Requirements Honesty

Francisco A. C. Pinheiro
Universidade de Brasília
facp@cic.unb.br

Abstract

This article discusses issues related to the inconsistency between requirements principles and the need for faster and faster ways of developing software. Requirements principles are related to the purpose of the system and to the appropriateness of requirements that correctly describe what is necessary for the system to fulfil its objectives. I argue that the quest for speed in software development may have the undesirable effect of weakening these principles.

Since the beginnings of software engineering, there is a search for faster ways to develop software. Many techniques and development models have been proposed that contribute for shortening development time, although the reduction in time comes almost as a side effect, as a result of improving some key aspect of software development. Agile methods are the first to place time-to-market as the prominent feature. The risk is to view other quality features as secondary.

1. Introduction

Requirements honesty is the adherence to certain requirements principles. I argue that the quest for faster and faster ways of developing software may impair the satisfaction of these principles. The argument starts with the consideration of two questions: "why would someone want a functionality in one to three weeks time?" and "why that same person is unwilling to wait two to three months to have the same functionality, if the resulting software is proved to be more appropriate, stable, or easier to maintain?".

Several answers can be given. One could stress the point that is very difficult to assure appropriateness, stability, and other quality factors. If no guarantee can be given that these quality factors are better preserved in a two months development time, then there is no point in taking the longer path to get similar uncertainties. Another answer could note that for some systems the cost of assuring quality factors in a longer time-scale outweighs the benefits. This line of reasoning does not diminish the importance of producing quality products. It just recognises that there are costs involved and difficulties to overcome, and that these should be taken into consideration when deciding the amount of effort devoted to assure quality.

A second line of reasoning takes the market as the driving force. The need to produce in a short time is derived from the need to be the first in the market. It is a business requirement and a very strong one because it is perceived as related to survivability. I present time-to-market as a mandatory requirement, implying a trade-off between development speed and other quality characteristics that are also important to business survival.

This article discusses the impact of the agile methods on requirements research. The discussion is focused on the concepts and practices of extreme programming (XP [2]), taken as a representative of the agile methods. It is an iterative process built around a short iteration length of one to three weeks. The short delivery time is central to the model. Other XP practices and artefacts like user stories, pairwise programming, on-site customer, and the test-first approach are organised to maintain the increments being delivered in a constant speed within that time. I argue that this time-oriented approach is dictated by market and may have a negative impact on the requirements principles.

This market orientation is very strong. We already have several evidences of the wide acceptance of XP concepts. Well known market players are moving towards incorporating agile concepts. McCormick [15] comments on the adaptation of RUP to embrace XP practices and Paulk [24] presents CMM and XP as complementary, exhorting people to consider XP if they are interested in process improvement, and to consider the CMM management infrastructure if they want agility. Requirements engineering is also adopting the new terminology. A search on the Internet produces dozens of references on "extreme requirements", from consulting services [26] to academic proposals [9]. Even the characterisation of people discussing XP seems to be market oriented. Booch [3] quotes Dick Fairley as calling the XP supporters athenians, or entrepreneurs, as opposed to the disciplined software engineers, the spartans. We also have some academic exercises showing that

XP may be the right choice for small projects where the time-to-market is the most important factor [19].

The quest for quicker ways to develop software is natural. Many development models and software techniques contribute in a positive way for shortening development time. However, the reduction in time comes almost as a side effect, as a result of doing better what should be done. In requirements terms, what should be done is characterised as the adherence to the requirements principles of purposefulness, appropriateness, and truthfulness: engineering the requirements that appropriately describe what is necessary for a system to fulfil its objectives.

I see the agile methods as the first to strongly raise time to the status of a quality factor and to take time-to-market as the governing force of software development. This is dangerous for requirements principles because the prevalence of time in these methods is independent of the system being developed; thus, considerations of purpose, adequacy, and correctness may become secondary.

After discussing these issues, I suggest that research on requirements should remain focused on the maintenance of the requirements principles, with reduction in development time coming as a consequence. In situations in which this orientation is not possible, requirements research should help to clarify the possible losses. I also make some questions to be considered when investigating to what extent the requirements principles can be preserved in agile, market-oriented environments.

The requirements principles I use to build the argument are presented in Section 2. Section 3 discusses the quest for agility in software development and presents some proposals that, although contributing in a positive way to reduce development time, are based on the preservation of the requirements principles. The basics of XP are presented in Section 4 and a critique of its concepts is developed in Section 5. The links between agile methods and market are discussed in Section 6, where I also present possible directions for research. Section 7 contains my conclusion.

2 Requirements principles

Elicitation, analysis, and validation are at the heart of the requirements process. The determination of what is to be achieved and of what is required to accomplish the objectives is a key aspect of software development. A careful process of study, understanding, and analysis of requirements is necessary to deal with the complexities of the requirements elicitation. A validation procedure is essential because if we do not know whether we have the right requirements, then we also do not know if software built to meet these requirements will fulfil its objectives. These activities are associated with the following requirements principles:

Purposefulness. There should be an objective to be fulfilled.

Appropriateness. Requirements should be appropriate to the system. They should express what is necessary to achieve the system's objectives.

Truthfulness. Requirements should express what is actually required.

The first two principles are closer to elicitation and analysis, and the third to validation. These principles are sometimes related to requirements properties like correctness and consistency. The ideas behind them are present in old and recent surveys. Roman [27] uses the term *appropriateness* referring to requirements specification. Nuseibeh and Easterbrook [22] say that one of the most important goals of elicitation is to find out what problem needs to be solved. They also rightly present validation as a very difficult problem concerning truth and knowledge. Similar points, mainly related to the importance of goal discovery and modelling, are made by van Lamsweerde [31].

2.1. Purposefulness

We build software for some reason. If that reason is fulfilled then the software is successfully built. Of course this idea of success is subjected to ethical considerations. One may, as I do, reject as successful a system built to satisfy unethical or immoral objectives. The point here is that there is always a purpose.

Software requirements express part of what is necessary to achieve the system's objectives by means of software — we may say that software requirements express the software's contribution to the achievement of these objectives.

The determination of the system's objectives is a difficult task, aggravated in situations in which they are not, or cannot be, clearly stated. Usually there are several objectives satisfying different interests, and determining the right ones is hard. To complicate things a bit more, the system may have unclear or no absolute goals. However, there always will be a negotiated construction of the objectives and the system will always serve some purpose.

2.2. Appropriateness

Software is just a piece of larger systems; thus, it has to be appropriate to the system in which it is inserted.

Determining the software appropriateness amounts to getting the right requirements. In situations in which the objectives are unclear or there are no absolute objectives, the determination of the right requirements is more difficult. In those situations, defining what is needed for a software to contribute to the accomplishment of the objectives

comes *pari pasu* with the determination of the objectives themselves.

One of the consequences for the development process is that no user may be regarded as having all the necessary information. Negotiation is usually necessary to define the appropriate requirements and this involves listening to all parties playing a role in the negotiation process.

2.3. Truthfulness

The assurance that we have the right requirements and subsequently the assurance that we get the requirements right are paramount for the development of software. For complex systems the number of mistakes and errors, and the number of them that may go unnoticed, is large. There are two ways to deal with those mistakes and errors on requirements: one is through requirements analysis and the other is through a process of verification and validation.

Requirements analysis try to establish the presence of desirable attributes like consistency, precision, and completeness, among others. Although these desirable properties are no guarantee that the requirements are correct, their presence is a good indication in that direction and the lack of them is a sign that something is wrong or improperly stated.

Validation is always a process of building confidence. It is concerned with the understanding of things and it is necessary if we do not want to walk in the dark. There are several ways to verify and validate requirements. For example, using inspections and walk-through sessions to discuss what was understood with other people, using scenarios to explore the consequences of something being true, using prototypes to show what was understood, or simply restating the requirements. Testing is a necessary step not only to identify errors introduced in the construction of the various artefacts, but also to build confidence on the truthfulness of the system — that the system is indeed what it should be.

3. Quick, fast, rapid, agile

The first two major conferences on software engineering [21, 25] had already stressed the fact that the techniques and methods of the time were inadequate to cope with the increasing demand for software products. This fuelled the search for quicker and faster ways of developing software. Automation of tasks, improved techniques, and new development processes were proposed. Rapid prototyping, joint application design, automated test case generation, and incremental development are just few examples.

Each one of the proposed models, methods, and techniques can be viewed as capable of abbreviating product delivery time. In common, their proponents advocate improvements in key areas of software development. Design patterns stress the use of proven solutions. Modularisation avoids propagation of errors and facilitates testing and maintenance. Rapid prototyping anticipates problems and reduces uncertainty. Formal methods preserve correctness and help to shorten the testing phase.

All of this have a positive effect on development time but the reduction in time comes almost as a side effect, in virtue of successfully addressing some key aspect of the development. I illustrate this point showing how prototyping, incremental development, and user involvement impact development time.

3.1. Prototyping

Prototypes are broadly classified in two kinds: the exploratory and the evolutionary prototypes. This section discusses the first one. The evolutionary prototype is discussed in the context of incremental development.

Exploratory prototypes are used to explore uncertain issues and to clarify questions related to them [7]. They are also known as throwaway prototypes because the whole idea is to dispose of it after the unclear aspects have been clarified. In practice the prototype is kept either as part of a specification, to illustrate the prototyped aspects, or just to register how required things came into existence.

The principle behind the use of prototypes is that they should be useful for uncovering hidden aspects of the system; for eliciting requirements. The reduction in development time comes as a consequence of the reduction of uncertainty. Therefore, a prototype is good only if it helps reducing the uncertainties. To this end, a lot of preparation is required. It is a common advice to have good programmers, skilled at the tools used for prototyping, and good users, having the necessary time to try the prototype and being able to give competent feedback.

Moreover, not everything that is unclear is a candidate for prototyping. If we suspect that the issue may be solved by interviewing the right people or observing how things work in a proper setting we may consider using these other kinds of elicitation techniques.

3.2. Incremental development

The use of prototypes leads naturally to a development based on successive prototypes [7, 23]. The idea is to build parts of the system as increments. The development process continues until the whole system is delivered.

The important feature in this form of development is the early delivery of useful functionalities. This is a great advantage and very valuable in many circumstances. We also have other possible benefits: by dividing the system in increments we may simplify the analysis tasks concerning

each increment, with a positive effect on construction and test.

The development time is not necessarily shorter: it is usually perceived as such because of the delivery of useful functionality early in the process. Besides, it is never implied that the time should lead the way. One must take the necessary time to set up an appropriate architecture and to conceive and build each increment. The requirements should be understood at least to a point where the overall functionality may be clustered, and the interface of each cluster is well defined.

Each increment is built to deliver the associated functionality and it should be prepared to work together with the next increments. A poorly understood system leads to a large amount of rewriting, with several problems. This could soon diminish any benefit gained from the functionalities delivered earlier.

3.3. Participatory development

Participatory development refers both to participatory design [17] and to the methods collectively known as FAST [4] (facilitated application specification techniques). They have characteristics making them useful to shorten the life cycle. The user involvement helps bringing the controversial and contradictory issues earlier in the process. These methods promote discussion with the interested parties, which helps to solve difficult issues. This accounts for a design that reflects more closely what is required, resulting in a better acceptance of the product.

The proponents of participatory development claim that the design and implementation phases are shortened and maintenance is reduced. They argue that the user involvement in the specification and design artefacts serves also as a process of validation. I do not discuss the possibility that the user acceptance comes in fact more because of the complicity built in the process than for any improved design outcome. For one reason or another, the verification and validation phases are made shorter.

3.4. The path to short delivery times

The above approaches adhere to the requirements principles in the sense that they try to preserve some of its associated qualities. There are assumptions related to purpose, appropriateness, and truthfulness that are key for their effective use. The prototypes should clarify obscure issues, the increments should be based on a well thought architecture, and the user involvement should be carefully prepared and conducted. A short development time is a consequence of doing what should be done in a better way.

Recently we have proposals explicitly intended to shorten the delivery time. This comes not as a consequence of improved methods and techniques for software design and construction. Quite the contrary, these recent proposals are explicitly devised to support the delivery of functionality in a reduced time length. Not surprisingly they are collectively known as *agile* methods, stressing the quest for velocity. In fact, agility requires a quick and easy move or reaction, but being quick is a prominent aspect of the definition of the word.

4. Extreme programming concepts

Agile methods are the first to place time-to-market as the prominent feature of software development. I discuss XP as a representative of the agile methods because of its wide acceptance. Also, it seems that many of its extremeness come from this time-to-market prominence.

The description of XP is based on [2, 36]. Its project structure, illustrated in Figure 1, is very simple. The project begins with a release meeting where a plan to deliver the software in increments is elaborated. The software is delivered through several iterations. Each one is planned in detail defining, for example, which functionalities should be added and which errors from previous iterations should be fixed. The increments are delivered into production line after passing the acceptance tests. The process continues with replanning and the necessary adjustments until a complete system is in operation. The main points of this process are summarised as follows:

The release plan is based on user stories, short pieces of text expressing in customers' terms what they want to get.

In each iteration a small set of user stories is chosen for implementation.

The design of the software should be kept simple. Only the functionalities being delivered receive attention in each iteration.

Programming is the heart of XP. The coding follows two principles: one is pairwise programming and the other is the test-first approach — to create tests before coding the functionality.

The acceptance tests are designed from the user stories as black-box tests. They are kept as part of a test suite to run every time an increment is delivered.

The unit tests are also black-box tests. Programmers write their own tests before they code.

Integration is responsibility of everyone programming the software. Each team integrating a piece of code is responsible for running the integration tests.

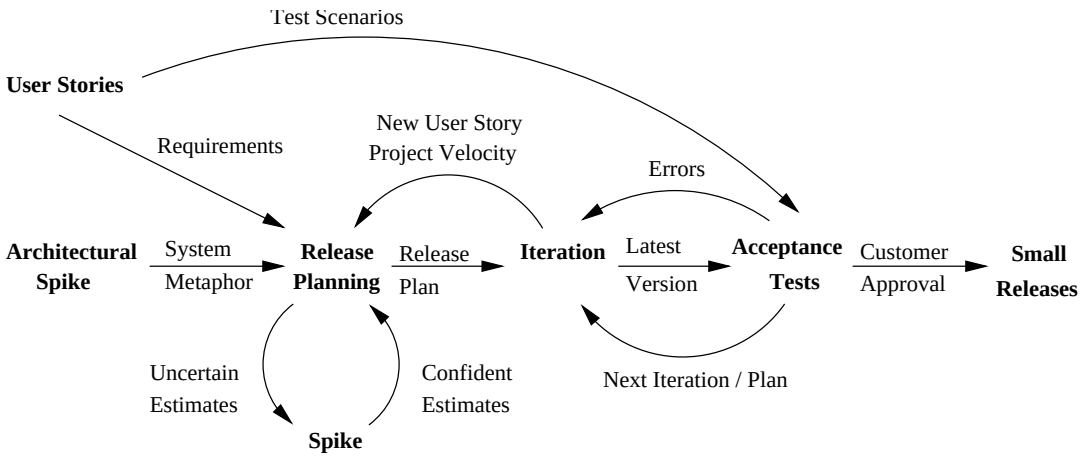

Figure 1. XP project structure (adapted from [36]).

4.1. User stories

User stories are the prominent artefact of XP. They are used to describe things the user wants to get, to elaborate time estimates for planning each release, and to develop unit and acceptance tests.

They are written by customers and express what the system needs to do for them. Each user story consists of two to three sentences of text. They should provide just enough detail to make possible the determination of the number of weeks it will take to implement them.

When time comes to the actual implementation, the programmers developing the functionalities associated with a user story talk to the customer and receive a detailed description of the requirements in face-to-face meetings.

4.2. Planning

A XP project is based on making frequent and constant small releases. Each release is developed in one or more iterations and the project is planned to have several iterations. There is a initial release plan laying out the overall project.

In each iteration a subset of the user stories is chosen to be implemented. Customers and developers choose the user stories taking into consideration their delivered value, the estimated time to implement, the length of the iteration and the project velocity.

The iterations are planned to have a fixed length and the team should strive to maintain this length constant throughout the project. A period from one to three weeks is considered ideal. The project velocity is calculated summing up the number of user stories or programming tasks finished in each iteration. This number is used to estimate how many user stories can be implemented in the next iteration.

4.3. Designing and coding

The design tasks are intended to develop a set of classes providing the services required to implement the user stories. There is a general guidance to make things simple and avoid unnecessary details. The proper place for details is during programming.

The programming tasks are the heart of XP approach. Programming is done by pairs of programmers. There are two necessary things: the user should be always available and the unit tests should be created first, with the actual code coming later.

4.4. Acceptance tests

In each iteration the user stories being built are translated into acceptance tests and incorporated into a set of regression tests. At the end of the iteration the entire release is tested, not only the increment being delivered.

Whenever an error is found during software operation, if not corrected right away, the error is scheduled for correction in future releases. An acceptance test to catch it is always added to the test suite.

5. The extremeness of extreme programming

If one point can be selected to characterise the XP process, it is the duration of its iterations. Other points like user stories, pair programming, and testing also have peculiar aspects, but they seem to be oriented and subjected to the delivery of increments in short-duration iterations.

The ideal duration of an iteration is one to three weeks. If we take this off and allow iterations of two to three months, then we need no extremeness any more. Except for a deliberate attempt to reduce documentation through user sto-

ries and face-to-face meetings, the XP practices may be conceivably applied in other development models without being considered extreme.

Most critiques of XP address the extremeness of its practices, showing them as taking to extreme lengths what would otherwise be considered normal. My position is different: the practices deserve attention and should be investigated but the main source of XP extremeness is the length of its iterations. This is extreme because it may hinder the requirements principles. If we allocate less and less time to perform some task we are certainly running into trouble, and to make this task simpler and simpler by way of compensation for the risk may not be the best solution. The following discussion is centred on the negative impact XP practices have on the requirements principles.

5.1. Selecting the right users

The choice of an appropriate customer to work on XP projects is recognised as a key factor of success. Ability to participate, communicate, and specify are some skills the on-site customers are expected to have [30].

The extremeness related to on-site customers is that they seem to be the only source of requirements. This goes against the purposefulness and appropriateness principles. Users, customers, and other requirements' sources should be chosen based on what is necessary to define the system's objectives and requirements.

Deciding who are the right people or what is the right situation to use as requirements' source, who should be interviewed, and what should be observed is a complex matter. In some cases it is possible that the customer has a wrong conception about what is required, even if he knows precisely what he wants to achieve.

Also, some objectives may only be achieved with the concurrence of a group of people working together. People in this group may have personal objectives differing from the system's objectives or, at least, they may have different views on how the system's objectives should be achieved. It may be unwise to leave some of them out of the elicitation process.

5.2. Using the right techniques

Deciding what techniques to use when eliciting requirements is another problem and another source of extremeness. Face-to-face meetings between customer and programmers are the only method formally conceived in XP to elicit requirements. This is extreme because it limits the choice of elicitation techniques. This goes against the principle of appropriateness.

Different situations, and even different customers, require different elicitation techniques. In some situations one may use unstructured interviews, in others it may be better to apply structured interviews, to use scenarios, or to observe people at work [11]. It all depend on the sources of requirements and what kind of requirements we are dealing with.

Elicitation methods should be based on the situation at hand. An appropriate choice is fundamental for the elicitation process and may also contribute to other requirements related tasks. For example, a structured interview may be used to gather information and, using control questions, also to validate information previously obtained.

5.3. Getting the right requirements

The extreme aspect related to the truthfulness principle is concerned with the verification and validation of requirements. First, there is no provision for requirements validation. Second, there is little room for a proper verification through independent testing.

An important point of testing is to have something required, some expected behaviour or result for testers to test against it. In XP there is no detailed descriptions on what to base the elaboration of tests. The only descriptions of what should be built are the user stories, which seem an insufficient base for both unit and acceptance tests.

It is accepted that the inexistence of independent testing brings the risk of creating tests that conform to the programs being tested. The XP practice of creating tests before coding does not make the situation any better: the risk now is to create programs that conform to the tests, and the question becomes: who tests the tests?

6. The mandatory requirement

In economic terms there is one mandatory requirement: to survive in business, i.e., to make profits. This is usually taken for granted and it seems too basic to be mentioned. However, we should not overlook the crucial importance of such a requirement.

The quest for agility is a business requirement in a broad sense and it is unrelated to any particular application. It is in fact driven by market. Most of the time, a product has to be delivered quickly as a market strategy. In software terms the mandatory requirement may be stated as

> *The software should be built to make possible its delivery first than any other competing software.*

There are other possible formulations. The main point of this definition is the conquering of a market position. Of course this is not an absolute requirement in the sense that it is derived from the need to conquer a market position and it depends on other requirements to satisfy this objective.

For example, to come first with a faulty product may hinder rather than help the conquering efforts.

The point of being the first to secure a market position always involves a trade-off between agility and correctness. A decision on to what extent one may go in producing a faulty product to be the first in the market has to be made; and that decision does not rely on technical arguments only. There are other resources like advertising, legal and political actions, FUD (fear, uncertainty, and doubt) tactics [12], pain infliction [16], and other market practices that come into play. Considerations like correctness, quality, and appropriateness may be relativised if market conditions allow a product or service to live with less than optimal features, and this is usually the case. This is what is called relative quality in [1].

Of course it is inconvenient to tell people that some values are not being considered as they should be. So a justification discourse is developed to make things palatable. (see [6] for a description of the use of propaganda for shaping acquiescence in corporate hegemony and [29] for a general discussion on business ethics and public interest). Usually the discourse takes the form of convincing people of the inevitability and necessity of new ideas and approaches. This is one aspect of what is called "engineering of consent" in [5].

If the new ideas are associated with the sign of modernity or post-modernity, even better. In software terms we may start talking about continuous improvement instead of deliver-and-fix practices. Other claims and incentives like courageous way of doing things and customers in control, having what they want when they want, are other facets of the same justification discourse.

6.1. What should be researched?

The justification discourse is by no means completely destituted of sense. There should be some anchors to reality, no matter how diffuse, weak, or inappropriate those anchors may be. In XP we have many ideas and practices that are indeed perceived and accepted by many as contributing to improve the quality of work and product. Even the leitmotiv of the agile methods — to develop software to be the first in a market niche — is a legitimate desire and therefore a requirement that should be seriously considered.

The ideas and practices related to the agile methods should be investigated in two ways. First, whether they can indeed lead to short development times; what is it that works and what does not? Second, whether the assumptions behind each practice are preserved in agile environments; to what extent the XP ideas and practices weaken the principles of purposefulness, appropriateness, and truthfulness.

The first line of investigation requires case studies and experimentation. Some XP practices have already been tried in other contexts: a daily build approach to software development, organised around teams responsible for the complete development of customer features, is described in [14] and it is said to help XP to scale up.

The second line of investigation requires a careful scrutiny and would benefit from research already done concerning the roots of XP practices.

Continuous improvement. The idea of continuous improvement, taken from the quality movement [8, 13], deals with change but from a stable perspective. A process modification should last and be present in future instances of the (improved) process. In a situation of instability it becomes difficult to insert any modification and call it an improvement because we do not know even if the situation is going to appear again. How can we be sure that we are indeed improving the product and not just working with a bad process that requires constant product fixes?

User stories. A concise description of the system and its desired functionalities early on the development is necessary for planning, making estimatives, and defining its scope. Can requirements and design be based on these concise descriptions without further elicitation activities? Are face-to-face meetings between programmers and customers enough to elicit all the necessary details? Can the informally elicited details be effectively shared with others in the development team?

Pair programming. Pair programming and promotion of shared code are related to the idea of egoless programming [33]. There are published lessons teaching pair programming [35] and reports containing evidences of its benefits [34]. This is a very appealing idea but may not be applicable in all situations [20]. What is necessary to work with such a social and psychological concept? To share something is a very deep social attitude. Is it possible such an attitude in a selfish and competitive environment? Is it possible to have an altruistic software development process in a market-oriented company?

Test-first approach. The test-first approach has some positive points and may be wise in a situation of partial knowledge. A positive result is that it obliges the programmers to think about what they are going to code, not only in terms of what should be coded but also in terms of what should not, i.e., the exceptions. This orientation has long been advised as a good practice, but is it sufficient to replace independent testing? An experiment with the test-first approach found that it supports better programming understanding, although it neither accelerates the implementation nor results in more reliable code [18].

On-site customers. The constant presence of the customer resembles the concepts of superuser and of participatory design [17], but the motivation for user involvement is quite different. Is that presence sufficient to assure the quality of elicitation? Having customers as the only source of requirements, how can we distinguish what they want from what they need? Can elicitation and programming be done by the same person? Is not a different frame of mind necessary for each of these activities?

Apart from investigating the concepts related to agile methods, we may also consider some research topics that have a positive impact on development time.

6.1.1 Domain and organisational models

Organisational models have been used as software analysis devices, to build models of the system being developed [10]. Their use in system development should help the acquisition and analysis of the domain, environmental, social, and other organisational related requirements. This requires an effort that may not be appropriate for situations where time-to-market is the prominent requirement.

I believe that the proper use of organisational models is to model organisations and not particular applications. Therefore, we do not need to build a model for every application, we need just to consider the application's impact on an existing model. Assuming there is an organisational model available to the development team, how the short descriptions of functionality can be related to this pre-existing description of the organisation? Generally, how are things expressed in an organisational model related to what is said in the user stories? Is it possible to use an organisational model to guide the conversation between customers and programmers and so improve the elicitation process?

Are organisational models compatible with rapidly changing environments? In principle, they are for organisations that want to organise themselves and, therefore, they should be stable and last for long periods with small modifications. This is not the case, for example, of many dot-com and Internet companies, representatives of these hectic times, and natural candidates for using agile methods.

6.1.2 Reuse and patterns

The design and architectural patterns and the reutilisation of artefacts may also be useful in the context of agile development. Two important problems in reuse are the proper description of the components and the retrieval of an existing reusable component. Some form of structure should be enforced to help people find what they want, otherwise the reuse effort is useless. One line of research could investigate the relationship between descriptions of reusable components and user stories. Are user stories sufficient for the identification of components? How may they be used to guide the XP team in finding what is available?

If an organisation wants to reuse context situated components it should create and organise its own reusable components. This requires a development process where tasks related to the identification, analysis, modification, and registration of reusable artefacts could take place. Is it compatible with the way XP projects are carried out?

6.1.3 Scenarios

Scenarios are used in several ways during software development. They have potential to save development time because of their power for the identification of problems and requirements, facilitation of learning and agreement, and validation of knowledge. They can also be used to derive test cases and to help management [32].

Scenarios have a natural relation with XP. It is usual to associate user stories with use cases and scenarios. Indeed, it has been said that a user story can be thought of as "the amount of a use case that will fit on an index card" [2].

Can the informal face-to-face meetings be guided by scenario construction? Can scenarios be derived from the building of unit tests and be used as a validation artefact? What level of automation and what notational devices do work with the "no waste of time on documentation" philosophy of XP? Sutcliffe et al. [28] propose an approach for a scenario-based requirements engineering that facilitates the reuse of knowledge, increases the validation power, and can be integrated with enterprise concepts. Leite [9] presents a concrete proposal for using scenarios in modified XP projects.

7. Conclusion

I do not think requirements need to go extreme and I see as inappropriate the election of time as the prominent aspect of requirements research. On the other hand, we still have a long way to go and clearly many advances from RE research will have a positive impact on the development of timely products.

I support the idea that requirements research should strive to identify, study, and use as a framework, principles to guide the investigative labour. The requirements principles may be different from the ones I have described, the point being that we should have principles against which to discuss advances and contraventions. The adherence to the requirements principles is what I call requirements honesty. Of course, I do not imply that people supporting XP are dishonest. I also do not propose that RE research should try to preserve these principles at all costs — they can even be subverted. What I claim is that they should be understood

and taken into consideration so that possible deviations can be made clear.

In this article I discuss how requirements research should be conducted in the light of the agile methods. Many of the issues discussed here are general, but my specific points and conclusions cannot be extrapolated to all agile methods. In particular, I focus my attention on XP, describing it as biased towards time. I present time-to-market as a mandatory requirement and argue that this market orientation may weaken requirements principles.

The XP process introduces into the discussion of software development a strong appeal to human considerations. This comes not only with the reliance on the programmer's job and customer involvement, but also with the invitation to be fast, quick, agile, and other characteristics perceived as good ones. For example, it is said that XP improves a software project in four essential ways: communication, simplicity, feedback, and courage. The first three aspects are also present in other models and the courage aspect, if nothing else, is a clear invitation for one to be on the right side — the side of courageous people. The point developed in this article is that this is part of a justification discourse elaborated around the need to be the first in a market niche. The agile trend reflects more and more a market based requirement.

Nevertheless, the desire to be the first in the market is legitimate and most of the XP concepts and practices have been proposed before in other contexts. Therefore, they should be thoroughly investigated to see how they work together and what are their implicit assumptions. We should only be careful not to embark on the market's justification discourse and start proposing things like extreme requirements. In terms of requirements the point to make is not related to extremeness, or to be the first in a market niche — this is just another requirement. The main point is whether requirements engineering is possible in agile environments and what are the limits; what are the gains and losses in trying to be faster and faster?

Natural lines of research would inquire to what extent the XP practices hinder the requirements principles. Some questions were proposed to illustrate possible points of investigation concerning the concepts of continuous improvement, pairwise programming, user stories, test-first approach, and customer involvement. I also suggest research on the relationships of some XP artefacts like user stories to some conventional practices and artefacts like domain and organisational models, reuse, scenarios, and architectural patterns.

A last point, unexplored in this article but serving well as a way of conclusion, is related to the status of requirements education. There are few classes teaching how to elaborate and apply surveys and questionnaires, how to observe and interview people, and how to use requirements related techniques in practice. There are some books and articles talking about the elicitation techniques, their structure, properties and appropriateness. However, expecting someone to learn how to elicit just by reading about elicitation techniques is similar to expecting people to learn how to program just by reading a programming language book. One should practice, practice, practice; and practice again. We should realize that there are some skills that need to be mastered before we can make sense of more advanced concepts. This lack of experience on elicitation techniques contributes to the acceptance of lesser ways of doing requirements engineering.

Acknowledgements

I am grateful to the valuable and constructive comments and suggestions that I received from the anonymous reviewers. They pointed out errors and made very interesting points that helped me to improve the contents of the paper.

References

[1] J. P.-H. Balasubramaniam, R. Baskerville, and L. Levine. How internet software companies negotiate quality. *IEEE Computer*, 43(5), May 2001.

[2] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.

[3] G. Booch. Developing the future. *Communications of the ACM*, 44(3):119–121, March 2001.

[4] E. Carmel, R. D. Whitaker, and J. F. George. PD and joint application design: A transatlantic comparison. *Communications of the ACM*, 36(4):40–48, June 1993.

[5] N. Chomsky. Market democracy in a neoliberal order: Doctrines and reality. Davie Lecture, University of Cape Town, May 1997. (Available at www.bigeye.com/chomsky.htm, last visited 06/2002).

[6] D. J. Collison. Corporate propaganda: Its implication for accounting and accountability. Discussion papers series, University of Dundee, Accountancy and Business Finance Department, 2000. (Available at www.dundee.ac.uk/accountancy/papers/010.doc, last visited 06/2002).

[7] A. M. Davis. Operational prototyping: A new development approach. *IEEE Software*, pages 70–78, September 1992.

[8] W. E. Deming. *Quality, Productivity, and Competitive Position*. Massachusetts Institute of Technology, 1982.

[9] J. C. S. do Prado Leite. Extreme requirements. In *Jornadas de Ingeniería de Requisitos Aplicada*, Sevilla, Spain, June 11-12 2001.

[10] E. Dubois, E. Yu, and M. Petit. From early to late formal requirements: A process-control case study. In *9th International Workshop on Software Specification and Design*, pages 34–42, Ise-Shima (Isobe), Japan, April 16-18 1998. IEEE Computer Society Press.

[11] J. A. Goguen and C. Linde. Techniques for requirements elicitation. In *First IEEE International Symposium on Requirements Engineering, RE'93*, pages 152–164, San Diego, California, 1993. IEEE Computer Society Press.

[12] E. L. Green. FUD 101, 2002. (Available at http://badtux.org/eric/editorial/fud101.html, last visited 06/2002).

[13] J. M. Juran. *Juran on Quality by Design: The New Steps for Planning Quality into Goods and Services*. Free Press, revised edition, 1992.

[14] E.-A. Karlsson, L.-G. Andersson, and P. Leion. Daily build and feature development in large distributed projects. In *22nd International Conference on Software Engineering, ICSE'00*, Limerick, Ireland, June 2000.

[15] M. McCormick. Programming extremism. *Communications of the ACM*, 44(6):109–111, June 2001.

[16] P. Meyer. Killer applications. *Business & Economic Review*, 44(2), January-March 1998. (Available at research.moore.sc.edu/research/bereview/be44_2/killapp.html, last visited 06/2002).

[17] M. J. Muller, D. M. Wildman, and E. A. White. Taxonomy of PD practices: A brief practitioner's guide. *Communications of the ACM*, 36(4):26–28, June 1993.

[18] M. M. Müller and O. Hagner. Experiment about test-first programming. In *Conference on Empirical Assessment in Software Engineering, EASE'02*, April 2002.

[19] M. M. Müller and F. Padberg. Extreme programming from an engineering economics viewpoint. In *Fourth International Workshop Economics-Driven Software Engineering Research, EDSER*, Orlando, Florida, May 2002.

[20] M. M. Müller and W. F. Tichy. Case study: Extreme programming in a university environment. In *23rd International Conference on Software Engineering, ICSE'01*, pages 537–544, Toronto, May 2001.

[21] P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by NATO Scientific Committee*, Garmish, Germany, 7-11 October 1968. Brussels, Scientific Affairs Division, NATO (1969). (Available at www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/index.html, last visited 06/2002).

[22] B. Nuseibeh and S. Easterbrook. Requirements engineering: A roadmap. In A. C. W. Finkelstein, editor, *The Future of Software Engineering*, Limerick, Ireland, June 2000. IEEE Computer Society Press. (Companion volume to the proceedings of the 22nd International Conference on Software Engineering, ICSE'00).

[23] M. B. Özcan and J. Siddiqi. Interchanging specifications and implementations in evolutionary prototyping. *Software Practice and Experience*, 26(9):999–1023, September 1996.

[24] M. C. Paulk. Extreme programming from a CMM perspective. In *XP Universe*, Raleigh, NC, July 2001.

[25] B. Randell and J. N. Buxton, editors. *Software Engineering Techniques: Report of a conference sponsored by the NATO Scientific Committee*, Rome, Italy, 27-31 October 1969. Brussels, Scientific Affairs Division, NATO (1970). (Available at www.cs.ncl.ac.uk/people/brian.randell/home.formal/NATO/index.html, last visited 06/2002).

[26] S. Robertson and J. Robertson. Extending requirements - a practical workshop. Seminar offered by The Atlantic System Guild, Inc., 2002. (Description available at www.systemsguild.com/GuildeSite/Robs/erw.html, last visited 06/2002).

[27] G.-C. Roman. A taxonomy of current issues in requirements engineering. *IEEE Computer*, 18(4):14–22, April 1985.

[28] A. G. Sutcliffe, N. A. M. Maiden, S. Minocha, and D. Manuel. Supporting scenario-based requirements engineering. *IEEE Transactions on Software Engineering*, 24(12):1072–1088, December 1998.

[29] P. Ulrich. Facing public interest. the ethical challenge to business policy and corporate communication. In *Topics of the 7th European Business Ethics Conference*, St. Gallen, September 14-16 1994. Institute for Business Ethics. Available from www.iwe.unisg.ch/org/iwe/web.nsf (last visited 06/2002).

[30] A. van Deursen. Customer involvement in extreme programming. *ACM SIGSOFT Software Engineering Notes*, 26(6):70–73, November 2001.

[31] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *22nd International Conference on Software Engineering, ICSE'00*, Limerick, Ireland, June 2000. IEEE Computer Society Press.

[32] K. Weidenhaupt, K. Pohl, M. Jarke, and P. Haumer. Scenarios in system development: Current practice. *IEEE Software*, pages 34–45, March/April 1998.

[33] G. M. Weinberg. *Psychology of Computer Programming*. Dorset House, silver anniversary edition, 1998. (An adaptation of chapter 4 on egoless programming is published in IEEE Software, 16(1):118-120, 1999).

[34] L. Williams and R. L. Upchurch. In support of student pair-programming. In *SIGCSE Conference on Computer Science Education*, Charlotte, NC, February 2001.

[35] L. A. Williams and R. R. Kessler. All I really need to know about pair programming I learned in kindergarten. *Communications of the ACM*, 43(5):109–114, May 2000.

[36] XP.org. Extreme programming: A gentle introduction, 2002. (Available at http://www.extremeprogramming.org, last visited 06/2002).