Agile Models Need to be Bottom-up
Adding Productivity to a Late Project Makes it Later

Pietu Pohjalainen
Department of Computer Science, University of Helsinki, Helsinki, Finland

Keywords: Agile Modeling, Bottom-up Modeling, Productivity.

Abstract: Model-driven architecture is a top-down approach to software engineering. Due to its heavy emphasis on tools and process, it has not been considered to be not a good fit for agile time-boxed iterations. Lightweight models are often a better alternative in agile development. However, we argue that in order to realize productivity gains, these models can and should be brought as software architecture level entities.

1 INTRODUCTION

Productivity is a key issue in professional software engineering. In many software businesses, being able to produce more functionality in a given timeframe is advantageous: in a productive environment customers gain more value from software projects, and project professionals have higher job satisfaction.

Agile process methods currently help projects to avoid big mistakes of producing the wrong product to the wrong customer at wrong time. Improved communication with the customer, learning effect within agile iterations and time-boxed development all contribute as process-level reinforcements to project work. However, relatively little attention is given to actual productivity improvements in agile projects. This is surprising, given the fundamental nature of productivity and productivity improvement in history of industrialized world.

Model-driven engineering is a recent movement with an attached promise for improved productivity. Productivity gains in limited domains, such as compiler construction makes the idea of raising the level of abstraction appealing. However, combining agile process with engineering approaches that include any significant investment, or up-front planning, before the engineering discipline can be employed can turn out to be problematic.

To fix these problems, we propose a flavor of model-driven engineering that accounts the restrictions imposed by agile software development process models. This bottom-up agile model-driven development recognizes smaller sub domains within the software that are amenable for lightweight modeling. These small models can be used in traditional source-to-target generative programming or in some cases the source code itself can be treated as the source model, thus reducing redundancy. The bottom-up modeling approach entails lighter initial investment than domain-specific modeling, and thus allows fast experimentation cycle within the limits of tightly time-boxed agile iterations.

Tools with higher investment costs, or steeper learning curve are problematic in the time-boxed development models. This causes a paradox: productivity is defined by the rate of output to effort. When investing to better productivity, the investment can eat up all the expected gains, due to the short visibility time in a cycle-driven process. In the conclusion of the *mythical man month* (Brooks, 1995) it was found out that adding more people to a late project makes the project later. A similar effect can be seen here: adding productivity at a late stage of a project makes the overall effort higher.

The rest of the paper is structured as follows. Section 2 introduces the basic equations of process improvement for software development and explains how they relate to agile processes. Section 3 reviews modeling related disciplines, namely model-driven architecture (MDA) and agile model-driven development (AMDD) and in the context of agile software development. In section 4 we propose a lighter approach, called bottom-up modeling. Bottom-up modeling takes the special characteristics of agile methods into account, and makes it feasible to realize productivity gains associated with generative programming in agile projects. Section 5 shows an example case on how different modeling formalisms can be combined

Pohjalainen P..
Agile Models Need to be Bottom-up - Adding Productivity to a Late Project Makes it Later.
DOI: 10.5220/0004155903950400
In *Proceedings of the 14th International Conference on Enterprise Information Systems* (MDDIS-2012), pages 395-400
ISBN: 978-989-8565-11-2

to produce meaningful systems. Section 6 sets out the summarizing words and casts out the future work.

2 ECONOMIC MODEL

When investing in process improvement in software development, the obvious question is how to estimate the return-on-investment. Return-on-investment is calculated by dividing the difference of benefits and costs of a change by the costs of the change, as shown in formula (1).

$$(1) ROI = \frac{Gain - Cost}{Cost}$$

Being an abstract activity, it is very hard to estimate causes and effects in software development. For this reason, also giving exact figures for estimating whether process improvement is justifiable, is problematic in many cases. However, the general principles of economic thinking can be used to guide in decision making, although the exact numbers for a given decision might be impossible to calculate.

When a software development team is given a task to implement a software product, the total cost for the project can be calculated to be

$$(2) OC * OT$$

Where OC stands for operational cost and OT stands for operational time.

For example, if the average cost of one man-month, including the salaries, office spaces, cost of computers and so on, in the project is 10 units, operating a software development team of ten people costs 100 units per month. If the software development project takes five months, the total cost for the development project is 500 units. Economic decision criterion for improving total cost, using any given technology or managerial decision can be expressed as:

$$(3) OC * OT > Cost + OC * OT'$$

In other words, the initial investment to implement new practice or employ new techniques within the project can be justified if the reduced costs with new operations amortized over total operational time are smaller than the alternative of running the operations without changes.

Let us have a hypothetical development practice that gives 25% improvement on productivity with no consequences on schedule or personnel wages. Implementing this practice has a fixed cost of 50 units and it is immediately implementable whenever the project chooses to. If available at the beginning of the project, it clearly should be employed, as the improvement on productivity shortens the required time to implement the project from five months to four months, thus enabling a faster time-to-market opportunity and clear savings on overall expenses.

However, with agile sprints, the productivity improvement should return its investment within an ongoing sprint. This is because there is no guarantee that the project continues after the current iteration. To be able to justify this investment with a one-month sprint length, the promised productivity gain should shorten the required operational time from one month to half, which equals a productivity gain of 100%. Improvements of this magnitude are likely to be available only for very poorly performing teams.

In reality, this kind of reasoning for real software projects is very hard, maybe impossible. The example assumes a fixed amount of work, which seldom is the case. Productivity rates vary between individuals and team mix-up in orders of magnitude, and the hypothetical instant productivity boon option is just a project manager's daydream. However, the basic principle behind equation (3) is often encountered in software development organizations: new practices and technologies are discovered, and a return-on-investment justification needs to be found for employing the practice in question.

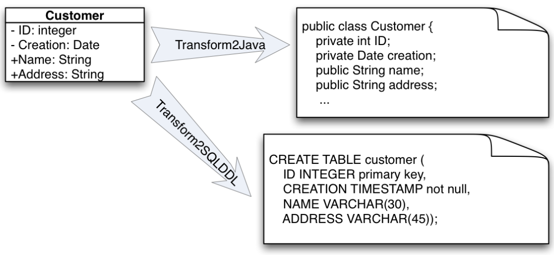
3 MODELING AND AGILITY

Agile process improvement literature focuses mainly on process-level practices. Less emphasis is given to the actual software structures that are designable in an agile process. This section reviews techniques that have an attached productivity promise: the Object Management Group's Model-Driven Architecture (Miller and Mukerji, 2003) and the Agile Model-Driven Development (Ambler, 2004).

We argue that the classical model-driven architecture's approach is not very suitable for agile process due to its heavy emphasis on tools and model transformations. Then we propose that agile model-driven development cannot be justified from productivity angle, as the lack of formality in agile models prevents the usage of automated handling. Domain-specific modeling is seen as a good trade-off between formality and agility, but is still staying short of good agility due to its emphasis on specific tool usage.

3.1 Model-driven Architecture

Model-driven architecture approaches re-usability by separating concepts into three layers: platform inde-



```
public class Customer {
   private int ID;
   private Date creation;
   public String name;
   public String address;
    ...
```

```
CREATE TABLE customer (
   ID INTEGER primary key,
   CREATION TIMESTAMP not null,
   NAME VARCHAR(30),
   ADDRESS VARCHAR(45));
```

Figure 1: A UML model with transformations to Java and SQL.

pendent model (PIM), platform-specific model (PSM) and program code. Traversal between these layers is done via transformers: a platform independent model is translated to a platform-specific model by using a transformer, which augments the model with platform-specific attributes. A similar transformation is applied when translating the PSM into program code.

A typical platform independent model is expressed as a UML class diagram, which contains only class attributes, maybe with programming language-level visibility information and data type annotations. A transformation creates corresponding programming language, e.g. Java classes, with accessing methods for each of the public attributes; or data-definition statements for a relational database.

Figure 1 represents a typical case, in which a class model expressed in UML is transformed to programming language code by one transformation. Another transformation generates the corresponding database definition. These transformations contain platform-specific parametrization, as the transformation contains information about the target platform. In the UML-to-Java transformation, UML standard visibility rules are followed; but a data type transformation from UML integer to Java int is performed. In the UML-to-SQL transformation, similar platform-specific knowledge is being encoded. Most notably, the transformations contain also information about the system that are not shown in the source model. For example, the knowledge about different field sizes for Name and Address fields that have the same data type in the source model is encoded into the transformation.

A notable shortcoming in the agile mindset of using UML class diagrams to express the platform independent models is the lack of extensibility. The class diagram can directly express only a limited set of parameters, such as visibility, data types, and default values. Further extensions require using UML profiles, which may or may not be supported by the used toolset. This is a heavyweight way of producing productivity: toolset evaluation takes time. Even through the after the most rigorous evaluation, a toolset's suitability to project needs are known up to a certain limit. Only practice will reveal whether the toolset actually delivers its promised functionality, as any seasoned professional can witness.

Another problem is that given the current fast rate of change in technology choices and architectural evolution in software engineering, the model transformations provided by the chosen toolset probably do not match the current architectural needs of the developed software. When this occurs, the development team has two choices: try to find an alternative, better suiting toolset or try to improve the existing toolset. The first option basically stalls development work, as the focus has changed to finding the right tool for the job instead of actually doing the job. The second alternative, if viable at all due to copyright reasons, requires specialized personnel who have the ability to modify the transformations used by the toolset. Since the development of the actual software cannot be delayed, the software's architecture evolves in parallel to transformation development. For this reason there is a good chance that any given set of model transformations is already obsolete its completion time.

For these reasons, unconstrained usage of model-driven architecture cannot be considered to be a good match for current agile development environment. However, we do not propose to canonically reject model-driven architecture. Our critique primarily bases on the combination of short-lived sprints of agile development and the uncertainty of toolsets and practices promised by MDA tool vendors. In cases where a toolsets abilities and limits are well known in advance, using the toolset-driven approach can be beneficial even in tightly time-framed situations.

3.2 Agile Model-driven Development

Agile model-driven development (Ambler, 2004) attacks the problems in model-driven architecture by relaxing the strong formality and tool support requirements. Instead of using complex and extensive models, the approach emphasizes models that are "barely good enough" for a given task. Modeling is mostly done top-down, although the approach does not discourage a bottom-up approach.

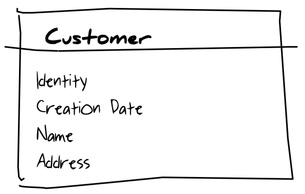


Figure 2: A hand-drawn sketch in agile UML modeling.

According to this philosophy, created models should not affect the agility principles of a given process. As long as a model can be created and exploited within a production cycle (usually 1-3 weeks), it is suitable for agile modeling. This is a promising approach, but it does not state much about the possible modeling tools - actually, the approach de-emphasizes the need for tools, and focuses on people. A model is suitable for the approach as long as it can be exploited within a production cycle.

Due to this requirement, most modeling is based on high-level abstract modeling languages with little formalism. They are easy to learn, simple to use and fit well within a given time period. However, they offer more to easier problem domain abstraction than to automatic productivity increase. Under our best knowledge, no productivity-related empirical validation has been done for agile model-driven development.

4 BOTTOM-UP MODELING

We argue that productivity gains chased with model-driven engineering should be combined with agile development models by examining the productivity problems that are encountered on project level. If a given productivity problem gives a feeling that its root cause is associated with the problem of lack of abstraction, or incorrect level of abstraction, then it could be a possible candidate for building a higher level model for that particular part of the software.

We call this approach as bottom-up agile model-driven development. In this approach bottom-up models are a way to introduce light-weight modeling to agile development process.

Instead of applying top-down methods, which recursively decomposes the problem to smaller pieces, we can alternate to a bottom-up approach. This approach identifies smaller problems and develops solutions to these. When this bottom-up cycle is repeated, gradually the solution for the whole emerges. Given a program domain, a bottom-up approach identifies sub domains that are amenable to modeling

Agile bottom-up modeling constraints the identification process to such tools and techniques that can be applied in an agile process model. Its application is thereby limited to a small number of tools, which can be evaluated and applied within a tightly time-boxed iteration. Yet, although this search and discover approach theoretically produces non-optimal solutions, it guarantees that progress is not stalled while searching for the optimal solution. This way, a bottom-up approach to modeling dodges the heavy up-front planning phase.

An essential property of bottom-up modeling in agile process is that the building of model languages and models can be decomposed into sprintable form. We mention both model languages and models because the essence of bottom-up modeling is to find suitable abstractions to the problem at hand, and often this means inventing a new language or reusing an existing language for modeling. This notion is contradictory to common wisdom of using the best existing tool for the job at hand. However, given the large number of different tools and techniques available on the market, it is not possible to do a throughout tool evaluation within the time frame of an iteration. For this reason, agile teams often need to build their own abstractions for modeling.

These abstractions or languages are not necessarily complex, meaning that there is no mandatory need to building complex modeling languages with associated tool support. Instead, existing languages can be piggy-backed and reused as is common with domain-specific languages (Mernik et al., 2005). Also a domain-specific modeling tool can be employed, once an initial understanding of the problem at hand has emerged. However, the obvious downside of this approach is that repetitive application of ad hoc modeling constructs might gradually erode the overall architecture of the software.

Bottom-up modeling does not limit the format of source models, as long as the model is expressible in machine-readable form. This means that the modeling language does not need to be a graphical boxes-and-arrows -type tool. Actually, although the traditional boxes-and-arrows kind of modeling can be beneficial in the drafting board, the lack of exact interpretation for the used symbols hinders productivity when forwarding these models to any type of automatic code generation or runtime interpretation. Often used alternatives are external domain-specific languages, but an interesting choice is to use the source code as the source model. This option is interesting for practical programming, as using the source code as the source model for further transformations builds increased robustness against modifications into the software.

5 A CASE STUDY

In order to illustrate the idea of using the bottom-up approach to modeling, we show a case of using bottom-up modeling for implementing a feature modeling (Kang et al., 1990) environment in a bottom-up way. Feature modeling is a formalized way of building option spaces. The formalism allows to define structures with mandatory features, optional features and combinations of them.

An example presented in (Czarnecki and Eisenecker, 2000) models an option space for a car is shown in Figure 3. The car needs to have the body, the transmission, and the engine. The transmisison can alternatively be automatic or manual. The engine can be electric or gasoline driven, or both. Optionally, there can be a trailer pulling hook. Given this option space, there is 12 distinct configurations that satisfy this model.

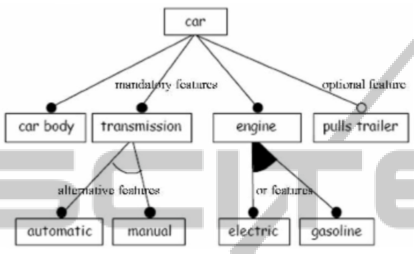


Figure 3: A feature model of a car.

For our discussion, the interest lies in how to implement functionality to handle these kinds of models. In model-driven architecture, the standard approach is to use transformations to bring the source model into the streamline of standard modeling languages. Thus, a transformation for translating the feature model into an UML model is needed. Literature presents various ways for doing this. Many researchers (Clau and Jena, 2001; Griss et al., 1998) have presented different flavours of using stereotypes for representing variability in UML.

These studies contain many fine points for implementing beautiful models of variability using the standard technologies. However, for practical software development cases, the variability is just one of the dozens, hundreds or thousands aspects that a development team needs to tackle. It can be impractical to start discussing about the academically correct way of implementing this variability, since it can be hard to demonstrate how this discussion brings value to the end customer. Due to the economic reasons discussed in Section 2, it probably never will.

An alternative is to work with this specific problem, using the standard tools offered by the implementation environment. For the variability example, (Pohjalainen, 2011) have documented a way of using standard regular expressions for modeling variability. This approach combines good parts from both of formal modeling and agile product development. The approach has the benefit that the customer can be shown steady progress, since modeling is concentrated on small subdomains that are suitable for modeling. On the other hand, since the models are implemented by using the standard implementation environment structures, the mismatch between modeling environments and implementation is kept at minimum.

For the feature model in Figure 3, the bottom-up modelled definition could be implemented using regular expressions as follows:

```
car body
transmission (automatic | manual)
engine (electric | gasoline)+
pullsTrailer?
```

This is a very concise way of using higher level abstraction to bring benefits of modeling into implementation level.

However, using this kind of modeling language-specific translation to the implementation language raises some questions. Using UML models gives the possibility to scale the scope of modeling to include also attributes of modeled entities. For example, if there is a need to specify the size or power of the engine being modeled. In a class diagram, it is very straightforward to add the attribute in question to the class model. But using regular expressions to model even three to five different engines would soon prove to be cumbersome.

Our approach to this problem is to exploit the nature of bottom-up problem solving. The chosen modeling implementation was fine for that specific problem, and when given a new problem, we look for a suitable solution. Re-using the notion of piggybacking existing languages, we can choose to use e.g. XML Schema (Biron and Malhotra, 2004) for data modeling. For example, a data model for defining 1.4 liter or 1.6 liter gasoline engines in XMLSchema could be as follows:

```
<?xml version="1.0" encoding="UTF-8" ?>
<schema>
  <simpleType name="enginesize">
    <restriction base="string">
      <pattern value="1,[4|6] liter" />
    </restriction>
  </simpleType>

  <element name="gasoline"
           type="enginesize" />
</schema>
```

With this approach, the data modeling is done by defining XML Schema models. The expressive power of the schema language greatly overpasses the one offered by standard UML class diagrams (Martens et al., 2006). Another benefit is that standard XML tools can be used to validate data transmissions and its

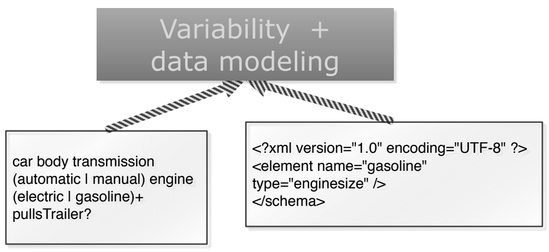

Variability + data modeling
car body transmission (automatic | manual) engine (electric | gasoline)+ pullsTrailer?
<?xml version="1.0" encoding="UTF-8" ?> <element name="gasoline" type="enginesize" /> </schema>

Figure 4: Combining two modeling languages.

semantics is well understood. As a formal language, the source documents can also be used for further model transformations; the schema language contains a well-defined variation point holder for defining new features for the data modeling tool.

This way, bottom-up modeling can use a mix-and-match approach for selecting the suitable tools for situations rising in development projects. In this case study, the developers chose to use regular expressions to model variability and the XML schema for data modeling. Combining these two allowed the developers to use models as first-class citizens in their product, since both of the used modeling languages were supported in the programming environment. Equally important, the developers were able to show steady progress towards the customer, since no delays were involved in tool evaluations.

Figure 4 shows a conceptual picture of the idea of combining two modeling languages into a meaningful entity.

6 CONCLUSIONS

We have argued that in agile software engineering the tight timeboxing of sprints creates problems for long-term planning. The option to change direction after every two weeks gives added flexibility, but deteriorates efficiency gains from the usage of models and model-driven engineering.

We propose that in order to bring the benefits of model-driven engineering to agile projects, the modeling activities should be placed in bottom-up fashion. A good choice for tackling the risks of unadequate tool support and overly optimistic tool vendor claims, the bottom-up models are chosen from the set of well-known tools with known implementations at hand.

The benefits of using this kind of reusing of existing implementation tools and techniques include the possibility of matching modeling needs with available technologies. Another benefit is that building the modeling practices from bottom-up enables a pay-as-you-go -type investment to modeling technologies: there is no need to heavily invest in unknown technologies with no guarantees of payback.

In the case study we showed how to combine variability modeling, implemented using a regular language engine, and data modeling, using the XML Schema language. This combination of modeling languages was used in the case study company, where the developers were able to gain a higher level understanding of their system via the use of these modeling tools.

REFERENCES

Ambler, S. (2004). *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press, 3rd edition.

Biron, P. V. and Malhotra, A., editors (2004). *XML Schema Part 2: Datatypes*. W3C Recommendation. W3C, second edition.

Brooks, Jr., F. P. (1995). *The mythical man-month (anniversary ed.)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Clau, M. and Jena, I. (2001). Modeling variability with UML. In *In GCSE 2001Young Researchers Workshop*.

Czarnecki, K. and Eisenecker, U. W. (2000). *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA.

Griss, M. L., Favaro, J., and Alessandro, M. d. (1998). Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, pages 76–, Washington, DC, USA. IEEE Computer Society.

Kang, K. C., Cohen, S. G., Hess, J. A., Novak, W. E., and Peterson, A. S. (1990). Feature-oriented domain analysis (foda) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute.

Martens, W., Neven, F., Schwentick, T., and Bex, G. J. (2006). Expressiveness and complexity of xml schema. *ACM Trans. Database Syst.*, 31(3):770–813.

Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

Miller, J. and Mukerji, J. (2003). MDA guide version 1.0.1. Technical report, Object Management Group (OMG).

Pohjalainen, P. (2011). Bottom-up modeling for a software product line: An experience report on agile modeling of governmental mobile networks. *Software Product Line Conference, International*, pages 323–332.