REQUIREMENTS SPECIFICATION AND ANALYSIS

The first step in engineering a software system is to understand what the system should do—this is referred to as the specifications of the requirements for the software system or, alternatively, as requirements specifications or software requirements specifications (SRS). Once the requirements have been specified, very often the next step is to analyze them to update, modify, and prioritize the requirements so that the software development team develops a better understanding of the system to be developed, the constraints on the system, and initial estimates on cost and schedule for developing the system—this step is referred to as requirements analysis. Thus, requirements specification and analysis is a two-step process that is also iterative—analysis may lead to revised specifications that could entail additional analysis. Requirements specification and analysis is a phase in software engineering; from a software engineering perspective, the requirements specification and analysis phase is followed by the software design phase.

The requirements specification and analysis phase of a software project is the most important phase of software development and should not be omitted under any condition. More than half of software projects have failed because of errors in the requirements specification and analysis phase, and the cost for correcting errors committed during this phase increases exponentially as software system development progresses through the remaining phases of design, implementation, testing, and maintenance. It has been estimated (1), as shown in Fig. 1, that if an error detected and fixed during the requirements phase will incur a cost ratio of 1, then the same error if detected and fixed during the design phase will incur a cost ratio of between 3 and 6, during the implementation phase will incur a cost ratio of 10, during the development testing will incur a cost ratio of between 15 and 40, during the acceptance testing will incur a cost ratio of between 30 and 70, and if the error is detected and fixed when the system is under operation (or during the maintenance phase), the organization will incur a cost ratio of between 40 and 1000. Therefore, proper requirements specification and analysis is important during software development.

A basic rule of SRS development is to capture "what" the software system should do and never rush to the "how" the requirement is to be achieved—violation of this simple rule will lead to an early commitment to design without suitably exploring alternative designs. Even though this rule seems simple enough, the history of requirements specifications is replete with examples of breaches of this rule that may be a reason for the large number of software project failures—the temptation to rush to design or even implementation without first understanding the requirements is sometimes too much for software engineers. Thus, very often solutions are produced for the wrong problem or wrong solutions are produced for the actual problem!

Requirements for a software system can be classified into two types: functional and nonfunctional. Functional requirements specify what the system should do, whereas nonfunctional requirements specify the global characteristics of the software system. Thus, the requirement "*The input to the software system should be by means of a keyboard entry, mouse click, or stylus movement*" is a functional requirement, whereas "*The software system should have fast responses to user inputs*" is a nonfunctional requirement (NFR) since the "fast responses" requirement is usually not achievable by means of just a few components in the software system but is a globally observed characteristic of the software system. Functional requirements include requirements related to inputs, outputs, processes (or functions), and stored data for the new system. Functional requirements also capture the interfaces between the system and its users as well as the interfaces between systems. Nonfunctional requirements (also referred to as quality requirements or system attributes) capture requirements related to characteristics such as performance, usability, security, reliability, availability, maintainability, and portability; effort, budget, and schedule estimations; documentation and training needs; quality management; and constraints under which the final system may be expected to operate, for example, operating system, processing speed, network bandwidth, memory size, or implementation language.

An SRS typically includes

1. Verbal descriptions of functional and nonfunctional requirements.
2. Analysis artifacts such as requirements prioritization, dependencies, versioning, cost and staff size estimations, formulation of acceptance tests, and analysis of NFRs.
3. All associated models developed during analysis.

Not only does the SRS serve as the starting point for subsequent phases of software development, SRS also often serves as a legal contract between the software developing organization and the customer who will actually be purchasing and/or using the software.

The process of developing software requirements and analyzing them is illustrated in Fig. 2, and the focus during this phase is on the customers and users of the proposed software system. The process starts invariably with an understanding of the current system and its problems—the current system may be manual or computerized. This task is also called problem analysis. The techniques to help with this task include the PIECES (performance, information, economics, control, efficiency, and service) framework, Ishikawa diagrams (also called fish-bone diagrams), study of forms and documentation used with the current system, interviews with users of current system, and observations of use of the current system. The outputs of this task include the problem statement and system improvement objectives

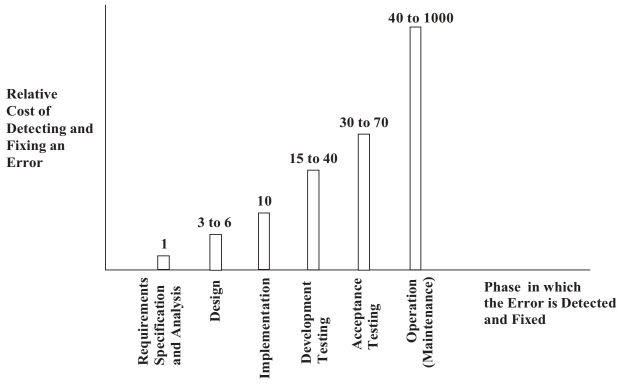


Figure 1. Relative costs of detecting and fixing an error in different software development phases (not drawn to scale).

that are used during the second task to guide elicitation of requirements from the users and customers. Usually, a subtle distinction is made between a user (also called an end user) and a customer. Customer is the organization or person(s) who will be paying for the software or sponsoring the software project, whereas users are the actual users of the software system who will be interacting with the software for the benefit of their organization; of course, customers may be users as well. The most important technique to elicit requirements is interviewing, but recently JRP (joint requirements planning) and discovery prototyping have emerged as supplementary techniques. The outputs of this task include notes, minutes, voice recordings, and updated prototypes. The third task uses the outputs of requirements elicitation to develop the initial SRS that documents the requirements using any combination of tabular listing, use-cases, and user stories. The first, second, and third tasks collectively form the requirements specifications step referred to earlier. The fourth task analyzes the initial SRS using reviews and customer/user clarifications to identify priorities, dependencies, versions, cost and manpower estimates, acceptance tests, and NFR analyses. The output of this task is the updated SRS. The fifth and final task models the requirements using any combination of UML (Unified Modeling Language) diagrams, ERD (Entity Relationship Diagrams) (2), DFD (Data Flow Diagrams), and formal methods. The output of the fifth task, also called logical design, is the updated SRS with models. The fourth and fifth tasks together form the requirements analysis step mentioned earlier. The SRS is now ready for the next phase of software development, namely, design, which is also called the physical design since implementation or physical aspects of the software system will have to be considered. As indicated in Fig. 2, the process is not linear but iterative; it is possible to go from one task to any of the earlier tasks whenever an error or omission is detected. Each technique is discussed in more detail in the subsequent sections. Typically, requirements specification and analysis phase lasts one third of the total project time and the durations of the individual tasks within this period are distributed almost evenly.

TASK 1: UNDERSTAND PROBLEMS WITH THE CURRENT SYSTEM

Very frequently the main driving force behind the development of new software systems is the set of problems faced by the end users in using the current system and processes, both of which may include a combination of manual and computerized aspects. Therefore, a good understanding of the problems with the current system will give the developers a set of measurable objectives that the new system

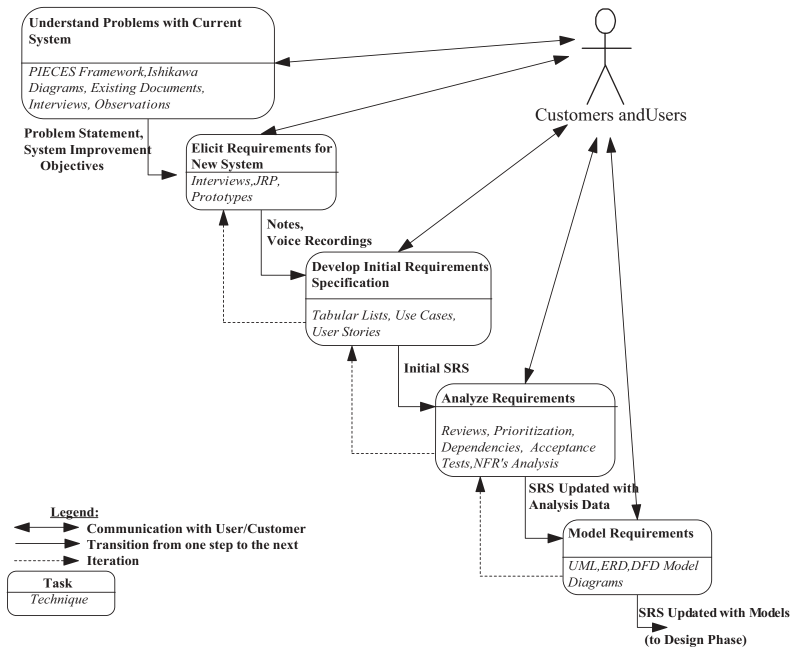

Understand Problems with Current System
PIECES Framework, Ishikawa Diagrams, Existing Documents, Interviews, Observations
Customers andUsers
Problem Statement, System Improvement Objectives
Elicit Requirements for New System
Interviews, JRP, Prototypes
Notes, Voice Recordings
Develop Initial Requirements Specification
Tabular Lists, Use Cases, User Stories
Initial SRS
Analyze Requirements
Reviews, Prioritization, Dependencies, Acceptance Tests, NFR's Analysis
SRS Updated with Analysis Data
Model Requirements
UML, ERD, DFD Model Diagrams
SRS Updated with Models (to Design Phase)
Legend:
Communication with User/Customer
Transition from one step to the next
Iteration
Task
Technique

Figure 2. Process of software requirements specification and analysis.

should satisfy. Examples of measurable objectives are "*Number of orders processed should increase by 10%*" or "*Response time decreases by 20%.*" These objectives can be achieved by fixing certain problems with the current system, and therefore, those problems provide not only opportunities for improving the current system but also serve as starting points for developing the requirements for the new and improved system. However, how does one identify the problems? The PIECES framework (3) is an excellent tool to document problems in an existing system—very frequently problems occur in the categories of performance, information, economics, control (and/or security), efficiency, and service. Other techniques include interviewing users of the current system, examining problem reports on the current system, and observing actual usage of the current system. When a large number of end users are involved, questionnaires may also be used. Once the problems are identified, they are documented in a problem statement matrix that lists the problems identified, the urgency of each problem, the monetary benefits in fixing each problem, and how each problem might be fixed such as, for example, by developing a new system or simply patching the existing one. Once the problems have been identified, it becomes necessary to determine their causes, and for this purpose, Ishikawa diagrams (also called fish-bone diagrams) (1) may be used. For each problem, an Ishikawa diagram is developed wherein the problem is mentioned along the main bone and the side bones represent categories under which causes for the problem may be grouped; the actual causes are listed off the side bones. Ishikawa diagrams help distinguish between symptoms and problems, and once all the problems are identified, a systematic effort may be undertaken to determine how the problems may be eliminated—called opportunities for improvement—and in this process, the preliminary requirements for the new system can be identified. As a result of this process, we develop the system improvement objectives matrix that lists, for each problem in the problem statement matrix, the probable causes and the measurable objectives for verifying the absence of the problem in the new system.

TASK 2: ELICIT REQUIREMENTS FOR NEW SYSTEM

As one might expect, software requirements are usually provided by the end users of the proposed software system. This process of gathering requirements from end users is called requirements elicitation. Even though elicitation seems like a simple activity, the trouble is that very frequently users are not sure of what they want (4)! Eliciting requirements should preferably be handled by software engineers well versed in communication and an understanding of human nature, because they will be required, quite often, to lead the process of elicitation by careful questioning of the end users. Interviews are the most common technique for eliciting requirements from end users. During interviewing, the questions necessary to elicit functional and nonfunctional requirements should be put to the end users. For example, questions such as "*What should the system do?*", "*What inputs will the system receive?*", "*What outputs should the system generate?*", and "*What are the data formats?*" are examples of questions that will help elicit functional requirements, whereas questions such as "*What is the maximum tolerable throughput of the system?*", "*What types of access control are required?*", "*What is the expected mean time between failures?*", "*What operating systems should the system work in?*", and "*What programming language will be used to implement the system?*" are examples of questions that will help elicit nonfunctional requirements. The user responses to these questions may be captured as notes or minutes on paper, or voice may be recorded digitally or on tapes. Important points to keep in mind are the types of questions to ask such as open-ended (for example, "*What is the maximum number of expected users for the system?*") and closed-ended (for example, "*Will the maximum number of expected users for the system be less than ten or between ten and hundred or more than hundred?*"), whether the interviews should be structured or unstructured, understanding body language since only about 7% information (1) is communicated verbally, and understanding proxemics or the relationship between people and space around them. When a large number of end users exists it may be cost effective to use questionnaires. An important factor in developing questionnaires is whether free-format questions or fixed-format questions need to be asked.

Another source of requirements are the physical forms and documents that are used for the existing system, including user manuals, filled-in forms used for data capture, standard operating procedures, notes, memos, e-mails, and forms capturing user feedback. Often, the new software system replaces physical forms and documents, and the requirements engineer will have to understand the form's contents and its usage, along with the processes listed in the documents to formulate requirements for the new software system.

JRP is another technique for eliciting requirements—JRP is part of the JAD (Joint Applications Development) philosophy that requires the relevant stakeholders (users and their managers) to participate in a workshop for collaboratively gathering the requirements. The JRP sessions are facilitated usually by an external moderator who ensures that the sessions focus on their objectives and not let egos dominate the proceedings. JRP requires attendance from the technical staff as well so that any requirements that may not be feasible may be immediately pointed out to the concerned users. JRP ensures that collective responsibility for system development occurs and that the users are actively involved in the development process, which encourages them to take ownership of the project.

The discovery prototype is yet another technique for eliciting requirements in which a small-scale working model (sometimes referred to as a "quick-and-dirty" implementation) of the proposed system is developed and given to the users to work with. Based on the philosophy that the users will understand the requirements once they actually see a working sample, the users are frequently able to better articulate their requirements based on their experiences with the prototypes. Discovery prototypes are useful in eliciting requirements that are not clearly understood (sometimes by the users themselves). It must be kept in mind that the prototypes cannot be considered as the first

Number	Software Requirements
1	Interface requirements.
1.1	User interface requirements.
1.1.1	The user should be able to interface with the software by means of keys, knobs, and mouse.
1.1.2	When the user presses a key, the value of the key pressed should appear on the front panel.
1.1.3	When the user turns the knob clockwise, the value of the entity at the cursor should increase; when the user turns the knob counter-clockwise, the value of the entity at the cursor should decrease.
1.1.4	When the user clicks the mouse, the cursor should shift to the position pointed to by the mouse.
1.1.5	The software system should be reliable enough not to miss key presses, knob turns, and mouse clicks, and be fast enough to respond to these events quickly.
1.2	Network interface requirements.
1.2.1	The software system should be accessible over the network using the TCP/IP protocol over Ethernet.
1.2.2	The software system functionality should be accessible over the network using a list of commands.
1.2.3	The software system should ensure that the parameter values updated over the network are reflected on the front panel.
1.2.4	The software system network access should be password protected for security.
1.3	System interface requirements.
1.3.1	The system should interface with the ATM system and the Web-based banking system.
2	System requirements.
2.1	The software system shall accurately manage customer accounts for the bank.
2.2	The software system should allow creation of new accounts, deletion of existing accounts, and operation of accounts; account operations include deposits, withdrawals, and transfers.
2.3	The software system should identify each person allowed to access the system by a unique user-id and password combination.
2.4	The software system should interact with the users by means of graphical user interfaces.
2.5	The software system should respond to queries and accept updates from the ATM system and the Web-based banking system.
2.6	All information should be stored in a central repository that can be accessed by all users and other systems.
2.7	All users should be able to access the central repository over the network.
3	Portability requirements.
3.1	The software system must execute in both Windows and Mac environments.
3.2	The software system should be distributed in a format that allows easy self-installation.
3.3	The software system should be downloadable from the bank's website.

Figure 3. Spreadsheet (or tabular) listing of software requirements for a bank account management system.

version of the system and that their intent was requirements discovery only.

TASK 3: DEVELOP INITIAL SOFTWARE REQUIREMENTS SPECIFICATION

Tabular SRS

Typically, software requirements are listed in categories with each requirement given a unique number for easy reference. Figure 3 shows an example list of requirements for a hypothetical software system that manages accounts for a bank. The software requirements are categorized at the highest level into Interface Requirements, System Requirements, and Portability Requirements; under each category, the requirements are subcategorized—for example, under the Interface Requirements category, the subcategories include User Interface Requirements, Network Interface Requirements, and System Interface Requirements. The simplest way to capture the nesting of levels of software requirements is to use the nested decimal notation as shown in Fig. 3. Typically, a requirement that a software system should have (also called a mandatory or essential requirement) is indicated by using words such as "*shall*," "*should*," or "*must*," with the importance decreasing in this order—that is, a requirement with a "*shall*" is more important than a requirement with a "*should*," and this is more important than a requirement with a "*must.*" In practice, an organization may adopt its own conventions and follow it consistently—it should be noted that modality of terms used in SRS could be different from their usage in ordinary conversation. More importantly, requirements should be unambiguous, correct, consistent, verifiable, complete, and unique (5); an ambiguous requirement is not specific, for example, "*When the user presses the key and/or turns the knob the value displayed should be updated*" —here it is not clear what value should be displayed when both the key and the knob are turned at the same time; a consistent requirement does not conflict with another requirement; a correct requirement is within the domain of the software system, that is, it is a requirement of the software system and not an extraneous factor; a verifiable requirement is one that can be subsequently verified as having been included in the final completed software system; a complete set of requirements capture responses to all possible classes of inputs; and the uniqueness of requirements ensures the that they do not repeat either in the same or another form.

Figure 3 shows both functional and nonfunctional requirements. Requirement 1.1.5 and requirement 2.1 are nonfunctional requirements, whereas the others are functional requirements.

Use-Cases

Another technique for documenting requirements, which is increasingly becoming popular, is use-cases (6); this includes both the use-case diagram and the use-case narratives. The use-case diagram captures all the interactions between the software system and the external entities, whereas the use-case narrative captures the sequential processing of an interaction. Together, they define the scope of the system, give a detailed insight

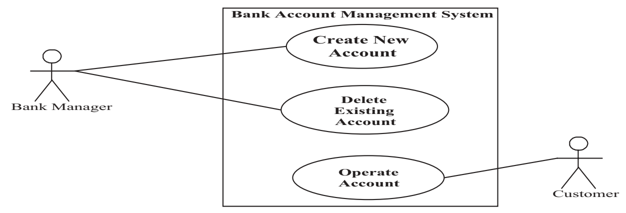
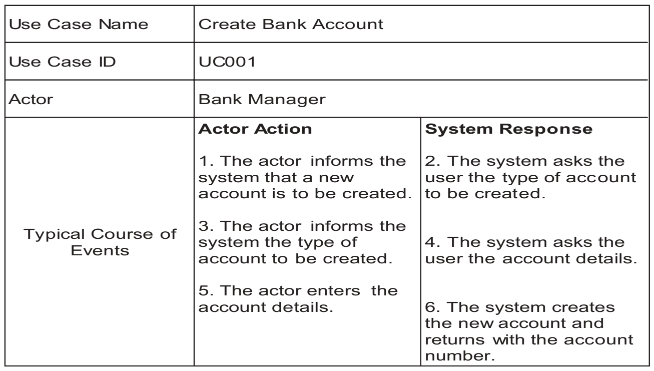


Figure 4. Modeling requirements with UML use-case diagrams.

into the software system to be developed, and allow more accurate effort and schedule estimates. Consider, for example, the use-case diagram shown in Fig. 4, where the stick figure represents a user in a specific role, also called an actor, interacting with a specific functionality or use-case of the software system represented by the oval shape. In Fig. 4, a *Bank Manager* actor exists, and its use-case is *Create New Account* that describes the sequence of activities occurring during interaction between the software system Bank Account Management System and the user *Bank Manager*. Also, it may be noted from Fig. 4 that the *Bank Manager* interacts with the use-case *Delete Existing Account* as well while the *Customer* interfaces only with the *Operate Account* use-case. More importantly, from the use-case diagram of Fig. 4, one may infer that the *Bank Manager* does not interact with the use-case *Operate Account* just as the *Customer* does not interact with the use-cases *Create New Account* and *Delete Existing Account*. Therefore, use-case diagrams also serve as a mechanism to define the scope of the system. Frequently use-cases and scenarios have been used interchangeably in the literature. However, a subtle difference does exist. A scenario is one instance of execution of the use-case, whereas a use-case is an abstract entity; therefore, one use-case can typically represent several scenarios of use. Also, one use-case may include one or more requirements for the software system.

The actual sequence of activities is described in the use-case narrative, an example of which is given in Fig. 5. The narrative of Fig. 5 captures the different ways the user *Bank Manager* can provide inputs to the system and receive outputs from the system, and it captures the detailed sequence of activities undertaken by the system in response to user inputs. Figure 5 is just one way of writing the use-case narrative—different formats are possible, including some that capture exceptional events, alternative events, and responses to multiple users (or, more appropriately, roles). Also, the details in the use-case narrative may be achieved iteratively over time—that is, the first version of the narrative may have only the intent of the use-case, the second version may have more details, the third even more details, and so on till all minutiae are uncovered in the final versions of the narrative. The details in a use-case may be modeled using UML activity diagrams during the modeling task.



Use Case Name	Create Bank Account	
Use Case ID	UC001	
Actor	Bank Manager	
Typical Course of Events	**Actor Action**	**System Response**
	1. The actor informs the system that a new account is to be created.	2. The system asks the user the type of account to be created.
	3. The actor informs the system the type of account to be created.	4. The system asks the user the account details.
	5. The actor enters the account details.	6. The system creates the new account and returns with the account number.

Figure 5. Use-case narrative for the use-case *Create Bank Account* in Fig. 4.

User Stories

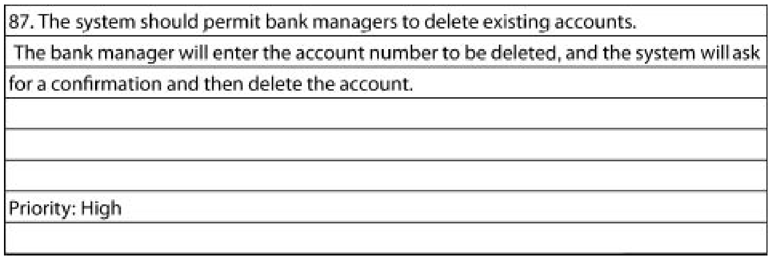
Another technique for capturing initial requirements is user stories (7)—a user story is a brief (usually about three sentences) description of an expected functionality written by the user. An example user story is given in Fig. 6. Each user story is usually captured on an index card and in many cases is written (actually handwritten) directly by the user or the domain expert using the language of the domain. That way it becomes a convenient medium to discuss business requirements with the users. Moreover, from a management perspective, each user story is given a unique number and a priority so that when the stack of user stories is collected, the scope of the system to be developed is well defined and the user stories may be prioritized for different iterations of the software system. In addition, since each user story is expected to take a fixed unit of time to implement, typically about a week, the schedule for the project can also be determined.

Decision Tables

Decision tables (1,8) are yet another technique for documenting requirements, although at a detailed level to capture business rules. Decision tables capture true or false values of conditions and their resultant effects on decisions. For example, if we have the following situation:

A bank account can be established only when the customer has both a social security number (SSN) and documented proof of residency; if the customer can electronically remit her salary to the account each month, then the monthly fee is waived; else a monthly fee is imposed.

To make the decision whether a new account may be approved and, if so, whether a fee needs to be imposed, the decision table shown in Fig. 7 will be useful. In Fig. 7, *T* stands for a condition that is true, whereas *F* stands for a condition that is false, *X* represents an irrelevant condition (that is, it can be either true or false, but its value does not affect the decision), and the applicable decision is indicated with the *a* mark. Therefore, in only one situation can a new account be approved without a fee and that is when all three conditions (customer has SSN, customer has residency proof, and customer can remit salary electronically) are true; when the first two conditions are true, but the customer is unable to remit salary electronically, then a new account is approved but with a fee imposed; in all other cases, a new account is not approved.



87. The system should permit bank managers to delete existing accounts.
The bank manager will enter the account number to be deleted, and the system will ask
for a confirmation and then delete the account.

Priority: High

Figure 6. Requirements capture using a user story.

TASK 4: ANALYZE REQUIREMENTS

The main purpose of requirements analysis is to refine the requirements collected during the requirements elicitation phase. Usually the requirements collected initially do not satisfy all the characteristics of being unambiguous, correct, consistent, verifiable, complete, and unique—the requirements may be considered “raw” that need to be refined and better understood for subsequent phases of software development. The chief technique for this purpose is reviewing the requirements. During reviews, teams of development personnel along with customers/users go over the SRS developed to determine errors and omissions. Any errors and omissions determined will have to be corrected before proceeding. Reviews can be used iteratively during this phase and can be applied to any artifact developed during the process of requirements specification and analysis. Other goals of requirements analysis include requirements prioritization, determining requirements dependencies, determination of versions, schedule formation, staff size determination, preliminary cost estimation, formulation of acceptance tests, and analyzing NFRs.

Requirements Prioritization

Even though at the end of the requirements specification phase we may have a list of all expected software requirements, it may not be feasible for the software development organization to implement all requirements in the very first version of the software, the chief reasons being the time constraints and the delays incurred because of the inevitable changes to software requirements. Therefore, only a subset of the requirements is usually delivered in the first version, and this subset includes only those requirements deemed most important by the customer and considered most feasible by the software development organization. Customers/users determine requirements priorities while their feasibilities are evaluated by the developers—thus, even though the customer, for example in Fig. 3, may state that requirements 1.1.1, 1.1.2, and 1.1.3 are more important than 1.1.4, the current state of hardware technology for the system may not incorporate a knob-based input; in this case, the developers may negotiate with the customer to provide only the requirements 1.1.1 and 1.1.2 with the first release and push requirements 1.1.3 and 1.1.4 to a subsequent release of the software system. An important aspect of negotiating with customers is to ensure that a “win–win” situation is reached for all concerned stakeholders.

Customer has SSN	T	F	F	T	T
Customer has residency proof	F	T	F	T	T
Customer can remit salary electronically	X	X	X	F	T
Bank account approved with fee				α	
Bank account approved without fee					α
Bank account not approved	α	α	α		

Figure 7. Documenting detailed business rules using decision tables.

Identifying Requirements Dependencies

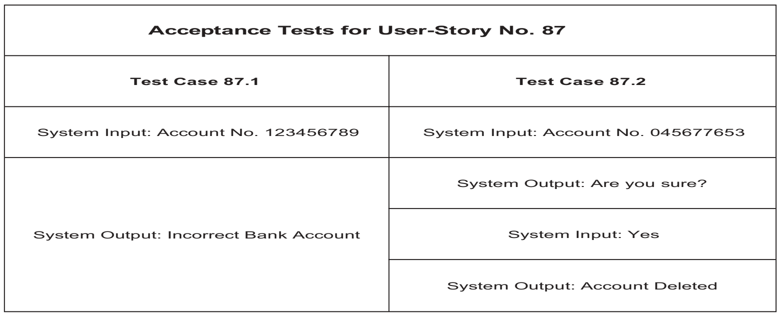
Requirements frequently have dependencies among themselves such that in order to fulfill one requirement, another requirement must have been fulfilled previously. For example, requirement 1.2.3 in Fig. 3 assumes that requirement 1.1.2 is already satisfied since front panel updating ability is a prerequisite for requirement 1.2.3; therefore, requirement 1.1.2 may need to be developed before requirement 1.2.3. Identification of such dependencies will also help with scheduling the software system development such that requirements upon which others are dependent are completed before the requirements that depend on them. When requirements are captured as use-cases, then the dependencies may exist among use-cases. Here the use-case upon which most other use-cases are dependent should be completed before the others.

Requirements Versioning

Prioritization of requirements and identification of their interdependencies can help identify the versioning requirements that determine the distribution of features (a feature is a use-case or a set of requirements) among the different versions of the software system. Therefore, for example, if a high-priority requirement is dependent on a low-priority one, it may be necessary to have the low-priority requirement along with the high-priority one as part of the first version of the system. Versioning assumes software development is based on an incremental process.

Schedule Formation

Versioning is closely tied to scheduling the software development project—thus, if the initial requirements identify the need for ten versions to accommodate all the features required by the customer, then by estimating the time needed to complete each version, an accurate schedule for the project may be developed. Since SRS frequently serves as a contract between the software development



Acceptance Tests for User-Story No. 87	
Test Case 87.1	Test Case 87.2
System Input: Account No. 123456789	System Input: Account No. 045677653
System Output: Incorrect Bank Account	System Output: Are you sure?
	System Input: Yes
	System Output: Account Deleted

Figure 8. Acceptance tests for the user story of Fig. 6 .

organization and the customer, determination of the schedule at this stage will allow the software organization to negotiate properly requirements and cost for the project. For example, using the process chosen for the project, if each version takes four months of development (design and implementation), then for ten versions and assuming sequential development, the project manager can quickly lay out a schedule lasting more than three years.

Staff Size Determination

Another objective of requirements analysis is to identify accurately the staffing requirements for the project—this will help the software organization quickly determine whether the expected schedule and cost can be satisfied or whether more negotiation with the customer is required. Since SRS frequently serves as a contract between the software development organization and the customer, it will be useful to consider staff size requirements at this stage itself. For determining staff size, if prior experience with the company's methodology indicates that a feature takes 10 person-days to design, implement, and test, then for ten features, 100 person-days will be required; based on versioning, which has already decided the approximate sequences of feature-sets to be delivered to the customer, the average staff size for development can be determined quickly.

Project Cost Estimation

Based on personnel requirements and expected length of the project, the project manager can develop or revise preliminary estimates for the budget so that the project benefits may be determined clearly. Again, since SRS frequently serves as a contract between the software development organization and the customer, it will be useful to identify or update project costs based on requirements so that, if needed, additional negotiation regarding requirements and schedule may be held between the two parties. For estimating project costs, past experience could serve as the guide; for example, if past experience indicates that each person-day costs $500, then a project requiring 100 person-days of effort will cost $50,000. However, cost estimation may also be performed using well-known models such as the COCOMO (Constructive Cost Models) that helps, especially COCOMO II (the latest version) (9), to calculate effort in terms of person-months during the requirements specification and analysis phase itself; therefore, if the cost per person-month is known, the project cost can be estimated.

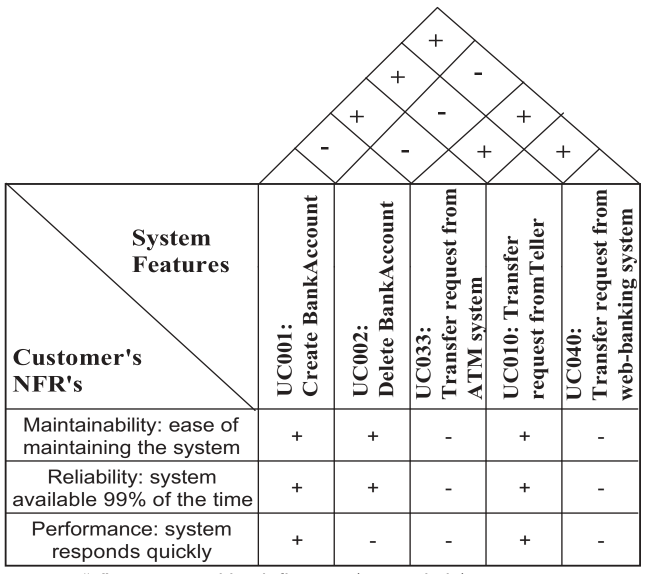
Formulating Acceptance Tests

Requirements analysis serves the very important purpose of developing acceptance tests that document the tests the user will execute on the final system for verifying that the system indeed satisfies the requirements—thus, for example, if we consider the user story shown in Fig. 6, the user may say that the acceptance tests shown in Fig. 8 will be executed to confirm the proper implementation of the user story. Development of acceptance tests during this phase will help clarify requirements and encourage proper design of software. For example, from the customer's acceptance tests, we can request the rules for identifying correct account numbers and, during design, develop a system that will first check that the entered account number is legal before continued processing.

Analyzing NFRs

Analyzing NFRs brings with it its own set of issues, chief among them being the inherent vagueness in defining NFRs. Although several techniques have been proposed, most seem to suffer from their own restricted definitions for the nonfunctional requirements they seek to analyze. However, from a wide applicability standpoint, two techniques stand out: the House of Quality and the NFR Framework. The House of Quality is part of the Quality Function Deployment process that was originally developed for the manufacturing industries in Japan and has been used for software requirements quality analysis as well. In its simplest form, the House of Quality is a matrix where rows represent the NFRs and the columns represent the technical aspects of the project that help achieve the NFRs, and at the intersection of each row and column, we identify the extent to which the technical aspects satisfy the customer's NFRs using qualitative measures such as strongly positive, positive, negative, and strongly negative. The "roof of the house" is the correlation between the technical factors themselves that captures the conflicting or synergistic interactions between the factors. The House of Quality may be extended with other aspects such as project planning and cost for achieving the NFRs. For example, for the bank account management system modeled in Fig. 4, if the user considers NFRs maintainability, reliability, and performance as being the most important, then an analysis of these NFR's using the House of Quality is presented in Fig. 9. In Fig. 9, the technical aspects considered are the two use-cases from Fig. 4, and the three additional use-cases identified for the system. In Fig. 9, we see that in the central part of the figure we denote how the use-cases affect the customer requirements—positive impacts are denoted by "+" whereas negative impacts are denoted by "–". Thus, for example, the use-case *UC033: Transfer Request From ATM System* has a negative influence on Performance, since two systems (the ATM system and the bank account management system) need to interface in order to complete this request, and such system interfaces are typically slower; however, the use-case *UC010: Transfer Request From Teller* has a positive influence on Performance, since only one system (the bank account management system) is involved and the required data are entered by the experienced teller. At the top of the figure, we indicate the trade-offs between the use-cases by using "+" to denote synergies and "–" to denote conflicts. Thus, *UC001: Create Bank Account* conflicts with *UC002: Delete Bank Account* since they are two opposing features with different constraints, whereas UC040 and UC010 synergize each other since developing one feature (UC040) helps in reducing dependence on the other (UC010). This information can be used for project planning, cost/effort determination, and use-case prioritization, and it can be captured in an extended version of Fig. 9.

Some drawbacks of the House of Quality technique include the inability to capture justifications and the flexibility to accommodate various definitions of NFRs. The NFR framework (10) addresses these drawbacks. The chief artifact of the NFR framework is the Softgoal Interdependency Graph (SIG), which captures all information pertaining to the achievement or otherwise of the NFRs. In a SIG, the NFRs are captured as NFR softgoals (depicted by a cloud-shape) that are to be achieved during the process of software development (we are concerned with the requirements specification and analysis), and three relationships are possible between NFR softgoals: AND (depicted by a single arc) means that the parent NFR softgoal is achieved only if all child softgoals of the AND-relationship are achieved; OR (depicted by a double arc) means that the parent NFR softgoal is achieved if even one of the child softgoals in the OR-relationship is achieved; and EQUAL relationship relates (by a line) one child to a parent and the parent is achieved if the child is achieved. Even though we used the word "achieved," the actual NFR framework term is satisfied, which means relative satisfaction and not absolute satisfaction. The system features are captured in the SIG by means of operationalizing softgoals (depicted by a dark-bordered cloud-shape), and the contributions of the operationalizing softgoals to the NFR softgoals are captured by contributions that come in four flavors: strongly positive (++), positive (+), negative (–), and strongly negative (– –). The contributions can be propagated up the SIG using well-defined propagation rules of the NFR framework. Finally, the reasons for the contributions (and, in fact, any element of the SIG) can be captured by means of argumentation softgoals (depicted by dashed-bordered cloud-shapes). The information in the House of Quality of Fig. 9 regarding two use-cases (UC001 and UC033) is captured easily by the SIG of Fig. 10—in this figure, the use-cases have been represented as operationalization softgoals (operationalizations refer to the artifacts currently under consideration that help achieve or deny NFR softgoals, and during the requirements specification and analysis phase, the artifacts could be requirements, use-cases, user stories, or decision tables). In Fig. 10, it may be noted that the argumentation softgoals capture justifications, the decomposition of the NFR softgoals (the AND–OR–EQUAL relationships) capture the definitions of the NFRs, and the propagation rules permit the determination of whether the NFRs are achieved, and, more importantly, they help identify the reasons in either case.



System Features / Customer's NFR's	UC001: Create BankAccount	UC002: Delete BankAccount	UC033: Transfer request from ATM system	UC010: Transfer request fromTeller	UC040: Transfer request from web-banking system
Maintainability: ease of maintaining the system	+	+	-	+	-
Reliability: system available 99% of the time	+	+	-	+	-
Performance: system responds quickly	+	-	-	+	-

Legend: "+" means positive influence (synergistic)
"–" means negative influence (conflicting)

Figure 9. Application of House of Quality to analyze nonfunctional requirements (NFRs).

TASK 5: MODEL REQUIREMENTS

Modeling helps to clarify requirements with the customers and users, so that the SRS is unambiguous, complete, and captures all essential requirements. Modeling is based on the premise that "a picture is worth a thousand words" and that information can be captured concisely using pictures. Modeling may be accomplished using modeling notations or formal logic.

Modeling Notations

Common modeling approaches include unified modeling language (UML) diagrams (11), data flow diagrams (DFDs) (1), and entity relationship diagrams (ERDs) (1). The common diagrams used for requirements analysis in UML include use-case model diagrams (as in Fig. 4), activity diagrams, class diagrams, sequence diagrams, and state charts. Noun-parsing of use-case narratives helps discover potential classes for developing class diagrams. DFDs are useful for modeling business processes and for capturing the interaction between manual and computerized processes (8). DFDs can be hierarchical with each level expand-

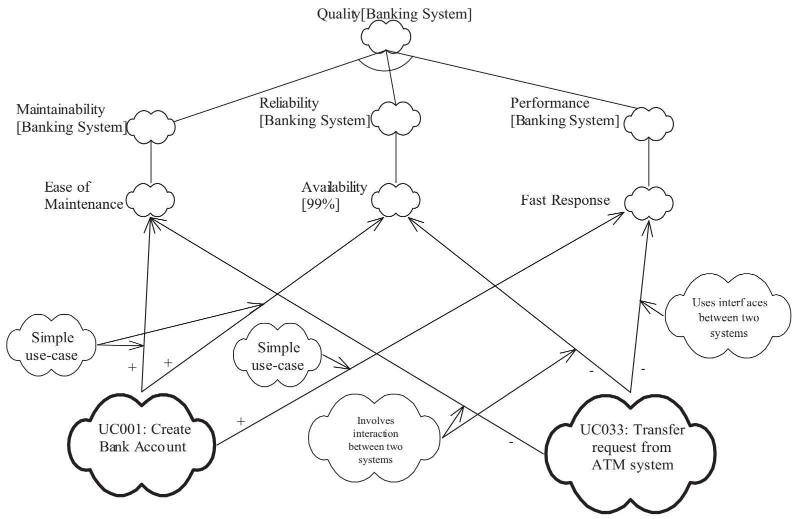

Quality[Banking System]
Maintainability [Banking System]
Reliability [Banking System]
Performance [Banking System]
Ease of Maintenance
Availability [99%]
Fast Response
Uses interfaces between two systems
Simple use-case
Simple use-case
UC001: Create Bank Account
Involves interaction between two systems
UC033: Transfer request from ATM system
+
+
+
-
-
-

Figure 10. SIG of the NFR framework.

ing on the processes at a higher level—level 0 is the highest level and is called the context diagram, which considers the entire system as one black box; level 1 is the lower level that expands the black box of level 0; level 2 is the next lower level; and so on. A rule of thumb is to ensure that a higher level process is decomposed into at least five detailed processes at the lower level. ERDs can capture relationships between real-world entities in the problem domain and serve as an excellent technique to model data requirements for the domain. Petri nets have also been used to model requirements, but their semantics may be difficult for the average user to understand. Petri nets help to model concurrent activities where the order of occurrence is not important (12).

Formal Methods

Formal techniques for modeling requirements include first-order logic, temporal logic, Object Constraint Language, Z language, algebraic specifications, Specification and Description Language, and Software Cost Reduction (12). Formal techniques attempt to avoid problems associated with misinterpreting semantics of a model or inherent ambiguity in using natural languages (such as, for example, English) for documenting requirements. Again without going into details, it may be safely mentioned that practicality and scalability of these methods is limited even though claims have been made that for mission-critical systems these techniques are useful.

REQUIREMENTS MANAGEMENT

One of the major issues with requirements analysis is what to do when an error is discovered in the requirements—the error may be an incorrect requirement, ambiguous requirement, or even a missing requirement. The processes by which changes to the requirements specifications are handled fall under the category of requirements management. An important aspect of requirements management is ensuring that all stakeholders during the requirements phase always refer to the most updated set of requirements. Frequently a committee, called the Change Control Board, is formed to manage requirements, and all requests for changes need to be forwarded to this committee, which then decides on the necessity for the changes and, if required, updates the requirements and disseminates change notices.

Another facet to requirements management is the development of traceability tables where each traceability table relates requirements to an aspect of the system. Thus, for example, there could be a features traceability table, source traceability table, dependency traceability table, subsystem traceability table, and interface traceability table. The features traceability table relates each requirement to a feature (or functionality) of the system, the source traceability table relates each requirement to its source, the dependency traceability table indicates how requirements are dependent on each other, the subsystem traceability table relates each requirement to the subsystem the requirement applies to, and the interface traceability table captures the relationship between requirements and system interfaces.

TOOL SUPPORT FOR REQUIREMENTS SPECIFICATION AND ANALYSIS

The most important software tools for requirements specification are a word processor or spreadsheet software. Both of them help capture software requirements—the advantage with a word processor is that diagrams and detailed descriptions can be captured, whereas the main advantage of a spreadsheet software is that the requirements can be represented in tabular forms that help to add, delete, and modify requirements easily. Also, both tools help capture categorized or nested requirements. Some modeling environments such as IBM's Rational Requisite Pro (13) and Telelogic's System Architect (14) help draw use-cases and determine requirements dependencies. Drawing applications such as Smartdraw (15) and Microsoft Visio (16) have libraries for modeling requirements using UML, ERD, and

DFD notations. In order to capture different versions of requirements a configuration management tool such as open-source Concurrent Versions System (17) or IBM's Rational ClearCase (13) may be useful. Very often the versioning system is integrated with a Web-based system that is usually developed for each project so that interfaces for viewing and uploading the latest requirements are readily available. Sometimes, the Web-based repository serves as a bulletin board service as well so that all update notifications are issued at one place—the Web-based system then becomes a central repository for all data pertaining to the project including requirements.

DOORS (Dynamic Object Oriented Requirements System) (18) marketed by Telelogic is a tool for collaborative requirements management that helps capture, view, and update the latest requirements for a project. In addition, DOORS can help with change management and can be integrated with third-party application tools.

The Automated Requirements Measurement (ARM) (19) tool from NASA can evaluate requirements specified in the English language for words, including imperatives, directives, continuances, options, and weak phrases. These constructs are measured in terms of size, specification depth, readability, and text structure.

RESEARCH PROBLEMS IN REQUIREMENTS SPECIFICATION AND ANALYSIS

As stated, one of the important problems with the requirements specification and analysis phase is that users are frequently unable to articulate their needs—the roots for this problem may lie in an interdisciplinary area overlapping the fields of neuroscience, psychology, and cognition. A team of interdisciplinary researchers need to analyze the causes of this problem so that tools—both hardware and software—to better elicit requirements from people may be developed. Another problem is that of modeling requirements for communication among stakeholders (20)—current communication tools that use natural language such as English are prone to ambiguities, whereas a modeling language such as UML requires training for proper usage. Another issue that requires additional study is that of analysis paralysis, which refers to the problem of being in a state of perpetual analysis with no end to the phase in sight—the answer to the question "*When has enough analysis been done?*" is of considerable interest to practitioners (21). A very useful development in this area would be identification of conceptual patterns of user needs so that these concepts may be used as building blocks for developing requirements for new systems; in addition, these concepts may be used as a mechanism for communicating with users. For example, concepts such as human–machine interaction, data access, and networking seem to be common for most systems, and these could be designated as patterns that can be used for building new systems. Yet another research area is quantification of requirements quality (10) so that practitioners know when they are done—currently this field is in its infancy with tools such as ARM providing mainly syntactic metrics, but it needs to be developed to include semantic or conceptual levels.

BIBLIOGRAPHY

1. IEEE Std 830-1998, *Recommended Practice for Software Requirements Specifications*, June 25, 1998.
2. J. L. Whitten and L. D. Bentley, *Systems Analysis and Design Methods*, 7th ed. New York: McGraw-Hill, 2007.
3. R. S. Pressman, *Software Engineering: A Practitioner's Approach*, 6th ed. New York: McGraw-Hill, 2005.
4. S. R. Schach, *Object-Oriented & Classical Software Engineering*, 6th ed. New York: McGraw-Hill, 2005.
5. S. L Pfleeger and J. M. Atlee, *Software Engineering: Theory and Practice*, 3rd ed. Englewood Cliffs, NJ: Prentice Hall, 2006.
6. L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*. Boston, MA: Kluwer Academic Publishers, 2000.
7. R. Denney, *Succeeding with Use Cases: Working Smart to Deliver Quality*. Reading, MA: Addison-Wesley Professional, 2005.
8. M. Cohn, *User Stories Applied: For Agile Software Development*. New York: Wiley, 2004.
9. J. Wetherbe and N. P. Vitalari, *Systems Analysis and Design: Traditional, Best Practices*, 4th ed. St. Paul, MN: West Publishing, 1994.
10. B. W. Boehm, C. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, Cost models for future life cycle processes: COCOMO 2.0, *Ann. Soft. Eng.*, **1**(1): 57–94, 1995.
11. L. Chung and N. Subramanian, Process-oriented metrics for software architecture adaptability, *Proc. International Symposium on Requirements Engineering*, IEEE Computer Press, Aug–Sep. 2001, pp. 310–311.
12. A. van Lamsweerde, Requirements engineering in the year 00: a research perspective, *Proc. 22nd International Conference on Software Engineering (ICSE'00)*, Limerick, Ireland, June 5–9 2000, pp. 5–19.
13. P. Zave and M. Jackson, Four dark corners of requirements engineering, *ACM Trans. Soft. Eng. Methodology*, **6**(1): 1–30, 1997.
14. www.uml.org
15. http://satc.gsfc.nasa.gov/tools/arm/
16. http://www.telelogic.com/corp/products/doors/index.cfm
17. http://www-306.ibm.com/software/rational/
18. http://www.nongnu.org/cvs/
19. www.smartdraw.com
20. http://office.microsoft.com/en-us/visio/default.aspx
21. http://www.telelogic.com/products/systemarchitect/index.cfm

Narayanan Subramanian
University of Texas at Tyler
Tyler, Texas