Most Stable Elements First Approach

Ivan Maffezzini
Université du Québec à Montréal
CP 8888 Succ. Centre Ville
Montréal H3C 3P8 Canada
Maffezzini.Ivan@UQAM.ca

ABSTRACT

After the introduction of the hypothesis upon which our research is based, we describe the conceptual framework of the methodology we propose. The framework presents an overview of computer-based automation where the most important artefact is a non-formal document describing concepts and requirements. Follows a description of an approach based upon a classification of requirements, concepts, objectives and constraints. Based upon a stability criterion three categories are established: *Hard Core, Protective Belt, Fluctuating Elements.* The levels of stability impose an order on the implementation. The artefacts for the requirements of the three categories of stability are outlined. The approach named "Most Stable Element First" furthers the introduction of the details from the beginning. The conclusion presents a few positive elements that our approach brings to the automation domain.

Keywords: Information and Technology - Computing - Information Systems – Product Quality (or Method and Methodologies)

INTRODUCTION

The problem.

Engineering does not exist without methods and software engineering (SE) is far from breaking the rule. One can probably also state that it is precisely because methodology is so important in SE that different methodologies have appeared and disappeared at speeds inconceivable for other engineering sciences. Changes of paradigms in SE occur so often that it sometimes seems as though we are dealing with a branch of the humanities rather that with a technical domain. Although in its beginnings, SE forced activities, methods and deliverables to conform to an overrigid structuration in order to avoid the laisser-faire of ad hoc methods made up for every new kind of problems, nowadays approaches such as extreme programming praise a laxness which threatens to erase many of the gains acquired by SE. It is essential to find some kind of compromise between these two extremes even if it's very likely that the opposition between the proponents of agile development — claimed to be adaptive and people oriented — and the proponents of "traditional" software engineering — processes and documentation driven — will continue for years. Both sides champion two truths that, unfortunately, are often transformed into non-truths by opponents who make it simplistic or by dogmatic militants who ignore the contribution from the other camp.

For sure, both sides are interested in methodologies.

Unified Process (UP) [1] is a good example of a compromise even though we believe that the downside is that UP lacks details in the definition of the means to establish iterations and increments.

In this communication we will put forward an approach which is general enough to be applicable to processes such as UP and to more classic ones, but at the same time which is precise enough to add value to already existing methodologies.

« Axioms »

Our method is based on the following elements which are not demonstrable but emerge from a rather well shared commonsense:

1. SE, as defined by works quoted by SWEBOK [2], is a branch of automation.
2. *Every domain contains at least one part that can be automated*. This principle states that however the world is divided, whatever the size of the domain, it will always contain elements which can be put in connection with elements of a machine and in such a way as the "whole" can function with a certain degree of autonomy. This principle does not imply that everything in a domain can be automated.
3. Maintenability is the foremost intrinsic quality in every engineering approach of software development. Where maintenability is not important, we are dealing with research or prototypation.
4. The efficiency of a methodology is "inversely proportional" to its scope; and, therefore: The "ideal" methodology has an applicability restricted to a few domains.
5. A language valid for all the domains and all the processes (as UML pretends to be) must pay for its generality with a loss of efficiency.

CONCEPTUAL FRAMEWORK

In this section we will describe our reference frame, introduced in [3]

Definitions

Here are a few definitions in order to set our conceptual framework:

World. Physical reality enclosed by natural language: reality as it "appears" to humans. It is not merely physical nature.

Domain. A domain is a part of the world, a linguistic point of view on reality. The link between the world and a domain is homeomeric with an invariance which is never total: meaning that a domain has the same characteristics than a world and can never be completely separated from the whole. Because a domain implies language as element of observation, it always is a domain of knowledge. For example, an atom is not a domain but atomic physics are. The language of the domain is a specialization of natural language and its scope depends on the domain´s characteristics. The fact of being a specialization does not imply that natural language in its entirety is not present when humans share ideas about the domain. Domains condition each other thus creating huge grey areas which might be called no domain lands. The boundaries drawn to facilitate comprehension are often very arbitrary and they artificially clarify the areas where domains are overlaid. Creating clear-cut boundaries, and thus impoverishing domains is required by science/technique in order to make language operative.
The number of domains, even in a small part of the world, is unlimited because domains are continually being created, modified or destroyed. The rhythm of creation, modification and destruction depends on the domain´s characteristics and on the evolution of technique and of knowledge. There are also cultural and historical phenomena that influence the organisation of the world into different domains. In physics, for example, a branch (or a domain?) such as quantum mechanics which did not exist a century ago, has a huge importance nowadays, and not only in physics.

Machine. A human artefact composed of parts which are required for the realisation of functions aimed at by its construction. A part of a machine can be a machine. A program is a machine.

Automation. The use of machines aimed at transforming data received from the exterior and executing autonomous actions having an impact on the world. This definition restores a meaning that the use of the term in process control has somehow erased. Even though we consider mostly automation realised by computers, a larger definition allows to characterize the software more easily. By way of counterexample we do not consider as automating machines (automatons):

- the water which moves the wheel of a water-mill or the worker on an assembly line who performs mechanical actions, because they are not artefacts;
- a book, for although it is an artefact aiming at transforming the world, it lacks autonomy;
- a car qua car because even if it is a machine, it does not transform data but energy;

By way of example we consider automatons:

- a cruise control for a bus;
- an application that calculates taxes.

Automation cycle

The following figure shows the world with, inside of it, an outline of the process of automation when the automation is performed through computers.
The full arrows represent transfers between machines or between machines and a domain. The dotted arrows represent transfers that require the intervention of humans considered most of all, but not only, as beings endowed with language.

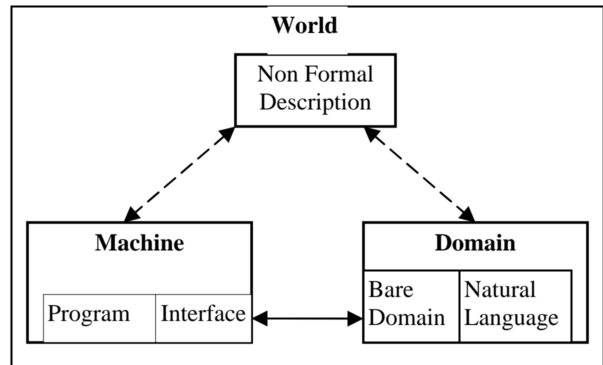

Fig. 1 : Automation Cycle

We say bare domain even when the machine interacts with humans: in this case, unlike in the passage from the domain to the description, natural language is only a means to allow the machine to interact with a semantically impoverished domain — an impoverishment operated during the transformation of the domain into a sequence of bits in order to control the bare domain. Hence the human being, based on natural language and eventually helped by special notations (UML, for example) creates a Non Formal Description (NFD) which contains two types of information:

- *The domain´s characteristics*. They are constituted of concepts and associations between concepts which the machine will not be able to modify and to which it has to adapt. For example, F=ma, in classical mechanics is a relation between three concepts which can not be modified by a machine
- *The controls over the domain*. That which is imposed on the domain; requirements and constraints fixed on the domain. That for which the machine was created. In the previous example a requirement could be to prevent F from being superior to a fixed limit. The possibility of this does not depend on the law of Newton but on other characteristics of the domain.

It is essential to keep in mind the more precise the NFD hence the closer to the machine, the further from the domain because it operates a semantic impoverishment of the language of the domain.

What we just said is not true when the domain is constituted only of operational machines, because in this case the NFD is only a set of references to the complete descriptions of the machines or to formal descriptions.

Conclusion.

This framework allows us to point out elements which condition the Software Life Cycle process:

1. Natural language is an impassable presence in the automation. This implies that we can not get rid of its ambiguities. These ambiguities will be transformed into errors in the exchanges between the domain and the machine. The advantage of formal specifications introduced after the NFD is that they allow to find errors earlier, not to avoid them!
2. The types of knowledge, the skills, the necessary approaches in the activities which "transform" the domain into NFD are very different from, and sometimes even in contradiction with those that are required to go from the NFD to the machine.
3. The comprehension of the domain is not important in the part of the SE that goes from the NFD to the computer, which means that the persons who operate this transformation (software designers and programmers) only need to understand the NFD.

The NFD being the central element in the construction of a machine, what must it contain? There is no precise and unique answer to this question: the content depends on the domain, on the standards, on the methods of development, etc. One thing is sure, it is that the requirements qua phenomena of the domain [4] can not be missed, which implies the necessity of a conceptualisation of the domain — with the help of UML classes for example.

In order that point 3. be true, the NFD has to be close enough to the machine without being formal. How can we measure this "neighbourly proximity"? Indirectly by the precision and the lack of ambiguity, thus implying even more detailed elements. But how can we introduce these details so that the NFD might reflect the domain without having the details "hiding the forest"?

The approach that we put forward in the next section — approach that "converts" NFD into three types of documents — is a possible answer to this question.

MOST STABLE ELEMENT FIRST

Introduction

As the tenants of UP, we believe that the IEEE approach which is clearly stated in SWEBOK [2], i.e., "*the specification of all general requirements right from the start*" is not an acceptable one. Indeed, it is impossible to determine when "all" requirements have been defined, on the one hand because of the subjective elements of choice (due to the language used in the domain) and on the other hand because some requirements may change during the lifetime of the machine (due to perfective maintenance mostly)

To this impossibility to obtain "all" we add a methodological consideration which, in our opinion, is commonplace when the system is not too simple:

The progression from the more general (needs and requirements) toward the more specific (the executable program) cannot be achieved without classifying the components of the problem at stake.

We could certainly say that a system is complex when it is not possible to realize it without some form of classification of the components.

Our approach can be seen as the simple common denominator for methodologies applicable to a large number of domains that gives a crucial position to the "stability during the development" as presented in [5]. For this reason, we call it MSEF (Most Stable Elements First). The approach is freely inspired by the epistemology of Imre Lakatos [6]

Définitions

In MSEF, a project's first step must be to characterize the domain's functional and quality requirements with respect to stability with the goal of classifying the requirements into one of the three following categories:

1. *Hard Core* (HC): A set of requirements, constraints, concepts and goals judged stable during all the system life cycle by the stakeholders. The main reasons to introduce a Hard Core are :
   - To provide a solid point of view during the entire life cycle.
   - To make "digging" of the problem easier.
   - To introduce at a very early stage "details" important to understand the problems.
   - To make the addition of new people to the project team easier.
2. *Protective Belt* (PB): A set of requirements, constraints, concepts and goals with a good stability at the beginning of the process but with a high probability that they will change before the end of the life cycle. The main reason to introduce a protective Belt is to protect the HC from the continuous (or presumably continuous) changes of the elements in the outer circle. The Protective Belt gives some initial assurance that the requirement engineers and the developers share a common solid environment even if the HC is empty (or almost empty).
3. *Fluctuating Elements* (FE). A set of requirements, concepts and goals coming up, changing and dying out during the whole system life cycle. The main reason is to allow managing the instability in a systematic manner, i.e. not only by *ad hoc* interventions.

The figure below shows a graphical representation of the categorization

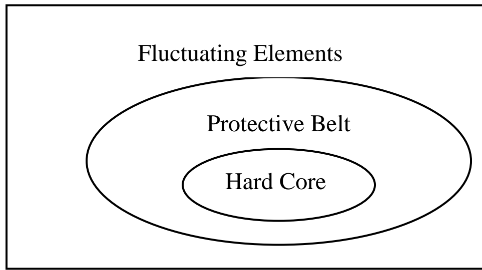

Fig. 2 : Stability Categories

In order to make MSEF a practicable approach, it is important to define “stability” in an objective (ideally based upon some stability metric) and operational manner (categorization should not be too difficult). At this stage of the research we can be satisfied with the definition of [5]:

Stability can be expressed in terms of the number of expected changes to any requirement based on experience or knowledge of forthcoming events that affect the organization, functions, and people supported by the software system.

Even if this definition refers only to requirements, it may be very well applied to the domain concepts *qua* elements allowing a description/specification of the requirements

In accordance with this definition the stability depends on the points of view of stakeholders. For a practical application of the approach it will be necessary to establish some operational rules. Here is an example: *an element must be put into HC if and only if all the stakeholders agree on is stability*.

The next two sections (adaptation of [7]) describe the activities and the artefacts concerning requirement engineering in accordance with the MSEF approach

Project workflow

The project workflow is based on the previous classification. The choice of stability as the first discriminator instead of necessity — the other classification element of [5] — implies that some « essential » elements can be delayed if they are not stable enough. The client may not necessarily be willing to accept these delays without discussion.

Here is the overview of a possible workflow in a UP like frame:

1. Users, clients and domain experts assign a stability level to the domain requirements, concepts, objectives, etc.
2. The most stable elements are kept separated and inserted into HC, i.e. a subset independent of priority, necessity, and functionality is created. Generally all the constraints will be in HC.
3. Based on HC a sub-project is created. Requirement engineering, design and code construction are realized with some overlapping to create the synergy necessary to truly understand the problem at stake. If the sub-project born from HC is too big because of too many stable requirements, other criteria must be employed to reduce the dimension of the increments.
4. The remaining elements are split up in two categories and allocated to PB or FE.
5. If HC sub-project need some new not so stable functions to implement a stable useful functionality a priority is assigned to the new functions that belong to PB or, less frequently, to FE.
6. The HC sub-project artefacts support the analysis of PB elements that become the starting points for a new series of increments to add to the product created from HC elements.
7. All new user requirements are inserted in PB or FE

Artefacts

During the analysis three types of documents can be written. At least one must be non empty:

- Hard Core Specification (HCS).
- Protective Belt Specification (PBS).
- Fluctuating Elements Description (FED).

The firsts two documents are machine independents, the third one can be machine dependent.

The next table outlines the differences between the three types of documents according to 1) the beginning of the activity; 2) the changes after the first release; 3) the main stakeholders and 4) the type of validation.

	Start	**Changes after release**	**Stake-holders**	**Validation**
HCS	Concept exploration	Very infrequent	Domain experts	Review Domain history
PBS	HCS well advanced	Infrequent	Domain experts, users, managers	Review Domain history
FED	PBS well advanced	Very frequent	Users	Running program

Table 1: Artefacts characterization

In accordance with SWEBOK the first document produced in the analysis activities (Concept of Operation) is not a specification whereas the two documents that follow it are specifications. In our approach it is the opposite: the first two are specifications and the last one is not a specification. Our approach implies that is not the documents production sequence that determines the level of formalism and the quantity of details but the level of stability. The last one (FED) is the less formal and in some cases it could even be reduced to some messy annotations.

HCS is a specification without any software element. It is totally independent from the structure of the machine that will execute the required functions. Even if this specification has the same position in the development cycle as the *System Definition Document* of SWEBOK, it has a different role and a very different content. The role of HCS is not to “*define high level system requirements*” but to “*specify the stable elements with as much detail as it is possible*”. In other words: the concepts and requirements of HCS do not need to be more detailed in

other documents because all the details are described in the HCS at the beginning of the product life cycle. Concerning the contents: the *System Definition Document* defines all high level requirements *whereas* HCS defines only *some* elements (particularly constraints and concepts) but with all the possible details. In an iterative and incremental development architecture based, HCS provides a minimum of stability in order that the machine subsequently will not be destroyed by a functional or quality "tsunami". In a domain where only the other machines are stable, HCS will be filled with references to the specifications of the other machines.

PBS is a document more agile than HCS but with the same structure. It rarely contains constraints and objectives. It is above all concerned by requirements, i.e. by the limits that the machine must force on the domain. When one must create a machine for a domain in which all the interfaces are with other already specified machines, the PBS in not necessary because all requirement are stable and specified. When the human machine interface is for only one role, the PBS can be empty because a prototype or/and a FED is a better choice.

FED is the document that describes the more volatile elements of the system. It is a document less structured than HCS and than PBS. Sometimes the information of the FED can be written in the program listings as comments or in an executable prototype

Exemples

1. *High level Data Link Control* (*HDLC)* for UNIX operating system. All the functional requirements for this software are part of the HCS. They are written for several decades in an ISO standard describing the protocol. On the other hand, some non-functional requirements should be put in the PBS. We can imagine some non-functional requirements concerning the person machine interface for testing the protocol to be part of FED.
2. *Communication networks and Systems in substations* where the IEC 61850 interoperability standard is enforced. The interoperability quality requirement is changed into a constraint by the IEC standard. The requirements concerning the types of command model (direct with normal security, direct with enhanced security, etc), for example, will be in the HC but the choice of the type will be in PBS or FED. The choice between PBS and FED depends on others quality requirement as changeability.
3. *WEB site for a new publisher.* Only some objectives and the main concepts (as book, writer…) will be in the HCS. The usability requirements will be into the PBS and most functional requirements will be into the FED.
4. *Cruise control for cars*. The safety requirements are elements that will certainly be into HCS.
5. *Application for the 31 bottles of Bordeaux cellar of a friend*. Some concepts will go into the HCS and all the others requirements into the FED. The HCS reduced to two or three pages. No others documents are needed. Extreme programming is certainly the ideal development methodology in this case.

The following table comments the eight combinations of presence (x) and absence (0) of a kind of document

HCS	PBS	FED	Comment
0	0	0	There is not a true problem to solve.
X	0	0	All requirements are stable and no perfective maintenance foreseen
0	X	0	During the product life cycle there will be significant changes
X	X	0	There will be no unforeseen changes.
0	0	X	À research or toy problem
X	0	X	A lot of changes with a solid kernel.
0	X	X	Nothing is very stable but… a very realistic problem
X	X	X	The majority of important projects will be here.

Table 2: Artefacts characterization

CONCLUSION

Even if the approach we are putting forward is far from totally elaborated we hope it brings a few positive elements to the automation and software engineering theory.

Subset versus "all"

The HC elements provide a strong base from which the stakeholders can examine in more details the concept and the requirements. The more stable elements play an important part as catalysts for the deepening of the field. Since stability is independent from necessity and from functionalities, there is a good probability that when the more stable elements are established, we end up with diversified functional elements and so — unlike a use case based approach — the risk of realizing some functions at the expense of some other is reduced. For instance, let's consider the next figure.

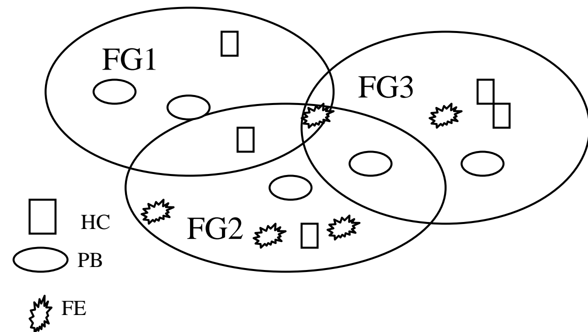

Fig. 3 : Functional groups and stability

This figure includes three functional groups (FG1, FG2 and FG3) and each of them includes elements from the three previous defined categories. The project begins with the construction of the HCS, starting from a private element pertaining to FG1, from a private element pertaining to FG2, from 2 private elements pertaining to FG3 and from an element shared by FG1 and FG2 (the rectangles of the figure). We get a transversal section of

three functional groups allowing a progression on all the aspects of the project without broaching « all » the requirements.
Among the implications of our approach, let's mention the following: resources and time being constant, it is better if we broach in details a functions' subset rather than broaching "all" them in a more general way.

Domains's concepts versus requirements

Pretending that the concepts of a domain are more stable than the functions is a platitude in the field of automation. One consequence of this consideration is that HCS will contain more conceptual diagrams than requirements or use cases. Applaying MSEB the analysis of the domain will therefore precede the system design, in almost all the cases. This is very important with the aim of countering the present primacy of uses cases (an efficacious system design tool) in processes like UP [4]

Details and refinements

Going in depth means adding details. But details are, on one hand, a way of understanding and, on the other hand, they are factors preventing us to form a clear picture of the overall structure. But, since the more stable elements are the ones stakeholders know best, we can go further in the analysis of the domain studying the last details (eventually by programming them) without running the risk that the details could cloud the comprehension of the domain.

In a project, no matter the method you use, it is often difficult to determine the starting moment of the artefact which gives a more detailed version of the preceding ones. The moment in which a Software Requirements Specification should be started is never easy to determine and it's even more difficult to determine when you can regard it as complete. In a project of the slightest complexity, you can even say it is never completed. With MSEF, the HCS can be completed and « locked » very early in the process, and for the whole life of the product. This locking of HCS furthers the progression of the rest of the project because the products "generated" by HCS, will probably not be changed.

Agile versus classic methodologies

The beginning of the implementation and the overlapping of analysis, design and code construction are the most important dissimilarities between agile and classic life cycle. If the classic methodologies' rigidity is often too restrictive, the agile methodologies "vagueness" hides big dangers – for instance the danger of jumping into the coding activity because you don't have the intellectual strength to carry on the analysis with the risk of getting a product quite difficult to modify. This opposition being too universal to be true, hides the fact that, sometimes overlapping is useful and sometimes useless and dangerous. We believe that MSEF makes possible to apply the classic methods to HC and, eventually, to do extreme programming with FE. From that point of view, you regard HCS and its implementation as constraints to be respected by the others elements. And, in a project, it's always easier to respect realistic constraints than unsettled requirements

The importance of writing

Automation, as an exchange between machines and bare domains (i.e., without the participation of human creativity), must be based upon a clear conceptual structure. Even if, as argued by extreme programming supporters, exchanges between programmers are really important, it seems obvious to us that, concerning complex problems (the only ones that deserve to be the subject of discussions and papers), exchanges must be established in writing not only because "scripta manent" but also and above all because what is written acquires its own autonomous life and for that raison becomes an object that ask for analysis.

ACRONYMS

FE	Fluctuating Elements
FED	Fluctuating Elements Description
HC	Hard Core
HCS	Hard Core Specification
HDLC	High level Data Link Control
MSEF	Most Stable Element First
NFD	Non Formal Description
PB	Protective Belt
PBS	Protective Belt Specification
SE	Software Engineering
UML	Unified Modeling Language
UP	Unified Process

REFERENCES

[1] I. Jacobson et al., *Unified Software Development Process*, Addison-Wesley, 1999.

[2] IEEE, SWEBOK 2004 version, http://www.swebok.org

[3] I. Maffezzini et al., "Prolègomènes à une critique du génie logiciel- Partie 1 : Contextualisation", ***Génie Logiciel,*** No. 66, September 2003. pp .2-16.

[4] M. Jackson, *Problem Frame*. Addison Wesley, 2001.

[5] IEEE, Recommended Practice for Software Requirement Specification, Std.830-1998.

[6] I. Lakatos, *Proofs and Refutations*. Cambridge: Cambridge University Press.

[7I. Maffezzini, L. Martin, "Prolègomènes à une critique du génie logiciel- Partie 1V: Exigences", ***Logiciel,*** No. 72, March 2005. pp .2-23.