A Decision Making Support Method in Agile Software Development

Neli Maneva

Institute of Mathematics and Informatics, BAS, Sofia
neman@math.bas.bg

Abstract: The paper describes how a method, supporting decision making, can be used during software development in an agile framework. The method of Comparative analysis is briefly presented. An incremental approach to object modeling is explained and some models, constructed for a number of intermediate products (requirements, user stories, designs, programs and increments) are shown. The application of the method is illustrated by a few examples.

Ключови думи: agile software development, decision making, quality object models

ACM 1998 Classification Keywords: D.2: Software Engineering, D.2.8 Metrics, D2.9 Management

1. Introduction

Agile software development appears as an alternative to conventional development, trying to overcome some of its stated deficiencies and responding to the need of a new approach. Nowadays there is a big family of many and different agile methodologies, sharing the key ideas of the signed in 2001 Manifesto for Agile software development [1], but proposing some specific guidelines for their implementation.

In Agile software development requirements and solutions evolve by collaboration between members of self-organizing, cross-functional teams. Adaptive planning, evolutionary development, early delivery, continuous improvement and encouragement of rapid and flexible response to change are promoted [2]. This new development paradigm influences the content and performance of any activity, including decision making [3]. From agile perspective, decisions have to be made just in time, quickly, on the base of all available information, using appropriate methods and tools.

In this paper we describe how a rigorous decision support method can be used during agile software development. The rest of the paper is organized as follows. Section 2 describes the method of Comparative analysis. In Section 3 the most difficult part of method application – the object modeling – is presented and some generic quality models for basic objects are given. In Section 4 there are a few examples of cases, describing the context of the Comparative analysis. Some ideas for future research and development work are mentioned in Conclusion.
2. The method of Comparative Analysis

Let us present briefly a formal method for a reasonable choice. It supports the multi-criteria decision making and has been developed, taking into account the best achievements in this area [4].

Comparative analysis (CA) is a study of the quality content of a set of homogeneous objects and their mutual comparison so as to select the best, to rank them or to classify each object to one of the predefined quality categories.

The compared objects can be products, processes or resources, identified as significant for the activity under consideration. For CA practical use, we distinguish two main players: the Analyst, responsible for all aspects of CA implementation, and a CA Customer – a single person or a group of individuals, who should make a decision in a given situation and would like to use the CA as a supporting method.

Depending on the identified problem to be solved by a CA Customer at a given moment, a case should be opened to determine the context of the desired comparative analysis. Each case is specified by the following six elements:

\[
\text{case} = \{\text{View}, \text{Goal}, \text{Object}, \text{Competitors}, \text{Task}, \text{Level}\}
\]

The View describes the CA Customer’s role and the perspective from which the comparative analysis will be performed. The Analyst has to identify the potential decision makers. For example, in Scrum software development the following roles of CA Customer can be considered: Product owner, Scrum master, Member of the development team, Stakeholders and Agile mentor. Taking into account the responsibilities and typical tasks of these players, several cases will be defined, considering their respective point of view to the analyzed problem.

The Goal expresses the main Customer’s intentions in CA accomplishment and can be to describe, analyze, estimate, improve, or any other, formulated by the Customer, defining the case.

The Object represents the item under consideration. For each object in CA application a quality model should be created – a set of characteristics, selected to represent the quality content in this context, and the relationships among them.

According to the stated Goal, the instances of the objects to be compared are selected and described in the set of Competitors – \(C = (C_1, C_2, \ldots, C_n)\).

The element Task of a case can be Selection (finding the best), Ranking (producing an ordered list), Classification (splitting the competitors to a few preliminary defined quality groups) or any combination of them.

The Depth Level defines the overall complexity (simple, medium or high) of the CA and depends on the importance of the problem under consideration and on the resources needed for CA implementation.

A step-wise procedure for the CA accomplishing has been proposed and successfully used in a number of application areas. Next follows a brief description of the steps.

The first step is Initialization. When a problem, for which the CA seems to be useful, appears, the Customer creates a CA request – an informal description of the problem situation, moment and area of consideration. This CA Request is studied by
the Analyst, who evaluates its conceptual and technical feasibility. When the CA request is approved, a new case is opened and its elements View, Goal, Object, Competitors, Task and Level are defined, using some relevant sources of information.

During the Construction a hierarchical object model is created and the weights of all included in the model quality characteristics are defined. Then the method for accomplishing the Task (selection, ranking or classification) is selected, taking into account the defined Task and depth Level.

During the Execution stage the Analyst evaluates each object from the set of Competitors and fills the objects-factors table, applies the selected method for implementing the CA task and documents the obtained results.

The Completion stage comprises preparation of the Final report, describing and saving some elements for re-use and collecting historical data to be available for further processing.

Each step covers a number of activities. For each activity its aim, the results to be obtained and how the activity must be performed have been specified. In this way it is clear who has to do what and when, thus making the comparative analysis manageable and properly utilized.

3. Quality models for some intermediate products in agile software development

The case-driven CA begins with construction of the hierarchical object quality model. The first level comprises m quality factors F1, F2, ..., Fm. They characterize the local state of the object as regards the defined case. Each factor Fj must be weighed in accordance with its relative importance. Depending on the defined depth level and the cognitive complexity of the factors, the latter can be further decomposed. The obtained hierarchical structure describes which quality characteristics will be considered at different levels and what the relationships among them will be.

One of the most challenging problems in CA implementation is the construction of a quality model for a given object. Our suggestion is to apply an incremental approach. When a new object appears in a case definition, the Analyst has to create its first model, using different sources of information. The Customer’s requirements for the studied object should be stated and further analyzed so as to be reformulated in terms of quality characteristics, described in a hierarchical model. Then the created model is saved as a generic (basic) model for the object under consideration. When a request for CA with the same object arises, the generic model is found and is made available for modification in two directions – expanding the generic model and constructing its derivative image, relevant to the defined case. First, there should be an attempt to find some new quality characteristics to be added at some levels of the hierarchy so as to reflect a new view or goal of the current case. If the attempt is successful, the expanded hierarchical structure will be stored as a generic model. The second step is to decide whether some characteristics can be ignored as irrelevant to the current case. So a derivative of the model is obtained. A special pattern,
describing the deviation from the basic model is created and saved for future re-use with reference to the case.

In order to facilitate the CA application, a few quality models for intermediate products in agile development have been proposed. They can be used as generic ones and further modified to reflect the peculiarities of the defined situation.

3.1. Quality model for object “requirement”
Agile Requirements Engineering comprises elicitation, validation, estimation and prioritization of requirements at different levels [5]. The Comparative analysis can support decision making at some situations appeared when these activities have been performed.

A simple linear model for object “requirement” has been created. According to this model the requirement should be:
- **Unambiguous** – to possess the capability to have a unique interpretation,
- **Testable** – the fulfillment of each requirement is measurable or testable,
- **Consistent** – there no contradictory statements in requirement description,
- **Traceable** – each requirement has a rationale,
- **Understandable** – requirement is fully understood when used for developing software,
- **Complete** – it is possible to make references to precisely identified entities,
- **Detailed** – the description of requirement is given in detail and it is traceable to user needs.

3.2. Quality model for object “set of requirements”
A simple linear model for object “set of requirements” has been created. The selected characteristics are:
- **Completeness** – i.e. the set \( R \) comprises all requirements, necessary to describe the chosen functionality of a system or a system’s compound element;
- **Consistency** – the involved in the set \( R \) requirements are in reasonable and logical harmony, without any contradictions in their content;
- **Relevance** – all included in \( R \) requirements should possess a direct and clearly identified connection to the studied system’s functionality.

3.3. Quality model for object “user story”
In agile software development user stories are small units of work. They are a few sentences in simple language that outline the desired functionality. The goal of a user story is to deliver a particular value back to the customer.

Usually the user stories are sketched out by the Customer/Product owner. Having such initial version, the entire development team collectively decides about more detailed descriptions. They are the granular pieces of work that help define the implementation items for the story and the upcoming iteration.
A two-level quality model for object “user story” has been constructed (see Table 1).

<table>
<thead>
<tr>
<th>Factors</th>
<th>Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accuracy</td>
<td></td>
</tr>
<tr>
<td>Trackability</td>
<td></td>
</tr>
<tr>
<td>Adequacy</td>
<td>Robustness</td>
</tr>
<tr>
<td></td>
<td>Completeness</td>
</tr>
<tr>
<td></td>
<td>Correctness</td>
</tr>
<tr>
<td>Understandability</td>
<td>Self-descriptiveness</td>
</tr>
<tr>
<td></td>
<td>Conciseness</td>
</tr>
</tbody>
</table>

3.4. Quality model for object “design of increment”

One of the most important characteristics of agile development is its incremental nature – working software is delivered fast and frequently. As noted in [6] such short, focused iterations make the entire agile development cycle much more like a maintenance phase.

Bearing in mind this, for object “design of increment” we propose a linear model, comprising 4 characteristics: conceptual integrity, effectiveness, maintainability and correctness. Their explanation is given in Table 2.

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conceptual Integrity</td>
<td>Conceptual integrity defines the consistency and coherence of the overall design. This includes the way that components or modules are designed.</td>
</tr>
<tr>
<td>Effectiveness</td>
<td>The degree to which a design is able to achieve the desired functionality and behavior.</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Maintainability is the ability of the system to undergo changes with a degree of ease. These changes could impact components, services, features, and interfaces when adding or changing the functionality, fixing errors, and meeting new business requirements.</td>
</tr>
<tr>
<td>Correctness</td>
<td>Extent to which the design conforms to its specifications and declared objectives.</td>
</tr>
</tbody>
</table>

3.5. Quality model for object “intermediate product”

The agile development process is iterative with several releases of intermediate products approaching the final one. Each iteration produces intermediate deployable code that has been discussed, designed, implemented, and tested. An increment is a
subset of the final product under development and builds on the functionality of the prior iteration, reflecting the revised set of requirements. That is why for the object “intermediate product” we propose a quality model, comprising the characteristics from the Product Revision perspective (ability to undergo changes) in the McCall Quality model as described in [7]. The selected quality factors and criteria are shown in Table 3.

<table>
<thead>
<tr>
<th>Quality factors</th>
<th>Explanation</th>
<th>Quality criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maintainability</td>
<td>The capability of the software product to be modified so as to incorporate corrections, improvements or adaptions to changes in environment and requirements.</td>
<td>Simplicity, Conciseness, Self-descriptiveness, Modularity</td>
</tr>
<tr>
<td>Flexibility</td>
<td>Ease of changing the software to meet revised requirements.</td>
<td>Self-descriptiveness, Expandability, Generality</td>
</tr>
<tr>
<td>Testability</td>
<td>Ease of validation that software meets requirements.</td>
<td>Simplicity, Instrumentation, Self-descriptiveness, Modularity</td>
</tr>
</tbody>
</table>

3.6. Quality model for object “program”

It is well known that quality is not something that can be easily added later. So it is better to ensure quality from the very beginning, writing quality code. Analysis of the code quality can be done on the base of different quality models. As an example, let us describe a number of code characteristics, proposed in [8]. Next follows a brief explanation of each code property and a comment about its measurability:

**Conciseness:** A program is said to be concise when its meaning is expressed in a small volume of text.

To estimate conciseness, one can start with a simple metric of program size, such as number of tokens in a program. But conciseness is essentially a relative measure: to tell if, or how much concise a program is, one has to compare sizes of different programs that solve the same problem by possibly using different methods and tools, including different programming languages.

**Straightforwardness:** Straightforward expression means that the programming language is used in a directly understandable way, according to generally accepted rules, including idioms.

As the notion of straightforwardness reflects the users' opinion, it can only be measured indirectly, through asking experts and using a scale for evaluation.

**Adequacy:** Faithful representation of the abstract model of a solution by the (text of the) program.
Unless a formal modelling tool exists in which the abstract solution is described, and a method that compares the program solution against its model, adequacy can only be estimated vaguely.

**Explicitness:** A property of a program object, or a relation between such objects is explicit when it is expressed within the program (as opposed to only assumed).

Cannot be measured in general, since explicitness, as well as the inverse property – non-explicitness – is something that relates a program text with the programmer's intention. However, expert estimation can be used in place of formal measurement method. And true measurement is still possible for several isolated manifestations of this property.

**Apparency:** Visual information is apparent when it can be readily grasped. Besides visual clarity of the information itself, certain preparedness (w.r.t. knowledge) is implied on behalf of the observer.

Not measurable (except my means of expert estimation).

**Memory load:** With respect to certain program fragment, this is the amount of ‘memory’ required, related to mentally interpreting the fragment, as well as knowing of other elements of the program, required to understand the fragment.

Applicable to small fragments, and can be measured based on the internal structure of the text, along with its relations with the rest of the program. For example, a highly hierarchical construct requires more memory load in order to be interpreted, as opposed to a ‘flat’, or sequentially interpreted construct.

**Notationality/verbality:** This concerns whether a text is primarily symbolic or verbal.

Can be measured by comparing the number of symbolic or verbal tokens to the total number of tokens in a program. It is more a characteristic of the programming language than of particular programs written in it.

**Mnemonicity:** The ability of program tokens to convey meaning by making use of associations with something known.

This property, although rather important in practice, is too vague and abstract to admit measuring.

**Uniformity:** Sameness or similarity of expressing similar ideas in a language.

This too characterizes not only particular programs, but the programming language as a whole. It is hard to devise a general measure but measuring with respect to simple benchmarks is feasible.

**Visual characteristics and attributes:** May include form, layout, special graphics, colour, fonts and typography, etc.

As should be apparent, these are not meant to be measured even informally.

Some of the above described characteristics can be studied as significant for understandability and maintainability of a program. Our suggestion for a code quality model is to use a linear model, comprising 5 characteristics, namely **Conciseness, Straightforwardness, Adequacy, Explicitness** and **Uniformity**.
4. How the Comparative Analysis can support the decision making in agile software development

Generally speaking, CA can be used in any situation, in which a problem is identified and a case can be defined, describing the context of the CA – when, who and how makes a decision.

Example 1: How the CA can be used for requirements prioritization

One difficult problem in requirement engineering is to create an ordered list of all elucidated requirements. Such prioritization will facilitate further requirements analysis so as to be accomplished in a more systematic and efficient way.

A number of cases for this problem can be defined as a combination of the following elements:

- **View** – that of the Product Owner, a stakeholder, a member of the development team;
- **Goal** – to compare the requirements, selected for prioritization;
- **Object** – a requirement;
- **Competitors** – all requirements, chosen to be prioritized;
- **Task** – ranking so as to produce an ordered list of the compared requirements;
- **Level** – any chosen – simple, middle or high.

Example 2: How the CA can be used for analysis of a set of requirements

- **View** – that of the Product Owner, a customer or a member of the development team;
- **Goal** – to analyze a set of requirements before a new iteration;
- **Object** – a set of requirements
- **Competitors** – only one element – the constructed set;
- **Task** – evaluating the quality of the set of requirements;
- **Level** – simple

Example 3: How the CA can be used for evaluating the effect of refactoring

Refactoring is an agile practice which deals with changing the design or structure of the code without changing its results. Refactoring involves rewriting the code to improve its understandability and maintainability, while explicitly preserving its behavior.

- **View** – that of the developer, coding the program
- **Goal** – to compare code quality before and after refactoring;
- **Object** – a program
- **Competitors** – two versions of the same program – before and after refactoring;
**Task** – ranking  
**Level** – middle.

**Example 4: How the CA can be used for communication strategies selection**

The Product owner wants to compare different forms of communication with stakeholders to be used during an agile project.

**View** – that of the Product owner;

**Goal** – to compare the communication strategies, identified as appropriate

**Object** – a communication strategy

**Competitors** – 9 communication strategies, chosen to be compared [9], namely Face to face (F2F), F2F at Whiteboard, Email, Online chat, Teleconference calls, Videoconferencing, Overview diagrams, Overview documentation, Detailed Documentation.

**Task** – ranking so as to produce an ordered list of the compared strategies;

**Level** – simple.

A linear model for the object “communication strategy” has been created, comprising only two characteristics – *effectiveness* and *efficiency*.

**Conclusions**

The main goal of the present paper is to describe the method of Comparative analysis and how it can support the decision making during the agile software development. One of the most difficult activity – object quality modelling – has been illustrated by presenting some models for a number of objects – requirement, user story, set of requirements, design, program and product. As example, four CA cases in Scrum framework for software development have been given.

Some possible directions for future research and development work can be:

− To identify and describe entirely some other CA situations during agile software development, in which the decision making is crucial;
− To examine the usefulness of the CA in a real-life agile project, expanding the re-use library with many different items – cases, models of other objects, etc.;

**References**

1. http://agilemanifesto.org/ Agile Manifesto
Метод, подпомагащ вземането на решения 
при гъвкаво разработване на софтуер

Нели Манева

Резюме: Статията описва как метод, подпомагащ вземането на решения, може да се използва при гъвкаво разработване на софтуер. Методът на Сравнителния анализ е представен накратко, заедно с процедура за използването му. За една от най-трудните дейности – моделиране на изследвания обект, е обяснен предлаганият инкрементален подход и са описани модели, конструирани за основни междинни продукти: изисквания, потребителски истории, проекти, програми и частични продукти. Няколко полезни приложения на метода са илюстрирани с примери.