Institution för datavetenskap
Department of Computer and Information Science

Final Thesis

Toward Agile Development Methods & Non-functional requirements

By

Mousa A. Al-kfairy

LIU-IDA/LITH-EX-A--09/047--SE

2009-09-29
Institution för datavetenskap
Department of Computer and Information Science

Master Thesis 30 ECTS

Toward Agile Development Methods & Non-Functional Requirements

by

Mousa A. Al-kfairy

LIU-IDA/LITH-EX-A--09/047--SE

2009-09-29

Supervisor & Examiner: Professor. Kristian Sandahl

Department of Computer and Information Science
Linköping University
SE-581 83 LINKÖPING, SWEDEN
Tel: +46 13 28 10 00
Fax: +46 13 14 22 31
To the soul of my father.

- To my Family far away there in Jordan who supports me through my studying.
- To my supervisor Kristian Sandhal.
- To my best friends Mazen and Abdullah.
Abstract

Many NFRs (Non-functional requirements) problems have been encountered when using agile development methods.

In this thesis, we tried to solve those problems by adapting agile development methods with Non-functional requirements-framework (NFR-Framework).

In this thesis, we have inspected many research papers, and we have met industrial experts for feedback regarding our theoretical results.

As a result of the inspection, we have been able to adapt agile development methods (extreme programming (XP)) with NFR-framework. We use XP since it is more practically oriented process than other agile development methods.

In the first try for this process model, we got three alternatives for applying it. The first one is based on collecting all NFRs from the beginning of the development process. The second one is based on updating the SIG (software interdependency graph) every time we have new functional requirements (FR) and the third one is based on the incremental nature of agile development methods.

Each one of these alternatives has its own advantages and disadvantages. We tried to extract those advantages and disadvantages by brainstorming and reading research papers. The most important issue in all of the three alternatives is the applicability. Finally we got industrial feedback regarding all of them.

As a result of the industrial feedback, we were able to find another alternative of how to apply the process model which is presented in 7.2.

This thesis first introduces agile development methods and XP, and then we list some NFR problems, introduce NFR-framework, introduce the process model, industrial expert feedback and results of that meeting and finally case study of how we can use this process model.

Keywords: NFR, Non-functional requirements, Agile Development methods, XP, extreme Programming, NFR-Framework, Non-functional requirements framework, NFR in Agile development methods, Non-functional requirements in extreme Programming, NFR in XP and NFR-framework in XP.
## Contents

1) Introduction ....................................................................................................................... 1
2) Methodology ..................................................................................................................... 3
3) Introduction to Agile Development Methods (XP) .......................................................... 5
   3.1) Overview of Agile Development Methods ................................................................. 5
   3.2) Overview of XP ........................................................................................................... 6
   3.3) XP planning .................................................................................................................. 8
      3.3.1) Planning Game phases .......................................................................................... 8
      3.3.2) Planning the Iteration .......................................................................................... 10
   3.4) Requirement engineering in XP ................................................................................... 11
      3.4.1) Software Requirements ....................................................................................... 11
      3.4.2) we need a process ................................................................................................ 13
      3.4.3) properites of XP - RE process .............................................................................. 14
      3.4.4) FR as user stories .................................................................................................. 16
      3.4.5) NFR as user stories .............................................................................................. 16
   3.5) Testing in XP ............................................................................................................... 17
   3.6) Refactoring ................................................................................................................. 17
   3.7) Agile Design ................................................................................................................ 18
4.0) The Problem: NFR problems with Agile (XP) ............................................................... 20
5.0) Toward the Solutions ..................................................................................................... 23
   5.1) NFR framework: ........................................................................................................ 24
      5.1.1) Using NFR-framework ......................................................................................... 24
      5.1.2) SIG (softgoal interdependency graph) ................................................................. 27
6.0) The Solution .................................................................................................................. 31
   6.1) The Process Model ..................................................................................................... 31
   6.2) drawing SIG: ............................................................................................................... 32
   6.3) Process model(approaches) VS XP ............................................................................. 35
7.0) Industrial meeting and feedback .................................................................................... 38
   7.1) Small software Projects ............................................................................................. 40
   7.2) Large Software Projects ............................................................................................ 40
8.0) Case Study: Pizzeria calls handling system: ................................................................. 44
9.0) Conclusion ..................................................................................................................... 48
References ............................................................................................................................. 50
Software requirements’ engineering takes a lot of attention, since of it is criticality in any software project. Many solutions have been developed to make software requirements clearer and can respond to changes. Most of these approaches concentrate on functional requirements, but non-functional requirements take less attention. [1] [2]

One of these software development methodologies is agile development methods. They were first introduced by the agile manifesto which can be defined as a set of principles that has been set up by a number of IT experts to solve current IT development problems. The agile manifesto consists of a set of software development methodologies that followed the four agile principles:

1- Individual and interactions in any software project are more important than processes and tools.

2- Build working software is more important than writing a complete and comprehensive documentation.

3- Customer shall be collaborated and involved through the development process and that is more important than contract negotiation.

4- Team shall be able to respond to changes even if they do not follow the plan.

Those principles are supposed to be an extraction of many years of experience and said to be best practice development principles. [1]

Agile development methods are supposed to solve the problem of vague requirements by involving the customer through the whole development process. It also solves the problem of late changing in requirements by small and frequent iterations. [4]

Agile development methods - especially SCRUM and XP - are considered as one of the most used methodologies in the IT industry. As a result of this popularity, recently they receive a lot of attentions from researchers and organizations.

Although agile development is reported as a very good technique for small to medium software projects, some problems have been reported in satisfying NFRs. [5][6]

On the other hand, NFRs have a lot of problems by its nature (See 4.0). Also, some solutions have been developed for solving NFRs problems. One of these solutions is NFR-framework (see 5.0). [7]

Through this thesis we tried to solve the problem of NFRs in agile (extreme programming as an example of agile development methods) development methods, by developing a process model that is derived from NFR-framework.

We have the following requirements for any process model that is supposed to solve NFRs with agile development methods:
1- Simple model: the user shall be able to apply this model with a little help from the process model developers and without effect on XP process of development.

2- Understandable: anyone who has a technical background shall be able to understand the flow of the process and able to explain it in simple words.

3- Can help satisfying NFR: this model shall help XP developer (we will use XPer instead or XP developers through this thesis) understand NFRs, NFRs importance and develop in a way that enables those satisfying NFRs.

4- This model shall be able to solve agile development methods problems with NFR without having any risk in exceeding the time or the budget of any software project.

In this thesis, we first start by introducing agile development methods, go deeply through XP (extreme Programming), its rules and the problem each rule solves. Next, how XP treats requirements and specially NFRs? Then, we introduce NFRs and it is main problems. Next, we introduce NFR-framework briefly and concentrate on software interdependency graph (SIG). Then, we introduce the process model and how it can be applied through the process, the result of industrial feedback are introduced and discussed after that.

At the end of the process model development, we have one example to explain how we can use this model in practice.
2) Methodology

This thesis is mainly based on theoretical background and already completed experiments. Through this thesis we inspect a number of research reports about NFRs, NFR-framework and Extreme programming. We use some other useful books for same topics.

Our main selection criteria were based on answering the following questions for filtering papers and books:

1. Does the paper/book/online resource title relate to our research interests? If not, leave it and find another one.
2. Does the resource abstract and introduction relate to our research interests? If not, leave it and complete with another one.
3. Go through the selected resources and extract all useful information, data or knowledge, summarize them and finally put them in a proper order.

We heavily used Internet and especially Google for finding online documented resources. It was very helpful. We got some good ideas on how to proceed through this project and good comments from some of our colleagues.

We have used the university press to research through well known scientific, computer and software engineering journals.

We mainly search ACM, and IEEE.

We used the following queries in researching Google and journals; we search for ever one of the following phrases independent of others (AND):

1. Non-functional requirements in extreme programming.
2. NFR in XP.
3. Extreme Programming requirements engineering.
5. NFR problems within extreme programming.
6. Agile requirements engineering.
7. Non-functional requirements in agile development methods.
8. Agile development methods issues, non-functional requirements.
9. NFR.
11. Agile development methods.
13. Availability in extreme programming.
By using this method we have been able to go through a huge number of resources. We gained a lot of information about our research, but at the same time that took a lot of time to go through all of them.

Because of time limitation, we only read the abstract for filtering all the resources. This also had some drawbacks since we classify some un-useful resources as a useful one, but through the quick reading we were able to filter them out.

After developing the process model, which is mainly based on theoretical background and researchers’ recommendation, we have met an industrial expert for getting feedback regarding our results. As a result of this meeting, we were able to update our model in a way that- to some extent- follows researchers and industrial practices.

We have not used any formal research method neither qualitative nor quantitative method.
3) Introduction to Agile Development Methods (XP)

In this part, we start by explaining agile development methods basic principles, and then we go deeply through XP.

3.1) Overview of Agile Development Methods

Agile development methods are a set of methods which were based on agile manifesto:

1- Individual and interactions in any software project is more important than processes and tools.
2- Build working software is more important than writing a complete and comprehensive documentation.
3- Customer shall be collaborated and involved through the development process and that is more important than contract negotiation.
4- Team shall be able to respond to changes even if they do not follow the plan.

[20]

It contains a set of development methodologies e.g.:

1- SCRUM.
2- XP.
3- Lean software development.
4- Dynamic system development method. (DSDM)
5- Crystal.

[21]

Those development methods are based on the agile manifesto.

All agile development methods are based on incremental/iterative development of functionalities and customer collaboration through development process.

Scrum uses a fixed size iteration time called “sprint” which is used to be one month, but in some cases it can be less. Scrum daily meeting is very important aspect of the development process, since developers discuss work progress. When the developers start developing products, they should have a scrum planning meeting. [3]
In the above paragraph, we explained briefly how scrum- one of the agile development methods- works. Through next sections we deeply discuss XP, because it is the agile method we used for developing our process model.

3.2) Overview of XP

In the middle of 90s, a lot of discussions have been carried out to find a software development methodology to solve the main software projects problems. These problems include misunderstanding and changing in requirements which led to late delivery of software and going over the budget or sometimes build wrong software.

As a result of these discussions, a group of software experts who have either led or participated in some way in a large amount of software projects have met. In their meeting, they came with the idea of agile software development with four main principles called Manifesto of Agile Alliance:

1. People are more important than processes and tools.
2. Concentrate on building working software instead of losing time working on documentations.
3. Customer involvement in the process of development is very critical for software success.
4. The team shall be able to respond to changes even late ones which are more important than sticking to plans.

[3]

The above principles led people to think about finding comprehensive methodologies to apply them.

A year before this meeting Beck thought about XP (eXtreme Programming) and after participating in this meeting he came out with the basic principles of XP:

1. Customer involvement through the team member: customers and developers shall work together, so that developers can get a continuous feedback from the customer.
2. Making requirements documents as user stories: XPers ask the customer to write requirements as stories on cards and then ask him to priorities these stories.
3. Short software iterations: that includes iterations and at most each iteration is two weeks. Releases are almost planed to be six iterations in advance.
4. Acceptance test: The customer along with developers writes the acceptance test.
5. Pair programming: programming is always carried out by two programmers working together on the same working computer at the same room.

6. Writing test case and unite test first: the code is written to pass the test cases, so programmers first program and write the test code, then, they write the system.

7. Collective ownership: each developer can change to the code.

8. Metaphor: agreeing on common names and common ways of addressing issues.

9. Continuous integration: the system shall be integrated several times a day.

10. Code refactoring: the system and in most of the cases the requirements are likely to change, which means changing the structure of the system.

11. Simple design: the design shall be as simple as possible.

12. Planning game. See 3.3.

13. Sustainable pace: 40 working hours per week.


[1][3][8]

Beck when developing these principles he thought about what he called the four variables (cost, time, quality and scope) and he promised that by following that principles, companies will be able to solve the following main problems.

Delivering the code in time, so companies can avoid being late and postponing the delivery day. He suggested short releases as a solution for this problem and through each release developers use short iterations (one to four weeks each iteration). During the project time, the developers get continuous feedback from the customer and that helps achieving the time variable.

1. The customer chooses small releases and that helps to avoid project cancelling without making any progress in the project.

2. XPers avoid the big cost of changing in the software by running test each time they make changes in the software.

3. Defect rate: after the software has been developed, it is not put into production because of its high defect rate, XP solves this problem by testing from both sides; programmers test functionality-by-functionality and customer test feature-by-feature.

4. XP solves the problem of requirements misunderstanding by customer contribution in the project. If the developers meet any misunderstanding problem, they ask the customer directly to explain it.

5. The big problem of business changes is solved in XP by short releases, and solve the changing through the releases by continues feedback from the customers.[4][8]
3.3) XP planning

From requirements engineering point of view, planning is a critical part of the whole process, since requirements analysis is part of it and effected by each process.

Figure 1-3.3 describes the whole planning process in XP. XPers start by customer and developers sitting together. Developers ask the customer to write user stories; developers estimate each user story in term of points where each point is full time of one week work. If they found a long story, they ask the customer to divide it into as many small stories as possible.

Then, developers ask the customer to prioritize these stories (what the customer would like to be implemented within the specified release). Customer should select stories that fit the velocity (the number of points that can be done within a release) no more, no less.

In the following two parts, we will describe the planning process in detail, to understand how we plan releases and how iterations. [8]

3.3.1) Planning Game phases
Beck in his book “XP explained: embrace change” described the planning process by three main phases which are:

- **Exploration**: finding the new requirements (functionalities) to be developed.
  1. Write a story: the customer writes one of the system functionalities, the story should be written on a card using short paragraph.
  2. Estimate the story: the developers sit down and brainstorm to estimate each story time. If the developers can not estimate the story, they ask the customer to clarify or split the story.
  3. Split a story: if the developers do not understand the story, feel one part is more important than other parts or it is a very long story, they ask the customer to split the story.

- **Commitment**: find out which requirements will be implemented.
  1. Sort the stories by value: the developers ask the user to prioritize stories as (A) essential which the system cannot work without them. (B) Less - essential but provide good business values and (C) those it would be good to have.
  2. Sort by risk: the developers sort stories into three types (A) known stories and the stories they can estimate precisely. (B) Stories they can estimate in near time with a reasonable percentage of wrong estimation. (C) Stories that they feel it is impossible for them to estimate.
  3. Set velocity: the developers decide how many functionalities and points they can complete within a month calendar time and tell the customer about that.
  4. Choose scope: the customer chooses the stories (cards) that should be completed within a release. The selection process shall be based on the stories points and the project velocity.

- **Steer**: guiding the process and update the plan.
  1. Iteration: in every iteration (1-4 weeks), the customer chooses the most important stories to be implemented, as a result of the first iteration the system shall work end-to-end.
  2. Recovery: if the developers realize that they overestimate its velocity, they can ask the customer about the most important and valuable stories to be implemented within the new velocity.
  3. New story: if the customer finds out that she needs a new story, then the developers ask her to write the new story. Then, developers estimate and replace less important with equivalent estimation with the new story.
4. Re-estimate: if the developers feel that they wrong estimate the stories, they re-estimate the remaining stories and setup new velocity.[3][4][8]

3.3.2) Planning the Iteration

Next, customer and developers meet to plan the iteration, the planning process for iterations follow these steps:

- **Exploration:**
  1. Write a task: in this phase, the developers turn stories into tasks. Sometimes one task may serve many stories and sometimes it is not related to any story.
  2. Split/combine tasks: if the developers estimate one task to take more time, they split it. If the developers find out very small tasks like tasks that take one hour each, they combine them.

- **Commitment:**
  1. Accept a task: one developer takes the responsibility to complete that task.
  2. Estimate the task: the responsible developers shall estimate the time for the task with the help of other developers. If they feel that the task will take more than a couple of days they split it into many tasks with a reasonable time slots.
  3. Set the load factor: each developer set the percentage of time will spend on actual developing process.
  4. Balancing: each developer add their tasks and multiply by load factor. If some developers overcommitted they must turn some tasks to the developers who are less committed.

- **Steering phase:**
  1. Implement the task: a developer takes the chosen task card and choose a partner. Then, they write test cases for the task and develop the task to pass all the test cases. Then, release and integrate the implemented task with other tasks when the system test is ready.
  2. Record progress: every 2-3 weeks one developer ask each developers how long they spent on each task and how many days are left.
  3. Recovery: the developers who are overcommitted ask for help.
  4. Verify story: the story will be verified by functional test.
3.4) Requirement engineering in XP

One of the most important parts of developing a software is to understand the user needs (software requirements), “what the user wants”.

In order to address this problem, many suggested solutions have been developed which are claimed to be the best practice methods of capturing user requirements.

In this part we will have a look into XP requirements engineering- almost described in planning game part- starting from:

- What is a requirement?
- Which classifications do we have for requirements?
- What process do we want?

Then, we study the advantages and disadvantages of XP-RE (requirements engineering), take a look of how XP works with NFRs as a user stories and finally find main problems in capturing and satisfying NFRs.

3.4.1) Software Requirements

Requirements, we always hear this word when we talk about software, either if we are developers or customers.

If you look for software requirements definition, you will find a lot of them and a lot of standard definitions; simply it is “what the user and customer want”.

In the following table you can find some of software requirements definitions:

<table>
<thead>
<tr>
<th>Source</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Guide to SWEBOK [10]</td>
<td>Is a property which must be exhibited in order to solve some problems in real world by using software.</td>
</tr>
<tr>
<td>Software Engineering, Ian sommerville, 7th edition [9]</td>
<td>It may range from a high-level abstract statement of a service or of a system constraint to a detailed mathematical functional specification</td>
</tr>
<tr>
<td>Wikipedia [23]</td>
<td>It is a statement that identifies a necessary attribute, capability, characteristic, or quality of a system in order for it to have value and utility to a user</td>
</tr>
</tbody>
</table>

Table 3.4.1-1
These examples of software requirements definitions lead also to different classification of software requirements. Somerville stated that there are two types of software requirements; user requirements and system requirements. Software engineering, theory and practice divided them into functional and non-functional requirements. Another distinction can be found on SWEBOK which has many divisions like, product and process requirements, system requirements and software requirements and functional and non-functional requirements.

Since our concern in this thesis is NFRs, we will stick to one classification which is functional and non-functional requirements.

Functional requirements as defined in SWEBOK are the requirements that described which functionality will the system contain e.g. the user shall be able to add, delete new employee. [1][9][10][11]

For non-functional requirements, we do not have one formal definition and it is one of the main NFRs problems. Glinz [7] summarizes the major definitions of NFR in the following table:

<table>
<thead>
<tr>
<th>Resource</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anton [24]</td>
<td>Describe the non-behavioural aspects of a system, capturing the properties and constraints which a system must operate.</td>
</tr>
<tr>
<td>Davis [25]</td>
<td>The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability, understand ability, and modifiability.</td>
</tr>
<tr>
<td>IEEE 610.12 [26]</td>
<td>Term is not defined; the standard distinguishes design requirements, implementation requirements, interface requirements, performance requirements, and physical requirements.</td>
</tr>
<tr>
<td>IEEE 830-1998 [27]</td>
<td>Term is not defined. The standard defines the categories functionality, external interfaces, performance, attributes (portability, security …) and design constraints. Project requirements (such as schedule, cost, or development requirements) are explicitly excluded.</td>
</tr>
<tr>
<td>Jacobson, Booch and Rum Baugh [28]</td>
<td>A requirements that specifies system proprieties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement.</td>
</tr>
<tr>
<td>Kotonya and Summerville [29]</td>
<td>Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet.</td>
</tr>
</tbody>
</table>
“… global requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like. (…) There is no complete list of non-functional requirements.”

The behaviour proprieties that specify functions must have, such as performance, usability.

A property, or quality, that the product must have, such as an appearance, or a speed or accuracy property.

A requirement on a service that does not have a bearing on its functionality, but describes attribute, constraints, and performance considerations, design quality of service, environmental considerations, failure and recovery.

A description of a property or characteristic that a software system must exhibit or a constraint that is must respect, other than observable system behaviour.

Requirements which specify criteria that can be used to judge the operation of a system, rather than specific behaviours.

Requirements which impose constraints on the design or implementation (such as performance requirements, quality standards, or design constraints).

By taking a quick look to table 2.3.1-2 someone can notice IEEE 830-1998[7] definition which is “Term not defined” and that emphasises that we do not have any formal definition for NFRs.

These problems mainly because someone can express it in a way that we consider it NFRs and some other express it in a functional way, e.g. “The system shall only authorize access to those who are registered as system users”. This can be rewritten as “the system shall have login screen that force the user to enter user name and password”. [7]

All of those NFR requirements and more will be discussed later on this part.

3.4.2) we need a process
Discussions at the previous parts lead us to think about, which process of capturing NFRs shall be used, if such a process does not exist, then how can we develop processes that decrease the probability of system misunderstanding by developers?

As it is described in [12], there are four main principles to apply when we develop and evaluate methodologies:

1. Face to face communication is the best way for exchanging information: if the user can explain his/her needs directly to the analyst, then the analysts can ask directly about what s/he does not understand. This means; we avoid the time delay for returning to the customer to ask him. As well as, we avoid building wrong software. Then, the requirements process would not take any time overhead through the whole process.

2. Do only what you have to do: so any unnecessary documentation or artefacts shall be avoided to keep the time for the actual development.

3. Bigger teams need heavier methodologies: if you have a team of 5 people working on the same site and room. This is not as having a team of 200 who works on different location. Of course, the methodology you use for the five will be easier than the methodology needed for 200 developers.

4. Formal processes are needed for greater criticality projects: project criticality is the major factor in choosing development methodology. If a company develops a personal website or a newspaper website that is different from developing a nuclear weapons control system. Since any mistake in newspaper website will not affect any human life, but in case of nuclear weapons control system it may kill thousands of people.

When anyone thinks of developing a requirements process methodology s/he should think about those four main factors.

We can add one main principle which is: develop a process that works well with the company environment and projects types. So, companies may use a like RUP software development methodology by adapting RUP in its projects and company environment or it may develop a complete new software development methodology.

3.4.3) properties of XP- RE process
When it comes to measure how strong our requirement engineering process is we have to measure the quality of it is output (requirements artefact).

The following table shows the quality factors of XP software requirements:

<table>
<thead>
<tr>
<th>1. Unambiguous</th>
<th>+</th>
<th>13. Electronically Scored</th>
<th>+/-</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Complete</td>
<td>-</td>
<td>14. Executable/Interpretable</td>
<td>+/-</td>
</tr>
<tr>
<td>3. Correct</td>
<td>+</td>
<td>15. Annotated by Relative Importance</td>
<td>+</td>
</tr>
<tr>
<td>5. Verifiable</td>
<td>+</td>
<td>17. Annotated by Version</td>
<td>+</td>
</tr>
<tr>
<td>6. Internally Consistent</td>
<td>+/-</td>
<td>18. Not Redundant</td>
<td>-</td>
</tr>
<tr>
<td>7. Externally Consistent</td>
<td>+/-</td>
<td>19. At the Right Level of Detail</td>
<td>?</td>
</tr>
<tr>
<td>8. Achievable</td>
<td>+</td>
<td>20. Precise</td>
<td>?</td>
</tr>
<tr>
<td>9. Concise</td>
<td>+</td>
<td>21. Reusable</td>
<td>?</td>
</tr>
<tr>
<td>10. Design Independent</td>
<td>+/-</td>
<td>22. Traced</td>
<td>?</td>
</tr>
</tbody>
</table>

+/- indicates XP both assists and degrades. + indicates XP may assist in this area. - indicates XP degrades this area. ? indicates XP has little bearing on the area.

In table 1-3.4.3 we notice that by using XP we generate very good requirements, unambiguous, complete, correct and all the plus (+) quality factors of software requirements. Also, there are a lot of question marks (?) about some other attributes like cross-referenced and reusable. The following paragraphs include analysis of table 1-3.4.3.

1. Unambiguous, correct and understandable: since the customer is involved through the development process. Then, the developers can ask, get information any time they want. As well as, all requirements are written by customer as user stories. Then, XP requirements are said to be unambiguous, correct and understandable.

2. Verifiable: acceptance testing is written and run by the customer which yields they are verified by the customer.

3. Modifiable: the main purpose of XP is to accept changes at any point of time through the development process. XPers design it to do so. In XP- at any time- the customer can change the requirements by writing new user stories. As long as, systems are developed incrementally. Then, we can say XP shall have a high grade in modifiability.

4. Annotated by relative importance: in XP, the developers ask the customer to prioritize stories. Then, they implement the most important stories first and that is why XP is given a high grade in this quality factor.
5. Achievable: XP divides the development process into releases. In each release they provide some business values to the customer. If there are some critical parts that should be implemented first, they start with them. This means that at the end of the development process developers will be done with all stories.

6. Design independent: this is considered one of the main requirements for software requirements. Nevertheless, user stories may be very design independent, but since XPers write unit test first which is based on user stories. In reality, system design and architecture is mainly affected by this.

7. Electronically stored: in XP it is recommended to use any automated processor, because customer can update and change as s/he wants. But it is not a condition and they can use a carton (paper) cards for writing the stories.

8. Complete, concise: since XPer mainly focus on the most important features through the development process then it is said to be concise.

3.4.4) FR as user stories

As you can notice in the previous sections, each requirement in XP shall be a user story (functional or non-functional). In this part we introduce two examples of functional user stories.

First, the customer writes “I want to be able to add new employee, delete and update his/her data”. This story can be broken into three tasks 1) add employees to the database 2) update employees information 3) delete employees from the database.

Another example is “I want to be able to assign an employee to a team and therefore to a department”. This is story includes two tasks 1) assign employee to a team 2) assign employee to a department.

The above examples show how XP works with functional requirements as a user stories. [4]

3.4.5) NFR as user stories
As we have discussed earlier in this thesis, XP deals every single requirement as a user story. This section discusses how XPers can express NFRs as user stories.

We will use some examples. Starting from the following user story “I want the developed system to work in UNIX, all versions of windows and MAC” this user story expresses portability requirements. Another example, “the system shall respond very fast to any change in the database” this expresses the performance requirements, and so on.

The above two examples show how customers can express NFRs as user stories, but this raises a big question in developer’s mind; what if our customer is not from a technical background? What if customer does not have any idea about NFRs?

There is a fact said that “You always have to dig for NFR”. This problem along with some other problems is discussed in 4.0. [14]

3.5) Testing in XP

One of the major principles of XP is writing (programming) test cases before start programming. In XP, developers write a program module or a function to pass the test cases, so they do not write any new line of code until testing fails because the program does not have that line of code.

Someone may ask what I can benefit from writing test cases before programming. Is not better to write unit test after completing programming? Well, in case when we write unit test before program we can gain the following:

1. We do not write any extra single line of code, since we write that line when it is needed.
2. Testing first gives us the ability to get the right structure of our program.
3. We can guarantee that all of our code is fully tested.
4. It reveals the design earlier.

For integration testing, every time we integrate our system (every one or two hours a day), integrator (developer) shall make sure that the system debugs correctly and shall test it.

Acceptance testing is a critical part of software testing, since the customer decides whether developers develop the right system or not. Agile acceptance testing is always written by the customers, developers help them writing their acceptance testing. After each iteration acceptance testing carried out to ensure that developers are in the right track. [3]

3.6) Refactoring
In any software development methodology, there is always a need for refactoring. Refactoring refers to change in the entire code (module, function and class levels) in such a way that it does not affect the external behaviour of the system. This refers to only clean our code and improve our code’s internal structure.

Someone may ask, why do we have to refactor? We are refactoring when we need to change something in the code to complete its own task e.g. if we have a function that calculate the summation, multiplication and division of two float numbers. We could do it in one function called Calac (float x, float y). Next we realize that it would be better if we divide each function so instead of having one function. We can build three functions sum (float x, float y), multiply (float x, float y) and divide (float x, float y). After that we may think it would be better if we build a class and includes all the three functions, so if we need a new function like maximum of two float we could easily add it to that class.

By this way, we make our code more readable and easy to change. [1]

3.7) Agile Design

Before start building your house, you have to have the blueprint (design) of your house. It is the same in software development, before start writing any code of my program we shall draw our architecture. [9]

In agile development methods we want our design to be as simple as possible, high cohesive and low coupled. Martin, in his book [3] described the bad design as:

1. Rigidly: the system is hard to change, because every change you made intend many changes through other code modules.
2. Fragility: changing one module causes problems in other modules that are not conceptually related.
3. Immobility (Un-reusable): you cannot reuse your own code in other modules.
4. Viscosity: doing things right is harder than doing things in a wrong way.
5. Needless complexity: the software infrastructure contains extra functionalities that adds no benefits to our code and only adds complexity.
6. Needless repetition: the design repeats structures which can be coded as a single structure.
7. Opacity: It is hard to read and understand the system design.

Then, he described the good design as:

1. The single-responsibility principle: when we design our code, we shall design it to have one responsibility and nothing more. If our
class has many responsibilities. Then, when we change we may add many other responsibilities. Thus, our class or module becomes more coupled.

2. The open-closed principle: you should design your code in such a way that when you refactor it, you refactor it by adding new code and not by changing the old code.

3. The Liskov substitution principle:”what is wanted here is something like the following substitute property: “for each object O1 of type S there is an object O2 of type T such that for all programs of type P defined in term of T, the behaviour of P is unchanged when O1 is substituted for O2 then S is a subtype of T” [3].

4. The dependency-inversion principle: this principle includes two parts:
   a) High level classes or modules should not depend on low level one and low level modules should depend on high level module.
   b) Low level modules should not depend on lower level detail and details shall depend on low level modules.

5. The interface segregation principle: this is based on the fact that we should not have fat interfaces or classes. That means we can separate the interface or the class into a group of methods, each group serves a type of client.

For more details, see [3] (page 85-147)

In agile we are trying to apply all the five SOLID principles, so we have a good design.
4.0) The Problem: NFR problems with Agile Methods

In this part of thesis, we discuss main problems in satisfying NFRs when applying XP practices. Generally, we can divide NFRs problems into two categories:

1. NFR general problems that are natural to them, like definition and classification problems.[7]
2. XP- specific problems that only appear when using XP, like security problems.

As you can see in 3.4.1, researchers and organizations have not agreed on a specific definition for NFRs, this is the first problem when discussing NFRs, how to discuss something that we cannot define?

Second, the classification problem of NFRs, as a result of definition problem, sub-classifying problem appears for researchers, so no agreement about sub-classifying NFRs.

Third, the representation problem, in some context one requirement said to be functional, but if you rewrite it in other words you can turn it into non-functional one. This problem can be extended with the question how can we represent NFRs in diagrams? Are there any methods for representing NFRs? In the next part, we try to find some solutions for this problem. [7]

Fourth, NFRs are said to be subjective, which means, different people may have different opinions, e.g. for my friend MS word2007 is not a user friendly system but for some other people it is very user friendly. [2]

Fifth, NFRs are interacting, so achieving one NFR may affect other NFRs either positively or negatively. E.g. if customer wants a very secure system, in most of the cases this will hurt the system performance. [2]

Sixth, NFRs are relative, means the importance of NFRs varies from one application to another application. E.g. the performance requirements for Wall Street system are very important, but in on-line dating system are less important. [2]

The above problems are general. All of those problems appear in all software development methodologies. Maybe there are more general problems, but those are the most known problems.

Weinberg and Schulman [22] in their old-famous experiments about NFRs obtained that if developers (team) focus on one or a set of NFRs, they will forget other NFRs. That means developers focus on satisfying a set of NFRs only. E.g. if a team is told to build a system that is maintainable and easy to use, they may
build that system correctly, but there is a very high probability they will forget other NFRs. For example, they will forget the performance requirements.

One more important issue with NFRs, as we have mentioned previously in this part, NFRs are interacting and in other words there is a lot of trade-offs among NFRs. In reality, security and performance are crosscutting requirements and they are said to harm each other. For example, if we want very secure systems, then we have to use a good technique and highly secure algorithms. As well as, we may use some cryptography algorithms or zero knowledge techniques which are in most of the cases very slow. This will slow down our system, and then harm the system performance. In case if these two requirements are important like in bank automated teller machines, then we have to find a way for satisfying these two NFRs. [2]

One of the agile methods RE weakness is considering NFRs only at implementation level, which may lead to some problems in the delivered software. In reality, some projects may fail because of lack of features. Here comes a need for considering NFRs in another way or by using some techniques like misuse case or NFR-framework. [2][15]

A lot of questions have been raised regarding security in XP. In Common Criteria (CC), there are seven evaluation levels (EAL1-EAL7). Since XP does not use a formal method, how can we evaluate it?

Some researchers try to identify the security level that can be achieved when using XP and the result was as follows:

1. XP cannot be rated over 4, since it misses formal and semi-formal design documentation which is the condition for level 5, 6 and 7.
2. XP covers part of assurances in EAL3 and EAL4.
3. XP fills requirements of EAL2.

Generally, XP shows some weakness in covering and fulfilling CC requirements in higher levels, so we cannot guarantee that we will get a very secure system. [15]

Another general problem is dealing with crosscutting requirements, means how we can work with functional and non-functional requirements at the same time. In actual development, some functional requirements effects other NFRs or may some NFRs effects other NFRs. [16]

Customer collaboration through the development process helps the developers to achieve a very usable system, since the customer can complain about the usability directly. On the other hand, Weinberg and Schulman experiment comes to the front. If we think about that carefully, we can say that our customers can complain about usability, but what about the system maintainability or security.

In addition to all of the above, most of methods that deals with NFRs, are found to evaluate them after the implementation is completed and software is up and working. For example think aloud method for evaluating system usability and the above CC method for evaluating software security. [2]

XP and agile development methods test first driven approach helps the developers- if they consider NFRs from first- to think about developing automatic
testing of requirements that are testable i.e. performance which is already built in Xunit testing frameworks.[17]

As a result of this, we need a method that helps the developers satisfying NFRs. Some solutions have been proposed such as using some UML notations for representing and connecting NFRs with FRs, using misuse cases, i* framework and NFR-framework.[2][18][19]

In this thesis, we chose NFR-framework as a solution for our problem, because of its simplicity and operability.
5.0) Toward the Solutions

Many solutions have been developed for solving NFRs problems.

First of all, if someone interested in NFRs and searches through the internet, s/he can find documented guidelines for electing NFRs. One of these approaches that guide us in electing NFRs is on figure 1-5.0:

Figure 1-5.0 shows the analysts how they start electing (defining) NFRs, by electing or asking for all kinds of NFRs from usability requirements until implementation constraints requirements.

The following section (5.1) introduces systematic approaches for modelling NFRs.
5.1) NFR framework:

**Definition**: NFR-framework is a set of notations and graphs for representing NFRs in a systematic way. It helps developers and analyst to deal with NFRs by choosing among different design and decision alternatives.

**Summary of the definition**: by reading the definition we can extract the following:

1. NFR-framework consists of different notations and diagrams.
2. NFR-framework is a way for representing NFRs.
3. The graphs and notations include design and decision alternatives.
4. NFR-framework helps developers choosing among different design alternatives in order to satisfice NFRs.

[2]

5.1.1) Using NFR-framework

First of all, we introduce how to use this framework and how it deals with NFRs.

Before starting with NFR-framework, there are a number of terminologies which are important to know:

**Operationalizations** which refer to simplify our problem by extending our softgoals into subset of those softgals, by identifying our design alternatives to meet NFRs. Specific opertioanlizationsn can also contain another subset of operationalizations and so on…

**Design rational** which is a justification of design decision, which more specifically means “why we choose a specific design decision in our model.

The following shows how NFR-framework works:

1. Collect information about the development environments.
   A. Domain information.
   B. FRs.
   C. NFRs
2. Identify the system NFRs.
3. Decompose those NFRs.
4. Identify “operationalizations”.
5. Identify and deal with ambiguities, tradeoffs, priorities and interdependencies between NFRs and operationalizations.
6. Select the design rationale from the design alternatives.
7. Select operationalizations.
8. Evaluate the impact.

One more thing to add here, these steps are not necessary to be sequential, but some of them should be completed before some others. Next in this section, we describe the whole process behind NFR-framework.

We can represent the flow of process as in fig 1-5.1.1. From that figure we can observe that, we can complete some processes simultaneously, so we do not have to complete them in the same order as described before. For example, we collect information about development environment and identify project NFRs at the same time, so we save our time and the same thing for selecting design rationale and selecting operationalizations.
Collect information about Dev. Env.

Identify project NFR

Decompose NFR

Identify amb., prioritize and trade offs

Select Design Rational

Select Operationalisations

Evaluate the Impact

[2]
5.1.2) SIG (softgoal interdependency graph)

SIG is the graphical representation of the framework. It is used by developers and analysts (next in this thesis, we use developers to refer to developers and analysts) to identify NFRs which is represented as a cloud in this framework; decompose those NFRs into sub-NFRs i.e. System performance can be decomposed into response time and amount of storage used on hard drive. Then, go down more through sub-softgoals. Priorities are identified by exclamation mark (!). Interdependencies links are used to connect softgoals. An evaluation procedure is used to identify which softgoals are achieved and which are not.

SIG is very important for developers because it helps them dealing with NFRs in a graphical, simple and systematic way. We use SIG in our process model as the most important part for achieving NFRs. For full list of diagrams of SIG see [2].

![Figure 1-5.1.2][2](image)

Figure 1-5.1.2 shows a complete example of SIG that can be used for credit card system. In this example we have three main NFRs:
1. Security: the system shall be secure and it should achieve all the three parameters of security (confidentiality, Integrity and availability).

2. Performance: the system shall have a reasonable response time, so the customer should not wait for long time for the data to be processed.

3. Usability: the system shall be easy to use and user friendly.

Through the rest of discussion, we explain how can we collect information and go though all steps for constructing this diagram.

**Cataloguing design knowledge:** NFR-framework enables developers to document acquired design knowledge. This step is mainly based on the previous experience of developers.

There are three types of cataloguing developers can deal with:

1. NFR being considered, such as system performance and its associated terminology.

2. The second type includes development techniques that can help in meeting these requirements.

3. Interdependencies among softgoals.

The knowledge of the first type can be acquired from multiple resources such as text books and developers’ personal experience. The other types of knowledge can be acquired from industry specialists and academia.

**Acquiring domain knowledge:** collecting and documenting knowledge about the system domain. This may include functional requirements and requirements priorities.

**Acquiring and cataloguing NFRs:** developers draw cataloguing of NFRs along with its development techniques. There are three types of those catalogues:

1. Collecting knowledge about particular NFRs types and associated concepts.

2. Design techniques. Such as “use uncompressed format” for performance.

3. Implicit interdependency which refers to the effect of choosing one design technique on other NFRs.

**How to construct and build SIG for a specific application?**

1. Identifying NFRs: first developers shall identify which NFRs their application will provide. They can do so by classical tools for collecting software requirements or by user stories as in XP, but in case of XP the developers shall ask the customer to provide them with a special user stories about such requirements.

2. Decomposing softgoals: in this stage the developers decompose softgoals into sub-softgoals according to NFR type (i.e. Performance) or topic (i.e. for credit card system, we have the same topic which is the account). The main purpose of this stage is to break
down NFRs into more detailed ones and identify the relationship between sub-softgoals. In our example security can be broken down into three sub-goals (integrity, confidentiality and availability) so for the system to be secure it shall satisfy all of the above three mentioned requirements. Identify the relationship among sub-goals in our security it is logical AND means if one sub-goal has not been satisfied then security has been violated.

3. Dealing with priorities: the developers in this stage identify which goal has more priority than others. In our example the most important softgoal that shall be achieved is “accurate account” so we mark it by exclamation mark (!).

4. Identifying operationalizations: in order to achieve NFRs and as we can see in 4.0, we have to think about NFRs at all levels starting from collecting requirements, design and implementation techniques. Choosing implementation technique is very important for us, because there are always gaps between NFRs and implementation techniques. Our task is to bridge the gap between them. In our example, for response time we have decided to use indexing and uncompressed format. This decision helps in acquiring a very good response time therefore achieving the account performance.

5. Implicit interdependency: while we are trying to achieve some softgoals i.e. security, some other NFRs may be affected either positively or negatively. In order to enhance our decision of choosing among different operationalizations we have to take these affected NFRs into consideration. For example in 1-5.1.2 when we are trying to choose which implementation technique to be used for achieving confidentiality, we found that one choice may affect usability. Thus, we decide to eliminate that choice from our alternatives.

6. Selecting among alternatives: in this stage, after we list all possible design alternatives and design rationale. Next, the developers shall select the design alternatives. Then, they decide and record their decisions by using the dark bordered cloud.

7. Evaluation the impact of the decisions: in the step, developers determine theoretically if the high level softgoal is achieved or not.

8. Relating FRs to NFRs: through 1-5.1.2 we have only concentrated on NFRs, decomposing them and listing design and implementation technique. But it is very important to connect NFRs to FRs; NFR-framework enables us to do that, so in our example we can finally connect the entire graph to “maintaining account” FR.

By looking through this example, and all over the steps we can extract the following about NFR-Framework:

1. NFR-framework helps us understanding NFRs more and more.
2. SIG solves the representation problem of NFRs.
3. SIG enables developers to define sub-goals (sub-NFRs) and concentrate on all of them at one time.

4. SIG enables developers to list and choose the appropriate implementation techniques.

5. SIG helps the developers to connect FRs to NFRs and bridge the gap between the two types of requirements.

6. SIG is an easy way of documenting NFRs.

7. SIG is simple, easy to learn and applicable in most of development cases.

The last part of this section provides developers with heavy and valuable information about documenting and representing NFRs.

More on NFR-framework can be found on [2]. Next section (6.0) includes more information on how to apply this SIG on XP.
6.0) The Solution

In this part, we build a process model which is compatible with XP for helping experts’ satisficing NFRs. This model is mainly based on two previously discussed concepts:

1. User stories. See 3.4.
2. SIG. See 5.1.

This model also has a set of requirements that shall be satisfied:

1. It shall be simple and can be applied by XPers and all agile development methods.
2. It should help in satisficing NFRs in XP.

In another words, what we try to build is a set of guidelines that help developers in satisfying NFRs. This process model is introduced in 6.1.

6.1) The Process Model

As we have seen in 3.4, every single requirement shall be written as a user story either functional or non-functional. So, if we try to work with any single requirements it shall be written as a user story. Our first suggestion to companies is to start by writing a special story. Let’s call it NFR-story. This story shall be written using automatic text editor to be kept for future reuse and at the first iteration, for the first approach.

Somebody may say “what if my customer is not from a technical background? How can s/he write such a user story”. The answer for this question is rather simple. Since our customer is available all the time and s/he writes user stories by our help, then we can help our customer by:

1. List all NFRs related to your project. There are some checklists for NFRs. Those checklists can remind developers with all kinds of NFRs and by double-checking which of those requirements are related to their domain.
2. Asking the customer NFR-related questions and writing down the answers. I.e. who should be able to use the system, delete, update and add? What type of users do you have and how they are connecting to each other?
3. Check every NFR the customer stresses more.
4. You may need to divide this story into two or three, because of paper space availability, but we prefer them to be on single paper. After that you have to write it down using one of the text editors like Open Office or MS word.

5. Complete your planning phase as usual.

6. After completing all the user stories and the NFR-story, we have to draw the SIG for the listed NFRs. By drawing the SIG we reveal the design and implementation techniques in advance, but here we have two problems, first do we have to have all NFRs from the first? But that violate the XP rules and the second problem: we do not have all functional requirements. So, how can we connect NFRs with FRs if some NFRs are related to FRs that is not listed? For simplicity we will leave these two questions for now and we will back to them in 6.2.

7. Write our test cases according to our user stories and the decisions we have made when you draw the SIG (implementation techniques and design constraints).

8. Next, we build our system design to be equivalent with the output of testing phase and SIG.

9. Implementing the system and for each implementation technique, check if what we have developed is also equivalent to what we have planned by SIG.

From the above discussion, we can extract some extra benefits from applying SIG other than helping in satisfying NFRs, those benefits are:

1. It is helpful for revealing the system design, compare to test-first principle.

2. It helps us in advance to know which implementation technique is more helpful and enforce us to think about NFRs from the first stages of the process.

6.2) Applying the process model:

Somebody may ask “how can I apply this model? What is the best way to apply it? And when?”. In this section we tried to find solutions for these questions. The flow of the processes in our model is shown in Fig 1-6.2.

There are three main choices for the time of drawing SIG:

1. **Draw it once**: means that you have to inspect all NFRs and you have to draw SIG for all of them from the first release.

2. **Draw it every release** and compare the output with the intended implementation technique to be used. This means you have to list all NFRs and draw the updated version of SIG in every release.
3. **Make it incrementally**: for each release think for only a sub-set of NFRs - this subset shall include all NFRs in that release-and each time you plan for iteration add the new subset of NFRs.

Many people may have different perspectives in the way they want to apply it. So, you can find people who prefer to apply it using the first approach, some people does not like it all, because they think it breaks XP rules or maybe they have a different way of applying this process model.

For the first approach, it is recommended by most of researchers to think about NFRs from the first stages of development. It can help developers to reveal all the implementation techniques that they will use. That is very good, but in order to make it complete you have to have all related FRs and this may not happen. Since, in XP they only complete subset of FRs and think about those FRs, so it will not be useful if there are some other FRs that are not considered to be connected with all NFRs.

Another problem, one of the main XP rules is to think only about what we are doing now, and do not prepare anything in advance. That is because everything is going to be changed and by using this approach we violate this XP rule.

We also have the fact of refactoring; this intends that any change to the system design or implementation technique will violate SIG. Thus, it becomes completely useless. Then, we lost our time by applying it.

For solving such problems we can think about what is the most important; sticking to XP and agile rules or implementing what our user wants. As well as, how much these NFRs are important in our project. The decision shall be made by the whole team. In case of applying it once, we have to inspect our design every new release and to solve the problem of connecting it with FRs we have to take a look to our SIG and update it with the new FRs. In this case, we are really doing it incrementally, but it is needed since our design and decisions may differ when we have new FRs and if we stick to the old one we may not succeed in achieving NFRs.

In the second approach, when we apply the model every time and compare it with the implementation technique, this could be more applicable than the first one. Since in each time we refactor we have to think about those NFRs more and more.

In the same way, every time we add new functionalities we update our design and we apply the whole model again.

Comparing to the first one it is also violating the same XP rules that the first approach violates. But it is more applicable by enforcing developers to apply it each time we develop new release. Then, developers in advance releases will not forget to think about NFRs or applying the model and comparing it. At the same time it solves the problem of the first approach of adding new functionalities. Thus, every time we add new functionalities, we update our SIG and we do not stick to one which may be useless in advance releases.

It may also be harder to do it each time since we already have it from the first iterations and Xpers may think we lost our time by doing so. From our point of view, we do not lose any time by applying it every release since we only update (refactor) what we already had and we may not need to update it in some cases.
In this approach, we also need to have all NFRs from the first release, so we only add the new functionalities each time we apply it. In general, this approach is more applicable than the first one although it has some problems.

It may seem to us as applying the third approach is more suitable in the case of XP and agile development methods. That could be if we connect the intended FRs with NFRs and each time we add new functionality we apply it. In the case of the third approach we solve all the problems in the first and second approach.
First, we draw the SIG for only our FRs and NFRs in the first release and when we have new functionalities to be added, we connect it with its NFRs and with the old SIG. Then, we do not need to have all NFRs from the first release and we do not have the problem of connecting NFRs with unimplemented (listed) FRs. The second problem of violating the XP rule of thinking about FRs or NFRs in advance is solved by only working with the current and old requirements.

In this approach developers are forced to inspect the system design each time we draw new SIG. It may be harder to find the intended NFRs which can be connected with the new FRs and it may take more time.

One more problem which is applied for all approaches, how can we write test cases for NFR-story? Actually there are some tools that support automated testing of NFRs like Junit, but these tools do not support automated testing for all NFRs. Then, we may use acceptance testing for inspecting NFRs, but that may violate the test first driven approach. It seems that it is more complicated to automate NFRs testing, so we can use observation testing as we have mentioned in acceptance testing.

Unfortunately, we do not have any experimental studies that show us which one is more applicable. Also we do not have any experiment that proof the applicability of this model at all.

6.3) Process model(approaches) VS XP

In this part, we discuss the compatibility between our process model and XP principles. First, we summarize our work in the following table, which includes all of the XP rules and our approaches.

The sign “+” indicates that we follow XP rule, “–” shows that there is no way to follow that rule, and “+/–” means it depends on how we apply our approach. See table 1-6.3

<table>
<thead>
<tr>
<th>XP rule\approach</th>
<th>First alternative</th>
<th>Second alternative</th>
<th>Third alternative</th>
</tr>
</thead>
<tbody>
<tr>
<td>1- Customer involvement</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>2- Using user stories</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>3- Short iterations</td>
<td>-</td>
<td>+/-</td>
<td>+</td>
</tr>
<tr>
<td>4- Acceptance testing</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>
While we investigated XP-rules and match them with our alternatives, we discovered that most of XP-rules are related to the process of development and not with the entire implementation. This helps us in positively matching almost most of the principles with our approaches, since all approaches follow the same procedure.

For example, 1, 2, 4, 6, 7, 8 and 12 are completely compatible with our process model in all the three approaches.

For the first principle, it is obvious here that in our process model, we rely too much on customer involvement through the process and then it is part of our process model.

For short iterations, it is obvious that we follow that rule even in building NFR-SIG in third approach, since we follow the incremental approach.

In the first alternative we shall have all NFRs from the beginning of the development that means we only have one SIG iteration, but we may have many FRs iterations.

In the second alternative, it is not obvious if we really have many iterations or not, but in most of the cases we have to update our SIG many times during the
development process. It depends on the project itself, if it has many updates then we follow that principle otherwise we do not follow it.

In our process model, we do not identify if XPers shall use pair programming or not and the same for 9, 11, 13 and 14. For those principles, we recommend all developers to follow them while applying our process model.

In most of the cases, we need to re-structure our code; the second and third alternatives follow this rule of refactoring. In the first approach we stick to one and only one SIG, which means that we do not follow this rule.

In all the three alternatives, the SIG affects the design, which means we cannot guarantee that we will have the simplest design every time.

From table 1-6.3, we can observe that we follow almost all the XP-rules which are related to the process of development in the three approaches. We also observe that, for the second and third alternatives we have more positive signs.

We recommend developers who are using this process model to follow other XP-rules that are not mentioned in the model such as metaphor, pair programming…etc.
7.0) Industrial meeting and feedback

In this thesis we have relied on theory, starting from the first parts until the model construction and model developments. In order to be more precise in this thesis we have to have some kind of practical experiments. Since we do not have the time and budget for that, we have decided to present our model to a very experienced consultant that has been involved in many projects from small to very large scale projects.

In Thursday, the 3rd of September, we met Thomas Nilsson from Responsive development technologies, presents our model and discuss our approaches for 1 hour and 20 minutes.

Nilsson and his partners have almost 30 years of experience in the industry. They have a great experience in method development and deployment. In the last few years they become more and more specialized in agile methods and principles.

We first introduced our process model and the three approaches. The first one, which is based on collecting all NFRs from the beginning of the development process and connect them with the intended FRs. The second one, which is based on collecting FRs incrementally and all NFRs from the beginning and update our SIG each time we have new FRs. The third one, which is based on incremental development process, by drawing the SIG only for FRs and NFRs in that release and each time we have new release we update our SIG.

The conclusion of his feedback was:

1. The first approach is not applicable in medium to large project, but it could be good for small projects like building a personal website, when we have all functional and non-functional requirements from the beginning of the project.

2. We discussed the idea of Non-functional story, according to Nilsson “in most of the cases, developers forget to think about Non-functional story; it would be good to have it”. On the other hand, he was not satisfied with the idea of having one or two documents holding non-functional requirements, since we may have very big non-functional requirements that need a number of iterations to implement. For example building a GSM-system to handle one million user, then we have to have one Non-functional story and we have to split its implementation over many iterations- and the splitting stories may become a complete big project by itself.

3. For the second approach, we discussed the idea of updating the NF-story (ies), at that time the idea of NF-story refactoring (see 7.3) up to the discussion. A comparison on how we could work with small systems when real customers are available all the time or when we have to have a connector with customers, since we may have many customers and they cannot appear (see 7.1 and 7.2).
For the second approach Nilsson was satisfied with some restrictions regarding large projects.

4. After that, we discussed the third approach; Nilsson thinks this is the real agile implementation, since we implement it in the agile way. His feedback was “we always have to change in our requirements, we add new requirements and we eliminate others” and according to his estimation “more than half of the requirements could be changed through the development process”. In case of the third approach we will be able to handle most of the changes and refactor during the whole process”. He also has some restrictions regarding large systems and the hard implementation of non-functional requirements.

5. According to Nilsson, we have to distinguish between the testing processes; unit testing is completely different than acceptance testing. We perform unit testing at code level, but we can do black box acceptance testing. He also brings the idea of writing test cases for NFR-requirements as early as possible for system and integration testing.

6. The main purpose of unit testing is to have all functionalities tested, but the main purpose of acceptance testing is to know how system is working.

7. Overall, according to Nilsson, our model is really applicable in reality. But we have to think more about large systems, for that reason we include a new part that describes how to apply it when we have a very large systems.

8. For usability and maintainability, our expert thinks that these should be the team responsibility, from his point of view, usability requirements can always be handled by having the customer with the development team. In case of maintainability, in his point of view, in most cases companies take contract for both building and supporting the system. Thus, team shall give more attention for building a maintainable system. Another way of thinking is that refactoring and continuous integration of systems is also another meaning of maintainability.

9. In case of large or critical software projects, we could use SIG as a communication way between development team, connector (the people who acts as a customer in the team hierarchy, more precisely the people who has a contact with customers and knows what actually the customer wants) and real customer as UML diagram, since it is easier to communicate with customers using such kind of diagram.

10. A new way of handling NFRs is to write on the header of each story what kind of NFRs is affected by that story. For example we can write “security” as a header of a sign-on page for a web-based application.
In the last two sections, we will discuss more regarding the feedback we got from industry, in 7.1 we discuss the applicability of the process in small systems and how to apply it in those kinds of systems, and in 7.2 we discuss large scale systems.

7.1) Small software Projects

In our meeting with the expert we noticed that we have to differentiate between two types of project. Those are small projects, when the customers are available and can be contacted at any time and large scale project when the customers are not available all the time.

In the case of small systems, it is feasible to have one or two documents for handling Non-functional requirements as NF-stories. Since it is easy to contact the customer in this case, we will not have any problem in changing and we can follow the agile process of development. Someone may ask how we can apply the first and second approaches in this case.

We can answer this question easily, since we do not have a very large, large or even medium sized project. We will not have too many changes through the process of development, but in case of large amount of changes, that will affect our development process too much. Even though working with the first approach will have some problems, but we are pretty sure that either second or third approach will work well in this case.

7.2) Large Software Projects

In our meeting with the industrial expert, he asked us to differentiate between small and large projects. In this part we focus more on large/complex software projects.

When you apply agile methods you suppose that all the team members are working on the same room and one customer is always available to support feedback regarding the project progress. Is this really applicable when you have a project with 200 developers and your target customer is not available all the time? [3]

Our expert said “suppose that we are asked to implement a system that should be able to handle 1 million users at the same time, do you think it is feasible to implement this in one iteration or part of the iteration”. Then he completed “the customer and the developers can write it in a one story, but it needs a number of iterations and when we split it using the SIG as a way of implementing it, how can we solve it?”

Here, the real answer and agile answer for this question, is to implement it using many Non-functional stories. Then we use the SIG for decomposing this
non-functional requirement will help us divide it into stories and tasks. See fig 1-7.2

By following this procedure we could get into the right track of developing single/agile non-functional stories, but that makes some conflicts with the original model. In the original one we write NF-story, draw the SIG and then write test cases, but in this case we get the second version of the process model.

![Diagram of Non-functional Story Decomposition](image)

Another problem may come into reader’s mind, which is how to integrate those NF/F stories. First of all, we want to emphasize one thing. The new NF-stories are not real NF-stories, because we implement NFRs in code or by using some kind of coding or designing technique. Thus, we prefer to name them as NF/F stories. For the integration problem we could benefit from Agile continues integration.

By deeply looking into the new defined process in 2-7.2, you may notice that we have a new way of applying our process model.

The only difference in this process model is we need to decide if we need to decompose the story into many new stories or not. This decision should be accomplished by expert developers. Since they can estimate the difficulty of this NF-story and divide those stories into iterations if needed and complete as usual.

Always in case of having iterations and versions we have to have some kind of configuration- or some kind of versioning method- management that can handle all changes. That kind of configuration management shall be for the whole development process. Since we are working here only with NFRs, we can leave
this process for the development team who is responsible for the whole process to make the NFRs part of this versioning method.

On the other hand, the difficulty of target customer availability and feedback in large project still arise, but- and according to the expert- we can have a connector who knows well the customer needs and can translate them.

After all what we gained by meeting the industrial expert gives us a good opportunity to improve our model development. It also helps us much in gaining more knowledge about the industry and the real difficulties they meet through the whole process.
Start

Write user Stories

NF-story

Draw SIG

SIG document

Need Decomposition

yes

Decompose

New F/NF story(ies)

Stop

Integration process

Acceptance testing

Story1

Story2

StoryN

Write Test cases

implementation

Fig 2-7.2
8.0) Case Study: Pizzeria calls handling system:

In this case study we are going to try a process of developing a system for handling incoming calls for a pizzeria delivery system.

Station Customer: Bob.

Team: Mazen, Ahmad and Basem.

As we have described in “applying the model”, first we ask our customer to describe what he wants and we ask him to write his user stories by helping of Mazen.

**Story1:**

“I want a system that is able to handle all incoming calls, by the customer phone number. I want the system to be able to retrieve all of my customer’s information by his/her phone number, I want the system to be able to show me the map to that customer’s house so I will be able to print it and give it to the guy who is going to deliver the pizza and I want the system to be able to retrieve all of my customer’s favourite stuff (name of the pizza and which kind of sauce s/he likes ...etc).”

Mazen reads the story and he feels it is a very big and long story “our customer describes the whole system in one story”, and then he asked him to divide the story into multiple stories.

Mazen gives our customer the following hint: “divide this story into functionality, you can divide it statement by statement (comma) and I will be with you all the time”.

Now and by the help of Mazen, our customer was capable of writing the following user stories.

**Story1:**

“I want a system that is able to handle all incoming calls, by the customer phone number.”
After completing these stories, Mazen was satisfied with them and feels now we could start. Basem (the one who is responsible for applying the model) asks the customer for a few minutes meeting, in order to write NFR story and he starts to dig for NFRs.

Bob: what shall I write now?
Basem: just answer my questions, and I will write.
Bob: OK!
Basem: do want your system to respond quickly?

**Story2**

“I want the system to be able to retrieve all of my customer’s information by his/her phone number these information include

1. Name.
2. Phone number
3. Address
4. Map to his/her Address”.

**Story3:**

“I want to be able to update/delete and print all the information for my customer”.

**Story4:**

“I want the system to be able to list all of my customer favourite stuff for regular customers and enables me to add such stuff for new customers, these stuff include:

1. Pizza Name.
2. Sauce name.
3. Extra things to be added on the pizza”
Bob: off course! I do not want to receive the information after my customer ends the call.

Basem: Ok! Who shall be able to see this information and who is going to use this system,. How many users are there?

Bob: we are 7 in this pizzeria “pripri” , I do not want to enter a password every time I get a call, but I do not want anyone to be able to update my customer information and if they are doing so I want to be able to know who has done that!

Basem:OK!

Basem wrote all of information he got and calls for a team meeting.

Developers start the meeting, Mazen leads that meeting (NFR meeting), customer attends and they start to make decisions:

Basem reads what he wrote and explained everything for the developers.

Ahmad says: our customer wants to get information once he gets a call, shall we use www.hitta.se for retrieving information about new customers and all the customers are new since the first call.

Bob: Aha, but I have my own paper database, this database includes all the information about my regular customers, is not a good idea to use it and only add the maps?

Ahmad: great!

Mazen: what about private calls (people who uses private numbers).

Bob: I do not get that much calls from private numbers and I used to give all of my regular customer ID number, so he gives me it when he calls.

Basem: can you estimate the percentage of private number calls?

Bob: less that 2%.

Basem: good we can use this ID along with the phone number, so if it is private we enables the user to enter this ID and returns all the in for.

Ahmad: can you estimate how many calls a day you get from irregular customers:

Bob: 1-3 calls out of 50.

Ahmad: then, it is not that much if we use hitta for those calls, but you have to have very speed internet connection, do you have?

Bob: No! But I will update it to the highest speed internet if needed.

Mazen: good, do that as soon as possible.

Mazen: what about the system security.

Basem: as I have explained, we do not have that much security.

Mazen to Bob: can you take a look to our prototype when it finishes, you check its usability.

Bob: I am working with you all the time, we can check together.

After that, the write the NF-Story as the following:
Mazen to Ahmad: draw the SIG and let us check that, how long will it take from you to draw it?

Ahmad: 1 hour.

Mazen: first we have to plan everything “Planning meeting”.

This was the discussion between the team and the customer, they are going to draw the SIG after they plan and then write test cases according to the decision they have agreed on through the meeting. They document everything in that meeting, but they need the SIG for future releases.

They have to write test cases for non-functional requirements. For example by the current support from Unit, they can write test cases for system performance, usability can be checked incrementally by the customer himself. They have to write test cases for security problems. Try to hack the system.

Mazen thinks about reusability of the system and maintainability, since in Sweden there are a huge number of pizzerias, so they can use this system more and more.

All meet to design the system and inspecting the design with the SIG and documented NFR meeting.

Since their system was simple and does not have that much of functionalities. It was easy for them to implement and apply SIG in XP. In case of large system with a huge number of functionalities, is it that easy to apply it? As we have discussed before, they can use another alternatives, like using SIG incrementally or for each release use it is related NFR-SIG.

We do not want to go more deeply through XP development process, like writing test cases and implementing the whole system, but we want to show how developers can benefit from both agile development methods and using SIG.

<table>
<thead>
<tr>
<th>NF-Story:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1- <strong>Performance</strong>: the system shall retrieve the customer information within 5 seconds.</td>
</tr>
<tr>
<td>2- <strong>Security</strong>: every user shall have ID and password and the only user who is able to update customer info is the admin.</td>
</tr>
</tbody>
</table>

*Hint: usability can be checked by our customer by allowing him to try the prototype and the system.*
9.0) Conclusion

Throughout our discussion we have seen that many problems in the software development area like delay in delivering software, going over the budget and changing in requirements through the development process. Many researchers and organisations have come with some solutions that said to be the best practice solutions. For example RUP and agile development methods like scrum and XP.

SCRUM is considered as one of the most used software development methodology in the IT industry, by that we can realise that IT industry is turning into agile development methods; because of its simplicity and can respond to changes. [3]

On the other hand, many researchers and organisations have complained about agile methods, such as they are not useful for large projects and it has a lot of problem with NFRs. [6]

In this thesis we discussed NFRs problems when you use agile development methods and we tried to find a good combination that enable companies to use agile methods and satisfy NFR.

We used XP as an example of agile development methods; since it is widely used and it is said to be more practical oriented than other agile development methods.

We constructed a model in a simplified DFD graph on how can we benefit from applying one of the most useful approaches for modelling NFRs (NFR-Framework) when using XP as a development method.

We have seen three ways of applying this model and we tried to mention the main advantages and disadvantages for each of them. Then we discovered that the third one is more usable than the others in term of XP.

After that we had a meeting with an industrial expert who for a long time used and taught agile development methods. As a result of this meeting we concluded that he agreed with us regarding advantages and disadvantages of the three approaches, but they think we have to differentiate between applying it for small and large project. That led us to construct a second version of the model which can deal with both cases.

Overall, and from our point of view, the most important thing for us is to succeed in our business with studying the future impacts. Then companies and organisations have to estimate if they need to apply this process model or not.

Companies always looking for a process for helping them get satisfied with their work and they may find a better way for their work other than formal development methodologies.

Finally, we hope to be able to experiment this model in real world.
References


På svenska

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

For ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

In English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© [Författarens för- och efternamn]