" of even a logical model. However, the design of the LA is chosen to best represent the domain solution and to be as easy and quick as possible to develop and change. The best way to do this is usually to match the domain solution and logical model as closely as possible. As a result, many LAs would be object-oriented because object orientation is a natural way to model many problem domains. However, other LAs could be written using functional or other styles of programming.

The physical constraints on the LA should be minimal. For example, it should be assumed that the application has a very large amount (but not infinite) memory, a very fast (but not infinitely fast) processor, and a very high (but not infinite) network bandwidth. The LA should not be concerned with persistence since that is a non-functional requirement (i.e. that the software shall be able to maintain state when the application is not running). It should not be concerned with the solution application architecture, e.g. whether it is a one tier, two-tier or n-tier application for Web, desktop, mobile or an embedded platform, since these are architecture and implementation constraints of the physical model.

The DAM does not constrain the technologies used to implement the LA. They should be whatever is best able assist the developer to as easily and as quickly as possible model the problem domain solution in a natural way. Contemporary programming languages that would seem to meet these requirements would be languages like Ruby, Groovy or Python, i.e. the dynamic programming languages because of their dynamic typing, brevity and rapid application development. This is not to say that the LA could not be developed in pretty much any programming language.

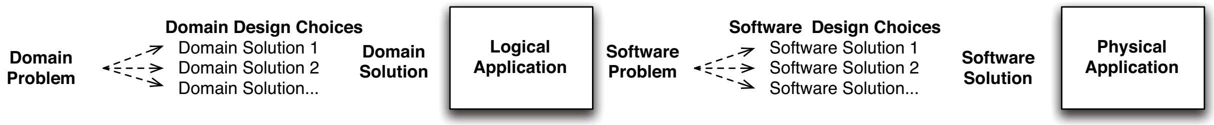

Figure 3. Logical Application becomes primary Requirements and Analysis Models

Whilst there may be some advantage of developing in LA in a programming language that can also be used for the PA it is not a requirement. It could be beneficial because it could allow the developer to copy some source code directly from the LA to the PA, at least initially. The LA can be considered a first draft for the PA [6]. The problem with using the same programming language for the LA and PA is that it could confuse developers with regard to which model they are working on. The temptation is also that developers may not develop a separate LA (since they believe it is just being copied to the PA anyway).

Recall that the benefit of the LA is not just that it can be used as a draft for the PA but also that even after the PA starts to be transformed into the physical application that will satisfy the non-functional requirements, the LA should still be a natural representation of the domain solution / software problem. Developing further user stories, use-cases, and functional requirements firstly within an unadulterated LA will be a lot easier than developing them in the PA, which will be more complex due to its persistence technology, UI design, and various optimizations.

3.3. Physical Application (PA)

The goal of the Physical Application (PA) is to firstly implement the functional requirements as represented in the LA and then the non-functional requirements of the software system. Developers of the PA have a clearly defined LA to work from. In effect, to implement the functional requirements the developer only need to translate / port (if necessary) the LA to the PA implementation and deployment environment. For example, logical types in the LA need to become physical types in the PA. If the LA and PA use the same or "compatible" languages at least this step could be a relatively easy and perhaps even a semi- or fully-automated process.

The full design of the PA is the primary task of software developers. The LA, which can become the first draft of the PA for software developers, should come almost for free from the domain experts and product owner. The PA then needs to be modified and changed until the PA meets the non-functional requirements. Aspects of the PA that could need to determined and changed include the architecture, the presentation and the persistence.

The PA is, of course, familiar to software developers since it is what they normally deliver. There is one significant difference, however, between development of the PA and the traditional executable that developers deliver. When developing the PA developers do not have to worry about defining the software problem in any way. They can work directly with what is specified in the LA. At this stage the developer is free to focus on optimizing the user interface, perfecting the persistence, scaling the architecture, … all to ensure the final deliverables meet the non-functional requirements. In these areas the developer is the expert (not the product owner), although of course the product owner and users have a say in the outcome through setting the non-functional requirements and feedback on incremental versions of the PA.

Key to the development of the PA will be a list of changes made to the LA that accompany the user stories, use-cases, and/or functional requirements for this particular iteration. As the LA is source code it can be kept in a version control system (e.g. SVN or Git) and changes can be tracked. The PA developer(s) will thus have a description of the functionality with source code changes that already implement this functionality in the LA. The PA developer(s) will also have a record of the non-functional requirements in some form, generally for the entire software solution or specifically for these user stories, use-cases, or functional requirements.

The PA developer(s) will then use their expertise to translate those changes (additions, modifications, and deletions) from the LA to the PA and implement the any required user interface, persistence, architecture or other changes necessary to meet the non-functional requirements. It seems possible that with a well-designed PA, many developers could actually work on this PA at the same time, e.g. UI specialists could work on the user interface, persistence specialists could work on the persistence, and others could translate the LA code and integrate with the other changes. Some changes to the LA may be superficial, e.g. renaming of variables or modules, and can perhaps be ignored by the PA developer. However, such changes are often a form of documentation, done to convey meaning and to correct or guide interpretations so perhaps they are important for the PA developer to consider as well.

3.4. Advantages and Disadvantages

Some possible advantages of the DAM include:

1. The LA clearly and as naturally as possible captures the static and dynamic nature of the desired software system, i.e. the problem domain solution and the software problem, and the PA captures the static and dynamic nature of the desired software solution (since it is the desired software system).
2. The LA works out any inconsistencies, incorrectness, incompleteness in the logical model before the more difficult PA is further developed. It is significantly more effective and efficient to find problems early in the software development lifecycle [9].
3. The DAM offers nearly complete separation of the analysis and design (as defined earlier), which should improve the results of each task. When developing the LA the developers focus primarily on capturing and defining the problem as related by the product owner and/or domain expert(s) and are not distracted by other concerns. When developing the PA developers focus primarily on implementing the provided parts of the LA in the chosen implementation technologies and designing the solution to meet the non-functional requirements and not about the domain solution or software problem.

Some possible disadvantages of DAM include:

1. The development of the LA and PA seems to involve more work than just the development of the PA. Although, as mentioned above some of this work may be "copied" over, at least initially, to the PA. It is suggested that any extra effort in developing this extra model has significant advantages and payoffs in the short to medium term for the software development.
2. The product owner needs to work with both the LA to check its functionality and the PA to check it meets the non-functional requirements. They may also be confused somewhat by the two different applications and hesitant to go back to work on the LA when the PA is available.
3. There is nothing physically stopping developers from implementing new functionality directly in the PA without implementing it firstly in the LA. This may be "strongly discouraged" by management (or technology) but it is hoped that the advantage of using the LA will become clear to developers and they will choose to develop functionality in it first.

Further, and as discussed earlier, it is not possible to directly generate the PA from the LA (or reverse engineer the other way), although some automation may be possible. The reason is that design is about making choices between alternatives and tradeoffs between these alternatives, e.g. the different user interface options, the different persistence options, the different architecture options, and different algorithm design options. So it is generally not possible for tools to perform the LA to PA transition.

However, it is entirely possible to uses tools to automate generation of various logical models from other logical models and various physical models from other physical models. For example, it is possible to use UML graphical analysis models and a CASE tool to generate (a skeleton of) the LA source code and to reverse engineer the UML graphical analysis models from the LA source code. Similarly, it is possible to use UML graphical design or deployment models and a CASE tool to generate (a skeleton) of the PA source code and to reverse engineer the UML graphical design models from the PA source code.

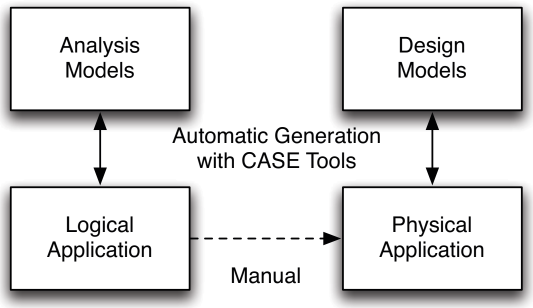

Figure 4. Automatic generation and reverse engineering of models in Analysis or Design

4. Further Discussion

4.1. Related Work

The notion of logical and physical models of software is obviously not new. Even the use of a traditional programming language for the logical model may not be novel (although TSE often used graphic models for their logical models). However, it is thought that preserving the LA and using source code changes (diffs) to communicate the incremental changes to the functional requirements to the developers of the PA is novel. The DAM encourages a full lifecycle iterative and incremental approach to development with all changes to the functionality of the PA being implemented in the LA firstly (i.e. always doing explicit analysis before design).

Model-Driven Development (MDD) [10, 11] and the OMG instance Model-Driven Architecture (MDA)

[12] are more ambitious attempts at automation of software development than the DAM by working at higher levels of abstraction (like current source code is a higher level of abstraction than machine code) and automating the compilation down to machine code. Whilst it is possible to abstract away from a concrete platform and to map an abstraction to a concrete platform, it is much more difficult to see any compilation being able to make all the designs and design tradeoffs that are needed to meet varied and custom non-functional requirements any time soon. Whilst the LA may correspond to the Platform Independent Model (PIM) and Platform-Specific Models (PSMs) may correspond to the initial draft PAs (for various platforms), this approach does not seem to address the tradeoffs, optimisations, and custom designs needed to meet the non-functional requirements.

In its simplest expression, the DAM approach is basically two software application written in regular programming languages, the LA language usually higher level and easier to develop in than the PA application. The LA application is focused only on functional requirement and has no specific project-related non-functional requirements. The PA is a reimplementation of the LA on chosen implementation technologies to meet the projects non-functional requirements. Source code changes (diffs) in the LA are used to specify required functional changes to the PA. The DAM approach does not see the LA as being at a higher level of abstraction to the PA; it is just using the simplest (physical) software language and environment to implement (only) the functional requirements as quickly and easily as possible. The LA is as detailed and complex as the domain solution / software problem requires it to be. The DAM approach does embrace transformations and reverse engineering between models at various levels of abstraction for the logical application and the physical application separately. However, it does not seek (at least at this stage) anything more than simple syntactic transformations between the LA and PA. This is seen to be where the true work and expertise of software developers.

4.2. Common Questions

Who develops the logical application? The LA will be developed by a software developer(s) with the co-operation and, perhaps, even the participation of the product owner (or users and domain experts). The product owner will raise a user story, use-case or functional requirement and the developer will implement that in the LA directly, sometimes even whilst with the product owner.

Is the LA the same as a prototype? The simple answer is, generally speaking, no. A prototype is usually a quick and dirty implementation of a physical design as a test of the appropriateness of the solution or the feasibility of finding a solution. LAs are supposed to be well designed to capture the functional requirements. Further, once the feasibility of the problem has been determined by a prototype it is supposed to be discarded for a better-designed solution. The LA is designed to stay around and always be the arbiter and “point of record” for the domain solution / software problem (especially as it changes).

In a way, the DAM approach mimics some aspects of rapid prototyping, especially as it was often done in the 1990s, e.g. some developers would implement a prototype in Smalltalk [13] and then re-implement the real system in C or C++ to meet the non-functional requirements (often speed). Smalltalk is an effective rapid application development environment, allowing developers to run the application and fill in missing code / modules as they are found at run-time. Most Smalltalk environments also employ the idea of an application image that is persistent (including the development environment), which would be very useful for LA development.

As [14] writes, Smalltalk "can also be used to avoid the shift between the description/specification of a system and its implementation." This is the goal of the LA. However, we think it is novel to use this approach within a highly iterative and incremental approach. Mostly prototyping is for early evaluation or specification and the prototype is subsequently discarded. DAM suggests maintaining the LA and the description / specification of the domain solution / software problem, and modifying it as needed in order to modify the PA subsequently.

Will the product owner just want developers to deploy the LA? The problem with many traditional prototypes is that they were part logical and part physical application. They often included persistence, screen designs (albeit rushed), and although they often did not meet all of the non-functional requirements they often met a few (or even many). Obviously then the customer would be interested in getting the software deployed immediately. The LA will usually have inappropriate user interface design, persistence and architecture and not attempt to meet any non-functional requirements. For example, the logical application for a Web application could be a desktop application, or vice versa (as discussed later).

What about applications that don’t have basic input and output screens (e.g. embedded software or games)? Applications that don’t have basic input and output still necessarily have some form of input and some form of output. In these cases the LA would still have logical and abstract representations of these interfaces. For the case of games, the 2D for 3D nature of the game can be considered a non-functional requirement and the LA can again work with abstractions of these inputs and outputs (e.g. game control inputs and abstract scene descriptions).

Why not just use one application and try to keep the logical and physical aspects separate in the same source code (e.g. by using interfaces and aspects)? This is an interesting idea but probably difficult and doomed to fail. The whole idea of the design of software is to bend and twist the logical application as little a possible but as much as needed to meet the non-functional requirements and implementation technology choices. Good luck trying to maintain the logical aspects of code that is implemented in assembly language for speed or memory optimisation. Whilst some very simple applications may see similarity between the LA and PA, in real world cases they will be very different.

4.3. A Framework or Platform for the LA

In essence, the LA can be developed from scratch in any programming language and software development environment that meets the requirements for a logical application. As mentioned, this may often be an interpreted single command-line or desktop application that the developer can share with the product owner directly (or via some source code control system) to validate.

There are, however, common aspects of the LA that could be factored out to make implementation easier. For example, the simple handling of inputs and outputs, making it easier for developers just to declare they need a user interface with these inputs, outputs, and/or commands and for such an interface to become available. Remember the user interface design is not central here but it does need to capture grouping of inputs, outputs, commands etc.

A first approach to this would be to develop a set of libraries or an application framework, which made developing applications easier than starting from scratch. A LA Framework (LAF) could provide all the foundational functionality for any LA. Similarly, there could be libraries for simple logical storage of information. These could be just extended version of collections that allow the LA to model the grouping of entities in the domain solution (e.g. a collection of customers, or a folder of documents). All of these seek to raise the level of abstraction that is used to create the LA and make it quicker and easier to implement. A second approach could go even further. A 4GL-like software application could be provided in which the product owner could define the logical interfaces. They could define logical screens with inputs, outputs, and or commands, grouping and describing the logical types of these inputs and outputs and perhaps textually describing what each of the commands will do. This application could be a single-user desktop application or a Web-based application for multiple users and multiple projects.

It is usually beyond the ability of the product owner to go much beyond the interfaces, typically what would be described in a use-case, i.e. the interaction between users (or actors as they are called in UML) and the interfaces of the software system. Usually a software developer would be required to develop the internals of the application with guidance and information from the product owner (e.g. how the output is computed, what are the steps in completing the actions). The software developer brings the logical application to life for the product owner with this coding of the domain solution.

An interesting question is what sort of software developer will be required to develop the LA. Recall, it is devoid of any technicalities related to non-functional requirement like usability, persistence, optimizations. Also the developer needs to work closely with the users, product owner, and/or domain experts to really understand the domain solution and software problem. Could a business analyst with the right training and skills undertake this task? This role of LA developer is somewhat distinct from the roles of traditional or agile software developers.

5. Summary and Future Research

This paper has highlighted the differences between analysis and design and logical and physical models. It has focused on some problems with Traditional Software Engineering and some problems with Agile Software Development as motivation for a new approach to software development called the Dual Application Model. It involves the development of two software applications instead of the regular one deliverable application. The first application is the Logical Application and it is a runnable logical model of the domain solution / software problem. It is independent of any solution technology and the non-functional requirements and is runnable. It is developed as a clear and natural record of the domain solution / software problem so that it can be rapidly developed and easily maintained. The Logical Application is developed with close involvement of the user(s), product owner, and/or domain expert(s).

The second application is the Physical Application and it is a physical model of the software solution and what is eventually delivered as the software solution.

The Logical Application is a first draft for the Physical Application but the application must be extended and modified to run on the physical implementation platform and to meet all the non-functional requirements of the software solution, e.g. persistence, ease-of-use, speed, and reliability. The development of the Physical Application is the domain and expertise of software developers, although the product owner through their desire for various non-functional requirements directs it. The development of the PA, like the LA, is done iteratively and incrementally and directed by changes to the LA source code, user stories, use-cases, functional, and also non-function requirements provided by the product owner.

This approach in line with the proposed principles of Agile Software Engineering, i.e. to have separate logical and physical models and to work forward from the logical to physical models (in each iteration). It also supports multiple models developed separately from the Logical and Physical Applications or used to generate (at least initially) parts of the Logical and Physical Applications. It does not include, however, the generation of the Physical Application from the Logical Application (as done in MDD) since this is where the real expertise of software developers is needed. A number of advantages and disadvantages of this approach are discussed with the belief that overall it will lead to the more efficient and effective development of higher quality software.

There are four important features of this approach. Firstly, the LA is a full model of each particular version of the domain solution / software problem not some high-level abstraction that leaves out details and may be inconsistent. Secondly, every iteration of the LA is a runnable application that can be verified with automated and manual testing and validated by the product owner, users, and other stakeholders before significant effort is made to implement that iteration of the PA. Thirdly, changes to the LA can be tracked by a source code version control system and provided to software developer to guide them in updating the PA. Compare this to TSE when changes to logical models were generally hard to track. Fourthly, the LA and PA can be developed iteratively and incrementally and in parallel because of the ease of tracking the changes to the LA

The approach has been “walked through” for simple software development problems (e.g. a library management software system). It needs, however, to be trialed in larger more real-world software development projects to see and measure how effective and efficient it may (or may not) be. There is also a lot of room for research and development of class libraries, application frameworks, and CASE tools to support the Dual Application Model. If successful this approach could become a central component of Agile Software Engineering.

6. References

[1] D. West, T. Grant, M. Gerush and D. D’Silva, *Agile development: Mainstream adoption has changed agility*, *Forrester Research*, 2010.

[2] J. Rumbaugh, I. Jacobson and G. Booch, *Unified Modeling Language Reference Manual, The*, Pearson Higher Education, 2004.

[3] P. Kruchten, *The rational unified process: an introduction*, Addison-Wesley Professional, 2004.

[4] K. Beck, M. Beedle, A. v. Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland and D. Thomas, *Manifesto for Agile Software Development*, http://agilemanifesto.org/, 2001.

[5] A. Aitken and V. Ilango, *A Comparative Analysis of Traditional Software Engineering and Agile Software Development*, *System Sciences (HICSS), 2013 46th Hawaii International Conference on*, IEEE, 2013, pp. 4751-4760.

[6] I. Jacobson, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

[7] K. Schwaber and J. Sutherland, "What is Scrum", URL: http://www.scrumalliance.org/system/resource/file/275/whatIsScrum.pdf (2007).

[8] K. Hiranabe, *Kanban applied to software development: From agile to lean*, http://www.infoq.com/articles/hiranabe-lean-agile-kanban, 2008.

[9] B. W. Boehm, "Software engineering economics", Software Engineering, IEEE Transactions on (1984), pp. 4-21.

[10] S. J. Mellor, T. Clark and T. Futagami, "Model-driven development: guest editors' introduction", IEEE Software, 20 (2003), pp. 14-18.

[11] O. Pastor, S. España, J. I. Panach and N. Aquino, "Model-driven development", Informatik-Spektrum, 31 (2008), pp. 394-407.

[12] A. G. Kleppe, J. Warmer, W. Bast and M. Explained, *The model driven architecture: practice and promise*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 2003.

[13] A. Goldberg, *Smalltalk-80: The Interactive Programming Environment*, Addison-Wesley, Reading, MA, 1984.

[14] F. Kordon, "An introduction to rapid system prototyping", Software Engineering, IEEE Transactions on, 28 (2002), pp. 817-821.