ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

The Art of Creating Software Requirements

Adnan Shaout[1] and Anthony Walker[2]

The Electrical and Computer Engineering Department, The University of Michigan – Dearborn[1,2]

Abstract: The definition of good software requirements has been the topic of numerous debates between software engineers and test engineers for a long time. The purpose of this research is to find out what could be contributing to the various methods and rules/guidelines for writing requirements. In doing this research, we found out that it is not so much that there are different methods to writing requirements it is more so that there is insufficient training on writing requirements. We will go into detail on what the IEEE standard states are the characteristics of well-formed requirements that support our view of how requirements should be formed. We will also touch a bit on how this debate can be clarified.

Keywords: Software Requirements, Test Engineers, Software Engineers, IEEE Standard, Characteristics of Software Requirements.

I. INTRODUCTION

The characteristics of a well written requirement are that the requirement is unambiguous, consistent, complete, singular, feasible, traceable, and verifiable [1]. The majority of the debate on writing requirements seemed to be on what should be contained in a requirement. Zowghi and Paryani [3] discuss in their paper that teaching requirement engineering should be through role playing. This is the underlying concept that is causing the debates.

The art of requirements engineering is not taught at the academic level. Due to the importance and the cost savings that is constantly linked to requirements, this should be a topic that is covered in academia as well as reinforced in the workplace. Peer reviews are an excellent way to get feedback on requirements.

This also allows for the opportunity for the requirements engineer to have others insight on various characteristics of writing software requirements. The information that the IEEE standard for writing well-formed requirements [1] further supports the fact that the art of writing requirements needs to be discussed more in detail in both academia and the workplace.

Creating software requirements is more than just writing requirements. As noted in [6], software requirements engineering consists of software requirements elicitation, software requirements analysis, software requirements specification, software requirements verification and software requirements management. In the following sections, we will expand on how each of these characteristics contributes to writing good software requirements and provide a detailed explanation of each characteristic.

This paper will also provide an overview of the various methods of creating software requirements. It also presents a comparison of the various methods and rankings.

II. CHARACTERISTICS OF SOFTWARE REQUIREMENTS

An unambiguous requirement is one that can be interpreted in only one way, easy to understand and stated simply [1]. In certain situations where in an effort to sound really intelligent, requirements may not be stated in a simple way. The complexity of software today is high enough without adding more to it by not simplifying requirements. Taking the extra time to ensure that the requirement is being stated in a simple way can reduce the time spent trying to make sure that it is being met. By focusing on keeping the wording of the requirement simple, this may lead to the requirement being easy to understand. Due to the global nature of software and requirements engineering, the use of natural language in writing requirements is a point of concern. There’s a high possibility that the engineers who are tasked with implementing and verifying the requirements do not have the same native language as the engineer who wrote the requirements.

In an effort to reduce the language ambiguities associated with writing software requirements, there have been various studies to come up with solutions to resolve language barrier issues. One of those solutions is the Model based Object oriented approach to Requirements Engineering (MORE) [2]. While modeling requirements using UML (Unified Modeling Language) and other object oriented techniques can be used to ensure requirements are unambiguous, it is important to understand that not all software requirements can be captured in a single diagram.

It is clear that these techniques can aid in the understanding of the system and design, however a limit has to be instilled as to how much detail is put into these diagrams. Too much information can lead to implementing the design of the system instead of simply meeting the requirements of the system. Requirements should state ‘what’ is needed, not ‘how’ [1]. A common technique used to identify a requirement is that the sentence uses the word ‘shall’. The use of words such as ‘could’ and ‘should’ are not to be used when writing requirements as these types of words are non-binding [1].

The following are types of ambiguous terms that engineers shall avoid using when writing requirements [1]:

- Superlatives (such as 'best', 'most')
- Subjective language (such as 'user friendly', 'easy to use')
- Vague pronouns (such as 'it', 'this', 'that')
- Ambiguous adverbs and adjectives (such as 'almost always', significant', 'minimal')
- Open-ended, non-verifiable terms (such as 'provide support', 'but not limited to', 'as a minimum')
- Negative statements (such as statements of software capability not to be provided)

If any of these types of terms are used in requirements, they will lead to having a software system that does not meet the intended purpose as expected by the customer. Requirements cannot be open-ended, use negative statements, and be vague or subjective. Using these types of terms will make the requirements engineering phase last longer than it otherwise would without using such terms. The use of superlatives makes requirements seem to be more suggestive as opposed to something that is required.

The next characteristic of a well written requirement is that it is singular. This means that the requirement should address only one item. If there is not only one item in the requirement without the use of conjunctions [1], the requirement is overloaded. An overloaded requirement is not simple to understand. Sometimes this may be done unintentionally due to insufficient training on writing requirements. A requirement may require multiple inputs or conditions to be met, however it must remain unambiguous, i.e. be stated simply and easy to understand.

An example of an overloaded requirement is as follows:

Accelerometer data shall be collected at a rate of 1ms and after 10ms the data will be averaged and provided to other software modules via an API.

This requirement can be broken down into 3 separate requirements as follows:

The software shall collect accelerometer data at a rate of 1ms.

The software shall average 10 samples of data and store it for later processing

The averaged accelerometer data shall be provided to other software modules via an API.

A requirement must also be consistent, meaning the requirement does not conflict with other requirements [1]. Creating multiple requirements that have the same meaning adds confusion to an already potentially complex system or component. This duplication can lead to wasted time; multiple test cases being created that verify the same functionality in a different way and add increases the difficulties encountered when dealing with the language barrier on making requirements unambiguous.

A well written requirement is also complete and feasible. Complete means that the requirement needs no further amplification because it is measurable and sufficiently describes the capability and characteristics to meet the customer's need [1]. This does not mean that the requirement or set of requirements to meet this need should talk about how the software needs to be designed to meet the requirement. If a requirement locks the designer into a certain implementation this is typically not a good requirement. There could be numerous ways to implement software in order to meet a requirement or set of requirements. Unless the architecture or algorithm to use is defined by the company as a standard to follow, requirements need to be written without imposing unnecessary bounds on the solution space [1].

Having an understanding of what the system is capable of while working on the software requirements is helpful. This will help in determining the feasibility of a requirement. The requirement must fit within the system and be technically achievable [1]. For example, a requirement that states:

The software shall collect 16-bit acceleration data from the x-axis sensor at a rate of 250us. is not technically feasible if the system only allows for acceleration data to be collected at a rate of 2ms.

The last characteristics of well-formed requirements are that they are traceable and verifiable. The traceability of the requirements goes from top to bottom and vice versa. From the customer document to the test case in verification, each software requirement must be traceable. If it cannot be traced to a customer requirement and to a point where it is verified, there is a gap in the software, documentation or test case(s).

If sufficient time is spent with external and internal customers during the early phases of requirements engineering, a lot of time and ultimately money can be saved. The savings comes from spending the time upfront to clearly define the expectations (i.e. requirements) of the system that is being designed to meet the requirements of the customer. The pie chart in figure 1 [7] shows where the majority of bugs are introduced in a system (RE means Requirements Engineering):

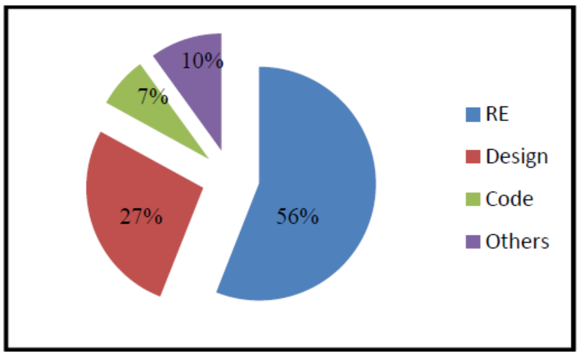


Figure 1. Percentage of bugs in a topical process phases.

III. Software Requirements Through Modeling

One of the modeling techniques in existence today is referred to as the RDDA (Requirements-Driven Design Automation) framework which uses the SysML (Systems Modeling Language) framework. As stated in [4], SysML supports several types for describing requirements,

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

including a Requirements Diagram, where textual requirements statements and their inter-relationships are represented visually. As part of the RDDA framework project, the team created a method for users to add sematic descriptions in SysML for specifying system resource and QoS (Quality of Service) constraints.

Figure 2 shows an example of a requirements diagram using SysML [4] that describes the requirements and constraints for an LBS application.

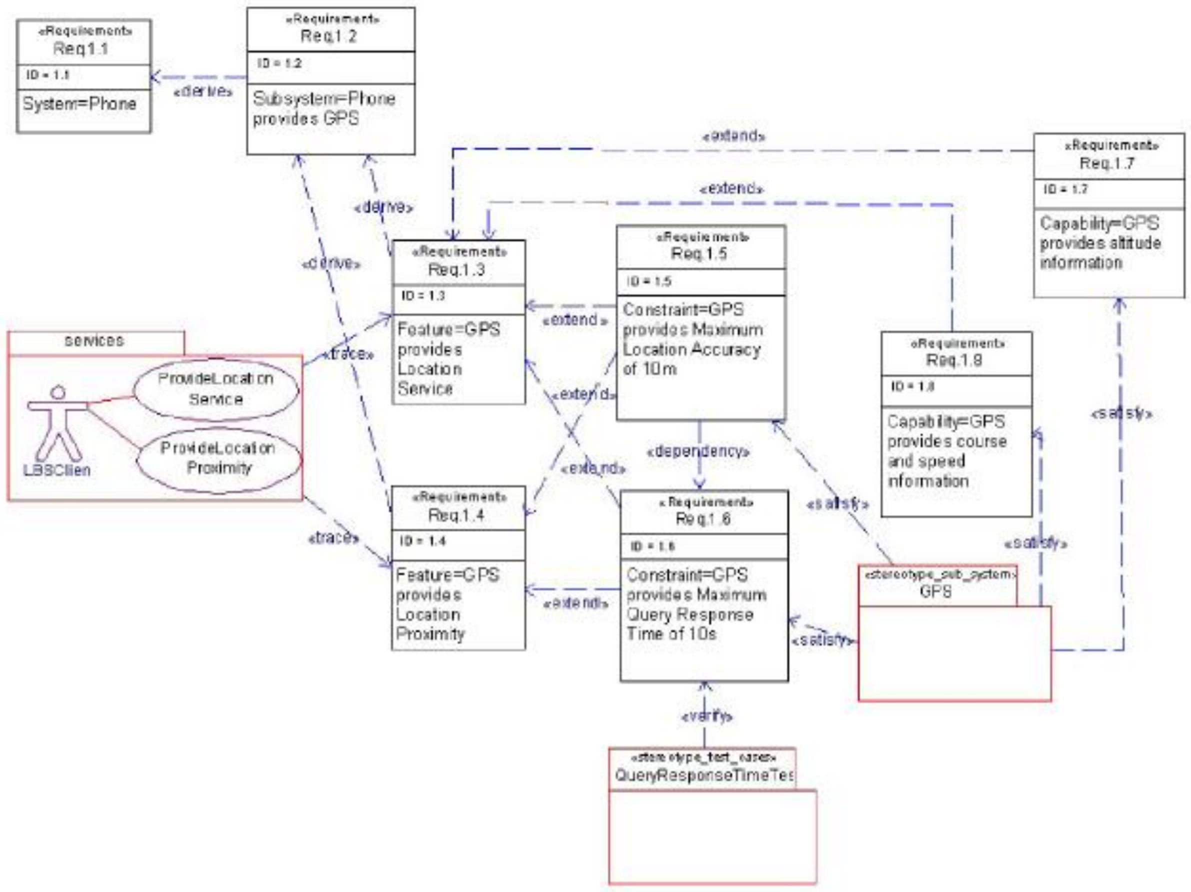


Figure 2. A requirement diagram example.

The use of modeling as described in the RDDA framework is a good method, but it is still imperative that one will be able to accurately create textual software requirements. One of the input methods described with the RDDA framework is that textual requirements can be implemented visually.

Using this modeling technique presents the opportunity to remove the language ambiguities associated with textual requirements. It is generally easier for the customer to understand how their requirements are being met pictorially versus trying to understand written requirements that are not written in their native language. The most important aspect of software requirements that must not be compromised with modeling is that requirements must be written without
imposing unnecessary bounds on the solution space [1].

IV. Removing Ambiguity In Natural Language

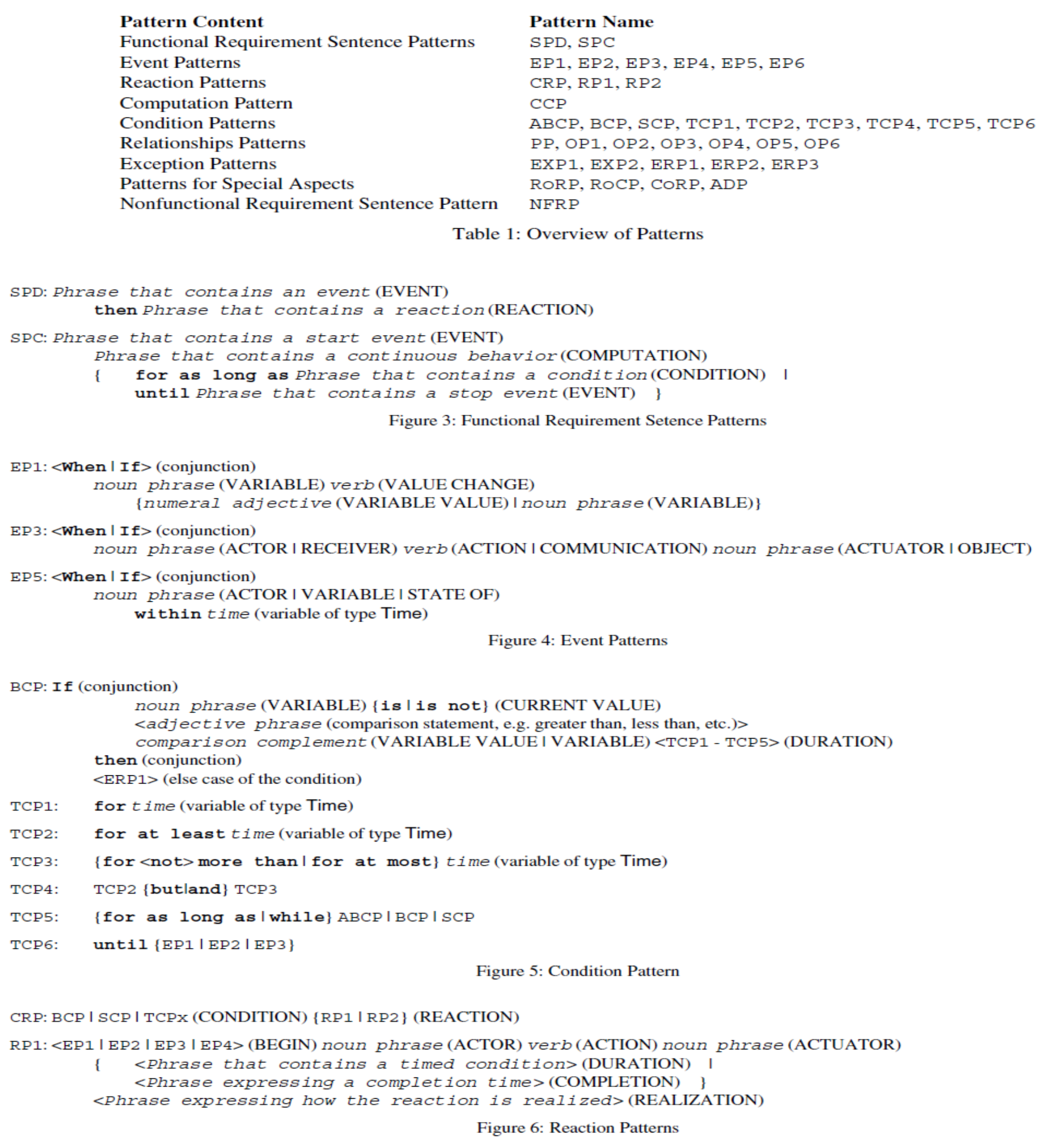
Denger, Berry and Kamsties [5] discuss how to create higher quality requirements through patterns. By defining and using patterns, the ambiguities associated with writing requirements can be resolved as long as the patterns are well defined. Figure 3 [5] lists some of the various patterns.

Sentence patterns are classified as two types: discrete behavior and continuous behavior. As stated in [5], a discrete behavior pattern specifies a systems reaction in response to an event.

A continuous behavior pattern specifies a reaction of the system that is started by a certain event and is performed until another event occurs or a certain condition comes true [5].

An event pattern is a change in the value of a variable or a change in the system state. A condition pattern is a test of the current value of a variable or a test of the current value of a variable or a test of the current system state [5].

Using the patterns they created, they were able to remove several ambiguities found in the following requirement:



Pattern Content	Pattern Name
Functional Requirement Sentence Patterns	SPD, SPC
Event Patterns	EP1, EP2, EP3, EP4, EP5, EP6
Reaction Patterns	CRP, RP1, RP2
Computation Pattern	CCP
Condition Patterns	ABCP, BCP, SCP, TCP1, TCP2, TCP3, TCP4, TCP5, TCP6
Relationships Patterns	PP, OP1, OP2, OP3, OP4, OP5, OP6
Exception Patterns	EXP1, EXP2, ERP1, ERP2, ERP3
Patterns for Special Aspects	RoRP, RoCP, CoRP, ADP
Nonfunctional Requirement Sentence Pattern	NFRP

Table 1: Overview of Patterns

```
SPD: Phrase that contains an event (EVENT)
        then Phrase that contains a reaction (REACTION)
SPC: Phrase that contains a start event (EVENT)
        Phrase that contains a continuous behavior (COMPUTATION)
        {   for as long as Phrase that contains a condition (CONDITION)  |
            until Phrase that contains a stop event (EVENT)   }
```

Figure 3: Functional Requirement Setence Patterns

```
EP1: <When | If> (conjunction)
        noun phrase (VARIABLE) verb (VALUE CHANGE)
            {numeral adjective (VARIABLE VALUE) | noun phrase (VARIABLE)}
EP3: <When | If> (conjunction)
        noun phrase (ACTOR | RECEIVER) verb (ACTION | COMMUNICATION) noun phrase (ACTUATOR | OBJECT)
EP5: <When | If> (conjunction)
        noun phrase (ACTOR | VARIABLE | STATE OF)
            within time (variable of type Time)
```

Figure 4: Event Patterns

```
BCP: If (conjunction)
            noun phrase (VARIABLE) {is | is not} (CURRENT VALUE)
            <adjective phrase (comparison statement, e.g. greater than, less than, etc.)>
            comparison complement (VARIABLE VALUE | VARIABLE) <TCP1 - TCP5> (DURATION)
        then (conjunction)
        <ERP1> (else case of the condition)
TCP1:   for time (variable of type Time)
TCP2:   for at least time (variable of type Time)
TCP3:   {for <not> more than | for at most} time (variable of type Time)
TCP4:   TCP2 {but|and} TCP3
TCP5:   {for as long as | while} ABCP | BCP | SCP
TCP6:   until {EP1 | EP2 | EP3}
```

Figure 5: Condition Pattern

```
CRP: BCP | SCP | TCPx (CONDITION) {RP1 | RP2} (REACTION)
RP1: <EP1 | EP2 | EP3 | EP4> (BEGIN) noun phrase (ACTOR) verb (ACTION) noun phrase (ACTUATOR)
        {   <Phrase that contains a timed condition> (DURATION)  |
            <Phrase expressing a completion time> (COMPLETION)  }
        <Phrase expressing how the reaction is realized> (REALIZATION)
```

Figure 6: Reaction Patterns

Figure 3. A list of patterns [5].

R3.1.4. Stuttered dial tone: EMS shall support notification by stuttered dial tone played for as long as no key is pressed; that is, EMS shall interact as necessary with other systems so that when the subscriber has one or more new messages, the subscribed phone will give a stuttered dial tone rather than a standard dial tone.

Here are the flaws of this requirement as stated in [5]: The first clause, "EMS shall support notification by stuttered dial tone played for as long as no key is pressed", is refined in the second clause, "EMS shall interact as necessary with other systems so that when the subscriber has one or more new messages, the subscribed phone will give a stuttered dial tone rather than a standard dial tone.", following the phrase "that is". The first clause is recognized as matching RoRP (Realization of Reaction Pattern). The clause contains a condition "when the subscriber has one or more messages" and a reaction "the subscriber's phone plays a stuttered dial tone". The event that triggers the reaction is not specified because the event is implicitly described via the vague phrase "EMS shall interact as necessary with other systems" and the name of the requirement "Stuttered dial tone". This vague phrase contains two sources of imprecision, namely the phrases "as necessary" violates one of our authoring rules, namely the one that is against using phrases that are open to subjective interpretations. The phrase "other systems" is also ambiguous, since it is not clear which other systems are referenced. Finally the phrase "rather than a standard dial tone" is removed as redundant, since a stuttered dial tone is not the standard dial tone.

In addition to this analysis, we feel there are too many requirements listed in the single requirement of the stuttered dial tone. Here is the rewritten requirement using the language patterns:

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

IJARCCE

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

3.1.4 FR.Stuttered_Dial_Tone: If the subscriber picks up the phone and a message is waiting, then the subscriber's phone plays a stuttered dial tone for as long as no key is pressed. For this purpose, the central office system sends the signal 'has new messages' to the subscriber's phone. Then, if the subscriber's phone state is 'has new', the EMS sends the signal new messages' to the central office. Then, the central office sends a stuttered dial tone to the subscriber's phone for as long as no key is pressed. If the subscriber's phone state is 'has no new', the EMS sends the signal 'no new messages' to the central office. Then, the central office sends a standard dial tone to the subscriber's phone, for as long as no key is pressed.

Both the original requirement and the rewritten requirement still have room for improvement in my opinion. Neither requirement covers one of the characteristics of a well written requirement, which is that it is singular. We would take the rewritten requirement further and break it up into 6 individual requirements as follows:

If the subscriber picks up the phone and a message is waiting, the subscriber's phone shall play a stuttered dial tone until a key is pressed.

When there is a message waiting and the subscriber's phone is playing a stuttered dial tone, the central office system shall send the signal 'has new messages' to the subscriber's phone.

If the subscriber's phone state is 'has new', the EMS shall send the signal 'new messages' to the central office.

While the state of the subscriber's phone is 'has new', the central office shall send a stuttered dial tone to the subscriber's phone until a key is pressed.

If the state of the subscriber's phone is 'has no new', the EMS shall send the signal 'no new messages' to the subscriber's phone.

While the state of the subscriber's phone is 'has no new', the central office shall send a standard dial tone to the subscriber's phone until a key is pressed.

Each of these requirements is now singular, complete, consistent, unambiguous, feasible, traceable and verifiable.

While creating and using natural language patterns seems like a good idea, we still believe there are flaws to it (as shown above by my rewriting of the requirement to make it singular). A pattern created by one group has a high possibility of being created completely different by another group to address the same topic/concept.

The Unified Modeling Language (UML) [9] is a standard maintained by the Object Management Group (OMG). Earlier versions of the UML were a convergence of three prominent OO (Object Oriented) modeling methods: The Object Modeling Technique (OMT) [11], the Booch Method [12] and the Jacobson's Object-Oriented Software Engineering (OOSE) approach [13] [10]. According to [9] and [10], use cases are used to capture the requirements of a system. The following text will describe a use case diagram and its notation.

Figure 4 shows an example of a use case diagram of an ATM system.

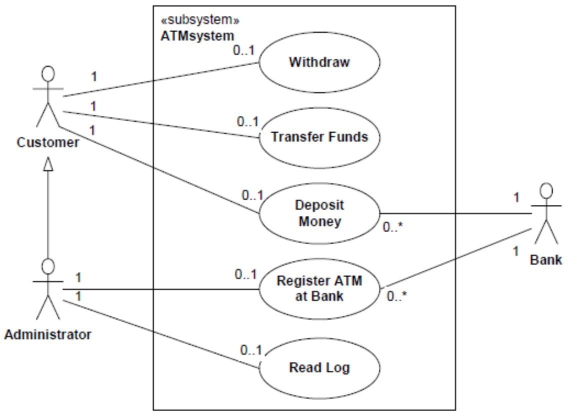


Figure 4. A use case Diagram of an ATM system.

A use case is depicted by an ellipse with the name of the use case in it (the name of the use case can appear below the ellipse as well instead of inside of it). It is standard practice for a use case to have a name that is associated with its functionality/purpose. Each use case specifies some behavior/functionality that the subject can perform in collaboration with one or more actors [9]. This behavior /functionality must always be completed for the use case to complete [9]. It is deemed complete if the subject will be in a state in which no further inputs or actions are expected and the use case can be initiated again or in an error state [9]. The subject of a use case (depicted by the rectangle) could be a physical system or any other element that may have behavior, such as a component, subsystem, or class [9].

Each stick figure is referred to as an actor. The name of the actor is typically placed above or below it. "An actor specifies a role played by a user or any other system that interacts with the subject, but which is external to the subject (i.e., in the sense that an instance of an actor is not a part of the instance of its corresponding subject). Actors may represent roles played human users, external hardware, or other subjects. Note that an actor does not necessarily represent a specific physical entity but merely a particular facet (i.e. "role") of some entity that is relevant to the specification of its associated use cases. Thus, a single physical instance may play the role of several different actors and, conversely, a given actor may be played by multiple different instances" [9].

An alternative of the UML conventional use case is Essential Use Case (EUC). According to [8], the EUC approach is defined by its creators Constantine and Lockwood as a "structured narrative, expressed in a language of the application domain and of user,

ISSN (Online) 2278-1021
ISSN (Print) 2319-5940

International Journal of Advanced Research in Computer and Communication Engineering
Vol. 4, Issue 9, September 2015

comprising a simplified, generalized, abstract, technology free and independent description of one task or interaction that is complete, meaningful, and well-defined from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction" [14]. The EUC description is generally shorter than a conventional UML use case because it only comprises the essential steps (core requirements) of intrinsic user interest [8]. Figure 5 shows an example taken from [8] (which is adapted from [15]) that shows how natural language requirements are translated into EUCs.

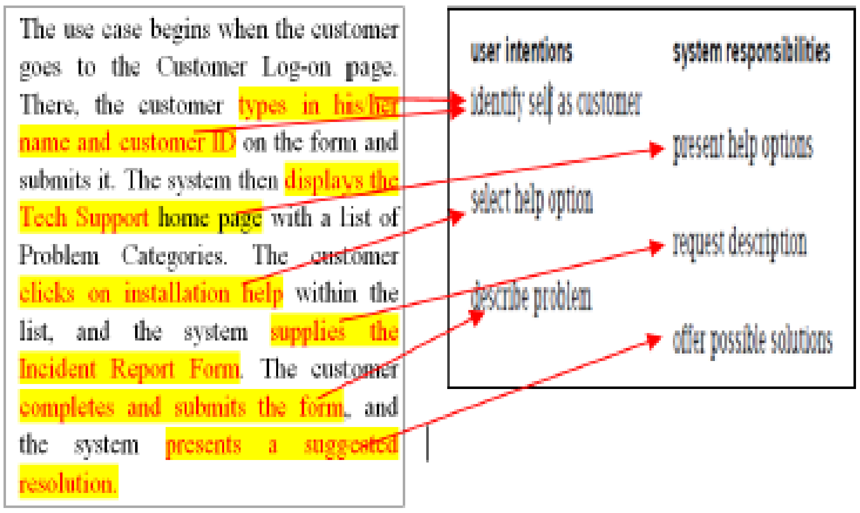

The use case begins when the customer goes to the Customer Log-on page. There, the customer types in his/her name and customer ID on the form and submits it. The system then displays the Tech Support home page with a list of Problem Categories. The customer clicks on installation help within the list, and the system supplies the Incident Report Form. The customer completes and submits the form, and the system presents a suggested resolution.

Figure 5. An example of a natural language requirements being translated into EUCs.

As stated in [8] EUCs simplify captured requirements compared to conventional UML use cases, requirements engineers still face the problem of "finding the correct level of abstraction, which also takes time and effort" [16].

V. Comparison and Ranking of Techniques

Table 1 shows the comparison of various techniques with respect to several characteristics such as singular, unambiguous and complete.

Table1 : Comparison Of Techniques

Technique	Characteristics						
	Singular	Unambiguous	Consistent	Complete	Feasible	Traceable	Verifiable
Natural language (following IEEE guidelines)	1	1	1	1	2	2	2
UML	2	2	2	2	1	1	1
MORE	5	4	5	4	5	4	4
Patterns	3	5	3	3	3	5	5
EUC	4	3	4	5	4	3	3

The characteristic of each technique is ranked 1 through 5 (1 is the highest rank, 5 is the lowest)

Based on the comparison shown in table 1, we still believe the using natural language is the best option for capturing software requirements with the use of UML closely following it. Ultimately we came to this conclusion based on our own experiences capturing requirements as well as what is stated in [9]. According to [9], "the detailed behavior defined by a use case is notated according to the chosen description technique, in a separate diagram or textual document". This further supports the theory that there are limitations on how much detail can be put into a diagram. As more and more detail of the functionality of a requirement (or requirements block) is captured in a diagram, you begin to get into the detailed design of how the requirement is to be implemented rather than what is to be implemented.

UML deserves a very close second place ranking given the ease of communicating across multiple teams, backgrounds and it provides the ability to remove ambiguities of using natural language. As engineers, the goal of communicating what is needed must be kept simple. For example, the instructions on how to assembly a bookshelf contains both pictures and words. UML brings that same philosophy to software requirements engineering. Keeping things as simple as possible greatly reduces the chance for misinterpretations and missed requirements.

EUC deserved a rank of third mostly due to the fact that it also uses use cases. An issue with this technique is that it may be too simplified. As stated in [8], "some of the main reasons EUCs are not commonly used are: a lack of tool support; engineer's lack of experience in extracting essential interactions from requirements; and a lack of integration with other modeling approaches" [16] [17]. There is also currently no tool that exists to support engineers working with EUC models [8].

Patterns as described in [5] received a rank of fourth due to the lack of using use cases or diagrams. Natural language lacks the ability to communicate to the customer or end user who is not a software engineer. This concept does offer the benefit of being able to define natural language requirements as discussed in [1] and in this report. By combining the natural language technique along with patterns, the ambiguities can be virtually eliminated.

VI. CONCLUSION

The purpose of this research was to focus solely on the software requirements specification component of software requirements engineering. However we found it difficult to talk about the various aspects of requirements specification without thinking of what it takes to get to this point. Each of these components is an important piece to writing good software requirements whether it is using one's native language or through modeling.

We would rank the methods discussed in this paper in the following order: following the IEEE requirements fundamentals (guidelines), using language patterns and finally modeling. However, we think a solution that should be investigated further is a combination of all three facets discussed here. We feel there is an opportunity to combine all the pros of each technique into one technique that would be very useful in requirements engineering.

REFERENCES

[1] Systems and software engineering – Life cycle processes – Requirements engineering, IEEE 29148, 2011-12.
[2] C.W. Lu, W.C. Chu, C.H. Chang, C.H. Wang, "A Model-based Object-oriented Approach to Requirement Engineering (MORE)," presented at the 31st Annual International Computer Software and Applications Conference, 2007.
[3] D. Zowghi, S. Paryani, "Teaching Requirements Engineering through Role Playing: Lessons Learnt," proceedings of the 11th IEEE International Requirements Engineering Conference, 2003.
[4] I. Cardei, M. Fonoage, R. Shankar, "Model Based Requirements Specification and Validation for Component Architectures," SysCon 2008 – IEEE International Systems Conference, 2008
[5] C. Denger, D. Berry, E. Kamsties, "Higher Quality Requirements Specifications through Natural Language Patterns," Proceedings of the IEEE International Conference on Software – Science, Technology & Engineering (SwSTE'03), 2003
[6] R. Thayer, M. Dorfman, "Introduction to Tutorial: Software Requirements Engineering", 2000
[7] K. Khan, P.V.V. Kumar, A. Ahmad, T. Riaz, W. Anwer, M. Suleman, O. Ajmal, T. Ali, A.V.K. Chaitanya, "Requirement Development Life Cycle: The Industry Practices", Ninth International Conference on Software Engineering Research, Management and Applications, 2011
[8] M. Kamalrudin, J. Grundy, J. Hosking, "Tool Support for Essential Use Cases to Better Capture Software Requirements", Proceedings of the IEEE/ACM International Conference on Automated software engineering, pages 255-264, 2010
[9] The Object Management Group (OMG). Unified Modeling Language. Version 2.0, OMG, http://www.omg.org/spec/UML, July 2005
[10] Robert France, Cris Kobryn, "UML for Software Engineers", IEEE, 2001
[11] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991
[12] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, Menlo Park, CA, Second edition, 1994
[13] I. Jacobson. *Object oriented software engineering*. Addison-Wesley, 1992
[14] Constantine, L.L. and Lockwood, A.D.L. Software for use: a practical guide to the models and methods of usage-centered design. ACM Press/Addison-Wesley Publishing Co., 1999.
[15] Constantine, L.L. and Lockwood, A.D.L. Structure and style in use cases for user interface design. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA 2001.
[16] Biddle, R., Noble, J. and Tempero, E. April 2000. Pattern for Essential Use Cases. Technical Report. Victoria University of Wellington at Wellington, New Zealand
[17] Biddle, R., Noble, J. and Tempero, E. "Essential use cases and responsibility in object-oriented development", Australian Computer Science Communications, 2002.

BIOGRAPHIES

Dr. Adnan Shaout is a full professor and a Fulbright Scholar in the Electrical and Computer Engineering Department at the University of Michigan – Dearborn. At present, he teaches courses in logic design, computer architecture, cloud computing, fuzzy logic and engineering applications and computer engineering (hardware and software). His current research is in applications of software engineering methods, computer architecture, embedded systems, fuzzy systems, real time systems and artificial intelligence. Dr. Shaout has more than 33 years of experience in teaching and conducting research in the electrical and computer engineering fields at Syracuse University and the University of Michigan - Dearborn. Dr. Shaout has published over 170 papers in topics related to electrical and computer engineering fields. Dr. Shaout has obtained his B.S.c, M.S. and Ph.D. in Computer Engineering from Syracuse University, Syracuse, NY, in 1982, 1983, 1987, respectively.

Anthony Walker – A graduate student (Master of Science in Software Engineering) from the department of Electrical and Computer Engineering at the University of Michigan - Dearborn.