Prototizer: Agile on Steroids

Aram Hovsepyan

CODIFIC
aram@codific.eu

Abstract. The model-driven software development (MDSD) vision has booked significant advances in the past decades. MDSD was said to be very promising in tackling the "wicked" problems of software engineering in general. However, a decade later MDSD is still far from becoming widely recognized within the mainstream software development. At the same time Agile software development methodologies are widely considered as the way to go. This seems mysterious as MDSD seems to be the right methodology to boost Agile approaches. From a MDSD perspective, the design models are nothing but a waste within an Agile software development process.
In this experience report, we present Prototizer - a tool based on model-driven software engineering that could boost the Agile vision. We present a validation of Prototizer on a recent case study and discuss the main lessons learned throughout the past years.

Keywords: agile software development, minimum viable product, model-driven development process, prototizer, lean entrepreneurship

1 Introduction

Given the advances in the hardware technologies software development in general is becoming an increasingly complex activity. For about a decade the model-driven software development (MDSD) vision has seemed very promising in efficiently tackling the essential complexities of the software development process [1]. The MDSD vision, primarily focused on the vertical separation of concerns, aims at reducing the gap between problem and software implementation domains through the use of models that describe complex systems at different abstraction levels and from a variety of perspectives. Automated code generation, claimed to be one of the most valuable assets of MDSD, allows us to translate these models instantaneously into code. Strangely, MDSD is still far from becoming accepted within the mainstream software development [2].

As opposed to MDSD, Agile methodologies are currently considered as best practices within software development. The spectrum of agile methodologies is very broad. The core principles are typically focused on fast iteration cycles, responsiveness to change, early customer involvement, "good design", etc. Agile approaches aim at reducing the waste of big up-front analysis, planning and documentation. Nonetheless, even the most code-centric agile methodologies suggest

the use and evolution of design models for communication and documentation purposes. From this perspective, it is our strong belief that MDSD is a necessary building block within a mature agile methodology. Introducing a lightweight MDSD within Agile methodologies could further reduce the waste, enforce a good design, aid at rapid prototyping and customer involvement. Indeed, if we use models for representing software design, we might as well use them as software artefacts, rather than merely a communication and documentation tool.

In this experience report, we present Prototizer - an MDSD supporting tool that could be used within any Agile software development process. Prototizer embraces only a small part of the MDSD philosophy. Nevertheless, this small part is the enabler of the true Agile vision. Over the course of the past five years we have validated Prototizer on numerous data-driven software systems. More importantly, we are still maintaining and expanding these software systems using Prototizer.

The remainder of the paper is structured as follows. In section 2, we provide essential background information on agile software development and model driven software development. We also describe the problem statement in detail. In sections 3 and 4, we present our solution and its application on a recent case-study. Section 5 provides a brief evaluation of Prototizer. Finally, section 6 concludes this paper.

2 Agile vs Models

This section provides a brief overview of agile and MDSD methodologies by focussing on the key concepts that are in the core of the two paradigms.

2.1 Agile Software Development

Agile software development is a rather overloaded term and there are many approaches that claim to be agile. The most known approaches in the literature include eXtreme Programming (XP) [3], Scrum [4], Feature Driven Development (FDD) [5], Kanban [6], Dynamic systems development method (DSDM) [7]. Despite their intrinsic differences, which are out of the scope for this paper, all agile approaches are focused on concepts such as adaptive planning, evolutionary development, early delivery, continuous improvement, etc. More importantly, agile has become a synonym of lean, i.e., reducing waste and focusing on efficiency. This means that the heavyweight planning, documentation, software architecture and design phases are reduced. Fast iteration cycles (referred to as timeboxes or sprints) with a duration of weeks or even days resulting in a completely implemented, validated and verified subset of requirements is a central theme in all agile approaches [7]. Nonetheless, even the most code-centric agile approaches advocate the use of design models [3]. While models are used for documentation and communication purposes established agile approaches advice the use of UML and do not exclude the use of "complex models using specific notations" [7]. It is our strong belief that a lightweight model-driven software development process

can substantially increase the added value of agile approaches. Keeping models up-to-date in an agile process could be challenging and requires a substantial rigor, while MDSD forces the developers to update the models accordingly. Fully automated, but partial code generation will provide instant prototyping highly praised within agile.

2.2 Models

Similar to agile, model-driven software development (MDSD) also covers a relatively broad spectrum of ideas, techniques and tools. Despite their differences, we believe two key ideas are essential within most MDSD approaches.

Code generation Code generation is a central selling point behind MDSD [8]. Code generation is instantaneous and saves time for the developers[1]. Code generation improves the source code quality as generated code can be tailored to follow best coding practices and can be considered to be bug-free [8]. Finally, this instant code generation enables developers to play with the solution and quickly deliver prototypes of the final system. Fast prototyping enables early validation that is a central theme in agile approaches.

Models as primary software development artefacts The models within a MDSD approach are no longer a mere piece of documentation, but actually an essential software development artefact. Indeed, models have to be precise and complete as they are fed to a code generator. As a result, models are always up-to-date with the source code that makes them a valuable "lingua franca" between the stakeholders. Software systems developed using MDSD are less likely to evolve to a spaghetti-like systems where only the developers can manage to find their way [8].

2.3 Problem Statement

It is our strong belief that there are two key obstacles that prevent MDSD from entering the mainstream software development and boosting the agile software development processes.

Rigid Code Generators Typically, in a MDSD approach the code is generated based on a template (e.g., Eclipse JET) or a script that is a programming language on its own (e.g., Acceleo [9]). One of the essential problems in MDSD is that these templates/scripts are claimed to be reusable. In extreme cases, out of the box generators are created and published, such as "THE Java code generator", "THE C code generator", etc. It is unlikely that two different software

[1] Throughout this work we only consider the MDSD vision where the code generation is partial. Concretely, this means that the developers will typically generate the overall structure and the behaviour will be manually programmed by the developer.

development firms will be happy with the same generator. Unfortunately, the existing generators are often too rigid when it comes to having a quick editing mechanisms to the already messy code generation templates. Finally, code generators must enable iterative development, hence, the manually written code should not be rewritten by the automated code generation. Although MDSD has always stressed the importance of this issue its solution is far from trivial. The concept of a "protected code section" seems to solve the problem at a first glance, however it is not clear how to properly use them.

Steep Learning Curve Even the most simple MDSD approach has a rather steep learning curve. We believe that the main cause is that MDSD approaches often try to oversell and become too heavyweight [8]. For a developer who is new to MDSD and only has a rather superfluous understanding of UML class diagrams it would be extremely difficult to join the models club. There is a lack of simple lightweight MDSD success stories the potential followers could start playing with. Once again, the technology providers typically try to provide ready-to-use generators, rather than focusing on the mechanisms on how to modify the existing generators or create new generators. As a result even the early adopters are unable to step into the world only the technology providers understand.

In the next section, as an early adopter we present a toolset that we have created based on existing MDSD technologies.

3 Prototizer

We refer to Prototizer as the toolset that enables the model-driven and agile software development process. In this section we describe both the process we follow as well as the toolset.

3.1 Prototizer Software Development Process

Figure 1, illustrates a structural view of the development process that presents each development activity along with their structural connections to other activities. The solid lines on the figure are both workflow and artefact transition from one development activity to another. The dashed lines are traceability links between different artefacts. Traceability information currently falls out of the scope for our approach and will not be discussed in this paper. The presented process process is in line with the V-Model. We briefly describe each of the development phases in the proposed development process along with the underlying technology that we have used.

Requirements Analysis This phase refers to both business requirements analysis as well as their translation into the technical requirements analysis. This activity is done using a more traditional approach, i.e., by using a text editor.

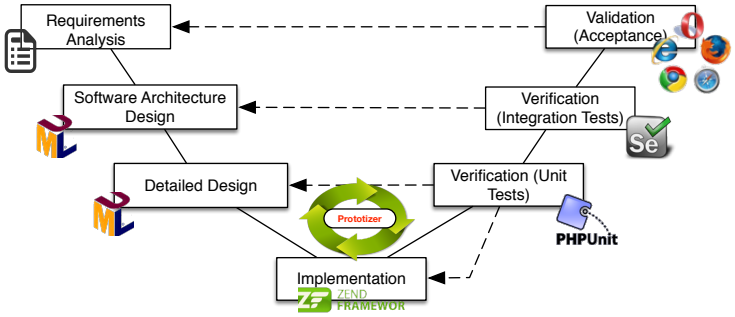

Fig. 1. Prototizer Software Development Process

Software Architecture Design This phase defines the architecture of the overall software system. The software architecture is created in UML by the means of component/connector and deployment diagrams. Currently, we do not leverage the software architecture explicitly in the code generation.

Detailed Design Traditionally, this phase described the detailed UML class diagram that is further used for code generation.

Implementation Our firm mainly leverages the PHP Zend Framework as an underlying platform. However, virtually any programming language and platform can be used for the implementation.

Verification (Unit and Integration Tests) The verification phase focuses on automated unit and integration tests that are an essential part of any systematic software development process. Given the PHP implementation platform, we further rely on PHPUnit and Selenium Webdriver for the unit testing and integration testing.

Validation Finally, the end customer is expected to perform the validation of the product release and officially accept the release. This is done by using a modern web-browser.

The V-model on its own has no agility as it represents an extension of the waterfall model. Thus, for the dynamics of this process we leverage the Dynamic System Development Method (DSDM) Atern agile project delivery framework used for software development [7]. The idea behind DSDM is to develop a solution iteratively starting from global view of the product. Figure 2 presents the timebox concept that is a key technique in DSDM Atern. It represents the iterative process to control the creation of the product under development with specific review points to ensure the quality of the product and the efficiency of the delivery process. A more detailed description of DSDM Atern is out of the scope of this paper [7].

3.2 Prototizer Tool

Prototizer is a tool implemented as an Eclipse plug-in and can be downloaded from [10]. Prototizer is largely based on MOFScript that is an open-source code

Fig. 2. Timebox

generation technology developed by SINTEF [11]. Prototizer transforms the input UML model into code using a specified generation cartridge. Currently, we have developed two different generation cartridges for generating code towards Zend Framework 1 and 2 respectively. The generation cartridge contains two components, i.e., resource copier and generation script.

Resource Copier The resource copier simply copies various static resources, such as, libraries, Javascript/HTML/CSS files, into the file structure of the project. The set of resources can be easily manipulated by the developer by simply managing the static files within the plugin cartridge folder. These resources are typically specific for a certain company or even project domain.

Generation Scripts The generation scripts are used by Prototizer to translate the UML model into code. Modifying a generation script is straightforward as MOFScript is an imperative language syntactically similar to Java. For the specifics of the generation scripts we refer to MOFScript specification [11]. We leverage two complementary techniques in order to make the code generation scripts sufficiently flexible when it comes to manual code refinements.

1. We use protected code sections that are placeholders for manually refined source code that are kept intact upon subsequent generation steps.
2. In certain cases protected code sections could place unnecessary constraints on the manual coding. In order to overcome this problem we leverage the generation gap pattern [12]. The generated code is placed in abstract superclasses that can be easily subclassed with manually written code.

4 Case study: CODIFIX

The case study presented in this paper is our own enterprise resource planning system named CODIFIX. CODIFIX initially consisted of a rather primitive content management system for our website. However, we have gradually added various new modules that have introduced a substantial set of new functionalities. In this section, we will briefly describe two of the CODIFIX modules, i.e., the content management system and the issue tracking system. We focus mainly on the models from which the source code is continuously and incrementally generated. At the end of this section, we provide an overview of the artefacts that are actually generated from the design models.

4.1 Technology Stack

CODIFIX is implemented in PHP by leveraging the Zend Framework version 2. Note that we do not use the Doctrine framework that provides a transparent database storage. Rather, the generation step creates the complete database API ready to use by the developers. We also rely on client-side functionality written by third parties in Javascript. As an underlying database we use MySQL.

4.2 Content Management System

Figure 3, presents the diagram of the simple content management system (CMS) model we have designed to use for our informative website. The CMS consists of menus (*Menu*) denoting the pages. Each menu has a specific language (*Language*) and can have a parent menu. The website information is represented as contents (*Content*) where each content can belong to a menu object. The attributes of the classes and the semantics that is assigned to the model elements and used in the generation are out of the scope of this paper.

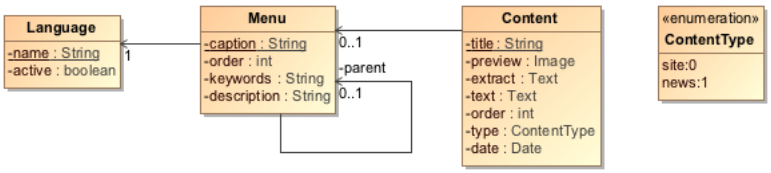

Fig. 3. Content Management System

4.3 Issue Tracking System

Figures 4 presents the diagram of the issue tracking system model (ITS). The ITS groups tasks (*Task*) in projects (*Project*). Additional task information is represented by attachments (*Attachment*) and comments (*Comment*). The task and project related information is obviously linked to users (*User*) each of whom belongs to a certain client (*Client*). The access control is currently hard-coded and depends on the linking between *User*, *Role* and *System*.

4.4 Generated artefacts

This section provides an illustration of the artefacts generated by Prototizer.

Database scheme and API The complete data layer as well as the communication API is generated by Prototizer. The database scheme is generated as an .sql dump. We have developed a simple scripting mechanism to synchronise

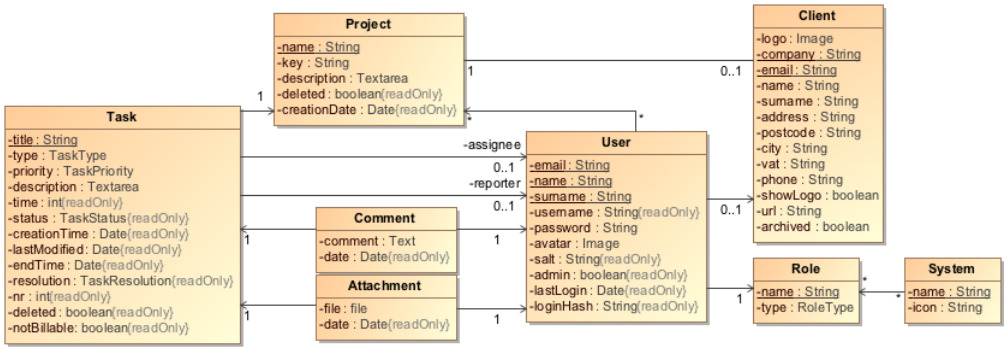

Fig. 4. Issue Tracking System

the generated database scheme with the actual running database. The database communication API is a collection of classes that allows a systematic manipulation of the database entries for each of the classes within the UML models. Currently we do not leverage on frameworks like Doctrine (the PHP version of Hibernate) and the database communication API represents a relatively large codebase. Typically, this part of the generated code is never modified manually as the UML model within the Prototizer philosophy is semantically complete.

Model classes Each of the modelling entities (along with its relationships) is translated into a corresponding PHP Class. These classes represent the models in MVC terminology. Obviously, the classes have pre-generated list of attributes as well as getter and setter functions. In addition, we also generate validation rules for each attribute as defined by its type. For instance, the *order* attribute in *Menu* class must be an integer. The model classes are typically manually refined, hence, protected code sections denote the places where this can be realised. In general, the complete attribute information as well as their getters and setters should never be modified manually. On the other hand, methods like toString() and any additional methods that could be added by the developer are protected by Prototizer.

Controllers and views The basic features for most of our systems (including CODIFIX) are the web-based create, retrieve, update and delete (CRUD) interfaces to manipulate the database entities. These interfaces are also generated by Prototizer. In MVC terminology these are the controllers and the views. The controllers are in charge of processing the HTTP requests and constructing the HTML views that are sent back to the web-browsers by the web-server. The views are mainly HTML scripts with template parameters that are dynamically instantiated by the corresponding controllers.

Generated code percentage Overall, our currently running CODIFIX project contains 17546 lines of generated PHP/HTML code from a total codebase of 22281 lines of code (including comments and white spaces). Current best practices in Zend Framework would require one to write all this code manually. Nonetheless, the productivity increase is limited by two factors. Firstly, not all of the generated code is actually used. Given the ease of code generation we generate quite a few helper functions that are not always needed and used. Secondly, the generated code is by definition trivial as it constitutes the repetitive part of the code. However, writing the code manually when it can be generated is a 100% waste of time.

5 Discussion

In this section, we provide an overview of the lessons learned within the context of our firm.

5.1 Benefits

The process behind Prototizer is in line with agile and virtually any agile framework can be plugged in as a concrete dynamics of the process. Prototizer is a clear realization of the MDSD vision regarding the centrality of models, instantaneously generated source code and the increased prototyping abilities. This saves a substantial amount of time and allows our developers to focus on core problems rather than spending time in typing code. The generated code is considered to be bug-free as we assume the generators are bug free. Prototizer forces the designers to create complete UML models of the system, hereby making the communication between designers and developers in the context of the system structure much more clear. Prototizer also enforces a specific code structure that is valuable especially for the less experienced developers within our firm. In our experience the long-term benefits of Prototizer are even more critical. The existence of a complete and up-to-date UML model aids the system longevity and substantially reduces the maintenance cost. Within our firm the cost of maintenance (e.g., new features, change requests) often by far exceeds the original cost of development. This is confirmed by various studies in the industry (e.g., [13]).

5.2 Drawbacks

Virtually any code generation approach introduces additional constraints for the developers. The protected sections where developers are expected to operate sometimes lead to two problems. Firstly, inexperienced developers inevitably misplace manually written code that leads to overwritten code on subsequent iterations. While the code is not really lost (thanks to version control) it introduces an additional overhead. Secondly, certain manual refinements could be relatively complicated to fit within the protected sections. As a result, developers are sometimes forced to duplicate code in order to achieve the desired result.

5.3 Evaluation

From a research point of view Prototizer is not innovative. In fact, the building blocks of Prototizer, i.e., MOFScript and EMF (the de factor UML standard) were stable almost a decade ago. However, from a state-of-the-practice point of view, Prototizer is a pragmatic answer to both problems presented in section 2.3. Generation cartridges can be easily modified and new cartridges can be quickly created by example. The resource copier requires simply moving around files and folders that are needed for a specific project type. MOFScript is a powerful, yet very simple language for creating and modifying code generation scripts. Prototizer is a very lightweight approach only focusing on a very small subset of UML, i.e., class diagrams. Instead of focussing on a programming language (PHP code generator), we have created generation cartridges for a specific framework (Zend code generator). We believe that these aspects contribute to a less steep MDSD learning curve. However, we have not validated in a systematic fashion either of the claims.

6 Conclusion

In this experience paper, we have presented Prototizer - a tool that enables a boosted agile software development approach. Prototizer, which is based on existing model-driven software engineering building blocks, enables the use of the design models as actual software development artefacts, rather than mere documentation.

References

1. France, R., Rumpe, B.: Model-driven development of complex software: A research roadmap. In: Proceedings of the 29th International Conference on Software Engineering, IEEE Computer Society (2007) 37–54
2. Fieber, F., Regnat, N., Rumpe, B.: Assessing usability of model driven development in industrial projects. CoRR **abs/1409.6588** (2014)
3. Beck, K., Andres, C.: Extreme Programming Explained: Embrace Change (2Nd Edition). Addison-Wesley Professional (2004)
4. Schwaber, K., Beedle, M.: Agile Software Development with Scrum. 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (2001)
5. Palmer, S.R., Felsing, M.: A Practical Guide to Feature-Driven Development. 1st edn. Pearson Education (2001)
6. Anderson, D.: Kanban. Blue Hole Press (2010)
7. DSDM Consortium: The DSDM Atern Handbook. DSDM Consortium (2008)
8. Frankel, D.: Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons, Inc., New York, NY, USA (2002)
9. Obeo: Acceleo. (http://www.eclipse.org/acceleo)
10. CODIFIC: Prototizer. (http://prototizer.com)
11. SINTEF: MOFScript. (http://modelbased.net/mofscript/)
12. Fowler, M.: Domain Specific Languages. 1st edn. Addison-Wesley Professional (2010)
13. Erlikh, L.: Leveraging legacy system dollars for e-business. IT Professional (2000)