

Formalizing Agile Software Development Methods

Bedir Tekinerdoğan

Department of Computer Engineering, Bilkent University,
Bilkent 06800, Ankara, Turkey

bedir@cs.bilkent.edu.tr

Abstract

Agile Software Development has been advocated as an appropriate lightweight programming paradigm for high-speed and volatile software development. In this paper we explicitly express the artifacts, the related heuristic rules and the lightweight process of agile software development approaches using method engineering techniques. The result of this formalization process supports the objective analysis of agile software development approaches and its comparison with more rigid software development methods.

1. Introduction

Agile software development has been advocated as an appropriate lightweight programming paradigm for high-speed and volatile software development [7]. In contrast to the conventional methods and process improvement criteria such as, for example, defined by the Capability Maturity Model (CMM) and Software Process Improvement and Capability Determination model (SPICE), agile software development is less document-oriented, includes less rules and is more focused on programming. The basic motivation for this is that in case of too many rules and documents the process becomes very hard to follow correctly. It is claimed that the process itself becomes one of the bottlenecks for flexible and on-time development of software. Nonetheless, to cope with the inherent complexities of software development it is widely accepted, also by the agile software development community, that applying a disciplined process is necessary.

Agile software development aims to provide a compromise between no process and too much process by reducing the amount of documents and rules that need to be developed. As such agile methods do not focus on rigid rules that are applied in a strict order but rather it introduces a set of practices with which it is aimed to develop software in a more effective and efficient way.

Several agile processes have been introduced such as SCRUM [4], Crystal [6], Adaptive Software Development [10] and most significantly Extreme Programming [5].

Agile software development has been criticized by, process oriented researchers and practitioners due to its lack of the necessary rigor which as such might be too risky to life critical and high reliability systems [13].

Although both its advocates and the opponents seem to agree that agile software development lacks a rigorous process a close look at these practices shows that there are still some rigorous steps even though they are implicit and not systematically formalized.

In this paper we attempt to formalize (the implicit) rigorous steps of Extreme Programming (XP), which is a prominent agile software development method. For the formalization process we will apply method engineering [17] which aims to engineer designing, constructing and adapting methods, techniques and tools. With the formalization of methods we mean making explicit the artifacts and heuristic rules so that we can reason and, if necessary, manipulate the methods. In this context, we will present our so-called *Artibot* model which is an agent-based model for representing artifacts, rules and processes of software development methods.

This formalization process of XP will have the following benefits.

First of all, although formalizing the agile design steps seems directly contrary to the agile software development paradigm we think that this is useful to provide a more objective basis in analyzing and comparing these approaches with more formal software development approaches.

Second by formalizing the steps of XP, we might consider integrating some of the useful rules and/or practices in existing formal approaches.

Third, agile software developers might reflect on the best practices among the agile methods and compose an improved agile software process.

The remainder of this paper is organised as follows. In section 2 we will describe method engineering in more detail. Section 3 will discuss the basic ingredients of methods. Section 4 will provide our *Artibot* model which will be used to formalize the

artifacts and the rules of a method. Section 5 provides the formalization of Extreme Programming. Finally in section 6 we will provide our conclusions.

2. Method Engineering

During the history of software development a considerable number of methods have been introduced. Although most popular methods have a general character [11], some methods are targeted to specific application domains, such as real-time system design [8]. Some methods are specifically defined for a given phase in software life cycle, such as requirement analysis [21] or domain analysis [1]. The variations among these methods show that there is actually no universal method for each application. Even existing general-purpose methods which have been designed for as much as wide range of applications they may still fail for individual applications. To cope with the various approaches *method engineering* is introduced to develop a dedicated method for each problem domain, that is, *engineer* methods. Thereby not a single fixed method is adopted but a method is first produced similar to the production of software artifacts. Method engineering is defined as an engineering discipline for designing, constructing and adapting methods, techniques and tools for the development of information systems [17].

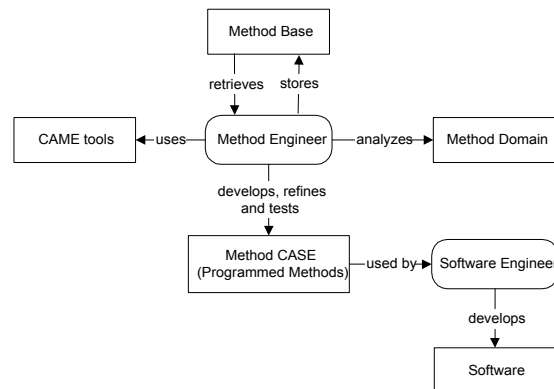


Figure 1. Method Engineering Process

The method engineering process is shown in Figure 1. Method engineers model methods and for this they will typically analyze the *method domain*, that is, the area in which the method is described. This may include method experts, books, existing CASE tools supporting the method etc. In this paper we will focus on the method domain of agile software development.

Method engineers usually use meta-modelling techniques to formally represent methods or parts of methods and store these in a so-called *method-base* for later reuse. Engineering methods using automatic tool support is called Computer Aided

Method Engineering (*CAME*). The meta-models are supported by *CAME* tools, with which methods may be tailored. If needed suitable method fragments can be retrieved from the method-base, if necessary adapted, and finally integrated into a new method. The software engineer will use *CASE* tools, which provide programmed method, and guide the software engineer in developing software. In this paper we will basically focus on the analysis of the method domain and its formalization.

3. Method

Before we can engineer methods we need to understand the basic ingredients of a method. In essence we define a software development method as consisting of artifact types, the corresponding method rules and the process that is enforced for applying the method rules.

3.1 Artifact types

Artifact types are descriptive forms that the software engineer can utilize for producing artifacts. In this sense, artifact types reflect the properties of the artifacts in the system. For example, the Unified Process [11] provides artifact types for use cases, classes, associations, attributes, inheritance relations and state-charts. Artifact types are represented basically using textual or graphical representations. Artifact types include descriptions of the models in the method description. In addition to these, the method itself may define intermediate or subsidiary artifact types to produce the final software products. An intermediate artifact type, for example, in OMT [16] is the artifact type *Tentative Class*, which describes the entities that are potentially an artifact *Class*, but which may later be eliminated or transformed to the artifact *Attribute*.

3.2 Method rules

Method rules aim at identifying, eliminating and verifying the artifacts. Most methods define rules in an informal manner. Nevertheless, method rules can be expressed using conditional statements in the form IF <condition> THEN <consequent> [20]. The consequent part may typically be a selection, elimination or an update action. For example, the Unified Process advises the following (selection) rule to identify classes:

IF an entity in a use case model is relevant
THEN select it as a class

If there is no condition a rule becomes just an *action*. In general, most rules are *heuristic* rules [15]; they support the identification of the solution but there is actually no guarantee that the solution can be found by anybody at anytime by applying the corresponding heuristic rules. The heuristic rules are generally built up over a period of time, as

experience is gained in using the method in a wider domain.

3.3 Software process

Very often, the term *process* is used to indicate the overall elements that are included in a method, that is, the set of method rules, activities, and practices used to produce and maintain software products. Sometimes the term process is also used as a synonym for the term method. We make an explicit distinction between method and process. We adopt the definition of a process as a (partially) ordered set of actions for achieving a certain goal. The actions of a process are typically the method rules for accessing the artifacts. Process actions can be causally ordered, which represents the time-dependent relations between the various process steps. We adopt the currently accepted term *workflow* to indicate such an ordering [11]. Workflows in software development are, for example, analysis, design, implementation and test. Formerly, this logical ordering of the process actions was also called *phase*. Currently, the term *phase* is more and more used to define time-related aspects such as milestones and iterations [11].

To support the understanding of software processes and improve the quality we may provide different models of processes [1]. Several process models have been proposed, including the traditional waterfall model and the spiral model, which have been often criticized because of the rigid order of the process steps. Recently, more advanced process models such as the Rational Unified Process [12] and the Unified Software Development Process [9] have been proposed.

4. ARTIBOT Model

To formalize the artifacts of a method we apply our *Artibot* model (acronym for Artifact-Robot), which is shown in Figure 2.

The artifact model consists of the following components:

- *Artifact Kernel*, which consists of the artifact structure and artifact properties.
- *Production Rules*, are the heuristic rules of a method which access and manipulate the artifact structure and artifact properties.
- *Inference Engine*, consists of a Scheduler for selecting appropriate production rules and an Interpreter for presenting and firing the selected production rules. In essence it represents the process of the software development process.

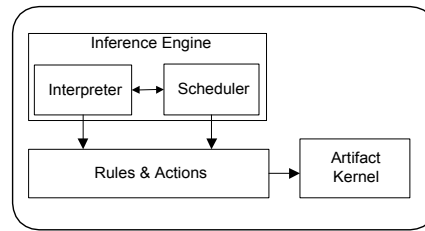


Figure 2. Agent-based artifact model

Our method engineering framework is based on this model and each artifact type is represented using this model. We make a distinction between *artifact classes* and *artifact instances*. Artifact classes define the basic artifact kernel, production rules and inference engine for a set of artifact instances. Artifact classes can be instantiated into artifact *instances* which share the specified behavior but differ with respect to their specific states. In fact, this is just similar to the distinction between a class and object in object-oriented programming languages. The artifact types are derived from the given method description [20].

5. Engineering agile methods

In our previous work we have formalized object-oriented methods which have a rather rigid process [20]. In this paper we aim to formalize Extreme Programming (XP) which can be considered as a prominent approach of agile software development approaches. In the following we will first describe the basic artifacts of XP, define its heuristic rules and the related process.

5.1 Artifacts

We can identify the following basic artifacts in XP:

User Stories: User Stories are in the format of about three sentences of text written by the customer in the customers terminology. They are used to represent the requirements and create time estimates for the release planning meeting.

Iteration plan: An iteration plan is a collection of user stories selected by the customer according to a budget established by the developers. It represents a minor delivery and is usually two weeks in length.

Release plan: A release plan maps out a set of (e.g. six) iteration plans consisting of prioritized collections of user stories. It is usually three months of work and represents a major delivery that can be usually put into production.

Task: Tasks are sub-parts of user stories, which can be developed in the order that makes the most technical and business sense. Tasks are determined during the iteration planning.

Metaphor: A metaphor represents a system of names that represent the big picture of the whole

system. The names provide a vocabulary for elements in the system and help defining their relationships.

Spike: a spike solution is a very simple program to explore potential solutions.

Unit Test: Unit tests are white-box tests that verify the internal structure the system.

Acceptance Test: Acceptance tests are black-box tests and are created from user stories by the customer. Each acceptance test represents some expected result from the system.

Code: Represents the implementation of selected user stories.

Note that XP does not describe any other design artifacts explicitly such as class diagram, sequence diagram, use case diagram etc.

5.2 Heuristic Rules

Heuristic rules represent the production rules. Agile software development provides a focus on customer. Therefore for each rule we specify the person responsible who is responsible for the rule, which can be either the customer (C) or the developer (D). The customer is the person or group who defines and prioritizes features. These can be business analysts, marketing specialists, user representative or the paying customer. The developer can be the designer or the programmer.

User Stories

- R1. C: Write user stories.
- R2. C: Cancel user story.
- R3. C: Define/change priority of user story.
- R4. D: Provide for each user story the time estimation for development.
- R5. D: if user story is too detailed then merge related user stories.
- R6. D: if user story is too general then split into different user stories.

Task

- R7. D: Define tasks by splitting user stories (during iteration).

Spike

- R8. D: If user story seems risky, implement a spike solution.

Metaphor

- R9. D: Develop a related metaphor.

Acceptance Test

- R10. C: Develop one or more acceptance test for each user story.
- R11. DC: Check acceptance test.

Release plan

- R12. C: Select important user stories for next release using metaphor.
- R13. D: Develop release plan consisting of several iteration plans (consisting of selected user stories).
- R14. D: Re-estimate release plan based on any spike solutions.

Iteration plan

- R15. DC: Select important user stories from release plan for current iteration for given budget

Unit Test

- R16. D: Create a unit test for given code.

Code

- R17. D: Implement selected tasks.
- R18. D: Check against related unit test.
- R19. D: Check against acceptance test.
- R20. D: Refactor code.
- R21. D: If code fails against unit test update code.
- R22. D: If unit test passed then include in release

5.3 Process

The artifacts in a given method are usually not isolated but *depend* on each other. An artifact is dependent on the other artifact if it needs it to produce artifact instances. This dependency usually follows directly from the method descriptions and the applied heuristic rules. To visualize the dependency relation we introduce the *artifact dependency graph*. In this graph each node represents an artifact whereas each arc represents a directed dependency. A dependency $a \rightarrow b$ means that b needs a to produce artifacts, or after production of a the production of b follows. An artifact may be dependent on more than one artifact. An example is, $a \rightarrow b$ and $c \rightarrow b$, which describes that b is both dependent on a and c. One artifact may be used to produce more artifacts. For example, $a \rightarrow b$, $a \rightarrow c$ which means that after the production of a both b and c may be produced. The artifact dependency graph for Extreme Programming is shown in Figure 3. Hereby, the nodes of the graph represent the artifacts and the arcs represent the heuristic rules for producing these artifacts. Note that heuristic rules for the customer are either written above or to the left of each arrow, whereas the heuristic rules for the developer are written below or to the right of each arc.

Besides of the dependencies among artifacts for defining the process we should also consider the ordering of various rules. For this we use the symbols ‘;’ or ‘||’ denoting sequence and parallel ordering of rules respectively:

R1;R2	R1 is executed before R2
R1 R2	R1 and R2 can be executed in parallel

Rule orderings take place across (inter) and within (intra) artifacts. The ordering across artifacts is basically determined by the artifact dependencies. The ordering within artifacts is dependent on the semantics of the corresponding rules.

For XP we can denote the rule ordering across artifacts as follows:

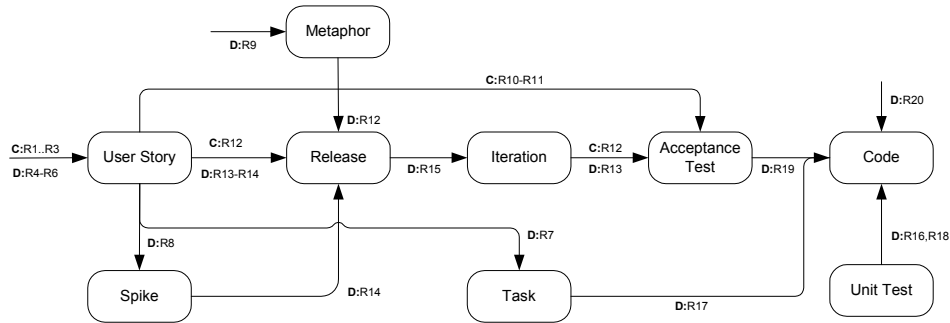


Figure 3. Artifact Dependency Graph of Extreme Programming

(C:R1 || C:R2 || C:R3) || (...)

This means that the application of rules R1 to R3 can be applied by the customer in parallel to any other rule. More specifically, user stories can be added/changed/removed at any time during the software development process. Due to space limitations we do not provide an elaborate specification of the rule ordering.

6. Discussion

Our basic concern in this paper is not to provide a complete formalization of XP but rather to show that XP, a good representation of agile software development, can be formalized just as any other conventional method. In the previous section we have formalized the artifacts, the heuristics and the process of XP. Several interesting observations can be drawn from this study:

- *Heavy User Story Dependency*

It appears that many artifacts are dependent on user stories. The artifact *User Story* in Figure 3 has the dependent artifacts *User Story*, *Spike*, *Release*, *Task* and *Acceptance Test*. If the user stories are not well-defined this might immediately mean that the subsequent artifacts, including the code, will not be appropriate. User stories are similar to requirement specifications and/or use cases and represent the view of the customer. Requirements however might be imprecise, ambiguous and redundant and may not represent a good basis for solution abstractions. In XP it is indicated that detailed requirements analysis will be done if needed. For this, it seems that the developer will still need to use existing requirements analysis methods.

- *Implicit role of Metaphor*

The role of metaphor is somehow implicit. Metaphor should play almost a similar overall role of *software architecture*. However from the method descriptions on XP we can not derive explicit rules for how to define this metaphor. It is assumed that the right metaphor is provided early in the development process and that this will be an

appropriate guidance for the development of the subsequent artifacts.

- *No explicit software architecture*

Besides of an implicit metaphor there is no explicit artifact or rules for defining the software architecture of a system [3][18]. Software architecture is generally considered to play a fundamental role in coping with the inherent difficulties of the development of large-scale and complex software systems. A common assumption is that architecture design should support the required software system qualities such as robustness, adaptability, reusability and maintainability [1]. In XP it is tacitly assumed that the architecture of the system will emerge during the software development process. This might have the risk that the project becomes unmanageable since there is no explicit overall guidance role which the software architecture usually fulfills.

- *Explicit code and test driven, implicit design*

It is well-known that agile software development puts the first focus on code and less on documentation. The documentation consists basically of test units. This might ensure that the code is well-documented and tested; however design seems almost completely implicit. For example, XP does not provide any explicit rules for identifying artifacts in the design, such as for example class, attribute, association, inheritance subsystem etc. It has one rule (R20) on refactoring which implicitly includes these kinds of heuristics. Because this is implicit it heavily depends on the experience and background of the software engineer who does the refactoring. If the team that applies XP is not well-trained and is not aware of basic design skills this might lead to ill-defined designs and ill-defined less stable code.

- *Heavily Practice-Oriented*

The agile process of XP is basically caused by the reduced set of artifacts and heuristic rules. Rather than rule-oriented we can say that XP is practice-oriented. Some practices that we have not explicitly described before are for example *pair*

programming, collective ownership, continuous integration, 40-hour week, on-site customer etc [5]. These practices might be useful and as such complementary to existing software design methods. However, it is very difficult to formalize these practices and likewise automating XP requires more effort than conventional heavyweight processes [18].

- *Heavily Human-Oriented*

XP puts lots of responsibility to humans in the software development process. The customer is actively involved in the process. This might really help to clarify the customer needs. Further much dependency is put to the developer who is assumed to have sufficient design refactoring skills. Developers can also select tasks themselves even if their background for completing the task does not match exactly. As such XP has the characteristic of a learning system. It is understandable that some might find this too risky especially if we are dealing with life and safety critical systems.

- *Implicit quality factors*

The basic claims of XP and agile software development in general is that it claims to produce software fast and in a flexible way. It appears that indeed XP can be successful in these matters. However, in general software development also should explicitly consider other quality factors such as adaptability, reusability and maintainability. In our experience with XP this is somehow implicit. Again it is implicitly assumed that this is done in the refactoring phase.

7. Conclusion

In this paper we have defined a software design method as consisting of artifact types, method rules and process. Based on this definition we have developed the ARTIBOT model, an agent-based model to actively guide the software engineer in developing software. We have identified the various artifact types in Extreme Programming (XP) a prominent agile software development method and identified its rules and the related process. Our study shows that similar to traditional software development methods also agile software development method shows some rigorous steps. We were able to identify the method rules and the logical ordering of these method rules as described in the method descriptions of XP.

The results provide an objective basis to compare XP with more formal software development methods. Comparing to our previous work on formalizing object-oriented method we can conclude that XP has far less artifacts and rules.

Because agile software development is heavily human and practice-oriented this has of course

made the formalization and the automation [19] of the method very difficult. We have not explicitly discussed the practices of XP, which might be useful. From our study, however, we can conclude that too much reliance on practices and human beings makes many steps in the software development process implicit. We have seen that there are no explicit artifacts and rules for defining the software architecture and the artifacts of the object-oriented design such as class, attribute, inheritance, etc. Moreover, although agile software development focuses on providing software fast, the other important quality factors remain implicit. XP might be very successful for programming-in-the-small but it is in our opinion questionable whether XP can be really successful for developing programming-in-the-large.

Acknowledgements

I would like to thank professor Mehmet Aksit with whom I have earlier worked on formalization of object-oriented methods and the method engineering framework as presented in this paper.

References

- [1] Akşit, M. *Software Architecture and Component Technology: The state of the art on Research and Practice*, Kluwer Academic Publishers, 2001.
- [2] Arrango, G: Domain Analysis Methods, in: *Software Engineering Reusability*, R. Schafer, R. Prieto-Diaz, & M. Matsumoto (eds.), Ellis Horwood, 1994.
- [3] Bass, L., Clements, P., & Kazman, R. *Software Architecture in Practice*, Addison-Wesley 1998.
- [4] Beedle, M. & Schwaber, K.. *Agile Software Development With SCRUM*. Upper Saddle River, New Jersey: Prentice Hall, Inc, 2001.
- [5] Beck, K: *Extreme Programming Explained*, Addison-Wesley, 2000.
- [6] Cockburn, A. *Crystal Clear: A Human-Powered Software Development Methodology for Small Teams*. <http://members.aol.com/humansandt/crystal/clear/>, 2001.
- [7] Cockburn, A: *Agile Software Development*, Addison-Wesley Longman, 2001.
- [8] Ellis, J.R. *Objectifying Real-Time Systems*, SIGS Books, 1994.
- [9] Gane, C: *Computer-Aided Software Engineering: The Methodologies, the Products, and the Future*, Englewood Cliffs, NJ: Prentice Hall, 1990.
- [10] Highsmith, J.A.. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. New York: Dorset House Publishing, 2000.

- [11] Jacobson, I., Booch, G. & Rumbaugh, J: *The Unified Software Development Process*, Addison-Wesley, 1999.
- [12] Kruchten, P. *The Rational Unified Process: An Introduction*, Addison-Wesley, 2000.
- [13] Paulk, M.C., Curtis, B., Chrissis, M.B., Weber, C.V. *Capability Maturity Model*, Version 1.1. *IEEE Software*, 10, 4, pp. 18-27, 1993.
- [14] Paulk, M.C. *Extreme Programming from a CMM Perspective*, in Proc. of XP Universe Conference, Raleigh, NC, 23-25 July, 2001.
- [15] Riel, A. *Object Oriented Design Heuristics*, Addison-Wesley, 1996.
- [16] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., & Lorenzen, W: *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
- [17] Saeki, M. *Method Engineering*. in: P. Navrat and H. Ueno (Eds.), *Knowledge-Based Software Engineering*, IOS Press, 1998.
- [18] Tekinerdoğan, B. *Synthesis-Based Software Architecture Design*. PhD Thesis, Dept. of Computer Science, University of Twente, The Netherlands, 2000.
- [19] Tekinerdogan, B., Saeki, M., Broek, P., Sunyé, G., & Hruby, P. *Automating Object-Oriented Software Development Methods*. in: A. Frohner, *Object-Oriented Technology, ECOOP 2001 Workshop Reader, Lecture Notes in Computer Science*. Vol. 2323, Springer Verlag, 2002.
- [20] Tekinerdogan, B., & Aksit, M. *Providing automatic support for heuristic rules of methods*. In: Demeyer, S., & Bosch, J. (eds.), *Object-Oriented Technology, ECOOP '98 Workshop Reader, LNCS 1543*, Springer-Verlag, pp. 496-499, 1999.
- [21] Wieringa, R.J. *Requirements Engineering: Frameworks for Understanding*. Wiley, January 1996.