



SOFTWARE REQUIREMENTS

Course Material for SWEN 645 Software
Requirements

Abstract

SWEN 645 – Software Requirements - An examination of major models of software requirements and specifications, existing software standards and practices, and formal methods of software development. Topics include writing system and software requirements, formal specification analysis, formal description reasoning, models of "standard" paradigms, and translations of such models into formal notations.

University of Maryland Global Campus

Date Modified: 3/26/2020

Contents

FOREWORD.....	5
CHAPTER 1 INTRODUCTION TO REQUIREMENTS	6
DEFINITION OF SOFTWARE REQUIREMENTS	7
DEFINE REQUIREMENTS ENGINEERING	8
ON THE INTERSECTION BETWEEN REQUIREMENTS, BUSINESS, AND THE LAW	9
OTHER THINGS YOU NEED TO KNOW	12
CHAPTER 2 REQUIREMENTS PROCESS.....	19
PROCESS MODELS	19
PROCESS ACTORS.....	19
PROCESS SUPPORT AND MANAGEMENT	20
PROCESS QUALITY AND IMPROVEMENT	20
REQUIREMENTS FOR DIFFERENT SOFTWARE DEVELOPMENT METHODS.....	20
<i>Waterfall</i>	20
<i>Spiral Model</i>	21
<i>Agile</i>	21
<i>Scrum Model</i>	21
CHAPTER 3 REQUIREMENTS ELICITATION	22
DEFINITION	22
<i>References</i>	22
TRADITIONAL METHODS	22
WEB-BASED METHODS.....	23
ON GETTING THE INFORMATION FROM THE STAKEHOLDERS	23
CHAPTER 4 REQUIREMENTS DOCUMENTATION	27
REASONS TO DOCUMENT REQUIREMENTS	27
WHAT IS THE REQUIREMENTS DOCUMENT USED FOR	28
HOW USED THROUGHOUT DEVELOPMENT LIFE CYCLE.....	29
AUDIENCE FOR THE REQUIREMENTS DOCUMENT.....	30
COMPONENTS OF REQUIREMENTS SPECIFICATION DOCUMENT.....	30
HOW CAN REQUIREMENTS DOCUMENT LEAD TO PROJECT FAILURE	31
TYPES OF REQUIREMENTS DOCUMENTS.....	31
GOVERNMENT REQUIREMENTS DOCUMENTS	33
REFERENCES	33
CHAPTER 5 FUNCTIONAL DECOMPOSITION	35
INTRODUCTION.....	35
REQUIREMENTS STRUCTURE FORMATS	36
<i>Using mode:</i>	37
<i>Using user class:</i>	37
<i>Using object:</i>	37
<i>Using stimulus:</i>	37
EXAMPLES.....	38
CHALLENGES OF THIS APPROACH.....	39
GOVERNMENT PROJECTS AND SAFETY-CRITICAL SYSTEMS	39
REFERENCES	39
CHAPTER 6 USE CASES	41
INTRODUCTION.....	41

<i>References</i>	43
<i>Use Case Diagram</i>	44
Use case diagrams	44
Actor	45
Use Case	45
Subject	45
Graphical Representation	45
Association between Actors and Use Cases	46
Use Case Relationships	46
Include Relationship	46
Example	46
Notation	47
Example	47
Notation	47
Generalization Relationship	48
Example	48
Notation	48
Identifying Actors.....	48
Identifying Use cases.....	48
Guidelines for drawing Use Case diagrams	48
<i>Textual Use Case</i>	49
Tags	49
Alternative scenarios	49
Specific section.....	49
Step identifier	49
Advice use case	50
<i>References</i>	55
WHAT'S WRONG WITH USE CASES?.....	60
<i>Use Cases Unsupported by Domain Descriptions Are Vague</i>	61
<i>Use Cases Do Not Capture Important Information about the Problem Context</i>	63
<i>For Some Jobs, Use Cases Are Just the Wrong Tool</i>	64
<i>Nobody Really Knows What a Use Case Description Looks Like</i>	65
<i>Use Cases Are Unsystematic</i>	65
<i>Use Cases Are Not Object-Oriented</i>	66
<i>The Proper Use of Use Cases</i>	66
<i>Acknowledgements</i>	67
<i>References</i>	67
CHAPTER 7 USER STORIES	68
INTRODUCTION.....	68
BASIC TERMINOLOGY	68
<i>Software Requirement</i>	68
<i>Use Case</i>	68
<i>User Story</i>	68
<i>Epic</i>	69
<i>Story map</i>	69
<i>Technical User Story</i>	69
<i>Storyboard</i>	70
<i>User Journey</i>	70
HISTORY	71
HOW DOES IT WORK	72
<i>User Story and Storyboard</i>	72
Creation	72
How to write good user stories	75
<i>User Journey</i>	77
TEMPLATES AND EXAMPLES	78
<i>User Story User story's basic format is simple and straight forward and is typically expressed as:</i>	78
<i>Storyboard</i>	80

<i>User Journey</i>	82
PROS AND CONS	84
WHEN TO USE USER STORIES AND WHEN NOT	85
REFERENCES	87
IMAGE CREDITS	89
CHAPTER 8 MODELS	90
INTRODUCTION	90
ADVANTAGES	90
BUSINESS DOMAIN MODEL	91
BUSINESS PROCESS MODEL	91
WORKFLOW MODEL	92
DATA FLOW DIAGRAMS	93
UNIFIED MODELING LANGUAGE DIAGRAMS	95
PROTOTYPES	97
REFERENCES	98
IMAGE CREDITS	99
CHAPTER 9 REQUIREMENTS NEGOTIATION	101
THE FIVE PHASES OF NEGOTIATION	102
PHASE 2: DETERMINE YOUR BATNA	102
PHASE 3: PRESENTATION	103
PHASE 4: BARGAINING	104
PHASE 5: CLOSURE	104
NEGOTIATION STRATEGIES	105
<i>Distributive Approach</i>	105
<i>Integrative Approach</i>	105
AVOIDING COMMON MISTAKES IN NEGOTIATIONS	105
<i>Failing to Negotiate/Accepting the First Offer</i>	105
<i>Letting Your Ego Get in the Way</i>	105
<i>Having Unrealistic Expectations</i>	106
<i>Getting Overly Emotional</i>	106
<i>Letting Past Negative Outcomes Affect the Present Ones</i>	106
EXAMPLE OF UN-VERIFIABLE REQUIREMENT	107
INTRODUCTION	107
WELL WRITTEN REQUIREMENTS	108
EVALUATING DRAFT REQUIREMENTS	110
PROBLEMS AND CHALLENGES	111
MORE EXAMPLES	113
REFERENCES	114
IMAGES CREDITS	114
CHAPTER 11 REQUIREMENTS VALIDATION	115
CHALLENGE OF VALIDATION IN REQUIREMENTS ENGINEERING	115
<i>Highlights</i>	115
<i>Abstract</i>	115
<i>Introduction</i>	115
<i>What is requirements validation?</i>	116
<i>Why requirements validation?</i>	117
<i>Who validate requirements?</i>	118
<i>When validate requirements?</i>	119
<i>How to validate requirement?</i>	120
<i>Summary of approach</i>	121
<i>Conclusion</i>	123
<i>References</i>	123

CHAPTER 12 MANAGING REQUIREMENTS.....	126
WHAT IS REQUIREMENTS MANAGEMENT	126
STORING REQUIREMENTS.....	126
REQUIREMENTS TRACEABILITY	127
COLLECTING REQUIREMENTS STATUS	129
MANAGING CHANGE TO REQUIREMENTS	130
MISTAKES AND CHALLENGES IN MANAGING REQUIREMENTS.....	131
WHAT ABOUT AGILE RM.....	132
GOVERNMENT RM GUIDELINES	132
TOOLS	132
REFERENCES	134
IMAGE CREDITS:.....	135

Foreword

This book contains lecture notes for SWEN 645 Software Requirements. All of the material in this book is Creative Commons or Open Access with the exception of the Guide to the Software Engineering Body of Knowledge. It can legally be used, reproduced, printed and modified. The material was reviewed, validated and modified as needed for this course by UMGC staff.

The Guide to the Software Engineering Body of Knowledge is used under the following copyright permission.

COPYRIGHT AND REPRINT PERMISSIONS. EDUCATIONAL OR PERSONAL USE OF THIS MATERIAL IS PERMITTED WITHOUT FEE PROVIDED SUCH COPIES 1) ARE NOT MADE FOR PROFIT OR IN LIEU OF PURCHASING COPIES FOR CLASSES, AND THAT THIS NOTICE AND A FULL CITATION TO THE ORIGINAL WORK APPEAR ON THE FIRST PAGE OF THE COPY AND 2) DO NOT IMPLY IEEE ENDORSEMENT OF ANY THIRD-PARTY PRODUCTS OR SERVICES.

Michael Brown, PhD

Chapter 1 Introduction to Requirements

DR. MICHAEL BROWN, UMGC.

By this point in your education you probably have a good idea what requirements are. If you ever wrote a program, you probably read requirements to know what the program should do. Requirements are the most difficult part of whatever software development methodology you may use. It is truly the document that bridges communication between technical people and non-technical people. This is no easy task.

I am an engineer. So my opinion is that non-technical people use words incorrectly. Words like “never”, “are” and “always” mean something different to non-technical people. I once built a system for a government agency. A field in the system was called case number. The customer told the requirements team that, “Case numbers are 10 digits.” We built the system and it goes live. Then I get a call from the customer, saying that they can’t put in cases. They show me an example of a case number that starts with the letter A. I explain that the requirements say that case numbers are 10 digits and ask if that is a correct statement. They respond, “Yes, case numbers are 10 digits, unless it is a letter and 9 digits.” As an engineer I would claim that the first statement is wrong, because it is incomplete. The customer didn’t see it that way. But it is not just the definitions of words that cause issues.

The English language is ambiguous. In fact all human languages are ambiguous. Consider this sentence: *Fruit flies like grapefruit*. What does that mean? There are two ways to understand that sentence. The first is that there exists an animal called “fruit flies” and they like to eat grapefruit. The second is that there are these objects called “fruit” and they know how to fly. How they fly is similar to how grapefruit flies. For humans this example is easy to figure out, because we know that fruit cannot fly, so we eliminate that option. But we can’t always eliminate possibilities. Consider this sentence: *Be careful those chicken wings are hot*. In this sentence am I warning you that the wings have a high temperature or that they are very spicy? You don’t know.

Normally as the quantity of information on a topic increases, ambiguity decreases. As I give you more information it eliminates ambiguous options. In my example above, I would say: Fruit flies like grapefruit. In fact fruit flies can eat a grapefruit in only a few hours. But sometimes as the amount of information increases, it becomes difficult to understand. Consider this requirement.

Assume that there is an invisible grid across the screen with 100 rows and 100 columns. We number the rows 1 to 100 and the columns 1 to 100. Rows and columns are of the same size. On the screen fill in each cell of the grid that fall within a direct line between the following order pairs. In each ordered pair the first number is the row and the second number is the column. (25, 1) and (25, 100); (75, 1) and (75, 100); (1, 25) and (100, 25); (1, 75) and (100, 75).

This is a very detailed requirement. What does it do? We all see that it draws some lines, but what is the “big picture”. Now I am going to re-write the requirement a different way.

Draw 2 horizontal lines and 2 vertical lines across the screen to make 9 equal size squares, assuming that the border of the screen counts as lines.

OK, this has a lot less detail. But it makes it easier to understand. I clearly know that I am drawing 4 lines. Now I am going to re-write the requirement again.

Draw a tic-tac-toe board across the entire screen.

Which of the three is easier to understand?

These two exercises demonstrate the difficulties faced by requirements analysts. How do you make a requirement document unambiguous and easy to understand? Most of the class will focus on this.

Definition of Software Requirements

BOURKE, P., & FAIRLEY, R. E. SWEBOK V3. 0-GUIDE TO THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE.

The Software Requirements knowledge area (KA) is concerned with the elicitation, analysis, specification, and validation of software requirements as well as the management of requirements during the whole life cycle of the software product. It is widely acknowledged amongst researchers and industry practitioners that software projects are critically vulnerable when the requirements-related activities are poorly performed.

Software requirements express the needs and constraints placed on a software product that contribute to the solution of some real-world problem.

The term “requirements engineering” is widely used in the field to denote the systematic handling of requirements. For reasons of consistency, the term “engineering” will not be used in this KA other than for software engineering per se.

For the same reason, “requirements engineer,” a term which appears in some of the literature, will not be used either. Instead, the term “software engineer” or, in some specific cases, “requirements specialist” will be used, the latter where the role in question is usually performed by an individual other than a software engineer. This does not imply, however, that a software engineer could not perform the function.

A risk inherent in the proposed breakdown is that a waterfall-like process may be inferred. To guard against this, topic 2, Requirements Process, is designed to provide a high-level overview of the requirements process by setting out the resources and constraints under which the process operates and which act to configure it.

An alternate decomposition could use a product-based structure (system requirements, software requirements, prototypes, use cases, and so on). The process-based breakdown reflects the fact that the requirements process, if it is to be successful, must be considered as a process involving complex, tightly coupled activities (both sequential and concurrent), rather than as a discrete, one-off activity performed at the outset of a software development project.

The Software Requirements KA is related closely to the Software Design, Software Testing, Software Maintenance, Software Configuration Management, Software Engineering Management, Software Engineering Process, Software Engineering Models and Methods, and Software Quality KAs.

Define Requirements Engineering

WELLSANDT, S., HRIBERNIK, K. A., & THOBEN, K. D. (2014). QUALITATIVE COMPARISON OF REQUIREMENTS ELICITATION TECHNIQUES THAT ARE USED TO COLLECT FEEDBACK INFORMATION ABOUT PRODUCT USE. *PROCEDIA CIRP*, 21, 212-217, WITH MODIFICATIONS BY UMGC.

Requirements engineering (RE) is a process conducted during the early phases of product development. The ideas of RE come from the domain of software development, therefore, earlier work has a dedicated focus on software. Due to the increasing complexity of physical products, RE is relevant for the development this kind of products as well. According to Ebert, RE concerns the elicitation, documentation, analysis, evaluation, negotiation and management of requirements [8]. At the end of the whole process, stakeholder needs are codified and translated into technical specifications of the product [2]. Product specifications are used in subsequent development steps, as well as further processes like manufacturing and service.

The involvement of users in the elicitation of requirements is of major concern, with respect to the user-centered design approach. Requirements elicitation covers the systematic extraction of user requirements from different sources (e.g. the user or service documents). It consists of subsequent process steps like stakeholder and success factor identification, systematic framing, documentation, structuring, modeling and consolidation of requirements [8]. While the requirements elicitation typically marks the beginning of the product lifecycle, the inbound information for the elicitation process originates from several product lifecycle phases (e.g. manufacturing, use, service, recycling and disposal). Product use information (PUI) is the main source of user requirements with respect to user-centered design. PUI is typically conveyed by retrospective user feedback from the actual use phase of the product. According to Abramovici et.al, feedback related to PUI can be subjective or objective [9]. While retrospective user feedback is subjective, sporadically acquired data from sensors and service personnel is more of an objective kind. In order to collect feedback information to elicit requirements, different techniques in many variants can be applied [10].

References

- [2] Jiao JR and Chen CH. Customer requirement management in product development: a review of research issues. *Concurr Eng*; vol. 14, no. 3, 2006. p. 173–185.
- [8] Ebert C. *Systematisches Requirements Engineering*. 4th ed. dpunkt.verlag Heidelberg; 2012.
- [9] Abramovici M, Lindner A, Walde F, Fathi M, and Dienst S. Decision support for improving the design of hydraulic systems by leading feedback into product development. In: *Proceedings of the 18th International Conference on Engineering Design (ICED11)*. vol. 9, 2011. p. 1–10.

[10] Davis A, Dieste O, Hickey A, Juristo N, and Moreno AM. Effectiveness of Requirements Elicitation Techniques: Empirical Results Derived from a Systematic Review. In: Requirements Engineering, 14th IEEE International Conference; 2006. p. 179–188.

On the Intersection between Requirements, Business, and the Law

SHELDON LINKER, PH.D. — UMGC FACULTY

The small print: The framed parchments here say "Computer Science" and "Software Engineering". They say nothing about being a lawyer, nor anything about having any sort of a business degree. Yet, I find myself doing business and writing and evaluating contracts. Just the same, don't take anything here as financial or legal advice. Should the need arise, seek pro guidance from your financial advisor, lawyer, and/or shaman. 8-)

When I left school, I got a job. I thought that I was going to be a programmer. Three weeks into the job, I was told that I was now a data base consultant. When I arrived at the project to which I was assigned, I was told that the project was in trouble, and that I was to get the data base "handled". I quickly found that the data base requirements were unknown, and that the nature of the interface between the programs and the data base were also not well defined. That was one of the main reasons that the project was in trouble. That, in turn, caused contractual issues, which in turn was on the verge of causing legal issues.

Every business is much like a living organism. Organisms want to eat and grow. Businesses want to earn money and grow. Even nonprofits want to earn money, even if the money coming in is used for charitable purposes, or as a cash reserve. The programming (and the requirements gathering which elicits the programming) take place so that the business can better its bottom line. Depending on the nature of that business, bettering the bottom line usually means one or more of spending less in the long run, earning and/or producing more in the long run, and/or becoming more effective. Your job as a programmer or analyst will seldom directly be producing "cool" stuff; it will be the job of producing the needed stuff. If that stuff is cool, so much the better. If what you're producing doesn't affect the bottom line as described above, then it's useless or worse.

When you produce software for internal use, you can use the Waterfall model, in which you have charter, then requirements, then design, then construction, then unit testing, then integration, then integration testing, and then final documentation, and then you're done. Another alternative is Agile, in which these things start almost at the same time, and advance together. There is also Spiral, my favorite, in which you have an Agile-like conglomeration of little Waterfall-like versions. Note that I'm not telling you that Spiral is best — just that it's my favorite. When the software is producing something customer-facing, or for use in a rapidly-changing environment, you're pretty-much relegated solely to Agile, or you'll never match the current circumstance. When your software is for an external customer, there's going to be a contract. That contract will be either constructed on a time-and-expenses basis, or on a fixed-price basis.

Time-and-expense contracts are very simple, generally. Such contracts are basically a statement that I'm going to charge you so much an hour, plus any direct expenses. There are parameters on how much I can spend, and the time period. For instance, with one customer, the contract was for 80 man-hours per week for 5 months, at a given price per such hour. In such contracts, the customer usually has the right to terminate the work with no notice. Here, it's not the contract that controls what's going on. Instead, it's the customer's perception of how well you're doing for them. Show them good stuff early and often, and you'll keep getting the money. Succeed in the project (whatever that means to the customer), and you're likely to get repeat business. That's where the best profits come from.

Fixed-price contracts, on the other hand, almost always force you into the Waterfall model. That's because in the fixed price model, you're going to get a specific amount of money for doing a specific thing. I most strongly suggest that in such a contractual arrangement, that the specifications must be complete before entering into the fixed price contract. Here's why: Let's say I offer you \$30,000 for doing a job for me. If I want you to do \$30,000 worth of work for that, everyone will be happy. If it turns out that it's only \$30 worth of work, the customer will not be happy, and the contract might or might not be enforceable, since there's a doctrine of fairness. If there's \$30,000,000 worth of work to do, you're going to go bankrupt and/or get sued or be the one who sues. Don't get yourself into that. If the customer wants a fixed-price contract, then the customer needs to deliver the specifications as to what is needed, or the requirements gathering process needs to be a separate time-and-expenses contract.

When combining time-and-expense specification writing with fixed-price systems production, keep the two contracts separate. Once the specifications are done, you and the customer can negotiate a price, or the customer may go elsewhere for that phase. That's not always a bad thing. For instance, we produced a specification. We then bid a price. The customer didn't like the price and went elsewhere. That was fine, since we could not have delivered the specified product for the price that the customer and the third-party provided decided on. (As it turns out, they couldn't produce the product for the price they quoted, either.)

A use-case style specification is usually fine for time-and-expense projects. Both sides know generally what's going to happen, and things pretty-much go in that direction. If minor or even major mid-course corrections are needed, then they can be made within the wiggle room of the use case or user story. The cases and stories can be adjusted as needed, too. Fixed-price contracts are best served by IEEE-style specifications. The more everything is nailed down, the better everything will go for both sides. Very often, a fixed price contract will either directly include the specification document, or include it by reference. Thus, the specification is, in effect, a contract in and of itself. Thus, the specifications should be written in contract-like language.

Before I go in to the language of contracts, let me explain the concept of Offer, Acceptance, Performance, and Payment. In this concept, all contractual arrangements involve these four phases, which are not necessarily in this order. Typically, you or the customer makes the other an offer. The other accepts the offer. Then, you perform the contractual function, and finally they pay you. In some arrangements, payment occurs before performance, during performance, or some before and some after, or what have you. The presence of any three of these components is sufficient to prove or require the fourth. For instance, acceptance, performance and payment is sufficient to prove offer. Offer, performance and payment is sufficient to prove

or fully imply acceptance. Offer, acceptance and payment is sufficient to force or require performance. Offer, acceptance and performance is sufficient to require payment.

The language of contracts and specifications must be as clear, concise and unambiguous as possible. Define everything, especially abbreviations. Generally, an abbreviation or defined term will generally be of a form in which you use a term in its full form, and then the capitalized abbreviation in parentheses and quotes. For instance:

This agreement is by and between The First Fake Bank of the West ("Lender") and John Q. Smith ("Borrower"). Lender intends to...

The exact same sort of thing is useful in specifications. but without the quotes. For instance:

The data base management system (DBMS) will communicate with the workstations using the local area network (LAN). The DBMS shall...

Note that the language of the contract is important, and that translations don't always work well. If a language or a specification is in more than one language, or used in translation, make sure that one of the versions is the reference version or controlling version. Despite all of its ambiguity, English is one of the least ambiguous languages. Below, I'll cover English only.

Many people confuse the meanings, or at least usages certain key words. Here is a table that may help:

Word	Meaning	Problems to avoid
Can	The event can occur. There's nothing that the development team can do to prevent the event, but the team can cope with the consequences of the event.	If you use "can" when you mean "shall", then expect to hear "Since the user could already do that, why would we bother implementing it?" If you use "can" when you mean "may", expect to hear "Well sure, the user can do that, but it's not allowed."
Does	You are stating that the event already occurs.	If you use "does" (or verbs ending in "s") when you mean "shall", expect to hear "It already did that! Why would we implement it again?"

Word	Meaning	Problems to avoid
May	The event is allowed, or should be allowed. If you use "may" when you mean "shall allow for", then expect to hear "We were allowed to do that, but we didn't have to."	
Shall (or Must)	This is a direct order. Your development team must do this. The testers must test for this.	If you use "shall" when you mean something else, the most common result is that there is confusion, but there can still be contractual problems. For instance, "The user shall log off when done" has two problems. First, the user <u>may</u> log off when done, so you probably wanted to say that. Second, the user is not reading the specification document, so no edict on the user in the document will even be known to the user.
Should	This has the same weight as "it would be nice if".	This has zero contractual weight.
Will	You are stating that something will happen with or without anything provided by your development team or product.	Use "will" when you meant "shall", and expect to hear "If it was going to happen anyways, why should we worry about it?"

One more general rule in writing contracts and specifications: In most jurisdictions, if one party writes the contract or the specifications, then the other party is entitled to any reasonable interpretation of the contract or specification. A common trap is that someone will write a contract or spec, and then ask you to sign that the contract or specification is a "common work product". Signing such is likely to deprive you of your entitlement to a reasonable interpretation.

Other Things You Need to Know

SHELDON LINKER, PH.D. — UMGC FACULTY

Most of the recommendations presented here are based on my experience and are thus my opinion. Don't take what I have to say here as gospel, but at the same time, don't discard what I have to say without consideration if your opinion differs. I welcome discussion on any of the issues presented here. My recommendations appear in italics.

This book covers requirements gathering and documentation, and to an extent, requirements tracking. However, to be an excellent practitioner of this art, it would be most helpful to have some additional domain-specific knowledge. Requirements engineering is a very broad field. The concept applies not only to software engineering, but to all other forms of engineering, and to most forms of businesses. For instance, if I asked you to set up a food establishment for me, as after successful completion of this course, you'd be better at it than someone off the street, because you'll have a better idea of what sort of questions to ask. But, back to software engineering: The following is a list of some things that would be very helpful to know before visiting with your first customer or stakeholder.

I Language classes

- 1 **Shells:** System-level command programs where you issue commands to run programs. Includes Windows BAT files, unit shells, and IBM's JCL.
- 2 **Programmable shells:** Shells with some programming capability, such as a form of IF. *When using shells, use programmable shells, and make sure to check for error conditions at each stage of the processing.*
- 3 **In-betweeners:** Shells that are so programmable as to seem more like programming languages, or programming languages often used to start other programs. *If efficiency is an issue, and it usually is, even if you don't plan it that way, plan on using a compiled language.*
- 4 **JavaScript:** The preeminent language used to program front-end web pages. (Can also be executed at the back end, but seldom used there.) *Don't use JavaScript at the back end. Do use it at the front end, but not when static HTML or CSS would do the job.*
- 5 **Loose/tight languages:** Loose languages allow item type to be changed at run time. Tight languages do not, and thus provide more checking capability. *Using loose languages will get you running faster, so are nice for prototypes, but tight languages will decrease your overall costs.*
- 6 **SQLs**
 - A **Brands & loyalties:** You will find that some people have brand loyalty. *Don't argue with them. Others do not. For those without brand loyalty, if they have an SQL in use already, stick with it. If not: SQL Server is the fastest at ad hoc SQL. Oracle stores data in the most compact form. DB/2 is the fastest at stored queries.*
 - B **Tables:** Data is stored in tables, in rows & columns. There are 0 or more rows that come and go, but 1 or more columns, which are static for a table.
 - C **Queries:** A query (Select command) is how we get data from tables, as is, or processed/correlated in some manner. Queries may be stored in the database as views. *Better to use a stored view than to send a query from your program.*
 - D **Joins:** Joins are how programs link 2 or more tables together.

- i **Inner Join:** A join from table A to table B in which only matched items are used.
 - ii **Left Join:** A join from table A to table B in which the rows from table A are used even if no corresponding B row is present.
 - iii **Outer Join:** A join from table A to table B in which the rows from tables A and B are used, even if no corresponding B or A row is present.
 - iv **Exception Join:** A join from table A to table B in which the rows from table A are used, except where there is a matching row in table B.
- E **Indices:** An index is a way to vastly speed up reading at the cost of slower writes and more storage. Indices can be regular or enforce uniqueness. *Proper index design is the key to fast execution in SQL systems. A missing index will cost you time, as will an extra index.*
- F **Insert/Update/Delete:** These commands add rows, change values within rows, and delete rows, each affecting 0 or more rows, depending on conditions.
- G **Stored procedures:** Stored procedures are SQL programs stored on the server. *Whenever possible, move logic involving multiple SQL commands into a stored procedure.*
- H **Types:** Within a program, data will have a type. Not all SQLs have all of these types, and more types are possible.
 - i **Boolean:** Stores true/false values. *In most SQLs, use CHAR(1) with a value of 'Y' or 'N'.*
 - ii **Numbers:** Can be integers, decimal, or floating point.
 - iii **Strings:** Can be fixed width, variable width with a maximum length, or of unlimited size (but slow performance).
 - iv **Dates & times:** DATE specifies a day, DATETIME second, and TIMESTAMP specifies a millisecond. In Oracle DATE is actually DATETIME.
 - v **Selectors:** A field that specifies a value in another table, usually actually stored as a numeric type with a FOREIGN KEY constraint.
 - vi **Nulls:** The "no value" value.
- I **Normalization:** *Designers (that's you, to an extent) get into trouble with unnormalized databases. (But sometimes, you need to violate the following rules for speed.)*
 - i **1:1 relationship:** *Don't store two kinds of things in one table, and don't store one kind of thing in two tables.*
 - ii **Move common data:** *If the same kind of multipart data is stored in 2 or more places, consider moving that data to a table specifically for it.*

- iii **Don't store derivable data:** *Don't store data you can figure out, unless it takes too long to figure it out.*
- J **Constraints:** The database can do run-time checking for you. *Take advantage of every opportunity to add constraints. They take a little time, but protect the data.*
 - i **Not Null:** Prevents a column's value from being NULL in a row.
 - ii **Check:** States a fact that must be true of each row, as a formula.
 - iii **Unique:** States that the value in this column or these columns must be unique within the table.
 - iv **Primary Key:** States that this value for this column or these columns are both Unique and Not Null.
 - v **Foreign Key:** States that this column (or these columns) must (if not null) refer to the primary key of a given table.
- K **Program interfaces:** How people and programs interface to SQL
 - i **Command line:** You can type commands at SQL. *Useful for creating the database.*
 - ii **ODBC & JDBC:** The program communicates with the database via subroutine calls. This is the most common interface.
 - iii **Language+:** A means used by Oracle and IBM in which SQL is embedded directly into the calling program, without subroutine call syntax, also called EXEC SQL. *This can be a big time-saver, as it allows compile-time SQL checking.*
 - iv **Stored procedures:** *As mentioned above, move SQL logic to the database server whenever possible.*
- 7 **Compiled languages vs. not:** Some languages compile to native code. Some compile to pseudocodes (such as Java and .Net), and others are interpreted. Native code gives the fastest execution, but self-cleaning pseudocoded languages are easier to write in. *For most business applications, self-cleaning languages are the most cost-effective, but when you need the fastest possible execution, native code is the way to go.*
- 8 **Object orientation:** A style of programming in which classes of data are assigned behaviors, as opposed to procedural programming, in which things are done to data. *Each can do everything the other does.* Inheritance: In object oriented programming, one class of objects can be built on another, inheriting its data and behaviors, but overriding some. This forms "class heirarchies", which are sometimes part of the requirements documentation. *Such class design (and even whether to use object orientation or to use procedural programming) belongs in the design documentation, and not in the requirements.*
- 9 **Importance of interfaces:** Many things are called "interfaces"...

- A **User interfaces:** The user interface, or UI or GUI is the form the user interface takes. This might be a windowed interface, something entirely graphical, such as a game, the ol' 24x80 green screen, or something embedded, such as a steering wheel and peddles.
 - B **Internal program interfaces:** Internal application program interfaces (APIs) are the interfaces between components of a system. *If components of the system are to be built by 2 or more teams, then the entirety of the API had better be a part of the requirements document.*
 - C **Storage:** Programs will read and write data to external storage, such as disk files or SQL data. *If data is to be shared by components produces by 2 or more teams, then the entirety of the data structure has better be a part of the requirements document.*
 - D **Things called "interfaces" in object languages:** In object languages, an "interface" is a definition of a set of methods (procedures and function calls) that a class promises to implement.
- 10 **Importance of sameness:** *You need to implement consistency in these areas:*
- A **User interfaces:** *Consistency of user interface is important to the learning curve of the user. Maintain consistency, except where you need to deviate. The best book on this, regardless of your platform, is Inside the Mac, Volume 1.*
 - B **Coding style:** *Consistent coding style makes the program easier to understand, and thus cheaper to test and maintain. Coding standards should be in place for a project, but should be at the level of suggestion, rather than requirement.*
 - C **Language class:** *At any given tier, there should be one language in use, or at least all languages should be of the same class. For instance, using C# and Visual Basic.Net together works well. PL/I and Cobol do. C# and C++ don't.*
- 11 **Frameworks/inversion:** Inversion is where we take control outside of a program, and move it to some outer, containing framework. *A little of this, such as you find in redirection in Unix and data definition in JCL is helpful. To much, and your program becomes unfollowable.*
- 12 **Aspects:** There is something called aspect-oriented programming, in which you divide classes into aspects. Each aspect does a different type of thing. *100% of the time, this is a waste of time and effort.*
- II **Tiers:** Programs are often divided up into 2 or more of the following tiers, which typically run on separate computers: Database, Business, Presentation, Web service, and Client.
- III **Solution-oriented architecture (SOA):** Different servers handle different parts of the problem. *This is a good idea if 2 separate companies are involved. This is a bad idea if not.*
- IV **Email:** Servers can send and receive eMail and texts. EMail becomes insecure the moment it leaves the building. EMail is secure if both users are on the same server.

- V **Web Languages:** There are a number of languages used in constructing a web site:
- 1 **HTML** to hold the structure and content at the client (browser) tier.
 - 2 **CSS** to hold the styling controls at the client tier.
 - 3 **JavaScript** to control programmability and behavior at the client tier.
 - 4 **JSON** to hold data as it moves between JavaScript control and the server tier.
 - 5 **Back-end languages** (hundreds of them) to control what happens at the server tier and above.
- VI **Ways:** There are a lot of ways to get things done:
- 1 **SQL:** SQL is not the only way to store data. Consider also indexed sequential (ISAM) files, random-access/direct access files, and sequential files.
 - 2 **OS:** Operating systems can have a large influence on how things are done. OSes fall into a number of categories:
 - A **Unix/Linux:** Unix is commercial; Linux is the freeware equivalent. Both run on everything. Properly written Unix/Linux programs are fully hardware portable.
 - B **Windows:** The server version of windows doesn't need a keyboard or screen. Windows is not portable. Serving more users costs more than serving fewer users. There's a lot of proprietary software here.
 - C **IBM:** These are mainly mainframe OSes. They support big-iron solutions in which very powerful machines with very wide data paths are thrown at a problem. They include batch programs which will finish their task or restart themselves even after an unplanned reboot.
 - D **Embedded real-time operating systems (RTOS):** Real-time operating systems can execute tasks at exactly the right time. Embedded systems are those that have no user interface that's obviously part of a computer.
 - 3 **Stacks:** A "stack" in this context is a collection of programs (operating system, web server, web languages, and specific programs) that "sit on top of each other", such as the Windows stack (Windows, Internet Information Server, and ASPX or some other server web language, and Microsoft SQL Server), LAMP (Linux, the Apache web server, the MySQL database, and PHP), and Java web service (Linux/Unix/zOS, Apache, a database, Java, and likely Java Server Pages).
- VII **Security:** Security falls on a spectrum. Some programs don't need any, like games. Other products need a lot, like military and banking. Know the required security level, and the probable threat level.
- 1 **Outside:** How secure your software is from all types of outside attacks.
 - 2 **Inside:** This is the level of security needed and used inside the building, from none through just as much as Outside security.

- 3 **Database:** The database may need to be secured, such as internal permission levels and encrypted columns. For instance, passwords may be stored with one-way encryption, so that they can be compared encoded, but can never be decoded.
- 4 **Code at the user tier:** Once code is delivered to the user tier, it's in the user's control. *Never trust code running at the user tier, such as the JavaScript you sent.*
- 5 **Penetration:** Attacks in which there is an attempt at some sort of "penetration" of your system, or the insertion of something into it.

VIII Nice ways in & out: Consider these:

- 1 **Input validation:** *Whenever a field is entered, the program should reject invalid types of characters. For instance, in a numeric field, ignore letters. Always check every field for valid input.*
- 2 **Crystal Reports™:** Lets you set up a report format, and have that report filled from server data, using parameters. *Best practice: Separate the report from the view or procedure that delivers the data.*
- 3 **Kofax™:** A scanning/character recognition system which can scan documents, determine, their type, do validation, and put data into the database.

IX **Scaling:** Scalable systems allow your system to continue to run once its usage grows.

- 1 **The Microsoft way:** Sends your maintained data to the user's tier, encrypted, so that the user can't modify it. The user's system can reconnect to any of a number of servers, delivering it accurate knowledge of what the state and data is. *Plus: Virtually unlimited PC scaling. Minus: Extra data transmission.* This is called Horizontal scaling.
- 2 **The PHP way:** Machines are told which machines to reconnect to. *Plus: You can connect a lot of cheap machines quickly. Minus: It's hard for the systems to coordinate with each other, so the problem may have to be broken down.* This is also Horizontal scaling.
- 3 **The Google way:** Makes copies of the data to each server, so that no one machine needs to communicate with an other for most circumstances. This is also Horizontal scaling.
- 4 **SOA:** As discussed above.
- 5 **The Unix/Linux way:** You can write your application for small machines, and run it on the largest systems built. *Plus: Cheap & easy at first. Any level of power may be applied to a problem, a little at a time, without any coding to account for Horizontal scaling. Minus: Requires replacement of machines. You don't necessarily take full advantage of any special hardware present.* This is called Vertical scaling.
- 6 **The IBM way:** Choose a big-iron machine, and take advantage of all of its special features from the get-go. *Plus: It's fast to get going, and there is no accounting for the possibility of Horizontal scaling. Minus: It's expensive, and you've got unused capacity at first.*

Chapter 2 Requirements Process

BOURKE, P., & FAIRLEY, R. E. SWEBOK V3. 0-GUIDE TO THE SOFTWARE ENGINEERING BODY OF KNOWLEDGE.

Process Models

The objective of this topic is to provide an understanding that the requirements process

- Is not a discrete front-end activity of the software life cycle, but rather a process initiated at the beginning of a project and continuing to be refined throughout the life cycle
- Identifies software requirements as configuration items, and manages them using the same software configuration management practices as other products of the software life cycle processes
- Needs to be adapted to the organization and project context

In particular, the topic is concerned with how the activities of elicitation, analysis, specification, and validation are configured for different types of projects and constraints. The topic also includes activities that provide input into the requirements process, such as marketing and feasibility studies.

Process Actors

This topic introduces the roles of the people who participate in the requirements process. This process is fundamentally interdisciplinary, and the requirements specialist needs to mediate between the domain of the stakeholder and that of software engineering. There are often many people involved besides the requirements specialist, each of whom has a stake in the software. The stakeholders will vary across projects, but always include users/operators and customers (who need not be the same).

Typical examples of software stakeholders include (but are not restricted to)

- Users: This group comprises those who will operate the software. It is often a heterogeneous group comprising people with different roles and requirements.
- Customers: This group comprises those who have commissioned the software or who represent the software's target market.
- Market analysts: A mass-market product will not have a commissioning customer, so marketing people are often needed to establish what the market needs and to act as proxy customers.
- Regulators: Many application domains such as banking and public transport are regulated. Software in these domains must comply with the requirements of the regulatory authorities.
- Software engineers: These individuals have a legitimate interest in profiting from developing the software by, for example, reusing components in other products. If, in this scenario, a customer of a particular product has specific requirements which compromise the potential for component reuse, the software engineers must carefully weigh their own stake against those of the customer. Specific requirements, particularly constraints, may have major impact on project cost or delivery because they either fit well or poorly with

the skill set of the engineers. Important tradeoffs among such requirements should be identified.

It will not be possible to perfectly satisfy the requirements of every stakeholder, and it is the software engineer's job to negotiate trade-offs which are both acceptable to the principal stakeholders and within budgetary, technical, regulatory, and other constraints. A prerequisite for this is that all the stakeholders be identified, the nature of their “stake” analyzed, and their requirements elicited.

Process Support and Management

This topic introduces the project management resources required and consumed by the requirements process. It establishes the context for the first subarea (Initiation and scope definition) of the Software Engineering Management KA. Its principal purpose is to make the link between the process activities identified and the issues of cost, human resources, training, and tools.

Process Quality and Improvement

This topic is concerned with the assessment of the quality and improvement of the requirements process. Its purpose is to emphasize the key role the requirements process plays in terms of the cost and timeliness of a software product, and of the customer's satisfaction with it. It will help to orient the requirements process with quality standards and process improvement models for software and systems. Process quality and improvement is closely related to both the Software Quality KA and the Software Engineering Process KA. Of particular interest are issues of software quality attributes and measurement, and software process definition. This topic covers

- Requirements process coverage by process improvement standards and models
- Requirements process measures and benchmarking
- Improvement planning and implementation
- security/CIA (Confidentiality, Integrity and Availability) improvement/planning and implementation.

REQUIREMENTS FOR DIFFERENT SOFTWARE DEVELOPMENT METHODS

DR. MICHAEL BROWN, UMGC.

There are many different types of software development methods. Here is a quick review of some of the more common ones. Notice how requirement activities take place at different times for each method.

LODHI, N. K. & DALAL, P. (2014). SOFTWARE DEVELOPMENT PROCESS AND METHODOLOGIES: A REVIEW. INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY, VOL 3, ISS 11.

Waterfall

Waterfall model was proposed by Winston Royce in 1970. It is linear, sequential and conventional model. It follows requirement gathering, design, coding, testing, delivery and maintenance in sequential manner thus no phase can be started before previous phase has been completed and there is no way to go in previous phase in order to make changes during the

development. Advantages: Waterfall model is useful where all the requirements can be gathered initially. It has well defined output after each phase. It is easy and simple to understand. Disadvantages: It takes long time due to sequential manner, no early prototypes, very rigid. Idealized hence do not match with reality.

Spiral Model

In order to remove causes of failure of waterfall model many methodologies have been developed that are based on the iterative development. Spiral model combines the iterative and incremental approach in order to offer opportunities to make changes or add new requirements. It focuses on risk analysis during each iteration/spiral. Advantages: Risk analysis. It is most suited for complex safety critical system, and early visibility of prototypes. Disadvantages: Expensive, consumes long time duration, complex to understand and implementation, requires risk analysts, does not work well for small, less risky projects.

Agile

Agile process follows several phases such as requirements gathering, analysis, design, coding, testing and delivery of partially implemented product and waits for customer feedback. These phases carried out continuously until customer gets satisfied. Customer satisfaction is the highly prioritized feature with short development time.

Scrum Model

Scrum methodology was introduced by Ken Swaber in 1995. Scrum methodology is based on the sprints, daily scrum meetings and sprint planning. Sprints are small functionality that is developed within 30 days. This duration never extends hence customer gets satisfied due to on time delivery. Scrum model resolves the problems and complexities of each team members in daily scrum meeting. Advantages: Short time duration, daily scrum meeting, user involvement, review sprints, focuses on team communication and can work with any technology. Disadvantages: Works well only with small sized application, requires collocated team members, employs skilled and experienced team.

Chapter 3 Requirements Elicitation

Definition

DURÁN TORO, A., BERNÁRDEZ JIMÉNEZ, B., RUIZ CORTÉS, A., & TORO BONILLA, M. (1999). A REQUIREMENTS ELICITATION APPROACH BASED IN TEMPLATES AND PATTERNS. WER, 1ST, 1999, BUENOS AIRES, ARGENTINA.

Following [15], requirements elicitation can be defined as the process through which customers and users of a software system discover, reveal, articulate, and understand their requirements.

References

[15] S. Raghavan, G. Zelesnik, and G. Ford. Lecture Notes on Requirements Elicitation. Educational Materials CMU/SEI-94-EM-10, Software Engineering Institute, Carnegie Mellon University, 1994.

FERNANDES, J., DUARTE, D., RIBEIRO, C., FARINHA, C., PEREIRA, J. M., & DA SILVA, M. M. (2012). ITHINK: A GAME-BASED APPROACH TOWARDS IMPROVING COLLABORATION AND PARTICIPATION IN REQUIREMENT ELICITATION. *PROCEDIA COMPUTER SCIENCE*, 15, 66-77.

Traditional Methods

Based on communication, the social nature of the Requirements Elicitation activity is undeniable [3]. As such, recent trends of investigation have been using methods derived from social sciences in order to increase chances of success of requirements elicitation [15]. Such methods include ethnography, interviews and group work.

Ethnography focuses the observation of people in their natural environment, translating stakeholders activities and interactions [16, 17, 15]. Although some researchers claim that ethnography may have satisfactory results eliciting [17], several limitations are recognized. These limitations include risk of incorrect interpretations, impossibility of identifying new requirements or difficulty of generalizing results [16].

Interviewing is an informal interaction where analysts explore needs asking stakeholders about the system in use and the system to be [15]. Several researchers have been studying the nature of conversations and interviews to progress their efficiency [14, 3]. Despite the improvements that their research results demonstrated, they admit that more research is needed [14]. Moreover, well known limitations are advanced, such as the limited stimulus response interaction and the need of participants to share basic concepts and methods [3].

Group work gathers stakeholders to collaborate reaching solutions about an identified problematic situation [15]. Although practice with methods such as JAD [18, 19], focus groups [20, 21] or creativity workshops has proven pleasing results, several limitations are known. Typical limitations include dominant participants, biased opinions, high logistic costs and difficulties on gathering stakeholders at the same time and place [15].

Web-Based Methods

Recognizing the importance of collaborative work and the huge difficulty of gathering stakeholders at the same time and place, researches have been proposing web collaborative tools to elicit requirements [22]. Such tools include variations of the WinWin spiral model [23], Athena, variations of wikis, iRequire, AnnotatePro or Stakesource [24].

For example, the CoREA method (Collaborative Requirements Elicitation and Analysis), based on the winwin spiral model, is a geographically distributed environment. It includes decision support for analyzing and selecting requirements. Nevertheless, this method was not empirically evaluated although their authors have initially planned it [25].

Athena is a collaborative approach supported by a tool to elicit requirements based on group storytelling. The stories are merged in a single story, transformed into scenarios and translated into use cases. Although Athena eased the asynchronous interactions between participants, it has several limitations. These limitations include a difficult usability; an inaccurate view since stakeholders do not construct a single view together; or time consumed when compared to interviews or a group dynamics approach [26].

Wikis were also widely studied to deal with distributed stakeholders, easing communication and increasing the participation of all stakeholders. This collaborative tool allows spatially distributed stakeholders to add, remove, and amend content on a common platform. There are several proposals based on wikis, such as WikiWinWin [27], SoftWiki [28], SmartWiki [29] or ShyWiki [30]. Although wikis proved to ease distributed collaboration, they lack the means to discuss conflicts among stakeholders. This limitation may origin misunderstandings about requirements elicited by stakeholders with different work practices and responsibilities [31].

AnnotatePro [32] obtains requirements by drawing annotations directly on the users screen and using snapshots in combination with ordinary picture editing functionality. The snapshots may easily be sent to the software engineers by email. Although easing involvement of stakeholders, this tool does not contain a method following a well-structured plan, does not provides a formal notation language and does not allow tracking own submitted requirements.

iRequire [33] is a tool for mobile phones and enables users to blog their requirements whenever their need is triggered. The main features of iRequire are the possibility to take a picture of the environment, document a user need, describe the main task and provide a rationale, and check the summary of a need. However, it does not support brainstorming of needs; does not document well-defined requirements; and the authors recognize that utility and usability studies are needed to improve the tool.

Stakesource 2.0 [24] is a web application using standard technologies that uses social networks and collaborative filtering, a crowdsourcing approach, to identify and prioritize stakeholders and their requirements. Stakeholders can invite other stakeholders to participate, suggest and rate requirements. However, this tool was not completely evaluated in real-world projects.

On Getting the Information from the Stakeholders

In the endeavor of "doing requirements", there are two very important main stages: Gathering the requirements, and documenting them. Documenting the requirements, once gathered, is a relatively straightforward process. Gathering them in the first place is the sticky bit. Below, I list a number of things to consider in gathering the requirements.

Do they know they're stakeholders? You have no doubt heard that you will be getting input from the stakeholders. When the stakeholders are external, there is no problem. The customers have come to you. There are only two stakeholders involved — the customer and the service provider or vendor. However, if the project is internal, then determining who stakeholders are is not as easy as you'd like. There are major stakeholders, minor stakeholders, people who are clearly not stakeholders, and pretenders to a stakehold. In many cases, people who are major stakeholders will know that they are, but some of them may think that they're minor stakeholders. Many minor stakeholders think that they're major stakeholders think that they're major stakeholders, or at least want to be treated as if they are. Some think that they are not stakeholders. Some people want their opinion heard, even if they have no connection to the project. Don't take their demands seriously, but at the same time, don't let a good suggestion go unheard or unheeded. One way to deal with uncertainty in the situation is to get an organizational char (often called an "org chart") and to see how everyone fits in.

Do they care? Not all of your stakeholders will care about your project, or will want to take the time to deal with the project. In some cases, you need to get management to schedule time, so that the reluctant stakeholders don't have the excuse of needing to do something else when you want to talk to them.

What they need, want, and ask for: An unfortunate truth is that what the customer or stakeholder wants, what they ask for, and what they need, are not always the same thing. They will often ask for something, but what they ask for is not what they really want, and not often not what they need. Your job is to sort through what they need. If you can't figure out or deliver what they need (sometimes they won't let you), then go with what they want. If you can't figure out or deliver what they want (sometimes they insist on what they ask for), then you just have to go with that. Note that what you yourself would like for them does not appear on the list at all.

The overuse of Excel: One of the big problems you will see out in the wild is that people are doing things in Excel. There are a few things that Excel excels at, but the list is very limited. You will find people using Excel files (with extensions XLS, XLSX, TSV, and CSV) as database files, as computer-to-computer interface files, as report generators, as a data entry system, and who knows what else. A very common situation is that on computer system A, person B asks for a report C in Excel format, producing file D. B then sends D to person E, who, on computer system F, will run import function G to load the data. Whenever you find this sort of thing happening, write a function on A and one on F, so that the data is automatically moved for them. Another common situation is that data is output to an Excel file, then manipulated by a person, and then the desired report is produced. In such circumstances, have the system produce the desired report in the first place. There are a number of nice reporting systems, including those built into SQL Server and Oracle, as well as Crystal Reports.

System hopping: As alluded to above, many times data is moved from one system to another. Make sure that the systems are tied together in the proper manner. For instance, should the

database be shared? Should messages be sent automatically? On demand? Should the data be shared at all? Make sure that all the data you need is sent or shared, and that none of the data you don't need is sent or shared.

Good/Fast/Cheap: There is a concept called "the engineering triangle". As the sides of the triangle, we have Good, Fast, and Cheap. The points on the triangle are Good-and-fast, Fast-and-cheap, and Cheap-and-good. The project goal point can be anywhere in that triangle. Generally, you'll be shooting for one of the points. Unfortunately, you can't usually have all three. Something has to be left out. For instance, VW bugs are good and cheap, but they're not fast. Bugatis are good and fast, but they're not cheap. A Fiat X-19 is fast and cheap, but it's not good. If you don't understand which of these two items you're shooting for, you can't win. This also ties in with project management methodologies. Given that we're going to be Good, that leaves us with the option of Fast or Cheap. The Waterfall methodology is actually cheaper than Agile, because everything is done only once (if you get the requirements right). The Agile method is faster than Waterfall, in that some of the project gets done right away, most of the project gets done quickly, and if done well, the projects will end at the same time that Waterfall would have. However, this occurs at a higher price because of reworks, retesting, and higher staffing levels.

What's better? This seems like a simple question, and actually it is, but it's one that has to be asked. There will be goal options. You need to examine all the goal options, and decide which one you're shooting for. Much like the vehicle considerations above, you can't build the "best" vehicle, or "best" any thing else. What's better, a moving van, a sportscar, or a small boat? Depends on what you plan on doing with it.

Looks good or Works well: One of the big goal option question applies to the user interface (if your project is going to have one at all). Generally, user interfaces fall into two major categories: Interfaces that look nice/cool, and those that are fast/accurate. If you're going to build a customer-facing web-site, then easy navigation, coolness, friendliness, and upselling are all important factors. If you're building an interface for internal use, none of those factors are necessarily important. What's important is the speed and accuracy. For internal systems, try for an interface in which the data entry operators need never lift their hands from the keyboard (minimize mouse usage).

Friendly or demanding: Similarly, know whether the interface should be friendly or demanding. Generally, for external-facing items, friendly is best, but demanding can be somewhat needed, too. For internal, the opposite is true.

Consider English: On one system, I had "inherited" a very busy user interface. There were many more options than the users wanted. It got to the point where I needed a second tab on the screen to hold the additional fields, checkboxes, and pull-downs. Then I decided to simplify the interface in a novel way. I ended up with a single multiline text field, labeled "What do you need?". For instance, the users could enter "I need a provider list covering California, and the Las Vegas Metro area. Include general dentists and periodontists. Include handicapped symbols and languages spoken."

Work there — My training as a riveter: I got a job doing Y2K remediation at an aircraft manufacturing plant. Before they had us do any work, they trained us in riveting. What's the point of having a programmer do riveting? Simple: You need to be able to "walk a mile in

someone else's moccasins". You may need to learn to do some or all of the job functions. You may even have to spend some time doing each of those job functions.

SMEs: Projects use subject matter experts (SMEs). If they (your company or your customer) have or has SMEs, use them to whatever extent you need to and can. If they don't have an SME, then you must become the SME.

Chapter 4 Requirements Documentation

McFADDEN, R. (2017). SOFTWARE REQUIREMENTS – REQUIREMENTS DOCUMENTATION, UNDER CONTRACT FROM UMGC.

Reasons to document requirements

According to McEwen (2004), Standish Group Survey on more than 352 companies reporting on more than 8,000 software projects found that 31% of projects are cancelled (cost \$81 billion) and 53% of projects cost 189% more than originally estimated while for large companies only 9% complete on time at budget and for small companies only 16% are on time and budget. The three top reasons are related to poor requirements: (1) lack of user input (12.8% responses), (2) incomplete requirements and specifications (12.3% responses), and (3) changing requirements and specifications (11.8% responses).

These findings make a very compelling reason to make sure that requirements are well defined and documented. Documenting requirements is important for a number of reasons. First of all, you cannot satisfy a user's need and build what you are supposed to unless you know what it is. The only way to discuss multiple requirements is to have them written down so they can be reviewed, fine-tuned, and approved. This is true regardless of the development approach whether traditional or agile. The only thing that is different is how much is documented at what point in the process and how it is documented.

Developing requirements documents costs time and money so it is an expense in the short term but in the long term it will cost less than it is worth. For example, it will help to get the right participation in the project and avoid useless meetings while providing a mechanism for tracking progress and project in general.

Users do not always know their current business needs especially as it relates to technology and in fact, at different levels of their organization, different users will have different views of their needs, priority, and urgency. A well written requirements document integrates all the different levels of the organization and different users' perspectives into one coherent system that considers all the needs. Having requirements documented allows different users to see the overall goal and how it fulfills their needs. It also sets the right expectations on what will be delivered as well as what will not be delivered in the current project.

Additionally, having customers and/or users and other major shareholders involved in writing the requirements not only clarifies the needs but also allows the different parties to come to an agreement. This in turn helps to prevent misunderstandings and thus saves time in wasted rework later in the project process.

Having thorough written requirements can help to make decisions earlier in the process on what is most cost effective and which "need" may be eliminated. Also, if one tracks the changes made because of meeting discussions, that information can be referred to later as needed to check what was agreed on and who made that decision.

Requirements that are not written down are not requirements or as some say, they never happened. In many ways, it is a contract between customer/users and the project team. If anyone

ever says that the program isn't working correctly, the first thing that you do is look at the approved requirements document.

Written requirements are used by both the development team and testers throughout the life of the project. They are first used to design the system and then to create test cases. They provide the information on what the product should do and help in scheduling time and resources to plan the project more accurately. In addition,

Requirements need to be managed throughout the development life cycle and in fact throughout the product's life regardless of the development approach. Customers/users may ask for changes or new features which means referring back to the original requirements with the customers. Sometimes the change is then avoided or customer/user has to agree to eliminate another feature to stay within the project deadline. When those features were already requested in previous project but then for whatever reason not implemented, the written requirements can save time and money so that work is not recreated. If there was an issue with those requirements in the first place, again time and money is saved so that the analysis is not done all over again.

Therefore, there are a number of problem that may be avoided or at least lessened by having a good requirements document:

- Building system which does not meet the needs of stakeholders
- Building a project from inconsistent or incomplete requirements
- Making too many preventable changes to requirements during development, which is costly
- Misunderstandings between customers/users and developers which result in incorrect product
- Conflicts with shareholders on what the requirements are and what was agreed to
- Inaccurate schedules and/or allocation of resources
- Late additions of new features because the system needs were not well understood

What is the requirements document used for

There are a number of activities that use requirements document for a number of purposes throughout the development life cycle. They include:

- Project scoping
- Cost estimating
- Budgeting
- Project scheduling
- Software design
- Software testing
- Documentation and training manuals

Project scoping establishes what the project will include and not include and seeks to get approval to the scope from all shareholders and interested parties. Some projects, especially traditional ones, are strict about defining scope because of the financial implications. Written requirements are critical information to define the scope. In agile-type development approach, a specific project will commit to a number of core or critical requirements as part of the scope and then have additional requirements identified to be included as time allows.

Once project scope is established using the written requirements, cost estimate can be derived. There are a number of different techniques that are used to estimate the cost of a project to make sure that it is financially feasible. Regardless of a technique however, written requirements are the key component that is used to determine the project costs such as personnel, hardware, software, equipment, materials, etc.

The next step is to create a budget for the project to track the project costs. Budget should be tied to the deliverables which are based on the requirements. As the project progresses and the requirements are implemented, one can determine if the project is on budget or not based on the original estimates. Project schedules then take the requirements and break them down into tasks with start and end dates. This allows the project manager to track and determine if the project is on time.

Requirements documentation provides the information on WHAT the system should have and do. The software design then uses that information to determine HOW the system will work and be implemented to satisfy the requirements. Software testing then uses the requirements document to derive the test cases to make sure that the built system does in fact work correctly and satisfies the requirements. Just like one cannot develop a product without knowing what it should be, a tester cannot check that it meets the user's needs unless he/she knows what those needs are.

Lastly, product documentation and training manuals are also derived from the requirements document because their goal is to explain and describe how the user would accomplish the tasks he needs using the product.

How used throughout Development Life Cycle

As mentioned earlier, the requirements document is used throughout the product life cycle. Once it is drafted, it is validated, revised and approved and it is used to record, organize, and prioritize requirements.

We will go over requirements validation in detail in another chapter but basically it is the process of verifying with the customer/user that the drafted requirements meet their needs and to make any changes and corrections based on these discussions. This process may take a few iterations until all shareholders are satisfied that the requirements are complete and clear and the user/customer signs off on the document.

During the product development, designers use requirements document to draft the system design which then is approved by shareholders to include product owner, developers, and testers. If there is any question on any aspect of the design or later in the development cycle during code review, the requirements document is checked to decide on the correct path. Sometimes there may be a need to go back to the customer/user to clarify the requirement and in agile development, it is not until the requirement is close to being developed that additional details for the requirement are worked out with the customer/user and documented.

Testers use the requirements document to derive their test cases and then during the testing they will refer back to it to make sure their test case is correct or whether the test case found a defect. If development adds a feature that is not in the requirements document, testing team may negotiate or even refuse to test the feature (due to time constraints) that was not requested by the customer/user.

At the end of the project, there may be requirements that did not fit into the release and those may be the start of the requirements list for the next project.

Audience for the requirements document

Based on the above uses of the document, requirements document is written for multiple audiences although not all sections of the document are of interest to all.

The primary audience for the document are customers, end users, designers, developers, testers, and documentation experts. The document is also reviewed and possibly approved by business analyst, product manager, technical lead, product owner, development and test managers, standards expert, quality auditor, and User Interface experts.

When drafting the document, one should consider the targeted audience for each section to make sure it has the needed information in the format most easily understood and useful for that audience. This way the users of the document will find it readable and of value, of appropriate complexity (neither confusing nor boring), and it will help them to avoid errors due to misunderstandings. It will also be of benefit to the author(s) of the document because it will avoid unnecessary re-work and/or unnecessary surplus detail, reduce cost/time to draft the document, and increase satisfaction of getting it correct the first time.

Components of Requirements Specification Document

In order to serve the different audiences, requirement document will need more than just a list of requirements. How much and what will be documented will depend on the product type (e.g. safety critical systems require more documentation), size and complexity of the product, and the development approach.

Regardless of the size and development approach, however, there should be an overview of the finished product whether in narrative or graphical form or both and the purpose of the product, to put the requirements in context. A reader of this overview should have a good grasp on the overall product without needing to look at the details contained in the requirements.

In addition, the document may have what the development tools will be used; plans for budget and staffing; initial development schedule; how the product will be distributed or accessed, etc.

Requirements documentation will differ across different organizations, products, and projects but it may include some of the following sections:

- Title of the product/project and the author(s) information
- Purpose of the document
- Scope of the project (what it contains and what is outside the scope)
- Primary audience for the document
- Market assessment or business drivers (business reasons for the product and both opportunities and problems being solved)
- Product overview
- UML Use Cases diagrams or other graphical representation of the high level business objective
- Requirements to include functional, non-functional, usability, environmental, support, and interactions (when works with other products)
- Assumptions (e.g. assumptions made when gathering and analyzing requirements)

- Constraints (e.g. customer imposed some design constraints forcing certain solution)
- Dependencies (e.g. on technology, environment, or other products)
- Initial high level plans such as budget, timelines, milestones, etc. (detailed plans are developed in project plan documents)
- Acceptance criteria and/or performance metrics (what needs to be true for a stage/iteration or the final product to be complete and accepted by the customer)

In agile development approach (e.g. extreme programming, scum, etc.) the initial documentation is more lightweight and is developed as the project progresses. However it still must have many of the above components at some point in the product such as the overview and product purpose initially, use cases or user stories (functional and non-functional) for the project backlog with more detail added as the project progresses, and acceptance criteria and tests.

How can requirements document lead to project failure

The first way is obvious, it may be missing altogether. Some projects may try for a proof-of-concept or a prototype first before developing even the most basic requirements document. So they start working on a solution before having a clear idea what needs they are trying to solve.

Then the requirements document may actually be a disguised design because instead of concentrating on the user needs and what the system should do, it concentrates on the solution and how the system will do something. This can lead to a number of problems such as missing the actual user's need because everyone concentrates on the solution which may or may not be correct or complete; it may miss a better solution that could have been developed if the needs were considered by whole team instead being presented with a solution; it may make it harder for the user to relate to the requirements because they are not written from his/her perspective in his/her language.

Keeping balance in documentation however is important. Having too much documentation can be as bad as having too little. Having too much detail upfront usually means that a solution is being discussed and not the need. Also, having detailed and long requirements documents means that one will have greater trouble having the user and other shareholders review it properly. It also may lock the team into one perspective and view of the system thus stifling creativity in the design stage. Whether traditional or agile, there needs to be "just enough" documentation as appropriate for that development approach.

Lastly, not enough time was taken to understand and analyze all types of user's needs through various research methods to make sure that the needs are thoroughly understood to include hidden and unmet needs and not just those that were initially expressed by the user.

Types of requirements documents

When one hears of "requirements document" one normally thinks of Software Requirements Specification (SRS) document which includes functional and non-functional requirements. This document may also be called Functional Specification Document (FSD), Product Specification Document (PSD), Product Requirements Document (PRD), Business Requirements Document (BRD), Functional Spec, or just Software Specification.

However, “requirements documentation” may represent other documents as well. Sometimes these are just a section of the SRS or a separate document depending on the size and complexity of the project. Here are some examples:

- **Stakeholder Analysis Specification** – every project needs to determine who the users and/or target audience for the product are as well as the subject matter experts. This information is necessary to collect accurate requirements and to get the appropriate review and approvals.
- **Business Analysis Plan** – before drafting SRS, the business analyst may first draft a plan that describes the activities and tasks for requirements elicitation, requirements analysis, as well as the validation and verification that needs to be done. This plan will include who is responsible for each task and there may be an appropriate schedule as well.
- **Current State Analysis** – whether the proposed product is providing a brand new solution or replacing an old one, there needs to be a good understanding of the current business process and domain in order to understand the customer needs and scope of the project. Depending on the complexity of the product and the business, it may be necessary to document the current processes and practices.
- **Market Requirements Document** – this document describes the customer’s wants from marketing view so it includes the target audience, competing products, and why customers would want this specific product. This document or this information would be used when there is no specific customer but rather an IT company is creating a product for many users. This information may be used for project proposal.
- **Scope Statement Specification** – every project requires the scope statement which provides a clear description of what problem the product is solving and for what business need, what is the scope of the solution (and what is outside the scope), and why the solution is worth the investment. In agile development approach this information may be contained in the epic user story.
- **Wireframes and Other Visual Documentation** – projects often have visual representations for the overall solution and or features in the form of wireframes, storyboards, UML diagrams, etc. to enhance the understanding of the project. These can be part of the SRS or as supplemental documentation. There may also be a section or a separate document for the User Interfaces (UI) which includes the mocks ups and descriptions of the product screens and pages.
- **Information or Data Model Documentation** – every project needs to provide a domain model and/or data dictionary and data mapping for the system being developed. This includes information about database or file and other data repositories, the information that is obtained and saved, the format of the data and the interfaces, etc.
- **Test Plans, Test Cases, or User Acceptance Test Plans** – test documents are usually separate from requirements specification but they support the requirements documentation by providing the means and methods that will be used to verify that the product meets the requirements and that it works correctly
- **Change Management Documentation** – depending on the development approach, when changes are needed to the documented and approved requirements, there may be separate documents drafted for that purpose especially if the changes are substantial.

Government requirements documents

Government agencies usually have stricter documentation requirements than private organizations and they often provide templates for their documents. For example, General Services Administration (GSA) under Assisted Acquisition Services (AAS) department provides AAS Requirements Document template at https://www.gsa.gov/cdnstatic/AAS_Requirements_Document_%281%29.doc.

This document includes a lot of detailed information such as:

- project name and number
- start and end date
- security level
- project background and history
- project objectives, scope and requirements
- desired skills for the vendor doing the work
- expected deliverables
- market research information to include alternative products and solutions, logistics, and resources to include how this information was obtained and from whom
- Federal Information Security Management Act (FISMA) of 2002 Compliance information
- HSPD-12 Compliance information
- Evaluation criteria, scoring guide, and rating
- Quality Assurance (QA) Planning to include surveillance plan matrix and QA guidelines
- Independent Government Estimated (IGE) for manpower, materials, travel costs, etc.
- Funds Management Tools

References

Battson, C. Determining your audience. Retrieved December 11, 2017, from <http://www.astauthors.co.uk/aboutast/articles/determiningyouraudience.php>

Brandenburg, L. What Requirements Documents Does A Business Analyst Create? Retrieved December 15, 2017, from <http://www.bridging-the-gap.com/requirements-documentation/>

De Witt, D. (2008, Nov 25). Top Ten Reasons You REALLY Do Need a Requirements Document. Retrieved December 11, 2017, from <http://www.drdobbs.com/architecture-and-design/top-ten-reasons-you-really-do-need-a-req/212200380>

Harrin, E. (2014, Jan 16). Why requirements documentation is essential. Retrieved December 15, 2017, from <http://www.bcs.org/content/conBlogPost/2272>

Inflectra. (2016, Oct 13). What is a System Requirement Specification (SRS). Retrieved December 11, 2017 from, <https://www.inflectra.com/ideas/topic/requirements-definition.aspx>

Kramer, T. (2017, Mar 24). 3 Ways Requirements Documents Kill Product Development Projects. Retrieved December 15, 2017 from <https://www.designnews.com/design-hardware-software/3-ways-requirements-documents-kill-product-development-projects/24388564156526>

McEwen, S. (2004, Apr 16). Requirements: An Introduction. Retrieved December 11, 2017, from <https://www.ibm.com/developerworks/rational/library/4166.html>

McGovern, F. (2010). Blending Traditional and Agile Project Documentation. Retrieved December 15, 2017, from <https://www.visiblethread.com/wp-content/uploads/Lean-Documentation-Blending-Traditional-and-Agile-Project-Documentation.pdf>

Product requirements document. Retrieved December 11, 2017, from https://en.wikipedia.org/wiki/Product_requirements_document

Smith, R. Writing a Requirements Document. Retrieved December 11, 2017 from, <http://www.acqnotes.com/Attachments/Writing%20a%20Requirements%20Document%20For%20Multimedia%20and%20Software%20Projects%20by%20Rachel%20S.%20Smith.pdf>

Types of Requirements Documents. Retrieved December 15, 2017, from http://www.requirementsmanagementschool.com/w1/Types_of_Requirements_Documents

Wensel, S. (2003, Oct 3). Writing Requirements Documentation for Multiple Audiences. Retrieved December 11, 2017 from, <https://www.stickyminds.com/article/writing-requirements-documentation-multiple-audiences>

Chapter 5 Functional Decomposition

McFADDEN, R. (2017). SOFTWARE REQUIREMENTS – REQUIREMENTS DOCUMENTATION, UNDER CONTRACT FROM UMGC.

Introduction

Functional decomposition is the most traditional way of approaching the requirements. It is still used across different organizations especially government agencies but for many it has been replaced by more modern approaches that are user-centered such as use cases or user stories.

In this methodology, requirements concentrate on high level functions or features and how they interact with each other and then break them down into smaller functional units and elements thus creating a hierarchical structure. Each level of decomposition has additional detail added and traditionally the requirements are written to describe what the system should have such as "The system shall have a menu with ..."

IEEE Standard 830-1998 Recommended Practice for Software Requirements Specifications recommends this approach with a section in the System Requirements Specification (SRS) section for requirements. It states that "requirements should include at a minimum a description of every input (stimulus) into the system, every output (response) from the system, and all functions performed by the system in response to an input or in support of an output." (IEEE 830-1998, p. 21) And that "Functional requirements should define the fundamental actions that must take place in the software in accepting and processing the inputs and in processing and generating the outputs. These are generally listed as 'shall' statements starting with 'The system shall'." (IEEE 830-1998, p. 22). Thus, the requirements are presented as the system's capabilities and constraints.

The standard then gives a list of the actions and processing that should be included for the requirements such as (IEEE 830-1998, p. 22):

- a) Validity checks on the inputs
- b) Exact sequence of operations
- c) Responses to abnormal situations, including
 - Overflow
 - Communication facilities
 - Error handling and recovery
- d) Effect of parameters
- e) Relationship of outputs to inputs, including
 - Input/output sequences
 - Formulas for input to output conversion

The constraints then should include what will be supported and how relative to (IEEE 830-1998, p. 19):

- System interfaces – list of system interfaces with description which functionality will satisfy the system requirements
- User interfaces – logical characteristics of the interface between user and the system such as screen formats, page and window layouts, function keys, etc.

- Hardware interfaces – logical characteristics of interfaces between the software and hardware such as number of ports, which devices are supported, protocols, etc.
- Software interfaces – description of what other software products or systems will be used or interfaced with such as data management system, mathematical package, accounts receivable system, etc.
- Communications interfaces – list and description of interfaces to communication such as local protocols
- Memory – description of characteristics and limits for primary and secondary memory
- Operations – description of normal and special operations requirements such as different modes of operations (e.g. user initiated versus automatic), when system will be used versus unattended, what data processing support functions are needed, and the backup and recovery processes
- Site adaptation requirements – description of the requirements for data or initialization that are specific to a site and how the software needs to handle these unique requirements of each installation

Requirements structure formats

The typical format of the requirements using this approach is to have a list of features with the functional requirements such as:

3 System Features and Modules

3.1 System Feature 1

3.1.1 Introduction

3.1.2 Stimulus/Response Sequence

3.1.3 Functional Requirements

[R1.1] requirement 1 description

[R1.2] requirement 2 description

...

[R1.n] requirement n description

3.2 System Feature 2

3.2.1 Introduction

3.2.2 Stimulus/Response Sequence

3.2.3 Functional Requirements

[R2.1] requirement 1 description

[R2.2] requirement 2 description

...

[R2.n] requirement n description

Where stimulus/response sequence describes the trigger for the feature (e.g. selecting or clicking an option) and response gives a high level view of what will happen for the trigger. Then each functional requirement describes details of some aspect of that feature. Each requirement can then be broken down further into sub-requirement as needed.

Other recommended formats include organizing the functional requirements by different identities instead of features such as mode, user class, object, or stimulus.

Using mode:

3.2 Functional requirements

3.2.1 Mode 1

3.2.1.1 Functional requirement 1.1

...

3.2.1.n Functional requirement 1.n

3.2.2 Mode 2

...

3.2.m Mode m

3.2.m.1 Functional requirement m.1

...

3.2.m.n Functional requirement m.n

Using user class:

3.2 Functional requirements

3.2.1 User class 1

3.2.1.1 Functional requirement 1.1

...

3.2.1.n Functional requirement 1.n

3.2.2 User class 2

...

3.2.m User class m

3.2.m.1 Functional requirement m.1

...

3.2.m.n Functional requirement m.n

Using object:

3.2 Classes/Objects

3.2.1 Class/Object 1

3.2.1.1 Attributes (direct or inherited)

3.2.1.1.1 Attribute 1

...

3.2.1.1.n Attribute n

3.2.1.2 Functions (services, methods, direct or inherited)

3.2.1.2.1 Functional requirement 1.1

...

3.2.1.2. m Functional requirement 1.m

3.2.1.3 Messages (communications received or sent)

3.2.2 Class/Object 2

...

3.2.p Class/Object p

Using stimulus:

3.2 Functional requirements

3.2.1 Stimulus 1

- 3.2.1.1 Functional requirement 1.1
- ...
- 3.2.1.n Functional requirement 1.n
- 3.2.2 Stimulus 2
- ...
- 3.2.m Stimulus m
- 3.2.m.1 Functional requirement m.1
- ...
- 3.2.m.n Functional requirement m.n

Examples

Here is an example of requirements for a game statistics feature:

4.2 Statistics

4.2.1 Description

Program needs to keep track of the game statistics.

4.2.2 Stimulus/Response Sequences

User clicks on “Display Stats” button on the initial screen. Stats screen pops up.

4.2.3 Functional Requirements

[R2.1] System shall display the game statistics screen with the number of accumulative count of the rounds player has won, lost, or had a tie

[R2.2] System shall initialize the statistics to 0 at the start of the game

...

[R2.n] xxxx

Another example for a library system and book search help feature:

3.3 Help

3.3.1 Description

When user selects Help, system will provide instructions on book search format.

3.3.2 Stimulus/Response Sequences

User selects “Help” option from the main menu. System provides help instructions.

3.3.3 Functional Requirements

[R3.1] System shall display instructions on logical operators that can be used (OR and AND) for the search.

[R3.2] System should provide explanation of the following key words:

AUTHOR: name – last name of author

TITLE: “title in double quotes” – partial or full title of the book

SUBJECT: “what category the book may be under in quotes”

YEAR: value – the year the book was published

[R3.3] System shall provide examples for the search, such as:

AUTHOR:Ammann AND TITLE:”Introduction to Software Testing”

...

Challenges of this approach

While these traditional requirements may provide context for the business and stakeholders, that is not the focus. The focus is on the system and how it will work. As a result, many assumptions can and are made about the stakeholders' needs and they may not be accurate.

Then the requirements documents have tendency to be very detailed and because they are written from the perspective of the system and not the user, they have tendency to concentrate on the solution and not the problem. As this approach goes hand in hand with waterfall development method, all of the requirements and their details are written up front, thus taking a long time, often becoming outdated, and require constants updates.

Any changes to the requirements need to be negotiated since the requirements are treated as a contract. Also, as the users concentrate on the interfaces and the interactions with the system and not the system itself, a small change in the user interface can impact many requirements. All of this can become quite costly.

In addition, the requirements have tendency to be technical and so they may not be easily understood by the stakeholders. As a result, development may have a number of features they think are important but which in fact do not satisfy any of the stakeholders' needs. There is also a tendency to have a greater churn and updates in general because of the design details that have tendency to be included with this approach.

Government projects and safety-critical systems

The traditional approach to eliciting and forming requirements have not led to successful projects over the years, hence the newer user-oriented approaches have been developed. However, government and safety-critical systems need to be concerned with meeting standards and regulations and have additional safety and security rules that need to be adhered to. This in turn means detailed requirements that need to be written and negotiated up front.

Nevertheless, these extra constraints do not mean that the requirements cannot be written from the user perspective instead of the system nor that they cannot be written in more modern formats such as use cases or even user stories. They do however may need to have more details and requirements that would be considered internal workings of the system and a design rather than the external behavior. For example, the requirements may need to specify the handling of error faults, invalid data or state checks throughout the code, dependencies of data and state, and handling of code paths that would typically be considered design and externally would not appear different to the user but yet be critical for safety or security rules or standards.

References

Eriksson, M. (2016, Mar 29). Product requirement documents must die. Retrieved January 11, 2017, from <https://www.mindtheproduct.com/2016/03/product-requirement-documents-must-die/>

IEEE Standard 830-1998 Recommended Practice for Software Requirements Specifications

Functional decomposition. Retrieved January 9, 2017, from https://en.wikipedia.org/wiki/Functional_decomposition

Suscheck, C. (2012, Jan 18). Defining Requirement Types: Traditional vs. Use Cases vs. User Stories. Retrieved January 11, 2017, from <https://www.cmcrossroads.com/article/defining-requirement-types-traditional-vs-use-cases-vs-user-stories>

Chapter 6 Use Cases

SAFDARI, R., FARZI, J., GHAZISAEIDI, M., MIRZAEI, M., & GOODINI, A. (2013). THE APPLICATION OF USE CASE MODELING IN DESIGNING MEDICAL IMAGING INFORMATION SYSTEMS. ISRN RADIOLOGY, 2013.

Introduction


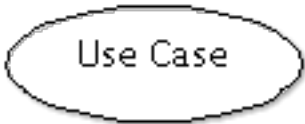


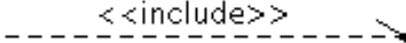
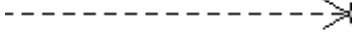

Background and Objective. “Use case modeling” is in fact a conceptual model and explains the required behaviors in the framework of related nontechnical and practical terms. For many organizations this model is a real model to show the performance requirements and visible outward behaviors of an information system [1]. Use case modeling simply shows in an information system which does what job and with what aim [2]. This technique has been developed as a part of UML in the form of a standard objective-oriented modeling language [3] which is based on the system’s simple structure and direct understanding, usually presenting considerable details about the interaction between a system and its users in a structured story-like manner [4].

Use case modeling can be done during software development so that the practical behavior of large systems in their interaction with several users having different requirements is shown [5]. Usually one environment is used in relation to modeling a process [6]. Use case modeling can divide every collection into two particular groups: the system and the factors interacting with this system [7, 8]. These factors are called actors. Actors are not limited to the real people. They can be other systems, equipment, or any other thing which can interact or perform a function in the system [9]. For example, the actors of a radiology use case model can be the radiologists, doctors, patients, technicians, students, researchers, or even Picture Archiving and Communication System (PACS) or Health Information System (HIS) [10]. Considering all the actors interacting with a system and through user cases showing the reason, cause, and manner of the actors’ interaction in the system, use case model can define a system [5, 10].

Use case model revolves around a scenario that defines a process including the system [1]. Usually use case model is in the form of successive events which is valuable for the user. Use case model defines: (a) the actors of the scenario [1], (b) the aims of the actors [10], (c) the related events and the prerequisites of the processes like the start time of the process [10], (d) the main flow of the events and the probable alternatives (the manner of process appearance according to the expression of the actors and system activities [1]), and (e) the final manner of the system based on the pre-conditions [10].

Model Structure. Developers utilize use case models primarily to collect performance necessities focusing on the practical conditions and the tasks users intend to do with the information systems [2]. In order to get to know the symbols used in this system, its syntactic structure has been defined in Table 1, according to Unified Modeling Language (UML) version 1.5 [1, 2, 11]. In this concept a system is considered as a close frame of behavior and the user cases are expected to be pattern-oriented and contextual while remaining stable during the lifespan of the information system in the semi-structured form.

Table 1. The syntactic structure of user case model.

Construct	Representation	Definition(s)
System		A system is a “top-level subsystem in a model”.
Use Case		A “coherent unit of functionality provided by a system”.
Actor		An “entity external to the system under development, that communicates with the system in order to achieve certain goals”.
Association		A relationship that represents “the communication between the actor and the use case.”
Include		A relationship between two use cases that shows that an instance of one use case will also contain the behavior specified by another use case.
Extend		The extending use case specifies “alternative, conditional or exceptional courses of interaction that can alter or extend the main flow of events embodied in some base case.”
Generalization		“A generalization relationship from use case C to use case D indicates that C is a specialization of D.”

The elements of this table model the interactions of external users and system and display the behavioral aspects of the system [11]. This is also a starting point for all other diagrams explaining the requirements, architecture, and implementation of the system. Another advantage

of the systematic approach based on use case modeling is that the implementation stages can be distinguished from the generation stages [2].

References

- [1] B. Dobing and J. Parsons, "Understanding the role of use cases in UML: a review and research agenda," *Journal of Database Management*, vol. 11, no. 4, pp. 28–36, 2000.
- [2] A. Tirado-Ramos, J. Hu, and K. P. Lee, "Information object definition—based unified modeling language representation of DICOM structured reporting: a case study of transcoding DICOM to XML," *Journal of the American Medical Informatics Association*, vol. 9, no. 1, pp. 63–71, 2002.
- [3] Y. Lee, Y. Chae, and S. Jeon, "Integration and evaluation of clinical decision support systems for diagnosis idopathics pulmonary fibrosis [IPF]," *Healthcare Informatics Research*, vol. 16, no. 4, pp. 260–272, 2010.
- [4] R. Martinez, J. Rozenblit, J. F. Cook, A. K. Chacko, and H. L. Timboe, "Virtual management of radiology examinations in the virtual radiology environment using common object request broker architecture services," *Journal of Digital Imaging*, vol. 12, no. 2, pp. 181–185, 1999.
- [5] M. Odeh, T. Hauer, R. McClatchey, and T. Solomonides, "A usecase driven approach in requirements engineering: the MammoGrid project," in *Proceedings of the 7th IASTED International Conference on Software Engineering and Applications*, pp. 562–567, Marina del Rey, Calif, USA, November 2003.
- [6] R. Warren, A. E. Solomonides, C. del Frate et al., "MammoGrid—a prototype distributed mammographic database for Europe," *Clinical Radiology*, vol. 62, no. 11, pp. 1044–1051, 2007.
- [7] T. C. Pan, M. N. Gurcan, S. A. Langella et al., "Informatics in radiology—GridCAD:Grid-based computer-aided detection system," *Radiographics*, vol. 27, no. 3, pp. 889–897, 2007.
- [8] F. Estrella, T. Hauer, R. McClatchey, M. Odeh, D. Rogulin, and T. Solomonides, "Experiences of engineering Grid-based medical software," *International Journal of Medical Informatics*, vol. 76, no. 8, pp. 621–632, 2007.
- [9] A. Cohen, A. Laviv, P. Berman, R. Nashef, and J. Abu-Tair, "Mandibular reconstruction using stereolithographic 3-dimensional printing modeling technology," *Oral Surgery, Oral Medicine, Oral Pathology, Oral Radiology and Endodontology*, vol. 108, no. 5, pp. 661–666, 2009.
- [10] A. A. T. Bui, R. K. Taira, J. D. N. Dionisio, D. R. Aberle, S. El-Saden, and H. Kangarloo, "Evidence-based radiology: requirements for electronic access," *Academic Radiology*, vol. 9, no. 6, pp. 662–669, 2002.
- [11] G. Irwin and D. Turk, "An ontological analysis of use case modeling grammar," *Journal of the Association For InFormation Systems*, vol. 6, no. 1, pp. 1–37, 2005.
- [12] S. M. Asch, E. A. McGlynn, M. M. Hogan et al., "Comparison of quality of care for patients in the veterans health administration and patients in a national sample," *Annals of Internal Medicine*, vol. 141, no. 12, pp. 938–945, 2004.

- [13] A. S. Young, E. Chaney, R. Shoi et al., "Information technology to support improved care for chronic illness," *Journal of General Internal Medicine*, vol. 22, no. 3, pp. 425–430, 2007.
- [14] R. Safdari, H. Dargahi, L. Shahmoradi, and A. Farzaneh Nejad, "Comparing four softwares based on ISO 9241 part 10," *Journal of Medical Systems*, pp. 1–7, 2011.
- [15] M. Glinz, S. Berner, and S. Joos, "Object-oriented modeling with Adora," *Information Systems*, vol. 27, no. 6, pp. 425–444, 2002.
- [16] J. Farzi, P. Salem Safi, A. R. Zohoor, and F. Ebadi Fardazar, "The study of national diabetes registry system: model suggestion for Iran," *Journal of Ardebil University of Medical Sciences & Health Services*, vol. 3, no. 8, pp. 288–293, 2008.
- [17] R. Safdari, M. Torabi, J. Farzi, M. mirzaee, and A. Goodini, "Intelligence risk detection models: tools to promote patients safety level," in *Proceedings of the 3th Symposium for E-Hospital and Telemedicine*, p. 56, Tehran University of Medical Sciences, Tehran, Iran, 2012.
- [18] T. Sohmura, N. Kusumoto, T. Otani, S. Yamada, K. Wakabayashi, and H. Yatani, "CAD/CAM fabrication and clinical application of surgical template and bone model in oral implant surgery," *Clinical Oral Implants Research*, vol. 20, no. 1, pp. 87–93, 2009.
- [19] Z. Daw, R. Cleaveland, and M. Vetter, "Formal verification of software-based medical devices considering medical guidelines," *International Journal of Computer Assisted Radiology and Surgery*, 2013.
- [20] S. Ogata and Matsuura, "A review method for UML requirements analysis model employing system-side prototyping," *Springerplus*, vol. 2, no. 1, p. 134, 2013.
- [21] J. F. Cook, J. W. Rozenblit, A. K. Chacko, R. Martinez, and H. L. Timboe, "Meta-Manager: a requirements analysis," *Journal of Digital Imaging*, vol. 12, no. 2, supplement 1, pp. 186–188, 1999.
- [22] T. Liebler, M. Hub, C. Sanner, and W. Schlegel, "An application framework for computer-aided patient positioning in radiation therapy," *Medical Informatics and the Internet in Medicine*, vol. 28, no. 3, pp. 161–182, 2003.

Use Case Diagram

MODELING UML USE CASE DIAGRAMS AND CAPTURING USE CASE SCENARIOS, RETRIEVED FROM <http://vlabs.iitkgp.ernet.in/se/3/theory/>

Use case diagrams

Use case diagrams belong to the category of behavioural diagram of UML diagrams. Use case diagrams aim to present a graphical overview of the functionality provided by the system. It consists of a set of actions (referred to as use cases) that the concerned system can perform, one or more actors, and dependencies among them.

Actor

An actor can be defined as an object or set of objects, external to the system, which interacts with the system to get some meaningful work done. Actors could be human, devices, or even other systems.

For example, consider the case where a customer *withdraws cash* from an ATM. Here, customer is a human actor.

Actors can be classified as below:

Primary actor: They are principal users of the system, who fulfill their goal by availing some service from the system. For example, a customer uses an ATM to withdraw cash when he needs it. A customer is the primary actor here.

Supporting actor: They render some kind of service to the system. "Bank representatives", who replenishes the stock of cash, is such an example. It may be noted that replenishing stock of cash in an ATM is not the prime functionality of an ATM.

In a use case diagram primary actors are usually drawn on the top left side of the diagram.

Use Case

A use case is simply a functionality provided by a system.

Continuing with the example of the ATM, *withdraw cash* is a functionality that the ATM provides. Therefore, this is a use case. Other possible use cases includes, *check balance*, *change PIN*, and so on.

Use cases include both successful and unsuccessful scenarios of user interactions with the system. For example, authentication of a customer by the ATM would fail if he enters wrong PIN. In such case, an error message is displayed on the screen of the ATM.

Subject

Subject is simply the system under consideration. Use cases apply to a subject. For example, an ATM is a subject, having multiple use cases, and multiple actors interact with it. However, one should be careful of external systems interacting with the subject as actors.

Graphical Representation

An actor is represented by a stick figure and name of the actor is written below it. A use case is depicted by an ellipse and name of the use case is written inside it. The subject is shown by drawing a rectangle. Label for the system could be put inside it. Use cases are drawn inside the rectangle, and actors are drawn outside the rectangle, as shown in figure - 01.

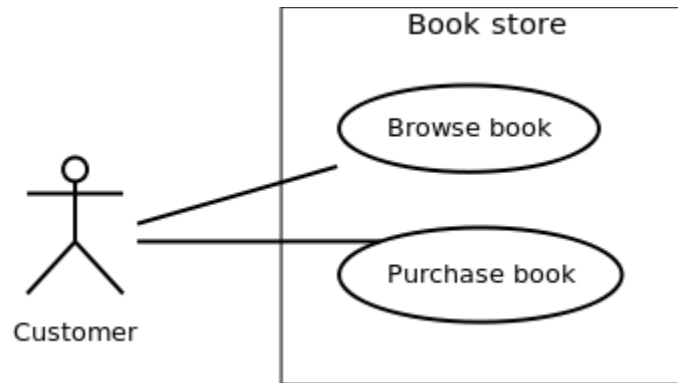


Figure - 01: A use case diagram for a book store

Association between Actors and Use Cases

A use case is triggered by an actor. Actors and use cases are connected through binary associations indicating that the two communicates through message passing.

An actor must be associated with at least one use case. Similarly, a given use case must be associated with at least one actor. Association among the actors are usually not shown. However, one can depict the class hierarchy among actors.

Use Case Relationships

Three types of relationships exist among use cases:

- Include relationship
- Extend relationship
- Use case generalization

Include Relationship

Include relationships are used to depict common behaviour that are shared by multiple use cases. This could be considered analogous to writing functions in a program in order to avoid repetition of writing the same code. Such a function would be called from different points within the program.

Example

For example, consider an email application. A user can send a new mail, reply to an email he has received, or forward an email. However, in each of these three cases, the user must be logged in to perform those actions. Thus, we could have a *login* use case, which is included by *compose mail*, *reply*, and *forward email* use cases. The relationship is shown in figure - 02.

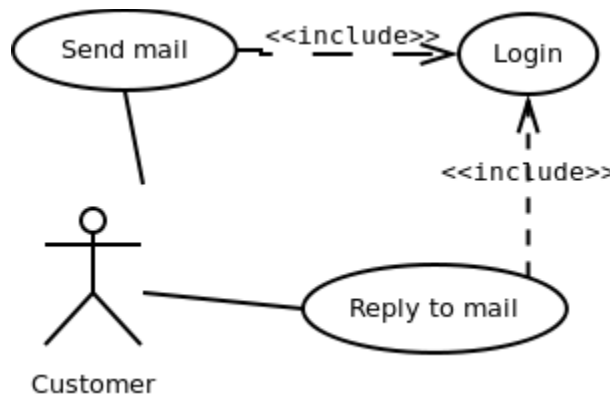


Figure - 02: Include relationship between use cases

Notation

Include relationship is depicted by a dashed arrow with a «include» stereotype from the including use case to the included use case.

Extend Relationship

Use case extensions are used to depict any variation to an existing use case. They are used to specify the changes required when any assumption made by the existing use case becomes false.

Example

Let's consider an online bookstore. The system allows an authenticated user to buy selected book(s). While the order is being placed, the system also allows to specify any special shipping instructions, for example, call the customer before delivery. This *Shipping Instructions* step is optional, and not a part of the main *Place Order* use case. Figure - 03 depicts such relationship.

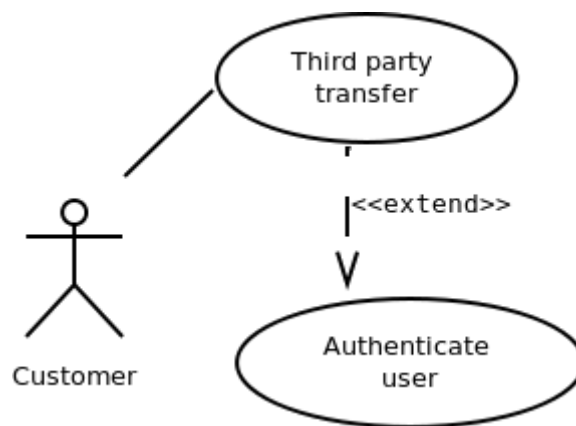


Figure - 03: Extend relationship between use cases

Notation

Extend relationship is depicted by a dashed arrow with a «extend» stereotype from the extending use case to the extended use case.

Generalization Relationship

Generalization relationships are used to represent the inheritance between use cases. A derived use case specializes some functionality it has already inherited from the base use case.

Example

To illustrate this, consider a graphical application that allows users to draw polygons. We could have a use case *draw polygon*. Now, rectangle is a particular instance of polygon having four sides at right angles to each other. So, the use case *draw rectangle* inherits the properties of the use case *draw polygon* and overrides its drawing method. This is an example of a generalization relationship. Similarly, a generalization relationship exists between *draw rectangle* and *draw square* use cases. The relationship has been illustrated in figure - 04.

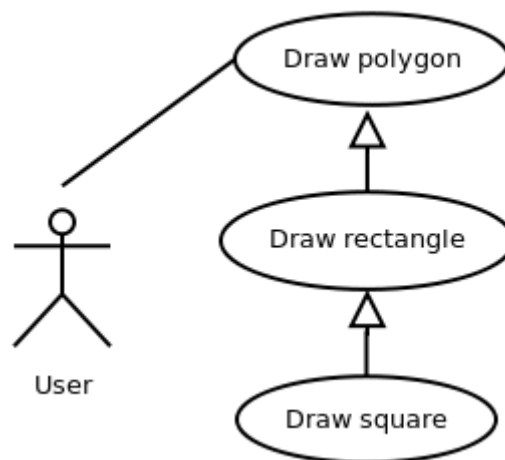


Figure - 04: Generalization relationship among use cases

Notation

Generalization relationship is depicted by a solid arrow from the specialized (derived) use case to the more generalized (base) use case.

Identifying Actors

Given a problem statement, the actors could be identified by asking the following questions:

- Who gets most of the benefits from the system? (The answer would lead to the identification of the primary actor)
- Who keeps the system working? (This will help to identify a list of potential users)
- What other software / hardware does the system interact with?
- Any interface (interaction) between the concerned system and any other system?

Identifying Use cases

Once the primary and secondary actors have been identified, we have to find out their goals i.e. what are the functionality they can obtain from the system. Any use case name should start with a verb like, "Check balance".

Guidelines for drawing Use Case diagrams

Following general guidelines could be kept in mind while trying to draw a use case diagram:

- Determine the system boundary
- Ensure that individual actors have well-defined purpose
- Use cases identified should let some meaningful work done by the actors
- Associate the actors and use cases -- there shouldn't be any actor or use case floating without any connection
- Use include relationship to encapsulate common behaviour among use cases, if any

Textual Use Case

SANTOS, I. S., ANDRADE, R. M., & NETO, P. A. S. (2015). TEMPLATES FOR TEXTUAL USE CASES OF SOFTWARE PRODUCT LINES: RESULTS FROM A SYSTEMATIC MAPPING STUDY AND A CONTROLLED EXPERIMENT. JOURNAL OF SOFTWARE ENGINEERING RESEARCH AND DEVELOPMENT, 3(1), 5.

Textual Use Case templates can be used to describe functional requirements of a Software Product Line (SPL). One of the requirements artifacts most used in SPL development are use cases (Alves et al. 2010). In Software Product Line development, the requirements engineering activity needs to cope with common and variable requirements for the whole set of products in the family. For this purpose, there are several use case templates available in the literature to describe the functional requirements of an SPL.

In the SPL paradigm the concepts of feature and feature model are essential. A feature is an attribute, quality or aspect visible to the user (Kang et al. 1990). According to the approach of Kang et al. (1990), the features can be “mandatory”, “optional” or “alternative”. Mandatory features are those available on all systems built within the family. Optional features are those features that may or may not be included in the products. Alternative features represent a selection “exactly-one-of-many” made from a set of features. A feature model represents the information of all possible products of an SPL in terms of features and the relationships among them (Benavides et al. 2010).

Tags

The use case template uses tags (e.g. [Vo], [ALT]) to indicate the variation points within the use cases. With the tags, use cases can also have a section where the variations are defined.

Alternative scenarios

The use case template describes the variations through alternative scenarios within the use cases description.

Specific section

The use case template describes all information about the variation points in a specific section. In this section, the variation type, a brief description and the use case steps that are affected by the variation are specified.

Step identifier

The use case template uses the step identifier of the use case to describe variants in use case scenarios.

Advice use case

The use case template describes the variabilities as advice use cases. The advice use cases capture crosscutting requirements and are defined in the same form as normal use cases, but they may only have some of the use case sections. The linking of advice use cases with affected base use cases is based on syntactical matching of joinpoints and pointcut expressions.

Figure 1 presents the use case “Withdraw Money” in the template found in the Bragança and Machado work (Bragança and Machado 2005). The interesting about this template is the use of questions (e.g. Is the identification done through the PIN? and Is there another identification type?) related with the variabilities aiming to guide the instantiation of the product use cases. In this example, the optional variant is described between the tags < variant OPT> and < /variant> . On the other hand, the alternative variants are described through the tags < variant ALT> and < /variant> .

Name: Withdraw Money

Short Description: Withdraw Money from the ATM

Actor: Customer (primary)

Precondition: ATM in idle state

Post-Condition: The Customer withdrew the money

Main Scenario:

1. The Customer inserts the chip card into the ATM.
2. The System request that the customer should be identified.
3. Is the identification done through the PIN?

<variant OPT>

The Customer presents its PIN.

The System verifies the PIN.

</variant>

4. Is there another identification type?

<variant ALT 1: yes, with the fingerprint>

The Customer presents the fingerprint to the ATM.

The System verifies the fingerprint.

</variant>

<variant ALT 2: yes, with the voice sample>

The Customer presents the voice sample to the ATM.

The System verifies the fingerprint.

</variant>

<variant ALT 0: no >

Ø

</variant>

5. The Customer selects the amount of money to withdraw.
6. The System deducts that amount from the customer account and dispenses the cash to the Customer
7. The Customer takes the cash from the Money dispenser.
8. The System is put into idle state.

Figure 1. Example of textual use case with tags

Figure 2 presents the use case “Withdraw Money” specified in the template proposed by Gomaa (2004). In this template, we can observe the description of the variabilities at the end of the use case. We highlight that Nguyen’s template (Nguyen 2009) extends Gomaa’s template (Gomaa 2004) to specify non-functional requirements. Then, in both templates the specification of the variabilities is made with the name of the variability, type of requirement (optional or alternative), line number of the use case affected by the variability, and the variability description.

Name: Withdraw Money

Reuse Category: Mandatory

Summary: Withdraw Money from the ATM

Actors: Customer (primary)

Precondition: ATM in idle state

Description:

1. The Customer inserts the chip card into the ATM.
2. The System request that the customer should be identified.
3. The Customer selects the amount of money to withdraw.
4. The System deducts that amount from the customer account and dispenses the cash to the Customer
5. The Customer takes the cash from the Money dispenser.
6. The System is put into idle state.

Variation Points

Name: PIN

Functionality type: Optional

Number of lines: 2

Description: The Customer presents its PIN. The System verifies the PIN>

Name: Identification

Functionality type: Alternative

Number of lines: 2

Description: The Customer presents the fingerprint to the ATM and the System verifies the fingerprint. The mutually exclusive alternative is the voice sample. The Customer presents the voice sample to the ATM. The System verifies the voice sample.

Figure 2. Example of textual use case with Specific Section

The template of Oliveira et al. (2013) describes the variabilities as alternative scenarios. Figure 3 shows the use case “Withdraw Money” described in this template, where the variabilities were specified with the following associated features: “PIN”, “Fingerprint” and “Voice”.

Name: Withdraw Money

Associated feature: Withdraw **Actor(s):** Customer

Pre-condition: ATM in idle state **Post-condition:** The Customer withdrew the money

Main Success Scenario

Step	Actor Action	Black box System Response
1	The Customer inserts the chip card into the ATM	The System request that the customer should be identified
2	The Customer selects that amount of money to be withdraw	The System deducts the amount from the customer account and dispenses the cash to the Customer
3	The Customer takes the cash from the Money dispenser	The System is put into idle state

Alternative Scenario Name: PIN

Associated Feature: PIN

Step	Actor Action	Black box System Response
1.1	The Customer presents its PIN	The System verifies the PIN
Alternative Scenario Name:		Fingerprint
Associated Feature:		Fingerprint
Step	Actor Action	Black box System Response
1.2	The Customer presents its Fingerprint	The System verifies the Fingerprint
Alternative Scenario Name:		Voice Sample
Associated Feature:		Voice Sample
Step	Actor Action	Black box System Response
1.2	The Customer presents its Voice Sample	The System verifies the Voice Sample

Figure 3. Example of textual use case with Alternative Scenarios

The Step Identifier structure used by Eriksson et al. (2005) is interesting because it uses the step identifier to specify the alternative and optional steps. In this template, for example, several steps identified with the same number identify a number of alternatives for one mandatory step (see step 3 in Fig. 4) while a number step identifier within parenthesis identifies an optional step in the scenario (see steps 2 and 3 in Figure 4).

Step	Actor Action	Black box System Action
1	The user case begins with The Customer inserts the chip card into the ATM	The System request that the customer should be identified
(2)	The Customer presents its @PIN_SIZE character long PIN	The System verifies the PIN
(3)	The Customer presents the fingerprint to the ATM	The System verifies the fingerprint
(3)	The Customer presents the voice sample to the ATM	The System verifies the voice sample
4	The Customer selects the amount of money to be withdrawn	The System deducts the amount from the customer account and dispenses the cash to the Customer

5	The Customer takes the cash from the Money dispenser	The System is put into idle state
---	--	-----------------------------------

Figure 4. Example of textual use case with Step Identifier.

Finally, the Advice Use Case structure is present in two templates (Anthonysamy and Somé 2008; Bonifácio and Borba 2009). Figure 5 shows the “Withdraw Money” specified in this template. In this case, we have one base use case named “Withdraw Money from the ATM” specifying the common behavior and three advice use cases that extends the behavior of the base use case: “Use PIN for user identification” that introduces an optional behavior before the step P2 ; “Use fingerprint for user identification ” and “Use voice sample for user identification” that introduce an optional alternative behavior also before the step P2 .

Id: SC 01

Description: Withdraw Money from the ATM

ID	User Action	System Response
P1	The Customer inserts the chip card into the ATM	The System request that the customer should be identified
P2	The Customer should select the amount of money to be withdrawn	The System deducts the amount from the customer account and dispenses the cash to the Customer
P3	The Customer takes the cash from the Money dispenser	The System is put into idle state

ID: ADV01

Description: Use PIN for user identification

Before: P2

Id	User Action	System Response
B1	The Customer presents its PIN	The System verifies the PIN

ID: ADV02

Description: Use Fingerprint for user identification

Before: P2

Id	User Action	System Response
C1	The Customer presents fingerprint to the ATM	The System verifies the fingerprint

ID: ADV03

Description: Use Voice Sample for user identification

Before: P2

Id	User Action	System Response
D1	The Customer presents voice sample to the ATM	The System verifies the voice sample

Figure 5. Example of textual use case with Advice Use Case

References

- Alves V, Niu N, Alves C, Valença G (2010) Requirements engineering for software product lines: A systematic literature review. *Inf Softw Technol* 52:806–820
- Alferez M, Bonifácio R, Teixeira L, Accioly P, Kulesza U, Moreira A, Araújo J, Borba P (2014) Evaluating scenario-based spl requirements approaches: the case for modularity, stability and expressiveness. *Requir Eng J* 19:355–376
- Anthony P, Somé S (2008) Aspect-oriented use case modeling for software product lines. In: *Proceedings of the 2008 AOSD Workshop on Early Aspects*. ACM, New York, NY, USA. pp 5–158
- Asadi M, Bagheri E, Mohabbati B, Gasevic D (2012) Requirements engineering in feature oriented software product lines: an initial analytical study. In: *Proceedings of the 16th International Software Product Line Conference*. ACM, New York, NY, USA. pp 36–44
- Assuncao WKG, Trindade DFG, Colanzi TE, Vergilio SR (2011) Evaluating test reuse of a software product line oriented strategy. In: *Proceedings of the 12th Latin American Test Workshop (LATW)*. IEEE Computer Society, Washington, DC, USA
- Azevedo S, Machado RJ, Bragança A, Ribeiro H (2012) On the refinement of use case models with variability support. *Innov Syst Softw Eng* 8:51–64
- Bertolino A, Gnesi S (2003) Use case-based testing of product lines. In: *Proceedings of the 9th European Software Engineering Conference*. ACM, New York, NY, USA. pp 355–358
- Bertolino A, Gnesi S (2004) Pluto: A test methodology for product families. *Lect Notes Comput Sci* Volume 3014:181–197
- Bertolino A, Fantechi A, Gnesi S, Lami G (2006). In: Käkölä T, Duenas JC (eds). *Software Product Lines Research Issues in Engineering and Management*. Springer, Brazil. Chap. 11 - Product Line Use Cases: Scenario - Based Specification and Testing of Requirements
- Benavides D, Segura S, Ruiz-Cortés A (2010) Automated analysis of feature models 20 years later: A literature review. *InfSyst* 35(6):615–636
- Blanes D, Insfrán E (2012) A comparative study on model-driven requirements engineering for software product lines. *Revista de Sistemas e Computação (RSC Journal)* 2:3–13

- Bonifácio R, Borba P (2009) Modeling scenario variability as crosscutting mechanisms. In: Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development. ACM, New York, NY, USA. pp 125–136
- Bragança A, Machado RJ (2005) Deriving software product line's architectural requirements from use cases: an experimental approach. In: Proceedings of the 2nd International Workshop on Model-Based Methodologies for Pervasive and Embedded Software. Turku Centre for Computer Science, Turku, Finland
- Bragança A, Machado RJ (2006) Extending uml 2.0 metamodel for complementary usages of the <<extend>> relationship within use case variability specification. In: Proceedings of the 10th International on Software Product Line Conference. IEEE Computer Society, Washington, USA. pp 123–130
- Bonifácio R, Borba P, Soares S (2008) On the benefits of scenario variability as crosscutting. In: Proceedings of the 2008 AOSD Workshop on Early Aspects. ACM, New York, NY, USA. pp 6–168
- Cheng BHC, Atlee JM (2007) Research directions in requirements engineering. In: Proceedings of the Future of Software Engineering (FOSE '07). IEEE Computer Society, Washington, DC, USA
- Chen L, Babar MA, Ali N (2009) Variability management in software product lines: a systematic review. In: Proceedings of the 13th International Software Product Line Conference. Carnegie Mellon University, Pittsburgh, PA, USA. pp 81–90
- Cockburn A (2000) Writing Effective Use Cases. 1st edn. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
- Choi W, Kang S, Choi H, Baik J (2008) Automated generation of product use case scenarios in product line development. In: Proceedings of the International Conference on Computer and Information Technology. IEEE Computer Society, Washington, DC, USA
- Colanzi TE, Assunção WKG, Trindade DFG, Zorzo CA, Vergilio SR (2013) Evaluating different strategies for testing software product lines. J Electronic Testing 29:9–24
- Dahan M, Shoval P, Sturm A (2014) Comparing the impact of the oo-dfd and the use case methods for modeling functional requirements on comprehension and quality of models. Requir Eng J 19:27–43
- Erikssona M, Borstler J, Borg K (2004) Marrying features and use cases for product line requirements modeling of embedded systems. In: Proceedings of the Fourth Conference on Software Engineering Research and Practice in Sweden. Institute of Technology, Unistryck, Linköping University, Linköping, Sweden. pp 73–82
- Eriksson M, Börstler J, Borg K (2005) The pluss approach: domain modeling with features, use cases and use case realizations. In: Proceedings of the 9th International Conference on Software Product Lines (SPLC'05). Springer-Verlag, Berlin, Heidelberg. pp 33–44
- Eriksson M, Morast H, Börstler J, Borg K (2005) The pluss toolkit?: Extending telelogic doors and ibm-rational rose to support product line use case modeling. In: Proceedings of the 20th

- IEEE/ACM International Conference on Automated Software Engineering. ACM, New York, NY, USA. pp 300–304
- Fantechi A, Gnesi S, Lami G, Nesti E (2004) A methodology for the derivation and verification of use cases for product lines. In: Proceedings of the International Software Product Line Conference. Springer-Verlag, Berlin, Heidelberg
- Fantechi A, Gnesi S, John I, Lami G, Dörr J (2004) Elicitation of use cases for product lines. In: Proceedings of International Workshop on Software Product-Family Engineering. Springer-Verlag, Berlin, Heidelberg
- Fant JS, Gomaa H, Pettit RG (2013) A pattern-based modeling approach for software product line engineering. In: Proceedings of the 46th Hawaii International Conference on System Sciences (HICSS). IEEE Computer Society, Washington, DC, USA. pp 4985–4994
- Ferrari R, Miller JA, Madhavji NH (2010) A controlled experiment to assess the impact of system architectures on new system requirements. *Requir Eng J* 15:215–233
- Gallina B, Guelfi N (2007) A template for requirement elicitation of dependable product lines. In: Proceedings of the 13th International Working Conference on Requirements Engineering: Foundation for Software Quality, REFSQ'07. Springer, Berlin, Heidelberg. pp 63–77
- Galster M, Weyns D, Tofan D, Michalik B, Avgeriou P (2014) Variability in software systems - a systematic literature review. *IEEE Trans Softw Eng* 40:282–306
- Gomaa H (2004) Designing Software Product Lines with UML: from Use Cases to Pattern-based Software Architectures. Addison Wesley, Redwood City, CA, USA
- GREat (2015) Group of Networking, Software and Systems Engineering. <http://www.great.ufc.br/index.php/en.html>
- Hadar I, Kuflik T, Perini A, Reinhartz-Berger I, Ricca F, Susi A (2010) An empirical study of requirements model understanding: Use case vs. tropos models. In: Proceedings of the 2010 ACM Symposium on Applied Computing. ACM, New York, NY, USA. pp 2324–2329
- Hollander M, Wolfe DA (1999) Nonparametric Statistical Methods. John Wiley & Sons, USA
- IBM (2002) The Rational Unified Process for Systems Engineering Whitepaper. <ftp://ftp.software.ibm.com/software/rational/web/whitepapers/2003/TP165.pdf>
- IBM (2015) Predictive Analytics Software and Solutions. <http://www-01.ibm.com/software/analytics/spss/>
- Jeyaraj A, Sauter VL (2007) An empirical investigation of the effectiveness of systems modeling and verification tools. *Commun ACM* 50(6):62–67
- Jirapanthong W (2009) Analysis on relationships among software models through traceability activity. In: Proceedings of the 3rd International Conference on Advances in Information Technology (IAIT). Springer-Verlag, Berlin, Heidelberg
- John I, Muthig D (2002) Product line modeling with generic use cases. In: Proceedings of the Workshop on Techniques for Exploiting Commonality Through Variability. Springer-Verlag, Berlin, Heidelberg

Kang KC, Cohen SG, Hess JA, Novak WE, Peterson AS (1990) Feature-oriented domain analysis (foda) - feasibility study.

Technical report, Software Engineering Institute, Carnegie Mellon University

Kamsties E, Pohl K, Reis S, Reuys A (2003) Testing variabilities in use case models. In: Proceedings of the 5th International Workshop Software Product-Family Engineering. Springer-Verlag, Berlin, Heidelberg. pp 6–18

Kitchenham B, Charters S (2007) Guidelines for performing systematic literature reviews in software engineering. Technical report, EBSE Technical Report

Kuloor C, Eberlein A (2002) Requirements engineering for software product lines. In: Proceedings of the International Conference Software Systems Engineering and Their Applications. CNAM - Conservatoire National des Arts et Métiers, Paris, France

Mustafa BA (2010) An experimental comparison of use case models understanding by novice and high knowledge users. In: Proceedings of the 2010 Conference on New Trends in Software Methodologies, Tools and Techniques. IOS Press, Amsterdam. pp 182–199

Morelli LB, Nakagawa EY (2011) A panorama of software architectures in game development. In: Proceedings of the International Conference on Software Engineering and Knowledge Engineering(SEKE). Knowledge Systems Institute Graduate School, Skokie, USA. pp 752–757

Nakanishi T, Fujita M, Yamazaki S, Yamashita N, Ashihara S (2007) Tailoring the domain engineering process of the plus method. In: Proceedings of the 14th Asia-Pacific Software Engineering Conference. IEEE Computer Society, Washington, DC, USA. pp 486–493

Neiva D (2009) Riple-re: A requirements engineering process for software product lines. Master's thesis, Federal University of Pernambuco

Neto PAMS, Machado IC, McGregor JD, Almeida ES, Meira SRL (2011) A systematic mapping study of software product lines testing. *Inf Softw Technol* 53:407–423

Nguyen QL (2009) Non-functional requirements analysis modeling for software product lines. In: Proceedings of the ICSE Workshop on Modeling in Software Engineering (MISE '09). IEEE Computer Society, Washington, DC, USA

Niu N, Easterbrook S (2008) Extracting and modeling product line functional requirements. In: Proceedings of the 16th IEEE International Requirements Engineering Conference. IEEE Computer Society, Washington, DC, USA

Northrop LM (2002) Sei's software product line tenets. *IEEE Softw.* 19:32–40

Northrop LM, Clements PC (2007) A Framework for Software Product Line Practice, Version 5.0. http://www.sei.cmu.edu/productlines/frame_report/

Oliveira RP, Insfran E, Abrahão S, Gonzalez-Huerta J, Blanes D, Cohen D, Almeida ES (2013) A feature-driven requirements engineering approach for software product lines. In: Proceedings of the VII Brazilian Symposium on Software Components, Architectures and Reuse. IEEE Computer Society, Washington, DC, USA

- Oliveira RP, Blanes D, Gonzalez-Huerta J, Insfran E, Abrahão S, Cohen S, Almeida ES (2014) Defining and validating a feature-driven requirements engineering approach. *J Universal Comput Sci* 20(5):666–691
- Petersen K, Feldt R, Mujtaba S, Mattsson M (2008) Systematic mapping studies in software engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering*. British Computer Society, Swinton, UK, UK. pp 68–77
- Reinhartz-Berger I, Sturm A (2014) Comprehensibility of uml-based software product line specifications. *Empirical SoftwEng* 19(3):678–713
- Santos IS, Neto PAS, Andrade RMC (2013) A use case textual description for context aware spl based on a controlled experiment. In: *Proceedings of the CAISE'13 Fórum*. CEUR-WS.org, Valencia, Spain
- Santos IS, Andrade RMC, Neto PAS (2014) How to describe spl variabilities in textual use cases: A systematic mapping study. In: *Proceedings of the Eighth Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS)*. IEEE Computer Society, Washington, DC, USA
- Santos IS (2015) Experiment Data. <https://sites.google.com/site/ismaylesantos/spl-use-case-experiment>
- Souza EF, Falbo RA, Vijaykumar NL (2013) Knowledge management applied to software testing: a systematic mapping. In: *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*. Knowledge Systems Institute Graduate School, Skokie, USA
- Tiwari S, Gupta A (2013) A controlled experiment to assess the effectiveness of eight use case templates. In: *20th Asia-Pacific Software Engineering Conference*, vol. 1. IEEE Computer Society, Washington, DC, USA. pp 207–214
- Urli S, Blay-Fornarino M, Collet P (2014) Handling complex configurations in software product lines: A toolled approach. In: *Proceedings of the 18th International Software Product Line Conference - Volume 1*. ACM, New York, NY, USA. pp 112–121. <http://doi.acm.org/10.1145/2648511.2648523>
- Wieringa R, Maiden N, Mead N, Rolland C (2006) Requirements engineering paper classification and evaluation criteria: a proposal and a discussion. *Requir Eng J* 11:102–107
- Wohlin C, Runeson P, Host M, Ohlsson M, Regnell B, Wesslen A (2000) *Experimentation in Software Engineering: An Introduction*. Kluwer Publishers, Norwell, MA, USA
- Yu W, Zhang W, Zhao H, Jin Z (2014) Tdl: a transformation description language from feature model to use case for automated use case derivation. In: *Proceedings of the 18th International Software Product Line Conference*. ACM, New York, NY, USA
- Zhou J, Lu Y, Lundqvist K, Lonn H, Karlsson D, Liwang B (2014) Towards feature-oriented requirements validation for automotive systems. In: *Proceedings of the IEEE 22nd International*

FERG, S. (2003) WHAT'S WRONG WITH USE CASES? WHAT'S WRONG WITH USE CASES?.

What's Wrong With Use Cases?

What's wrong with chocolate?

At one level, of course, nothing at all is wrong with chocolate. There aren't many essential nutrients in chocolate, of course, but in and of itself that is not a problem. Consumed in moderation, as part of a well-rounded diet, chocolate adds variety and pleasure to the diet, and in some cases may provide needed calories.

But of course, that isn't the end of the story. What's wrong with chocolate is that it is delicious. Addictive, in fact! There was a reason for the invention of the word chocoholic. Where moderate consumption might be useful, chocolate tempts us to immoderate consumption. Where we should consume a variety of foods to obtain a balanced intake of nutrients, chocolate tempts us to ignore other foods and eat only chocolate. If we succumb to its charms, we eat an unbalanced diet. If we eat an unbalanced diet, we suffer. The consequences of excessive chocolate consumption are very real and easy to see (on the bathroom scale, for instance). But (and here is part of the problem) the causal connection between chocolate consumption and its consequences is (in contrast to chocolate's easily perceived immediate charms) virtually invisible.

The problem with chocolate, then, lies in a combination of things: it does not provide a complete, balanced nutritional intake; it is extremely attractive, tempting one to consume only chocolate and ignore other foods; and it is very difficult to make the connection between excessive chocolate consumption and its undesirable consequences.

Use Cases have the same problems, caused by the same combination of characteristics:

- A set of Use Cases does not provide a system developer with all of the information that he needs about his client's needs, in order to produce a system that meets those needs. Use Cases are nutritionally deficient.
- Use Cases are extremely attractive. Much of the attraction is probably due to their simplicity. One doesn't have to work very hard to understand the basic Use Case concepts. With such a low effort, no-sweat requirements-gathering technique available, it is tempting indeed to believe that all one has to do when gathering requirements is to create a list of Use Cases.
- The use of Use Cases, to the complete or virtual exclusion of other requirements-gathering techniques, has undesirable consequences, in the poor quality of the systems developed. But these consequences -- the connection between requirements gathering and the eventual quality of the system as built -- are largely invisible to both developers and developer management.

In fairness, it should be noted that the Use Case approach is not the culprit here. The connection between the quality of the requirements-gathering for a system, and the quality of the system that is eventually developed, seems to be invisible to most organizations regardless of whatever particular requirements-gathering methodology the organization is officially practicing. That's

part of the reason why we -- as a profession and as an industry -- continue to do such a poor job of requirements gathering, and continue to have so many system failures.

In short, the analog of the Chocoholic's Diet (in which the unfortunate dieter consumes only, or mostly, chocolate) is what we might call the Use Case Approach (UCA) in which the unfortunate software development organization uses only, or mainly, Use Cases to gather and document client requirements. We have been warned for years, by everyone from the government to our own doctors, about the dangers of junk food diets, of which the Chocoholic's Diet is an example. Now it is time to issue a warning about the dangers of junk requirements-gathering methodologies, of which the Use Case Approach is an example.

The time has come to issue such a warning because the Use Case Approach is quickly gaining in popularity in software development shops. In most cases, UCA is riding on the coat-tails of UML. There are some UML proponents who advocate the requirements-gathering equivalent of a balanced diet: a requirements-gathering methodology in which Use Cases figure as only one element of a well-rounded diet of requirements-gathering techniques. But for every such reasonable guru, there is at least one snake-oil salesman pushing Use Cases as The Solution For All Your Requirements Methodology Needs. Some of these gentlemen give lip service to the idea of a well-balance diet, but it is only lip service. In their methodology books and courses, they pause briefly to give a cursory nod to (for instance) describing the problem domain, before spending the overwhelming bulk of discussion on Use Cases.

Unfortunately, the software development community seems to be eating it up. Many organizations see UCA as a silver bullet for their biggest requirements gathering problem... which, incidentally, seems often not to be How do you gather requirements effectively?, but What do you say when somebody asks you what your official requirements gathering methodology is? It used to be said that no one ever lost their job by buying IBM; today, no one ever loses their job by adopting Use Cases and UML.

This stampede of enthusiasm has produced the software development equivalent of a national health crisis. Just as the popularity of the "Sugar Buster's Diet" has forced doctors and dietitians to issue warnings about the nutritional deficiencies of such a diet, the popularity of the Use Case Approach makes it imperative to point out the deficiencies of Use Cases as a requirements-gathering tool. That's the purpose of this article.

Use Cases Unsupported by Domain Descriptions Are Vague

When we embark on a system development project, the system to be developed is the putative solution to some problem in the user's environment (the "application domain").

Naturally, we would expect the problem-solving method to look something like this:

- (1) Study the problem until you are confident that you understand it.
- (2) Describe a proposed solution for the problem. (In our field, the description of the proposed solution is a requirements specification document for a computer system.)
- (3) Implement the solution. (That is, build the system.)

A Use Case — as a description of an actor's interaction with the system-to-be — is both a description of the system's user interface and an indirect description of some function that the system will provide. In short, as descriptions of the system-to-be, Use Cases belong in step 2 --

describing the proposed solution to the problem. So the development of Use Cases has a place in the problem solving process... but that place is not as the first step, and it is not as the only step.

The first activity in the requirements-gathering process must be the study and description of the problem-environment, the application domain. To put it bluntly, the requirements analyst's first job is to study and understand the problem, not to jump right in and start proposing a solution. (This is a major theme of Michael Jackson's 1995 book, *System Requirements and Specifications* and his new book, *Problem Frames*. In *System Requirements and Specifications*, see especially the entry on "The Problem Context".)

One way that the problem domain can be described is by creating a model. The developer begins by creating a model of the "real world", i.e. the part of the real world that is relevant to the problem at hand, the part of reality with which the system is concerned and which furnishes its subject matter.

We start by modeling the real world (rather than describing the functionality that we wish the system-to-be to provide) because the model supplies essential components that we need in order to create our descriptions of the system functionality. In describing a university library application, for example, in the real-world model we would describe books and copies of books, describe what counts as being a university member for the purposes of using the library, and so on. In creating the domain model we, in a sense, construct a dictionary of words. We can then use those words when we write our descriptions of the functions and Use Cases that we wish the system to support. In the case of the university library, once we have described the objects in the model, we can specify any system functions that can be described using a vocabulary of "books", "copies of books", and "university members".

Note that the scope of the vocabulary that was created in the domain model implicitly defines (and defines the limits of) a set of possible system functions. [Jackson, 1983, p. 64] We can specify any function that can be described using the vocabulary of words that appear in the model/dictionary, but we cannot specify a function if its description would require terms that are not in the model. For instance, we can describe the process of a member borrowing a book and the process of returning it, the process of reserving a book and the process of notifying a member when a reserved book becomes available for checkout, the process of sending out overdue notices, and so on. But we cannot describe a Use Case in which a university member presses a button that triggers the rocket launch of a weather satellite into orbit. "Rocket", "satellite", and "launch" were not part of the conceptual vocabulary that was created in our domain description of the university library.

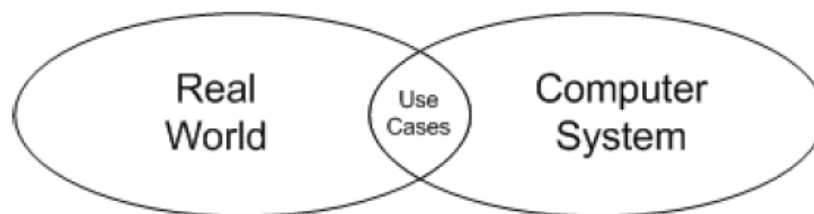
Note that the model of reality isn't just something that is nice to have in order to support the descriptions of the Use Cases. It is an essential foundation for those descriptions. If we produce just the Use Cases without first creating the description of the problem domain, then the descriptions of the Use Cases are fundamentally flawed by using undefined terms. A Use Case description for Member Checks Out a Book will use the terms "library member" and "book", but if those terms have not been defined earlier (in the model of the application domain), then the Use Case specification is necessarily vague (i.e. not clearly defined).

Consider the term "book". In this context, the term "book" is ambiguous between book as a work of art, which has no physical location, and book as a physical object (a "copy of a book") that does. Only after we have disambiguated the word "book", can we explain, for instance, why the

book involved in the Member Checks Out Book Use Case is not the same as the book in the Member Reserves Book Use Case. I often use the university library as a teaching example in my data modeling classes, just because of this ambiguity with the word "book". It exposes the students to the issue of ambiguity in domain descriptions, and helps me make the point that one of the requirements analyst's most important tasks is the detection and removal of such ambiguities.

Such ambiguities are quite real, and quite common. By now, you would expect that we would have recognized that fact, and have learned to deal with it. Yet one of the most common causes for major project problems is the failure of developers and their clients seriously to consider that ambiguity might exist in their requirements documents. Clients and developers tend to think that their primary job on the project is to describe the "requirements" for the system (that is, to describe what the system is supposed to do, the solution to the problem), not to describe the problem or the problem domain. Project participants often simply assume that the meanings of familiar terms are so clear and so well known to everyone present, that no explicit definition is necessary. This assumption is often false, and when it is false, the consequence is that significant ambiguities in the system specification remain hidden until they emerge and plague the later stages of the development project. In a case that I heard about recently, a requirements analyst was working on a project for a large American railroad corporation. He was having problems capturing the requirements because his users did not use the word "train" consistently. To some of them, a "train" was a particular collection of rolling stock (a locomotive and all the cars it pulled). To others, a "train" was just the locomotive. To others, a "train" was a regularly scheduled run, as in "I'll catch the 6 o'clock train to Boston". To others, a "train" was a specific instance of a regularly scheduled run, so that the train that left for Boston today at 6 o'clock, and the train that left for Boston yesterday at the same time, are two different trains. And so on. In this case, fortunately, the ambiguities were so glaring that they could not be ignored. On many projects the ambiguities are not so intrusive, so they are left hidden, like ticking time bombs.

Use Cases Do Not Capture Important Information about the Problem Context
Another reason that we start development with capturing information about the real world is there are properties of the real world that constrain the system, or that the system must know about, or that the system relies on, in order to satisfy the customer's requirements. Here, the downside of the Use-Case approach is that it draws the requirements analyst's attention away from the task of describing properties of the real world, and focuses his attention on the narrow area where the real world interacts directly with the system.



In embedded systems, the system's reliance on properties of the surrounding real world are very real, and can often be safety-critical. In *Software Requirements and Specifications* (entry "Requirements") Michael Jackson describes an incident in which an airplane overshot the runway when attempting to land. The runway was wet, and the plane's wheels were aquaplaning

instead of turning. The plane's guidance system thought, in effect, "I'm on the ground, but my wheels aren't turning. So I must not be moving," and would not allow the pilot to engage reverse thrust. Aquaplaning, a very relevant property of the real world, was not considered by the developers when they created the guidance system. The consequence is a plane in a ditch past the end of the runway, instead of safely docked in the terminal. (Jackson says that the error, which could have been catastrophic, fortunately was not.) For computer professionals, it is tempting to blame such problems on the clients. Knowing about aquaplaning, we are tempted to say, is our client's problem. Our only job is to figure out how to make the system do what they tell us they want it to do, in their Use Case descriptions. But even if our clients will let us get away with wiggling out of all responsibility for planes in ditches (something that seems extremely unlikely), this still won't cut it. As Jackson points out in a recent paper, it is almost impossible for software developers to build correct software if they don't understand the problem domain, and how what they are doing relates to what happens there.

One potentially attractive view, is that the concerns of computer science are bounded by the interface between the computer and the world outside it.... So if we restrict our concerns to the behaviour of the computer itself we can set aside the disagreeably complex and informal nature of the problem world. It is somebody else's task to grapple with that.

...

Unfortunately, ... the specification of computer behaviour at the interface, taken in isolation, is likely to be a description of arbitrary and therefore unintelligible behaviour. ... Practising programmers who try to adhere to this doctrine will find themselves devoting their skills to tasks that seem at best arbitrary and at worst senseless. [Italics mine -- SRF]

-- Michael Jackson, "The Real World"

Rephrasing this point in terms of Use Cases: it is almost impossible for programmers to build correct software if they are given Use Case information but no information about the problem domain and how their program relates to what happens there. Yet this is what the Use Case Approach encourages requirements analysts to do. This is the real problem with the Use Case Approach. It discourages the requirements analyst from examining the problem domain, by focusing attention only on what happens at the system boundary.

For Some Jobs, Use Cases Are Just the Wrong Tool

For some applications, there clearly is domain information that must be specified in the system requirements, but which has no natural home in any particular Use Case. Often, for example, an important requirement for a system is that it enforce (or at least not violate) a set of business rules or governmental regulations. Other systems (e.g. systems in the physical and social sciences) are heavily algorithm-driven. The Use Case Approach provides no natural mechanism for capturing such mathematical algorithms, business rules, and government regulations. Certainly, it is very unnatural to embed them in the descriptions of specific Use Cases. Specifying a single big module that contains all of the customer's business rules and (in UML terminology) extends every Use Case, is possible but clumsy and unnatural. Use Cases are simply not the best tool for capturing such requirements.

Nobody Really Knows What a Use Case Description Looks Like

Nobody really knows what a Use Case description looks like. Use Cases can be written at a very high level of detail, or at a very low level, or anywhere in between.

In some organizations, Use Cases may be written at a very high level. Use Case descriptions that are written at too-high a level are often useless. Sometimes, they are worse than useless, because they give the impression that the system requirements have been completely specified, when in fact that is not true at all. A recent case in point was a project to develop a securities information system. The customers knew that they wanted the system to generate reports, so the system specification included a Use Case to Run Reports. The problems with this Use Case didn't emerge until the requirements analyst began to ask the customers about the contents of the reports that they wanted to generate. Then it emerged that the customer had in mind 140 reports of radically varying content, and the real requirement for the system -- what the customer really needed -- was a system that could store a history of all of the kinds of events that could affect securities. Virtually all of the system complexity, and all the information needed to design the system, was hidden under the single Use Case Run Reports.

In other organizations, Use Cases may be written at a detailed, implementation-specific level, describing the mechanics of the graphical user interface (GUI), complete with buttons, menus, and drop-down list boxes. Often, at a stage in the project when the primary concern should be understanding the business context and functions that the system must support, the client is instead engaging in premature user-interface design.

In addition, as the editor of www.uidesign.net observed, in many organizations the clients or endusers are the ones who write the Use Case descriptions. But without any training in user-interface design

... there is little hope that [the end-users] will make a good job of it. ... The adoption of Rational Unified Process in its complete form is likely to set the development of good User Interface Design back by perhaps 20 years.

-- http://www.uidesign.net/1999/imho/feb_imho.html

The result, in short, is not only that the user-interface design is being done at the wrong time, it is being done by the wrong people.

Use Cases Are Unsystematic

An important feature of any useful requirements-gathering methodology is that it provides a systematic approach to identifying all of the system requirements. A requirements-gathering methodology that consists of writing Use Cases is no methodology at all, because it provides no help in systematically addressing the problem. The only guidance that the Use Case approach provides is the most generic question possible: "What would you like to do with the system, and how would you like it to behave?"

As Ben Kovitz points out, writing software requirements by writing Use Cases as they come to mind, is the requirements-writing equivalent of programming by hacking around. What it produces is simply a sprawl of Use Cases. How can you tell if you've identified all of the Use Cases for the system? How can you tell if the Use Cases conflict? How can you tell if the Use Cases leave any gaps? It is not possible for an ordinary human to understand all of the ways that a grab-bag of 50, or 100, or 200 Use Cases are inter-related. Suppose the customer wants the

system to support some new functionality. How can you tell which, if any, Use Cases will be affected by the change? There simply are no answers to these questions.

In short, as a methodology the Use Case Approach is too mushy to provide any real guidance in gathering requirements. The Use Case Approach, in fact, is not a methodology at all; it is merely a notation. Once an organization has settled on the template that it wants to use for describing Use Cases, that's it. At that point, the organization has got all the methodology help that it is going to get out of the Use Case Approach.

Use Cases Are Not Object-Oriented

There is really nothing at all that makes the Use Case Approach especially "object-oriented". Confined to describing behavior at the system boundary, a set of Use Cases describes neither objects in the real world nor objects inside the computer.

Because the Use Case Approach is not object-oriented, it is completely compatible with non-object oriented methods. As Grady Booch observed

[A] very real danger is that your development team will take [its] object-oriented scenarios and turn them into a very nonobject-oriented implementation. In more than one case, I've seen a project start with scenarios, only to end up with a functional architecture, because they treated each scenario independently, creating code for each thread of control represented by each scenario.

--entry "Scenarios" in [Booch, 1998]

There is nothing wrong with not being object-oriented, of course. If a technique is useful, it is useful, object-oriented or not. But given the popularity of the buzzword "object-oriented", I think it is important to point out that the object-oriented-ness of Use Cases is a myth. Use Cases are enjoying their current popularity because the Three Amigos bundled them into UML along with other, truly object-oriented, techniques. Whatever object-oriented-ness Use Cases have, they have acquired solely by rubbing up against true object-oriented methodologies into whose company they have fortuitously fallen.

The Proper Use of Use Cases

The best metaphor for methodology skills is a handyman's toolbox. No handyman could be successful if he relied on a single tool (a hammer, say) for everything that he did. Hammers don't work well when you're trying to drive a screw, nor when you need to cut a plank. You need different tools for different jobs.

A software developer is a software handyman. To do everything that he needs to do, a software developer needs a toolbox that contains a variety of tools. Use Case description shouldn't be the only tool in the toolbox. It should simply be there along with the other tools, so that it is available when it is the right tool for the job.

Most of the problems with Use Cases that we've discussed, like the problems with chocolate, come not from the thing itself, but from its improper or excessive use. In spite of all of the problems we've discussed, Use Cases can be useful — when they are used properly. Most importantly, Use Cases can be an effective tool when they are developed in a disciplined manner, as part of a methodology that first creates a well-defined domain model.

The domain model provides an infrastructure for the requirements-gathering process, so that the development of Use Cases can proceed systematically, in a way that could never happen without the domain model. The domain model describes the objects in the problem domain, including the events in the lives of objects. Once the events in the lives of the objects have been described, the requirements analyst can approach the task of writing Use Cases in a systematic fashion, by writing a Use Case for each of the events. If the life of a library book includes events such as being acquired, being borrowed, being returned, etc., then the analyst will develop a Use Case for each of those events. Each Use Case answers questions derived from specific events: "How does the system get told that a book has been returned?" or "How does the system know when to generate overdue-notice letters?"

In actual practice, of course, deriving Use Cases from the domain model is rarely as simple and mechanical as we've just described. But there are techniques for dealing with more intricate situations. The whole purpose of systems analysis methodologies, in fact, is to provide guidance for situations in which the process is not so simple and mechanical. But whether the process of developing a particular Use Case is simple or tricky, its foundation must always lie in a solid understanding of the problem, and a carefully-constructed description of the problem domain.

In summary, a set of Use Cases is a description of the system to be constructed, the thing to be built, the solution to the problem. But, as Michael Jackson points out, before you can effectively start building the solution to a problem, "first you must concentrate your attention on the application domain to learn what the problem is about." [Jackson, 1995, p. 158]

Acknowledgements

Several of the ideas in this paper are derived from the works of Michael Jackson, and from Ben Kovitz's short discussion of Use Cases in *Practical Software Requirements* (pp. 251-252). I hope that Ben and Michael will consider theft the sincerest form of flattery. Neither, of course, is responsible for any mistakes that I might have made in presenting their ideas, or for cases in which my opinion or terminology differs from theirs.

References

- [Booch, 1998] Grady Booch, *Best of Booch* (ed. Ed Eykholt), Cambridge University Press, 1998
- [Jackson, 1983] Michael Jackson, *System Development*, ACM Press, 1983
- [Jackson, 1995] Michael Jackson, *Software Requirements and Specifications*, Addison-Wesley, 1995
- [Jackson, 2000] Michael Jackson, "The Real World" in *Millennial Perspectives in Computer Science: Proceedings of the 1999 Oxford-Microsoft Symposium in Honour of Sir Anthony Hoare* (ed. Jim Davies, Bill Roscoe, Jim Woodcock), Palgrave, 2000
- [Kovitz, 1999] Benjamin L. Kovitz, *Practical Software Requirements*, 1999

Chapter 7 User Stories

McFADDEN, R. (2017). SOFTWARE REQUIREMENTS – USER STORIES, UNDER CONTRACT FROM UMGC.

Introduction

It is a well-known fact that many software projects fail due to badly written requirements whether because they are ambiguous, inconsistent, incomplete, or incorrect. One approach used by both traditional and agile methodologies is to write requirements from the user perspective. This user-centered approach helps in making sure that the requirements contain what the customer and/or user really needs in the system and that there is less ambiguity and guesswork on the part of the developer. It also makes sure that requirements are written from the view of what the system should do (functionality) as opposed to how it will do it and that they are written in narrative textual form that is easy to understand and not in information technology (IT) jargon. In addition that are additional graphical documents that help visual the requirements better. In turn this improves the communication between customers and developers and others involved in the project, gives a better understanding of the user interaction with the system, and lends itself perfectly to writing documentation and test cases. In this chapter we will explore a user-centered approach to writing requirements using user stories, storyboards, and user journeys.

Basic Terminology

Software Requirement

Software requirement is a functional or non-functional need for a particular system. The format of a written requirement and the details provided differ but it includes information related to attributes, capability, characteristic, and/or quality of a system and it represents a solution to meet the objective of the system to be developed. Traditional requirements concentrate on how the system should act and work and it guides the developers in what they should implement.

Use Case

Use case represents a requirement in the form of a list of actions or events from the view of a role, such as end-user, and the interaction that role has with the system. The level of detail in a use case will differ but at minimum it includes the primary role, goal or scope of the requirement, and the description of the actions and/or events.

Use cases often have accompanying use case diagrams usually using the Unified Modeling Language (UML) format to help the customers and the developers better visualize the particular scenario. Other UML diagram may also be included to provide additional information and understanding of the requirement such as activity diagrams, sequence diagrams, communication diagrams, or state machine diagrams.

Use cases are used in both the traditional and agile development approaches.

User Story

User stories are also written from the perspective of the user and represent a description of a feature. They are informal and significantly less descriptive than traditional requirements or even use cases.

Just like use cases they are written in narrative textual form using everyday language but they concentrate on the needed feature unlike the use case which focuses on the user's interaction with the system. User story's format is much more informal and in fact can be simply stated as:

As a <type of user>, I want <some goal> so that <some reason>

Epic

Epics represents a large user story that is broken down into smaller user stories. How big will depend on an organization but if a user story can be broken down into five separate user stories, it is big enough to be an epic.

An epic description is usually very broad and does not have many details. It is often used as a placeholder with a few lines of description. It may represent a feature or customer request, or some business requirement.

Related epics are often grouped into themes.

Story map

Story maps are often used with user stories to provide the graphical, two-dimensional visualization of the product backlog. The top of the map has headings for the epics (or themes or activities) and then underneath each epic (vertically) there are boxes with each user story ordered by priority.

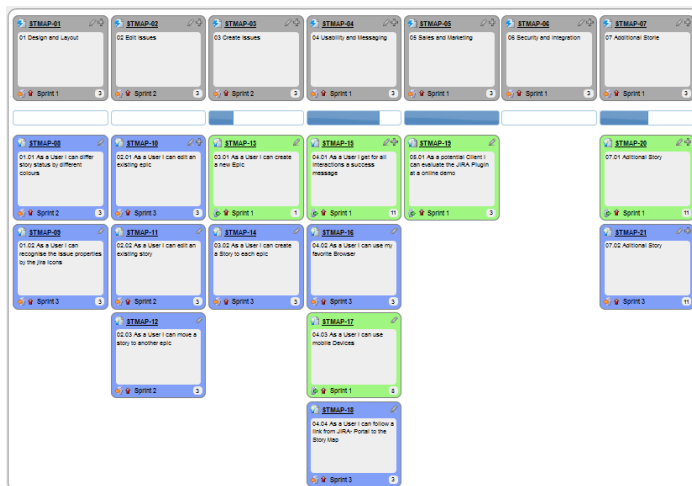


Image 1: A story map in action

Technical User Story

Technical user stories focus on non-functional requirements such as security, performance, scalability, serviceability, etc. or some support of base functional feature. As a result, the “user” part of the user story may not be appropriate. This type of stories, sometimes called infrastructural stories, are more technically oriented and require more skill to write than a typical user story written by a user to identify a feature.

Technical user stories would normally be written by product owner or by the project team during brainstorming activity. They would address a number of areas such as (a) product infrastructure – stories to support functional user stories; (b) team infrastructure – stories to support the team in developing the software such as tooling, testing, design, and planning; (c) refactoring – stories to identify refactoring candidates whether code, tooling, automation, etc.; (d) bug fixing – stories to identify clusters of packages of bugs; and (e) spikes – research stories that eventually result in additional stories to support functionality.

Storyboard

Storyboard relative to software requirements is a graphical illustration or a sequence of images that demonstrate the important steps of the user experience. It looks like comic strip squares with each square representing a step in the flow of the feature.

The storyboard is reviewed together with customer/user and developers and changes are made as needed to clarify the customer's/user's needs. The graphical aspect of the storyboard helps to better clarify the requirements than just a written description.

A storyboard can be used for an epic or individual user story to flush out and illustrate the details of the feature as it relates to the user experience. It can also be used for a user journey. This approach is especially useful for web pages and user interfaces in general because it can cheaply demonstrate the look and feel and flow across screens.



Image 2: Storyboard for: The Radio Adventures of Dr. Floyd #408

User Journey

User journey describes a user's experience, often when interacting with web page or other user interface using a series of steps (i.e. 4-12) to complete a specific task. It is focused on the user's experience such as what they can see and what they can click or select. For a website, it can also be described as a user's path through the site. For example, a user may be a patient who starts his user journey by scheduling an appointment to have blood tests done and ends the journey with getting his results accessible online.

User journey describes a specific user's experience, whether a role or persona, and different users will have different user journeys. User journey is usually textual and is often fed into storyboard or user journey map (visual representation).

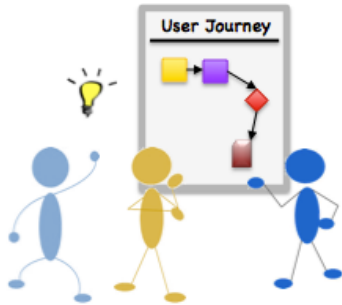


Image 3: User Journey discussion

History

User Stories approach originated in 1998 as part of Extreme Programming methodology and was described as being “like use cases” and used as part of the planning game. However, at the time, a user story was just a few words written on a card that was meant as a starting point. Something a developer could take to the customer and start the conversation about what was needed. In fact, Alistair Cockburn, one of initiators of agile movement, is said to have coined the phrase “A user story is a promise for a conversation.” (Cockburn, 2017).

Over time the concept evolved and spread to all of agile methodology and beyond. In 2001 a “role-feature-reason” template was developed (As a *<type of user>*, I want *<some goal>* so that *<some reason>*) which is still commonly used for the user stories.

Also in 2001, Ron Jeffries, one of founders of Extreme Programming, proposed “Card, Conversation, Confirmation” formula, also called 3 Cs model, to capture the components of a user story. The formula was meant to make it clear that user stories were in fact not like use cases.

The “Card” was the physical card or post-it note with the basic concept. The “Conversation” stood for the discussion to take place between the customers, developers, testers, etc. whether just verbal or supplemented by documentation. Lastly the “Confirmation” stood for reaching the objectives of the conversation

Storyboards were originally developed at Walt Disney Studios in the 1930s as comic-like story sketches to be used for animated cartoons. As animation and film started to use more technology, the storyboards became more sophisticated and spread to other areas such as software development. Then as usage of user stories became more popular way of developing requirements, storyboarding was used to provide a visual illustration of the interaction and flow of the system.

User journey is a relative new concept although it builds on many years of customer-oriented and user-oriented approach to products and business especially e-commerce. Often the “user journey” is used interchangeably as “customer journey” although in software engineering they

are not the same. Customer in general is the person paying for the software while user is the person who will use the software. So for example, a university may buy software for online classroom so the university is the customer and the “person” who selects the software may be the university board. The users of the system, the end-users, however are the students, faculty, and the administrative support for the system.

Relative to journey, many argue that there does not seem to be clearly defined difference between user and customer journey especially that in many cases a customer (buyer) is also a user. However, there has been an attempt by some to define the differences. A recent blog by Tom DiScipio defined Customer/Buyer journey as “instinctive process of a buyer to research, consider, and make a decision to purchase the best solution to solve their challenge” and User Journey as “strategically constructed, interactive experience designed to guide someone to their decision.”

He then explains that Buyer’s Journey can be traced back to marketing, who would build Buyer Personas to determine how buyers become aware of some product, what they use to consider it and then finally how they make their decisions. User Journey on the other hand started in web design to improve the user interface and user experience. A User Journey describes the steps and actions a user would take to accomplish some specific tasks. So while marketers will also use User Journeys, they are heavily used by software engineers to design a more usable and attractive user interface and user experience of the product.

How does it work

User Story and Storyboard

Creation

User stories are created by end users, customers, product owners, developers, testers, and etc. for some new functionality/capability to be added to the product. They can be written on an index card (or sticky note) whether paper or digital. Different organizations will have different approaches and use different tools for storing user stories but in more recent years they have tendency to be digital rather than paper.

They are meant to be a starting point for a discussion so they are rather simple and informal. The scope however can be very large (weeks or months of work needed) or very small (one day of work). In some organizations, an epic will be the user story that originally is submitted identifying some functionality (assuming that it’s for a bigger scope) and then user stories are created by breaking it down into tasks, smallest unit of work that developers will implement. Each user story still represents a requirement and functionality but its scope is relatively small such as person-days. Some describe user stories as vertical slices of functionality.

Epics and user stories are prioritized and given approximate estimation of work needed. When project starts, the highest priority epics and user stories are worked on first and new user stories are created and prioritized as the project progresses. A typical agile project will last about six months although that will depend on the organization and the product. The user stories then represent the product backlog.

A blog posting by Mike Cohn (2012, Dec 17) gives a couple real life examples of how epics can be broken down into user stories. So for example, the original user story (an epic) stated:

“As a hotel operator, I want to set the optimal rate for rooms in my hotel.”

Then the development team worked on breaking it down into smaller user stories ending up for example with user stories such as:

“As a hotel operator, I can define a 'Comparable Set' of hotels.”

“As a hotel operator, I can add a hotel to a Comparable Set.”

“As a hotel operator, I can remove a hotel from a Comparable Set.”

And so forth where “Comparable Set” was a term used widely in the company for the comparable hotels that could be used for comparing rate in order to set the optimal rate. At this point each user story was small enough in scope to complete within a few days or less.

Adding Details

There are three basic ways of adding details to the user stories: (1) break down into smaller user stories, (2) add “condition of satisfaction” also called “acceptance conditions” or “confirmation”, or “acceptance criteria” or acceptance tests and (3) graphic representation. Usually a combination or all of these are used.

In the above example for the optimal rate for hotel room, the team first came up with one of the user stories as: “As a hotel operator, I want to set the optimal rate for rooms based on what hotels comparable to mine are charging.” Then when it broke down this story into smaller user stories, it provided more detail. Satisfying this user story meant to define a “Comparable Set”, ability to add a hotel to that set, remove a hotel from the set, delete the “Comparable Set”, and so forth. If these stories were to be broken down even further, they would provide additional detail.

In addition to the smaller scope user stories expressing the functionality and user experience, additional detail is provided through adding technical user stories which provide the system perspective. They are created as part of planning and drill down into the user story and usually are created by product owner and/or development team. These stories complement the user stories and provide additional details of how the functionality would work that user would not directly be concerned about.

For example, a user story may be stated as: “As an online shopper at ACME Widget Company, I want to ensure that my order is complete and valid when I submit it at the online store, so that I can get the products I ordered in a timely manner without mistakes or delays” (Badri)

The corresponding technical user story may be “In Order to ensure that only valid orders are accepted by the system, System A must map the Order XML to the main canonical and provide the main canonical to SYSTEM B to run order validation rules.” (Badri)

There would normally be a number of technical user stories associated with a single user story and they can get as granular and detailed as needed.

The conditions of satisfaction or acceptance criteria are written for both the user stories and the technical user stories. They provide additional detail and can be used for defining test cases. So for example, for the above user story, there can be acceptance criteria such as:

AC1 “User must get a message if the order is valid that it has been accepted for processing” (Badri)

AC2 “User must get an error message if the order is invalid with prompts on the issues that must be fixed before it is accepted for processing.” (Badri)

When user stories are written on an index card, it is common for one side to express the user story and the other side to define the conditions of satisfaction (high-level acceptance test) that again provides the additional details.

For example, we may have a user story:

“As a VP Marketing, I want to see information on TV ads when reviewing historical campaigns so that I can identify and repeat profitable campaigns.” (Cohn, 2012).

And the conditions of satisfaction may state:

“See how many viewers by age range.

See how many viewers by income level.” (Cohn, 2012).

Graphical representation of the user story can also add additional detail and clarification of the requirement and illustrate the user story. Storyboards are frequently used for this purpose. A storyboard contains a sequence of wireframes (series of screens) which provide the basic screen flow, key components of the feature, and the look and feel of the user interaction.

However, they are not meant to have all the details of the feature and the user interface. Instead they are a visual aid and can be thought of as cheap and easy to change prototypes. They can be created on a piece of paper or using a number of software tools that help with the process. The user story already provides who, what, and why that can be the starting point for the storyboard. In addition, one would consider the trigger (how does the scenario initiate), action (how is the action executed), and reward (what is accomplished for the user) in order to draft the storyboard.

So for example, using the above user story “As a hotel operator, I can remove a hotel from a Comparable Set”, the starting point may be the screen showing the comparable set and then the screens for removing a hotel:

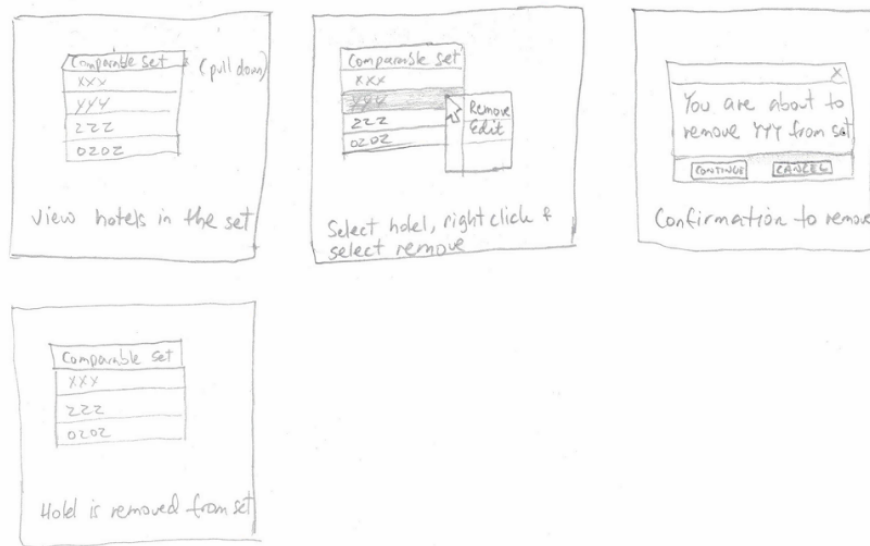


Image 4: Storyboard example

Storyboards describe a scenario such as the user story and usually contain the sketch of the visual elements, descriptions of animation, description of interactions, description of sounds and any other media used. Thus they help in the discussion with the user/customer to clarify the requirement and provide a quick reference during the development.

How to write good user stories

Although a user story itself is simple and brief, there are still a number of characteristics it should have to make it well-formed. There is a well-known model called INVEST created by Bill Wake that gives a number of criteria for a user story:

I – Independent – should not overlap with other user stories such that they can be scheduled and implemented in any sequence

N – Negotiable – it captures the essence and not the details of the requirement and it is flexible rather than a contract. Over time the user story is given more details and acceptance criteria but those details are not necessary initially to prioritize and schedule the story

V – Valuable – it needs to be valuable to the user so it needs to be from the user's perspective and satisfy user's need. It is not about what the developer wants or needs but rather what the user needs

E – Estimable – it needs to be understood well enough and small enough in scope so that it can be estimated. If it cannot be estimated it does not belong in the product backlog and cannot be developed.

S – Small – smaller user stories have more accurate estimate and are better understood. Typically that means just a few person-days of work.

T – Testable – if a user story cannot be tested than it is never completed. Basic tests or at least acceptance criteria should be identified before implementation starts. It verifies that the user story is well understood.

There are some other considerations to keep in mind when writing user stories. Ideally they are originally written by the end-users who are the domain experts and from their perspective. However, if a product owner or developer writes them, he/she needs to know who the user is and have excellent understanding why and how a user will be using the product. That means observing and interviewing users and possibly developing a number of personas.

Personas are generalized characters of a target group for the product. They usually contain a picture and a name, such as for example “Joe, the Senior Citizen” and all the relevant characteristics, behaviors, attitudes, and goals of that persona relative to the use of the product. Only by truly understanding the user’s needs can the developer form the right user stories.

Again, user stories are just a starting point for a conversation between users and developers. They concentrate on who, what, and why of the feature but not how. That means that after a user story is written and before any implementation starts, there should be a discussion between users, product owner, and development team to work out the details. That includes breaking the user story into smaller stories as needed, and updating or changing the user story. User stories need to be refined until they provide the needed detail, are clear to understand, are feasible, and testable.

That also means working out all the supplemental artifacts such as storyboard and acceptance criteria. The storyboard provides the visual representation with additional details that aid in the discussion and the acceptance criteria describe what needs to be fulfilled for the user story to be complete. Thus they enrich the story and make it testable. In general there should be at least three to five acceptance criteria for a user story.

User stories should not use technical jargon and they should neither be too long or too short. They need to express a functionality from the user’s point of view and each user story must be complete and testable. If a user story cannot be tested than it is not a valid user story.

For example, user story “As the HR Manager, I want to properly screen engineering candidates so my company can get the best possible hiring outcomes” is too broad and does not have enough details to be implemented or tested. A more specific user story may be stated as “As the HR Manager, I want to screen an engineering candidate, so I know their skills profile.” That is still pretty broad however, because what does it mean to “screen”? So instead we may have “As the HR Manager, I want to create a screening quiz so, so I know their skills profile.” This gives us more detail for the feature but what about the goal/reward? That is still too vague and not testable. So we may have “As the HR manager, I want to create a screening quiz so that I make sure I’m prepared to use it when I interview job candidates.” Now it has detail and is testable. (Cowan, Your Best Agile...)

In order to form the basic user story for the typical format:

As a <type of user>, I want <some goal/do something> so that <some reason/realize a reward>

The following questions should be considered and answered:

“Who is this user?”

What makes them tick?

Who's an example of such a person?

Why do they want to do this?

What's the benefit/reward?

How will we know if it's working?" (Cowan, Your Best Agile...)

Without knowing the answers to all of these questions, you cannot write a well-formed and effective user story.

Experienced developers know that often users are not very good expressing what they really need. They may get caught up on some details on what they want and not express something critical for the system that they really need. On the other hand, developers are known for making too many assumptions and getting too caught up in wanting to deliver something fancy that is not in the best interest of the user.

One of the most common mistakes that developers make is to treat the product like a toy and provide multiple ways of accomplishing the same thing. Yet most users just want a simple single way to accomplish what they need. After all, the product is just a tool and not the end by itself. For example, when a receptionist needs to schedule an appointment for a client, her need is to have that appointment scheduled and tracked easily. But to a developer the scheduling product is the end in itself. Thus by nature of their jobs, the user and developer have very different perspective.

That is why, a user story needs to be from the user's perspective and it has to clearly specify not just what the feature must have but also what the end goal and reward is. Then one should consider what the value to the user is. If there is no additional value, then it is not a valid user story. There is no point in working on a solution for something that is not a problem.

Lastly, user stories need to have priority and estimate assigned by a developer before they can be added to the product backlog and they may address functional or non-functional requirements. They should also be assigned some unique identifier for reference and to have traceability between the user story and other supplemental documents and artifacts.

User Journey

User stories identify specific functionality to be added to the software that a user needs. However, they do not address the overall user's experience and the flow across the software. That is where user journeys can be beneficial.

User journey may be created for the existing software or for the one being developed and it concentrates on how the user interacts with the service or website or product. It is developed by a product owner or developers.

The first step in developing user journey is to understand users and/or to develop personas. Each user journey addresses a specific user or persona so you would normally develop a number of user journeys.

Some of the information that needs to be gathered about users include: users' goals and tasks that they want/need to achieve, their motivation for using the software, what problem do they have that needs to be solved (pain points), and the overall character of the users.

User Journey is written as a sequence of steps that reflect the user's experience to accomplish some specific task from the beginning to the end. It is the path that the user takes through the software such as website. It focuses on what the user can see and what they would do (i.e. click) as they interact with the software to reach their goal. For each step then there are a number of considerations such as: (1) current context – where is the user using the software (i.e. is the environment distracting in some way); (2) device – what device are they likely to use, how is software dependent on that device's features, and how skilled is the user with that device; (3) progression – how is each step enabling them to go to the next step and gets them closer to accomplishing the task; (4) functionality - what functionality are the users expecting and is that achievable; and (5) emotion – what is the users' emotional state at that step (i.e. excited, bored, annoyed). When the user journey is about an existing software, it should highlight the changes that are needed to solve the pain points. If the user journey is about new software, it should highlight the ideal solution and the benefits to both the user and the business (Mears, 2013).

Other things to consider when designing user journey are motivation (why user wants to interact with the product and why this product versus another), mental model (user's perceptions of the product and what concepts and connections come naturally versus need to be taught), and pain point or the challenges/obstacles the user is having that the product can help with or that the product is causing (Kaytes).

There is no specific template used for user journeys but they focus on a specific goal/task and the flow as that task is being accomplished. So a step may be a page or a process during a single "visit". The user journey may take just a few seconds or minutes depending on the task. For example, a user journey may describe user searching for a product and then making a purchase or just searching and displaying movie show times.

Each user journey would typically include the user/persona, brief description of the goal or task, and then the steps. The steps should be a liner path, the simplest and shortest path, to reach the goal. How much detail is included for each step is up to the person developing the journey but it would usually describe the interaction (i.e. click), functionality being used (i.e. login), and could have the emotion (what user may be thinking and feeling).

Since user journey represents a particular solution for a task or problem, it may be beneficial to develop multiple user journeys for the same goal to explore the best user's experience. The first solution that a developer designed may not be the best one.

Lastly, the user journey may also be visually represented using storyboard which would aid in the discussion of the user journeys, how to improve them, and which should be implemented.

Templates and Examples

User Story

User story's basic format is simple and straight forward and is typically expressed as:

As a <type of user>, I want <some goal> so that <some reason>

Where *<type of user>* is a role or persona that represents the user. Often the persona is given a name and even a picture to humanize them. So for example we might have the persona “Jane the Student” and a user story may be stated as:

‘As Jane the Student, I want to view my assignment grades to determine my progress in the course.’

The goal in *<some goal>* refers to the functionality that the user needs to accomplish some task. In the example above, the functionality is being able to view assignment grades.

And lastly, *<some reason>* needs to represent some value that the functionality gives to the user and as such it needs to be testable. If there is no value or it cannot be tested, then it is not a valid user story.

A similar template can also be expressed as:

As a *<type of user>*, I want *<do something>* so that I *<realize a reward>*

Where the reward just means that the goal for the story was successfully accomplished. So in the Jane the student user story above, being able to view the assignment grades should allow the user determine how well they are doing in the class – such as passing or failing.

Another popular template introduced by Rachel Davies states the same concept in slightly different way (Pichler, 2016):

As *<user role>* I want *<what?>* so that *<why?>*

For example: “As a student I want to order official transcripts so I can apply for a job”

Rachel Davies and Tim McKinnon suggested below template which expresses the same user story as above templates (Marcano, 2011):

As *<some role>* I want *<some capability>* so that *<some benefit>*

In some cases the business value may want to be emphasized and the following template may be used instead:

In order to *<receive benefit>* as a *<role>*, I want *<goal/desire>*

Another template that may be used is modeled after the Five Ws (**Who** was involved, **What** happened, **When** did it take place, **Where** did it take place, and **Why** did it happen) expressed as:

As *<who>* *<when>* *<where>*, I *<what>* because *<why>*

Mishkin Berteig (2014) describes a slightly different template, developed at Capital One agile session in 2004, which focuses on the behavior of the system and indicates a real-world benefit that can be for other users and not just the user in the story:

As a <role> I can <action with system> so that <external benefit>

So for example he takes the below user story and splits it by process step:

Original User Story: “As a job seeker, I can post my resume to the website so that I have a better chance of being noticed by potential employers.”

Split user stories:

“As a job seeker, I can post my unreviewed resume to the website so that ...”

“As a site editor, I can approve an unreviewed resume so that ...”

What is important to note that while the templates may be expressed in slightly different ways, they all have the same basic components in common which include the user/role/persona, some action/functionality to be accomplished by the system for the user, and some benefit/reward that is testable and that the user will receive from doing the action.

For more user story examples, check out the below URL (toward the bottom of the article). The author not only gives excellent user stories but also gives the corresponding acceptance criteria for each: <http://www.ascendle.com/blog/writing-user-stories-its-not-as-difficult-as-you-think>

Also, you may want to check out some examples of not well formed user stories and explanation of what they are missing at <http://blogs.collab.net/agile/user-story-examples-and-counterexamples#.Wetzf4gpBhF>

Storyboard

Storyboards do not have a set format. They are basically boxes or images or also called wireframes that look like comic strip. They can be hand drawn or you may use tools to create them.

They should not take a lot of time to create. They are meant to provide graphical representation for user story (or user journey) and some additional detail about the user interface. They do not have all the details however and are not meant to be a complete design. They are supposed to aid in the conversation and to work out some of the details of what the user needs at a higher level.

Below are a couple of images of templates that have been used by some organizations that can help you get started. They show the boxes that would have drawings of the interface and additional notes below.

SCENE		PAGE	
SHOT #		SHOT #	
ACTION		ACTION	
DIALOGUE		DIALOGUE	
FX		FX	

Image 5: Storyboard template

Title:

Page:

Action	Dialogue
--------	----------

Action	Dialogue
--------	----------

Translation	Timing
-------------	--------

Translation	Timing
-------------	--------

Image 6: Storyboard template 2

Next image gives an example of a quick hand-drawn storyboard. As you can see, it gives you an idea of what a user would see and the flow the interface would follow. There are additional details that a user story would not have but again, storyboard is not a complete design. Some storyboards will get more detailed than others but they are not meant to show the final design. The design may change as the user story and storyboard are reviewed and as the development works through the details with the user.

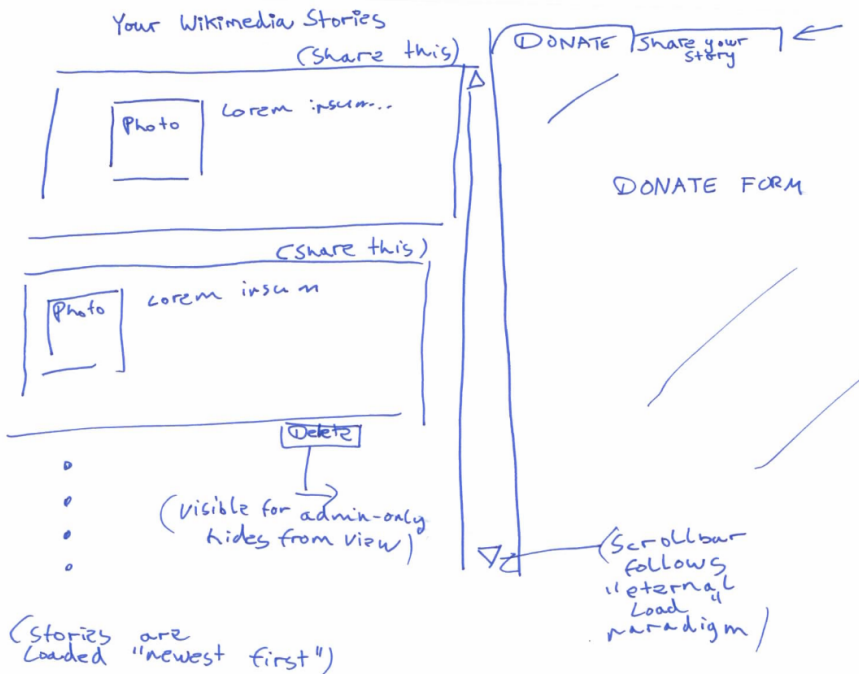


Image 7: Quick'n'dirty mock-up for Wikimedia "storyboard" extension to highlight donor/supporter stories

User Journey

User Journeys also do not have a specific format to be used and how they are designed will depend on the organization and what particular journey they are trying to represent.

For example, when considering user's interaction with a company (site or product), it is helpful to consider the stages that the user goes through such as: awareness (identify need), consideration (research), purchase, retention (stay loyal) and advocacy (share with others) as shown in image 8. In this image it just shows different media that a user may use or come in contact with but other user journeys may have short bullets discussing each stage.

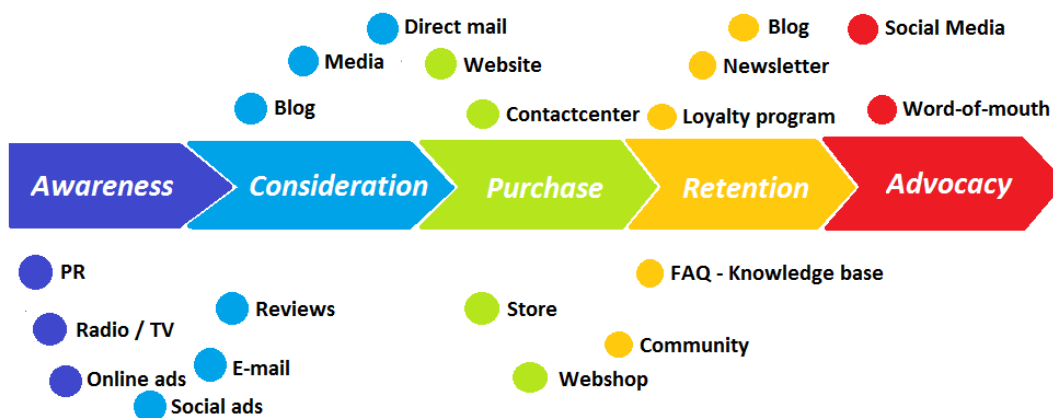


Image 8: Overflow of phases in the customer journey with media

The stages may also be presented as awareness, familiarity, consideration, purchase, and loyalty or awareness, consideration, conversion (expert validation or reviews), loyalty, and advocacy. They can also be simple awareness, consideration, purchase, and retention. There are many different variations of these that are used.

Each stage may have some categories such as “needs and questions” and “channels”. So for example under consideration there may be a price for “needs” and google search for “channel”.

Another way to represent a user journey is by simple boxes showing a sequence and bullets in each box as drafted in the image below. There might be some additional notes under each box or an image for emotion at that step.

So for example, a simple textual user journey for student submitting an assignment into online classroom may be:

Additional comments may be added for issues there might be with that step (pain points) or more detail on the action, and emotion or thinking the student may have at that step.

User Journeys are also often presented as a table with a sequence of steps as a timeline and additional information about each step such as pain points, touchpoint (what user does at that step), channel, and experience (negative, positive, or neutral and why).

Timeline	Step 1	Step 2	Step 3	...	Step n
Pain Point					
Touchpoint					
Channel					
Experience					

Other formats may also show persona information on the left and include rows for Feeling and Thinking as well, as shown in next image.

Timeline	Step 1	Step 2	Step 3	...	Step n
Pain Point					
Touchpoint					
Channel					
Experience					

See the following URLs for a collection of user journey examples: <http://blog.uxeria.com/en/10-most-interesting-examples-of-customer-journey-maps/> and <https://www.appcues.com/user-onboarding-academy/user-journey-map>

Pros and Cons

There are a lot of benefits to using user stories. Because they are written from the perspective of the user, there is a greater probability that they will capture the user's real need as opposed to development's assumption of a need. In addition, if used correctly, they initiate and promote the conversation with the user that might not happen using other methodology. And since the details are worked out throughout the development cycle and not locked down up front, they give the user the ability to make changes as the product design and development progresses.

Having user stories that describe a small functionality, makes it easier to change priority and reschedule which stories are being developed when. This decreases the chance that the development gets blocked from making progress because of some design or technology challenge.

Having exact acceptance criteria that is written for each user story helps development implement exactly what is needed, in the same way test-driven development does, and gives them clear indication of progress. It also helps test effort as the acceptance criteria feed easily into test cases. In the same way, the ongoing feedback through the discussions with the user provide the clarification and verification that the functionality is being developed correctly to meet the user's needs.

Traditional requirements have tendency to mix what the customer needs (what should be developed) with how it would be developed. As a result, time may be wasted on tangents and the user's need may not be explored well enough to make sure that it is understood clearly. User stories however, provide a clear separation because they only address the "what" which is where user's or product owner emphasis is and allow the "how" to be developed by the development in separate supplemental documents.

Traditional requirements have tendency to give a false feeling of having all the details. Yet since those details are written at the beginning of the development cycle, there is still a lot of unknowns and as the product gets developed, often there are many changes done without going back and changing the original documents. As a result, a lot of assumptions are made when the functionality is being developed. This is in contrast with user stories which by their nature require developers to discuss the user stories to be developed in near future with the users. At this point whatever has been developed already is well known and so the decisions whether for solution or details can be based on what really is as opposed to what was assumed to be at the beginning of the development.

User stories have the benefit of capturing a smaller scope of work and as such are easier to estimate the development effort. They also allow moving some of the functionality to the next release as needed as opposed to more traditional requirement methodologies which drive the all or nothing delivery because requirements are written and developed in large chunks.

Using user stories also has some drawbacks or limitations. First of all they are not meant to have all the details that a formal requirement or use case would have especially when written initially. They are meant as a starting point to be fine-tuned during the development cycle (as opposed to having the details up front) and require user's involvement. The level of the detail that a user story has should increase as it gets closer to the time of it being implemented and that means having conversations and discussions with the user. By definition therefore, if a user cannot be

engaged throughout the development process, user stories may not be the appropriate methodology to use.

Also, since user stories are written from the user's perspective they normally do not include non-functional details. Those details however are needed for a high quality product and require technical user stories to be written. If a team only concentrates on the user stories provided by the user, they may miss some important aspects of the functionality that should be developed.

Sometimes user stories may not be done correctly where they have either too much detail up front thus taking away some of the advantages and giving wrong assumption that a conversation with a user is not needed or they do not have enough detail and/or supporting documents.

Storyboards have the obvious advantage of providing a visual representation and some additional details that a user story (or user journey) may not have. As such they aid in the discussion of how the function would work and help in clarifying user's need and functionality. They also allow to easily compare some alternative solutions especially related to user interface.

It is hard to image having a drawback for using storyboard however, they do require some time to develop and as any tool they have to be used correctly. For example, they should not have too much detail to confuse the user and they cannot be assumed to be the final solution or design.

Lastly, where a user story concentrates on a specific functionality of the system, user journey describes a user's complete experience for a particular task or interaction with the system. In that way a user journey provides a higher level view and puts the functionality in perspective.

Therefore, user journeys can demonstrate a high level vision for the system being developed, provide a better understanding of the user's interaction with the system, help identify functional requirements that may be missing, and help clarify the system's flow and user interface (Mears, 2013).

User journeys however, require a thorough understanding of the user and their needs and that means significant time observing and interviewing users. They also require additional time to develop and an experienced developer to complete.

To conclude, user stories, storyboards, and user journeys are an excellent approach if they are used correctly and with the correct expectations but like any methodology they can backfire if not done right. When done correctly however, they provide a greater confidence that the product being developed is the right product and that it will meet the user's needs.

When to use user stories and when not

User stories are not necessarily the best choice for all products and/or requirements. For example, as already discussed, they require users' involvement throughout the development process and if that is not possible, then they may not be a good choice. This collaboration is often already part of the agile environment but not usually part of non-agile projects.

Even in agile environment, if the stakeholders are spread across different departments and/or different time zones and/or have conflicting interests, getting agreement may be difficult and using user stories may not be appropriate. In those cases it may be beneficial to spell out the details up front in use cases to help reach agreement than to have never ending conflict and re-work for each user story.

Having said that, it does not have to be either or situation. It is possible to use user story process to gather user requirements from user's perspective (personas and stories) with user's involvement but then use that information to draft the more traditional documents as needed

Some organizations may start out with user stories but if more detail is needed than a typical user story supplies or if they cannot have the user collaboration and/or agreement they need, they may develop the user stories into use cases. After all, there is nothing wrong with using all the tools and methodologies as needed to have the best quality and yet cost effective product.

If the code to be implemented is mostly for infrastructure as opposed to external functionality, user stories may also not be the appropriate methodology. Yes you can still write technical user stories but those are meant as supplement to user stories and without those, there is less benefit to using this methodology in the first place. As opposed to traditional requirements that are typically written from the system's perspective and so they lend themselves naturally to infrastructure.

John Dandeneau (2016) came up with a number of questions that can be asked to determine if user stories are the right approach for a particular project. They include:

“Is your project a “green-field” application or maintenance of an existing application?

How many stakeholders are driving requirements?

Are they in one or many departments?

Are they located in a single facility or span the globe?

Are there multiple variations of the same type of user?

Are stakeholders collaborative or adversarial?

Will changes in one set of requirements prompt changes to another set of requirements?

Do “outside” organizations need to understand your system's requirements to integrate with it or to otherwise understand it?

Is your regulatory environment such that your requirements, design, coding and testing could be scrutinized later?”

So for example, if the project is for maintenance or has many shareholders in many departments and located around the globe, and/or involves safety standards requiring detailed documentation, etc. this project may not be the best fit for user stories and one may consider use cases instead. It does not mean that it cannot be done and be beneficial, but one would need to consider the implications and whether some adjustments such as doing some hybrid process may not be necessary.

So when user stories are most useful? By definition they are a good fit when users or shareholders are available and willing to meet with development, especially face-to-face, throughout the development cycle. They are especially a good fit for user interfaces and all external functionality that can be expressed from user's perspective. They are also a good fit when requirements change a lot as the development progresses and to provide new functionality incrementally to stay competitive and meet the industry's needs. They allow development to start working on the highest priority requirements much earlier than traditional requirements and deferring the details of the lower priority requirements to a later time when there is more understanding of the needs and when the development is ready to start working on those items. They also encourage the continuous feedback which improves product quality.

Lastly, some developers feel that user stories may not give designers the big picture or the context to work from. Some of them therefore chose to turn to use cases instead. I would argue however, that this is where using epics with user stories grouped underneath and user journeys may be extremely beneficial. User stories should not be used in isolation but rather as a starting point, both to conversation and to develop supporting documents whether technical user stories, storyboards, and to see the bigger picture, the user journeys.

References

Agile Alliance Resources. Glossary. Retrieved October 12, 2017 from <https://www.agilealliance.org/glossary>.

Ambler, S. W. User Stories: An Agile Introduction. Retrieved October 18, 2017 from <http://www.agilemodeling.com/artifacts/userStory.htm>

Badri, A. User Stories and Technical Stories in Agile Development. Retrieved October 16, 2017 from <http://www.seilevel.com/requirements/user-stories-technical-stories-agile-development>

Berteig, M. (2014, Mar 6). User Stories and Story Splitting. Retrieved October 20, 2017 from <http://www.agileadvice.com/2014/03/06/referenceinformation/user-stories-and-story-splitting/>

Cardinal, M. (2013, Aug 8). Executable Specifications with Scrum: Refining User Stories by Grooming the Product Backlog. Retrieved October 12, 2017 from <http://www.informit.com/articles/article.aspx?p=2117898&seqNum=4>

Cockburn, A. (2008, Jan 9). Why I still use use cases. Retrieved November 1, 2017, from <http://alistair.cockburn.us/Why+I+still+use+use+cases>

Cockburn, A. (2017, Aug 16). Origin of user story is a promise for a conversation. Alistair.Cockburn.us Rss Feed. Retrieved October 12, 2017 from <http://howlodb.com/p/origin-of-user-story-is-a-promise-for-a-conversation-0kg130>

Cohn, M. (2011, Oct 24). User Stories, Epics, and Themes [Web log post]. Retrieved October 10, 2017, from <https://www.mountangoatsoftware.com/blog/stories-epics-and-themes>

Cohn, M. (2012, Dec 17). Two Examples of Splitting Epics. Retrieved October 16, 2017, from <https://www.mountangoatsoftware.com/blog/two-examples-of-splitting-epics>

Cowan, A. (2013). Storyboarding Tutorial. Retrieved October 10, 2017, from <https://www.alexandercowan.com/storyboarding-tutorial/>

Cowan, A. Your Best Agile User Story. Retrieved October 18, 2017, from <https://www.alexandercowan.com/best-agile-user-story/>

Dandeneau, J. (2016, Jan 18). The Nasty Effects of Mis-Using User Stories in Agile Projects. Retrieved November 1, 2017, from <https://cloudgps.astadia.com/user-stories-agile/>

DiScipio, T. (2017, Mar 28). Buyer Journey vs. User Journey: What's the Difference? Retrieved October 12, 2017, from <https://www.impactbnd.com/blog/buyer-journey-vs-user-journey>

Galen, R. (2013, Nov 16). Technical User Stories – What, When, And How. Retrieved October 12, 2017, from <http://rgalen.com/agile-training-news/2013/11/10/technical-user-stories-what-when-and-how>

Jones, S. A. (2011). User Stories: Not just Agile Anymore. Retrieved October 30, 2017, from <http://www.bridging-the-gap.com/user-stories-not-just-agile-anymore/>

Kaytes, G. A beginner's Guide to User Journey mapping. Retrieved October 23 2017, from <https://www.appcues.com/user-onboarding-academy/user-journey-map>

Marcano, A. (2011, Mar 24). Old Favourite: Feature Injection User Stories on a Business Value Theme. Retrieved October 20, 2017, from http://antonymarcano.com/blog/2011/03/fi_stories/

Mears, C. (2013, Apr 8). User Journeys – The Beginner's Guide. Retrieved October 11, 2017, from <http://theuxreview.co.uk/user-journeys-beginners-guide/>

Pichler, R. (2010, May 11). User Stories or Use Cases. Retrieved October 30, 2017, from <http://www.romanpichler.com/blog/user-stories-or-use-cases/>

Pichler, R. (2016, Mar 24). 10 Tips for Writing Good User Stories. Retrieved October 18, 2017, from <http://www.romanpichler.com/blog/10-tips-writing-good-user-stories/>

Rana, A. (2016, Jan 22). How Apply Storyboard in User Story? Retrieved October 10, 2017, from <https://www.visual-paradigm.com/tutorials/user-story-storyboard.jsp>

Requirement. (2017, September 10). In *Wikipedia, The Free Encyclopedia*. Retrieved 14:01, October 10, 2017, from <https://en.wikipedia.org/w/index.php?title=Requirement&oldid=799920593>

Roth, R. Write a Great User Story. Retrieved October 16, 2017, from <https://help.rallydev.com/writing-great-user-story>

Storyboard. (2017, October 5). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:34, October 10, 2017, from <https://en.wikipedia.org/w/index.php?title=Storyboard&oldid=803922466>

Trulock, V. Storyboards. Retrieved October 17, 2017, from http://hci.ilikecake.ie/env_storyboards.htm

Use case. (2017, September 21). In *Wikipedia, The Free Encyclopedia*. Retrieved 13:34, October 10, 2017, from https://en.wikipedia.org/w/index.php?title=Use_case&oldid=801742392

User journey. (2017, August 21). In *Wikipedia, The Free Encyclopedia*. Retrieved 13:56, October 11, 2017, from https://en.wikipedia.org/w/index.php?title=User_journey&oldid=796549952

User Stories. Retrieved October 16, 2017, from <https://www.mountangoatsoftware.com/agile/user-stories>

User Stories. Retrieved October 26, 2017, from <https://www.agilealliance.org/glossary/user-stories>

User story. (2017, October 10). In *Wikipedia, The Free Encyclopedia*. Retrieved 14:42, October 10, 2017, from https://en.wikipedia.org/w/index.php?title=User_story&oldid=804681195

Wake, B. (2003, Aug 17). INVEST in Good Stories, and SMART Tasks. Retrieved October 18, 2017, from <http://xp123.com/articles/invest-in-good-stories-and-smart-tasks/>

What is a User Journey? Retrieved October 19, 2017, from <https://www.scivisum.co.uk/resources/faqs/what-is-a-user-journey/>

Winter, H. (2016, May 14). The benefits of using Storyboards to gather requirements. Retrieved October 26, 2017, from <http://www.businessbullet.co.uk/business-analysis/the-benefits-of-using-storyboards-to-uncover-system-requirements-and-to-gain-an-agreed-common-understanding/>

Yodiz Team. (2016, Jan 20). What Is Epic In Agile Methodology [Web log post]. Retrieved October 10, 2017, from <https://www.yodiz.com/blog/what-is-epic-in-agile-methodology-definition-and-template-of-epic/>

Image credits

Image 1: By FlorianBauer79 - It is a screen shot from the JIRA agile Story Map plug in. Previously published: bauer-information-technology.com, CC BY-SA 3.0, <https://commons.wikimedia.org/w/index.php?curid=23797505>

Image 2: By <http://www.flickr.com/photos/tmray02/> (<http://www.flickr.com/photos/tmray02/1440415101/>) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons

Image 3: By Jagbirlehl (Own work) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons

Image 4: By Renata Rand McFadden under contract with UMGC.

Image 5: By 70Jack90 (Own work) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons

Image 6: By Whisternefet (Own work) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>) or GFDL (<http://www.gnu.org/copyleft/fdl.html>)], via Wikimedia Commons

Image 7: By Erik Möller.Eloquence at meta Wikimedia Foundation. (Transferred from meta.wikimedia to Commons.) [Public domain], from Wikimedia Commons

Image 8: By Nick Nijhuis (Own work) [CC BY-SA 4.0 (<https://creativecommons.org/licenses/by-sa/4.0/>)], via Wikimedia Commons

Chapter 8 Models

McFADDEN, R. (2017). SOFTWARE REQUIREMENTS – REQUIREMENTS DOCUMENTATION, UNDER CONTRACT FROM UMGC.

Introduction

In 1995 The Standish Group reported that on average only 16% of software projects were completed on time and on budgets and in large companies there were only 9%. When the research was repeated in 2015, that number only increased to 29%. One of the main recommendations for improvement was communication with stakeholders regarding the requirements. Modeling of the system and requirements is a valuable tool to enhance that communication.

Requirements modeling provides visual clarification to the written requirements. It still focuses on WHAT the features will look like and do and not HOW, which is the role of the design. Models should focus on the high level view as seen from the user's perspective and they should be simple, not get overly technical, as the point is to clarify the requirements.

Models should be used throughout the requirements process starting with elicitation through the requirements approval and change management. They promote understanding and clarity and help with the discussion and communication.

There are different types of visual models that can be used such as different types of UML models, diagrams, and even prototypes. Which models are used depends on the product and personal preference but the goal is to add value and clarification and help in the discussion and negotiation with the stakeholder.

Advantages

A visual model can help to start the conversation going in eliciting requirements (help in what questions to ask), provide a measure of progress (when model is done the requirements are complete), uncover issues (e.g. conflicting requirements, confusion over terminology, disagreements over scope or perspective), and help in understanding by showing the relationships or sequence between requirements.

Often it is easier to go over the requirements and make sure that everyone uses the terminology in the same way when there is a visual representation that can be referred to. In this way it can reduce some of the ambiguity that a natural language has. Visual model is also especially useful in understanding how different requirements fit together and to see the higher level and overall view of the features.

For example, business domain models clarify the concepts using business terms and help the stakeholders avoid confusion due to using different terminology or in a different way. Workflow models allow putting the processes in context, while wireframes and storyboards allow stakeholders to get a rough idea how the screens will look like and how some of the details will fit together.

When stakeholders have different perspectives or views of the system, having a separate model for each perspective can clarify how the requirements will satisfy their particular needs. Also, it

can help to focus the conversation on only those items that are important to that stakeholder(s) while at the same time making sure that no requirement is missed.

Lastly, the requirements models are also helpful and provide a transition to design as well as can be used to derive test cases and test scenarios.

Business Domain Model

Business Domain Model (BDM) represents the domain's behavior and data using the terminology of the domain so that it helps with the communication with non-technical stakeholders. It is usually at a high level of abstraction (simplification) and it logically represents different business concepts and how they relate to each other. They allow stakeholders to see the overall system or a piece of the system as it would be fulfilled by software and it helps in the elicitation and discussion of different key requirements for those concepts.

BDM can be represented using Unified Modeling Language (UML) class diagram as shown in the example below. A single diagram such as this can encompass hundreds of requirements and their relationships and as such makes it easier to understand the big picture and how they all fit together.

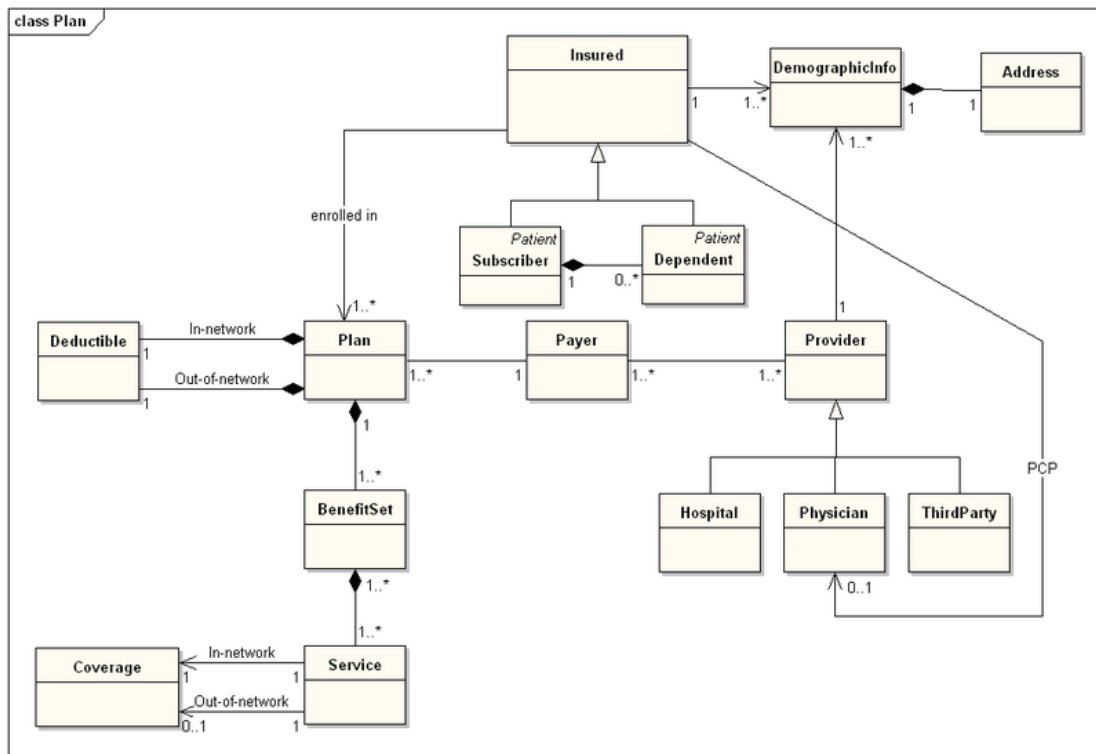


Image 1: Sample domain for health insurance plan

Business Process Model

Business process model (BPM) represents and maps the business processes and it is used to analyze and improve those processes. Relative to requirements it is used to understand how the

data flows across the system and how it is manipulated or used. As such it can be represented using flow charts, control flow diagrams, Gantt charts, PERT diagrams, UML use case diagrams, UML activity diagrams, etc.

BPM can represent different perspectives and together they can be used to get a clearer picture of the processes. Typical perspectives include:

- Functional – represents the process elements and the data that is relevant to them
- Behavioral (Dynamic) – represents the sequence of interaction between process elements
- Organizational – represents where and by whom the process elements are performed
- Informational – represents the origination of the information that is produced and analyzed

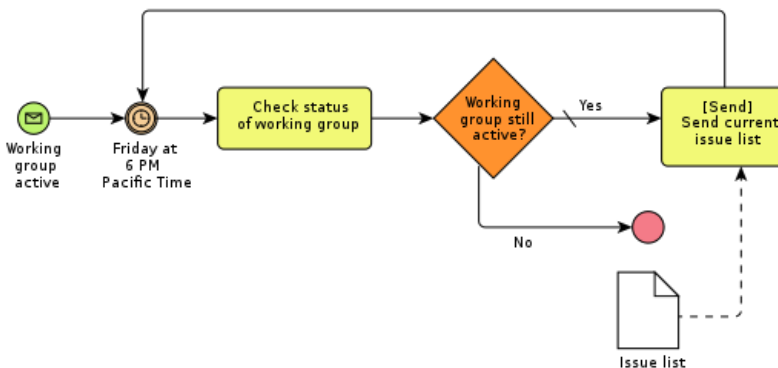


Image 2: Example of business process modeling of a process with a normal flow with the Business Process Modelling Notation

Workflow model

Workflow models represent a sequence of operations for some business activity using some resource to deliver something whether service or information. In its simplest form it has some input, then some transformation rules (some activity done with the input), and then output.

Often the workflow model is developed to represent the current flow in order to analyze it and improve it whether to achieve greater efficiency, responsiveness, or profitability. For example, the image below shows original workflow for a patch being integrated in the original requirements and then the modified workflow.

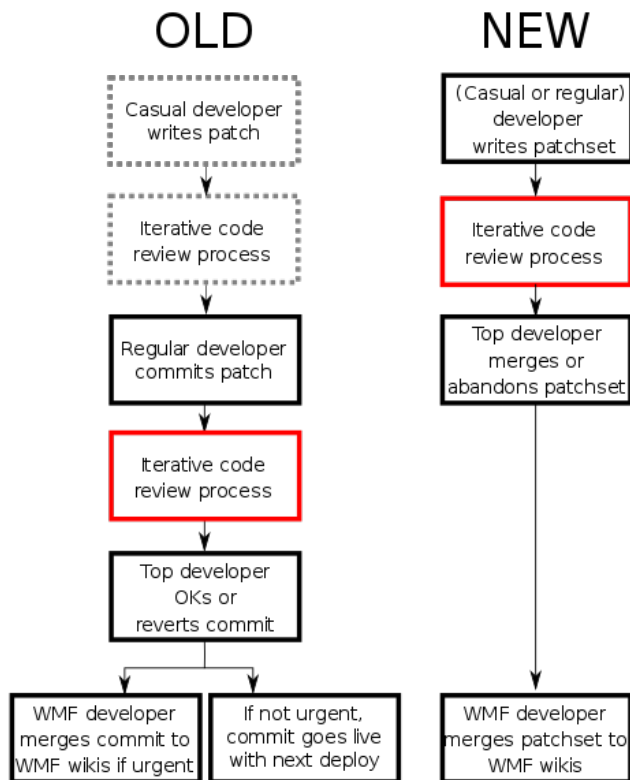


Image 3: Integrating patch workflow

Workflow models can use different notations such as simple graphs as show in the above example, or more mathematical graphs such as Petri net, or Unified Modeling Language (UML) activity diagrams. They are especially useful for analyzing flow control and data flow through a feature and a system. They can be used when designing a new system to put the work processes in real-life context and to make sure that the workflow matches the needs of the business. Just like other visual representations of the requirements, the advantage is that the stakeholders can understand them and provide feedback without any familiarity with programming.

Regardless of the notation however, the model will have the start of the workflow (input or trigger), a sequence of activities that represent a task, some decisions that guide the flow of control, an indication of who performs the tasks, and the final activities that end the workflow. The information includes the timing and whether processes are sequential or in parallel. A simple graph may just show the flow of data but the activity diagrams show both the control and data flow.

Data Flow Diagrams

Data Flow Diagram (DFD) represents the flow of data through a feature or the whole system. Just like workflow it shows the input, output, and how data will flow through the system but it also shows where the data will be stored and it does not show any process timing or whether the processes will be sequential or in parallel. It typically does not show the control flow either but it usually has the process, external entity, data store, and the data flow. DFDs can be used for requirements (e.g. use case) but also for the design.

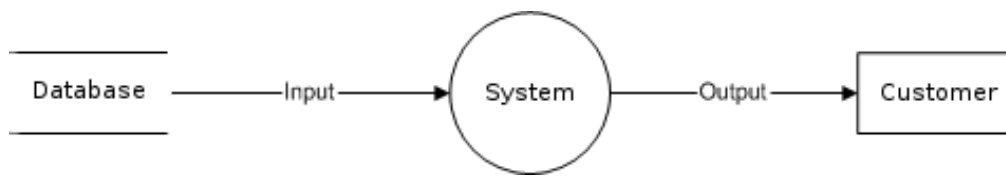


Image 4: Simple data flow diagram

DFDs are useful to represent the logical information flow of the system and they are relatively easy to create as they have simple notation. In fact, there are only four basic symbols that are used. Data store has an open ended rectangle with arrows coming in (data is written) or going out (data is read) to show the data flow, arrows are used to show data move through the system, external entity has the closed rectangle, the process (e.g. archive records, enter orders) is usually a rounded rectangle or a circle, and all of these have labels of course.

Some basic rules include:

- Entity cannot provide data to another entity without going through a process
- Data cannot move to data store or out of data store without going through a process
- Data cannot move from one data store to another without going through a process

Here is another more complicated example:

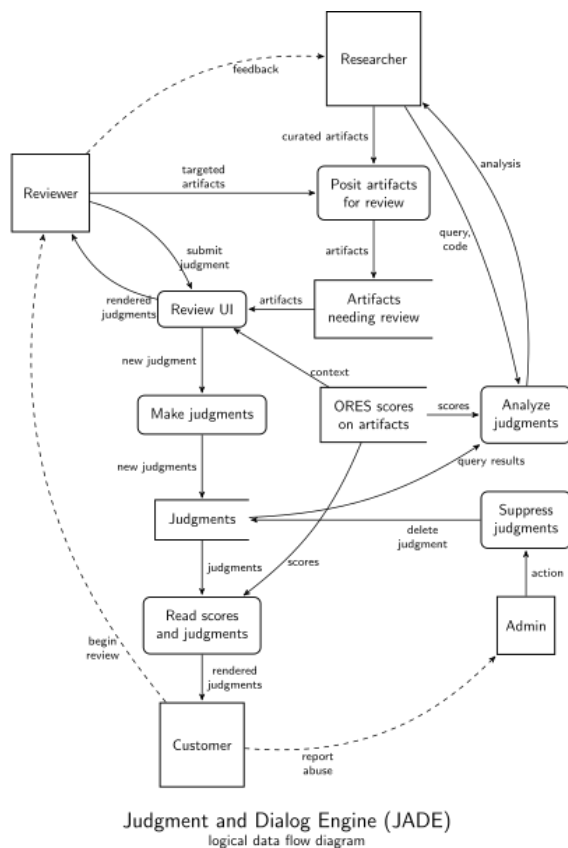


Image 5: Judgement and Dialog Engine data flow

Unified Modeling Language diagrams

Unified Modeling Language (UML) is a standard notation that is used for modeling different aspects of software program and it can describe the structure (framework of the system) or the behavior (interactions) of the system.

So for example, class diagram (mentioned earlier in section for BDM) is an example of structure diagram while use case and activity diagrams (mentioned earlier in sections for BPM and workflow model) are examples of behavioral diagrams. The other type of UML diagram most used for requirements is use case diagram. Software requirements involve both the structure and behavior of a system and hence a combination of structural and behavioral diagrams can be used.

Structure diagrams include:

- **Class diagram** – used to represents the objects in the system (to include characteristics and behaviors) and interactions between the objects. This diagram is also used for representing the design of the system
- **Component diagram** – used to model components of the system and the relationships (interfaces) between them
- **Deployment diagram** – used to represents how the software is deployed across different hardware devices
- **Object diagram** – similar to class diagram it shows the objects and relationships between them but it also includes data for specific state for extra clarification
- **Package diagram** – models the dependencies between packages in the system

Behavioral diagrams include:

- **Use case diagram** – this type of diagram represents a high level business goal of the system or can be a graphic representation of a feature or a specific requirement
- **Activity diagram** – models a workflow of a component in the system
- **State machine diagram** – similar to activity diagram but has a more technical notation and represents behavior of objects for a specific state. This diagram is also called state diagram or state chart diagram
- **Sequence diagram** – demonstrates how the object interact with each other in a specific order for a particular scenario
- **Communication diagram** – models the messages passed between objects

Here are a few more examples of how some of the diagrams look like:

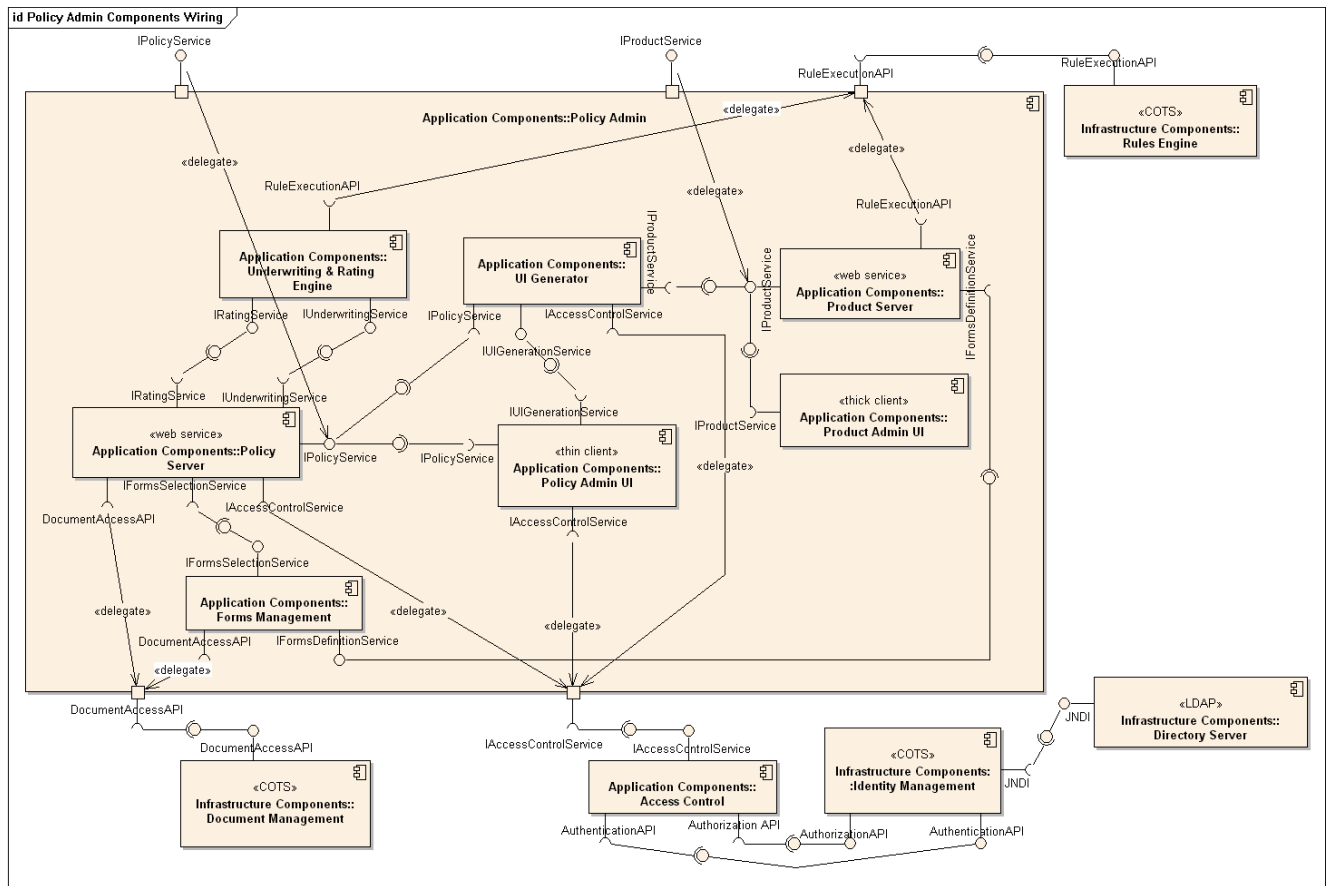


Image 6: A component diagram illustrating an Insurance policy administration software system

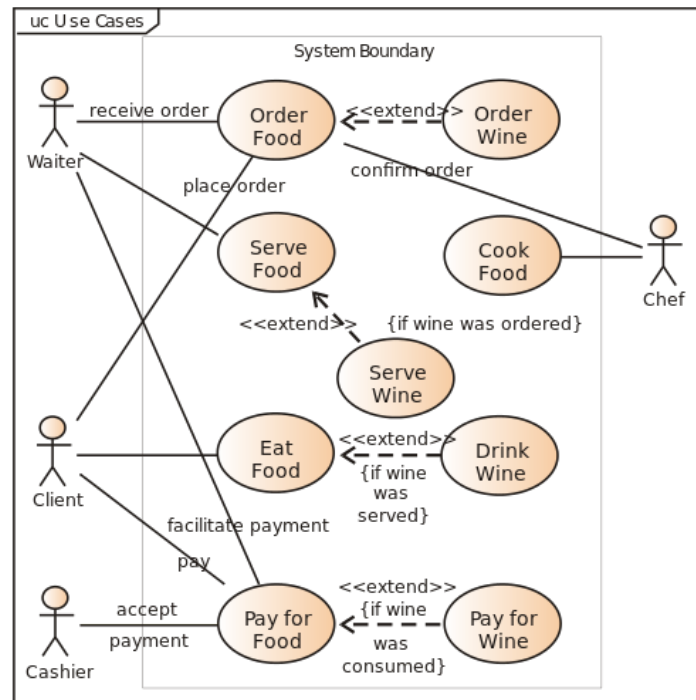


Image 7: Use case diagram for a restaurant

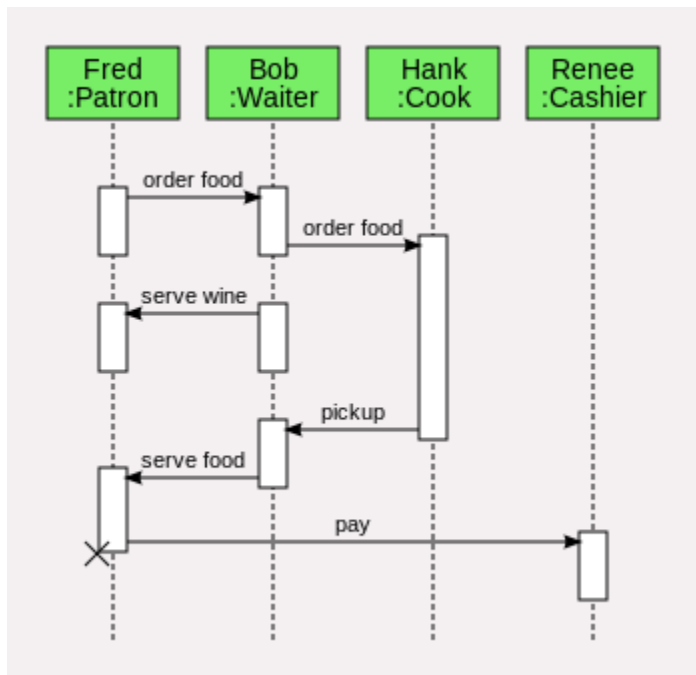


Image 8: Sequence diagram for restaurant interaction

Prototypes

In addition to models, requirements can be verified with stakeholders using simple prototypes. A prototype is simplified code that demonstrates some key elements of the product such as the basic screens or flow between screens or how the functionality would look or work from the external view. Most of the code to include data is hardcoded and often a prototype is only used to get feedback and may be thrown away or totally re-written for the actual product.

One way of thinking of prototype is that it is a mocked up face with nothing in the back. The code does not use coding standards, uses simplified algorithms, does not have any error checking, nor is it integrated with external resources such as database or other modules.

There are a number of advantages of prototyping such as: getting feedback from stakeholders early in the process and validate the requirements, an insight whether project estimates and schedules are accurate, and whether the technology that is planned to be used will in fact meet the project's goals. A prototype can also be used to show different possible interpretations of the requirements and to determine which interpretation is more accurate.

The basic types of prototypes include:

- Throwaway prototype – a quick model to demonstrate visually some key requirements, especially interfaces, and get feedback from stakeholders. It gets thrown away because rewriting it using the proper coding techniques is not worth the time. It is quicker to start fresh using the proper code and technology
- Evolutionary prototype – a functional system that initially just has the code elements to get feedback and then it is used to refine and rebuild by adding additional features and details to eventually become the final product. The partial product can be demonstrated or sent to stakeholders to get feedback along the way

- Incremental prototype – separate prototypes are built with different aspects of the requirements and then merged into the final product
- Extreme prototype – often used for web applications where the first prototype just has the static HTML pages, then the screens are programmed using simulated services layer, and finally the services are implemented for the complete product. At each step the prototype can be demonstrated to the stakeholders to get feedback.

There are a number of advantages and disadvantages to using prototypes. As any model, having visual representation of key requirements and interfaces provides better feedback from stakeholders earlier in the development process. As such it can reduce time and costs from misunderstandings of requirements or missing requirements and decrease the need to redo key elements later in the process.

However, prototypes have costs and time involved which needs to be calculated into the overall estimates. There can also be a misunderstanding on the progress of the project because stakeholders and managers may not understand how much time and effort is needed to make the prototype an actual application with the error-checking, security, and real back-end services implemented. There can also be an issue with the stakeholder getting attached to the prototype with some specific look and feel which is not meant to be the final product because of the technology differences used in prototype versus the one needed for the final product. A developer may also get attached to the prototype which is meant to be thrown away because he/she spent a lot of time on it and yet it does not have the underlying architecture needed so it cannot be efficiently converted into the final product. Lastly, a prototype should be done quickly but sometimes developers may spend too much time on it holding up the rest of the development.

References

Brandenburg, L. How to Make the Requirements Process Faster With Visual Models. Retrieved December 22, 2017, from <http://www.bridging-the-gap.com/how-to-make-the-requirements-process-faster-with-visual-models/>

Business process model. Retrieved January 2, 2018, from https://en.wikipedia.org/wiki/Business_process_modeling

Business Process Modeling in Software Development. Retrieved January 2, 2018, from <https://www.smartsheet.com/beginners-guide-business-process-modeling>

Data flow diagram. Retrieved January 6, 2018, from https://en.wikipedia.org/wiki/Data_flow_diagram

Domain model. Retrieved December 30, 2017, from https://en.wikipedia.org/wiki/Domain_model

Ghahrai, A. (2008, Nov 8). What is prototyping model? Retrieved January 9, 2018, from <https://www.testingexcellence.com/prototyping-model-software-development/>

Johnson, J., Crear, J., Mulder, T., Gesmer, L., & Poort, J. (2015). CHAOS Report 2015. Retrieved January 2, 2018, from <http://blog.standishgroup.com/post/50>

Leffingwell, D, & Widrig, D. (1999). Managing Software Requirements. Retrieved December 21, 2017, from <https://doc.lagout.org/programmation/C++/Addison%20Wesley%20-%20Leffingwell%20%26%20Widrig%20-%20Managing%20Software%20Requirements%2C%201St%20Edition.pdf>

Nishadha. (2012, Feb 2). The Complete Guide to UML Diagram Types with Examples. Retrieved January 4, 2018, from <http://creatly.com/blog/diagrams/uml-diagram-types-examples/>

Schedlbauer, M. (2007, Aug 13). Workflow modeling with UML activity diagrams. Retrieved January 7, 2018, from <https://www.batimes.com/articles/workflow-modeling-with-uml-activity-diagrams.html>

Software prototyping. Retrieved January 9, 2018, from https://en.wikipedia.org/wiki/Software_prototyping

The CHAOS Report (1995). The Standish Group. Retrieved January 2, 2018, from <http://www.csus.edu/indiv/v/velianitis/161/chaosreport.pdf>

Unified Modeling Language. Retrieved January 4, 2018, from https://en.wikipedia.org/wiki/Unified_Modeling_Language

What is Data Flow Diagram? Retrieved January 6, 2018, from <https://www.visual-paradigm.com/guide/data-flow-diagram/what-is-data-flow-diagram/>

Where do requirements models fit in the project lifecycle? (2007). Retrieved December 30, 2017, from <http://searchsoftwarequality.techtarget.com/report/Where-do-requirements-models-fit-in-the-project-lifecycle>

Workflow. Retrieved January 7, 2018, from <https://en.wikipedia.org/wiki/Workflow>

Workflow tutorial. Retrieved January 7, 2018, from <http://www.pnmsoft.com/resources/bpm-tutorial/workflow-tutorial/>

Image Credits

Image 1: Sample Domain Model. By Kishorekumar 62 (Own work) [CC BY-SA 3.0], via Wikipedia

Image 2: Business Process Model. By Hazmat2 (talk) Public Domain, <https://commons.wikimedia.org/w/index.php?curid=18126163>

Image 3: Integrating patch workflow. By Jarry1250, via Wikipedia, https://en.wikipedia.org/wiki/File:MediaWiki-Wikimedia_Git-Gerrit_workflow.svg

Image 4: Data flow diagram. By AutumnSnow, via Wikipedia, https://en.wikipedia.org/wiki/File:DataFlowDiagram_Example.png

Image 5: Judgement and Dialog Engine data flow. By Adamw (Own work) [CC BY-SA 4.0 (<https://creativecommons.org/licenses/by-sa/4.0>)], via Wikimedia Commons

Image 6: Component diagram. By Kishorekumar 62 (Own work) [CC BY-SA 3.0], via Wikipedia, https://en.wikipedia.org/wiki/File:Policy_Admin_Component_Diagram.PNG

Image 7: Use case diagram for restaurant. By Kishorekumar 62 (redrawn by Marcel Douwe Dekker) - Own work, redrawn of w:File:Restaurant Model.png [CC0], via Wikimedia Commons

Image 8: Sequence diagram. By Aflafla1 (Restaurant-UML-SEQ.gif by LeonardoG) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons

Chapter 9 Requirements Negotiation

DR. MICHAEL BROWN, UMGC.

In some projects all of the requirements can be met. But too often stakeholders will not agree on requirements. Your CTO might have a non-functional requirement that all systems log (ie. write to a file) activities to expedite the resolving of technical issues. But logging slows systems down and end users want fast systems. Managers that travel often might want all of a system's data accessible over the Internet, but the legal and cyber-security departments is worried about privacy. Sometime even stakeholders with the same role or job title, might not agree on what the system needs to do. In some cases, the organization might have agreed upon policies that allow these conflicts to be quickly resolved.

When Stakeholders do not agree on requirements or have conflicting requirements, we have to begin Requirements Negotiation. All Software Requirements Analysts should know the basics of negotiation.

ORGANIZATIONAL BEHAVIOR (2010). UNIVERSITY OF MINNESOTA LIBRARIES PUBLISHING.
RETRIEVED [HTTP://OPEN.LIB.UMN.EDU/ORGANIZATIONALBEHAVIOR/PART/CHAPTER-10-CONFLICT-AND-NEGOTIATIONS/](http://open.lib.umn.edu/organizationalbehavior/part/chapter-10-conflict-and-negotiations/)

A common way that parties deal with conflict is via negotiation. Negotiation is a process whereby two or more parties work toward an agreement. There are five phases of negotiation, which are described below.

The Five Phases of Negotiation



Figure. The Five Phases of Negotiation

Phase 1: Investigation

The first step in negotiation is the investigation, or information gathering stage. This is a key stage that is often ignored. Surprisingly, the first place to begin is with yourself: What are your goals for the negotiation? What do you want to achieve? What would you concede? What would you absolutely not concede? Leigh Steinberg, the most powerful agent in sports (he was the role model for Tom Cruise's character in *Jerry Maguire*), puts it this way: "You need the clearest possible view of your goals. And you need to be brutally honest with yourself about your priorities."

During the negotiation, you'll inevitably be faced with making choices. It's best to know what you want, so that in the heat of the moment you're able to make the best decision. For example, if you'll be negotiating for a new job, ask yourself, "What do I value most? Is it the salary level? Working with coworkers whom I like? Working at a prestigious company? Working in a certain geographic area? Do I want a company that will groom me for future positions or do I want to change jobs often in pursuit of new challenges?"

Phase 2: Determine Your BATNA

If you don't know where you're going, you will probably end up somewhere else.

Lawrence J. Peter

One important part of the investigation and planning phase is to determine your BATNA, which is an acronym that stands for the “best alternative to a negotiated agreement.” Roger Fisher and William Ury coined this phrase in their book *Getting to Yes: Negotiating without Giving In*.

Thinking through your BATNA is important to helping you decide whether to accept an offer you receive during the negotiation. You need to know what your alternatives are. If you have various alternatives, you can look at the proposed deal more critically. Could you get a better outcome than the proposed deal? Your BATNA will help you reject an unfavorable deal. On the other hand, if the deal is better than another outcome you could get (that is, better than your BATNA), then you should accept it.

Think about it in common sense terms: When you know your opponent is desperate for a deal, you can demand much more. If it looks like they have a lot of other options outside the negotiation, you’ll be more likely to make concessions.

As Fisher and Ury said, “The reason you negotiate is to produce something better than the results you can obtain without negotiating. What are those results? What is that alternative? What is your BATNA—your Best Alternative To a Negotiated Agreement? That is the standard against which any proposed agreement should be measured.”

The party with the best BATNA has the best negotiating position, so try to improve your BATNA whenever possible by exploring possible alternatives.

Going back to the example of your new job negotiation, consider your options to the offer you receive. If your pay is lower than what you want, what alternatives do you have? A job with another company? Looking for another job? Going back to school? While you’re thinking about your BATNA, take some time to think about the other party’s BATNA. Do they have an employee who could readily replace you?

Once you’ve gotten a clear understanding of your own goals, investigate the person you’ll be negotiating with. What does that person (or company) want? Put yourself in the other party’s shoes. What alternatives could they have? For example, in the job negotiations, the other side wants a good employee at a fair price. That may lead you to do research on salary levels: What is the pay rate for the position you’re seeking? What is the culture of the company?

Greenpeace’s goals are to safeguard the environment by getting large companies and organizations to adopt more environmentally friendly practices such as using fewer plastic components. Part of the background research Greenpeace engages in involves uncovering facts. For instance, medical device makers are using harmful PVCs as a tubing material because PVCs are inexpensive. But are there alternatives to PVCs that are also cost-

effective? Greenpeace’s research found that yes, there are. Knowing this lets Greenpeace counter those arguments and puts Greenpeace in a stronger position to achieve its goals.

Phase 3: Presentation

The third phase of negotiation is presentation. In this phase, you assemble the information you’ve gathered in a way that supports your position. In a job hiring or salary negotiation situation, for instance, you can present facts that show what you’ve contributed to the organization in the past (or in the previous position), which in turn demonstrates your value.

Perhaps you created a blog that brought attention to your company or got donations or funding for a charity. Perhaps you're a team player who brings out the best in a group.

Phase 4: Bargaining

During the bargaining phase, each party discusses their goals and seeks to get an agreement. A natural part of this process is making concessions, namely, giving up one thing to get something else in return. Making a concession is not a sign of weakness—parties expect to give up some of their goals. Rather, concessions demonstrate cooperativeness and help move the negotiation toward its conclusion. Making concessions is particularly important in tense union-management disputes, which can get bogged down by old issues. Making a concession shows forward movement and process, and it allays concerns about rigidity or closed-mindedness. What would a typical concession be? Concessions are often in the areas of money, time, resources, responsibilities, or autonomy. When negotiating for the purchase of products, for example, you might agree to pay a higher price in exchange for getting the products sooner. Alternatively, you could ask to pay a lower price in exchange for giving the manufacturer more time or flexibility in when they deliver the product.

One key to the bargaining phase is to ask questions. Don't simply take a statement such as "we can't do that" at face value. Rather, try to find out why the party has that constraint. Let's take a look at an example. Say that you're a retailer and you want to buy patio furniture from a manufacturer. You want to have the sets in time for spring sales. During the negotiations, your goal is to get the lowest price with the earliest delivery date. The manufacturer, of course, wants to get the highest price with the longest lead time before delivery. As negotiations stall, you evaluate your options to decide what's more

important: a slightly lower price or a slightly longer delivery date? You do a quick calculation. The manufacturer has offered to deliver the products by April 30, but you know that some of your customers make their patio furniture selection early in the spring, and missing those early sales could cost you \$1 million. So, you suggest that you can accept the April 30 delivery date if the manufacturer will agree to drop the price by \$1 million.

"I appreciate the offer," the manufacturer replies, "but I can't accommodate such a large price cut." Instead of leaving it at that, you ask, "I'm surprised that a 2-month delivery would be so costly to you. Tell me more about your manufacturing process so that I can understand why you can't manufacture the products in that time frame."

"*Manufacturing* the products in that time frame is not the problem," the manufacturer replies, "but getting them *shipped* from Asia is what's expensive for us."

When you hear that, a light bulb goes off. You know that your firm has favorable contracts with shipping companies because of the high volume of business the firm gives them. You make the following counteroffer: "Why don't we agree that my company will arrange and pay for the shipper, and you agree to have the products ready to ship on March 30 for \$10.5 million instead of \$11 million?" The manufacturer accepts the offer—the biggest expense and constraint (the shipping) has been lifted. You, in turn, have saved money as well.

Phase 5: Closure

Closure is an important part of negotiations. At the close of a negotiation, you and the other party have either come to an agreement on the terms, or one party has decided that the final offer

is unacceptable and therefore must be walked away from. Most negotiators assume that if their best offer has been rejected, there's nothing left to do. You made your best offer and that's the best you can do. The savviest of negotiators, however, see the rejection as an opportunity to learn. "What would it have taken for us to reach an agreement?"

Negotiation Strategies

Distributive Approach

The distributive view of negotiation is the traditional fixed-pie approach. That is, negotiators see the situation as a pie that they have to divide between them. Each tries to get more of the pie and "win." For example, managers may compete over shares of a budget. If marketing gets a 10% increase in its budget, another department such as R&D will need to decrease its budget by 10% to offset the marketing increase. Focusing on a fixed pie is a common mistake in negotiation, because this view limits the creative solutions possible.

Integrative Approach

A newer, more creative approach to negotiation is called the integrative approach. In this approach, both parties look for ways to integrate their goals under a larger umbrella. That is, they look for ways to *expand* the pie, so that each party gets more. This is also called a win-win approach. The first step of the integrative approach is to enter the negotiation from a cooperative rather than an adversarial stance. The second step is all about listening. Listening develops trust as each party learns what the other wants and everyone involved arrives at a mutual understanding. Then, all parties can explore ways to achieve the individual goals. The general idea is, "If we put our heads together, we can find a solution that addresses everybody's needs." Unfortunately, integrative outcomes are not the norm. A summary of 32 experiments on negotiations found that although they could have resulted in integrated outcomes, only 20% did so. [7] One key factor related to finding integrated solutions is the experience of the negotiators who were able to reach them.

Avoiding Common Mistakes in Negotiations

Failing to Negotiate/Accepting the First Offer

You may have heard that women typically make less money than men. Researchers have established that about one-third of the gender differences observed in the salaries of men and women can be traced back to differences in starting salaries, with women making less, on average, when they start their jobs. Some people are taught to feel that negotiation is a conflict situation, and these individuals may tend to avoid negotiations to avoid conflict. Research shows that this negotiation avoidance is especially prevalent among women. For example, one study looked at students from Carnegie-Mellon who were getting their first job after earning a master's degree. The study found that only 7% of the women negotiated their offer, while men negotiated 57% of the time. The result had profound consequences. Researchers calculate that people who routinely negotiate salary increases will earn over \$1 million more by retirement than people who accept an initial offer every time without asking for more. The good news is that it appears that it is possible to increase negotiation efforts and confidence by training people to use effective negotiation skills.

Letting Your Ego Get in the Way

Thinking only about yourself is a common mistake, as we saw in the opening case. People from the United States tend to fall into a self-serving bias in

which they overinflate their own worth and discount the worth of others. This can be a disadvantage during negotiations. Instead, think about why the other person would want to accept the deal. People aren't likely to accept a deal that doesn't offer any benefit to them. Help them meet their own goals while you achieve yours. Integrative outcomes depend on having good listening skills, and if you are thinking only about your own needs, you may miss out on important opportunities. Remember that a good business relationship can only be created and maintained if both parties get a fair deal.

Having Unrealistic Expectations

Susan Podziba, a professor of mediation at Harvard and MIT, plays broker for some of the toughest negotiations around, from public policy to marital disputes. She takes an integrative approach in the negotiations, identifying goals that are large enough to encompass both sides. As she puts it, "We are never going to be able to sit at a table with the goal of creating peace and harmony between fishermen and conservationists. But we can establish goals big enough to include the key interests of each party and resolve the specific impasse we are currently facing. Setting reasonable goals at the outset that address each party's concerns will decrease the tension in the room, and will improve the chances of reaching an agreement." Those who set unreasonable expectations are more likely to fail.

Getting Overly Emotional

Negotiations, by their very nature, are emotional. The findings regarding the outcomes of expressing anger during negotiations are mixed. Some researchers have found that those who express anger negotiate worse deals than those who do not, and that during online negotiations, those parties who encountered anger were more likely to compete than those who did not. In a study of online negotiations, words such as *despise*, *disgusted*, *furios*, and *hate* were related to a reduced chance of reaching an agreement. However, this finding may depend on individual personalities. Research has also shown that those with more power may be more effective when displaying anger. The weaker party may perceive the anger as potentially signaling that the deal is falling apart and may concede items to help move things along. This holds for online negotiations as well. In a study of 355 eBay disputes in which mediation was requested by one or both of the parties, similar results were found. Overall, anger hurts the mediation process unless one of the parties was perceived as much more powerful than the other party, in which case anger hastened a deal. Another aspect of getting overly emotional is forgetting that facial expressions are universal across cultures, and when your words and facial expressions don't match, you are less likely to be trusted.

Letting Past Negative Outcomes Affect the Present Ones

Research shows that negotiators who had previously experienced ineffective negotiations were more likely to have failed negotiations in the future. Those who were unable to negotiate some type of deal in previous negotiation situations tended to have lower outcomes than those who had successfully negotiated deals in the past. The key to remember is that there is a tendency to let the past repeat itself. Being aware of this tendency allows you to overcome it. Be vigilant to examine the issues at hand and not to be overly swayed by past experiences, especially while you are starting out as a negotiator and have limited experiences.

Chapter 10 Requirements Analysis

Example of Un-verifiable Requirement

DR. MICHAEL BROWN, UMGC.

Consider this example. An employee placement firm has a database of resumes of people that they can hire for temporary work. When an employer calls asking about the hire a temporary worker, they have manually go through the database to match the skills of the employers to the skills on the resume. To help automate this, they pay for an enhancement to their software.

They would like to be able to search on skills. Ideally, they would like to enter a skill, like “Java” or “Analyst” and receive a list of all of the resumes that have that skill. And they would like the list sorted by the best candidate.

The developers go off to design and build this new feature. They decide that the best way to sort the list is based upon the number of times that skill appears in the resume. So, if resume 1 has “Java” twice and resume 2 has “Java” once, then clearly resume 1 is a better candidate.

After building this new feature they demo it to the customer. But the customer is disappointed. He runs searches, but doesn’t agree with the ordering. He thinks that the best candidate should be based upon the number of years doing the skill. So, if resume 1 has “Java” twice (each job for 1 year) and resume 2 has “Java” once (but did Java for 10 years), then resume 2 is a better candidate.

What is issue here? Was this a design defect? Was it a coding defect? The mistake that this team made was in requirements. Ordering candidates by the best candidate is subjective. You cannot verify “best”, until you define exactly what that means.

McFADDEN, R. (2017). SOFTWARE REQUIREMENTS – REQUIREMENTS DOCUMENTATION, UNDER CONTRACT FROM UMGC.

Introduction

Requirements analysis, also called requirements engineering, includes eliciting and then analyzing and validating the requirements to make sure they are complete, clear, consistent, unambiguous, and testable. The analysis can take a long time and effort when the system is large and complex (e.g. hundreds or thousands requirements) and has multiple stakeholders who may have different and conflicting needs. It requires communication with the customers/stakeholders to reach consensus on the purpose, goals, and overall expectations of the system to be build. As such it is a critical part of the software development process.

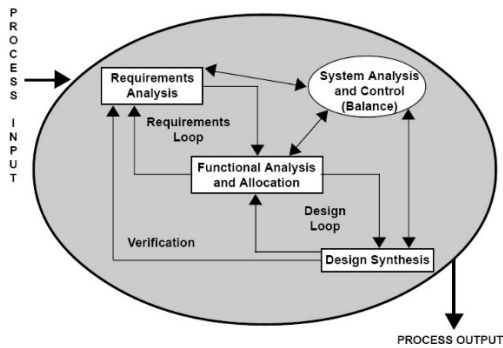


Image 1: A systems engineering perspective on requirements analysis

Well written requirements

Writing good requirements for a large and complex system is not easy. Requirements need to be organized in some themes and/or features, must be relevant and related to the business needs or opportunities, must be actionable and quantifiable, and have just enough detail needed for the system design.

IEEE Standard 830-1998 For Software Requirements Recommendations lists the below attributes for a good requirements specification. While the document itself may be dated, these characteristics still hold true:

- a) **Correct** – represents the true needs of the user
- b) **Unambiguous** – there should only be a single interpretation. Some examples that cause requirement to be ambiguous include when it is vague, incomplete or imprecise.
- c) **Complete** – all significant requirements related to the function, performance and external interfaces are accounted for
- d) **Consistent** – there are no requirements that contradict each other or are in conflict
- e) **Ranked for importance and/or stability** – identify relative importance such as critical versus desirable
- f) **Verifiable** – there exists finite cost-effective process that can check that the system meets the requirement
- g) **Modifiable** – the structure and style allows easy and consistent changes to the requirement
- h) **Traceable** – it can be referenced to future development documents (e.g. design document or test case) and it can be traced backwards from development documents (e.g. code or design document)

When drafting or analyzing requirements, all of these should be considered. In addition, one should pay attention to the following guidelines:

- a) **Avoid conjunctions** – requirements should not have words such as “and” or “or” as that adds unnecessary ambiguity to the requirements
- b) **Keep positive and active voice** – requirements should be written in positive and active voice using verbs such as “must”, “shall”, and “will”.
- c) **Identify dependencies** – a complex system will have many dependencies between requirements and/or other systems or projects that need to be identified and tracked to make sure that consistency between them is maintained
- d) **Order requirements** – it is beneficial to have requirements ordered or rated based on benefit, risk, and/or priority
- e) **Use identifiers** – each requirement should have an identifier to help track it across the process to include between high-level and low-level requirements, design document, and verification tests
- f) **Standardize language** – provide a terms section and use the same exact terms to express the same thing to avoid confusion and ambiguity
- g) **Make testable** – it cannot be stressed enough that each requirement must be testable. If something cannot be tested then it is not a valid requirement as it cannot be verified that it was satisfied
- h) **Keep implementation neutral** – requirements should state WHAT the system must do and not have design details which state HOW it will be done
- i) **Do not forget exceptions** – make sure to consider and document conditions when a requirement should not apply or should work differently because of some trigger. For example, if requirement says that form should be cleared once submitted, consider the behavior when the submission failed. Most likely in that case the form should not be cleared so it can be re-submitted.
- j) **Do not use weak words** – requirements must be measurable and testable and not up to readers interpretation so should not use words such as: efficient, powerful, fast, easy, effective, reliable, normal, few, most, quickly, timely, compatible, etc. Make requirements precise such as “table data will load in less than 5 seconds”
- k) **Avoid using symbols** – do not use symbols that may be interpreted differently such as “/” slash which could be interpreted as either, or, both etc. For example, if requirement had “... disengage the automatic cruise/steering system ...” does it mean both of them at the same time or one at a time?

- l) **Write from user perspective** – always write and consider the requirement from the user’s view and interaction with the system
- m) **Avoid redundancy** – there should not be multiple places with the same requirement which leads to confusion and inconsistency such as when one is updated and not the other
- n) **Keep feasible** – requirements have to be realistic and technically possible to implement given the budget and time constraints of the project
- o) **Cover everything** – make sure to consider every interaction in the product and do not leave anything to be guessed or assumed
- p) **Keep concise** – use the minimum number of words to be clear and precise
- q) **Group logically** – group requirements into logical blocks such as logical components to make it easier to understand and track
- r) **Use visuals** – mockups of screenshots and diagrams can enhance understanding quicker than long paragraphs of narrative
- s) **Avoid unnecessary requirements** – a requirement should satisfy a user’s need, add needed value to the system, or satisfy a standard. It should not be something a developer thinks is “fun” or “cool” to have
- t) **Don’t forget boundary conditions** – make sure that the requirement accounts for all possible numerical and date ranges. For example if there is a requirement for behavior when value is less than 50 and another for greater than 50, there should be a requirement for what happens when the value is exactly 50. Using the words “inclusively” and “exclusively” makes it clear whether the endpoints are included or not
- u) **Be careful with pronouns** – if using “this” or “that”, make it clear what it refers to and that there is no confusion
- v) **Be careful with abbreviations** – do not assume that the reader will know the abbreviation and be careful that you use it correctly. For example, abbreviation i.e. means “that is;” and whatever follows needs to be a complete list of items in that category. The abbreviation e.g. means “for example,” so what follows are just representatives of the complete set.

Evaluating draft requirements

Once requirements are drafted, they should be re-reviewed and inspected to make sure they meet the quality and best practice guidelines before they are formally presented to the stakeholders for requirements validation and approval.

The document format should be clear and consistent throughout the different sections. It should be checked for all the characteristics of well-written requirements, especially that there is no ambiguity that could result in incorrect product being implemented, that it is consistent across

requirements and logical components (e.g. features), and that all requirements are testable and traceable.

The document should be as complete as possible at that stage of software development and clean, although it may not be perfect. It is expected that the document will be fine-tuned especially the requirements, during the official validation and throughout the development cycle. However, there should be no spelling or grammatical errors, no logical errors, and no errors in the visual components which could distract or confuse the readers of the document.

A good approach is to first check the table of contents and make sure it is updated and then work through the document top to bottom making sure that whatever updates were made during the document creation are consistent across the document and that the information is precise and clear. It is always helpful to have someone proofread the document for an extra unbiased feedback as well.

Often the development team will review the document and make corrections or changes before it is presented to the customer. Also, some organizations may have a checklist such as the one suggested by Software Quality Consulting (http://www.swqual.com/images/requirements_checklist.pdf) which is used to help in the review of the document to make sure that all common trouble areas are addressed.

Problems and Challenges

There are a number of problems and challenges involved in drafting well written requirements document. Requirements documentation often does not adhere to the guidelines above, as indicated by the research study findings that we started this chapter with. Bugayenko (2015) does a great job of summarizing a number of the most common mistakes to watch out for that include:

1. **Not including glossary and not using terms consistently.** For example, “*UUID is set incrementally to make sure there are no two users with the same account number.*” begs the questions if UUID and account number are the same thing or different and what is UUID. It can stand for “unique user ID” or “unified user identity descriptor” or something else. So a better way of writing this requirement may be “*UUID is user unique ID, a positive 4-bytes integer. UUID is set incrementally to make sure there are no two users with the same UUID.*”
2. **Putting questions, discussions, suggestions, and opinions in the document.** The requirements document is supposed to have answers and it’s not a discussion forum. The discussions should happen as part of drafting the document but they need to be resolved before the document is approved and used for design, implementation, and test. An extreme example from a real document broke all these rules and read “*I believe that multiple versions of the API must be supported. What options do we have? I'd suggest we go with versioned URLs. Feel free to post your thoughts here.*” (Bugayenko, 2015). It had an opinion, question, suggestion, and ask for discussion. None of these are appropriate. Instead, it should have read as something like “*Multiple versions of the API must be supported. How exactly that is done doesn't really matter.*”

3. **Mixing functional and non-functional requirements and using ambiguous terms.** This practice makes it harder to verify, trace, and implement. For example in “*User must be able to scroll down through the list of images in the profile smoothly and fast.*” scrolling is the functional requirement but “smooth” and “fast” is non-functional and ambiguous. What exactly is smooth and fast and how would I measure it?
4. **Adding details in requirement that is part of design or should be supplemental data.** For example, “*User can download a PDF report that includes a full list of transactions. Each transaction has ID, date, description, account, and full amount. The report also contains a summary and a link to the user account.*” The requirement is the ability to download the transactions. What is included in the report and the format of the transaction are implementation details that should be in design or at least in a separate section (e.g. appendix) or document.
5. **Having un-measure non-functional requirements.** It is very common to have requirements that read something like “*User interface must be responsive.*” While there are many factors that can impact speed such as hardware, internet connection, and which particular aspect of User interface we are talking about, the requirements needs some qualification of what “responsive” means so that it can be measured.
6. **Include verbiage that is just noise.** For example “*Our primary concern is performance and an attractive user interface.*” If this concern is already expressed in functional and non-functional requirements then there is no need for this statement. And if there are no requirements for it then they are missing and should be added.

While having the document reviewed by colleagues is extremely beneficial, it is not without its own mistakes and challenges:

1. **Review too early.** The document or section of the document to be reviewed should be complete and not have a lot of remaining questions. If it does, it will be wasting the reviewers time and have a discussion that may not be appropriate for that particular group of people.
2. **Size of the document.** Requirements document can get large and examining a few hundred pages can be discouraging. However, that is even more of a reason to have it reviewed thoroughly. So instead of waiting until the end, it might be beneficial to do incremental reviews during the drafting of the document. One can also select more critical and high-risk sections to be reviewed by wider audience in more formal setting and the less risky sections through peer reviews.
3. **Not having correct reviewers.** The reviews should include representatives from all users and stakeholders who are impacted by the requirements and who will use the document.
4. **Size of inspection team.** The more reviewers the harder it is to manage the reviews and reach an agreement. In such a case it is important to keep the inspection focused on finding

issues and not get side tracked on education or a person's personal opinion. It might also be helpful to have representatives from each perspective to pool their comments and just have one person there. For example, have one representative from testing group instead one from each type of test (e.g. function, system, performance, etc.) to decrease the number of people in the review.

5. **Unprepared reviewers.** When reviewers come unprepared, time is wasted and the meeting becomes unproductive. It is important to send the materials out to be reviewed with plenty of time and better to re-schedule the review meeting.
6. **Boring review.** If reviewers get bored or tired, they will become unproductive and find less issues. The review needs to be engaging and preferably the document should not be read out loud. After all, the reviewers should have already read it before coming to the meeting. Instead, it may be more useful to point reviewers to some block or section and just briefly summarize it to refresh their memory. The reviewers should then have notes to share on their questions or comments for that section.
7. **Feedback becomes about typos.** Non meaningful content comments should be provided through email and not discussed in the review meeting. Meeting should be reserved for more significant issues and comments that may need to be discussed and clarified interactively.
8. **Meetings turns into design discussion.** It is important to keep meeting focused on collecting the issues with the requirements and postpone all design discussions to that stage in the process. Otherwise the meeting becomes unproductive and does not serve its purpose.

More Examples

Here are some more examples of requirements that would not be considered well formed:

- **Contradiction:** *R1 "Five-digit user id has to be entered in 'id' text box for submit button to be active" R15 "Unique ten-digit user id is be generated by the system"*
In this example there are two requirements that refer to user id but one is five-digit and the other is ten-digit. Requirements must be consistent. Either there is a mistake and both requirements should refer to either five-digit or ten-digit OR these are two separate ids and they should not be both called "user id"
- **Un-testable:** *"System is user friendly"*
There is no measurement for "user friendly". So instead there should be specific requirements that satisfy the characteristics of "user friendly". For example there may be requirement *"User Interface is menu driven..."* which is one of the characteristics of user friendly interface.
- **Ambiguous** *"Lock user id when user enters incorrect password up to 4 times"*
Do you read this as locking up user id after 4 attempts or after 3 attempts? I would read it as up to and not including the 4th attempt but if not everyone will interpret the same way, the

requirement is ambiguous. A better phrasing may be “*Lock user id after user enters incorrect password 3 times*”

- **Not feasible** “*When entering passwords, system should switch to hiding characters instantaneously*”
Computer programs do not do anything “instantaneously”. It just may appear like this to human eye. So this requirement as written is not technically feasible. Instead the requirement may read something like “*Password text box will display typed in characters as asterisk*”

References

Beatty, J. & Wegers, K. (2013, Aug 13). Requirements Review Challenges. Retrieved December 20, 2017, from <https://www.batimes.com/articles/requirements-review-challenges.html>

Bugayenko, Yegor (2015, Nov 10). 10 Typical Mistakes in Specs. Retrieved December 20, 2017, from <http://www.yegor256.com/2015/11/10/ten-mistakes-in-specs.html>

Eriksson, U. (2017, Feb 7). How to do a requirements review from a BA perspective. Retrieved December 20, 2017, from <https://reqtest.com/requirements-blog/requirements-review-from-ba-perspective/>

How to conduct a requirements review. Retrieved December 20, 2017, from <http://www.bridging-the-gap.com/requirements-review/>

IEEE Std 830-1998 - IEEE Recommended Practice for Software Requirements Specifications

Levy, D. (2017, Mar 28). Software Requirements Specification (SRS), What you need to know. Retrieved December 11, 2017, from http://www.gatherspace.com/static/software_requirement_specification.html

McEwen, S. (2004, Apr 16). Requirements: An Introduction. Retrieved December 11, 2017, from <https://www.ibm.com/developerworks/rational/library/4166.html>

Monteleone, M. A. (2012, Jan 10). Verifying Requirements Documentation. Retrieved December 18, 2017, from <https://www.batimes.com/articles/verifying-requirements-documentation.html>

Requirements: 21 Top Engineering Tips for Writing an Exceptionally Clear Requirements Document. Retrieved December 18, 2017, from <https://qracorp.com/write-clear-requirements-document/>

Requirements Analysis. Retrieved December 18, 2017 from https://en.wikipedia.org/wiki/Requirements_analysis

Wieggers, K. (2007, Feb). Ambiguous software requirements lead to confusion, extra work. Retrieved December 21, 2017 from <http://searchsoftwarequality.techtarget.com/tip/Ambiguous-software-requirements-lead-to-confusion-extra-work>

Images Credits

Image 1: SE Process. Retrieved from https://en.wikipedia.org/wiki/File:SE_Process.jpg

Chapter 11 Requirements Validation

MAALEM, S., & ZAROUR, N. (2016). CHALLENGE OF VALIDATION IN REQUIREMENTS ENGINEERING. JOURNAL OF INNOVATION IN DIGITAL ECOSYSTEMS, 3(1), 15-21, WITH MODIFICATIONS BY UMGC.

Challenge of validation in requirements engineering

Sourour Maalem LIRE Laboratory, Mathematics and Computer Sciences Department, Ens-C Higher Normal School-Constantine, Constantine, Algeria

Nacereddine Zarour LIRE Laboratory, Computer Sciences Department, Constantine 2 University, Constantine, Algeria

Highlights

- Framing the issue of validation in Requirements Engineering.
- Classification and taxonomy of existing techniques in requirements validation.
- A validation techniques is intended for a particular area.
- The combination of validation techniques is essential.
- Several iterations are necessary because of the multidisciplinary projects.

Abstract

This paper will review the evolution of validation techniques and their current status in Requirements Engineering (RE). We start by answering the following questions: What validate? Why the benefits of having the requirements validation activities during the RE process? Who are the stakeholders involved in the requirements validation process? Where applied the validation in the RE process? and How the techniques and the approaches of requirements validation?

Introduction

To error is human, and there is no reason to think that it does not occur during the development of the system. Problems can result from a misunderstanding between analyst and the customer an ambiguity in the documentation, etc. Errors that occur at this stage and are not corrected are often the most persistent and costly. It is therefore important to set in motion steps that will minimize errors, detect and correct them as soon as possible. Error prevention is a matter of good practice in software engineering. However, it is wise to assume that errors will occur and establish procedures to prevent. Thus, the requirements engineering process (such as sub-processes of the larger systems engineering and software engineering processes) must be validated. Validation has the purpose of ensuring that the correct functionality of the solution-system has been defined:

- If the problem domain behaves as described (in the requirements document);
- If the requirements are properly recorded;
- If the new system behaves as described (in the requirements document);
- If the inventive step (design interactions) is correct when the requirements are met.

The objective of validation is to ensure that all manufacturing steps result in a product that meets the requirements of stable and reproducible way. While the objective of requirements validation is to certify that the requirements on the set of specifications conform to the description of the system to implement and verify that the set of specifications is essentially: complete, consistent, consistent with standards standard, requirements do not conflict, does not contain technical errors, the needs are not ambiguous, etc. During our study in requirements validation, a problem set are appeared, we have classes in: Problems associated with validation in the software life cycle: the nature of information, what? How? When? Who? How (by what means technical)? Where? Duration the position of the validation activity compared to the software life cycle, etc. Problems related to validation during the requirements engineering process: is what an activity or phase? Which is the result of the validation, how can we validate requirements? What types of validation processes is best suitable for a project? How to ensure that the solution meets the needs of stakeholders and company? What is the best technique to use in validating? How to agree all stakeholders? The different validation modes (formal, semi-formal, informal) level verification model where requirements, validation of non-functional requirements and functional control of changing requirements, insufficient in negotiation techniques for validation, the lack of activity of Validation in RE in some ways, the lack of validation methods, lack of expert analysts, lack of business experts with a high level of analytical and communication and experienced users, etc.

The paper is organized as follows: After the introduction we will present the what, validation in RE and some quality criteria that must be based evaluation of requirements to Section 2. Then we describe the Why requirements validation in Section 3, before giving in Section 4 Who should be involved in the process. In Section 5, we will see, speak as validation against the RE process; Spent Section 6 for the techniques that can be used during requirements validation process. Finally we come to a conclusion and some prospects.

What is requirements validation?

Many areas merge between the definitions of validation and verification. Thus, it is necessary to agree on their explanations.

According to Artem Katasonov [1] Validation of the requirements is the process to determine whether the requirements as defined, do not contradict the expectations of the various stakeholders of the system and do not contradict each other; this is the control of the quality requirements. Requirements validation is concerned with the process of review of the requirements document to ensure that it defines the right software (the software that users expect). According Kotonya and al. in [2]“requirements validation is concerned to check the requirements document for consistency, completeness and correctness”, and in [3] states that the requirements should be checked to: validate, understand, consistent, traceability, completeness, realism and verifiability.

Because the terms verification and validation are often confused, Terry Bahill [4]defined requirements verification as a process to prove that each requirement has been satisfied. Verification can be done by logic, inspection, modeling, simulation, analysis, examination, testing or demonstration. Requirements Validation to ensure that (1) all of the requirements are: correct, complete and consistent, (2), a model can be created that meets the requirements, and (3) real-world solution to be built and tested to prove that it meets the requirements (see Fig. 1).



Fig. 1. Requirements validation process

The requirements validation process is not so clear. According to the EIA632 standard, the requirements validation process ensures that the requirements are necessary and sufficient for the appropriate design phase to meet the exit criteria for the lifecycle software phase and lifecycle phases of the company in which efforts occur for the engineering phase or reengineering.

Why requirements validation?

There are processes models in RE, which do not take the validation as sub-phase during the RE overall process [5,6]. To ensure a better support of requirements, requirements must be good quality; the guarantee of that quality is assured through the stages of validation and verification. These activities take place throughout the life cycle, when approving the interim filings for reserved phases. Most of the existing methods or practices are only to identify and gather the requirements. Compared to customer needs, validation activities fit naturally in the harsh process [7]. Jose and al in [8] show that only a few approaches provide techniques for requirements validation. Most of them only set guidelines on how developers and customers will need to review the specification of requirements to find inconsistencies and errors and to complete. A comparative study between web development methodologies and supported the activities of the RE process, they mention that only 4/10 of the methods considered by this validation and mention technique without any approach or methodology.

Lulu He and all [9] realizes that the requirements validation is often not sufficiently covered not only in the practical world but even in the academic world. As said Siew et all [10] consider that most books present it as a list of “best practices” and this validation requirements as much as a heterogeneous process based on the application of a variety of independent techniques. In their towers Nuseibeh B and all in [11] confront the problem of validating requirements with the problem of validation of scientific knowledge. Yet the requirements documents may be procedures for validation and verification. Davis [12] explains that the requirements can be validated to ensure that the analyst to understand the requirements and it is also important to check that a document meets the standards (standards) of business, and that it is understandable, consistent and complete. Similarly Bryne [13] and Rosenberg [14] said, it’s normal to explicitly provide one or more points in the process where the latter requirements are validated. The purpose is to identify or collect any problems before resources are committed to meet the requirements. In [15] Yves.C says that requirements must be validated by the various stakeholders. This validation is done at several levels. Future users often to validate as of writing, the client typically validate the entire document. And ends with the conclusion that the monitoring requirements validation is not an easy task. He defined validation as a process of obtaining, on the part of all stakeholders, a formal agreement on the specified requirements.

Who validate requirements?

Stakeholders are the actors of the RE process, they are the individuals involved in its implementation. They are identified by their role and not individually.

Requirements engineering reveals actors who are interested in the problem or its solution:

- Customers, users, domain experts
- Software Engineers, Requirements Engineers
- Project Managers

The role of a party during the RE process is represented by Gerald Kotonya and Sommerville [2] in a model. The latter brings up the RE process with stakeholder responsible, it starts Understand the business problem by performing (Requirements engineers, domain experts and end users) and the activity Establish requirements outline accomplished by (Requirements Engineers, end users), followed by the activity carried out by *Select prototyping system* (software Engineers, project Managers) suite activity *develop prototype* execute by (Engineers needs, software Engineers) and end the Evaluate activity *prototype completed* by (the end user, domain experts, Requirements Engineers, software Engineers).

The largest number of works in RE, does not explain the exact role of the Stakeholders in requirements validation. Sharp in [16], argue that the literature does not even distinguish between the roles of individuals or groups and in [17]we proposed a methodology for collaborative requirements validation in distributed platforms. Where we have determined in detail the roles and skills of stakeholders, the results of each activity in the validation process, and the means for its implementation. Selection of participants is based on certain criteria that differ across phases. We clung especially to have different points of view and to call for complementary multiple skills. We suggest selecting participants outside the development team. They will be more objective because less involved in the project. It is then necessary to assign a role to each. It is obvious that a participant may possibly take several roles. The roles and competences are summarized in Table 1.

Table 1. Roles and competences of stakeholders in requirements validation.

Stakeholder	Intervention	Roles	Competences and expertise
Analyst	Complete process	He is a moderated, direct the discussion. He prepares the meetings and ensures the sequence of steps. He ensures for the conduct of business objectives and maintain attaches not to neglect human factors. He presides over the decision.	Analysis of IS, animation and communication, order, decision, negotiation
Customer	Validation	Identify needs read the requirements to verify the correspondence with needs.	Communication
Managers project	Inspection	Use of specifications to plan supply and the development process of the	Problem domain management cost, delay,

		system	technical communication
Domain experts, domain problems and solutions communication	Validation	Identify Functional requirements	Domain problems and solutions communication
End user	Validation	Spread Functional and non functional requirements, organization, context, constraint	Domain problems and solutions knowledge of computers (operating systems, software, hardware)
System engineers and developer	Verification	Use requirements to understand the system under development	Communication HMI
System test engineers	Verification	Use the requirements to develop validation tests for the system	Test enable communication
System maintenance engineers	Maintain de la validation	Use requirements to help understand the system	Communication
Designers (including realized earlier versions)	Verification	Detailer et completer les requirements	Communication solution domain

When validate requirements?

The RE process can be seen as a set of activities containing a structure in each activity. These activities include words such as who will be responsible for each activity inputs for an activity and outputs generated by this activity, etc. According to Leite et al. [18] “The whole of the requirements engineering process is a subprocess of fabric, and it is very difficult to make a clear distinction between them”.

In [19], Pohl presented the RE process in three dimensions; representation, agreement and specification. The needs are discovered and described in accordance with a system of representation in the Dimension representation. The needs are traded based on their priority, costs and risks of their achievement during the Dimension consensus and end the specification Dimension where needs must conform to standards. Only after two years he joust validation and presents a process with 4 Elicitation activities, negotiation, specification and validation.

Loucopoulos, Karakostas [20] uses the appointment activities (not phase) of the RE process: Elicitation, Specification, and Validation. They present as a retroactive loop that passes from one activity to another in all ways, from the user to the problem domain.

Kotonya, Sommerville [2] keep the same appointment activities with more details. Feasibility study, Elicitation and requirements analysis, negotiation, documentation, validation and adds Management which is the change management process system requirements.

Larry Boldt, [21] presents the RE process in a hierarchical, fractioning task of creating the requirements of development and management. The development is divided into Elicitation, Analysis, Specification and Verification. Larry does not use the word in its RI validation process model.

Gerald Kotonya Ian Sommerville [22] presents a business model Coarse-Grain in quell; the RE process activities and continuous activity where:

The activities are: the elicitation, the analysis and negotiation and ongoing activities that are documentation, validation, approval. Ian Sommerville follows [23] improves the process presented a general consensus of the activities of traditional requirements engineering process to measure that consists of five main areas of process analysis, elicitation, negotiation, documentation, validation and management. Where the documentation and management are presented as a nucleus and turn them the remains of activities.

Like all phases of the RE process or activity, and produces a result which justifies its presence in all RE process models. It is necessary to provide an evaluation at each level activity. We thus providing a more general approach than that found in the literature that deals validation as an activity and not as a phase, which presents the validation as an ongoing, incremental, collaborative, which takes place throughout RE process, see Fig. 2.

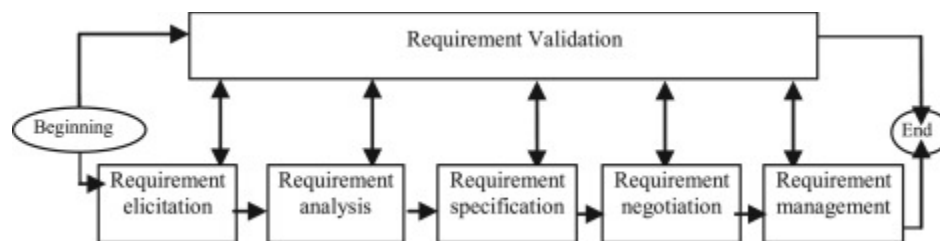


Fig. 2. When validate in requirements engineering.

The result of the validation in each step has an increment of the solution which it constructed as one moves in the process. Each increment will make the results of its phase carried out by the stakeholder concern and often generates a heterogeneous product, which continuously requires another pass.

How to validate requirement?

Most existing methods and practices aims to identify and gather the requirements. Due to delays and other considerations, the validation is done informally, either on an ad hoc basis or simply peer review [24]. Different organizations are possible, from simple personal interpretation to the highly organized and formalized review (walkthroughs).

There are different types of requirements validation techniques available in the literature, some of which are summarized in Table 2, with respect to their objectives, in particular that detects and ensures that?

Table 2. Summary of requirements validation techniques.

Summary of approach

Although the literature mentions several approaches to validation requirements, all can be

Techniques	Objective of the technique	What is detected?	Stakeholder
Techniques	Objective of the technique	What is detected?	Stakeholder
Pre revision	Saves the cost and time for revision	- Faults spelling	Involves people from different backgrounds
		- Not Standards compliance	
		- Typographical errors	
		- Requirements missing	
Revision of requirements	Minimizing changes and changes in the software	Identify inconsistencies, conflicts, omissions, etc.	Involves customers and developers
Inspection of requirements or Fagan inspection [25]	Make understandable product	Defects in artifacts	Used by people who study the state of the art
Inspection based test-cases Gorschek [26] , Nina [27]	Ensure that the requirements are good enough for the product and business planning is also	Eliminate defects before the project begins and during the project	The project manager and the tester
Reading techniques Gilb [24] , O. Laitenberger [28] , Laitenberger [25] , O. Laitenberger [24] , T. Thelin [27] .	Show how to read and what to look in Artifact	The mistakes, Typographical errors	The project manager, analyst
Prototyping	Understand the requirements of the system	The critical situations and panes, the blockages, etc.	End user, analyst
Based-model	Maintain traceability	The formulation, structuring, etc.	Designer, tester, developer
Based viewpoint Leite et al. [29]	Cover all requirements met, the different views	Conflict, consensus	Customer, developer, project manager, Tester
Based-test Wendland et al. [30]	Examine each requirement and derive a series of tests. It can define one or more tests that can be run when the system will be developed	Mainly involve the appropriate definition and data integrity, consistency, non-ambiguity and testability requirements.	Developers tester

classified according to the level of formality of the specification with what starts the specification validation process (formal, Semi-Formal, Informal) Terminated has a solution (automatic, Semi-Automatic or Manual) generalized into two levels of evaluation: an internal level, this activity is provided by the RI team, externally are appealing to customers, Mixed or that involves the work of master guided by the prime contractor. Applied on examples of systems that can be Critique (Embedded, Real-time, etc.), management Ordinary (see [Table 3](#)).

Table 3. Synthesis of validation approach in RE.

Levels	Approaches					
	Math and logic	Natural language	UML	Empiric	Expert system	Formal method
Level of specification formalism Formal/semi-formal/informal	Formal	Informal	Semi-formal	Informal	Informal	Formal
Automatically level Automatic/semi-automatic/manuela	Automatic	Automatic	Semi-Automatic	Manuel	Automatic	Automatic
Level of evaluation Interne verification/external validation/mixed	Interne	Interne	Mixed	Interne	Mixed	Interne
Type de system Critique/ordinary	Critique	Ordinary	Ordinary	Ordinary	Ordinary	Critique

It is interesting to understand the current state of practice of requirements validation. In a rapidly growing market and continuous change. The latter having topical work involving several technical requirements validation justifying none is perfect or sufficient. The detection of the methodical approach becomes more topical in hollowing work, comparing with older approaches that fail her approach and the clarity of the validation process for (who does what why when and how) The Table 4, shows the results of a synthesis of recent research in requirements validation analyzed to detect, what validate, validate why, when validating and how to validate.

Table 4. Synthesis of current work of requirements validation.

Approaches	What	Why	Who	When	How
Shahid Nazir Bhatti [31] [2015]	Functional and non functional requirement	Efficiency usability, functionality, portability,	Elicitor	Elicitation	Prioritization - Inspection -

Approaches	What	Why	Who	When	How
		maintainability			
Luca Sabatucci [32] [2015]	Requirement specification	Correct understand	Non-technical users\analyst	Specification	Scenarios -Test -
S. Zafar Nasir [33] [2015]	Requirement model	Maintenance	Enterprise manager	Validation	ERP data - model -
Alberto Rodrigues da Silva [34][2014]	Formal specification	Consistency, completeness, unambiguousness	Domain experts	Specification	Natural Language Processing (NLP) - reading techniques -

Conclusion

The literature tends to consider the validation of requirements, as a heterogeneous process based on the application of a variety of independent techniques; without being able to specify: What, Why, Who, When and How to validate the requirements. Requirements validation techniques play a pivotal role, to detect possible defects in the requirements. They are summed up in: the review, inspection, reading techniques, prototyping, validation model-based, validation test-bases and validation perspective -based. These techniques can help in the implementation of projects within the timeline, budget, and according to the desired functionality. The prospects for this sought after area varies between: transformation model, automatic generation of prototype, natural language processing to maintain the validation in the requirements management process that requires frequent review of all documents.

References

- [1] Artem Katasonov, Requirements Management and Systems Engineering, in: Lecture 6: Requirements Validation and Verification (ITKS451), University Of Jyväskylä, Autumn, 2008.
- [2] Gerald Kotonya, Ian Sommerville, **Requirements Engineering Processes and Techniques** John Wiley & Sons, England (1998)
- [3] G. Kotonya, I. Sommerville, **Requirements Engineering: Processes and Techniques**, John Wiley & Sons (2000)
- [4] A.Terry Bahill, Steven J. Henderson, Requirements development, verification, and validation exhibited in famous failures.
- [5] <http://www.sumitrawat.net/2009/09/requirements-engineering-for-software.html>
- [6] Karl E. Wiegers, **When telepathy won't do: Requirements engineering key practices** <http://www.processimpact.com/articles/telepathy.html>
- [7] Sommerville, P. Sawyer, **Requirements Engineering: A Good Practice Guide** John Wiley & Sons (1997)
- [8] José Reinaldo Silva, Eston Almança Dos Santos, Applying petri nets to requirements validation, in: 17th Int. Congress Of Mechanical Engineering 2004.

- [9] Lulu He, Dr. Jeffrey C. Carver, Dr. Rayford B. , Using inspections to teach requirements validation, Vaughn Mississippi State University, January 2008.
- [10] Siew Hock Ow, Mashkuri Hj. Yaacob, **A Study on the Requirements Review Process in Software Development: Problems and Solutions**, IEEE (1997)
- [11] B. Nuseibeh, S. Easterbrook, **Requirements engineering: A road map** Proc. of International Conference on Software Engineering, Limerick, Ireland, June 2000, Association of Computing Machinery (ACM) Press (2000), pp. 37-46
- [12] A.M. Davis, **Software Requirements: Objects, Functions and States**, Prentice Hall (1993)
- [13] E. Bryne, IEEE standard 830: Recommended practice for software requirements specification, in: Presented at IEEE International Conference on Requirements Engineering, 1994.
- [14] L. Rosenberg, T.F. Hammer, L.L. Huffman, Requirements, testing and metrics, in: Presented at 15th Annual Pacific Northwest Software Quality Conference, 1998.
- [15] Yves Constantinidis, **Expression des Besoins pour le Système d'Information Guide d'élaboration du Cahier des Charges**, Edition Eyrolles (2012)
- [16] Sharp, Finkelstein, Galal, Stakeholder identification in the requirements engineering process.
- [17] Mâalem Sourour, Zarour Nacereddine, Rénovation VECOD®, Méthodologie de validation des exigences collaborative dans les organisations distribuées, IWoRE 2013: International Workshop on Requirements Engineering Constantine-Algeria.
- [18] Julio Cesar Sampaio do Prado Leite, Peter A. Freeman, **Requirements validation through viewpoint resolution**, IEEE Trans. Softw. Eng., 17 (12) (1991)
- [19] Klaus Pohl, **The Three Dimensions of Requirements Engineering**, Springer (1993)
- [20] P. Loucopoulos, V. Karakostas, System requirements engineering: Chapter 2—processes in R.E.! p. 4.
- [21] Larry Boldt, **Trends in Requirements Engineering People-Process-Technology**, Technology Builders, Inc. (2001)
- [22] Gerald Kotonya, Ian Sommerville, **Requirements Engineering**, John Wiley and Sons (2004)
- [23] Ian Sommerville, **Integrated Requirements Engineering: A Tutorial**, IEEE Computer Society (2005)
- [24] T. Gilb, D. Graham, **Software Inspection**, Addison-Wesley Publishing Company (1993)
- [25] Michael Fagan, **Design and Code Inspections to Reduce Errors in Program Development** IBM Syst. J., 15 (3) (1976), pp. 182-211
- [26] Tony Gorschek, Nina Dzamashvili Fogelström, Test-case driven inspection of pre-project requirements-process proposal and industry experience report, in: Proceedings of the Requirements Engineering Decision Support Workshop Held in Conjunction with the 13th IEEE International Conference on Requirements Engineering, 2005.
- [27] Nina D. Fogelström, Tony Gorschek, Test-case driven versus checklist-based inspections of software requirements—an experimental evaluation, WER07 Workshop Em Engenharia De Requisitos, Toronto, Canada, May 17–18, 2007.

- [28] Oliver Laitenberger, **A survey of software inspection technologies**, Handbook on Software Engineering and Knowledge Engineering, Fraunhofer IESE (2002)
- [29] Julio Cesar Sampaio Do Prado Leite, Peter A. Freeman, **Requirements validation through viewpoint resolution**, IEEE Trans. Softw. Eng., 17 (12) (1991)
- [30] M.-F. Wendland, I. Schieferdecker, A. Vouffo-Feudjio, Requirements-driven testing with behavior trees, in: IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, ICSTW, March 2011, pp. 501–510.
- [31] Shahid Nazir Bhatti, Maria Usman, Amr A. Jadi, **Validation to the requirement elicitation framework via metrics**, ACM SIGSOFT Software Engineering Notes Archive Volume 40 Issue 5, ACM, New York, NY, USA (2015), pp. 1-7
- [32] Luca Sabatucci, Mariano Ceccato, Alessandro Marchetto, Angelo Susi, **Ahab's legs in scenario-based requirements validation: An experiment to study communication mistakes**, J. Syst. Softw., 109 (2015), pp. 124-136
- [33] S.Zafar Nasir, Tariq Mahmood, M.Shahid Shaikh, Zubair A. Shaikh, **Fault-tolerant context development and requirement validation in ERP systems**, Comput. Stand. Interfaces, 37 (2015), pp. 9-19
- [34] Alberto Rodrigues da Silva, **Quality of requirements specifications: a preliminary overview of an automatic validation approach**, Proceeding SAC'14 Proceedings of the 29th Annual ACM Symposium on Applied Computing, ACM, New York, NY, USA (2014), pp. 1021-1022

Chapter 12 Managing Requirements

McFADDEN, R. (2017). SOFTWARE REQUIREMENTS – REQUIREMENTS DOCUMENTATION, UNDER CONTRACT FROM UMGC.

What is Requirements Management

There are many activities and tasks associated with requirements throughout the project and across releases and Requirements Management (RM) is the overall process of controlling and tracking these activities.

Within a particular project, the RM process makes sure that the activities and tasks associated with requirements are on track and therefore it has a significant impact on success of the project. It starts with elicitation and analysis of requirements and then consists of negotiations of requirements with the stakeholders; prioritizing between the stakeholders; defining, getting approvals and commitments, and maintaining up to date requirements documents; keeping track of requirements updates throughout the project to include reviews, negotiations, approvals, and the implementation; mapping requirements to design, source code, and test cases and maintaining the mapping and tracking.

The more requirements, the more critical is the RM. Imagine dealing with hundreds or even thousands of requirements. How would you make sure that requirement number 435 is implemented? Or if the requirement number 372 changes, how do you determine what other requirements also have to be updated? Or how do you determine what percentage of requirements are implemented and tested? All of these and more are the reason we need requirements management.

Across the releases and projects, RM includes maintaining and keeping track of backlog of requirements, prioritizing and negotiating the requirements for a particular release and/or project, and estimating requirements.

Storing Requirements

Initially requirements are often documented in word processing documents and once finalized they may be stored in other formats such as spreadsheets, databases, or in a number of project or requirements management tools (e.g. IBM Rational, Enterprise Architect, Jama) to help with management and tracking. Whatever the means and format is used to store, some factors that need to be considered include:

- The ease of keeping documents current
- The ease of communicating changes to the requirements
- The ease of storing and use of the information
- The ease of tracking requirements to other project artifacts

In addition to the requirement information, other attributes may also be assigned and stored with the requirement as shown in the table below:

Attribute	Description	High-Level	Mid-Level	Detail (Testable)
Unique ID	A unique identifier for the requirement	X	X	X
Date Created	Date the requirement was first captured	X	X	X
Source/Ownership	Name of the person, organizational division or change request where the requirement came	X	X	
Functionality	Functional capability or constraint (Non-Functional category)		X	X
Requirement Type	Functional/Business, Non-Functional/Technical, Transition, Stakeholder, ...	X	X	X
Requirement	Concise description of requirements expressed in business terminology to ensure common understanding	X	X	X
Status	Reflects the current state of the requirement. Examples: Proposed, In Review, Approved, Denied, Verified, Implemented	X	X	
Priority	Indicates the importance of the requirement (Critical, High, Medium, Low)	X	X	
Rationale/Origin	Reflects backward (originating source) traceability and justifies the requirement	X	X	X
Difficulty/Complexity	Qualitative measure of how difficult the requirement is to implement		X	
Stability	Indicates how mature the requirement is - can it be worked on?	X	X	
Test Case ID	Reflects forward traceability to validation test case used during the Acceptance phase of			X
Assigned to	The person responsible for completing the requirement		X	
Risk	Assigned based on the expected associated level of effort and/or complexity/difficulty		X	
Impact	Business or technical effect of implementing this requirement	X	X	
Due Date	Target date for the deliverable associated with the requirement		X	
Stakeholders	Identified business staff responsible for approving changes that impact the requirement		X	
Approved By	Person who approved this requirement		X	
Approval Date	Date the requirement was approved		X	
Release Number/Build	Application release number targeted for the implementation of this requirement		X	X
Dependencies	Identification of other system elements and/or requirements that may be impacted by changes to this requirement		X	
Comments	Text field available for analyst, approver, etc. notes related to this requirement	X	X	
Verification	Specification of how this requirement will be tested (e.g., demo, analysis, inspection, walkthrough, QA test scripts/test cases)		X	
Subsystem(s)	Applications/systems where this requirement will be implemented	X	X	
Actual Iteration	Identifies the Actual, rather than the Planned iteration (Release Number/Build) where the requirement was implemented			
Revision Number	Used to track different versions of the requirement		X	
Cost	Quantified cost attributable to this requirement	X	X	
Defect	ID(s) of any associated defect(s)			X
Obsolete	Indicates that the requirement has been removed/is no longer active	X	X	X
Benefit	Need or goal that this requirement supports	X		
ROI/ROE	Associated return on investment/effort/... that may be expected when this requirement is implemented	X		

Image 1: List of commonly used Requirements Attributes

Requirements traceability

Traceability is a big part of the Requirements Management. It includes the identification and documentation of requirements in both downward and upward path and association to other artifacts whether other requirements, design, code, and tests within the project.

Traceability can be both pre- and post- requirements. Initially it can be used to trace the requirements to the source of the requirement whether a person or a group who asked for it as part of the requirement elicitation. This information can then be used for prioritization and for user testing. Then once requirements are established they can be traced to other artifacts associated with them.

The key for requirements traceability is that the connection is bidirectional so it can assist in management of scope and changes to the scope; assist in controlling and managing requirements changes; provide a road map to link all documents related to a requirement; help with verification and validation that all requirements are fully implemented; and help with planning of testing.

Creating and using requirements traceability is often considered an extra expense that some stakeholders may want to avoid. However, the benefits may minimize the overall cost and proper tools may significantly decrease the initial investment. For example, a number of studies demonstrate that requirements traceability improves productivity of completing development tasks when traceability is available (24% faster with 50% more correct) and it helps avoid software defects.

Tracing requirements can also have a number of other benefits such as:

- When a requirement changes it makes it easier to find the other artifacts that need to be updated for that change
- Helps to make sure that no requirement is missed especially in certifying safety-critical systems
- Allows keeping more accurate project status and progress by analyzing the traceability data
- It helps a new person entering the project or when future work is needed to visualize the relationships between the artifacts
- When tests fail, having the traceability helps to determine where the root cause of the issue was introduced (e.g. code, design, or requirement)

There are a number of methods for visualizing relationships between project artifacts using traceability. Some common ones include:

- **Traceability matrix** – table that maps one requirement to all the other artefacts (See Image 2 below). When there is a relationship the cell will have a value, otherwise it won't. The advantage of this method is that it is easy to see the relationships but for a large project it can get very large. This method can be especially useful for managing tasks.
- **Traceability graph** – each artifact is represented by a node with edges connecting those that have a relationship. A graph helps in identifying a missing artifact when a link is missing. This method may be useful for managing development tasks.
- **List** – each entry includes information about the source and target artefact and the attributes.
- **Hyperlink** – connect artefacts and allow jumping from source information to target artefact information. It allows accessing more detail information easily and to view the original documents as opposed to having that data summarized or repeated

Requirements Test Count: Tests by Task

Task	Total # QA Tests	Functional Requirement ID																	
		Camp_01	Camp_04	ClinOps_01	ClinOps_02	ClinOps_03	Event_01	Event_03	Event_04	Event_05	Form_02	Form_03	Form_04	Form_07	Loc_01	Loc_02	Loc_03	Medication_02	NYSHS_04
Add Patient Records	104			1	79	24													
Associate Medications	16								16										
Associate Providers	13									13									
Barcode Generation	14																	7	
Cancel Reservation	27																		27
Correct Invalid NDCCodes	6																	6	
Create a New Form	33										31	2							
Create Event	33						31	2											
Create Location	15														12	3			
Create New Campaign	26	25	1																
Edit Archived Form	1												1						
Edit Form	18										5	13							
Edit Locations	4																4		
Edit Published Form	2											1	1						
Edit Unarchived Form	5												5						
Edit Unpublished Form	3												2	1					
Edit Copy Existing Form	9										2	7							
Export Patient Records	1				1														
Generate Paper Form	2				1						1								
Total Tests per Functionality:	332	25	1	1	81	24	31	2	16	13	39	23	9	1	12	3	4	7	6
																			27
																			7

Image 2: Matrix showing tasks against functional requirements in order to facilitate traceability

Collecting requirements status

Managing requirements also means managing requirement status. This is done using requirement's status field in requirements management tool (or some other method such as spreadsheet) and through requirements status meetings. Both traditional and agile development approaches have regular status meetings to discuss the progress of the project. So for example, development team will meet to discuss their progress and any issues that have to be resolved and summarize this status for their manager or technical lead, and the test team will do the same. Then the development and test status is combined along with other teams' status such as documentation, build, etc. and project manager may hold a status meeting with stakeholders or higher level managers summarizing the project status and progress and any critical issues that have to be resolved.

Using requirements management tool makes collecting status easier and more accurate. For example, some typical status values includes:

- Proposed – a stakeholder submitted a requirement
- Approved – the requirement has been analyzed and assigned to a specific project and approved by stakeholders
- In progress – requirement is being worked on
- Implemented – the requirement has been designed, code written, and unit tested

- Verified – the requirement has passed acceptance criteria
- Deferred – requirement has been deferred to a future project
- Deleted – the requirement has been removed (there is explanation why)
- Rejected – the requirement has not been approved and is not planned to be implemented (there is explanation why)

When status is updated regularly in requirements management tool, project manager can generate reports and graphs to monitor the progress of the project relative to requirements. For example he can check what percentage of requirements is complete or how many are waiting for implementation or verification.

In Agile development, where user stories are used as requirements, similar status can be tracked:

- In backlog – user story with initial information
- Defined – details of the user story has been completed and acceptance tests were written
- In progress – it is being worked on in current iteration
- Completed – implementation and unit test has been completed
- Accepted – user story has passed the acceptance criteria
- Blocked – the user story is blocked by something and cannot be worked on at this time

Managing change to requirements

Leffingwell & Widrig (1999) gave an excellent analogy of the problem with getting correct requirements from customers on the first try: a customer may say “‘Bring me a rock.’ But when you deliver the rock, the customer looks at it for a moment and says, ‘Yes, but, actually, what I really wanted was a small blue rock.’ The delivery of a small blue rock elicits the further request for a spherical small blue rock. Ultimately, it may turn out that the customer was thinking all along of a small blue marble—or maybe he wasn't sure what he wanted, but a small blue marble—well, perhaps even a cat's eye small blue marble—would have sufficed. And he probably changed his mind about his requirements between the delivery of the first (large) rock and the third (small blue) rock.”

Changing requirements are unavoidable for a number of reasons. For example, requirements may change when the platform the product needs to run on changes, business needs change, regulations or standards that the product needs to adhere to change, there was an issue in the original requirement definition, there is a change or limitation in the technology used, or there is a change in security technology, to name a few.

Whatever the reason, when a requirement has to change (or a new one added) there are a number of activities that need to be done. The change has to be analyzed and depending on the size and complexity of the change, it may have to be approved. When the change is identified later in the development cycle, it may need to be analyzed for costs versus benefits and the impact it will have on the product and the schedules.

Then it has to be documented and the appropriate stakeholders need to be notified so that the corresponding artifacts are also updated such as supplemental documents (e.g. graphs, models, or diagrams), design documents, code, acceptance criteria, and user guides or other technical documentation for the product.

A simplified change process may look like this:

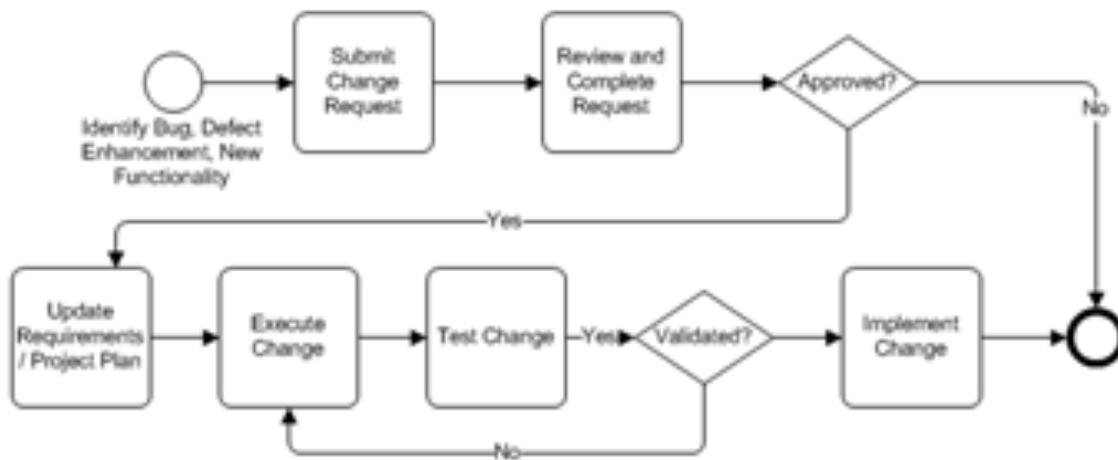


Image 3: Change Management Process diagram

Mistakes and challenges in managing requirements

When requirements are not managed properly, it can result in project failing or going over budget and/or deadline. Some of the typical issues include:

- Not having well defined requirements when developing the product.
- Having ambiguous and open to interpretation requirements and not specifying acceptance criteria.
- Not identifying or not involving in the project some key stakeholders, thus causing late changes to the requirements.
- Not setting or following management and control process to minimize requirements creep (product scope increase).
- Allowing developers to add functionality to the product that was not in the requirements and not needed or approved.
- Having conflicting requirements that are due to different stakeholders having different needs and not resolving these properly

There are different reasons across organizations why there is no proper care in making sure that requirements are managed correctly. Despite overwhelming research showing that managing requirements is critical to success of a software project, some organizations still believe RM costs too much especially for small projects or that it is not needed.

Some experts cite not having enough people or enough time or skills for using RM tools for requirements management and yet agree that their projects suffer from product failures, and not meeting budget and deadline goals. So while it is a matter of having initial investment that will pay for itself in the life of the project, it may still be difficult at times to convince the stakeholders to agree to that investment.

What about Agile RM

While agile approaches to development strive for minimal and just in time documentation and concentrate more on implementation, they still need to take the same care with requirements management to make their projects successful.

Just like in other approaches, Agile needs to collect the requirements, often in the form of epics and user stories, which represents the product backlog. This backlog has to be organized and managed such as assigned priorities, work estimated, meet with stakeholder to define details, create acceptance criteria, and assign user stories to a specific team and then assign to an iteration. Unlike traditional approaches however, the emphasis is less on documenting the requirements but more on effectively managing them using close collaboration with the stakeholders.

Some best practices for agile requirements management include:

- **Review requirements collaboratively** – requirements should be worked on collaboratively with the development team and stakeholders to discuss and negotiate the details and design the best solution
- **Use visuals for requirements** – using graphs, diagrams, models, and prototypes allows a quicker and clearer understanding and assists in the discussion of the needed details. This is true for any development and not just agile
- **Manage change to the requirements** - while agile is geared toward easier change, it still needs to be managed with documents updated, correct stakeholders involved in working out the changes to include dependencies and impacts on other requirements
- **Organize requirements documentation** – the documents need to be organized to make sure that the current state of the development is tracked, requirements are worked on in the correct order by priority, documents are updated in timely manner and accurately, etc.
- **Keep requirements small** - smaller requirements are easier to estimate accurately, prioritize, and provide greater detail
- **Automate traceability and avoid repeated data** – when projects get bigger, traceability helps in change management and to keep project artifacts linked to each other. Manual traceability however may be too costly for agile

Government RM guidelines

Some government agencies may have documentation outlining what processes need to be in place for requirements management of their projects. For example, Centers for Disease Control and Prevention (CDC) has a document that provides terminology definitions and detailed guidance on requirements gathering, requirements traceability, best practices, and appropriate practice activities. The document can be accessed at https://www2.cdc.gov/cdcup/library/practices_guides/CDC_UP_Requirements_Management_Practices_Guide.pdf

Tools

Requirements management tools help manage requirements documents, their dependencies, and the tasks associated with managing requirements. Some of the typical features a tool may have include:

- Creating and updating requirement
- Assigning requirement to the next step and person responsible
- Tracking of requirements through the development cycle and artifacts
- Document repository
- Version control of the requirement
- Generation and updating of the requirements documents
- Management of use cases (creation, users/actors, traceability)
- Generation of reports
- Change control management (request, review, impact analysis, approval)
- Management of permissions and assignment of roles and tasks
- Management of approvals
- Test case management
- Issue and defects log and management

Thus, there can be significant benefits of using a requirements management tool for a project, specifically decrease in time it takes to complete requirements tasks (and therefore faster time to the market) and improvements in the quality of the product.

Requirements management tools include benefits such as:

- Ease of creating, updating, and monitoring requirements
- Less chance that a requirement will be missed
- Easier collaboration
- Easy to enforce and monitor change control
- Ability to easily generate reports and collect progress status

However, selecting the right tool or tools for requirements management is not a trivial decision and there are many factors to consider. For example, for a project with 200 requirements or less and a small team of 5 co-located personnel, one could just use a spreadsheet, or word processor document, or a simple database. But for a project with 2,000 requirements or more and/or a large distributed team, it would be cost effective to invest in a commercial requirements management tool.

Some tools can be used “out of the box” and provide relative ease of use with some scalability and flexibility without high consultancy or setup costs. One example of this type of tool is Accompa Requirements Tool (<http://web.accompa.com/requirements-management-software/>).

Other tools are geared toward larger projects and provide a fully integrated solution for managing requirements through the whole lifecycle. They are a bigger investment as far as cost, setup, learning curve, and ease of use. One example for this type of tool is IBM Rational DOORS (<https://www.ibm.com/us-en/marketplace/rational-doors>).

Finally, some tools may also be geared toward specific areas or tasks of requirements management such as requirements elicitation versus specification versus traceability. Examples of free and fee tools for these and other categories can be found at:

https://en.wikipedia.org/wiki/Software_requirements

<https://www.capterra.com/requirements-management-software/>

References

Agile Requirements best Practices. Retrieved December 28, 2017, from <http://www.agilemodeling.com/essays/agileRequirementsBestPractices.htm>

Beatty, J. (2016). Tracking Project Status with Requirements. Retrieved December 25, 2017, from <http://www.seilevel.com/requirements/tracking-project-status-requirements>

Common Mistakes in Managing Requirements. Retrieved December 27, 2017, from http://www.requirementsmanagementschool.com/w1/Common_Mistakes_in_Managing_Requirements

Documenting and Managing Requirements. Business Analysis Guidebook. Retrieved December 22, 2017, from https://en.wikibooks.org/wiki/Business_Analysis_Guidebook/Documenting_and_Managing_Requirements

Leffingwell, D. & Widrig, D. (1999). Managing Software Requirements. Retrieved December 21, 2017, from <https://doc.lagout.org/programmation/C++/Addison%20Wesley%20-%20Leffingwell%20%26%20Widrig%20-%20Managing%20Software%20Requirements%2C%201St%20Edition.pdf>

Linman, D. (2012, Dec 15). 5 Best Practices for Managing Requirements in Agile Projects. Retrieved December 28, 2017, from <http://www.mymanagementguide.com/best-practices-managing-requirements-agile-projects/>

Requirements management. Retrieved December 21, 2017, from https://en.wikipedia.org/wiki/Requirements_management

Requirements Management Tools. Retrieved December 29, 2017, from http://www.requirementsmanagementschool.com/w1/Requirements_Management_Tools

Requirements traceability. Retrieved December 23, 2017, from https://en.wikipedia.org/wiki/Requirements_traceability

Rodriguez, L. L. (2004). Master Thesis: “Managing Software Requirements: Organizational and Political Challenges”. Retrieved December 27, 2017, from <https://dspace.mit.edu/handle/1721.1/17784>

What is Requirements Management. Retrieved December 21, 2017, from http://www.requirementsmanagementschool.com/w1/What_is_Requirements_Management

What is Traceability Matrix. Retrieved December 23, 2017, from http://www.requirementsmanagementschool.com/w1/What_is_a_Traceability_Matrix

Image Credits:

Image 1: Requirement Attributes. By Ethacke1 (Own work) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons

Image 2: Requirement Traceability Matrix. By Kelly Lawless (Own work) [CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>)], via Wikimedia Commons

Image 3: Change Management Process diagram. By Ethacke1 (Own work) CC BY-SA 3.0 (<https://creativecommons.org/licenses/by-sa/3.0/>), via Wikimedia Commons