DEVELOPING USER-CENTRIC SOFTWARE REQUIREMENTS SPECIFICATIONS

DEVELOPING USER-CENTRIC SOFTWARE REQUIREMENTS SPECIFICATIONS

By
HONGQING SUN, B.SC.

A Thesis
Submitted to the School of Graduate Studies
in partial fulfilment of the requirements for the degree of

Master of Applied Science
Department of Computing and Software
McMaster University

MASTER OF APPLIED SCIENCE (2007)
(Computing and Software)

McMaster University
Hamilton, Ontario

TITLE:
DEVELOPING USER-CENTRIC SOFTWARE REQUIREMENTS SPECIFICATIONS

AUTHOR: Hongqing Sun, B.Sc. (NANKAI UNIVERSITY, CHINA)

SUPERVISOR: Dr. Alan Wassyng

NUMBER OF PAGES: xiv, 183

Abstract

Software systems with intensive user-computer interactions account for a fairly large part of the total real world software applications, such as web applications, MS Windows applications, GNOME/KDE applications etc. We call this kind of software *user-centric* software, denoting a defining characteristic which is that they are usable directly by users.

Exhibited in this thesis is a systematic approach for developing a software requirements specification (SRS) for user-centric software. While this approach conforms to the well-recognized software requirements engineering process model, which contains the processes of requirements elicitation, analysis, specification and validation, it is tailored to user-centric software. The user-centric ideas are embodied and applied throughout our approach. In the elicitation process, the *joint requirements development* (JRD) sessions (known as requirement workshops) are advocated, and step-by-step guidance is developed leading to a natural flow from the raw problem descriptions to user requirements - the use case model. Further, based on the various object-oriented analysis paradigms, we build a systematic analysis process to seek analysis classes, where domain classes are harvested from the composed data hierarchies of all use cases, and application classes and functions are captured from sequence diagrams. Especially, our notation of boundary classes provides considerable flexibility in the user interface (UI) design phase. During the SRS process, functional requirements derived from the analysis model are specified according to a class specification template. Moreover, the three-level validation process positively involves the user's participation facilitating assurance that the right software is built. Also, to demonstrate the practicability of this approach, it is applied in a case study dealing with developing the SRS of a photodynamic therapy (PDT) treatment planning application.

Acknowledgements

I am greatly indebted to Dr. Alan Wassyng, my supervisor, for his good teaching, constructive guidance, enlightening suggestions, friendly help and encouragement throughout the entire period of my studies and this research. Without his support, it would have been impossible to finish this thesis. During the two-year studies I have known him as a knowledgable, humorous, and principle-centered person. His substantial experience and integral view on research and his mission for providing 'high-quality work', have made a deep impression on me. He could not even realize how much I have learnt from him.

I would like to express my sincere thanks to Dr. Tom Maibaum and Dr. Douglas Down, my defense committee, for their valuable time and comments on my research. I am very grateful to Dr. Tom Maibaum, for his generous and stimulating suggestions throughout this research.

Deep thanks to Dr. Ridha Khedri for his valuable recommendation and supply of research materials and suggestions. I would also like to thank Dr. Jacques Carette, Dr. Spencer Smith and Dr. Wolfram Kahl for their precious comments during this research.

Much thanks to University Health Network for providing the case study resources. Sincere appreciation to Sean Davidson, for him coming to McMaster and suggesting wonderful hints on the case study.

I owe much thankfulness to my parents for their endless love, encouragement and moral support. I wish their happiness and good health all the time.

Especially, I would like to give my special thanks to my wife Na whose love and care enabled me to complete this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In software engineering, a software development life cycle (SDLC) contains at least four phases which are requirements, design, construction and testing. Although there exist significant differences among various SDLC models, such as Waterfall [72], Spiral [11], Unified Process [50] and Agile [2] etc., all of them start with the requirements phase. A *software requirements specification* (SRS) is the deliverable of the requirements phase, grounding later development phases.

As the characteristics of different software types vary, such as embedded software, safety critical software, scientific computational software etc., we should not expect to be able to define a uniform approach for developing a SRS that can fit all types of software. This thesis addresses an approach which not only gives step-by-step guidance for activities involved in developing a *user-centric software* SRS, but also examines the underlying principles of software requirement engineering.

This chapter is organized as follows: Our meaning of *user-centric* is stated in section 1.1. Section 1.2 presents the context of this research — the software requirements engineering processes. The motivation of this research is presented in section 1.3. The research problem and scope are identified in section 1.4. Section 1.5 summarizes the contributions of this thesis. Finally, section 1.6 presents the organization of the remainder of the thesis.

1.1 Meaning of User-Centric

This section explains what we mean by "user-centric" in different situations.

1.1.1 User-Centric Software

Currently, there is no formal definition of a user-centric software system. In [65], a user-centric software system is described as an interactive system, where interactive systems are those that enable users to communicate with them [31].

With the popularity of software applications in the modern world, software systems with intensive user-computer interactions constitute a fairly large ratio of the total. Examples include web-based software, office software, accounting software, payroll software, patient information management software, etc. We define this kind of software to be *user-centric software.* User-centric software has the following characteristics.

- Intensive user-computer interactions.
- Oriented to users' responsibilities or goals.
- Usability sensitive - appropriate interface, easy operation.
- Users' HCI satisfaction is a priority.

1.1.2 User-Centric Development

According to [29, 38], user-centric development meets the following characteristics.

- The software development team includes some users.
- The user is an equal participant with the developers in making development decisions.
- The software development team develops the software using an iterative approach.
- A series of versions or prototypes of the software are developed and delivered with feedback from users of earlier versions of the software driving development of later versions.

So, in a user-centric development approach, the software system grows iteratively and incrementally. Users are continually involved in the process, giving the developers regular feedback. The development team responds quickly to users' requests and feedback.

The benefits of user-centric software development are the following.

- The right software is built - usability and appropriate quality ensured.
- Users will be satisfied with the software - user's satisfaction and acceptance ensured.
- Frequent communications ease the gap between the development team and the application domain users.

As a result, instead of having to work in a way that suits the software, users will have software that lets them work at the way they really want to. Also, users' involvement let them see their contributions as development progresses. It makes them enthusiastic about the development and delivery of the system, and reduces their reluctance to accept and use the system when it goes live.

1.1.3 User-Centric in Our Approach

The requirements phase unavoidably involves various stakeholders, such as customers, domain experts and users. We advocate user-centric development in our requirement engineering approach. Although our approach only focuses on the requirements phase, all the principles of user-centric development are applied.

Specifically, we use the following techniques.

- JRD. Joint Requirements Development originated from JAD[1]. JRD sessions are held in a "controlled environment, facilitated by a business analyst, wherein users participate in discussions to elicited requirements, analyze their details and uncover cross-functional implications." [87]

[1] Joint Application Design/Development (JAD), is a methodology that involves the client or end user in the design and development of an application, through a succession of collaborative workshops called JAD sessions. Chuck Morris and Tony Crawford, both of IBM, developed JAD in the late 1970s and began teaching the approach through workshops in 1980.

JRD Sessions are:

1. Very focused
2. Conducted in a dedicated environment
3. Quickly drive major requirements
4. JRD participants typically include:
   - Session leader
   - 1 Business analyst or requirements engineer
   - Various user types
   - 1 Developer
   - 1 Domain expert

- Modeling. Different models are used to communicate with users and capture various levels of requirements, such as the *problem context diagram*, the *scenario table*, the *activity diagram* and the *sequence diagram*.

1.2 Context of this Research

A common model of software engineering [28] states that the software development process involves the following phases: software requirements, preliminary design, detailed design, coding, unit testing, integration testing, system testing, deployment and maintenance. Specific development approaches, as mentioned earlier, may perform the phases differently from the above sequence. However, in all of the software development approaches, the requirements phase is the foremost one of any software development activities. A software requirements specification (SRS) is the output of the requirements phase, describing the desired external behavior of the system to be built and underlying the later design and testing phases. The theme of our research is the whole requirements phase covering all the activities related to developing an SRS.

In software requirements engineering, a common process model of the requirements phase consists of the following five processes [81].

- *Software Requirements Elicitation.* The process through which the analysts understand the user's needs and the constraints on the software system. Techniques include interviews, questionnaires, conversations, study of domain documents and use cases [49, 74].

- *Software Requirements Analysis.* The process of analyzing the user's needs to arrive at a definition (solution) of software requirements. Typical analysis techniques include object-oriented analysis, function-oriented analysis, state-oriented analysis. Details of these are introduced in almost all software requirements engineering books, such as well known ones like [28, 78].
- *Specifying Software Requirements.* The process of writing a document that precisely and completely specifies the software requirements of the software system. This process is based on the results of requirements elicitation and requirements analysis. Various templates, such as IEEE-830 [44], Volere [69], ESA [15], and specification languages, such as UML [62], Z [1], Tabular Expressions [66], are used for documenting the SRS.
- *Validating Software Requirements.* The process to check and ensure the Software Requirements Specification is in compliance with the user's needs and is adequate for proceeding to the design phase. Versatile tools and techniques exist for various kinds of specifications, like the SCR toolset [40], PVS [79], and formal reviews.
- *Managing Software Requirements.* The planning and controlling of the requirements elicitation, specification, analysis, validation and verification activities.

The use of the term "engineering" for the software requirements phase implies that systematic, efficient and iterative techniques should be applied to ensure that software requirements are complete, consistent, correct and reusable. In our research, each of the processes mentioned above shall be developed using an engineering approach to improve the likelihood of developing a high quality SRS.

1.3 Motivation of Research

A software requirements engineering approach and resulting software requirements specification not only provide a basis for subsequent development phases, but provides the basis also for the success of the whole project. For example, the London

Ambulance Service Dispatch System closed down in 1992 after only two days of operation. This well known system failure was caused by poor requirements engineering – "poor requirements analysis within the social domain" [80]. Another failed project, Performing Rights Society PROMS, was abandoned in 1992 after £11 millon was spent. It was reported that they failed to set out the requirements in a form that could be understood and checked by ordinary people and the specifications were ill-conceived [13]. A good requirements engineering approach and high quality SRS are essential to guarantee the success of a software project.

Moreover, some industrial software systems must be validated or certified to meet certain regulations, guidelines, or standards before they can be used. For example, FDA General Principles of Software Validation [35] is for medical software, and Software Considerations in Airborne Systems and Equipment Certification [73] for avionic software.

As stated previously, user-centric software systems account for a large proportion of the total number of systems in use, and our literature review did not unearth any detailed engineering oriented processes aimed at developing an SRS for this kind of software. This provided our inspiration to develop a practical approach for developing a high quality SRS for user-centric software.

1.4 Research Problem and Scope

The following questions drove our research.

1. What activities are included in each process?
2. What are the principles in all of the activities?
3. How do we ensure we produce a high quality user-centric software requirements specification?

To solve the research problem we conducted the following activities:

- We reviewed the literature about software requirements engineering approaches.
- We developed the elicitation process according to the characteristics of user-centric software.

- We evolved the analysis process from different object-oriented analysis methods.
- We suggested a tailored SRS template based on the IEEE 830-1998 template.
- We developed a three-level validation process.
- We exposed the principles of all activities.
- We built a systematic engineering approach for the requirements phase.
- We applied our approach to a case study to show the applicability of our ideas.

1.5 Contribution of this Thesis

This thesis provides a practical and systematic software requirements engineering approach for developing user-centric software requirements specifications.

- We clarified some basic concepts in software requirements engineering.
- We advocated a user-centric development method that ensures that the right software system is built.
- We built a problem domain decomposition/composition model during the elicitation process which provides a technique to thoroughly understand the problem.
- We developed a scenario table model to capture user requirements.
- We developed a systematic process to seek classes and their relationships.
- We defined the boundary class without restricting the design choices.
- We proposed a class specification template to document functional requirements.
- We determined and documented the guidance of all activities.

1.6 Thesis Structure

The remainder of this thesis is organized as follows.

Chapter 2 presents an overview of software requirements engineering, where key concepts are clarified and some typical requirements engineering approaches are introduced.

Chapter 3 and Chapter 4 constitute the key part of this thesis. Chapter 3 states the approach we developed. Each process is refined into operable activities, and underlying rationale and guidance are developed. Chapter 4 shows the application of our approach - a case study of a photodynamic therapy treatment planning software system.

Chapter 5 presents the conclusion of this thesis as well as recommendations for future work.

Two appendices are included. Appendix A presents the tailored SRS template with explanation of each subsection. Appendix B is the partial SRS of a case study.

Chapter 2

Overview of Software Requirements Engineering

This overview focuses on the clarification of concepts involved in software requirements engineering and introduces some typical software requirements engineering approaches.

2.1 Clarification of Some Concepts

Software requirements engineering is the science and discipline concerned with establishing and documenting software requirements [81]. A commonly supported process model of software requirements engineering includes processes of elicitation, analysis, specification, validation and management, that were already introduced in section 1.2.

While people are increasingly realizing the importance of the software requirements phase and making efforts to establish good software requirements, they are sometimes confused by some of the concepts in this area, such as *system requirements*, *business requirements*, *user requirements*, *software requirements*, *functional requirements*, *non functional requirements* etc. In this section, we will present a unified/consistent set of definitions which are used in our documentation and processes.

2.1.1 What is a System?

In the publication of IEEE 1233-1998, a system is defined as "An interdependent group of people, objects, and procedures constituted to achieve defined objectives or some operational role by performing specified functions. A complete system includes all of the associated equipment, facilities, material, computer programs, firmware, technical documentation, services, and personnel required for operations and support to the degree necessary for self-sufficient use in its intended environment."

However, the definition above is the meaning in the large sense [47]. In software requirements engineering, the software is focused on. We give a "small sense" definition in the context of software requirements engineering for user-centric software – "A system is a combination of software and the underlying general-purpose computer."

In this sense, people and other systems are certainly not included. The functionality of the system is achieved by the software. It is the software that transforms the computer into a system, which can accomplish the desired purpose. We suppose that the underlying general-purpose computer always works properly.

When we talk about software or software system, we mean the same thing. It refers to the software under discussion. The functionality of the system is the same as the functionality of the software.

2.1.2 What is a Requirement?

In Webster's Dictionary 1989, *requirement* is defined as "something required; something wanted or needed." In IEEE terminology [43], requirement is defined as: "(1) A condition or capability needed by a user to solve a problem or achieve an objective. (2) A condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents. (3) A documented representation of a condition or capability as in (1) or (2)." Alan Davis [28] describes the concept of requirement to be "A user need or a necessary feature, function, or attribute of a system that can be sensed from a position external to that system." Kotonya and Sommerville [53] define requirement as "A statement of a system service or constraint".

The above definitions do not distinguish the differences between the *system* *requirements*, *user requirements* and *software requirements*. In the following paragraphs, we will clarify the relationships and differences between them.

System Requirements. System requirements describe the behavior of the system as seen from the outside, for example, by the user [81]. So they are the high-level requirements that represent the system as a whole, which contains both hardware and software.

User Requirements. User requirements (also called Stakeholder requirements) describe the tasks the users must be able to accomplish with the product [86]. Sommervile and Sawyer [78] define the user requirements as "... abstract requirements describing the system services which people need to use the system and to integrate it with their business processes." User requirements are usually captured in use cases or scenario descriptions.

Apparently the user requirements represent the system's behavior from the user's point of view. As a result, in any pure software case, especially for stand-alone software systems which will be installed in a general-purpose computer, they could be regarded as system requirements.

Software requirements. We give the definition of a *software requirement* as "A statement of a function or a constraint of the system from the software developer's point of view."

Software requirements consist of all the requirements the software must demonstrate for the system to meet the user requirements. They are derived from analysis of user requirements. Software requirements include the so-called functional requirements and non-functional requirements, where user interfaces are considered to be part of functional requirements in our approach.

Functional requirements (behavior requirements) define what the system does, namely, the functions (actions) of the system. They describe all the inputs and outputs to and from the system as well as information concerning how the inputs and outputs interrelate [28].

Non-functional requirements define the constraints of the system as it performs its functional requirements. They include a description of the system's usability, reliability, performance, security, maintainability, portability, implementation, interface, operations, packaging and legal obligations.

2.1.3 What is a Software Requirements Specification (SRS)

Alan Davis [28] defines "A *software requirement specification* is a document containing a complete description of what the software will do without describing how it will do it." Another formal definition in [81] states "A *software requirements specification* is the document that clearly and precisely describes each of the essential requirements (functions, performance, design constraints, and quality attributes) of the software and the external interfaces."

The above two definitions summarize what an SRS should be. An SRS states what is to be built, it records the functional requirements, which present what behavior the software system should offer, and non-functional requirements, which describe the specific constraints on the system.

2.2 Software Requirements Engineering Approaches

This section briefly introduces the main software requirements engineering approaches that can be found in the literature. These approaches differ in their analysis methods, modeling techniques, specification languages, or their combinations of these components.

2.2.1 Function-Oriented Analysis Approach

The Function-Oriented Analysis Approach defines the required behavior as a mapping from inputs to outputs. The system functionality is decomposed into a function hierarchy. Each level of the hierarchy adds detail about the processing steps necessary to accomplish the more abstract function in the level above. The function above controls the processing of its subfunctions. Data flow diagrams (DFD), entity relationship diagrams (ERD) and data dictionaries are the main modeling techniques of this approach. The details of a function are defined using a textual specification called a "MiniSpec", in the form of natural language, decision tables, or a procedure definition language (PDL). Techniques belonging to this approach include Structured Analysis and Design Technique (SADT) [71], Structured Requirements Definition (SRD) [63], Structured Analysis and System Specification (SASS) [30], Structured System Analysis and Design Methodology (SSADM) [33] etc.

As a top-down functional decomposition approach, the resulting solution usually lacks flexibility and is hard to scale up and extend in the future.

2.2.2 Object-Oriented Analysis Approach

The Object-Oriented Analysis Approach (OOA) originated from Object-Oriented Programming (OOP) and Object-Oriented Design (OOD), and partitions the system into interacting analysis objects which are linked by various relationships. Each object encapsulates a set of services (also called functions or methods) and a state (a set of data, a data structure, or attributes). In object-oriented analysis, the analysis mainly contains the following activities: finding the analysis classes, structuring the analysis classes, describing interactions among analysis classes, defining services of analysis classes and defining attributes of analysis classes. The first object-oriented analysis approach, Object-Oriented Systems Analysis: Modeling the World in Data (OOSA) [77], adopts the entity-relationship model to capture the domain object relationships of a software system. Since then, numbers of OOA approaches have emerged such as Object-Oriented Analysis (OOA) [20], Object-Oriented Modeling and Design (OMT) [75], Object-Oriented Analysis and Design with Applications (OOAD) [12] and Object-Oriented Software Engineering (OOSE) [49], which support both the declarative and interactive modeling of a software system. In particular we note that, OOSE invented the use case technique.

Today, one of the market leading approaches is the Unified Process (UP) [50], which is roughly a convergence of [10, 12, 49]. Also, the Unified Modeling Language (UML) [62], which consists of a set of dynamic and static models, is gradually standardizing the modeling techniques used in the OOA world.

However, the OOA techniques of UP are deeply influenced by software design. The control classes introduced actually model internal aspects of a software application.

2.2.3 Goal-Oriented Approach

Goal-Oriented requirements engineering is concerned with the use of goals for eliciting, elaborating, structuring, specifying, analyzing, negotiating, documenting, and modifying requirements [83]. A goal is an objective the system should achieve. Once a preliminary set of goals is obtained and validated by stakeholders, many other goals can be identified by *AND/OR refinement* and by *abstraction*, just by asking *HOW and WHY* questions about the available goals, respectively. The refinements stop until subgoals can be assigned to individual *agents*[1] in the system, and in the environment. Lowest level goals become requirements in the former case, and expectations in the latter. A number of papers describe the detailed modeling techniques and tools used in goal based requirements engineering, see [5, 27, 59, 82]. One merit of this approach is that non-functional requirements are apparently analyzed and transformed into goals.

2.2.4 Problem Frames Approach

Jackson advocates a different methodology to develop requirements and specifications [47, 48]. In this approach, a problem is decomposed into sub problems. Each sub problem is a projection of the whole problem and should fit a certain *problem frame*. A problem frame is a kind of pattern that captures and defines a commonly found type of simple sub problem. There are five basic problem frames: required behavior frame, command behavior frame, information display frame, simple workpieces frame, and transformation frame. A key component of this approach is the *problem diagram* which contains both the problem context and related requirements. This approach focuses upon the problem domain; it is a return to what might be considered old-fashioned practice [13].

Moreover, there exist dozens of other approaches and supported specification languages such as state-oriented approaches (Z, VDM, and Petri Net), the software cost reduction (SCR) approach (SCR tables) [40], viewpoint-oriented approach [53], agent-oriented approach, volere requirements approach (a require-

1

Agent : a human, device or system component. A system agent is a part of the system being modeled. An environmental agent is a part of the system environment.

ments shell) etc. Also, there are several tools that support requirements capture and traceability, such as DOORS/ERS (Telelogic), Analyst PRO (Goda Software), and Rational RequisitePro (IBM Rational). Most modeling notations and specification languages have tool support. A requirements tools survey is located in http://www.volere.co.uk/tools.htm. Lastly, a large bibliography of requirements engineering is maintained by Alan Davis at his website: http://web.uccs.edu/adavis/UCCS/index.htm.

Chapter 3

A Practical Approach

This chapter explains the approach we developed. Firstly, a whole picture is given to demonstrate the approach. Then various processes and steps are elaborated on.

3.1 An Overview of the Approach

The approach we developed includes the well-recognized software requirements engineering processes: *Requirements Elicitation , Requirements Analysis, Software Requirements Specification and Requirements Validation.* While we put much attention on the characteristics of user-centric software, we also based our approach on solid underlying principles and rationale derived from the literature.

At the elicitation stage, the system user's responsibilities are refined into user tasks, and then scenarios fulfilling these tasks are explored and eventually captured into use cases, which form the user requirements. During the analysis stage, through analyzing each use case, the required functionality and behavior of the system are allocated into different functional parts of the system - analysis classes, which consist of domain classes and application (boundary) classes. The specification process uses the results of the elicitation process and analysis process, to specify the functional requirements and system constraints (non-functional requirements) in the form of a software requirements specification. The validation process embraces a number of steps to ensure that all the user requirements and constraints have been accurately captured and documented. Validation can occur during or after the other processes.

The approach we developed is systematic, incremental and iterative. It has somehow a style similar to UP (Unified Process) [50] although it just covers the requirements phase. Figure 3.1 and Figure 3.2 are two views of the approach.

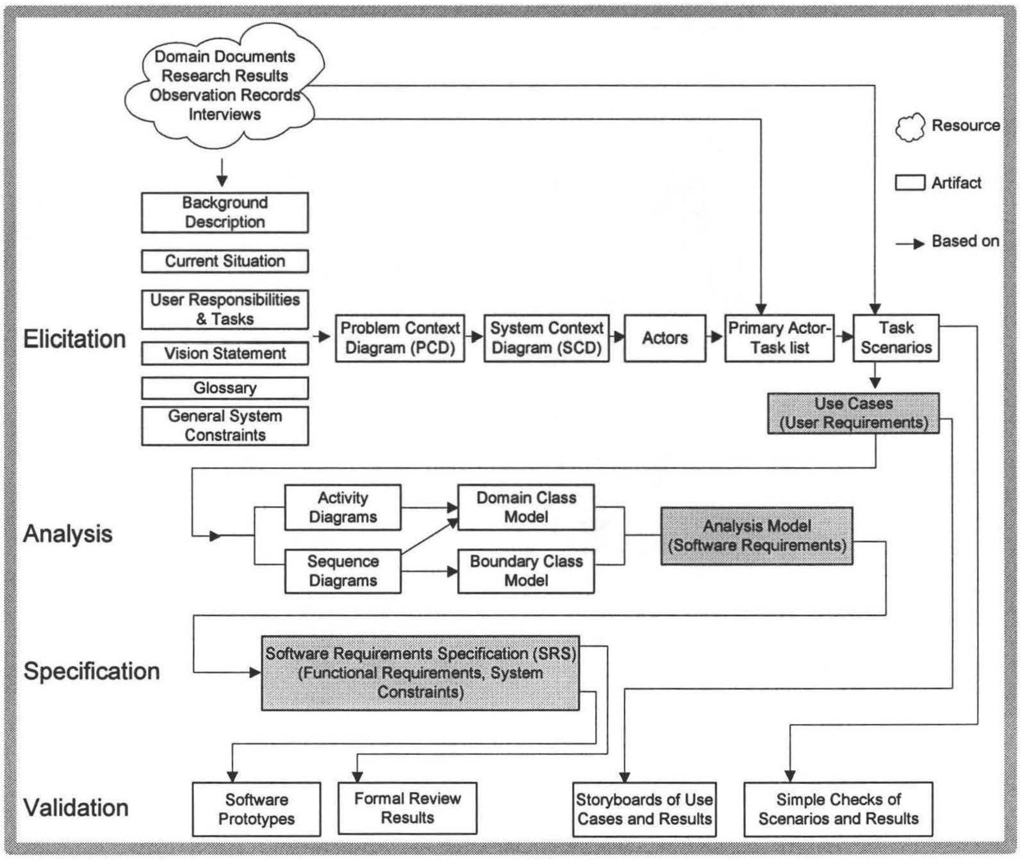
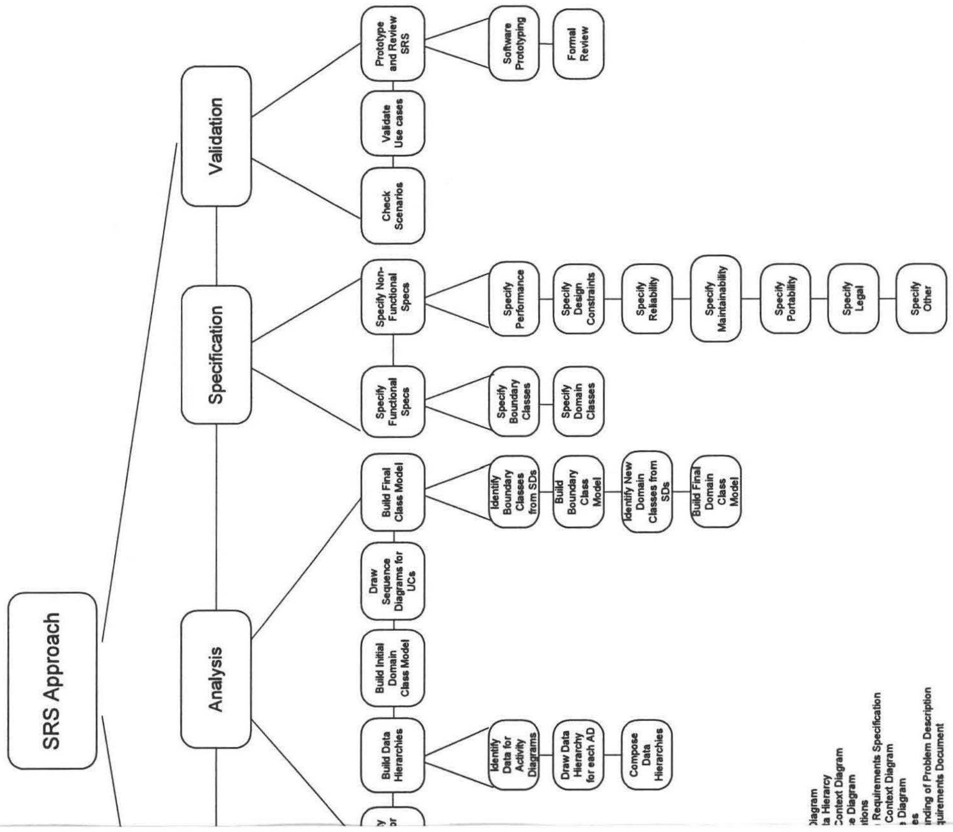


Figure 3.1: Artifacts View of the Approach

3.2 Requirements Elicitation

Software Requirement Elicitation is an essential process of Software Requirement Engineering. It is perhaps the most difficult, most critical, most error-prone, and most communication-intensive aspect of software development [86]. Eliciting requirements is about finding the real needs for the system [14]. We gather understanding of



Processes View of the Approach

the future system and captured in use cases. The users validate the use cases typically by reviewing the scenarios or by testing small prototypes.

Elicitation Process Steps:

1. **Gather understanding of the problem description.**
2. **Find the system boundary.**
3. **Identify actors.**
4. **Specify primary actor's tasks.**
5. **Specify use cases.**

problem descriptions by studying the user's environment and domain documentation, discovering the functionality of any existing systems, and interviewing users.

User-Centric

Most systems are designed to be used by people. This is especially true for user-centric software applications, in which the human/computer interface is extremely important. Each user type has some responsibilities for using the system. Correspondingly, the system must provide some services for a user type to fulfill its responsibilities. All such services provided by the system constitute the functionality of the system. So, we focus on studying what responsibilities each user type has and what behavior the system shall provide when each user type uses the system.

The results are specified in use cases (introduced in the next paragraph) as user requirements. Use cases identify the functionality of a system from the users' point of view.

Scenarios-Based

The customers and users are experts in their domain and have a general idea of what the system should do, but they often have little experience in software development. On the other hand, developers have experience in building systems, but often have little knowledge of the environment of the users.

Scenarios were designed to bridge this gap. A scenario is "a narrative description of what people do and experience as they try to make use of computer systems and applications" [18]. A scenario describes an example of system use in terms of a series of interactions between the user and the system. For many years, this technique has been used by analysts to help elicit requirements [56]. In 1992, Ivar Jacobson invented the *use-case* approach for object-oriented software engineering [49], which gave an informal definition of scenario-usage. A use case is an abstraction that describes a collection of scenarios for a primary actor to fulfill a goal or task using the system.

We elicit requirements by observing and interviewing users. We start by representing the user's current work process (work-flow) through as-is scenarios. After that we develop additional scenarios describing the functionality to be provided by the future system and captured in use cases. The users validate the use cases typically by reviewing the scenarios or by testing small prototypes.

Elicitation Process Steps:

1. **Gather understanding of the problem description.**
2. **Find the system boundary.**
3. **Identify actors.**
4. **Specify primary actor's tasks.**
5. **Specify use cases.**

Use cases are the outputs of the elicitation process as user requirements, and in a complete model, the use cases partition the functionality of the system and they may be properly organized according to functionality. Figure 3.3 indicates the input and output of the elicitation process.

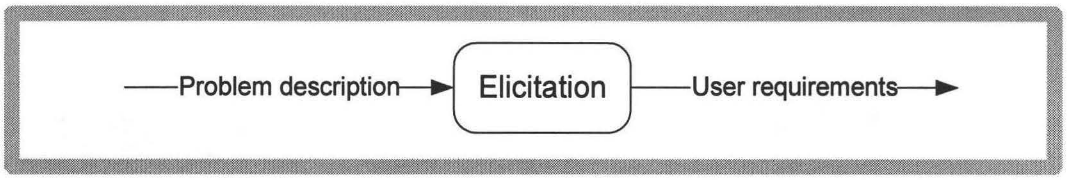


Figure 3.3: Elicitation Process

3.2.1 Gather Understanding of the Problem Description

Normally, we are not experts in the problem area for which our software systems provide solutions, so the first thing we must do is to become familiar with the problem area to understand the processes for the given field and to understand how our software should facilitate those processes.

During this step, considerable expansion of information and knowledge about the problem are collected, including all the constraints on the problem's solution. An understanding of problem description is an output of this step.

What Is a Problem?

Generally speaking, a problem is a state of difficulty that needs to be resolved or a question raised for consideration or solution. A useful definition of problem comes from Gause and Weinberg, "A problem is the difference between things as perceived and things as desired" [36].

What Is a Software Development Problem?

In software development, a software development problem refers to any problem that needs to be resolved by creating the software for a computer system that will serve some useful purpose in the world. Software development problems are about the real world where the system must have its effect.

What Is a Domain?

In dictionaries, domain has many definitions like "A particular environment", "A territory over which rule or control is exercised", "People in general; especially a distinctive group of people with some shared interest."

In their book Object Life Cycles, Sally Shlaer and Stephen Mellor define domain in a different way: "In building a typical large software system, the analyst generally has to deal with a number of distinctly different subject matters, or domains. Each domain can be thought of as a separate world inhabited by its own conceptual entities or objects" [57].

Our preference is Michael Jackson's definition of domain: "A particular part of the world that can be distinguished because it is conveniently considered as a whole, and can be considered - to some extent - separately from other parts of the world." [47]

What Is a Problem Domain?

A problem domain is a domain that is directly related to the problem. In a software development problem, the problem domain is what is given, while the system is what is to be built. The system provides a solution to the problem by interacting in some way with the problem domain.

A domain can be decomposed into sub domains, further, a sub domain can be decomposed into its sub domains and so on. A part of the problem domain is a problem sub domain, which can be a person, a system, a device, an organization, or a physical representation of some related information (a set of data).

In a traditional context diagram, we typically restrict our attention to what we call the environment of the system - all the problem sub domains that are directly connected with the system. So, a problem domain is more than the environment of the system to be built. It includes all the relevant parts of the problem.

In short, the environment just consists of something that physically surrounds the system, whereas the problem domain includes all the related parts of the world in which the customers are interested. This can include people, other systems, devices, company's products, buildings, intangible things like graphics images or timetables or employment payscales, and absolutely anything else that will interact with the system or furnish the subject matter of its computations [47]. To understand the problem, we need to explore the whole problem domain.

What Are Phenomena?

The Cambridge Advanced Learner's Dictionary defines phenomena as "something that exists and can be seen, felt, tasted, etc., especially something which is unusual or interesting". So, phenomena are any entities, relations, states or processes known through the senses rather than by intuition or reasoning. In other words, phenomena are what appear to exist when you observe the world or part of the world. The subject of study of phenomena is called *phenomenology*, which makes a contrast with *ontology*, which is about what really, truly, fundamentally, and objectively exists, independently of our perceptions and observations.

As software developers, to understand the problem, we need to capture all the existing phenomena of the problem and the desired phenomena of the system. We do not need to disclose the real essence of the phenomena - we deal with them as we experience them, as they appear to customers and users.

Meanwhile, rather than using the elaborate phenomenology, we want a simple phenomenology. So, we limit ourselves to three kinds of individuals (events, entities and values) and three kinds of relations (states, truths, and roles). An individual is something that can be named and reliably distinguished from other individuals. The distinguishability of individuals relies on our purpose of analyzing the problem. So, individuals may be any things we choose, as long as they can be distinguished one from another for our purpose. A relation is a set of associations among individuals. A relation consists of some number of tuples.

The various individuals and relations are defined below [48].

- **Event.** An *event* is an individual happening, taking place at some particular point in time. Each event is indivisible and instantaneous. It is a phenomenon located at a single point in space-time, which follows and is caused by some previous phenomenon.
- **Entity.** An *entity* is an individual that persists over time and can change its properties and states from one point in time to another, and it is perceived or known or inferred to have its own distinct existence.
- **Value.** A *value* is an intangible individual that exists outside time and space, and is not subject to change. We are interested in those values like numbers and characters, represented by symbols. A range is a pair of values.
- **State.** A *state* is a relation among individual entities and values; it can change over time. We often use state in place of tuple. We say a state holds (is true) or doesn't hold (is false).
- **Truth.** A *truth* is a relation among individuals that cannot possibly change over time. The related individuals are always values and the truth expresses a mathematical fact, such as *LengthOf ("ABCDE", 5).*
- **Role.** A *role* is a relation between an event and individuals that participate in it in a particular way.

What Is a User's Responsibility?

A user's responsibility is something a system user must do because of prior agreement. Each responsibility of a user type is a collection and summary of a user's tasks when using the system.

What Is a Task.

A task is a piece of work that needs to be done regularly [45]. It is a part of a set of actions which accomplish a job, problem or assignment.

What Is Included in the Understanding of Problem Description?

Understanding of Problem Description (UPD) are the abstract but explicit statements of the problem to be solved in a way that is familiar to people with experience in the problem domain. It describes the problem and the requirements at a high level. It includes the current situation, a vision statement, user responsibilities, system constraints, a glossary and the problem context diagram. Some part of the UPD are eventually transferred into SRS, such as the glossary.

Current Situation describes the current state of affairs. It describes how the responsibilities (tasks) of users supported by the new system are accomplished at the current time.

Vision Statement summarizes what the system is expected to accomplish. It should explain what the purpose of the system is and what the system should ultimately become.

User Responsibilities and Customer Authorities. Each user type has a set of responsibilities when it uses the system. To fulfill its responsibilities, it performs some tasks which are eventually represented as use cases. At the earlier requirements stage, each user type's responsibilities are identified through interviews, as well as the user tasks that are needed to fulfill each user's responsibility.

Customers' authorities are identified also. The customers of the system have limited authorities when they want to use the system. The system should be built so that customers cannot access functionality of the system that is beyond their authorities.

System Constraints, also known as non-functional requirements, are constraints the system must obey when the system performs its services (i.e. implements the functional requirements). Non-functional requirements involve considerations such as usability, reliability, performance, maintainability, implementation, interface, legal requirements, etc.

As in most cases, system constraints are associated with a particular user task, such as performance requirements (e.g. response time for an action), and they can be explored during the use task exploration. Some more generic constraints like reliability (e.g. mean time between failures), political and legal constraints (e.g. certification) can be examined in earlier stage. Table 3.1 (revised from [14]) is used as a general guide to find system constraints.

Category	Guide Questions
Usability	What is the level of expertise of the user? What user interfaces are familiar to the user?
Reliability (including robustness, safety and security)	How available and robust should the system be? How should the system handle the exceptions? What encryption levels are needed over internet? How much data can the system lose?
Performance	How responsive should the system be? Are any user tasks time critical? How many concurrent users should it support?
Maintainability	Who maintains the system? What efforts needed to maintain or enhance the system?
Portability	Does the system have the ability to easily move to different hardware platforms, operating systems, database management systems, network protocols?
Implementation	Are there constraints on the hardware system? Are there constraints on the programming language? Are there constraints imposed by the maintenance team?
Interface	Should the system interact with any other existing system? How are data exported/imported into the system?
Operation	Who manages the running system?
Packaging	Who installs the system? Are there time constraints on the installation?
Legal	How should the system be licensed?

Table 3.1: Guidance for Finding System Constraints

Glossary (Data dictionary) contains both a domain glossary and any other terms or abbreviations used in the specification. A glossary de-mystifies the jargon for anyone examining the document. Each entry in the glossary defines a term. Terms with the form of concatenation of several words denote that they are used by the system, such as an MRIImageSet. If a term is a data structure, the including data items are enclosed in parentheses and are separated by commas, such as MRIImage-Set(Name, SliceNumber, MRIImages). We use plural to denote a collection and [*i*] to denote the i^{th} element of a collection, such as MRIImages[1].

The analysts and developers should keep the glossary up to date as the requirements specification evolves.

Problem Context Diagram is a diagram that structures the world into the system domain, and the problem domain (which includes problem sub domains), and shows how they are connected. It is not limited to the parts of the world that are directly connected to the system. A problem context diagram shows what the real world will be when the system is running.

3.2.1.1 Study of Documents and Existing Software Systems

Studying documentation on the problem domain should be done as early as possible, and the resources may come from:

- A good introductory book suggested by customers.
- Organization documents including work procedures, job descriptions, policy manuals and business plans.
- Domain journals and reference books.
- Documents that describe current or competing systems.

3.2.1.2 Further Analysis of the Problem Domain

Once we have some basic knowledge in the problem area, we can begin typical tasks such as the following:

- Build a domain glossary. To facilitate clear communication, we should capture the significant terms in a glossary.
- Understand the underlying problem goals.
- Identify different types of users and corresponding representatives, and characteristics.
- Identify decision makers for the project.

Topic	Questions	When	Who	Times
Vision	Vision for the system Alternative minimally acceptable solution Other source of information System constraints	Start of project	A representative from each user type	1..n
User responsibilities and customer authorities	Responsibilities of each user type Authorities of customers Tasks needed to achieve each users type's responsibilities	Start of project	A representative from each user type	1..n
Task workshop	Work-flows (Scenarios) of each task (how they do now, how they desire with the new system)	After capture primary actor-task list	Representatives from related user types	1..n
Use case validation	Does each use case correctly capture corresponding user task?	After initial sketch of use cases.	Representatives from related user types	1..n

Table 3.2: Typical Structured Meetings

3.2.1.3 Interviewing Users Effectively

Elicitation can succeed only through an effective customer (user)-analyst (developer) partnership. As stated in Chapter 1, we advocate the joint requirements development (JRD) method. However, various interview forms also are suggested at the beginning of a project. After we have mastered some domain knowledge, we can arrange interviews with the customers and users, because customers and users are the best source of information about the problem domain. Interviews are the primary technique of fact finding and information gathering. Specifically, we may use the following approach:

- Structured meeting but open-ended questions recommended.
- Core meetings/requirements workshops suggested. Table 3.2 lists some typical meetings.
- Specific details for any issue: five w's - what, who, when, where and why.
- Pre-determined open-ended questions in the first meeting: What do you believe the application must do to be effective? What are the most important aspects of the problem domain? What are the processes now?
- Review with interviewees the items discussed.

Elicitation is a highly collaborative process, not just a recording of what customers and users say they need. We must probe beneath the surface of the requirements the customers present to understand their true needs. So open-ended questions are recommended to help us better understand the user's current scenarios and to see how the new system could serve.

User tasks and corresponding scenarios are a very important part in the elicitation process. Suggested questions in this part could be "What are your responsibilities for using the system? To fulfil your responsibilities, what tasks do you need to perform with the system?" Also, we should never neglect variations in the user tasks that might be encountered or ways that other possible users might need to use the system. Further, inquire about exceptions: what could prevent the user from successfully completing a task? How does the user think the system should respond to error conditions? Last but not least, discuss with the users the interactions and dialogues between the users and the system that they hope to complete each task. We will discuss the detailed process of task specification in later sections.

3.2.1.4 Observation of Users at Work

In most circumstances, it is difficult to obtain complete information about the problem description through interviews and the methods above. The customers may just convey fragmentary knowledge of a complete working process. In such a case, observation may be an effective fact-finding technique.

Observation has generally three forms [55]: passive observation, active observation and explanatory observation. In a passive observation, the analyst observes the user's activities without interruption, whereas, in an active observation, the analyst directly takes part in the users' team. In an explanatory observation, the user explains his or her activities to the analyst while doing the job.

As different people tend to behave differently even when following formal rules and procedures, we ought to abstract and generalize their activities and ensure that the requirements captured apply to the user type as a whole.

3.2.1.5 Capturing the Understanding of Problem Description

In a project, when the above steps are carried out and initial information is gathered, we can begin to specify our understanding of the problem description. The following is partial initial information about a *photodynamic therapy (PDT) treatment planning software* problem, which we use as an example throughout our approach.

Vision of the system: Treatment planning software is required for simulating treatment of cancer patients (e.g. prostate cancel patients) and for producing treatment plans for them. For each patient, the baseline MRI images will be loaded into the system and the important structures will be defined (e.g. prostate, rectum and urethra for prostate patients). Then, a virtual array of treatment devices (e.g. cylindrically diffusing optical fibers) is added to the virtual target volume (e.g. prostate). Once a set of treatment parameters (e.g. device numbers, energy etc.) is defined, the light dose distribution both inside the target volume and in its surroundings can be calculated by the software and the calculated results can also be visualized by superimposing the treatment effect onto the MRI images. Being iteratively changed, a set of treatment parameters is determined until an acceptable balance between efficacy and safety is achieved.

In this example, we have the hospital as the customer and a radiologist as a user type. Tables 3.3 and 3.4 demonstrate their authorities and responsibilities.

Customer	Authorities
Hospital	Run the software for treatment planning for patients

Table 3.3: Customer's Authorities

User	Responsibilities	Tasks
Radiologist	Ensure the correct target definition	Enter PatientInfo & TargetInfo Link MRIImageSet Define Target

Table 3.4: User's Responsibilities and Tasks

3.2.1.6 Draw a Problem Context Diagram

As mentioned above, a problem context diagram (PCD) is a part of the UPD. We separate this subsection because it includes additional information that needs to be clarified.

Most people directly draw a system context diagram at the very beginning of the requirements stage. How do they obtain it? Probably it is a result of rules of thumb; they focus on only the system itself and its surroundings. How can the surroundings be found? They may say by exploring the requirements. They never show the process of how to draw a system context diagram - we will, with the usage of the problem context diagram.

Another question is, how about other relevant parts of the problem that are not directly connected with the system? For example, in the treatment planning software, the patients are not directly connected with the system. Do we need to care about them?

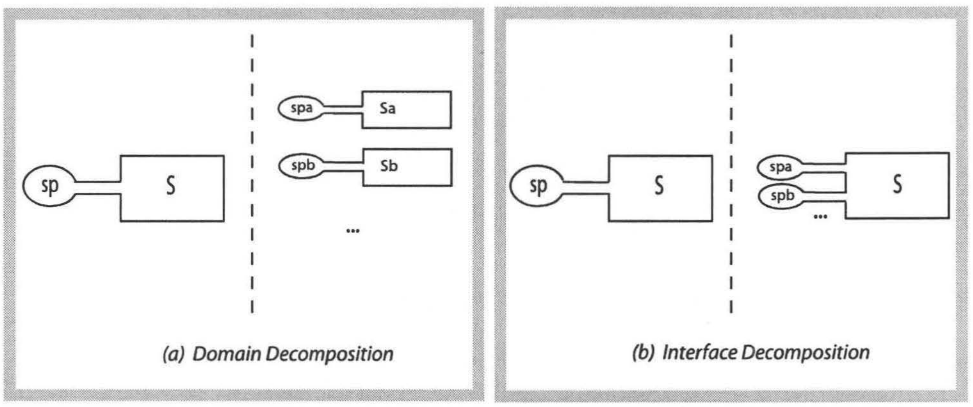
The answer is "yes". Showing all the relevant parts of a problem indicates that we have really understood the problem. At the requirements phase, our objective is to understand the problem, that is, to identify the problem, and then we analyze it and try to get a definition of a solution. Those parts not directly connected with the system also contain the necessary information to solve the problem, e.g., the patient name shall be related to a specific treatment plan, and thus they should not be omitted. So, at the early requirements stage we put emphasis on identifying the problem completely, and capturing it visually in a problem context diagram to show our understanding.

When we set about analyzing and structuring a problem, it is fundamental to determine what it is about - that is, where the problem is located, and what parts of the world it concerns. After we have done some research, studies, and initial interviews with users and captured the vision statement and users responsibilities and tasks, we record what we find from various descriptions of the problem into a problem context diagram. Based on the knowledge and information already developed, an analyst examines the various parts in the problem domain. These parts (called problem sub domains) form the context into which the planned system must fit. Then the analyst determines how the system will fit into this context. The result of this is a context diagram showing the vision of the problem context with the system installed in it. So, a problem context diagram locates the problem in the physical world. It identifies all the relevant parts of the world and abstracts the understanding of the problem.

It is an iterative process to draw a problem context diagram. As the elicitation activities go forward, the sub domains may be added, composed, or decomposed to reach an understandable level.

Decomposition of a (Sub) Domain

A domain can be divided into sub domains; further a sub domain can be divided into its own sub domains, and so on. The decomposition is based on our desire for abstraction. Figure 3.4 demonstrates the basic rules of domain decomposition. Any decomposition can be carried out by these two kinds of decompositions, or a combination of them.



S	*Super domain*
sp	*Interface of super domain*
Sa, Sb,...	*Sub domains, where S = Sa U Sb ...*
spa, spb,...	*Sub Interfaces, where sp = spa U spb ...*

Figure 3.4: Domain Decomposition Rules

In Figure 3.4 (a), a super domain S is decomposed into a finite number of sub domains Sa, Sb, ..., and its interface *sp* (a set of shared phenomena: events, states and values) with other (sub) domain(s) is decomposed into interfaces of its sub domains, where $S = Sa \bigcup Sb...$, $sp = spa \bigcup spb \bigcup ...$. The sub interfaces must be mutually disjoint; in case that they are not, further interface decompositions should be performed according to the rules in Figure 3.4 (b), and some sub interfaces are shared by two or more sub domains. Also, internal interfaces between sub domains may be derived, because the super domain S itself is something that, as a whole, can be separated from others, and we expect that there may exist some cohesion among its sub domains. For example, a company super domain may be decomposed into a sales department sub domain, accounting department sub domain and a supply department sub domain etc., and the sales department has an inner connection with the accounting department.

In Figure 3.4 (b), the interface sp of a super domain S is decomposed into a finite number of sub interfaces spa, spb, ..., where $sp = spa \bigcup spb \bigcup ...,\ and\ spa \bigcap spb \bigcap ... = \varnothing$. For example, when the company domain interacts with the outside world, its interface may be divided into customer contact interface, supplier interface, bank interface etc.

Figure 3.5 shows the decomposition of a company domain, in which the customer contact interface belongs to the sales department. Further, the customer contact interface of the sales department can be divided into a web interface and a telephone interface.

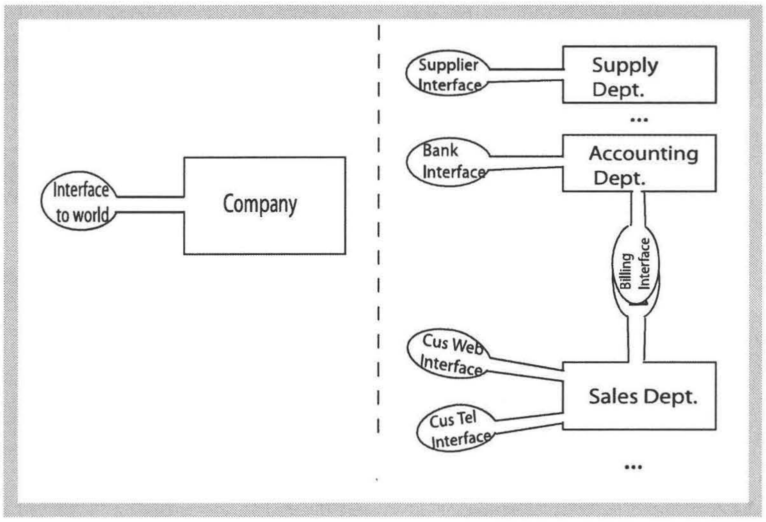


Figure 3.5: Domain Decomposition: a Company

To simplify the notation, we will not give extra notation for concurrent shared phenomena of an interface. We focus on the existence of shared phenomena. However, plural is used in case there is more than one instance of sub domains at any particular point of time in the problem.

Composition of Sub Domains

The composition of sub domains follows exactly the reverse rules of the decomposition, and likewise, the definitions of composition of sub domains are similar as well. We define a super domain S as a composition of sub domains Sa, Sb, ... such that $S = Sa \bigcup Sb \bigcup, \ldots \, and \; sp = spa \bigcup spb \bigcup \ldots$.

As needed, a problem context diagram can have levels through (sub) domain-decomposition and composition. The higher level problem context diagram shows the overview of the lower level problem context diagrams. Especially when a problem has many sub domains, the abstract level of the problem context diagram will be helpful in understanding the whole picture of the problem.

Top Level Problem Context Diagram.

The problem context diagram contains both the system and the problem domain, and shows how they are connected: that is, their interface(s). Figure 3.6 is a top level problem context diagram. The system and problem domain communicate or interact

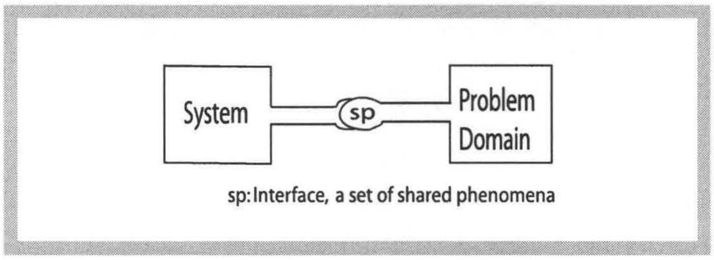


Figure 3.6: Top Level Problem Context Diagram

only at their interfaces. Interfaces are not dataflow or messages. At this early stage, we discuss the real world problem with our customers. We do not want to assume that all communications are of the dataflow kind, we will think in more general terms - that inter-actions between (sub) domains are shared phenomena, which include shared events, shared states and shared values.

Only through the communication or interaction with the problem domain, namely, sensing and affecting it, can the system fulfill its purpose. In Figure 3.6, the overlapping part of the system and problem domain is the hand-shaking area *sp*, which is the interface between the system and the problem domain. This interface is where the system and the problem domain meet and interact. It is a set of shared phenomena in which both the system and the problem domain participate.

First-Decomposition Level Problem Context Diagram.

In a First-Decomposition Level problem context diagram, the problem domain is decomposed into problem sub domains. In most cases, this level is detailed enough for us to understand the problem, otherwise it can be further decomposed. Each problem sub domain will have its own context diagram accordingly, which is a fragment of the whole. We will omit "First-Decomposition Level" from here on, because this level is our primary interest. When we talk about a problem context diagram, we always mean this level.

A problem context diagram captures all the relevant parts of the world and their connections and shows the problem world as it will be when the system is in operation. It structures the world into a system domain, problem sub domains, and shows the interfaces between them: that is, how the system domain is connected to problem sub domains and how problem sub domains are connected to each other. Figure 3.7 shows what a problem context diagram looks like. The union of interfaces between the system and the problem sub domains constitutes the interface between the system and the whole problem domain, *sp* in the top level problem context diagram. In Figure 3.8, a further-decomposition level problem context diagram is illustrated, where the sub domain B in Figure 3.7 is decomposed to produce its own context diagram as a fragment of the whole problem context diagram. Moreover, for the UHN treatment planning software problem example, a problem context diagram is illustrated in Figure 3.9.

Definitions inside the Problem Context Diagram

In the following few paragraphs, we will clarify some definitions that are based on the original concepts of Michael Jackson [47, 48].

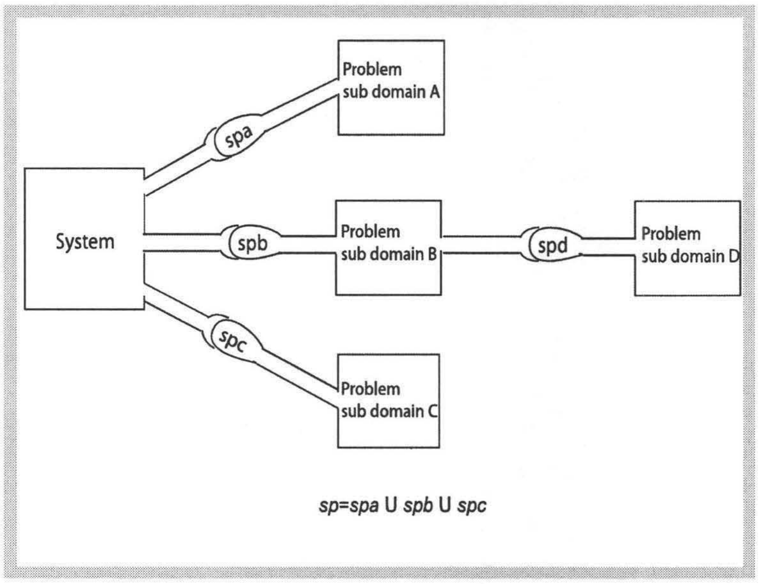


Figure 3.7: First-Decomposition Level Problem Context Diagram

System domain is the system under discussion. It consists of the software to be developed, and underlying general-purpose hardware and operating system.
Problem sub domains consist of all parts of the world where the problem is related at a fairly abstract level. Each part of the problem domain is a problem sub domain. A problem sub domain can be people, a system, a device, an organization, or a physical representation of some information (a set of data). There are two kinds of problem sub domains:

- A Data Sub Domain ("designed" sub domain as denoted by M. Jackson) is the physical representation of some information, for example, on a magnetic stripe card, or on a floppy disk or on a hard disk, or even on a screen or in printed output. You are free to design and specify its data structure and, to some extent, its data content during the software design stage which follows after the requirements stage.

- An Entity Sub Domain ("given" sub domain as denoted by M. Jackson) is a problem sub domain whose properties are given, that is, you are not free to design the domain. In some of them you can affect their behavior or state by

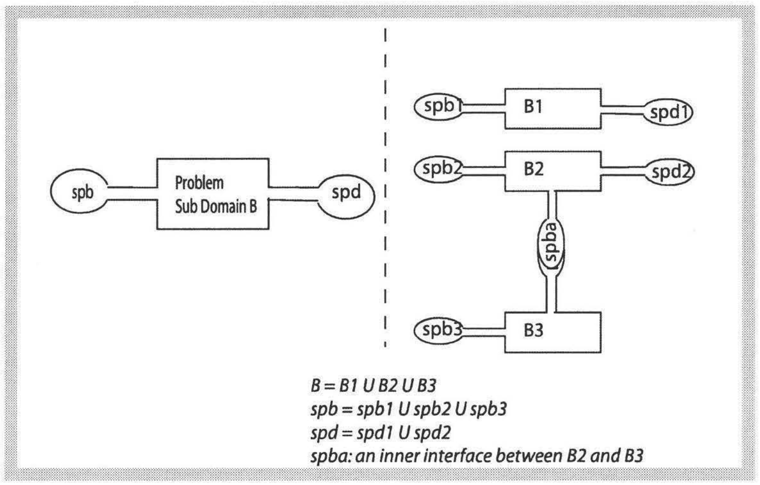


Figure 3.8: Further-Decomposition Level Problem Context Diagram

designing the system appropriately, e.g. the printer; some of the others cannot be affected by the system (e.g. patients).

All the problem sub domains in the problem context diagram are physical, they exist (e.g. patients) or will exist (e.g. treatment plan report). They identify the parts of the world in which the customer will check for observable effects. Problem sub domains are communicable or operable. Showing a sub domain as a data sub domain means that you will have the responsibility for doing the design work and the freedom that comes from being able to make design decisions in the later software development stage. For an entity sub domain you will not have that freedom, and your responsibility will just be to investigate and describe its properties and behavior, rather than developing a design.

An interface is a set of shared phenomena. The system domain and problem sub domains are physical, and the interfaces between them are physical. Interface is an area where (sub) domains connect and communicate. It is a place where (sub) domains partially overlap, so that the phenomena in the interface are shared phenomena - they exist in both of the overlapping (sub) domains. In a shared event, both of the sharing (sub) domains participate, but only one of them can cause it. In a shared state or value, both of the sharing participants can see the state or value, but only

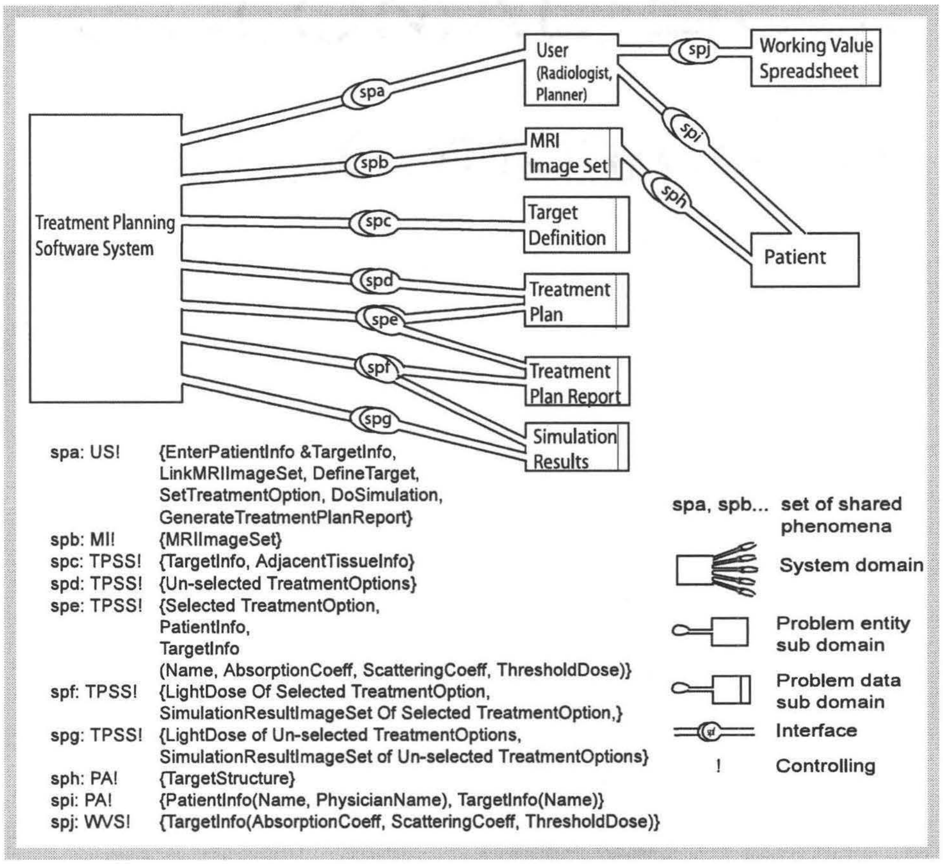

User (Radiologist, Planner)
Working Value Spreadsheet
MRI Image Set
Treatment Planning Software System
Target Definition
Patient
Treatment Plan
Treatment Plan Report
Simulation Results
spa
spb
spc
spd
spe
spf
spg
sph
spi
spj
spa: US! {EnterPatientInfo &TargetInfo, LinkMRIImageSet, DefineTarget, SetTreatmentOption, DoSimulation, GenerateTreatmentPlanReport}
spb: MI! {MRIImageSet}
spc: TPSS! {TargetInfo, AdjacentTissueInfo}
spd: TPSS! {Un-selected TreatmentOptions}
spe: TPSS! {Selected TreatmentOption, PatientInfo, TargetInfo (Name, AbsorptionCoeff, ScatteringCoeff, ThresholdDose)}
spf: TPSS! {LightDose Of Selected TreatmentOption, SimulationResultImageSet Of Selected TreatmentOption,}
spg: TPSS! {LightDose of Un-selected TreatmentOptions, SimulationResultImageSet of Un-selected TreatmentOptions}
sph: PA! {TargetStructure}
spi: PA! {PatientInfo(Name, PhysicianName), TargetInfo(Name)}
spj: WVS! {TargetInfo(AbsorptionCoeff, ScatteringCoeff, ThresholdDose)}
spa, spb... set of shared phenomena
System domain
Problem entity sub domain
Problem data sub domain
Interface
! Controlling

Figure 3.9: Treatment Planning Software Problem Context Diagram

one of them can change or determine it.

At this early stage of software development, we use shared phenomena concepts rather than input/output in that we try to use the natural or domain language to describe the connections (interactions) among the parts of the problem world. Moreover, not all the phenomena of the problem domain are data flows, but we need to capture all of them to understand the problem.

Non Directly-Connected Problem Sub Domains

As we discussed at the beginning of this subsection, a problem context diagram also should contain the problem sub domains that are not directly connected with the system, such as patient in Figure 3.9. We call them NDC (Non Directly-Connected) problem sub domains. Similarly, those directly connected with the system are called DC problem sub domains. We also mentioned that NDC sub domains are necessary parts to solve the problem (the patient name example). Usually, NDC sub domains connect with DC sub domains (Connections between NDC sub domains are possible).

In a problem context diagram, the shared phenomena are actually abstractions of behavior and information of related (sub) domains. The shared phenomena between the NDC sub domains and DC sub domains are physically transformed into the system through the DC sub domains. For example, in Figure 3.9, the *Name* of Patient is transformed into input data of the system through the User's *Enter PatientInfo & TargetInfo* event. All the shared phenomena in a problem context diagram are necessary to solve the problem. They are eventually abstracted in the system either by functions or by data.

So, the NDC problem sub domains are also important parts to understand a problem. They contain the behavior and data that the system should be aware of to solve the problem, which are transformed by their connected DC sub domains and abstracted by the system.

Problem Domain Decomposition Rules

To draw a problem context diagram, the first step is to structure and separate the problem domain into a number of sub domains: a number of distinctly different subject matters - that is, the internal properties and behavior (phenomena) of each sub domain must be largely independent [47]. To some extent this decomposition can be done intuitively, dividing the problem domain along obvious lines suggested by the problem and the context. We give guidance for the decomposition as follows:

- A little theory [48]: The principle of problem domain relevance: everything that is relevant to the requirements must appear in some part of the problem domain.
- A set of data that can be talked about and grouped together are composed into a Data Sub Domain (e.g. in an online shop problem, all the goods selected by a customer can be composed into a data sub domain), the set of data are the phenomena shared with the system domain or other sub domains.

- Each entity type is a potential Entity Sub Domain (e.g. user types, devices). Its interacting phenomena (events, states and values) with the system domain and other sub domains are the shared phenomena of its interfaces.
- Entity types that have common interacting behavior are composed into one Entity Sub Domain (e.g. two user types that share similar tasks with the system), their interacting phenomena with the system domain and other sub domains are the shared phenomena of the interfaces of the entity sub domain.
- A user task is abstracted to be a shared event between the user type and the system.

Problem Context Diagram Drawing Rules

The following rules apply to constructing a problem context diagram.

- System domain is included.
- Each problem sub domain is included.
- Where (sub) domains interact, an interface is recorded. The interaction takes the form of shared phenomena. Each interface is a set of shared phenomena and is given a unique identifier beginning with "*sp*".
- For shared phenomena of each interface, the controlling (sub) domain is identified with its acronym following a "!".
- If necessary, a sub domain may further be decomposed.

Boundary of the Problem Context Diagram

The problem context diagram shows all the domains and interfaces that we must take into account. It locates the problem within quite an exact boundary. Something out of it means that it will play no part in our work, it won't affect the outcome.

Identifying the customer's authority and user's responsibility can avoid broadening the problem too far and narrowing it too much. Customers have certain limited authority and users have necessary responsibilities. Use those limits and necessities as the touchstone when we are in doubt about the location and scope of the problem. User's responsibilities place a lower bound on the sub domains that must appear in the context diagram (some sub domains are affected and must appear). Customer's authority limits the scope of what the software system may legitimately be designed to do and on what assumption: it places an upper bound on the domains that may appear in the problem context and be affected by the software system.

Ask ourselves: must this requirement be in scope? Can it be in scope? And what are the consequences for the context diagram?

3.2.2 Find the System Boundary

To seek and decide where the system boundary is and further for the purpose of identifying user requirements (for finding actors of use cases), a traditional context diagram is an effective tool, which is originally employed as the top level of abstraction in a data flow diagram developed according to principles of structured analysis [63, 71]. We derive this context diagram from the problem context diagram and name it the system context diagram.

3.2.2.1 System Boundary

What Is a Boundary?

The line or relatively narrow space that marks the outer limit of something. [45]

What Is the System Boundary?

The system boundary is the interface between a system and the environment, where the environment consists of entities that are directly connected with the system - such as other systems, people and devices etc. - that expect some services from the system or provide services to the system. In other words, the system boundary is the interface where input/output data flows between the environment and the system.

The system boundary is different from the problem boundary. The problem boundary pays attention to what should or should not be included in the problem context, whereas the system boundary focuses on the data flow between the system and entities in the problem domain. The activity of finding the system boundary is the process of identifying the problem domain entities that are directly connected to the system.

3.2.2.2 System Context Diagram

Data flow diagrams (DFDs) have been used for many years prior to the advent of computers [28]. DFDs show the flow of data through a system. A popular notation of DFD denotes that a DFD is composed of data on the move, shown as a named arrow; transformations of data into other data, shown as named bubbles; sources and destinations of data, shown as named rectangles called terminators; and data in static storage (i.e., data bases), shown as two parallel lines.

A system context diagram is a high, abstract level DFD, with only one bubble-the system, showing all system terminators and external inputs and outputs. It consists of the system, its environment, and data flows between the environment and the system. It explicitly illustrates the boundary by showing the connections between the system and the outside world.

3.2.2.3 Deriving the System Context Diagram

Unlike in a problem context diagram, relevant physical parts of the world and their shared phenomena could be directly and easily identified as problem sub domains, external entities surrounding the system sometimes are implicit. Based on some rules, we can easily derive a system context diagram from a problem context diagram.

From definitions of the problem context diagram and the system context diagram, we note the assumptions below.

Assumption 1. *In a problem context diagram, there must exist shared events or states or values between the system and its directly connecting problem sub domains, which also implies that there exists interactions between them.*

Assumption 2. *In a problem context diagram, for each sharing event only one of the sharing participants can cause it; for a shared state only one of the sharing participants can change it; moreover, for a shared value, only one of the sharing participants can determine it.*

Assumption 3. *In a problem context diagram, interfaces are symmetrical in the sense that each (sub) domain may control some of the shared phenomena of the in-*

terface.
Assumption 4. *In a system context diagram, all terminators are directly connected to the system.*
Assumption 5. *In a system context diagram, there must exist input or output or both data flows between terminators and the system, which also stand for the interaction between them.*

Derive a System Context Diagram from the Problem Context Diagram

According to the assumptions above, we can deduce the rules for the derivation.

- The system domain becomes the system bubble.
- Problem sub domains not directly connected with the system domain are eliminated, but their effects on the problem domain are encapsulated by other sub domains or the system domain as discussed in page 37.
- Problem sub domains directly connected with the system become terminators, their shared phenomena with the system are converted to data flows.
- Shared phenomena (events, states, or values) controlled by the system are converted to output data flows, whereas those controlled by problem sub domains are converted to input data flows.

According to these rules, Figure 3.10 shows a derived system context diagram of the treatment planning problem from Figure 3.9.

3.2.3 Identify Actors

In our approach, we capture the user requirements in use cases. As stated, a use case describes a sequence(s) of interactions between the system and an external "actor" that results in the actor accomplishing a task. In order to identify use cases, we need to identify actors firstly from the system context diagram.

A *user* is a person who uses the system. Normally, a system has many types of users. Each type of user is represented as an actor.

An *actor* is anyone or anything with behavior [22], or anything that needs to exchange information with the system [49]. Basically, actors represent external

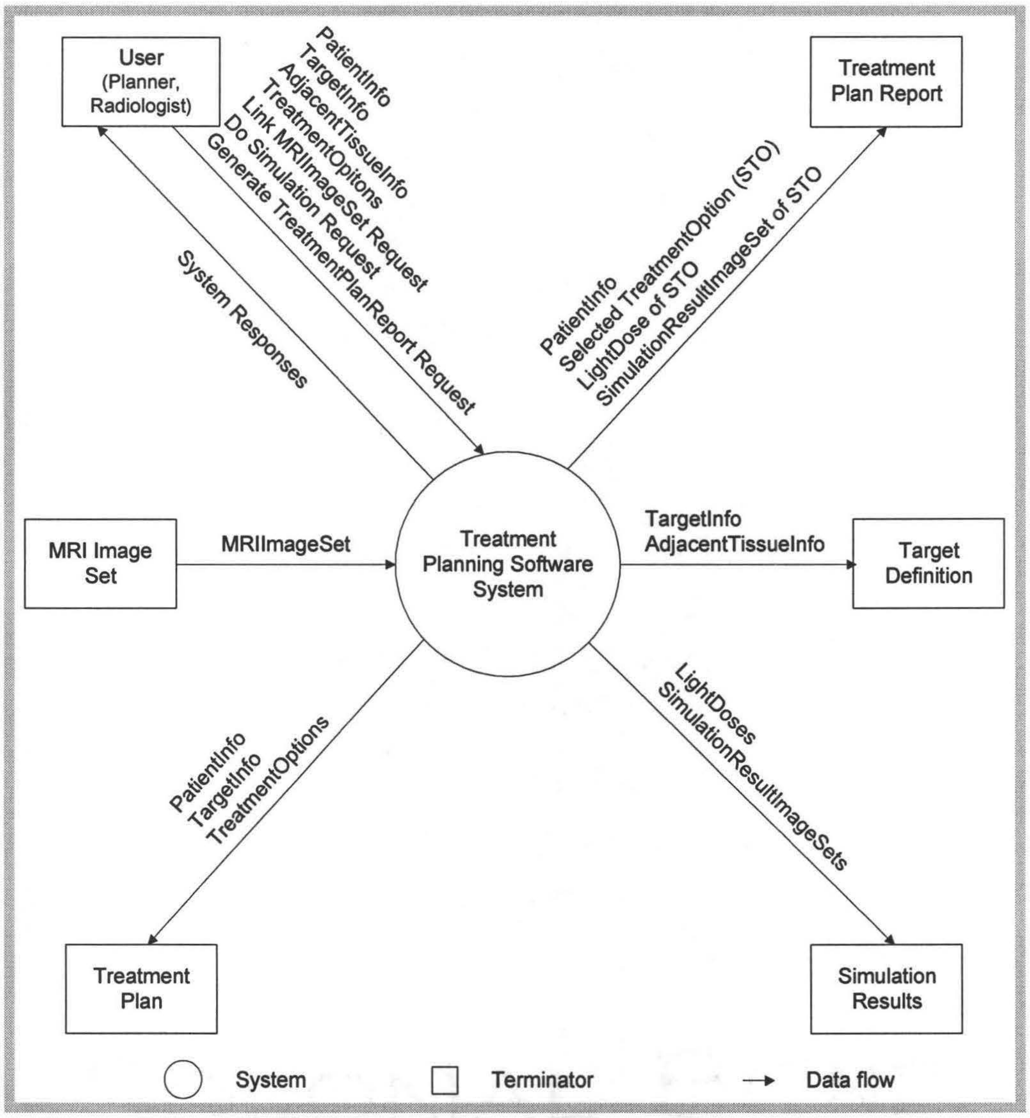


Figure 3.10: A System Context Diagram

entities that interact with the system. The actor is a user type or a category. It can be human, organizations, devices and an external system [14]. However, an actor could be an internal system entity such as a timer, e.g., when a system needs to print the system log at midnight automatically. Also, in our conception, we give the *actor* wider semantics which we will clarify in later sections.

An actor is a role abstraction. If it is a user type, an actor represents a certain role that a user can play, in other words, an actor is a user class and users are instances of that actor.

A *primary actor* is an actor who is going to use the system directly [49] or the one who initiates an interaction with the system for some purpose. The primary actors will govern the necessity of the main functionality of the system to be built.

A *supporting actor (secondary actor)* is an actor that provides a service to the system under development [22], e.g. printer, web service etc. It exists because of the primary actor using the system. Supporting actors are helpful for identifying the external interfaces the system will use and the protocols that cross those interfaces.

3.2.3.1 Rules for Identifying Actors

According to the rules for deriving the system context diagram, terminators of a system context diagram describe all the things that interact directly with the system. Therefore, we can identify actors from the system context diagram, for example, a certain user type appears in a problem context diagram as a sub domain, and then it is converted to be a terminator in a system context diagram, and then it can be identified as an actor. We have the following rules for identifying actors:

- All terminators in the system context diagram are actors.
- Each terminator that contains the user type(s) is a primary actor.

In the treatment planning software problem, from Figure 3.10, we can identify the User (including Radiologist and Planner), MRI Image set, Target Definition, Treatment Plan, Treatment Plan Report and Simulation Results as actors, and the User is a primary actor.

3.2.3.2 Extra Questions to Check Completeness of Actors

While we should check the completeness of the problem context diagram to ensure that we capture all the parts of the problem domain and based on that, theoretically we can get all the actors from the derived system context diagram, we still have chances to examine if we have identified all the actors at this stage. We may ask the

Actor	Profile: Background and Skills
User (Radiologist)	A physician specializing in diagnostic techniques for viewing internal organs and tissues without surgery. Radiological methods include X-ray, MRI, computed tomography (CT), scan, ultrasound, angiography, and nuclear isotopes.

Table 3.5: Profiles of Actors

following questions: Which user groups execute the system's main functions? Which user groups are supported by the system to perform their work? Which user groups perform secondary functions, such as maintenance and administration? With what external hardware or software system will the system interact?

3.2.3.3 Profile of the Actors

The background and skills of actors are one of the important sources based on which the designers will design the system behavior and user interfaces. An actor/profile table can be used to list the characteristics of each actor. Table 3.5 shows an example of the description of an actor from the case study.

3.2.4 Specify Primary Actor's Tasks

Our purpose of the elicitation process is to capture user requirements into use cases. Generally, there are several approaches to identify use cases [51, 54]:

- Identify actors and their roles first, then identify the business processes in which each participates to reveal use cases.
- Identify the external events to which the system must respond, then relate these events to participating actors and specific use cases.
- Express business processes or daily activities in terms of specific scenarios, derive use cases from the scenarios, and identify the actors involved in each use case.
- Derive likely use cases from existing functional requirement statements. If any requirements don't align with a use case, consider whether you really need them.

A use case is a complete course of events in the system, seen from a user's perspective [49]. In other words, a use case represents a complete unit of functionality for a primary actor to use the system.

Apparently, interactive systems are developed for users (abstracted by user types) to use. From the user's perspective, each user type has its own tasks (goals) in order to achieve its responsibilities when using the system. So, if we intuitively derive a task-oriented description for each user type and get a user type-task list which shows all the user's tasks that the system can support, we will get all the system's functional content. We can thus identify use cases by a more direct way - performing each task will become a use case. By interviewing users or examining available documentation, we can develop all the scenarios for performing each task, and capture them in use cases.

A use case involves only one primary actor role, and in most cases, it is a user type that triggers the use case, and the name of that actor is the name of the user type. However, if two user types can perform the same task (the same use case), they will play the same specific primary actor role in terms of this task. So, a specific primary actor role of a specific use case can be played by different user types. In this situation we can make a role name of this specific primary actor. So, what we really need is the primary actor-task list rather than the user type-task list, which contains all non-duplicated tasks the system should support.

By going through all the primary actors and defining all the tasks they need to do and will be able to do with the system, we will define the complete functionality of the system.

3.2.4.1 User's Responsibilities and Tasks

User's Task. A user's task is a task that a user must perform to fulfill one of his or her responsibilities by using the system. It is an elementary work process. It is the goal the user has in trying to get work done in using the system. Often a transaction in a transaction system corresponds to a user's task, such as withdrawing cash in an ATM system.

Action. An action is the behavior that triggers an interaction between the actor and the system [21] or behavior that triggers an internal state change of the system.

Actions can be given from both the actors and the system. There are two types of actions: flow of actions and primitive action. A primitive action is an action that triggers an atomic interaction between the actor and the system (e.g. a user enters the password which triggers an interaction), or an atomic internal state change of the system (e.g. system validates the password which triggers a validation running). A flow of actions is composed of several actions (e.g. a user requests to save a file). A flow of actions can have any one of the following semantics: sequence, alternative, repetition, and concurrency [70]. Figure 3.11 shows the composition model of the action.

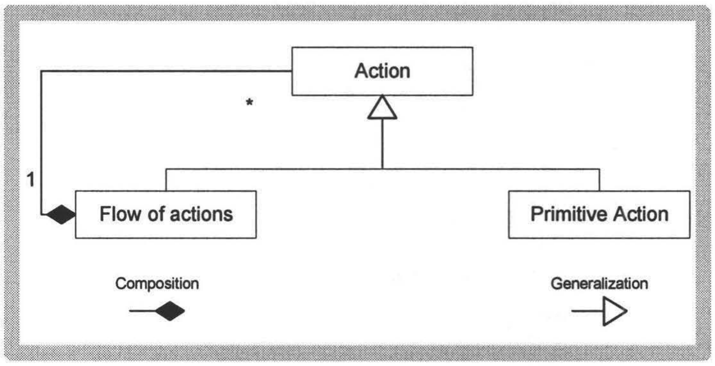


Figure 3.11: Action Composition Model

Formally, an action is a mathematical relation over objects, where objects are things of interest which can be referenced in requirements. Action applications define the state transitions. Each action has a precondition and postcondition, where precondition and postcondition represent certain states that must exist before and after the action.

Sub Task. A sub task is a sub action flow of a task. It is abstracted to be an action step in a scenario of a task. Often a reused section of a scenario could be rolled up into a sub task, like print a file. A sub task can have its own sub tasks.

Normal Scenario. A normal scenario is also called a basic scenario, a normal course, a normal flow, a main course, and is the normal sequence of actions to accomplish the intended task of the system.

Alternative Scenario. Any variation conditions and exception conditions in the normal scenario lead to various action steps to perform a task, they are alternative scenarios. So, an alternative scenario of a task can be successful, or failed.

Activity. An Activity is a process that users carry out to achieve a responsibility or perform a task or execute an action.

3.2.4.2 Carry Out a User's Responsibility

As stated previously, the users' objectives for using the system are to accomplish their responsibilities when they interact with the system. So, the complete set of functionalities which are needed for accomplishment of all users' responsibilities constitute the functionality of the system. Therefore, in our user-centered approach, we focus on how each user type can accomplish its responsibilities by using the system. We are looking at what functionalities the system should provide when each user type interacts with the system.

Each responsibility of a user type is carried out by performing some tasks. A task can be accomplished by performing a sequence of actions (known as a scenario). A task can be performed by different scenarios because of the variation and exception conditions. Figure 3.12 depicts the rationale for the accomplishment of a responsibility of a user type. We derive an $AND/OR\ refinement$ model from the traditional AND/OR graph structures. We call it *activity refinement.* There are two kinds of activity refinements: $AND\ refinement$ and $OR\ refinement$.

AND refinement. An activity A is AND refined by a finite set B of activities b_i, $i = 1..n$ such that (1) fulfilling all the activities of set B implies fulfilling activity A. (2) the failure of any activity in set B implies the failure of A.

OR refinement. An activity A is OR refined by a finite set B of activities b_i, $i = 1..n$ such that (1) fulfilling any activity of set B implies fulfilling activity A (2) The success of any activity in set B implies the success of A.

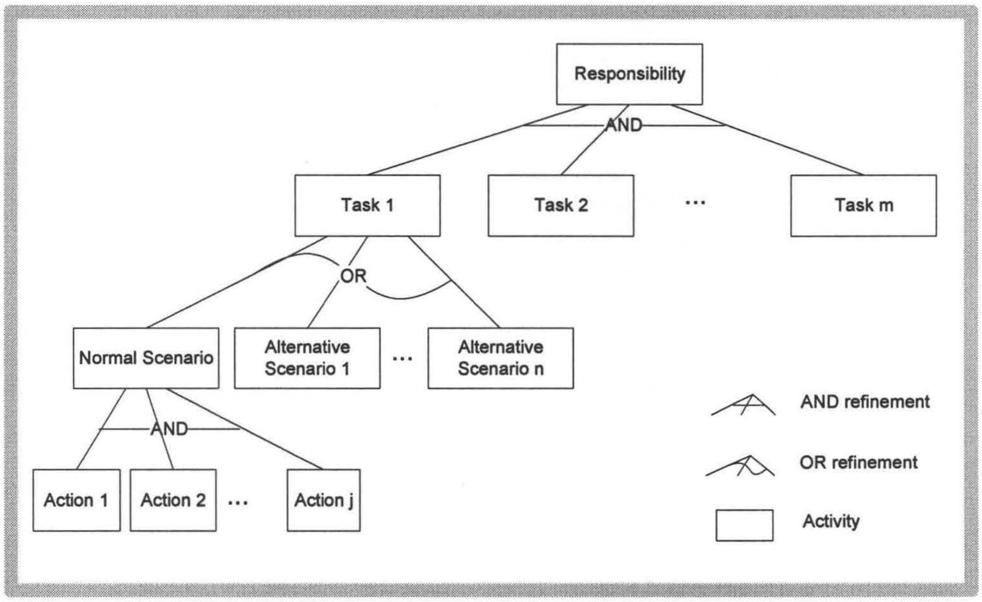


Figure 3.12: Fulfilling a Responsibility

In Figure 3.12, a responsibility is *AND* refined by a set of tasks and a scenario is *AND* refined by a set of actions. Similarly, a task is *OR* refined by a normal scenario and alternative scenarios.

3.2.4.3 Identify Primary Actor - Task List

As mentioned before, the primary actor - task list shows all the user's tasks that the system supports, showing the system's functionality. We start with investigating the tasks of primary actors because the tasks of supporting actors are entirely for supporting the tasks of the primary actors. Moreover, a primary actor is an abstract role which may represent more than one user types. Primary actor - task list also eliminates the redundancy of user type - task list.

Identifying rule:

- Each primary actor in the system context diagram is mapped to the corresponding problem sub domain of the problem context diagram, and the corresponding user tasks are identified from the shared phenomena in the problem context di-

Primary Actor	Tasks
User(Radiologist, Planner)	Enter PatientInfo & TargetInfo Link MRIImageSet Define Target

Table 3.6: The Primary Actor - Task List Example

agram.

- The identification is iterative and incremental, as the interviews go forward, the user tasks may be added, deleted and revised.

Table 3.6 lists part of the primary actor - task list in the treatment planning software example.

3.2.4.4 Perform a User's Task

In our user-centered approach, the core point is to figure out all the services the system should provide when each user type performs their tasks. So, we focus on what behaviors the system has to have when users use the system.

Task Workshop

As stated previously, we gather user's responsibilities, tasks and desired work-flows of each task directly from representatives of various user types. We advocate a task JRD, which takes the form of a series of 2-3 hour elicitation meetings or workshops. Each workshop's participants include user representatives, analysts and one or more developers. Developers serve as the voice of reality when infeasible requirements are suggested.

Each elicitation workshop may explore several tasks for certain user type. For each responsibility of certain user type, needed tasks are figured out first. For each task, it is the analyst who will capture the information. The participants begin by identifying a user type (the primary actor) who would perform the task. Next, they define the preconditions that have to be satisfied to perform the task, post-conditions that would describe the state of the system after the task is complete, and the estimated frequency of use which provides an early indicator of concurrent usage and capacity requirements. Then, the analyst asks the participants how they envision interacting with the system to perform the task. The resulting dialogue sequence of user actions and system responses becomes the flow that is identified as the normal scenario. In case that there are alternative normal courses, the normal scenario should be the one which is easy to understand and fairly typical for performing the task.

The normal scenario is the normal sequence of actions to accomplish the intended task with the system. A normal course can branch off into an alternative course at some decision point in the interaction sequence, then rejoin the normal course later. So, alternative courses can also result in successful task completion, which represent variations in the path to complete the task. Some of the steps in an alternative course will be the same as those in the normal course, but certain unique actions are needed to accomplish the alternative path. Conditions that result in the task being failed are usually documented as exceptions, which are also regarded as a type of alternative conditions in our approach. It is important to describe the exception paths, because they represent the user's vision of how the system should behave under specific conditions and they could cause the system to fail when they are overlooked.

It is impossible to complete all the information of a task in one meeting or workshop. Instead, we explore the task in increments, and then review and refine them iteratively, so that at a later stage, for example, the analyst may sketch the scenarios for explored tasks and give them to the workshop participants, who review them prior to the next workshop. These informal reviews can reveal many errors, such as previously undiscovered variations and exceptions, and missing steps in the action steps. Or alternatively, intense review workshops can be arranged to formally review the use cases after their sketching. The task-scenarios-use case approach provides a powerful way to improve requirements quality through such incremental reviews.

To summarize, each of the user's tasks will be accomplished through certain scenarios which the system must support. Alternative scenarios must also be supported when variations and exceptions happen. Each scenario includes actions from both the user and the system and the information needed for each action step. To explore a task, the following things should be considered:

- Precondition: conditions (system state) that must be true before the task can be performed.

- Normal scenario: a set of action steps which is the simplest and most common work flow.
- Alternative scenarios: alternative scenarios can complete the task itself, or rejoin the normal scenario after handling the alternative conditions, or fail.
- Success post condition: conditions that are guaranteed after successful completion of the task.
- Failure postcondition: what must be minimally guaranteed when task fails.
- System constraints: any non functional requirements that relate to this task.

Action Steps of a Scenario (Normal and Alternative)

The use case techniques can be used in exploring scenarios of tasks. According to Ivar Jacobson [49], a use case consists of a sequence of transactions and each transaction consists of several actions to be performed. A transaction has four parts (See Figure 3.13):

1. The primary actor sends request and data to the system. (For example, the user selects an MRIImageSet)
2. The system validates the request and the data. (System checks readability)
3. The system alters its internal state. (System sets the current selectedMRIImageSet)
4. The system responds to the actor with the result. (System presents the selectedMRIImageSet)

According to the composition of the compound interaction, we can derive a guideline of what should be described in an action step of a scenario:

- Among the four parts, each part, or combination of various parts, or all four parts can be an action step. The combination should depend on the complexity of each part and the natural breaks in the processing.

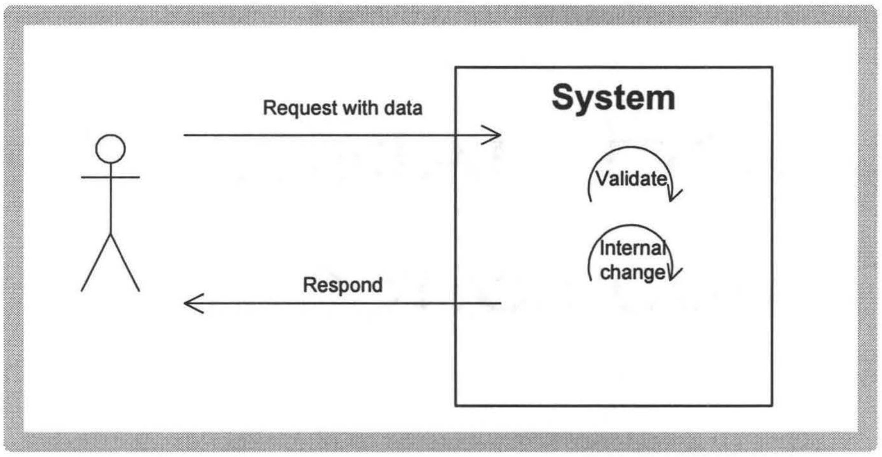


Figure 3.13: A Transaction Has Four Parts

- Identify explicitly what data items are passed through in each action step , and how these data items are retrieved or calculated.

If there is an existing system or manual process, it makes it easier for the users to give you the detail needed for these steps. If there is no existing system, encourage them to imagine it and to think of all the detail they need for the task – we advocate a user-centric way. Walk the users through the steps to encourage them to remember additional details.

Alternative Scenarios

In each action step of a normal scenario, variations and exceptions should be discussed. For each point where behavior can branch because of a particular condition (called *alternative condition*), write down the condition and then write the action steps that handle it. In most cases, these alternative handling steps end by simply merging with the normal scenario steps and lead to an alternative successful path. However, some alternative conditions can not be recovered and will produce failed alternative courses. Every alternative condition leads to an alternative scenario. Alternative conditions of an alternative scenario might be encountered. Based on the guideline below, questions like "What should happen if..." may be asked by analysts to find the alternative conditions. Once each task is fully explored and no additional variations, exceptions, or details are proposed, the workshop participants move on to another task.

Guideline to finding alternative conditions: *for each action step in a normal scenario, brainstorm what the system can detect differently compared with the normal situation. For example, invalid password or network not connected.*

Cockburn [22] lists a series of specific aspects to be considered, which are extremely helpful:

- An alternate success path (clerk uses a shortcut code).
- The primary actor behaves incorrectly (invalid password).
- Inaction by the primary actor (time-out waiting for password).
- Every occurrence of the phase "the system validates" implies that there will be an alternative condition (invalid account number).
- Inappropriate or lack of response from supporting actor (time-out for response).
- Internal failure within the system, which must be detected and handled (cash dispenser jams).
- Unexpected and abnormal internal failure, which must be handled and will have an externally visible consequence (corrupt transaction log discovered).
- Critical performance failures of the system (response not calculated within 5 seconds)

Patterns for Dealing with Alternative Conditions

When an alternative condition becomes true, we should consider how this can be handled by the system. The derived handling action steps branch from the normal scenario and lead to the alternative scenario of the task. Different alternative scenario patterns are illustrated in Figure 3.14. To enhance the understandability, action step 2 is specifically used as an example, and different alternative conditions of action step 2 are clarified.

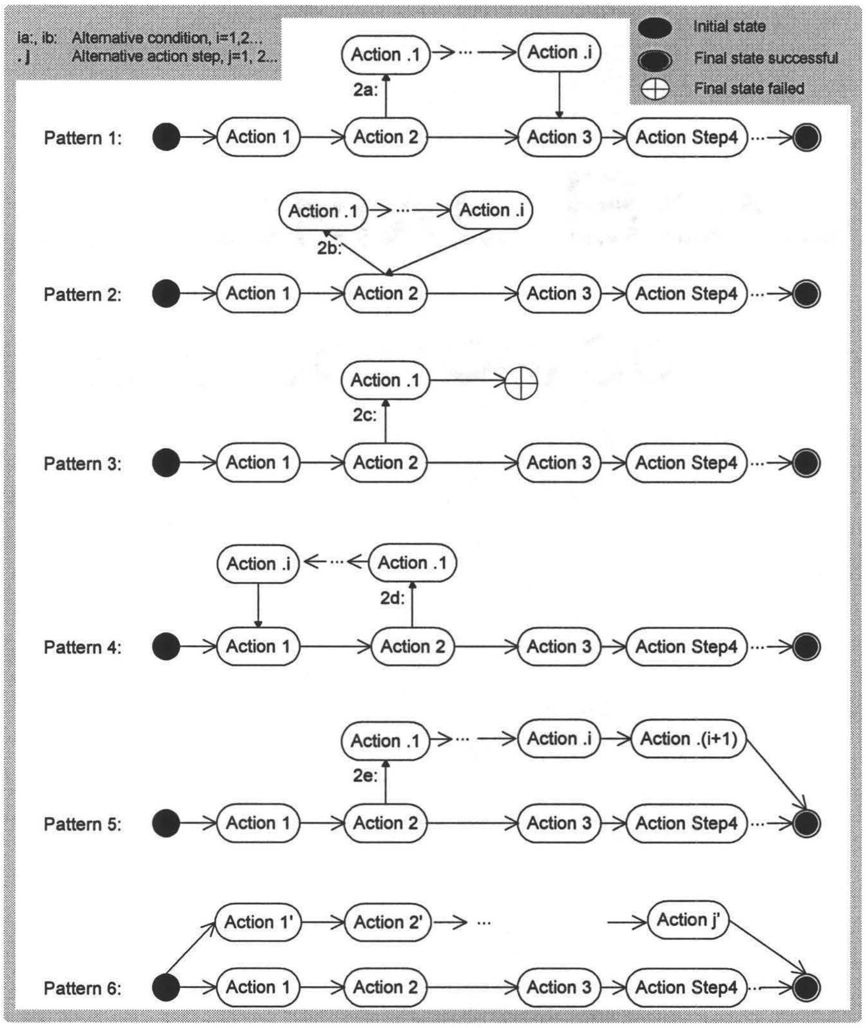


Figure 3.14: Patterns to Handle Alternative Conditions

A Tabular Form to Capture Scenarios of a Task - Scenario Table

When discussing with the users about performing a task, a scenario table will be an efficient tool for capturing the scenarios, see Table 3.7.

The condition table ideas and notations from [84] are the origins of this tabular form. In a scenario table, the *normal precondition* of an action step is identified to explore the alternative conditions. These conditions are actually

Task: Name of the task.
Task summary: Description of the purpose of the task.
Precondition: Things that must be true before the task can execute, they are predicates on the state of the system.
Success post condition: Things that must be true at the end of the task when the task succeeds.
Failure post condition: Things that must be true at the end of the task when the task fails.
System constraints: Any constraints to perform the task.

1	2	...	n
Precondition Normal step 1	*Normal precondition 2:* Normal step 2	...	*Normal precondition n:* Normal step n
▨	*Alternative condition 2a:* Alternative scenario steps.	...	...
▨	*Alternative condition 2b:* Alternative scenario steps.	...	...
▨	...	...	...

Table 3.7: The Scenario Table

predicates of the system state that must be true to perform the normal action steps, and they must meet the following disjointness and completeness properties:

Disjointness:

$\forall n, i, j, i \neq j,\ Normal\ precondition\ n \wedge Alternative\ condition\ ni \leftrightarrow FALSE$

$\bigwedge Alternative\ condition\ ni \wedge Alternative\ condition\ nj \leftrightarrow FALSE$, and

Completeness:

$Normal\ precondition\ n \vee Alternative\ condition\ ni \vee Alternative\ condition\ nj ... \leftrightarrow TRUE$

Construction of the table obeys the following rules:

- The horizontal header identifies the numbers of normal action steps.
- Each row of the grid contains at least one scenario, where the first row describes the normal scenario, each of the other rows describes alternative scenarios.
- Italic sentences state the normal preconditions of an action step, or alternative conditions. They end with a colon.

- Alternative conditions have identifiers with a normal step number followed by a lower-case character in alphabetic order, e.g. 2a stands for first alternative condition of normal step 2.
- Each grayed cell has the same action step as that of the normal scenario, while blank cells indicate there are no actions.
- Alternative conditions are explored from left to right, top to bottom of the table.
- Use 3 to 9 steps to perform a task. Complicated steps are rolled up to be a sub task (a sub task scenario table needed).
- For the alternative handling pattern 5 in Figure 3.14, the alternative condition handling steps are rolled up to be a sub task.
- For pattern 6, a separate scenario table may be needed.
- For steps that are sub tasks or other tasks, they are underlined.
- Alternative conditions of alternative scenarios must be explored.

The task "Link MRIImageSet" of a radiologist in the treatment planning software is used as an example to demonstrate this technique, see Table 3.8.

3.2.4.5 Gather System Constraints in a User's Task

We introduced categories of system constraints (non-functional requirements) in section 3.2.1 and noted that some general system constraints are gathered into the UPD in an earlier stage of the project. During task exploration, constraints that are common for many or all tasks are added to the system constraints section of the UPD. However, most non-functional requirements are related to a specific user task, such as requirements that specify the speed, availability, security, accuracy, response time, recovery time, or memory usage with which the system must perform a given task [50]. According to the guidance table 3.1, we capture the system constraints of each task and specify them in each user's task and then transfer them into the corresponding use case. After refinement and increment during use cases realization, eventually all these system constraints are concluded in the SRS in separate sections.

Task: Link MRIImageSet
Task summary: The user chooses to link an MRIImageSet to the current patient' current treatment plan.
Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. A new TreatmentPlan for current patient with initial data values has been created. Ref. Table 4.7
3. Required PatientInfo and TargetInfo have been stored. Ref. Table 4.7
Success post condition: An MRIImageSet is linked to current TreatmentPlan and presented.
Failure post condition: None.
System constraints: 1. Successive retries of the same MRIImageSet selection can be executed at most three times.

1	2	3	4
User selects to link an MRIImageSet.	*Number of available MRIImageSets* $\geq$ *1:* System presents a list of names of available MRIImageSets.	User selects an MRIImageSet of current patient.	*Selection of MRIImageSet is successful & Number of retried times* $\leq$ *3:* System links the selected MRIImageSet to current TreatmentPlan and presents the selected MRIImageSet, including presentation of the first MRIImage slice of the image set and list of MRIImage slices in the set.
▨	*2a. Number of available MRIImageSets* $<$ *1:* .1 System informs user. .2 Task fails.		
▨	▨	▨	*4a. Selection of MRIImageSet is not successful & Number of retried times* $<$ *3:* .1 System informs user that the selection of current MRIImageSet fails, and asks user to retry the current selection, or try another selection, or cancel the task. .2a User selects to retry current selection: system performs normal step 4. .2b User selects to retry another selection: system returns to normal step 2. .2c User selects to cancel: task fails.
▨	▨	▨	*4a. Selection of MRIImageSet is not successful &Number of Retried times* $\geq$*3:* .1 System notifies user of failure. .2 Task fails.

Table 3.8: A Scenario Table Example of Task: Link MRIImageSet.

When to Finish a Scenario Table?

A scenario table is finished when it is deemed correct (i.e, it captures the right requirements), complete (i.e., it describes all possible paths), and consistent. The scenario tables are evaluated by analysts and users at specific review meetings or at the beginning of the next task workshop.

3.2.5 Specify Use Cases

Use cases provide a way to represent the user requirements, which can be regarded as the system requirements. The objective of the user-centric approach to requirements elicitation is to describe all the tasks that the users (actors) will need to perform with the system. For easily exploring and clearly viewing reasons, we capture all the scenarios of performing the users' tasks in scenario tables. However, to document the scenarios, we use a well recognized mechanism - the use case model. A user case model consists of individual use cases and use case diagrams. Each use case is a textual description which collects the scenarios of certain user tasks. In theory and practice, the resulting set of use cases will encompass all the desired functionality of the system, because use cases are collections of scenarios of user tasks [86].

3.2.5.1 Rules for Specifying Use Cases

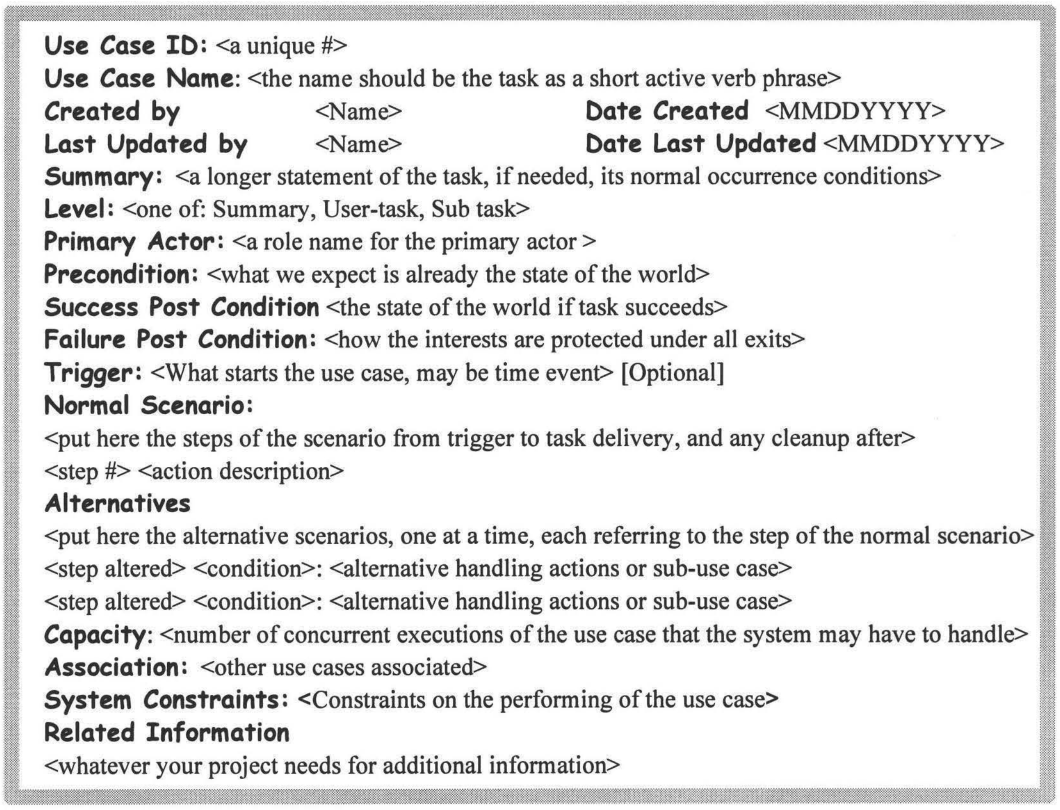
Using the scenario tables, we specify use cases according to the following rules:

- Each primary actor's task is a task level use case.
- Cells of the scenario tables of a task are transferred into corresponding part of the use case template.
- Each sub (sub) task is a sub task level use case.
- System constraints are also transferred.

According to the analysis of the scenario tables of each task, we sketch out the use case for each task and validate that with the users at review meetings.

3.2.5.2 A Use Case Template

We develop a use case template as a guide to specifying use cases (see Figure 3.15).



Use Case ID: <a unique #>
Use Case Name: <the name should be the task as a short active verb phrase>
Created by <Name> **Date Created** <MMDDYYYY>
Last Updated by <Name> **Date Last Updated** <MMDDYYYY>
Summary: <a longer statement of the task, if needed, its normal occurrence conditions>
Level: <one of: Summary, User-task, Sub task>
Primary Actor: <a role name for the primary actor >
Precondition: <what we expect is already the state of the world>
Success Post Condition <the state of the world if task succeeds>
Failure Post Condition: <how the interests are protected under all exits>
Trigger: <What starts the use case, may be time event> [Optional]
Normal Scenario:
<put here the steps of the scenario from trigger to task delivery, and any cleanup after>
<step #> <action description>
Alternatives
<put here the alternative scenarios, one at a time, each referring to the step of the normal scenario>
<step altered> <condition>: <alternative handling actions or sub-use case>
<step altered> <condition>: <alternative handling actions or sub-use case>
Capacity: <number of concurrent executions of the use case that the system may have to handle>
Association: <other use cases associated>
System Constraints: <Constraints on the performing of the use case>
Related Information
<whatever your project needs for additional information>

Figure 3.15: A Use Case Template

Also, when writing use cases, some guidelines are listed below. These are revisions of guidelines given in [22] and [64].

- Using words suited to user requirements, that are strong (enter vs input), precise (data item vs information) and flexible (present vs display), which will not constrain the future software design or evolution.

- Use simple grammar to describe an action step, with the form "Subject... verb...direct object...prepositional phrase", e.g. the system...deducts...the amount...from the account balance.

- Show clearly which actor is passing the message to the other (who to whom).
- Write like you are watching over the user using the system. The customer...The system...
- All the data that passes in one way gets collected into just one action step.
- Avoid "whether" when describing the success condition, usually describe the scenario as "succeeding", e.g. "the system checks whether the password is correct" should be written "the system validates that the password is correct."
- Optionally mention the timing, not always, because most steps follow directly from the previous one. However, occasionally, you will need to say that "at any time between steps 3 and 5, the user will..." or "as soon as the user has, the system will..."
- Write "Do steps x-y until condition." when some steps need to be repeated.
- When a use case references another use case, the referenced use case is underlined.
- Do not mention GUI design details such as buttons, drop-down lists etc.
- Do not mention design or implementation details such as applet, database. This would restrain the designs.

As an example, we use these rules and recommended guidelines to derive the use case Link MRI Image Set from the task scenario Table 3.8, see Table 3.9.

3.2.5.3 Use Case Diagram

A use case model contains all the use cases. It summarize all the possible uses of the system. All the different use cases for a system can be depicted in a use case diagram.

Use case diagrams address the static use-case view of a system. Once all the use cases are specified, a use case diagram will give a clear picture of the functionality of a system. Figure 3.16 is a part of the use case diagram of the treatment planning software system.

Use Case ID: UC002
Use Case Name: Link MRIImageSet
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user chooses to link an MRIImageSet to the current patient's current treatment plan.
Level: User Task
Primary Actor: User(Radiologist, Planner)
Precondition:
1. User has selected to make TreatmentPlan for this patient.
2. A new TreatmentPlan for current patient with initial data values has been created.
3. Required PatientInfo and TargetInfo have been stored.

Success Postcondition: An MRIImageSet is linked to current TreatmentPlan and presented.
Failure Postcondition: None.
Normal Scenario:
1. User selects to link an MRIImageSet.
2. System presents a list of names of available MRIImageSets.
3. User selects an MRIImageSet of current patient.
4. System links the selected MRIImageSet to current TreatmentPlan and presents the selected MRIImageSet, including presentation of the first MRIImage slice of the image set and list of MRIImage slices in the set.

Alternative Scenarios:
- 2a. Number of available MRIImageSets < 1:
  - .1 System informs user.
  - .2 Task fails.
- 4a. Selection of MRIImageSet is not successful & Number of retried times < 3:
  - .1 System informs user that the selection of current MRIImageSet fails, and asks user to retry the current selection, or try another selection, or cancel the task.
  - .2a User selects to retry current selection: system performs normal step 4.
  - .2b User selects to retry another selection: system returns to normal step 2.
  - .2c User selects to cancel: task fails.
- 4b. Selection of MRIImageSet is not successful & Number of Retried times ≥ 3:
  - .1 System notifies user of failure.
  - .2 Task fails.

Capacity: 1
Associations: None.
System Constraints: Successive retries of the same MRIImageSet selection can be executed at most three times.

Table 3.9: A Use Case Example: Link MRIImageSet.

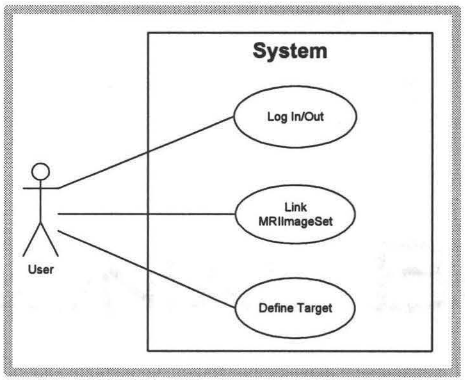


Figure 3.16: The Partial Use Case Diagram-Treatment Planning Software

3.2.5.4 Grouping Use Cases

Once all the use cases are specified, they can be organized in clusters according to the functional areas or actors. The categorization will help find the specific use case quickly, especially for a large system. Each use case group should have a use case diagram to provide an overview of the use cases within it.

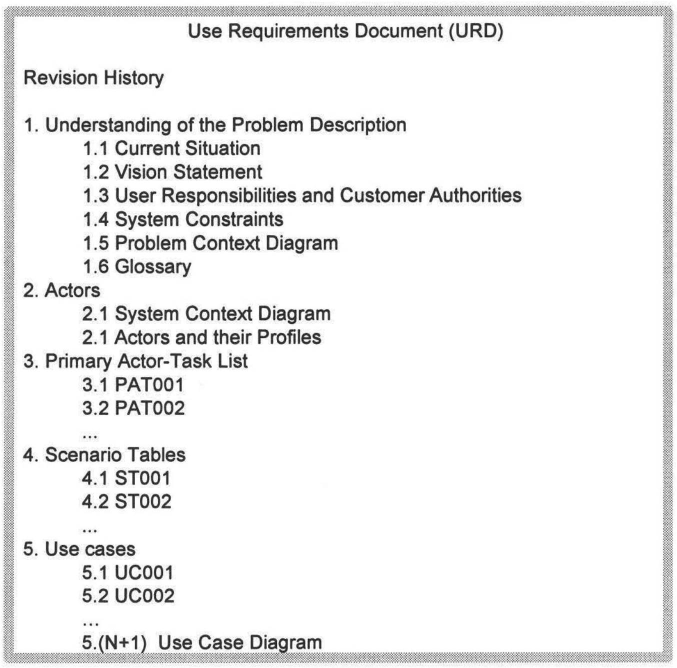
3.2.5.5 Review and Validate the Use Cases

Once all or part of the use cases are sketched, they should be reviewed by the users. Also, graphical models like flowcharts, interaction diagrams, entity relationship diagrams can be used to enhance the shared understanding of the use cases, both for the analyst and the users. These models depict different views of the use cases, that can often disclose inconsistency, incompleteness, omissions, ambiguities etc. We suggest a storyboard format to validate use cases using the activity diagrams. Details about the validation are stated in section 3.5.

Like the whole process of software development, the development of use cases is also incremental and iterative. After several times of informal or formal review, a set of complete, correct and consistent use cases is produced as the output of the elicitation process.

3.2.5.6 User Requirements Document, URD

In the process of elicitation, the following artifacts are produced: understanding of the problem description, actors and their profiles, primary actor - task list, scenario tables of tasks, use case descriptions. All together, they are captured into a *user requirements document.* A suggested template is introduced as below in Figure 3.17. Each primary actor task has a unique identifier "PAT#" mapping to a scenario table "ST#". A similar mapping applies between a scenario table "ST#" and a use case "UC#".


Use Requirements Document (URD)

Revision History

1. Understanding of the Problem Description
    - 1.1 Current Situation
    - 1.2 Vision Statement
    - 1.3 User Responsibilities and Customer Authorities
    - 1.4 System Constraints
    - 1.5 Problem Context Diagram
    - 1.6 Glossary
2. Actors
    - 2.1 System Context Diagram
    - 2.1 Actors and their Profiles
3. Primary Actor-Task List
    - 3.1 PAT001
    - 3.2 PAT002
    - ...
4. Scenario Tables
    - 4.1 ST001
    - 4.2 ST002
    - ...
5. Use cases
    - 5.1 UC001
    - 5.2 UC002
    - ...
    - 5.(N+1) Use Case Diagram

Figure 3.17: Use Requirements Document

3.3 Software Requirements Analysis, SRA

Software requirements analysis is a core process in software requirements engineering. As stated in the preceding chapter, it is the various methodologies of the analysis process that produce distinctly different approaches of software requirements engineering. However, no matter which approach is used, the general principle of the analysis process is to find, invent and define the "function and data" of the new software system.

In our user-centric approach, we use an object-oriented requirements analysis (OORA) method. To us, software requirements analysis is the process of studying user requirements to define the context of possible software solutions to the problem, namely, arriving at a definition of software requirements based on users' needs. We take user requirements (use cases) as inputs to this process, and software requirements (analysis model) as outputs. The resulting analysis model (class model, activity diagrams, sequence diagrams) demonstrates what is to be built and will form the base of the functional requirements of the software requirements specification. Meanwhile, key elements of user interfaces are developed and captured, as well as the newly found system constraints during analysis of each use case (tagged to the analysis model, or put into the relevant part of SRS directly). An abstract process model is depicted in Figure 3.18.

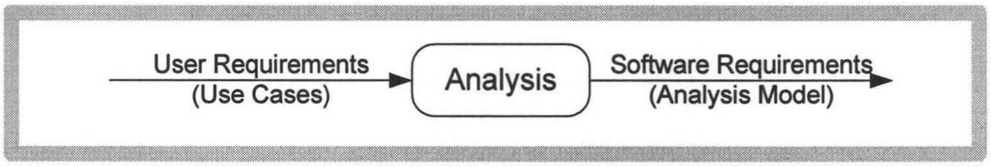


Figure 3.18: Analysis Process Model

3.3.1 Denotations of Analysis

Software development can be regarded as problem solving. Analyzing a problem is the core activity to get a solution for it. We should clarify what analysis means within the context of our approach.

Analysis.

In Merriam-Webster's Online Dictionary [45], the definition of analysis is "Separation of a whole into its component parts; an examination of a complex, its elements, and their relations".

Problem Analysis.

Alan Davis says in [28], "Problem analysis is the activity that encompasses learning about the problem to be solved, understanding the needs of potential users, trying to find out who the user really is, and understanding all constraints on the solution." He thinks that problem analysis can be thought of as defining the product space, that is, the range of all possible software solutions. The product space is that range of problem solutions that meets all known constraints. Moreover, he divides the whole requirements stage into two parts — problem analysis and writing the SRS. Also, he concludes that problem analysis is primarily a decomposition process: decomposing problems into subproblems with the goal of understanding the entire problem at hand.

System Analysis.

One pioneer of the structured analysis method, DeMarco, offers the following definition [30]: "Analysis is the study of the problem, prior to taking some action." In [20], Peter Coad and Edward Yourdon regard that "analysis is the study of a problem domain, leading to a specification of externally observable behavior; a complete, consistent, and feasible statement of what is needed; a coverage of both functional and quantified operation characteristics (e.g., reliability, availability, performance); analysis means the process of extracting the needs of a system - what the system must do to satisfy the client, not how the system will be implemented."

Our Meaning of Analysis

In software engineering, we may often encounter terms such as "analysis", "problem analysis", "system analysis", "problem domain analysis" etc. These terms are synonyms or near synonyms with the same meaning as above, covering both the requirements elicitation and the requirements analysis process. Other terms, like "requirements analysis", "software requirements analysis", or even sometimes "analysis" [50], often have the same meaning as ours (see paragraph below), which is commonly recognized in the software requirements engineering field.

Our meaning of analysis, precisely speaking, software requirements analysis (SRA), is a process through analysing the user requirements, to seek the required functions and data items of the system to be built, and allocate them to externally observable parts of the system in an object-oriented manner. It is applied specifically to the analysis purely as opposed to the elicitation or SRS documentation process of the requirements. It is an updating process from the Unified Process (UP) [50], along with some concepts and artifacts absorbed from [7, 10, 20, 50]. Our analysis process performs different activities from UP, which we believe are more practical, and simpler than UP.

Like most analysis methods, SRA is also a decomposition process - the process to seek the data involved, to partition and allocate the required functionality to different functional parts of the system, and to capture them and their relationships from the developer's point of view. In object-oriented analysis (OOA) terminology, the "functional parts" are *analysis classes* including *domain classes* and *application classes.*

In user-centric software, the domain classes come from the problem domain. Domain classes are conceptions of the problem domain, they physically exist or will exist in the problem domain. Domain classes are application independent[1], they are mainly the entity and data sub domains, abstracted for use in the system. The application classes exist only along with the application. They are abstractions of interfaces between the system and the problem domain, along with functions that the system needs to perform externally visible behavior (domain classes also contain some functions in the same sense). To avoid confusion with the notation of *interface classes* of some programming languages like Java, we will not call them interface classes. Instead, we call them boundary classes which is a name commonly used in OOA [49, 50].

The source of SRA is the use case model captured in the elicitation process.

[1] Although some domain classes are things that will be produced by the application, such as a treatment plan report, they are still application independent - the conception of them exists independently from any kind of applications, or even without an application.

The purpose of analysis is to achieve a more precise understanding of the user requirements and to produce a model of the software system (the analysis model) which is correct, complete, consistent, and verifiable. The analysis model specifies what is to be built — software requirements. It consists of other individual models, such as the static model — analysis class diagrams, the dynamic models — activity diagrams, sequence diagrams, and the state model — statechart diagrams (optional in our approach). Object Management Group (OMG) UML [62], a developers' language is used to specify the analysis model.

During analysis, analysts focus on realizing, structuring and formalizing user requirements — the use cases, finding ambiguities, incompleteness, inconsistencies and errors and updating them in use cases.

3.3.2 Software Requirements are at the Interfaces

Recalling our top-level problem context diagram that is redrawn here (Figure 3.19), the requirements lie in the shared phenomena SP (events, states, values) between the system and the problem domain. User requirements are the description of the shared phenomena from the user's point of view when they interact with the system, while the software requirements (specification) describe the behavior that the system must have at its interface with the world, from the developer's point of view. For example, any user input is an event from the user's view, while, to a developer, it is a function of a particular boundary class.

So, software requirements describe the system's behaviors that are externally "visible". The description contains the data items the system will operate on and the functions the system will provide. These data items and functions are held by objects of analysis classes. The objects of analysis classes are visible parts of the system, such as an MRI image. The shared phenomena at the interface of the system and the problem domain is handled by objects of analysis classes.

3.3.3 Why Objected-Oriented Requirements Analysis?

There are several reasons to use an object-oriented analysis method for a user-centric software SRS.

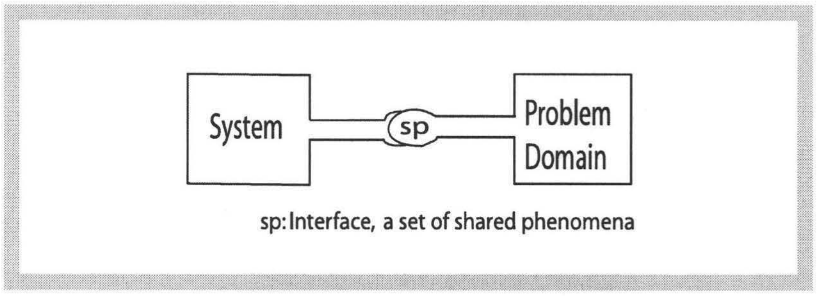


Figure 3.19: Top Level Problem Context Diagram

Popularity of Object-Oriented Development Approach

The traditional structured approach to software systems development, also called the functional or procedural approach, was popularized in the 1980s, including structured analysis and design technique (SADT) [71], structured system analysis and design methodology (SSADM) [33] etc. Most of these methods used data flow diagrams (DFD) for process modeling, and some used entity relationship diagrams (ERD) for data modeling. In this approach, the system is decomposed into functional parts with a process-centric hierarchy structure linked by data flows. The main drawback of this approach is that the whole functionality must be specified first, to do top-down decomposition, it is hard to modify and extend when requirements change. For example, when a functionality needs to be added, it may change many of the existing functional parts.

The object-oriented development approach became popular in the 1990s when object-oriented programming was catching on, especially because of the C++ programming language. The main object-Oriented approaches include Object-Oriented Analysis and Design with Applications (OOAD) [12], Object-oriented Software Engineering (OOSE) [49], Object-Oriented Modeling and Design (OMD) [75] and Unified Process (UP) [50]. The object-oriented approach partitions a system into parts of various granularity (subsystems, components and objects), with classes of objects at the bottom of this decomposition. Each object encapsulates a set of services (also called functions or methods) and a state (a set of data, a data structure, or attributes) on which the services can operate. Functionality is carried out by interactions among objects which are linked by various relationships. The system behavior is the result of specified object interactions.

With the advent of modern graphical user interfaces (GUIs), the approach to computing has changed dramatically. GUI (user centric) programs are event-driven and execute in a random and unpredictable fashion dictated by user-generated events from a mouse, keyboard, or other input devices [55]. Object-oriented programming is well-suited to the rising popularity of GUIs. Behind every event, there is a software object that knows how to service that event. Once the service is accomplished, control returns to the user and system waits for the next event. Since the object-oriented programming meets the development of the modern GUI systems so well that it dominates the programming world now, so does the object-oriented development approach.

Other reasons for the popularity of the object-oriented approach are the technical advantages of the object paradigm, such as reuse, inheritance, message passing, polymorphism and abstraction. These technical properties contribute to greater usability of code and data, shorter development times and increased programming productivity.

As a fact, the object-oriented approach to system development became fashionable in the 1990s and people will still be using it tomorrow.

On Construction of a Seamless Software Development Approach

Object-oriented requirements analysis (OORA) is a method of formulating the requirements for a software system in terms of objects and their interactions [7]. Based on the analysis result, a specific template and language are used to document the SRS that is the essential input of the software design phase. Together with the use case method in the requirement elicitation phase, all of this constitutes a SRS methodology, as a part of the whole software development approach which further includes design, implementation and test phases.

Different analysis criteria for decomposing a system make it difficult to proceed from one kind of analysis result to another kind of design phase, for example, from structured analysis to object-oriented design. The result of structured analysis — process-oriented decomposition, may not flow well into an object-oriented design — interacting parts decomposition, and more reorganization may be required to perform the transition [24, 76]. Object-oriented requirements analysis decomposes the system into interacting parts (analysis classes) according to the user requirements, in a way that is compatible with object-oriented design.

The object-oriented requirements analysis approach facilitates the seamless transition from requirements phase to design phase. Most of the analysis classes can remain during the design phase. It also simplifies the traceability of requirements throughout the development life cycle (e.g. when a function needs to be changed during design, it is easy to go back to the corresponding analysis class, and further the related use case).

Characteristics of User-Centric Software

As discussed in the introduction chapter, user centric software is a kind of application with intensive user-system interactions and multiple user interface elements (forms, list items, folders, buttons). It is the main type of modern GUI application. An object oriented development approach is appropriate, and so is an object oriented requirements analysis.

3.3.4 Analysis Classes

In an object-oriented requirements analysis method, the functionality of the system is decomposed into interacting functional parts which are linked by various relationships. It is the interactions among these functional parts that fulfill the functionality of the system and meet the user's requirements. We call these parts *analysis classes* in accordance with the object-oriented analysis notation.

Analysis classes are abstractions from the problem domain and the computer application to be built. They represent the externally visible parts of the system. As stated in [50], analysis classes represent abstractions of classes and possibly subsystems in the system's design. Each analysis class holds a set of functions that perform some exterior behavior of the system, and a set of data which are operated on by these functions. All the functions of analysis classes constitute the functional requirements of the system. There are two kinds of analysis classes, domain classes and application classes (in our case, they are boundary classes).

Although in most OOA methods [12, 49, 50, 75], people advocate another kind of application class — control class, or controller, we do not think it is appropriate at the requirements stage. According to their definition, instances of control classes are used to manage instances of domain classes (called entity classes) and boundary classes. These controllers seem to be more related to "how" than to "what" and would be more appropriate to the software design stage.

Domain Class

A domain class represents a problem domain concept such as an individual type (a doctor), a problem domain object type (an MRI image), an event (an arrival) or a set of information (a set of parameters). They are derived from the problem sub domains. A domain class abstractly models information that is long lived and often persistent in the system. They physically exist or will exist (after the system is built) in the problem domain. Most domain classes are from a problem domain that are meaningful outside of any application [74], even if some of them are produced by the applications, such as the treatment plan class in a treatment planning application, see Footnote 1 on page 67. Domain classes are identified from use cases, and only those containing information needed by the system are captured.

Domain classes not only hold the information the system is dependent upon, but they can have behavior related to the information they represent. However, in most cases the objects of domain classes are manipulated by objects of boundary classes in our approach.

Domain classes represent the information that needs to be handled by the system to be built. Domain objects (objects of domain classes) have a mapping from the real world. While this mapping is the reason object-oriented development approach started, it also ensures the stability and flexibility of requirements change throughout the whole development process. Domain classes survive through to the design phase.

Application Class (Boundary Class)

Application classes, in our case only boundary classes, are the main external behavioral parts of the system that perform some functionality, based on information in domain classes. Boundary classes are abstractions of interfaces between the system and its environment, together with the functions needed to perform the related system's behavior. There are various boundary classes such as *user boundary class (UBC)* and *system boundary class (SBC)*. A user boundary class abstracts the key elements of a particular user interface, or multiple user interfaces, or a part of a user interface, and some functions related to that user interface(s). Likewise, a system boundary class contains information of a particular interface to other systems, and related functions.

Boundary classes are application dependent - they exist only in the system rather than in the problem domain, and they are externally visible (perceivable) to users [74] or to other systems.

In UP [50], a boundary class is used to model the interaction between the system and its actors (i.e. users and external systems). In our approach, we give the boundary classes more interaction semantics, that is, a boundary class is an integration of certain functionalities of the system. The functionality includes not only the interaction between the system and its actors, for example, a user enters a password, but also the interaction among objects of analysis classes which produce the externally visible behavior of the system, for example, when the system switches from one screen to another screen, it is an interaction between user boundary objects.

So, while boundary classes represent the interfaces between the system and its environment (actors), they also present the functionality of the system. For actors representing users, they provide user interfaces (UI) such as windows, forms, panes, capturing commands and queries and presenting feedback and results. For actors representing external systems, they provide system interfaces (SI) such as communication interfaces, printer interfaces, sensors, terminals, and APIs. For objects of analysis classes, they provide the functions to communicate with.

At the requirements level, boundary classes do not define design attributes, but rather interface data items (key elements of interfaces) and their related behavior (needed functions). So, boundary classes often are kept on a fairly abstract level, especially the user boundary classes. Since not only is detailing them time consuming, also they usually will change even at the test phase. However, most of the boundary classes survive through to the design phase.

We advocate using a user boundary class to represent a user interface (UI), or multiple UIs, or even just a part of a UI, because we do not want to restrict the design of UIs. That is, UI designers can freely compose or decompose the user boundary classes to build UIs.

3.3.5 Analysis Model - What is to Be Built

Analysis is about finding what is to be built. We need to decompose a complex set of user requirements into the essential parts (analysis classes) and their relationships on which we will define the software solution of the problem. The output of the analysis process is the analysis model which includes a set of individual models: analysis class model, activity model, interaction model and state model.

The analysis class model describes the objects that are manipulated by the system, their properties and their relationships. This model gives a static view of the objects and their relationships and provides an infrastructure where the activity, interaction and state models can be grounded. We use UML class diagrams to depict the analysis class model, including classes, their attributes, and functions (services), and their relationships.

Dynamic models (activity and sequence diagrams and state charts) focus on the behavior of the system. An activity diagram shows the flow of control among the processing steps. A sequence diagram represents the interaction sequence among a set of objects of a use case. A state diagram represents the behavior of a single object. Dynamic models demonstrate and validate that the class model is feasible. In our approach, we will use dynamic models to find classes, their attributes, relationships and functions with a bottom-up flavor.

The analysis model describes different aspects of the system in a complete manner, presents what must be built to meet the user requirements and demonstrates our deep understanding of the user requirements from the developers point of view.

3.3.6 Overview of Analysis Process

An analysis class model is the cornerstone of objected-oriented system development. Unfortunately, classes are chronically difficult to find and the properties of classes are not always obvious. Bahrami [6] concludes the main points of the four most popular approaches for identifying classes, noun phrase approach, common class patterns approach, CRC (class responsibility collaborators) approach, and use-case driven approach [50]. Each of the approaches provides its own guideline to find classes. However, most of the guidelines are heuristic and half-systematic, for example, there are no systematic methods to decide if a datum is a class or an attribute. In these approaches, the analyst's knowledge, experience and even intuition will play important roles during the seeking of classes, attributes and their relationships.

As stated previously, all the use cases constitute the whole functionality of a system from the user's point of view. As a result, the aim of analysis intuitively is to find from all the use cases what is needed to fulfill the functionality. In an OO based system, the functions (services, methods) of all classes compose the functional requirements in a software requirements specification SRS. For each use case, the functionality and needed information will map to the methods and attributes of one or more classes. The bridge to finding the mapping is the activity diagrams and sequence diagrams in our approach. In the activity diagram of each use case, we record the needed information in each action step. This information is represented by either domain classes (objects) or their attributes. In the sequence diagram of each use case, we show the objects that are required at the top of lifelines. These indicate the application classes (boundary classes), existing domain classes or newly found domain classes, and the messages that are needed among them. These represent the functions and usage associations.

So, based on the guidelines of the use-case driven method of UP [50], we seek classes and their properties, but we have developed a more systematic process than UP. It has a bottom-up flavor: once the use cases are known and the interaction models (sequence and activity model) are at least partly defined, the objects used in these models lead to the discovery of classes. Meanwhile, while realizing the use cases, ambiguity, incompleteness, missing behavior can also be identified and revised.

The analysis process is iterative and incremental in the sense that any model can be updated, and any newly found classes, attributes or relationships can be added and any errors can be revised. The analysis process has the following steps.

1. Draw activity diagrams for use cases.
2. Identify data used in activity diagrams and draw data hierarchies.
3. Identify domain classes, their attributes and relationships from data hierarchies.
4. Construct sequence diagrams for use cases.

5. Identify boundary classes, class functions from sequence diagrams.
6. Construct the whole class diagram.
7. Draw necessary state diagrams (optional).

3.3.7 Draw the Activity Diagrams for Use Cases

Activity diagrams can be used to "execute" a use case through a flow of actions. During the realization of use cases, any newly found ambiguity, incompleteness, error should be revised and updated in the corresponding use case, as well as in the system constraints.

For each use case, at least one activity diagram should be constructed. Each sub use case should have its own activity diagram. For our preceding example "Link MRIImageSet" use case, ref. Table 3.9, we can realize it by drawing an activity diagram, see Figure 3.20. To simplify the example, we do not put the exception handling in this diagram.

3.3.8 Identify Data Used in Activity Diagrams and Draw the Data Hierarchies

Both domain classes and their attributes can be identified from the use cases [49]. To distinguish domain classes and their attributes, we use a data decomposition mechanism to construct data hierarchies. This mechanism originates from M. Jackson [46] and J. Chen et al. [19], but we have added some "rules" for identifying classes and attributes from the desired data hierarchy.

Data Decomposition

Data in a system should be decomposed if their components are operated on by some operations. In data decomposition, data relationships including sequence, iteration, and selection may exist between a datum and its components, as described next:

- Sequence relationships indicate that a datum contains one copy of its components. For example, Figure 3.21 (a) indicates that a "point" contains one copy of "x" value, one copy of "y" value, and one copy of "z" value.

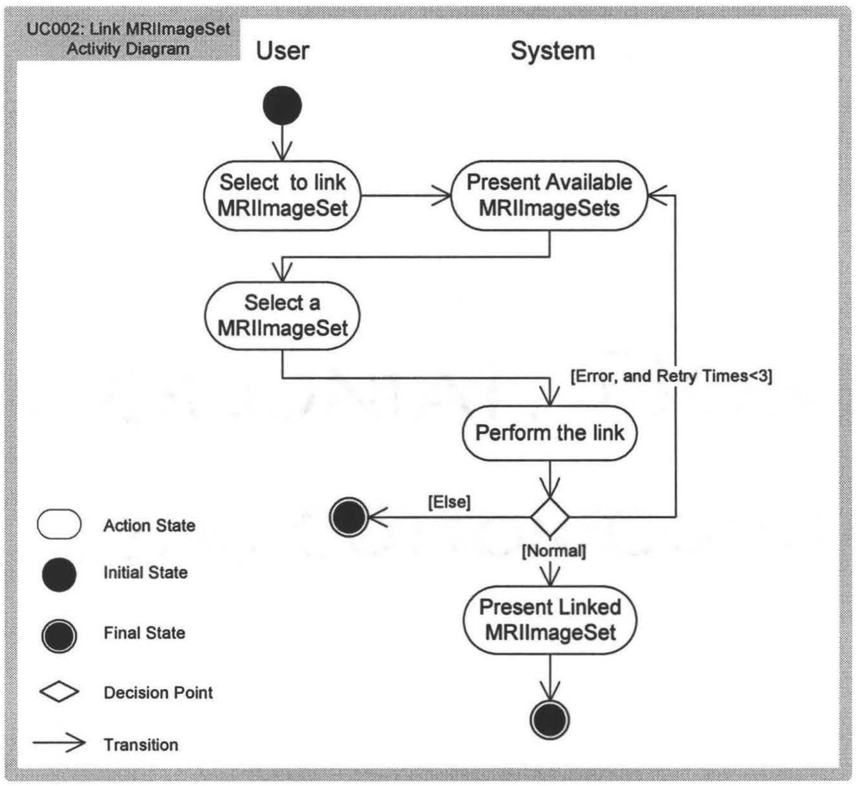

UC002: Link MRIImageSet Activity Diagram
User
System
Select to link MRIImageSet
Present Available MRIImageSets
Select a MRIImageSet
[Error, and Retry Times<3]
Perform the link
[Else]
[Normal]
Present Linked MRIImageSet
Action State
Initial State
Final State
Decision Point
Transition

Figure 3.20: Activity Diagram for the use case "Link MRIImageSet"

- Iteration relationships indicate that a datum contains multiple copies of its components. For example, Figure 3.21 (b) indicates that an "MRIImage" contains multiple copies of "Point."
- Selection relationships indicate that a datum is a generalization of its components. For example, Figure 3.21 (c) indicates that "Image" is a generalization of "MRIImage" and "CTImage."

Since data components can be further decomposed, a datum may be decomposed into a hierarchy called the *data hierarchy*. Figure 3.22 demonstrates what a data hierarchy looks like.

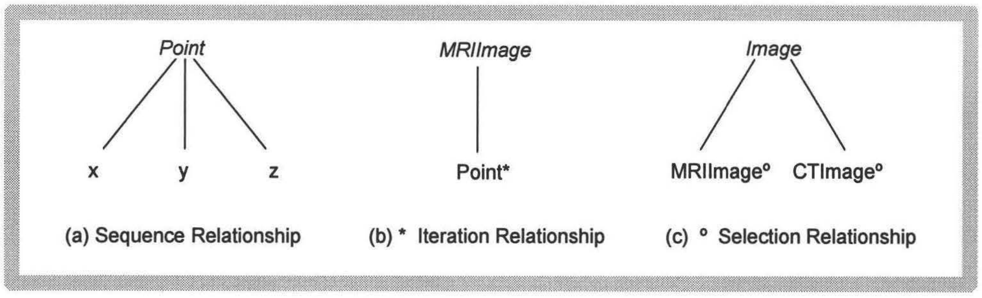

Point
x
y
z
MRIImage
Point*
Image
MRIImage°
CTImage°
(a) Sequence Relationship
(b) * Iteration Relationship
(c) ° Selection Relationship

Figure 3.21: Data Relationships

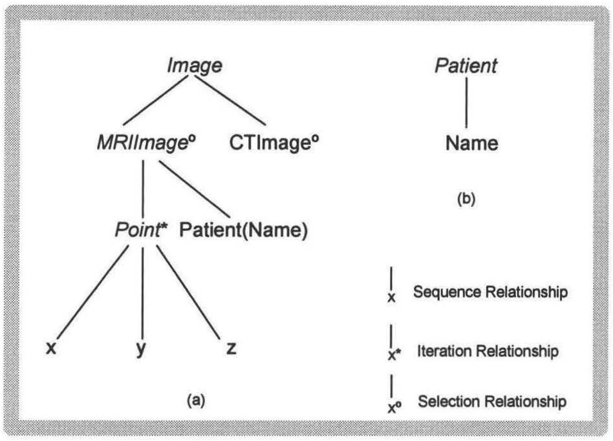


Figure 3.22: A Data Hierarchy Example

Generally, a datum can be transformed into a class if its components are operated on by some operations, because classes encapsulate attributes and the operations that operate on those attributes. That is, data that need to be decomposed can be transformed into classes, otherwise they will be attributes of classes.

Rules for Identifying Classes and Attributes from Data Hierarchy

Based on the discussion above, the following rules can be used to identify classes and their attributes from a data hierarchy.

1. The datum in a leaf node of a data hierarchy will become an attribute of its parent, and non-leaf nodes are classes. For example, in Figure 3.22 (a), "x" is an attribute of its parent "Point".

2. A non-leaf node with a parent should be regarded as a part class of its parent. In this regard, data hierarchies can be used to identify class aggregation (composition) relationships. For example, in Figure 3.22 (a), "Point" is a part class of "MRIImage".

3. If a non-leaf node has other non-leaf nodes as children and selection relationships exist among the children, then inheritance relationships exist between the parent and the children. In Figure 3.22 (a), "Image" is a generalization of "MRIImage" and "CTImage".

4. If a class contains a component of another class, there exists a uni-association[2] relationship between them, In Figure 3.22, "MRIImage" has a uni-association relationship with "Patient" because of its "Patient(Name)" component.

Construct the Data Hierarchies from Activity Diagrams

As discussed previously, finding the data involved in the user requirements and further identifying them to be domain classes are the primary tasks of the analysis process. In order to systematically seek the data demanded of use cases, we use the activity diagrams as the source and explore action steps one by one. The following guidelines govern this activity.

1. For each activity diagram of each use case, list data used by each action beside it, then related data are structured using these relationships: sequence, iteration, and selection.

[2] If class A has a uni-association with class B, it denotes that there exists a unidirectional link between objects of A and B. When A occurs, B must occur, not vice versa.

2. Create data hierarchies for each activity diagram. Data hierarchies that are included in previous use cases can be omitted.

3. Repeat 1-2 until all use cases have been explored.

4. Combine all of the data hierarchies into the final data hierarchy model (FDH) $FDH = \biguplus DH_i$, where $\biguplus$ denotes combination of all data hierarchies, $i = 1..N$, N is the number of activity diagrams.

5. In FDH, non-leaf data is expressed using italics. To simplify the diagram, detailed decomposition of a non-leaf node is shown only at one place.

Figure 3.23 shows the data used in use case "Link MRIImageSet" and the corresponding data hierarchy.

3.3.9 Identify Domain Classes, their Attributes and Relationships

Once all the use cases are realized by activity diagrams and the needed data relationships are captured in corresponding data hierarchies, we will get a stable data hierarchy model, FDH, by combining all the data hierarchies.

Based on the identifying rules above, we can easily identify from the FDH the domain classes, their attributes, and their relationships. We capture this information in a class diagram, we call the *initial domain class diagram.* For example, from the data hierarchies in Figure 3.23, we can get an initial domain class diagram as shown in Figure 3.24.

3.3.10 Construct Sequence Diagrams for Use Cases

The sequence model adds details and elaborates the informal themes of use cases. Each use case requires one or more sequence diagrams to describe its behavior. Each sequence diagram shows a particular interaction sequence of the objects participating in the use case, and it illustrates how the use case behavior is realized by the analysis classes.

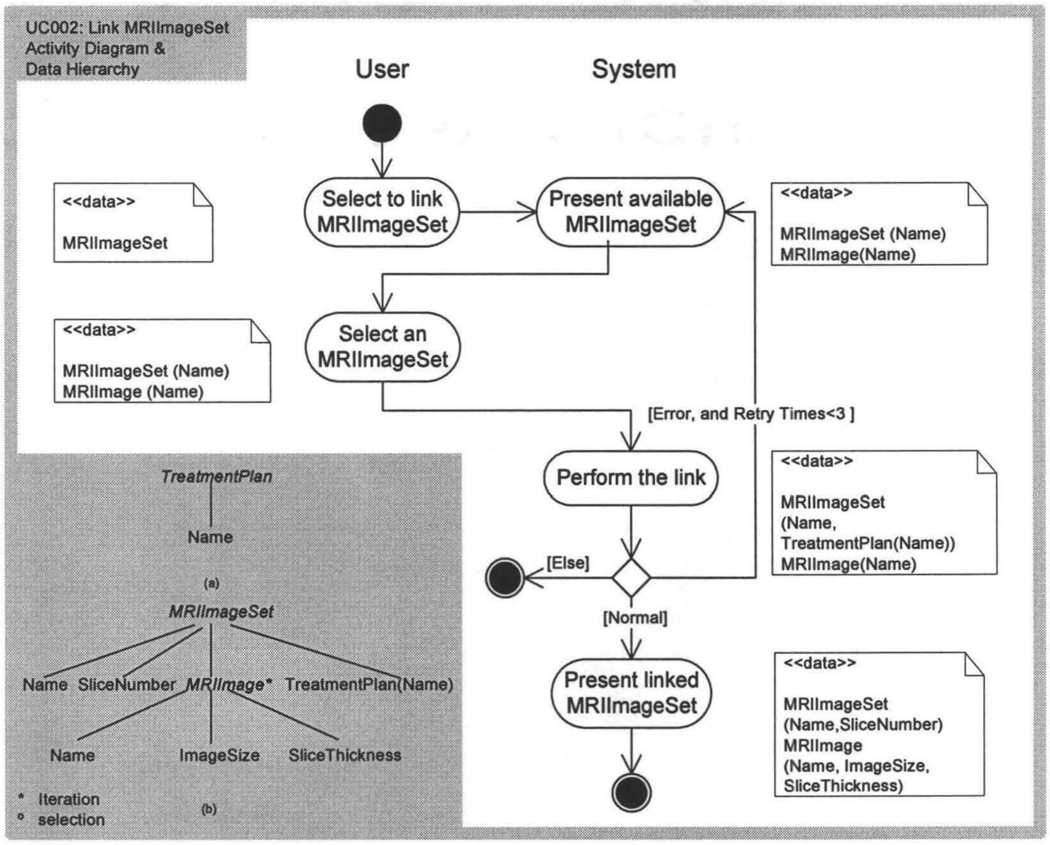

UC002: Link MRIImageSet Activity Diagram & Data Hierarchy
User
System
<<data>> MRIImageSet
Select to link MRIImageSet
Present available MRIImageSet
<<data>> MRIImageSet (Name) MRIImage(Name)
<<data>> MRIImageSet (Name) MRIImage (Name)
Select an MRIImageSet
[Error, and Retry Times<3]
Perform the link
<<data>> MRIImageSet (Name, TreatmentPlan(Name)) MRIImage(Name)
[Else]
[Normal]
Present linked MRIImageSet
<<data>> MRIImageSet (Name,SliceNumber) MRIImage (Name, ImageSize, SliceThickness)
TreatmentPlan
Name
(a)
MRIImageSet
Name
SliceNumber
MRIImage*
TreatmentPlan(Name)
Name
ImageSize
SliceThickness
* Iteration
° selection
(b)

Figure 3.23: Activity Diagram & Data Hierarchies of use case "Link MRIImageSet"

Good Object-Oriented Software System

Lower coupling between classes, or subsystems is one of the properties of good OO software. Although we are at the analysis phase, we need to understand successful architectural level design techniques, such as Model-View-Controller (MVC), client/supplier, multiple layers (tiers). Especially layers (in small sense, subsystems) help to reduce complexity by breaking the implementation up into more manageable chunks. As Bertrand Meyer states in [58]: "A serious software system, even a small one by today's standards, touches on so many areas that it would be impossible to guarantee its correctness by dealing with all components and properties on a single level. Instead, a layered approach is necessary, each layer relying on lower ones."

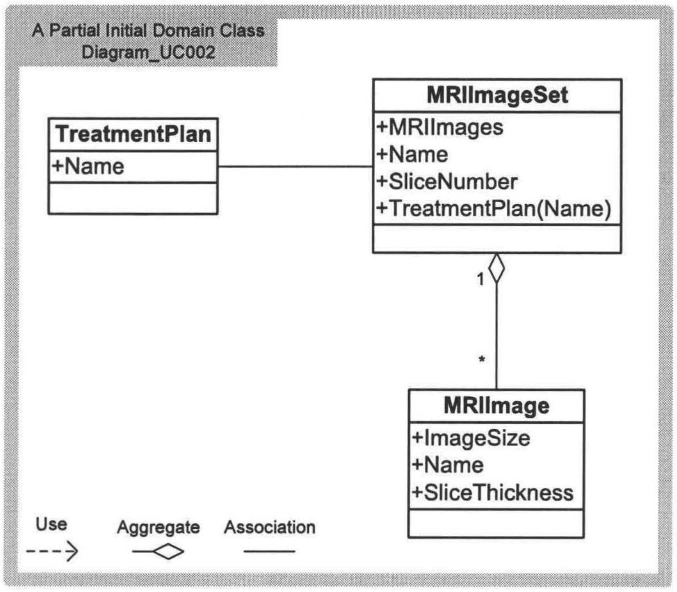


Figure 3.24: Initial Domain Class Diagram of use case "Link MRIImageSet"

At the requirements stage, we are describing externally observable behavior of the system. It is not the time to discuss the deep layers of the system. As a result, we have only two kinds of layers of classes to identify: domain classes layer (so called entity layer), and boundary classes layer (so called presentation layer). However, as the analysis model constitutes the basis of design, the layer principle still provides a consistent guide to the analyst for construction of the analysis model. When we allocate functions to classes, we should obey the layer principles where we can. For example, the boundary class layer should rely on the domain class layer, not vice versa.

Principles of Developing Sequence Diagrams

Another primary task of analysis in our approach is to find the boundary classes, functions of all analysis classes from the sequence diagrams, and to reach a definition of the software solution space - what must be done to fulfil the use cases. We choose a use case and realize it using at least one sequence diagram. During the realization of a use case, as the interactions go forward, we seek the needed objects and put them on the top of the diagram, along with their functions. Eventually, we get all the objects and functions required to carry out the use case.

Guidelines for Identifying Boundary Classes

- Each software system has a main user boundary class, named MainUB.
- Identify one central boundary class for each primary actor as its main boundary class. This is the place where the primary actor navigates required functions of the system.
- Identify a boundary class for each use case, if necessary.
- Identify boundary classes in which the primary actor needs to enter data, if necessary.
- Each different presentation of the system implies a boundary class.
- Identify one central boundary class for each external system actor (printers, terminals, alarm devices, sensors, DBMS systems, etc.).
- Boundary classes are reusable.

Guidelines for Drawing Sequence Diagrams

- Prepare at least one scenario per use case, usually the normal scenario.
- Abstract the scenarios into sequence diagrams.
- The first column is the actor who initiates the use case.
- The second column is a boundary object of the triggering actor.
- Domain objects are accessed by boundary objects.
- Domain objects never access boundary objects.
- Reuse boundary classes if applicable.

- Assign required functions to boundary classes or domain classes.
- Reuse functions of classes.
- Divide complex interactions if necessary.
- Prepare sequence diagrams for alternative conditions if necessary.

According to the guidelines above, we draw the sequence diagrams for use cases one by one. During the drawing of each use case, while we model the order of the interactions among the objects, we also assign services to each object in the form of functions (operations), namely, we decompose and distribute the functionality to participating objects. Again, we use the use case "Link MRIImageSet" as a realization example. Figure 3.25 depicts the realized sequence diagram.

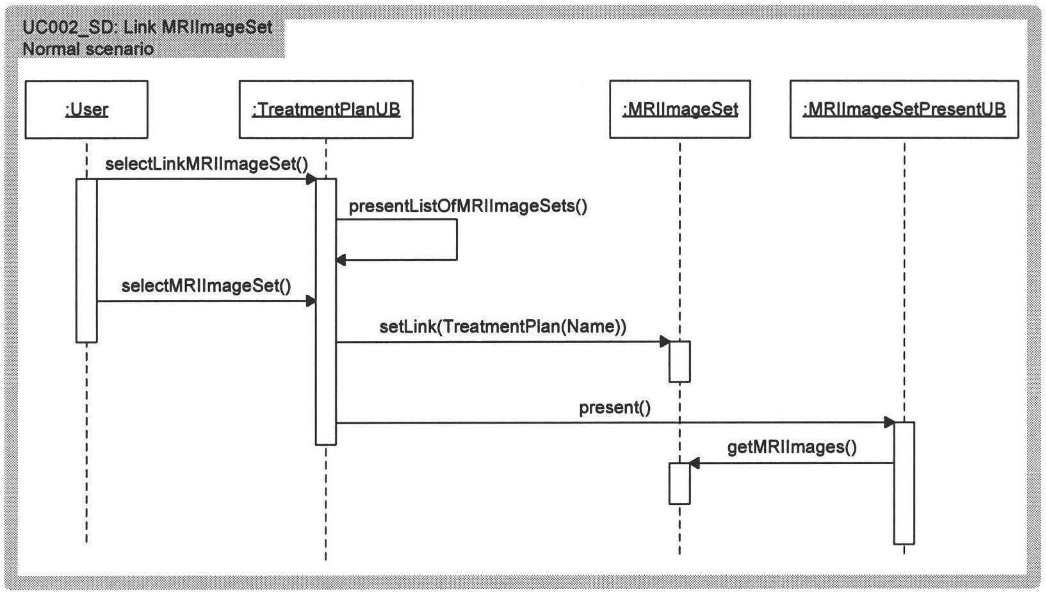


Figure 3.25: Sequence Diagram for the use case "Link MRIImageSet"

3.3.11 Identify Application (Boundary) Classes, Class Functions

When all the sequence diagrams are drawn, we arrive at a point to harvest boundary classes and their services, as well as the services of domain classes and newly found domain classes.

We know that sequence diagrams describe the interaction between objects. We then intuitively take for granted the semantics of sequence diagrams, and use a bottom-up approach to get the classes from the sequence diagram - those appearing at the top of the sequence diagrams. Similarly, messages transmitted between objects are services of the classes to which they belong - the functions of classes. Also, the associations between objects emerge automatically from those linking messages. All of the boundary classes, their attributes, and functions are captured in a *boundary class diagram*; similarly, domain classes are captured in a *final domain class diagram* which combines the initial domain class diagram.

We separate the domain class model and boundary class model because the domain class model is fairly stable during the development, also it is easy to trace back from the specification to analysis model. Figure 3.26 is the partial boundary class diagram captured from the sequence diagram Figure 3.25. Figure 3.27 is the partial final domain class diagram which also combines the initial domain class diagram in Figure 3.24.

Of course, as the requirements phase proceeds, the class diagrams can be drawn incrementally, or alternatively, they may be illustrated until all the sequence diagrams being finished.

3.3.12 Construct the Whole Analysis Class Diagram

A Class Model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects, and the attributes and functions for each class of objects. Class diagrams provide a graphic notation for modeling classes and their relationships, thereby describing possible objects. They are concise, easy to understand, and work reasonably well in practice. The combination of domain class diagram and boundary class diagram shows the whole static view of the system to be built. Figure 3.28 is a partial class diagram of the system.

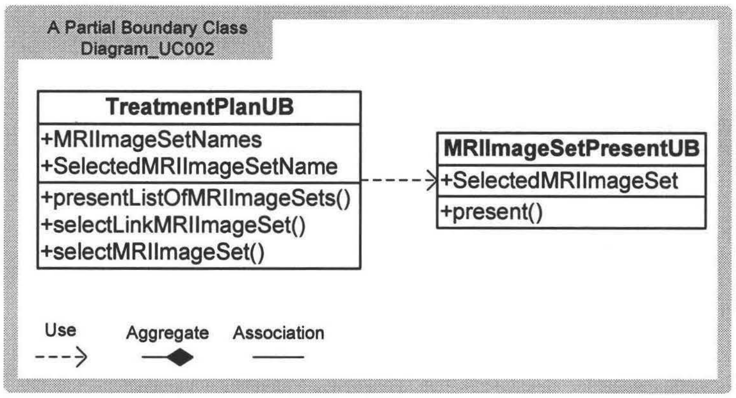


Figure 3.26: The Partial Boundary Class Diagram from Use Case "Link MRIImage-Set"

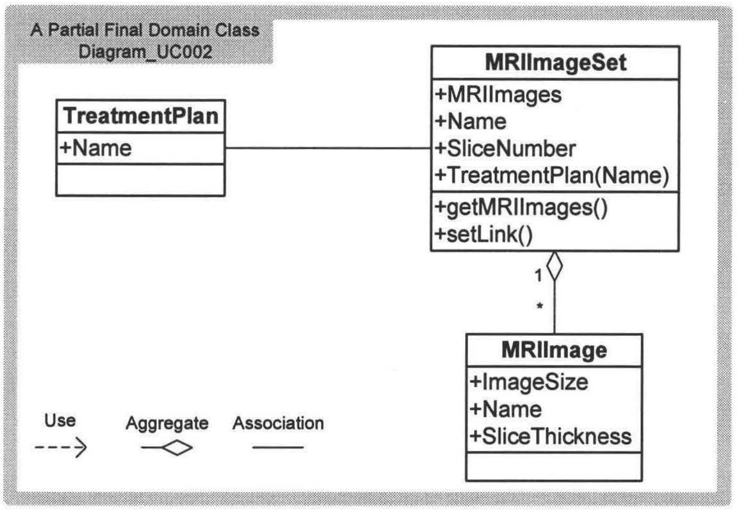


Figure 3.27: The Partial Final Domain Class Diagram from Use Case "Link MRIImageSet"

3.3.13 Analysis Model Document, AMD

The output of the analysis process is the analysis model which includes activity diagrams and data hierarchies, sequence diagrams, boundary class diagram, domain

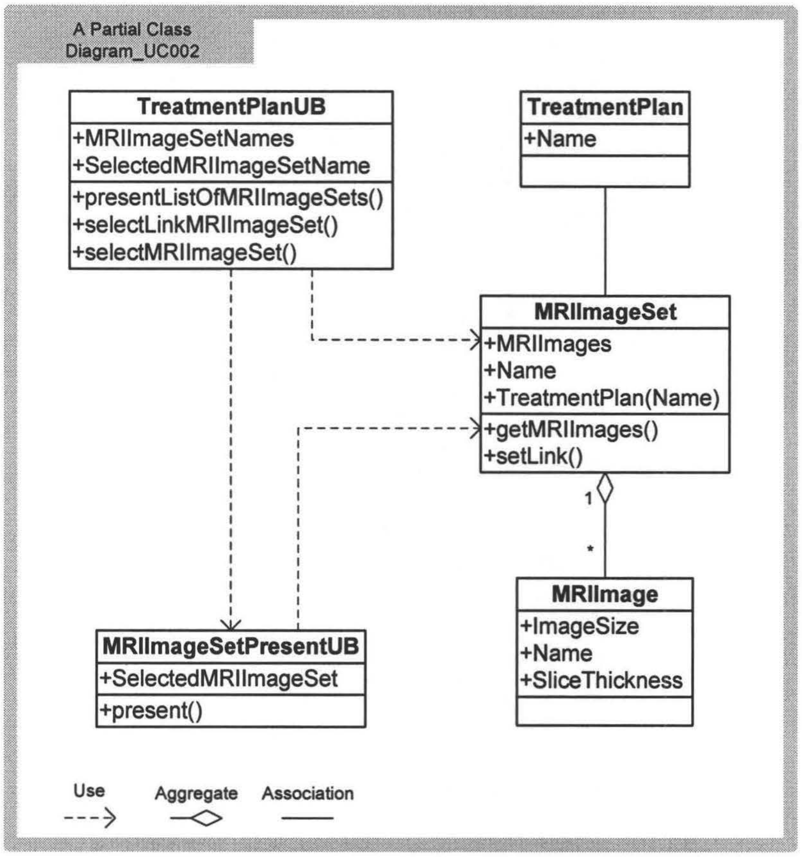
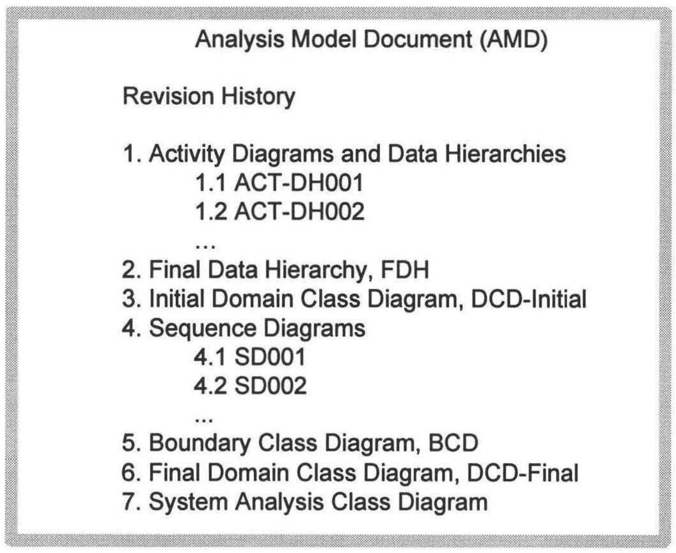


Figure 3.28: The Partial Class Diagram from Use Case "Link MRIImageSet"

class diagrams and system class diagram. All these individual models may be recorded as the appendix of the SRS or in an analysis model document. We suggest a template for this document in Figure 3.29.

Activity Diagrams and Data Hierarchies

This subsection captures activity diagrams and corresponding data hierarchies of each use case. We use ACT-DH00n to list all the activity diagrams and data hierarchies, where "ACT and DH" are acronyms of activity diagram and data hierarchy respectively, "n" corresponds to the use case number in URD UC00n.


Analysis Model Document (AMD)

Revision History

1. Activity Diagrams and Data Hierarchies
    1.1 ACT-DH001
    1.2 ACT-DH002
    ...
2. Final Data Hierarchy, FDH
3. Initial Domain Class Diagram, DCD-Initial
4. Sequence Diagrams
    4.1 SD001
    4.2 SD002
    ...
5. Boundary Class Diagram, BCD
6. Final Domain Class Diagram, DCD-Final
7. System Analysis Class Diagram

Figure 3.29: The Analysis Model Document Template

Reference URD 5. Use Cases, see sub section 3.2.5.6.

Final Data Hierarchy, FDH

This subsection lists the final data hierarchy that is combined by individual data hierarchies of all activity diagrams.
Reference AMD 1.

Initial Domain Class Diagram, DCD-Initial

This diagram captures the domain classes, their attributes, and relationships, where the DCD identifies domain class diagram. DCD-Initial is derived from the final data hierarchy.
Reference AMD 2.

Sequence Diagrams, SD

This subsection lists sequence diagrams of use cases, one for each. We use SD00n to organize the sequence diagrams, where "SD" stands for sequence diagram, "n" corresponds to the use case number in URD UC00n.
Reference URD 5. Use Cases.

Boundary Class Diagram, BCD

This Diagram depicts the boundary classes, their attributes, functions, and relationships.
Reference AMD 4.

Final Domain Class Diagram, DCD-Final

This Diagram lists the final version of domain classes, their attributes, functions, and relationships.
Reference AMD 2, 4.

System Analysis Class Diagram

This diagram illustrates the whole class diagram of the software to be built.
Reference AMD 5, 6.

3.3.14 More on Sequence Diagrams and Boundary Classes

What we want to emphasize hereto is the usage of sequence diagrams and boundary classes.

The sequence diagram model is widely used during the design phase to show the realization of use cases. Our usage of sequence diagrams also realizes use cases with objects of classes, but the main purpose is to seek the required boundary classes, required functions of the system and assign the functions to boundary classes and domain classes.

In our approach, we use the boundary classes in a flexible manner. As we discussed previously, boundary classes are abstractions of the data items of user interface(s), and abstractions of related functions of the system. Boundary classes can be composed, or decomposed for lower level purpose (UI design, or software design).

The next paragraph explains the flexibility of the usage of boundary classes. We still use the same example to illustrate the strategies. In Figure 3.30, 3.31 and 3.32, the sequence diagrams realize the same use case "Link MRIImageSet". All of the three diagrams contain the same 6 functions of the system. The differences are at the boundary classes, function allocations and function uses (who calls the functions). The sequence diagram in Figure 3.30 is the most abstract one, which is also our preference at the SRS stage. During the UI design activity, Figures 3.31 and 3.32 may be derived, and the boundary class TreatmentPlanUB in Figure 3.30 is decomposed into two boundary classes: TreatmentPlanUB and MRIImageSetSelectionUB. In Figures 3.31 and 3.32, the differences come from the function uses, that is, "who call the functions of other classes". Obviously, the sequence diagram in Figure 3.32 will lead to higher coupling among classes from the software designer's point of view.

At the step of drawing the sequence diagrams, the domain class model is already fairly stable. So, the functions of domain classes are fairly stable too, and so are the allocations of their functions.

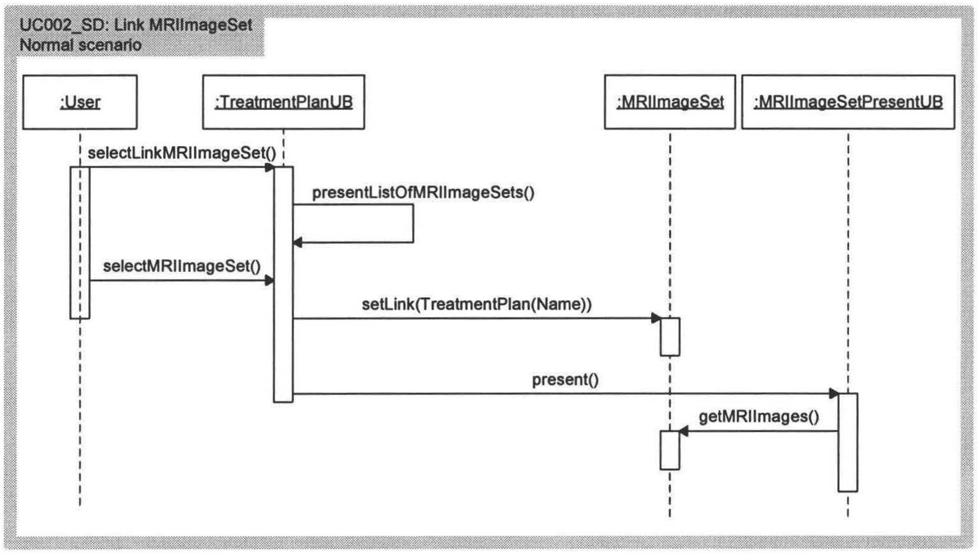


Figure 3.30: Sequence Diagram A at Requirements Level

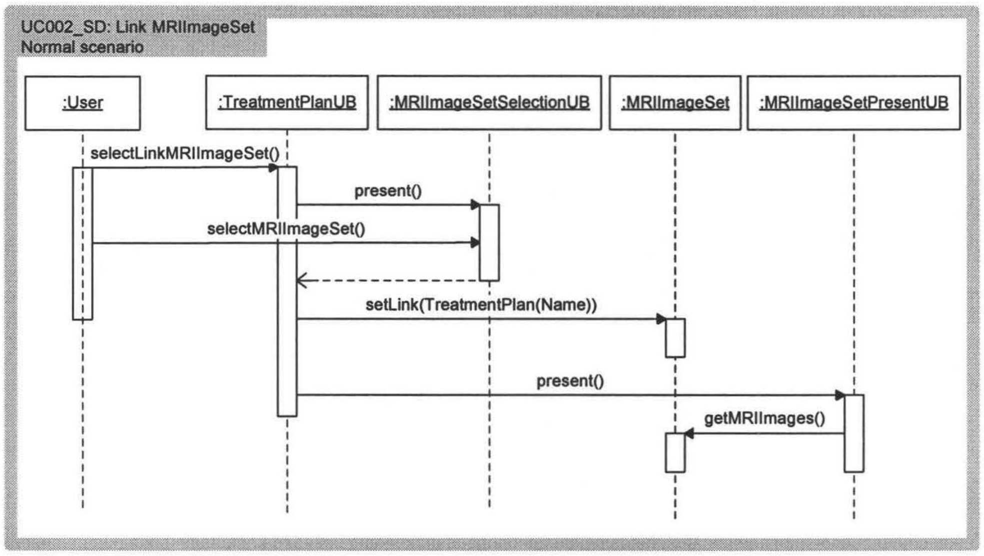


Figure 3.31: Sequence Diagram B at Requirements Level

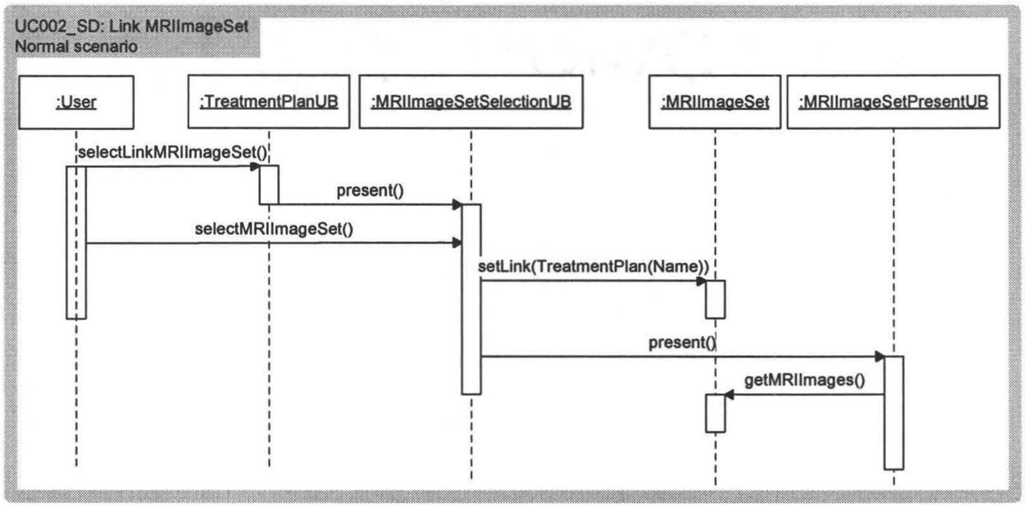


Figure 3.32: Sequence Diagram C at Requirements Level

3.4 Software Requirements Specification, SRS

Documenting the Software Requirements Specification (SRS) is the process of recording the results of the elicitation process and analysis process with various forms, including natural language, formal symbolic, or graphic representations. The SRS is the output of this process. The SRS unambiguously defines a software system behavior such that, given a problem, it specifies the functionality to carry out the user requirements.

3.4.1 Specify a Function of the System in the SRS

In mathematics, *function* means a mapping between two sets of elements (called *domain* and *range*, respectively) such that every element of the domain is mapped exactly onto one element in the range [67]. In the computer world, people usually call the domain of a function - the set of input(s), the range - the set of output(s).

At the requirements stage, a *function* is actually a function abstraction, which models some action of the system. An action application defines the state transition of the system, and each action has a precondition and post condition. So does a function.

In our approach, each function is specified with four clauses: *input* clause, *output* clause, *requires* clause and *ensures* clause. The input or output clause represents a vector of the related data items of the system, identifying the current state of the system. The requires and ensures clauses give, respectively, a precondition and postcondition for the function. A user's primitive action, e.g. a click, is regarded as a kind of input, because it abstracts a state of the system, such as the mouse position, click counter, button name etc. Moreover, we define any information displayed by the system to be a *presentation* type, which is considered to be a kind of output. In user-centric software systems, these kinds of descriptions are fairly effective to clearly capture the behavior of the system at requirements stage.

3.4.2 The Specification Language

We use natural language with structured forms as the specification language rather than other pure formal specification languages like B, Z, VDM, Petri Nets etc, because while the SRS should be precise and unambiguous, the maximum readability for domain readers should also be considered.

In addition to primitive data types, some other types are defined to precisely demonstrate the semantics of the input and output of a function. Also, we try to use some other simple notations to tidy the context. The following are some defined types and symbols:

User event - representing a user's primitive action.

Message - representing the invocation of a function.

Presentation - representing the presentation of any information.

[i] - representing the i^{th} element of a collection.

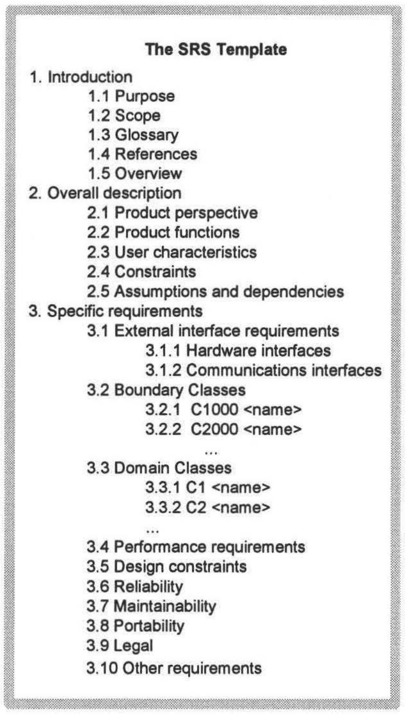
3.4.3 The SRS Template

We use a revised template from IEEE A4 [44] with tailored content, see Figure 3.33. In appendix A, the details of each category are described. The Software Requirements Specification (SRS) has three main parts: the introduction to the document, the overall description of the software to be built, and the specific requirements.

The introduction part states information about the SRS document itself. It includes a statement of the purpose of the SRS, a statement of the system's scope, a glossary used in the SRS, and a list of other referred documents, e.g. User Requirements Document. This part also contains an overview of the SRS document, which describes the structure of the SRS.

The overall description part describes an abstract and complete view of the system to be built. It has six sections: the product perspective, product functions, user characteristics, system constraints (general), assumptions and dependencies, and apportioning of requirements.

The specific requirements part constitutes most of the SRS. It is the part that we call software requirements, including functional requirements and system constraints (so called non-functional requirements).


The SRS Template

1. Introduction
    - 1.1 Purpose
    - 1.2 Scope
    - 1.3 Glossary
    - 1.4 References
    - 1.5 Overview
2. Overall description
    - 2.1 Product perspective
    - 2.2 Product functions
    - 2.3 User characteristics
    - 2.4 Constraints
    - 2.5 Assumptions and dependencies
3. Specific requirements
    - 3.1 External interface requirements
        - 3.1.1 Hardware interfaces
        - 3.1.2 Communications interfaces
    - 3.2 Boundary Classes
        - 3.2.1 C1000 <name>
        - 3.2.2 C2000 <name>
        - ...
    - 3.3 Domain Classes
        - 3.3.1 C1 <name>
        - 3.3.2 C2 <name>
        - ...
    - 3.4 Performance requirements
    - 3.5 Design constraints
    - 3.6 Reliability
    - 3.7 Maintainability
    - 3.8 Portability
    - 3.9 Legal
    - 3.10 Other requirements

Figure 3.33: The SRS Template

3.4.4 Specify Functional Requirements with Class Specifications

Based on our object-oriented analysis approach, we allocate the functions of the software system to different analysis classes. As a result, we specify the functional requirements with the category of analysis classes, and we call each category *a class specification*. All of the class specifications constitute the functional requirements of the SRS.

We also regard the specification of boundary classes as functional requirements, because at the requirements stage, boundary classes externally present both the key elements of interfaces and what the system will do.

To document the specification of each class, we combine some ideas from [13] and [26]. Also, we construct the class specification according to the following rules:

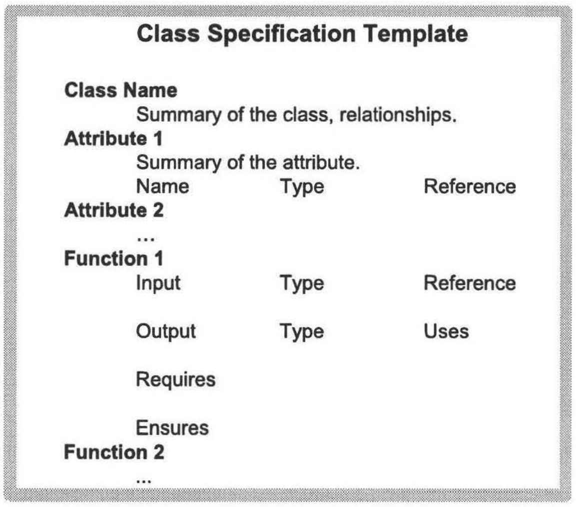
- Each class has an unique label "Cn", where C stands for class. For boundary classes, we let "n" start from 1000 and increment by 1000, for domain classes, we let "n" begin with 1 and increment by 1. We identify the classes from class diagrams together with other analysis models if necessary.
- For each attribute of a class, "Cn-Am" is labeled, where "Am" is the m-th attribute of the class "Cn". For each function of a class, "Cn-Fm" is labeled, where "Fm" stands for the m-th function of the class Cn.
- For each attribute of a class, the *type and reference* are identified, where reference usually refers to another class specification when a certain relationship exists, such as an aggregation or an association.
- For each function, requires, ensures, input, output and *uses* clauses are listed, where *uses* states the functions that are used by the current function. The contents of *uses* are identified when the output includes the *messages*.
- For each *uses* in the class functions, we get the used functions according to the sequence diagrams of use cases. This provides a dynamic view of the SRS and ensures the traceability and testability of the specification.
- Each alternative condition and corresponding output shall be documented in the *ensures* parts.
- To enhance the readability, the "object" is omitted when we talk about an object of an analysis class. For example, an MRIImageSet will mean an object of the MRIImageSet class.

Figure 3.34 demonstrates a suggested class specification template.

The following examples are identified from our analysis of the use case *Link MRIImageSet*, which has been used throughout this chapter, see its sequence diagram Figure 3.25 and class diagram Figure 3.28.

3.4.4.1 C2000, TreatmentPlanUB

The TreatmentPlanUB class shall provide the user the ability to navigate all the functions related to making a treatment plan, including *Enter PatientInfo & Target-*



Class Specification Template

Class Name
Summary of the class, relationships.
Attribute 1
Summary of the attribute.
Name Type Reference
Attribute 2
...
Function 1
Input Type Reference

Output Type Uses

Requires

Ensures
Function 2
...

Figure 3.34: The Class Specification Template

Info, Link MRIImageSet, Define Target, Set TreatmentOption, Do Simulation and Generate TreatmentPlanReport.

3.4.4.1.1 C2000-A1, MRIImageSetNames

A TreatmentPlanUB shall maintain a set of available MRIImageSet names to be chosen.

Attribute	**Type**	**Reference**
MRIImageSetNames	A set of strings	3.4.4.4

3.4.4.1.2 C2000-A2, SelectedMRIImageSetName

A TreatmentPlanUB shall maintain a selected MRIImageSet name, the default value is first one of the MRIImageSetNames or null.

Attribute	**Type**	**Reference**
SelectedMRIImageSetName	String	-

3.4.4.1.3 C2000-F1, SelectLinkMRIImageSet()

Input	**Type**	**Reference**
User selection	User event	-
Output	**Type**	**Uses**
Message of presenting the list of MRIImageSetNames	Message	3.4.4.1.4

Requires

A TreatmentPlan is active in the system.

Ensures

System shall perform PresentListOfMRIImageSets function.

3.4.4.1.4 C2000-F2, PresentListOfMRIImageSets()

Input	**Type**	**Reference**
MRIImageSetNames	A set of strings	-
Output	**Type**	**Uses**
A presentation of the MRIImageSetNames	Presentation	-

Requires

-

Ensures

If the MRIImageSetNames is not null, the system shall present a list of available MRIImageSet names, and set SelectedMRIImageSetName=MRIImageSetNames[1];

Else the system shall present the error information and stop current function.

3.4.4.1.5 C2000-F3, SelectMRIImageSet()

Input	**Type**	**Reference**
User selection	User event	-
Output	**Type**	**Uses**
SelectedMRIImageSetName	String	-
Message of setting the link of the selected MRIImageSet	Message	3.4.4.4.4
Message of presenting an MRIImageSetPresentUB	Message	3.4.4.2.2

Requires

The system presents a list of available MRIImageSet names (3.4.4.1.4).

Ensures

The system shall set SelectedMRIImageSetName=the selected MRIImageSet Name.

If the selected MRIImageSet is reachable and readable, the system shall perform the setLink function (3.4.4.4.4) and the presenting an MRIImageSetPresentUB function (3.4.4.2.2).

Else if the retried times < 3, the system shall allow the user to retry current function or choose to stop current function.

Else the system shall notify user the error information and stop current function.

3.4.4.2 C3000, MRIImageSetPresentUB

The MRIImageSetPresentUB class shall provide the ability to present the selected MRIImageSet.

3.4.4.2.1 C3000-A1, SelectedMRIImageSet

A MRIImageSetPresentUB shall maintain an MRIImageSet to be presented, the default value is null.

Attribute	Type	Reference
SelectedMRIImageSet	MRIImageSet	-

3.4.4.2.2 C3000-F1, Present()

Input	**Type**	**Reference**
An MRIImageSet(Name)	String	-
Output	**Type**	**Uses**
Message of getting the MRIImages from the selected MRIImageSet	Message	3.4.4.4.5
Presentation of the selected MRIImageSet	Presentation	-

Requires

The input MRIImageSet(Name) is not null.

Ensures

The system shall present the MRIImages[1] and the list of MRIImages of the selected MRIImageSet.

3.4.4.3 C1, TreatmentPlan

The TreatmentPlan class represents a treatment plan for a patient in the treatment planning software system. The system shall maintain a current TreatmentPlan for the current patient.

3.4.4.3.1 C1-A1, Name

The system shall maintain a TreatmentPlan name for each TreatmentPlan.

Attribute	**Type**	**Reference**
Name	String	-

3.4.4.4 C2, MRIImageSet

The MRIImageSet class represents the MRI image sets of patients that are managed by the treatment planning software system. Each MRIImageSet includes a collection of MRIImages.

3.4.4.4.1 C2-A1, TreatmentPlan(Name)

The system shall maintain a TreatmentPlan name for each MRIImageSet, the default value is null.

Attribute	**Type**	**Reference**
TreatmentPlan(Name)	String	3.4.4.3.1

3.4.4.4.2 C2-A2, Name

The system shall maintain a name for each MRIImageSet.

Attribute	**Type**	**Reference**
Name	String	-

3.4.4.4.3 C2-A3, MRIImages

Each MRIImageSet contains a set of MRIImages.

Attribute	**Type**	**Reference**
MRIImages	MRIImage	-

3.4.4.4.4 C2-F1, SetLink()

Input	**Type**	**Reference**
Current TreatmentPlan(Name)	String	
Output	**Type**	**Uses**
-		
Requires		
Current TreatmentPlan(Name) is not null.		
Ensures		
TreatmentPlan(Name) of Current MRIImageSet is set.		

3.4.4.4.5 C2-F2, GetMRIImages()

The system shall retrieve all of the MRIImages of an MRIImageSet.

Input	**Type**	**Reference**
-	-	
Output	**Type**	**Uses**
A set of MRIImages	MRIImage	-
Requires		
-		
Ensures		
All of the MRIImages of an MRIImageSet are retrieved.		

3.4.5 Specify System Constraints

The specific system constraints are transformed from the user requirements document (URD and user cases). They are specified according to the corresponding subsections in the SRS.

3.5 Software Requirements Validation

Software requirements validation is the process of checking that the software requirements of a software system fulfills its intended purpose, namely, the correct functionality for the solution system is defined.

Validation is different from verification. According to the Capability Maturity Model (CMMI-SW v1.1) [17], "Validation confirms that the product, as provided, will fulfill its intended use. In other words, validation ensures that 'you built the right thing'. Verification confirms that work products properly reflect the requirements specified for them. In other words, verification ensures that 'you built things right'."

We emphasize the process to be "validation" rather than "verification" in that the SRS is a definition of the solution of software, a view of user requirements at the aspect of developers. SRS just states what is to be built. The validation seeks to discover and correct any errors that occur during the requirements engineering phase, it aims to ensure that we are defining the correct functionality and building the right system. The following overall goals should be met:

- Scenario tables correctly and completely describe feasible user tasks.
- Use cases correctly record the user tasks.
- Software requirements are correctly derived from the use cases or from other origins.
- Software requirements are complete.
- All views of requirements are consistent.

Validation is accomplished through a variety of reviewing or testing procedures. It is not a separate phase and can occur at many levels and in different stages. The validation activities are threaded throughout the iterative elicitation, analysis and specification process. We recommend three levels of validation during the requirements phase: simple checks of scenario tables, storyboard of use cases, and formal review and prototype of SRS.

3.5.1 Why Requirements Validation?

"Software is not written, it is rewritten." - Everybody knows.

There is no error-free guarantee in human activities including software development, but at least we can reduce the probability of mistakes through certain processes and techniques.

Making changes early in the software development life cycle is extremely cost effective since there is nothing at that point to redo. If a project is changed after considerable work has been done then small changes could require large efforts to implement since software systems have many dependencies. According to statistics, it costs 68 to 110 times more to correct a requirement defect found by the customer after the system has been deployed than to fix an error found during requirements development [86]. According to [52], during requirements stage it needs 30 minutes to fix an error, whereas it will take 5 to 17 hours to correct a defect during the system testing phase.

Requirements validation can find the errors and ambiguities at the early requirement phase and thus reduces the system's life costs for maintenance.

3.5.2 Validation Techniques

Review, testing, and prototyping are the main techniques that can be used to validate an SRS. Some authors regard review as a form of testing [55], but we separate it from testing.

Review

Software review is "A process or meeting during which a software product is examined by project personnel, managers, users, customers, user representatives, or other interested parties for comment or approval" [41]. "Software product" means "any technical document or partial document, produced as a deliverable of a software development activity", and may include documents such as requirements documents, specifications, designs, source code, user documentation etc. Types of review include *management reviews, technical reviews, audits, inspections and walkthroughs.*

Reviews can be formal or informal. Informal reviews are conducted on an as-needed basis. The analyst chooses a review panel and provides and/or presents the material to be reviewed. The material may be as informal as a computer screen or hand-written documentation. Formal reviews are conducted at the end of each life cycle phase. The project manager appoints a formal review panel, who may make or affect a go/no-go decision to proceed to the next step of the life cycle. Formal reviews include the software requirements review, the software preliminary design review, the software critical design review and so on.

In software development, *peer reviews* are considered an industry best-practice for detecting software defects early and learning about software artifacts. Peer review refers to a type of software review in which a work product is examined by its author and/or one or more colleagues in order to evaluate its technical content and quality. Inspections and walkthroughs are two main forms of peer reviews.

A walkthrough is a form of peer review "in which the author leads members of the development team and other interested parties through a work product, and the participants ask questions and make comments about possible errors, violation of development standards, and other problems." [41] During walkthroughs, the author presents the work product in a step-by-step manner; a recorder notes all potential defects, decisions, and action items identified during the walkthrough meeting; a walkthrough leader conducts the walkthrough (and who is often the author). In general, a walkthrough has one or two broad objectives: to gain feedback about the technical quality or content of the document; and/or to familiarize the audience with the content. Walkthroughs can be informal or formal.

Inspections are peer reviews of any work product by trained individuals who look for defects using a well defined process. Like walkthroughs, inspections involve the line-by-line or step-by -step evaluation of the product being reviewed. Inspections, however, are significantly different from walkthroughs. In an inspection, a work product is selected for review and a team is gathered for an inspection meeting to review the work product: a moderator is chosen to moderate the meeting, one or more inspectors prepare for the meeting by reading the work product carefully and noting potential defects, a reader, who leads the team through the item, a recorder, who notes the faults, and the author, who helps explain the item being inspected. The goal of the inspection is to identify defects and for all of the inspectors to reach consensus on the work product. In most cases, inspections are formal.

IEEE 1028 defines a generic process for formal reviews based on the software inspection process originally developed by Michael Fagan [34].

1. Entry evaluation: The review leader (Moderator) uses a standard checklist of entry criteria to ensure that preconditions exist for a successful review.

2. Management preparation: Responsible management ensures that the review will be appropriately resourced with staff, time, materials, and tools, and will be conducted according to policies, standards, or other relevant criteria.

3. Planning: The review leader and the author identify the objectives of the review, organize a team of reviewers, and ensure that the team is equipped with all necessary resources for conducting the review.

4. Overview Meeting: In the meeting, the review leader ensures that all reviewers understand the review goals, the review procedures, the materials available to them, and the procedures for conducting the review.

5. Preparation: The reviewers individually prepare for group examination of the work product under review, by examining it carefully for defects according to a checklist of typical defects.

6. Inspection Meeting: The reviewers meet at a planned time to pool the results of their preparation activity and arrive at a consensus regarding the status of the document being reviewed.

7. Follow-up: The Author of the work product undertakes whatever actions are necessary to repair defects or otherwise satisfy the requirements agreed to at the inspection meeting. The review leader verifies that all action items are closed.

8. Exit evaluation: The review leader verifies that all activities necessary for successful review have been accomplished, and that all outputs appropriate to the type of review have been finalized.

At the requirements phase, a review, is a detailed examination of a document for the purpose of obtaining constructive criticism; particularly, the detection of errors

[13]. Both inspections and walkthroughs are involved during our validation process. They are conducted during and at the end of each process of the requirements life cycle to determine whether established requirements have been met.

Testing

Testing is the operation of the software product with real or simulated inputs to demonstrate that a product satisfies its requirements and, if it does not, to identify the specific differences between expected and actual results.

A test case is an input and an expected result [42]. In order to fully test that all the requirements of an application are met, there must be at least one test case for each requirement.

There are various levels of software tests, ranging from *unit testing*, which is based on minimal software components, *integration testing*, which is based on software architecture design, up to *software system testing*, which is based on functional requirements, and *acceptance testing*, which is based on user requirements [28, 68]. Testing plans and test cases should be developed and (conceptually) executed in each life cycle respectively.

During the requirements stage, although we can not run the test cases on the software to be built - as it has not been built yet, we still can create functional test cases based on use cases to find errors and ambiguities in our SRS and analysis models. We can conceptually run the test cases on the SRS and analysis models by walkthrough. Errors, omissions, ambiguities and nondeterminisms may be uncovered.

More requirements testing related topics can be found in [4, 9, 23]. Conceptual functional testing of software requirements is a powerful technique for finding errors at a very early stage of the project. It saves exponentially increased costs for fixing a requirement error not found until the coding stage.

Prototype

Prototype is defined in [45] as "something from which copies are made". Generally, a prototype is an original type, form, or instance of some thing serving as a typical example, basis, epitome, or standard for other things of the same category.

Unlike in other engineering disciplines, prototype in software engineering is a relatively recent phenomena from early 1970's [37]. In software engineering, prototypes provide the software developers with a "working model" for demonstration or use by customers, quality-assurance, business analysts, and managers to confirm or make changes to requirements, help define interfaces etc. Particularly, in software requirements engineering, prototypes demonstrate explicitly the behavior as defined in the SRS and make it obvious to users and clients whether their ideas and wishes have been correctly interpreted (or if their original thoughts need revision) [13]. Moreover, prototyping is, to date, the only effective way of developing user interfaces [78].

Prototyping has many variants in terms of types and forms, e.g., software prototype, GUI prototype (click dummy), paper prototype. Nowadays as the advent of powerful 4GLs (4th programming languages) and flexible CASE tools, software prototyping has become more feasible and more efficient. It is the primary format in prototyping. Further, prototypes can also be automatically generated from specifications. Usually this is based upon some formal specification techniques together with special software tools such as Statemate [39], SCR [40] method.

Software prototypes are constructed to visualize the system, or just part of it, in order to obtain feedback. A software prototype is a "quick and dirty" working model of the solution that presents a graphical user interface and simulates the system behavior for various user events. The information content is usually hard-coded rather than acquired from a database. Because a software prototype program runs like the real software, the users can easily judge the response behavior of the future system, compare the functionality with what they want, and determine if some functions are neglected.

There are two major types of software prototyping: *throwaway prototyping* and *evolutionary prototyping*. Throwaway prototyping refers to the creation of software that will eventually be discarded after the requirements phase. After preliminary requirements gathering is accomplished, a simple working model of the system is quickly constructed to visually show the users what their requirements may look like, from which users can re-examine their expectations and clarify their requirements early in the development of the software. This, in turn, leads to the accurate specification of requirements, and the subsequent construction of a valid and usable system from the user's perspective. When this has been achieved, the prototype model is "thrown away", and the system is formally developed based on the identified requirements specification.

Evolutionary Prototyping (also known as breadboard prototyping) targets to build robust software based on well-understood requirements so that the first version of the product can be released. An evolutionary prototype is usually an incomplete model of the whole system, and once built, it forms the heart of the new system, and the improvements and further requirements will be built on to it [25]. The partial system is sent to customer sites. As users work with the system, they detect opportunities for new features and give requests for these features to developers. Developers then change the software requirements specification, update the design, recode and retest. By a process of gradual refinement, the "prototype" becomes the product. Evolutionary prototyping is usually involved in certain software development approaches such as *Agile Software Development*.

The prototype process usually has the following steps [89]:

1. Identify basic requirements: determine basic requirements including the input and output information desired. Details, such as security, can typically be ignored.

2. Develop initial prototype: the initial prototype is developed that includes user interfaces and the basic functions.

3. Review: the customers, including end-users, examine the prototype and provide feedback on additions or changes.

4. Revise and enhance the prototype: using the feedback, both the specifications and the prototype can be improved. Negotiation about what is within the scope of the contract/product may be necessary. If changes are introduced then a repeat of steps 3 and 4 may be needed.

3.5.3 Validation Level 1: Simple Check of Scenario Tables

The goal of this level validation in our approach is to confirm that each user task (the problem domain behavior) is correct. This validation activity should be implemented during the task-workshops. Once each scenario table of a task is documented, it should be checked in the next workshop with the users. Checking points include: if any data is missed in each step? if any condition is neglected? are there any other alternative scenarios? Although we call this validation activity "simple check", it does not means that it is not important. Rather, it is of the first importance because the scenario tables are the basis of the later requirements processes.

3.5.4 Validation Level 2: Storyboard of Use Cases

The goal of this level validation is to confirm that the recorded requirements (use cases) are correct. Validating use cases helps to ensure that we have captured the right user requirements. We use storyboards because graphical views are always more understandable than textual forms.

A storyboard is a series of pictures, or diagrams, or screens from which a story is told. We use the activity diagram as the storyboard image type to validate the use cases on screen - the activity diagram should be developed by a UML tool like MS Visio, Rational Rose etc. This activity can start when all or part of the activity diagrams of use cases are drawn. Our goal is to communicate the use cases to the users to obtain a shared understanding of the functionality the software will provide, and to check for any errors and omissions. This validation is a kind of walkthrough and can be formal or informal.

Validation Guide:

- Prepare at least one storyboard for each use case.
- Use another storyboard to show subtask steps.
- Walk through the use case one step at a time; tell the story.
- Check each decision point, and walk through each alternative branch.
- Walk the user through all the functionality in the context of the use case.
- Record errors and omissions.
- Revise use cases and scenario tables, if applicable.

Storyboard Checklist:

- Is each use case a discrete user task?
- Is the use case written in the user's language?
- Are all alternative courses documented?
- Are there any common action sequences that could be split into separate use cases?
- Is the dialog sequence for each use case clearly written, unambiguous, and complete?
- Is every action step pertinent?
- Is each action step feasible?

3.5.5 Validation Level 3: Software Prototyping and Formal Review of SRS

The goal of this level validation is to confirm that the specification (system behavior) is correct.

3.5.5.1 Software Prototyping

The software prototyping can start when the class specification of the first use case is derived.

Prototyping Guide:

- Use case based prototype is built to implement each use case.

3.5.5.2 Formal Review of SRS

Formal review activity is conducted after the initial version of the SRS documentation is finished. We follow the process suggested by IEEE 1028 [41].

Participants.

The formal reviewer team includes the author (usually an analyst), a developer, a tester, a domain expert and the project manager.

Formal Review Entry Conditions.

The following is a list of entry conditions that must be satisfied before starting a formal review of the SRS.

- All other levels validation have been carried out.
- The documents conform to the proposed template.
- A spell check has been performed.
- The use case model and analysis model are available.

Inspection Checklist

The following checklist lists typical aspects for inspection:

Correctness	Are all use cases realized through a sequence diagram?
	Are all requirements in the SRS also in the analysis model?
Unambiguousness	Is there any use of the term "may", "might", "should" etc. to describe the system's behavior?
	Are there any uses of other terms that are vague?
Completeness	Are all attributes and methods included in the analysis model also specified in the SRS?
	Are all conditions and constraints on functional requirements specified in the SRS?
	Are all references fully defined?
	Are all uncommon terms defined?
Consistency	Is each attribute defined once?
	Is each method defined once?
Verifiability	Are all quantities specified in measurable terms?
	Can test cases be defined for each use case scenario?
Modifiability	Does the structure of the SRS match the structure of the analysis model?
	Have all redundant structures been removed?
Traceability	Is each requirement uniquely numbered in a manner that allows the attribute or method to be associated with its class?
	Is each requirement traced to the use cases it helps realize?

Formal Review Exit Conditions.

The following is a list of exit conditions that must be satisfied before ending the formal review of the SRS.

- All defects detected have been recorded and analyzed, and the SRS revised, accordingly.
- The revised documents have been spell checked.
- The documents have been "versioned" and checked into the configuration management system.

3.6 Our Practical Approach and "High Quality SRS"

We say the approach is practical because of its easy understandability and applicability for users. Also, our approach meets the characteristics of general engineering principles, i.e., it is systematic and iterative. For each activity of each process, we not only work out the step-by-step guidelines, but also analyze the underlying rationale. As a result, our approach will help to lead to high quality SRSs.

3.6.1 High Quality SRS

According to [28, 44] etc, a high quality SRS should be correct, unambiguous, complete, consistent, modifiable, verifiable, and traceable.

Correct

An SRS is correct if, and only if, every requirement stated therein represents something required of the system to be built.

Unambiguous

An SRS is unambiguous if, and only if, every requirement stated therein has only one interpretation.

Complete

An SRS is complete if, only if, it includes the following elements: all functional requirements, system constraints, definitions of the responses to all kinds of input data in any situation, full labels and references to all figures, tables, and diagrams, definitions of all terms and units of measure.

Consistent

An SRS is consistent if, and only if, no subset of requirements stated therein conflict, such as conflicting behavior, conflicting terms.

Modifiable

An SRS is modifiable if its structure and style are such that any necessary changes to the requirements can be made easily and still remain relevant attributes of the SRS.

Verifiable

An SRS is verifiable if, and only if, every requirement stated therein is verifiable. A requirement is verifiable if, and only if, there exists some finite cost-effective process with which a person or machine can check that the software product meets the requirement.

Traceable

An SRS is traceable if the origin of each of its requirements is clear and if it facilitates the referencing of each requirement in future development or enhancement documentation. Backward traceability ensures each requirement explicitly references its source in earlier documents. Forward traceability demands each requirement having a unique name or reference number.

3.6.2 Our Approach Leads to a High Quality SRS

In our SRS, the structure of functional requirements directly maps the class diagrams which show the static structure of the software system. Meanwhile, the dynamic message flows are identified as output references (uses) in functions which match the sequence diagrams.

Our approach will help to lead to a high quality SRS. On one hand, the SRS is directly derived from the analysis model, and on the other hand, the analysis model is the realization of use cases, which capture the real use requirements. This process greatly facilitates developing a correct and complete SRS.

Correctness is enhanced, in that every functional requirement in the SRS is one the software must implement - it originates from the user requirements.

Completeness is also enhanced, in that all necessary functions are included. We realize all use cases in analysis models - at least one sequence diagram for each use case, this can help ensure that we have accurate and complete required analysis classes for each use case. During the realization, needed functions are also allocated to analysis classes, and eventually they are identified in the SRS. As a result, the SRS includes and only includes the required functionality required by the user requirements.

Consistency is aided, in that according to the specifying rules, each attribute and each function will have a unique number in the SRS.

Verifiability is aided, in that functional test cases can be walked through courses of each use case.

Modifiability and traceability are aided, in that any change in any place can be traced forward or backward among the use cases, the analysis models and the SRS.

Chapter 4

A Case Study: PDT Treatment Planning Software

In this chapter, we use an example of treatment planning software as a case study to explain the application of our requirements specification approach. A prototype of such software was made available to us by the University Health Network (UHN), and we have agreed not to disclose details of this system while it is under development.

4.1 Apply Elicitation Process

4.1.1 Resources

Training Manual of Existing Software

An introductory-level user manual of the existing treatment plan software, provides an informal overview of how to use the software to produce treatment plans [16].

UHN Clinical Trials Treatment Planning Report

This report briefly explains the underlying principles of the PDT treatment planning software, including the PDT treatment principle, threshold model of treatment planning and computational model of the simulation. This report also briefly describes the process of how to implement PDT treatment planning for prostate cancer patients based on the current software.

Other Sources

There are many websites explaining the general principles of PDT, such as [60, 61, 88]. In terms of similarity with the UHN system, the following website can be explored: http://www.prostatepdt.com/.

Numerous papers also provide the domain knowledge and theoretical basis of PDT. Some recent ones are [8, 32, 85].

4.1.2 Understanding of Problem Description

There is no SRS for the current UHN treatment planning software. However, based on the available documents and other sources, we can get information that is as detailed as that we would get from the normal elicitation processes like interviews, workshops etc, and is sufficient for the purpose of applying our approach.

Background Description

PDT uses laser, or other light sources, combined with a light-sensitive drug (sometimes called a photosensitizing agent, or photosensitizer) to destroy cancer cells. Figure 4.1 [3] visually shows how a PDT treatment is performed for the prostate cancer. The prostate template is drilled with a 0.5-cm equal spaced grid. Cylindrical diffusing fibers (CDF) are inserted into the catheters to illuminate the entire prostate gland.

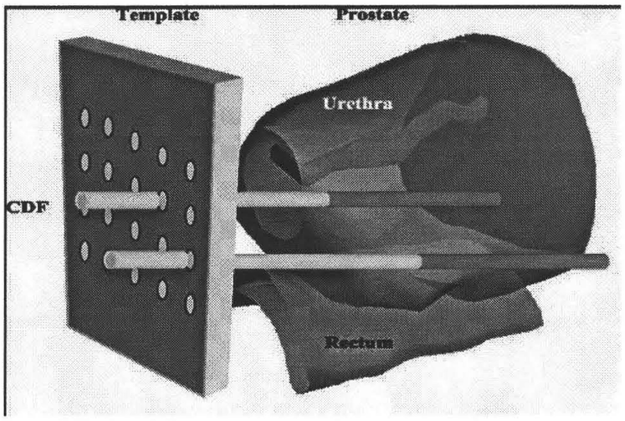


Figure 4.1: The PDT Treatment for Prostate Cancer

A photosensitizing agent is a drug that makes cells more sensitive to certain wavelength light. After injection into the blood stream, the drug is attracted to cancer cells. It does not do anything until it is exposed to a particular type of light. When a sufficient amount of light is directed at the area of the cancer, the drug is activated and produces the aggressive singlet oxygen which leads to death of cancer cells. Some healthy, normal cells in the body will also be affected by PDT, although these cells will usually heal after the treatment.

In our case study, the PDT treatment aims for radical treatment of tissue volumes (e.g. prostate cancer, brain tumors), where a significant treatment depth is required or in which multiple interstitial diffusing optical treatment devices (e.g. fiber sources) are used to achieve complete irradiation. To spare adjacent normal tissues from side-effects, a detailed treatment plan is needed for each treatment.

The solution lies in the treatment planning software, namely, the application for calculation of the effective PDT dose delivered for a specified set of treatment parameters (device number, position, depth, energy, time duration etc.) on specific tissues. In a treatment planning software system, a set of images is loaded into the system, Figure 4.2 (a) [85]; the target tissue (black line in Figure 4.2 (b) [85]) and its adjacent tissues are traced and the virtual treatment devices are set (the five black dots); and then the software calculates the light distribution throughout the tissues. Led by visually simulated treatment results, Figure 4.2 (b) [85] (threshold dose contour, the white line), users change the parameters to reach the optimized option delivering a balance between effectiveness and safety.

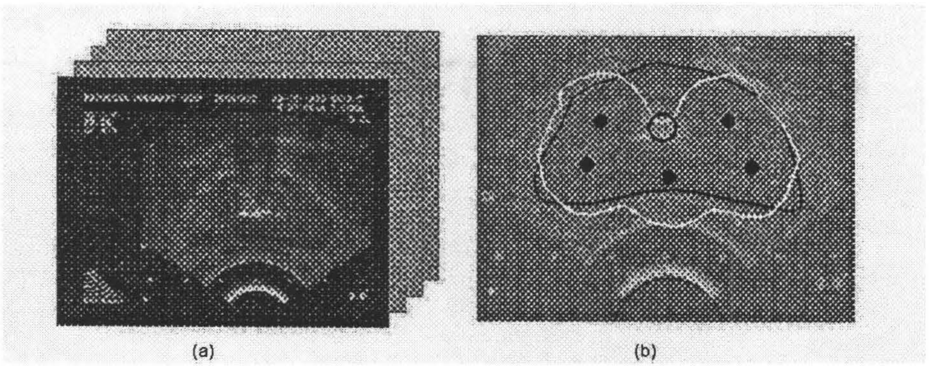


Figure 4.2: The Prostate Example: PDT Treatment Planning Software

Source (delivery fiber) array : $energy = power \times illumination\ time$

Name	Unit	Working Value	Comments	Related
number of fibers				light dose
length				light dose
positions within the target				light dose
power to each fiber	mW/cm			light dose
illumination time				light dose
wavelength	nm	700.00		PDT dose

Tissue optical properties:

absorption coefficient	cm^{-1}	average values	trial specific	light dose
scattering coefficient	cm^{-1}	average values	trial specific	light dose

Drug concentration: $drug\ concentration = delivered\ dose \times specific\ uptake\ ratio$

extinction coefficient	$cm^{-1}(\mu/g)^{-1}$	drug specific, unknown		PDT dose
delivered dose	mg/kg	tissue specific		PDT dose
specific uptake ratio		tissue specific, unknown		PDT dose
photobleaching rate	$cm^2 J^{-1}$	tissue specific, unknown		PDT dose

Threshold Dose

light threshold dose	J/cm^2			light dose
PDT threshold dose	$photons/cm^3$			PDT dose

Table 4.1: Data Involved in a PDT Treatment

Current Situation

Using the current treatment planning software platform, UHN can make treatment plans for prostate cancer patients who had prior radiotherapy.

A light dose, also referred to as light fluence, represents the amount of light received at a given location in tissue. It is a measure of the amount of the light energy that passes through the location and is given in units of fluence (J/cm^2).

A PDT dose at a certain location in tissue is calculated by combining the light dose, the drug concentration and local oxygenation. It is expressed in photons absorbed by the PDT drug per cm^3.

In a real treatment, a PDT dose calculation is affected by many factors which are listed below, see Table 4.1.

Since some parameters for calculating the PDT dose are unknown, such as Extinction Coefficient and the specific Uptake Ratio for the photosensitizing agent in the prostate, it is impossible to calculate the real PDT dose. As a result, the current software calculates Light Dose to evaluate each treatment option for treatment clinical plans.

The following assumptions are applied in the current treatment planning software.

- Oxygen-independent: an unlimited oxygen supply to the tissues.
- Drug-independent: constant and homogeneous photosensitizer concentration.

Vision Statement

Treatment plan software is required for simulating treatment of cancer (e.g. prostate cancer) patients and for producing treatment plans for them. For each patient, the baseline MRI images will be loaded into the software and the important structures will be defined (e.g. prostate, rectum and urethra). Then, a virtual array of cylindrically diffusing optical treatment devices (e.g. fibers) is added to the virtual target volume. Once a set of treatment parameters (e.g. device numbers, energy, illumination time etc.) are defined, the light dose distribution both inside the treatment volume and in its surroundings can be calculated by the software and the calculated results can also be visualized by superimposing the treatment effect onto the MRI images. Simulated treatment results are evaluated through the contours of light threshold dose of treatment volume and surrounding normal tissues. The treatment parameters are changed iteratively until an acceptable balance between efficacy and safety is achieved. This determines a near optimal set of treatment parameters. Eventually, a treatment plan report is created for guidance of clinical trial treatment, which includes the selected treatment option for the patient, and images showing the positions of the treatment devices and the target structures.

User Responsibilities and Customer Authorities

Basically the customers of treatment planning software will be hospitals and their authorities are listed in Table 4.2. According to current documents, there are two kinds of users, namely, radiologist and planner. Their responsibilities and tasks are listed in table 4.3. For simplicity, we will omit the administrator user type of the system.

Customer	Authorities
Hospital	Running the Software for Treatment Planning

Table 4.2: Customer Authorities: Case Study

User	Responsibilities	Tasks
Radiologist	Ensuring Correct Target Definition	Enter PatientInfo & TargetInfo Link MRIImageSet Define Target
Planner	Making Treatment Plan	Enter PatientInfo & TargetInfo Link MRIImageSet Define Target Set TreatmentOption Do Simulation Generate TreatmentPlanReport

Table 4.3: User's Responsibilities and Tasks: Case study

Glossary

This part is eventually transferred into the SRS, see the complete glossary in appendix B.1.3. The following are some typical examples:

MRIImageSet	=(Name, MRIImages, SliceNumber, TreatmentPlan(Name)) //A series of MRI Images for certain patient, usually the T2-weight series is used for treatment planning//.
MRIImage	=(Name, Points, SliceThickness, ImageSize) //A slice of image scanned by Magnetic resonance imaging (MRI) Machine.//
PatientInfo	=(Name, PhysicianName) //A patient whose treatment plan is produced by the treatment planning software.//
TreatmentOption	=(Number, TreatmentDeviceArray) //A treatment option is a set of treatment parameters.//
TreatmentDeviceArray	=(Name, TreatmentDevices, DeviceNumber) //An array of treatment devices//
TreatmentDevice	=(Name, Label, Length, Power, IlluminationTime, Position) //A light delivery source//

System Constraints

Table 4.4 describes the general system constraints known at this time.

Usability	Users are familiar with GUIs. Users have experience using Windows application.
Reliability (robustness, safety and security)	Authentication is required.
Performance	
Maintainability	
Portability	The software will run only on MS windows platform.
Implementation	
Interface	
Operation	
Packaging	
Legal	FDA standard.

Table 4.4: System Constraints: Case Study

Problem Context Diagram

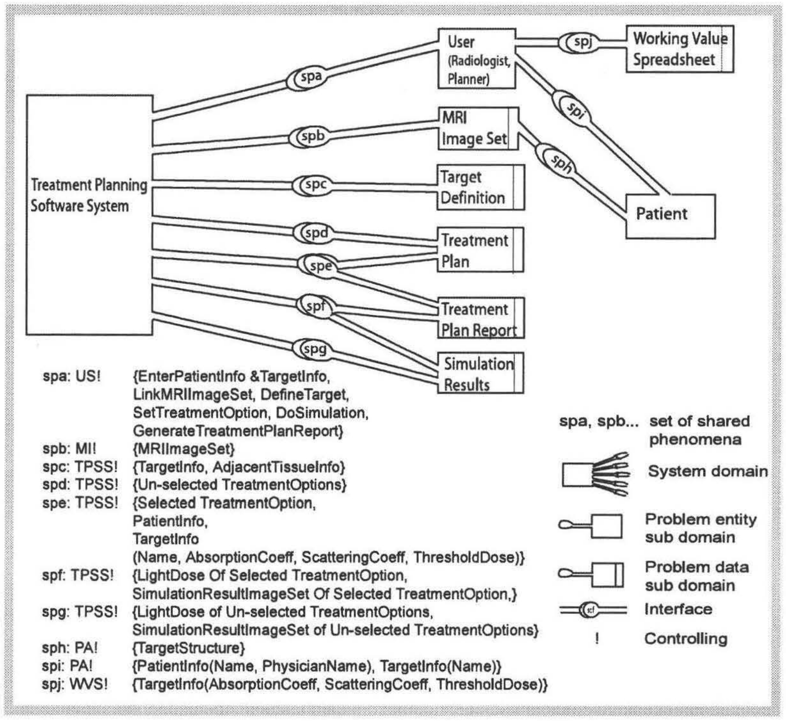
The advantages of drawing a problem context diagram are obvious. While a problem context diagram shows our understanding of the problem in an easily understandable way, it is also used to derive a traditional context diagram, which provides a basis for later stages. Figure 4.3 is the problem context diagram of our example.

In this diagram, we do not list all the problem sub domains, such as there exists an MRI Machine sub domain between the MRI Image set and Patient. Since the appearance of the MRI machine has no significant meaning for our problem, we regard these kinds of sub domain as *connection sub domains*. To simplify the diagram, connection sub domains can be omitted - if we always assume they work properly, they are transparent to the connected sub domains.

Also, we abstract the two user-type sub domains *Radiologist* and *Planner* to be one sub domain *User*, because they perform similar tasks and we do not care now the division of labor between them. However, you can separate them if you prefer.

4.1.3 System Context Diagram

A system context diagram is derived directly from the problem context diagram according to our stated rules, see Figure 4.4



spa: US!	{EnterPatientInfo &TargetInfo, LinkMRIImageSet, DefineTarget, SetTreatmentOption, DoSimulation, GenerateTreatmentPlanReport}
spb: MI!	{MRIImageSet}
spc: TPSS!	{TargetInfo, AdjacentTissueInfo}
spd: TPSS!	{Un-selected TreatmentOptions}
spe: TPSS!	{Selected TreatmentOption, PatientInfo, TargetInfo (Name, AbsorptionCoeff, ScatteringCoeff, ThresholdDose)}
spf: TPSS!	{LightDose Of Selected TreatmentOption, SimulationResultImageSet Of Selected TreatmentOption,}
spg: TPSS!	{LightDose of Un-selected TreatmentOptions, SimulationResultImageSet of Un-selected TreatmentOptions}
sph: PA!	{TargetStructure}
spi: PA!	{PatientInfo(Name, PhysicianName), TargetInfo(Name)}
spj: WVS!	{TargetInfo(AbsorptionCoeff, ScatteringCoeff, ThresholdDose)}

Figure 4.3: Treatment Planning Software Problem Context Diagram: Case Study

4.1.4 Actors and Their Profiles

All the things surrounding the system are actors, i.e. we regard any sub domain that needs information from or provides information to the system as an actor. So in our system context diagram, the surrounding actors include not only the external entities, but also those data sub domains - you can think of them as some form of physical representations or anything else that contains the data information. This conception is a little different from that of the context diagrams which people usually draw, they only include external entities, such as devices, people, and other systems. We prefer this conception in that we do not hope to restrict the future design - the data could be contained in any sources (network, local disk, etc).

Although in the context diagram we abstract the two user types *Radiologist* and *Planner* to be one actor *User*, their profiles should be given separately. Table 4.5 lists all actors and their profiles.

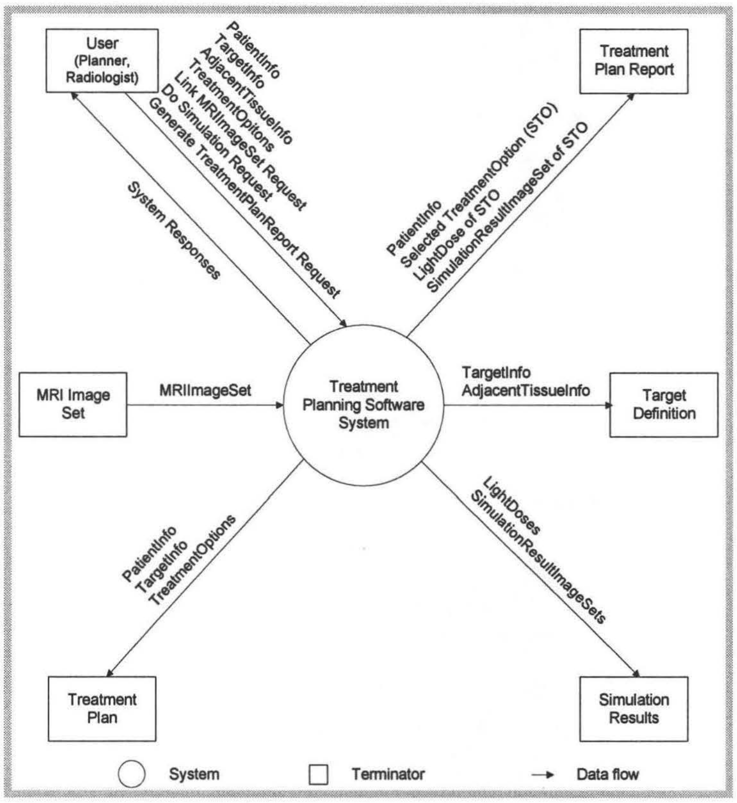


Figure 4.4: Treatment Planning Software System Context Diagram: Case Study

Actor	Profile: Background and Skills
User-Radiologist	A physician specializing in diagnostic techniques for viewing internal organs and tissues without surgery. Radiological methods include X-ray, MRI, computed tomography (CT),scan, ultrasound, angiography, and nuclear isotopes.
User-Planner	Beginner-level or expertise treatment planner
MRI Image Set	Storage or source of a serials of MRI Image slices of a target.
Target Definition	Storage of definitions of anatomical structures inside the target.
Treatment Options	Storage of patient information and sets of treatment parameters.
Treatment Plan Report	Storage of Treatment Plan Report.
Simulation Result	Storage of simulation results including light dose calculating results and images of simulating results.

Table 4.5: Profiles of Actors: Case Study

Primary Actor	Tasks
User (Radiologist, Planner)	Enter PatienInfo & TargetInfo
User (Radiologist, Planner)	Link MRIImageSet
User (Radiologist, Planner)	Define Target
User (Planner)	Set TreatmentOption
User (Planner)	Do Simulation
User (Planner)	Generate TreatmentPlanReport

Table 4.6: The Primary Actor-Task List: Case Study

4.1.5 Primary Actor - Task List

As discussed before, each user type is a primary actor. In some cases, a user type may perform some same tasks as another user type, for example, both the Radiologist and the Planner perform Enter Patient Information task. In this case, the radiologist and planner play the same primary actor role of the same task.

If you draw the problem context diagram according to the rules, and subsequently derive the system context diagram based on that, you eventually get all the actors. Each of the primary actors that you get may include more than one user types, for example, the *User* primary actor combines both the *Radiologist* and the *Planner* user types. That is the reason why we capture the *Primary Actor - Task* list rather than the *User Type - Task* list. We should avoid the duplication, but still we need to clarify the differences between the user types, for example, a radiologist performs less tasks than a planner. Table 4.6 lists what we can get in the treatment planning software example.

4.1.6 Scenario Tables of User Tasks

Based on the documents, we can get the scenario table of each task respectively. See Tables 4.7, 4.8, 4.9, 4.10, 4.11, 4.12, and 4.13.

Task: Enter PatientInfo & TargetInfo

Task summary: The user intends to enter the information of a patient and his/her being-treated target in order to make a new TreatmentPlan for that patient.

Precondition: User is logged in.

Success post condition: 1. Entered PatientInfo and TargetInfo are added into the system.
2. A new TreatmentPlan is created and stored with some initial data values.
3. A blank TreatmentOption is created and stored with entered TreatmentOption(Number), which is the active TreatmentOption of current TreatmentPlan.

Failure post condition: No stored data will change within the system.

System constraints: None.

1	2	3	4
User selects to make TreatmentPlan.	System responds by presenting required data items with default values where necessary, including PatientInfo(Name, PhysicianName), TargetInfo(Name, AbsorptionCoeff, ScatteringCoeff, ThresholdDose) and TreatmentOption(Number).	User completes all data items.	*No TreatmentPlan already exists for this patient:* System creates and stores a new TreatmentPlan for this patient with some initial data values: Name, entered PatientInfo(Name), entered TargetInfo(Name), a blank TreatmentOption with entered TreatmentOption(Number). System also stores entered data of PatientInfo, TargetInfo and TreatmentOption.
▨	▨	▨	*4a. TreatmentPlan already exists:* .1 System informs user that a TreatmentPlan already exists for this patient, and asks user to use the entered data for a new TreatmentPlan, or revise the current input, or cancel the task. .2a User selects to use the entered data for a new TreatmentPlan: system performs normal step 4 neglecting the normal condition. .2b User selects to revise the current input and revises the PatientInfo(Name): task returns to normal step 3. .2c User selects to cancel the task: task fails.

Table 4.7: The Scenario Table of Task: Enter PatientInfo & TargetInfo

Task: Link MRIImageSet

Task summary: The user chooses to link an MRIImageSet to the current patient' current treatment plan.

Precondition: 1. User has selected to make TreatmentPlan for this patient.

2. A new TreatmentPlan for current patient with initial data values has been created. Ref. Table 4.7

3. Required PatientInfo and TargetInfo have been stored. Ref. Table 4.7

Success post condition: An MRIImageSet is linked to current TreatmentPlan and presented..

Failure post condition: None.

System constraints: 1. Successive retries of the same MRIImageSet selection can be executed at most three times.

1	2	3	4
User selects to link a MRIImageSet.	*Number of available MRIImageSets* $\geq$ *1:* System presents a list of names of available MRIImageSets.	User selects an MRIImageSet of current patient.	*Selection of MRIImageSet is successful & Number of retried times* $\leq$ *3:* System links the selected MRIImageSet to current TreatmentPlan and presents the selected MRIImageSet, including presentation of the first MRIImage slice of the image set and list of MRIImage slices in the set.
	2a. Number of available MRIImageSets $<$ *1:* .1 System informs user. .2 Task fails.		
			4a. Selection of MRIImageSet is not successful & Number of retried times $<$ *3:* .1 System informs user that the selection of current MRIImageSet fails, and asks user to retry the current selection, or try another selection, or cancel the task. .2a User selects to retry current selection: system performs normal step 4. .2b User selects to retry another selection: system returns to normal step 2. .2c User selects to cancel: task fails.
			4b. Selection of MRIImageSet is not successful & Number of Retried times $\geq$*3:* .1 System notifies user of failure. .2 Task fails.

Table 4.8: The Scenario Table of Task: Link MRIImageSet.

Task: Define Target

Task summary: The user intends to define the target on a linked MRIImageSet of current patient for the treatment simulation, which includes the definition of target volume structure (TargetInfo(Tracings)) and adjacent normal tissue structures (AdjacentTissueInfo(Tracings)).

Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. A MRIImageSet has been linked to current patient. Ref. Table 4.8

Success post condition: The target volume structure and adjacent normal tissue structures are defined.

Failure post condition: No stored data will change in the system.

System constraints: None.

1	2	3	4	5	6
Uscr sclccts to dcfinc target.	System responds by presenting a list of names of structures that need to be defined, including being-treated target volume and adjacent normal tissues.	User selects the target volume or an adjacent normal tissue to define.	System presents the linked MRIImageSet of current patient, including presentation of the first MRIImage slice and list of MRIImage slices in the set; and other required data items with default values where necessary, including: TargetInfo(Name, VolumeValue) or AdjacentTissueInfo(Name, VolumeValue), and Tracing(PointsNumber, AreaValue) on current MRIImage slice.	User traces the Tracing(Loops) of selected structure by selecting points to form loops on each MRIImage slice where it is perceivable (visible).	*All Tracings are acceptable: (all loops are closed)* System stores Tracings to a TargetInfo(Tracings) or AdjacentTissue-Info(Tracings). System returns to normal step 2. User repeats normal step 3-6 until done.
▨	▨	▨	▨	▨	6a. *Not all Tracings are acceptable: (Not all loops are closed)* .1 System informs user that Tracing(s) on MRIImage slice(s) is (are) not acceptable. .2 User revises the corresponding Tracing(s). .3 Normal Step6.

Table 4.9: The Scenario Table of Task: Define Target.

Task: Set TreatmentOption

Task summary: The user intends to add a light delivery TreatmentDeviceArray into current TreatmentOption for the virtual treatment of the defined target.

Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. There exists an active TreatmentOption of current TreatmentPlan.
3. Target has been defined. Ref. Table 4.9

Success post condition: A delivery TreatmentDeviceArray is added into current TreatmentOption.

Failure post condition: Data of current TreatmentOption remains unchanged.

System constraints: None.

1	2	3	4
User selects to set TreatmentOption.	System responds by presenting the default data items of a TreatmentDeviceArray of current TreatmentOption, including a list of TreatmentDevices within current TreatmentDeviceArray and a default TreatmentDevice(Label, Name, Length, Radius, Power, IlluminationTime, Position).	User selects to Add a TreatmentDevice, or Edit a TreatmentDevice, or Delete a TreatmentDevice. User repeats step 3 until done.	*Current TreatmentOption is blank:* System adds current TreatmentDeviceArray into current TreatmentOp-tion(TreatmentDeviceArray).
1a. User intends to select another TreatmentOption: .1 User selects another TreatmentOption(Number). .2 *Selected TreatmentOption exists:* normal step 1. .2a. *Selected TreatmentOption does not exist:* .1 System creates a blank TreatmentOption with selected TreatmentOption(Number). .2 Normal step1.	▨	▨	▨
▨	▨	▨	*4a Current TreatmentOption is set already:* .1 System notifies user to update the current TreatmentOption or cancel the task. .2a User chooses to update current TreatmentOption: system performs normal step 4 ignoring the normal condition, and also updates all existing dependent data items to their default values. .2b User chooses to cancel the task: task fails.

Table 4.10: The Scenario Table of Task: Set TreatmentOption.

Sub Task: Add a TreatmentDevice

Task summary: The user intends to add a TreatmentDevice into a TreatmentDeviceArray for the virtual treatment.

Precondition: User has selected to set TreatmentOption.

Success post condition: A delivery TreatmentDevice is added into current TreatmentDeviceArray.

Failure post condition: The current delivery TreatmentDeviceArray remains unchanged.

System constraints: None.

1	2	3	4	5	6
User selects to Add a TreatmentDevice.	System presents the linked MRIImageSet.	User sets the Treatment-Device(Position) by selecting the distal tip point on an MRIImage slice.	*Selected TreatmentDe-vice(Position) is acceptable (selected point $\in$ TargetInfo(Tracings)):* System stores the selected point as the TreatmentDe-vice(Position) and ends the presentation of the MRIImageSet.	User completes remaining data items of the TreatmentDevice, see Table 4.10, step2.	System adds the TreatmentDevice to current TreatmentDeviceAr-ray(TreatmentDevice) .
▨	▨	▨	*4a. Selected TreatmentDevice (Position) is not acceptable (selected point $\notin$ TargetInfo (Tracings)):* .1 System notifies user of the mistake. .2 User revises. .3 Normal step 4.	▨	▨

Table 4.11: The Scenario Table of Sub Task: Add a TreatmentDevice.

Task: Do Simulation

Task summary: The user intends to request the system to perform treatment simulation for current patient.

Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. Current active TreatmentOption has been set. Ref. Table 4.10

Success post condition: The LightDose distribution is calculated through out the defined target volume and adjacent tissues.

Failure post condition: The SimulationResult of this patient remains unchanged.

System constraints: None.

1	2	3	4
User selects to do simulation.	System calculates LightDose distribution through out the defined target volume structure (TargetInfo(Tracings)) and surrounding normal tissue structures (AdjacentTissueInfo(Tracings)).	*SimulationResult corresponding to current TreatmentOption does not exist, or exists with the default values:* System stores the SimulationResult with the following data items: Name, LightDose, TreatmentOption(Number), SimulationResultImageSet.	System presents the SimulationResultImageSet containing list of slices in the image set, the fist SimulationResultImage slice which includcs thc first MRIImagc Slicc, ThresholdDoseContours, TreatmentDevices(Position, Label), TargetInfo(Tracing) and AdjacentTissueInfo(Tracing).
		3a. *SimulationResult corresponding to current TreatmentOption exists with none-default values:* .1 System informs user that the SimulationResult of current TreatmentOption already exists, and asks user to overwrite it or cancel the task. .2a User selects to overwrite it: system performs normal step 3 ignoring the normal condition. .2b User selects to cancel: task fails.	

Table 4.12: The Scenario Table of Task: Do Simulation.

Task: Generate TreatmentPlanReport

Task summary: The user intends to select a TreatmentOption and generates the TreatmentPlanReport for current patient.

Precondition: 1. User has selected to make TreatmentPlan for this patient.

2. At least one SimulationResult has been calculated and stored in current TreatmentPlan. Ref. Table 4.12

Success post condition: A selected TreatmentPlanReport is generated.

Failure post condition: Current TreatmentPlan data of the patient remains unchanged.

System constraints: None.

1	2	3	4	5	6
User selects one TreatmentOption(Number) of current TreatmentPlan.	*Selection of current TreatmentOption is successful:* System retrieves corresponding TreatmentOption (TreatmentDeviceArray) and SimulationResult.	User selects to present SimulationResult.	System presents the SimulationResultImageSet containing list of slices of the image set, the first SimulationResultImage slice which includes the first MRIImage Slice, ThresholdDoseContours, TreatmentDevices(Position, Label), TargetInfo(Tracing) and AdjacentTissueInfo(Tracing). User repeats step 1-4 until the selection is done.	User selects to generate TreatmentPlanReport.	System produces the TreatmentPlanReport which includes: PatientInfo (Name, PhysicianName), TargetInfo (Name, ThresholdDose), selected TreatmentOption (TreatmentDeviceArray), and snapshots of the presentation of selected SimulationResultImageSet.
	2a. Selection of current TreatmentOption fails: .1 System notifies user, and asks user to try another one, or cancel the task. .2a User selects to try another one: task returns to normal step 1. .2b User selects to cancel the task: task fails.				

Table 4.13: The Scenario Table of Task: Generate TreatmentPlanReport.

4.1.7 Use Cases

Based on our rules for capturing use cases, corresponding use cases can be derived from scenario tables of user tasks, see use cases in Tables 4.14, 4.15, 4.16, 4.17, 4.18, 4.19, 4.20. Also, a use case diagram is constructed to give an overview of the main functionality of the system, see Figure 4.5.

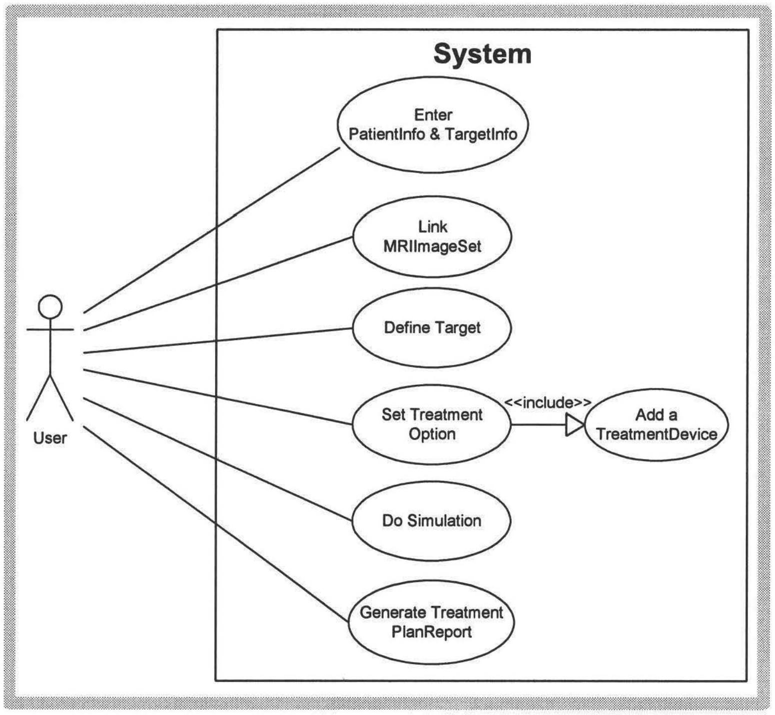


Figure 4.5: Use Case Diagram of Treatment Planning Software: Case Study

4.1.8 User Requirements Document

The above content may be specified in the User Requirements Documents according to the template introduced in 3.2.5.6 of precious chapter, we omit it here.

Use Case ID: UC001
Use Case Name: Enter PatientInfo & TargetInfo
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user intends to enter the information of a patient and his/her being-treated target in order to make a new TreatmentPlan for that patient.
Level: User Task
Primary Actor: User(Radiologist, Planner)
Precondition: User is logged in.
Success Postcondition:
1. Entered PatientInfo and TargetInfo are added into the system.
2. A new TreatmentPlan is created and stored with some initial data values.
3. A blank TreatmentOption is created and stored with entered TreatmentOption(Number), which is the active TreatmentOption of current TreatmentPlan.

Failure Postcondition: No stored data will change within the system.
Normal Scenario:
1. User selects to make TreatmentPlan.
2. System responds by presenting required data items with default values where necessary, including PatientInfo(Name, PhysicianName), TargetInfo(Name, AbsorptionCoeff, ScatteringCoeff, ThresholdDose) and TreatmentOption(Number).
3. User completes all data items.
4. System creates and stores a new TreatmentPlan for this patient with some initial data values: Name, entered PatientInfo(Name), entered TargetInfo(Name), a blank TreatmentOption with entered TreatmentOption(Number). System also stores entered data of PatientInfo, TargetInfo and TreatmentOption.

Alternative Scenarios:
- 4a. TreatmentPlan already exists:
  - .1 System informs user that a TreatmentPlan already exists for this patient, and asks user to use the entered data for a new TreatmentPlan, or revise the current input, or cancel the task.
  - .2a User selects to use the entered data for a new TreatmentPlan: system performs normal step 4 neglecting the normal condition.
  - .2b User selects to revise the current input and revises the PatientInfo(Name): task returns to normal step 3.
  - .2c User selects to cancel the task: task fails.

Capacity: 1
Associations:
System Constraints: None.

Table 4.14: Use Case UC001: Enter PatientInfo & TargetInfo.

Use Case ID: UC002
Use Case Name: Link MRIImageSet
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user chooses to link an MRIImageSet to the current patient's current treatment plan.
Level: User Task
Primary Actor: User(Radiologist, Planner)
Precondition:
1. User has selected to make TreatmentPlan for this patient.
2. A new TreatmentPlan for current patient with initial data values has been created.
3. Required PatientInfo and TargetInfo have been stored. Ref. Table 4.14.

Success Postcondition: An MRIImageSet is linked to current TreatmentPlan and presented.
Failure Postcondition: None.
Normal Scenario:
1. User selects to link an MRIImageSet.
2. System presents a list of names of available MRIImageSets.
3. User selects an MRIImageSet of current patient.
4. System links the selected MRIImageSet to current TreatmentPlan and presents the selected MRIImageSet, including presentation of the first MRIImage slice of the image set and list of MRIImage slices in the set.

Alternative Scenarios:
- 2a. No ImageSets available:
  - .1 System informs user.
  - .2 Task fails.
- 4a. Selection of MRIImageSet is not successful & Number of retried times < 3:
  - .1 System informs user that the selection of current MRIImageSet fails, and asks user to retry the current selection, or try another selection, or cancel the task.
  - .2a User selects to retry current selection: system performs normal step 4.
  - .2b User selects to retry another selection: system returns to normal step 2.
  - .2c User selects to cancel: task fails.
- 4b. Selection of MRIImageSet is not successful & Number of Retried times ≥ 3:
  - .1 System notifies user of failure.
  - .2 Task fails.

Capacity: 1
Associations: None.
System Constraints: Successive retries of the same MRIImageSet selection can be executed at most three times.

Table 4.15: Use Case UC002: Link MRIImageSet.

Use Case ID: UC003
Use Case Name: Define Target
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user intends to define the target on a linked MRIImageSet of current patient for the treatment simulation, which includes the definition of target volume structure (TargetInfo(Tracings)) and adjacent normal tissue structures (AdjacentTissueInfo(Tracings)).
Level: User Task
Primary Actor: User(Radiologist, Planner)
Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. A MRIImageSet has been linked to current patient. Ref. Table 4.15
Success Postcondition: The target volume structure and adjacent normal tissue structures are defined.
Failure Postcondition: No stored data will change in the system.
Normal Scenario:
1. User selects to define target.
2. System responds by presenting a list of names of structures that need to be defined, including being-treated target volume and adjacent normal tissues.
3. User selects the target volume or an adjacent normal tissue to define.
4. System presents the linked MRIImageSet of current patient, including presentation of the first MRIImage slice and list of MRIImage slices in the set; and other required data items with default values where necessary, including: TargetInfo(Name, VolumeValue) or AdjacentTissueInfo(Name, VolumeValue), and Tracing(PointsNumber, AreaValue) on current MRIImage slice.
5. User traces the Tracing(Loops) of selected structure by selecting points to form loops on each MRIImage slice where it is perceivable(visible).
6. System stores Tracings to a TargetInfo(Tracings) or AdjacentTissueInfo(Tracings). System returns to normal step 2.

User repeats normal step 3-6 until done.

Alternative Scenarios:
6a. Not all Tracings are acceptable (Not all loops are closed):
.1 System informs user that Tracing(s) on MRIImage slice(s) is (are) not acceptable.
.2 User revises the corresponding Tracing(s).
.3 Normal Step 6.

Capacity: 1
Associations: None.
System Constraints: None.

Table 4.16: Use Case UC003: Define Target.

Use Case ID: UC004
Use Case Name: Set TreatmentOption
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user intends to add a light delivery TreatmentDeviceArray into current TreatmentOption for the virtual treatment of the defined target.
Level: User Task
Primary Actor: Planner
Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. Target has been defined. Ref. Table 4.16
3. There exists an active TreatmentOption of current TreatmentPlan.
Success Postcondition: A delivery TreatmentDeviceArray is added into current TreatmentOption.
Failure Postcondition: Data of current TreatmentOption remains unchanged.
Normal Scenario:

1. User selects to set TreatmentOption.
2. System responds by presenting the default data items of a TreatmentDeviceArray of current TreatmentOption, including a list of TreatmentDevices within current TreatmentDeviceArray and a default TreatmentDevice(Label, Name, Length, Radius, Power, IlluminationTime, Position).
3. User selects to Add a TreatmentDevice, or Edit a TreatmentDevice, or Delete a TreatmentDevice.
   User repeats step 3 until done.
4. System adds current TreatmentDeviceArray into current TreatmentOption(TreatmentDeviceArray).

Alternative Scenarios:

1a. User intends to select another TreatmentOption:
  .1 User selects another TreatmentOption(Number).
  .2 Normal step1.
  .2a. Selected TreatmentOption does not exist:
    .1 System creates a blank TreatmentOption with selected TreatmentOption(Number).
    .2 Normal step1.

4a. Current TreatmentOption is set already:
  .1 System notifies user to update the current TreatmentOption or cancel the task.
  .2a User chooses to update current TreatmentOption: system performs normal step 4 ignoring the normal condition, and also updates all existing dependent data items to their default values.
  .2b User chooses to cancel the task: task fails.

Capacity: 1
Associations: UC005 (Table 4.18).
System Constraints: None.

Table 4.17: Use Case UC004: Set TreatmentOption.

Use Case ID: UC005
Use Case Name: Add a TreatmentDevice
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user intends to add a TreatmentDevice into a TreatmentDeviceArray for the virtual treatment.
Level: Sub User Task
Primary Actor: Planner
Precondition: User has activated the set TreatmentOption.
Success Postcondition: A delivery TreatmentDevice is added into current TreatmentDeviceArray.
Failure Postcondition: The current delivery TreatmentDeviceArray remains unchanged.
Normal Scenario:

1. User selects to Add a TreatmentDevice.
2. System presents the linked MRIImageSet.
3. User sets the TreatmentDevice(Position) by selecting the distal tip point on an MRIImage slice.
4. System stores the selected point as the TreatmentDevice(Position) and ends the presentation of the MRIImageSet.
5. User completes remaining data items of the TreatmentDevice, see Table 4.17, step2.
6. System adds the TreatmentDevice to current TreatmentDeviceArray(TreatmentDevice).

Alternative Scenarios:

4a. Selected TreatmentDevice (Position) is not acceptable (selected point $\notin$TargetInfo(Tracings)):
- .1 System notifies user of the mistake.
- .2 User revises.
- .3 Normal step4.

Capacity: 1
Associations: None.
System Constraints: Only straight delivery fiber is allowed.

Table 4.18: Use Case UC005: Add a TreatmentDevice

Use Case ID: UC006
Use Case Name: Do Simulation
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user intends to request the system to perform treatment simulation for current patient.
Level: User Task
Primary Actor: Planner
Precondition: 1. User has selected to make TreatmentPlan for this patient.
2. Current active TreatmentOption has been set. Ref. Table 4.17
Success Postcondition: The LightDose distribution is calculated through out the defined target volume and adjacent tissues.
Failure Postcondition: The SimulationResult of this patient remains unchanged.
Normal Scenario:

1. User selects to do simulation.
2. System calculates LightDose distribution through out the defined target volume structure (TargetInfo(Tracings)) and surrounding normal tissue structures (AdjacentTissueInfo(Tracings)).
3. System stores the SimulationResult with the following data items: Name, LightDose, TreatmentOption(Number), SimulationResultImageSet.
4. System presents the SimulationResultImageSet containing list of slices in the image set, the fist SimulationResultImage slice which includes the first MRIImage Slice, ThresholdDoseContours, TreatmentDevices(Position, Label), TargetInfo(Tracing) and AdjacentTissueInfo(Tracing).

Alternative Scenarios:

3a. SimulationResult corresponding to current TreatmentOption exists with none-default values:
 .1 System informs user that the SimulationResult of current TreatmentOption already exists, and asks user to overwrite it or cancel the task.
 .2a User selects to overwrite it: system performs normal step 3 ignoring the normal condition.
 .2b User selects to cancel: task fails.

Capacity: 1
Associations: None.
System Constraints: None.

Table 4.19: Use Case UC006: Do Simulation

Use Case ID: UC007
Use Case Name: Generate TreatmentPlanReport
Created by: **Date Created:**
Last Updated by: **Date Last Updated:**
Summary: The user intends to select a TreatmentOption and generates the TreatmentPlan-Report for current patient.
Level: User Task
Primary Actor: Planner
Precondition:
1. User has selected to make TreatmentPlan for this patient.
2. At least one SimulationResult has been calculated and stored in current TreatmentPlan.Ref. Table 4.19.

Success Postcondition: A selected TreatmentPlanReport is generated.
Failure Postcondition: Current TreatmentPlan data of the patient remains unchanged.
Normal Scenario:
1. User selects one TreatmentOption(Number) of current TreatmentPlan.
2. System retrieves corresponding TreatmentOption(TreatmentDeviceArray) and SimulationResult.
3. User selects to present SimulationResult.
4. System presents the SimulationResultImageSet containing list of slices of the image set, the first SimulationResultImage slice which includes the first MRIImage Slice, ThresholdDoseContours, TreatmentDevices(Position, Label), TargetInfo(Tracing) and AdjacentTissueInfo(Tracing).
   User repeats step 1-4 until the selection is done.
5. User selects to generate TreatmentPlanReport.
6. System produces the TreatmentPlanReport which includes: PatientInfo(Name, PhysicianName), TargetInfo(Name, ThresholdDose), selected TreatmentOption(TreatmentDeviceArray), and snapshots of the presentation of selected SimulationResultImageSet.

Alternative Scenarios:
- 2a. Selection of current TreatmentOption fails:
  - .1 System notifies user, and asks user to try another one, or cancel the task.
  - .2a User selects to try another one: task returns to normal step 1.
  - .2b User selects to cancel the task: task fails.

Capacity: 1
Associations: None.
System Constraints: None.

Table 4.20: Use Case UC007: Generate TreatmentPlanReport

4.2 Apply Analysis Process

4.2.1 Activity Diagrams and Data Hierarchies

According to our approach, activity diagrams are firstly constructed for each use case, and data used in each action step of an activity diagram are captured in the data hierarchies. Eventually, all data hierarchies are combined into the final data hierarchies.

Based on the use cases, activity diagrams and corresponding data hierarchies are made in Figures 4.6, 4.7, 4.8, 4.9, 4.10, 4.11, and 4.12. We eventually get the final data hierarchy model in Figure 4.13.

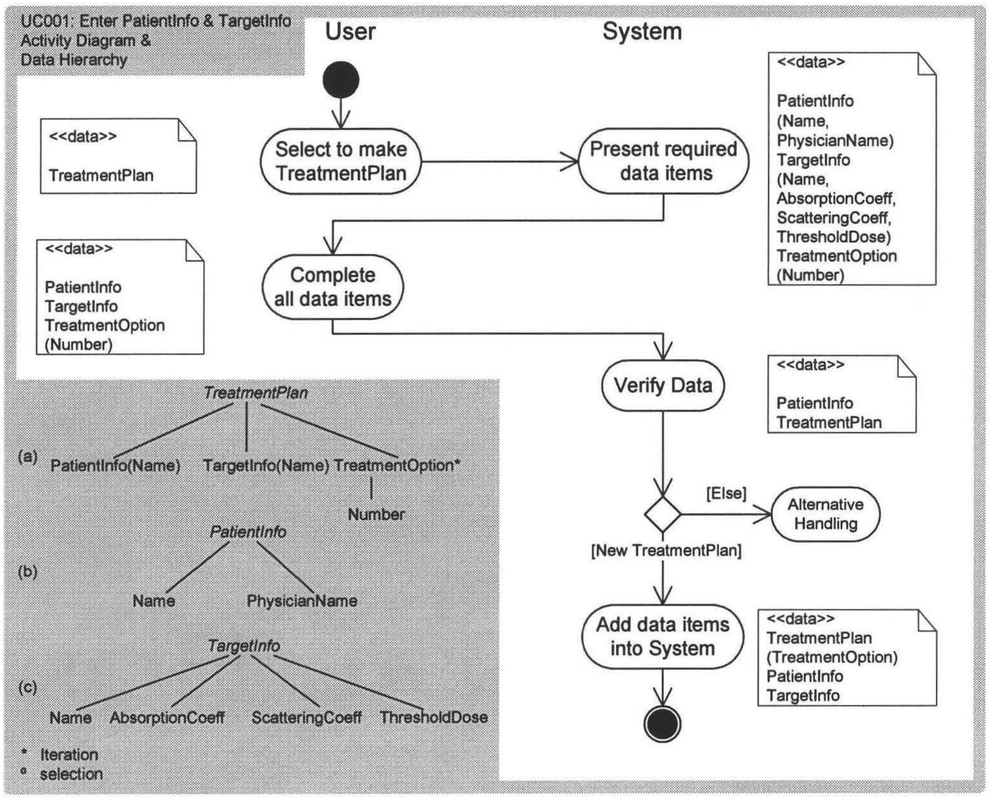

UC001: Enter PatientInfo & TargetInfo Activity Diagram & Data Hierarchy
User
System
<<data>> TreatmentPlan
<<data>> PatientInfo TargetInfo TreatmentOption (Number)
Select to make TreatmentPlan
Present required data items
<<data>> PatientInfo (Name, PhysicianName) TargetInfo (Name, AbsorptionCoeff, ScatteringCoeff, ThresholdDose) TreatmentOption (Number)
Complete all data items
Verify Data
<<data>> PatientInfo TreatmentPlan
[Else]
Alternative Handling
[New TreatmentPlan]
Add data items into System
<<data>> TreatmentPlan (TreatmentOption) PatientInfo TargetInfo
TreatmentPlan
(a) PatientInfo(Name) TargetInfo(Name) TreatmentOption*
Number
PatientInfo
(b) Name PhysicianName
TargetInfo
(c) Name AbsorptionCoeff ScatteringCoeff ThresholdDose
* Iteration
° selection

Figure 4.6: Activity Diagram and Data Hierarchy: Enter PatientInfo & TargetInfo Task.

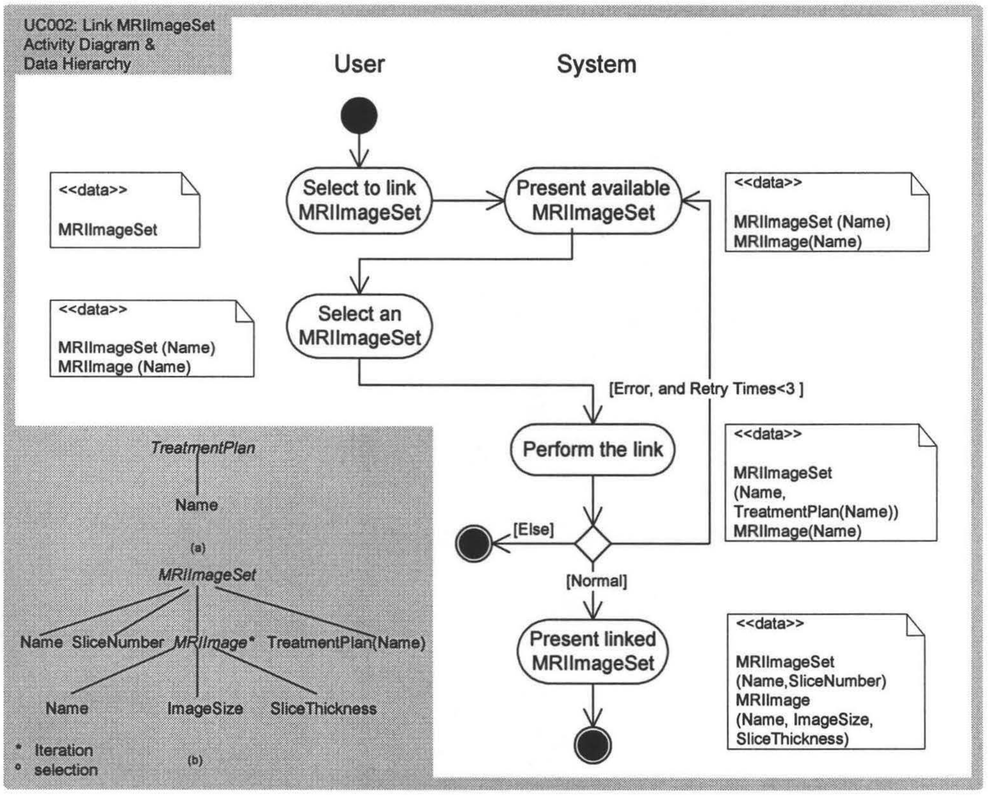


Figure 4.7: Activity Diagram and Data Hierarchy: Link MRIImageSet Task

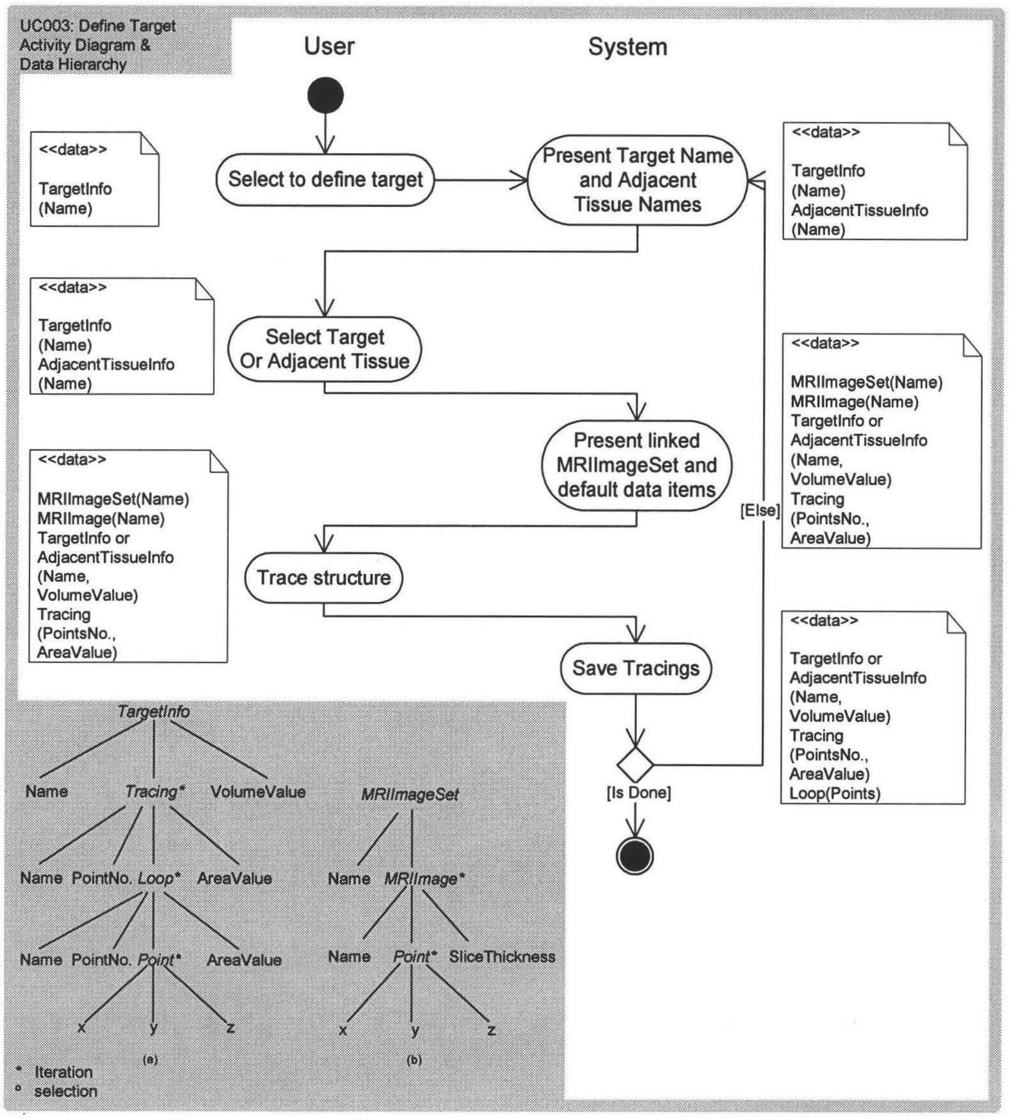

UC003: Define Target Activity Diagram & Data Hierarchy
User
System
<<data>> TargetInfo (Name)
Select to define target
Present Target Name and Adjacent Tissue Names
<<data>> TargetInfo (Name) AdjacentTissueInfo (Name)
<<data>> TargetInfo (Name) AdjacentTissueInfo (Name)
Select Target Or Adjacent Tissue
Present linked MRIImageSet and default data items
<<data>> MRIImageSet(Name) MRIImage(Name) TargetInfo or AdjacentTissueInfo (Name, VolumeValue) Tracing (PointsNo., AreaValue)
<<data>> MRIImageSet(Name) MRIImage(Name) TargetInfo or AdjacentTissueInfo (Name, VolumeValue) Tracing (PointsNo., AreaValue)
[Else]
Trace structure
Save Tracings
<<data>> TargetInfo or AdjacentTissueInfo (Name, VolumeValue) Tracing (PointsNo., AreaValue) Loop(Points)
[Is Done]
TargetInfo
Name Tracing* VolumeValue
Name PointNo. Loop* AreaValue
Name PointNo. Point* AreaValue
x y z
(a)
MRIImageSet
Name MRIImage*
Name Point* SliceThickness
x y z
(b)
* Iteration
° selection

Figure 4.8: Activity Diagram and Data Hierarchy: Define Target Task

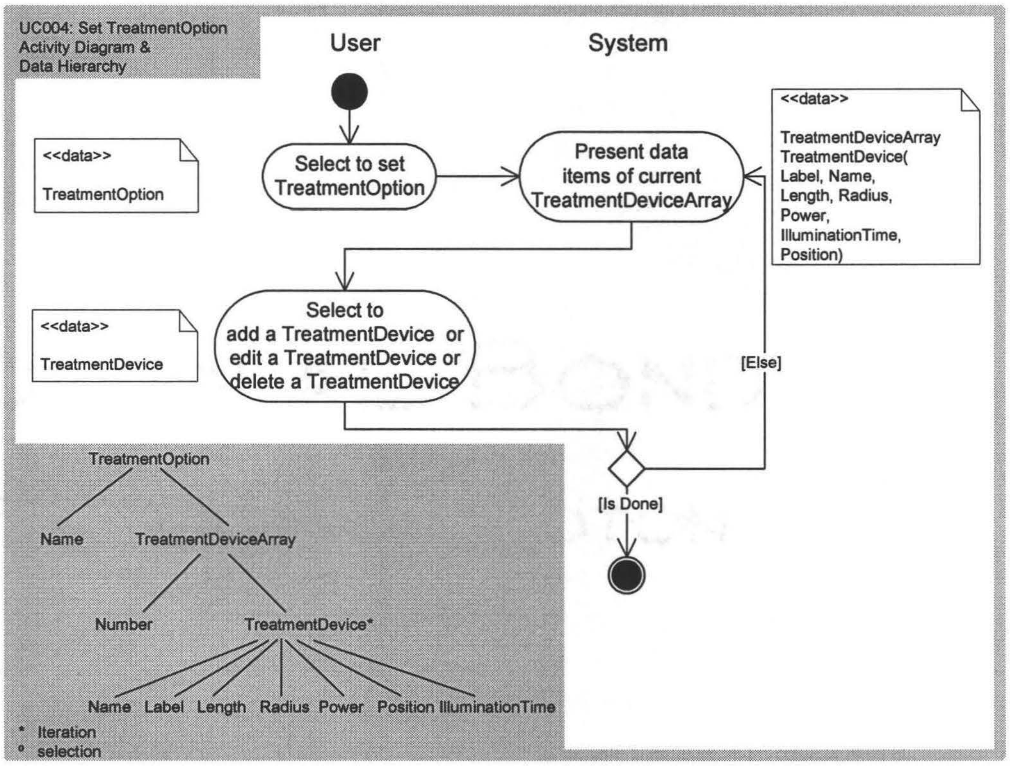

UC004: Set TreatmentOption Activity Diagram & Data Hierarchy
User
System
<<data>> TreatmentOption
Select to set TreatmentOption
Present data items of current TreatmentDeviceArray
<<data>> TreatmentDeviceArray TreatmentDevice(Label, Name, Length, Radius, Power, IlluminationTime, Position)
<<data>> TreatmentDevice
Select to add a TreatmentDevice or edit a TreatmentDevice or delete a TreatmentDevice
[Else]
[Is Done]
TreatmentOption
Name
TreatmentDeviceArray
Number
TreatmentDevice*
Name Label Length Radius Power Position IlluminationTime
* Iteration
° selection

Figure 4.9: Activity Diagram and Data Hierarchy: Set TreatmentOption Task

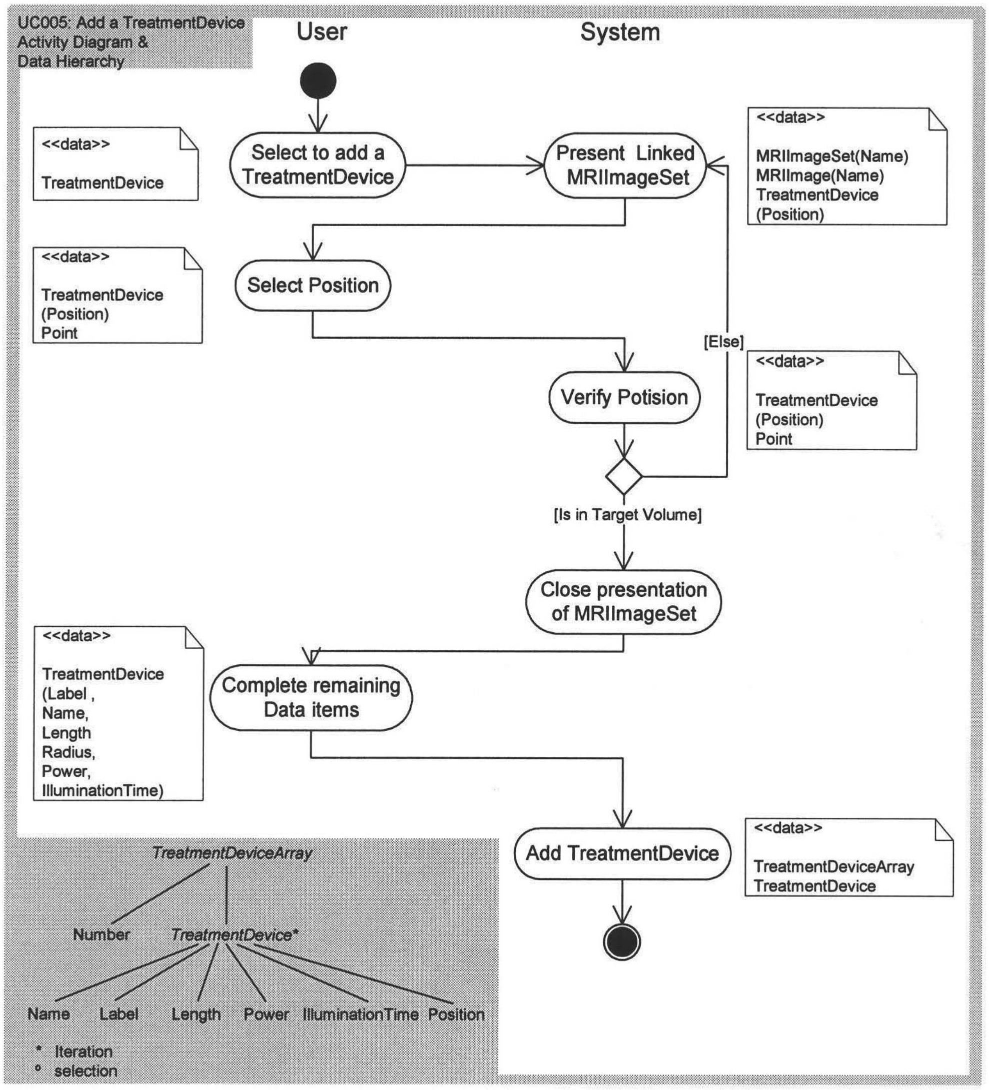

UC005: Add a TreatmentDevice Activity Diagram & Data Hierarchy
User
System
<<data>> TreatmentDevice
Select to add a TreatmentDevice
Present Linked MRIImageSet
<<data>> MRIImageSet(Name) MRIImage(Name) TreatmentDevice (Position)
<<data>> TreatmentDevice (Position) Point
Select Position
[Else]
Verify Potision
<<data>> TreatmentDevice (Position) Point
[Is in Target Volume]
Close presentation of MRIImageSet
<<data>> TreatmentDevice (Label , Name, Length Radius, Power, IlluminationTime)
Complete remaining Data items
Add TreatmentDevice
<<data>> TreatmentDeviceArray TreatmentDevice
TreatmentDeviceArray
Number
TreatmentDevice*
Name
Label
Length
Power
IlluminationTime
Position
* Iteration
° selection

Figure 4.10: Activity Diagram and Data Hierarchy: Add a TreatmentDevice Task

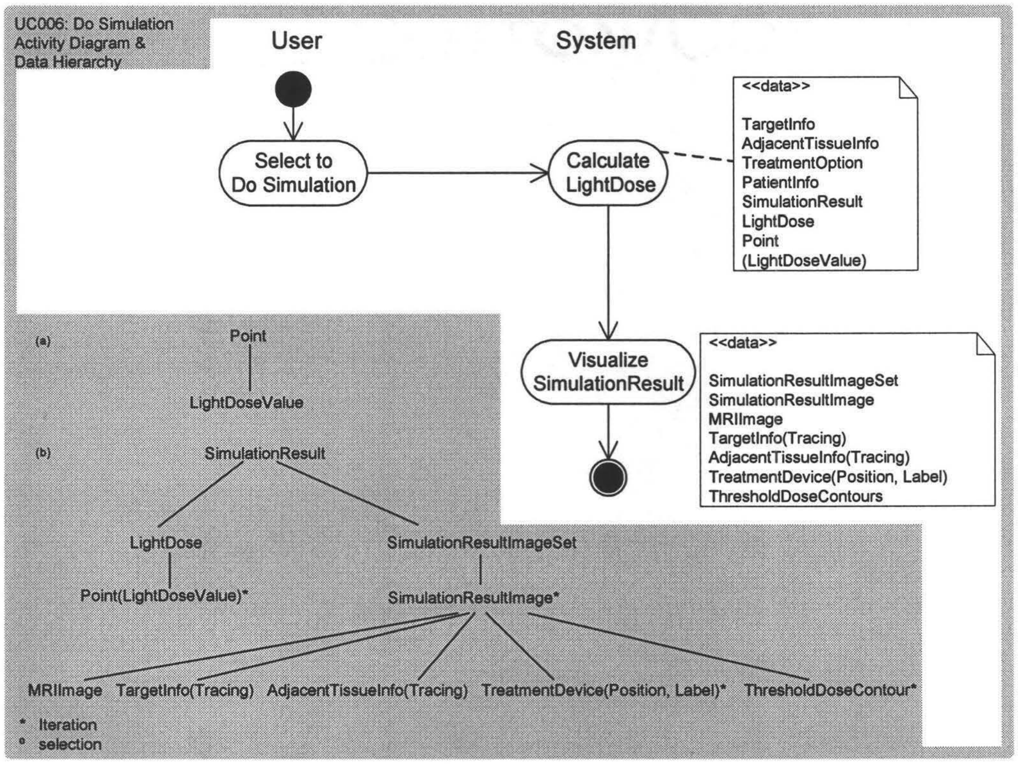


Figure 4.11: Activity Diagram and Data Hierarchy: Do Simulation Task

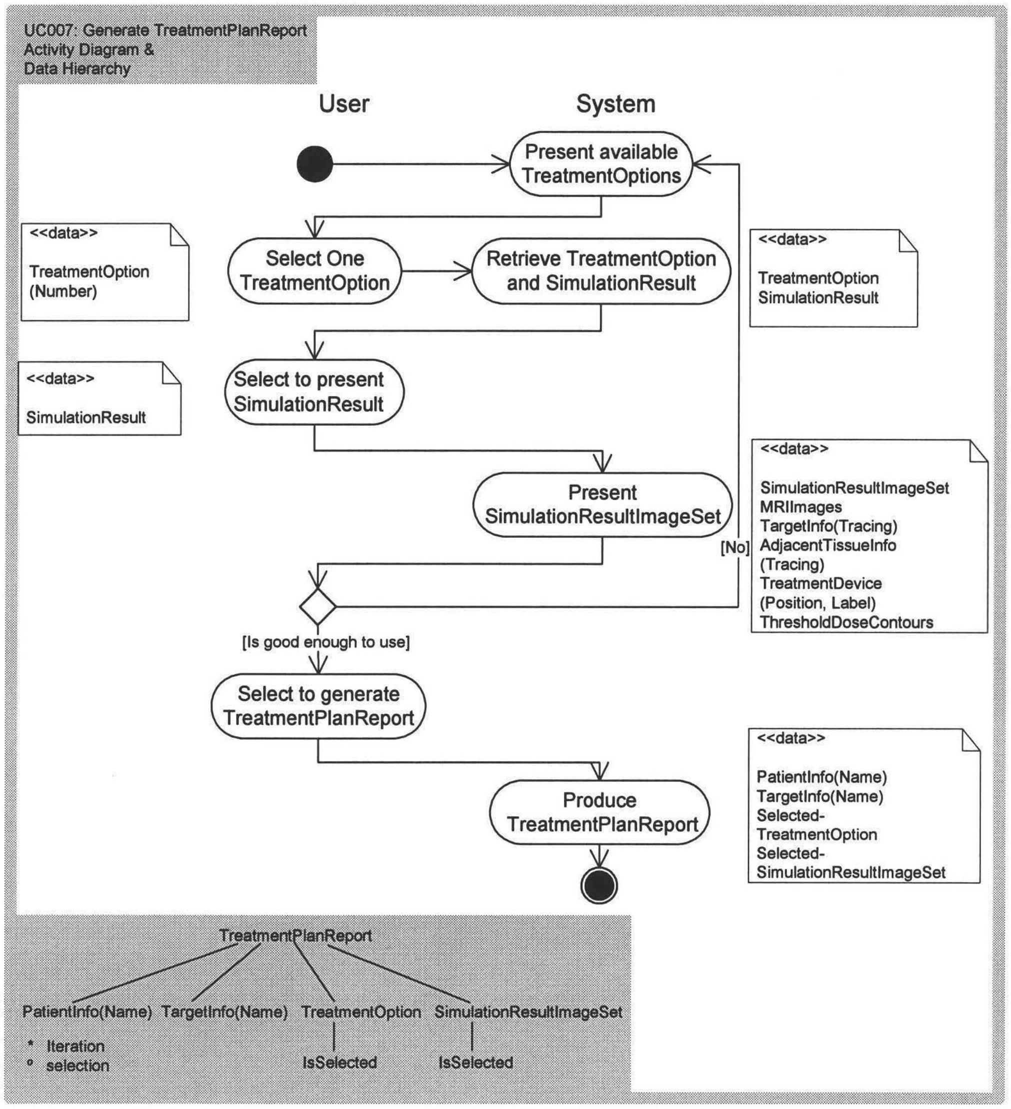


Figure 4.12: Activity Diagram and Data Hierarchy: Generate TreatmentPlanReport Task

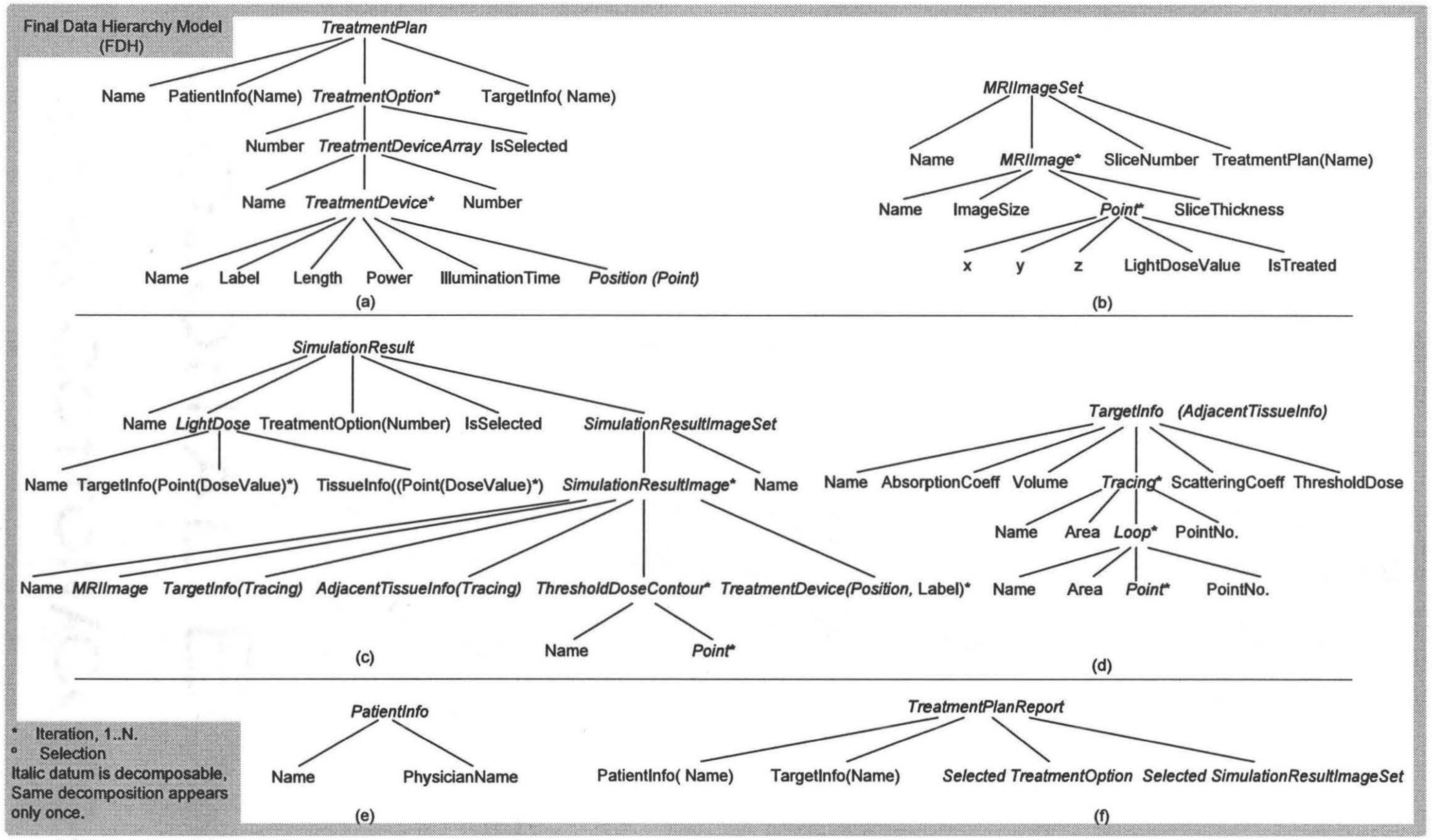


Figure 4.13: Final Data Hierarchy Model: Case Study

4.2.2 Initial Domain Class Diagram

From Final Data Hierarchy 4.13, we can find all the data used in use cases, which constitute the initial domain class model including domain classes, their attributes and relationships. According to the identifying rules, the initial domain class diagram is illustrated in Figure 4.14.

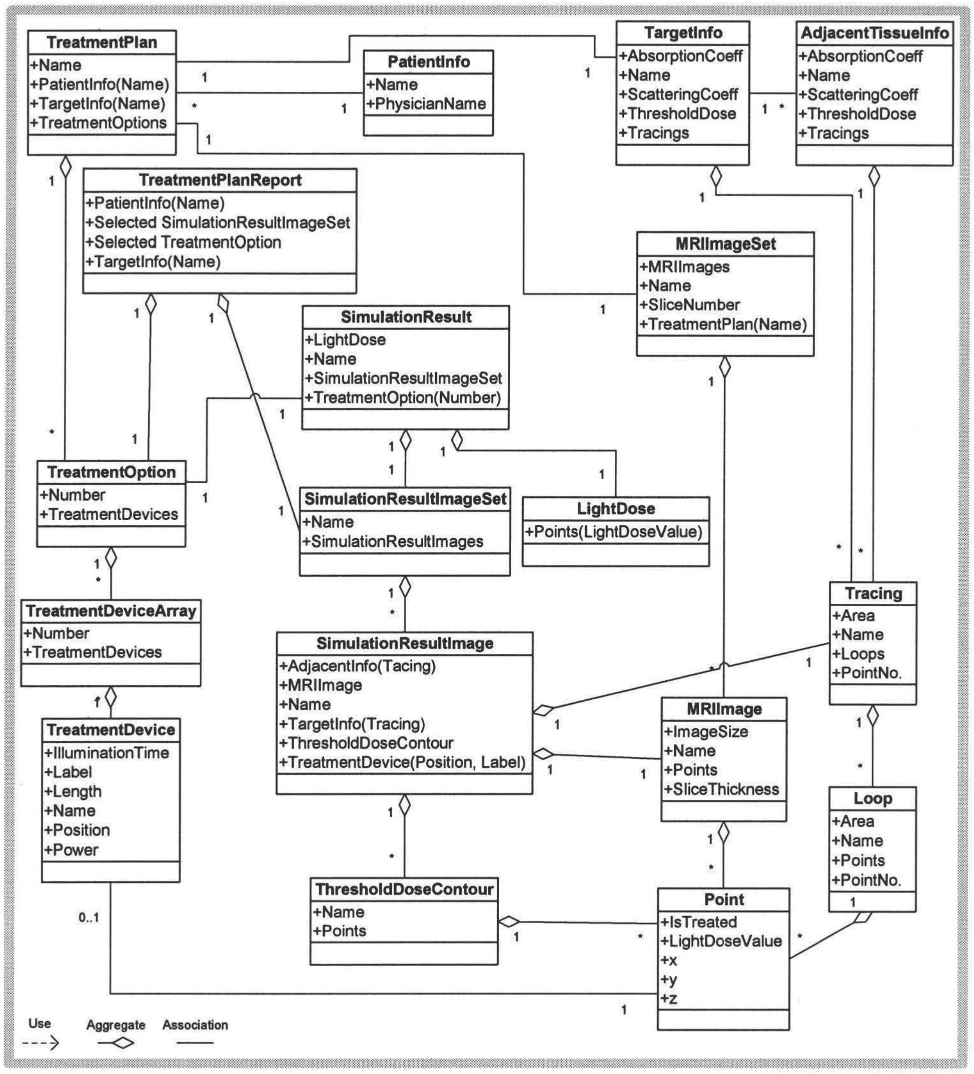


Figure 4.14: Initial Domain Class Diagram: Case Study

4.2.3 Sequence Diagrams, Boundary Classes and Class Functions

To find boundary classes and allocate the functionality to different analysis classes, sequence diagrams are a wonderful mechanism. Realizing use cases by sequence diagrams can help ensure that we have an accurate and complete analysis class model. While the behaviors of use cases are assigned to different classes, we can also check if any domain classes are missing. For each use case, we draw at least one sequence diagram. In Figures 4.15, 4.16, 4.17, 4.18, 4.19, and 4.20, the normal scenario of each use case is realized respectively.

4.2.4 Analysis Class Diagrams

The class diagram is an effective mechanism to show a static view of the system. From sequence diagrams, we identify the boundary classes, newly found domain classes and all the class's functions. They are specified in the boundary class diagram and final domain class diagram respectively. Figures 4.21 and 4.22 are what we captured from the sequence diagrams, where the final domain class diagram also combines the initial domain class diagram Figure 4.14. We separate the domain class diagram and boundary class diagram in order to show the different parts of the system and also to enhance the readability of the diagram.

As discussed in the previous chapter, the boundary objects are not only the user interfaces in our approach, they are also the holders of the system functions. Each boundary class is assigned some functions of the system, and all the system functions are navigated starting from a certain boundary object. At the requirements stage, we just hope to explore externally visible classes, the internal functional classes are the tasks of the design phase.

4.3 Apply Specification Process

The specification process is the activity to specify the results of the elicitation and analysis processes. In this case study, we focus on the class specifications - the most important part of an SRS. In the PDT case study, we begin with the boundary class specifications which we regard as part of the functional requirements. We specify the classes according to their appearance order in the class diagram, from left to right, top to bottom. For each function, we should also refer to the sequence diagrams, and activity diagrams if necessary. We list the partial class specifications in appendix B.

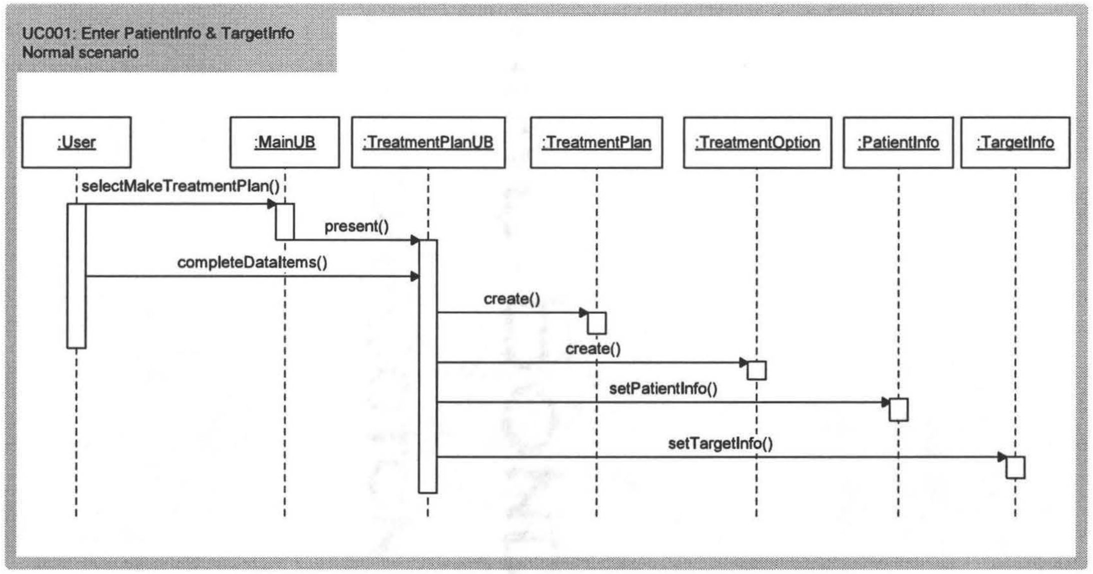


Figure 4.15: Sequence Diagram UC001: Enter PatientInfo & TargetInfo

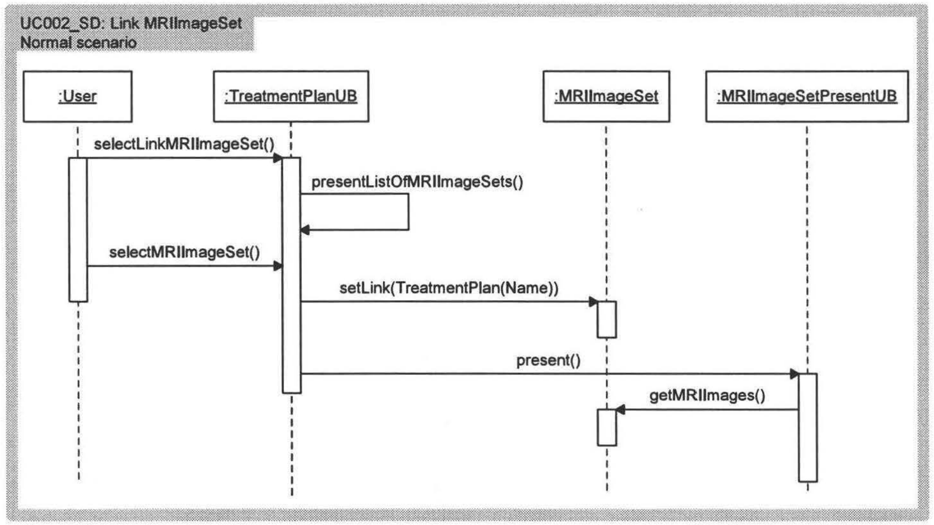


Figure 4.16: Sequence Diagram UC002: Link MRIImageSet

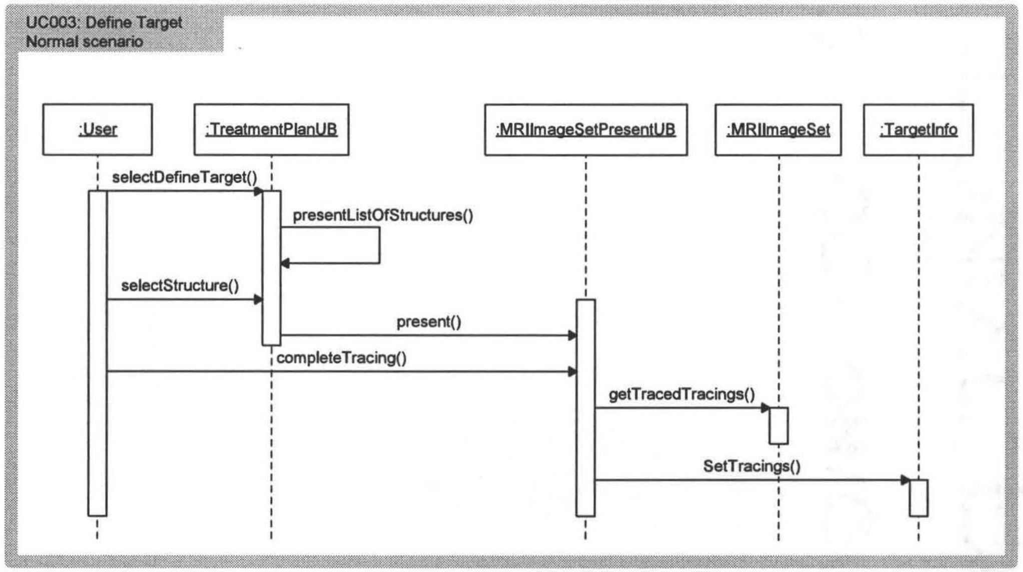


Figure 4.17: Sequence Diagram UC003: Define Target

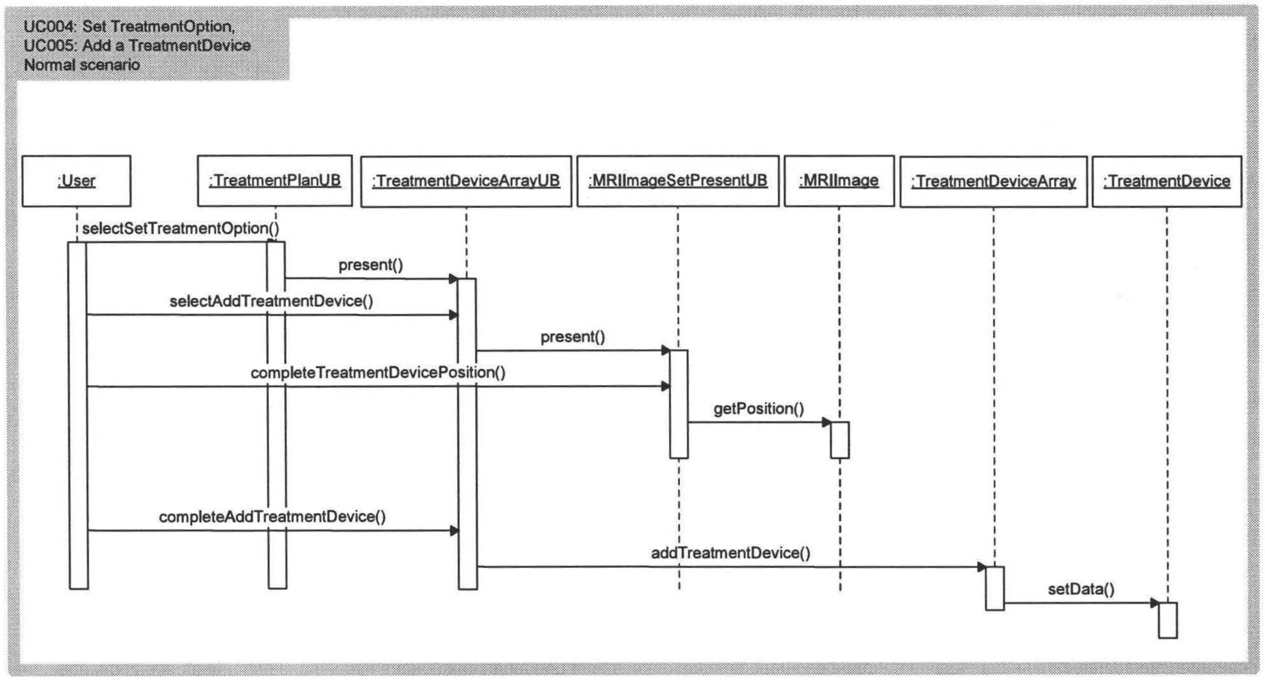


Figure 4.18: Sequence Diagram UC004: Set TreatmentOption

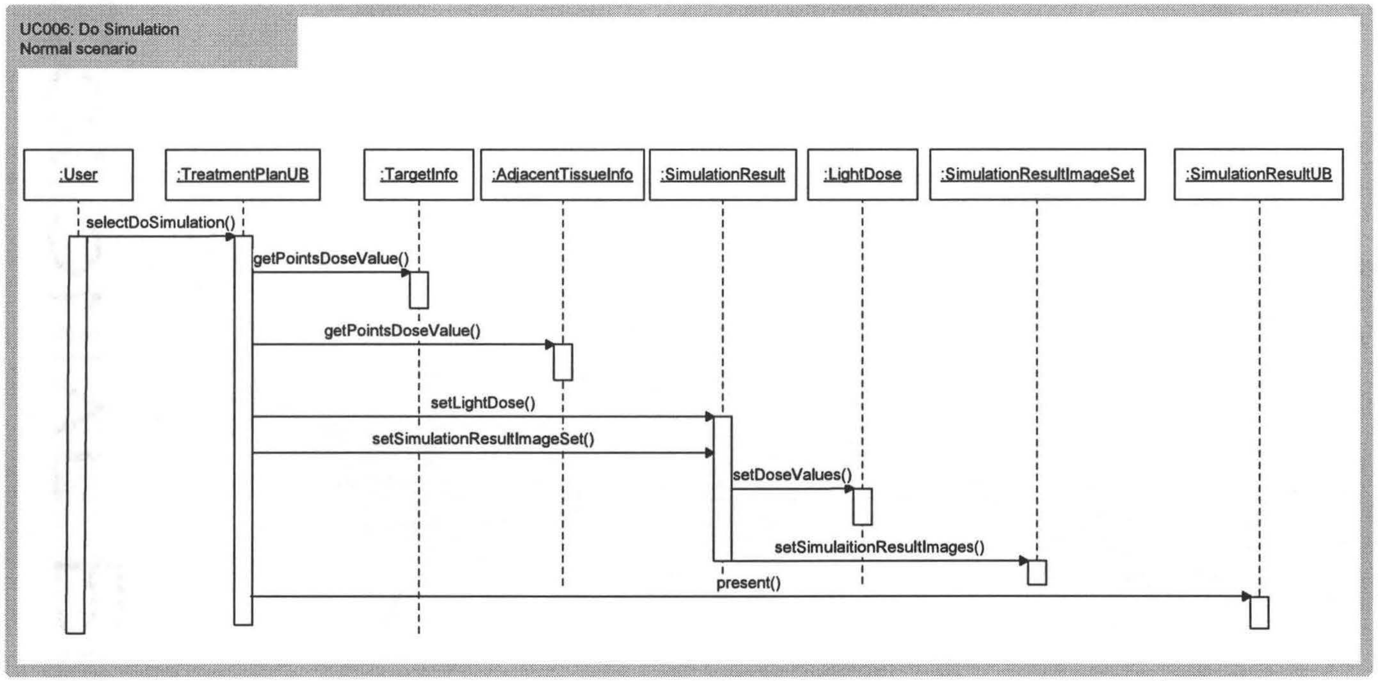


Figure 4.19: Sequence Diagram UC006: Do Simulation

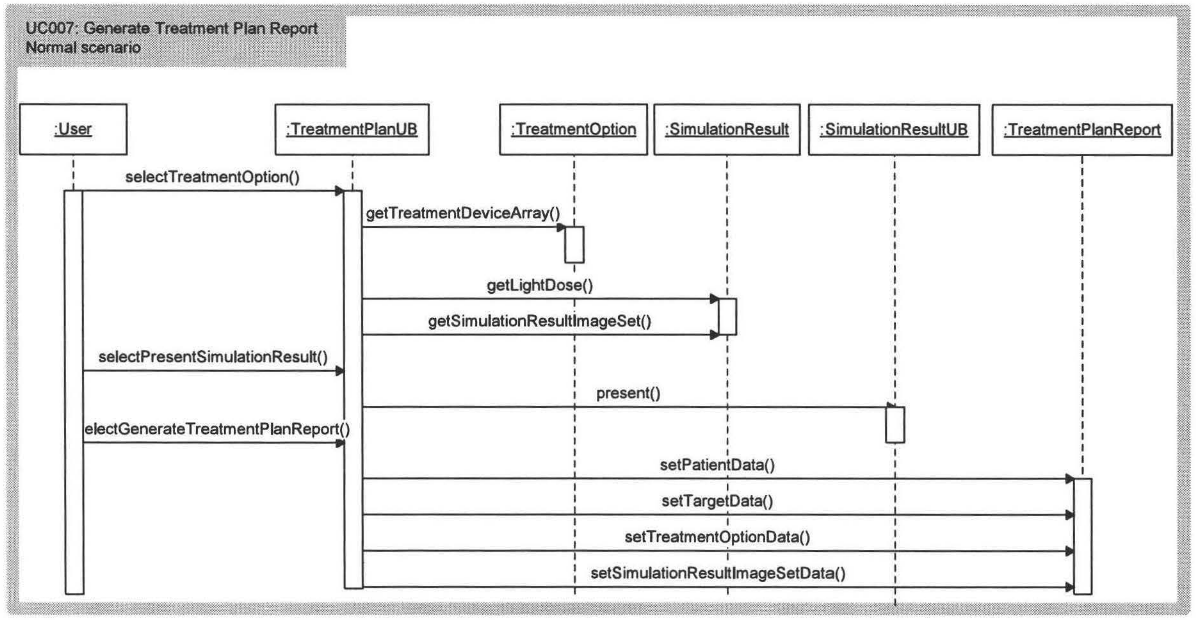


Figure 4.20: Sequence Diagram UC007: Generate TreatmentPlanReport

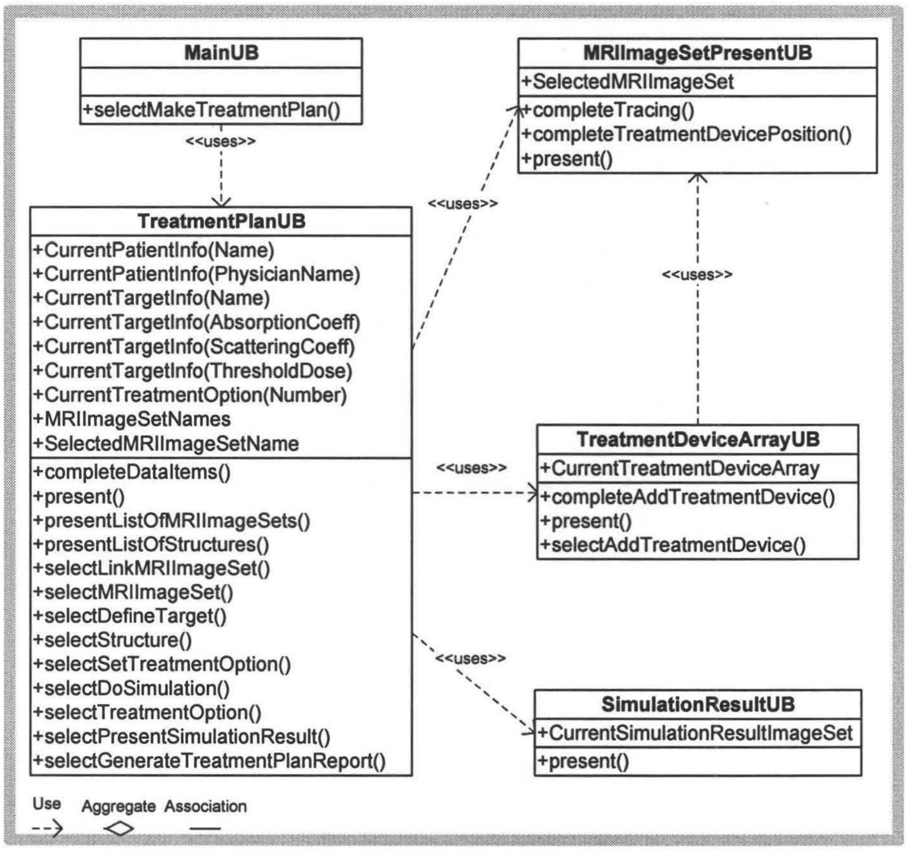

MainUB
+selectMakeTreatmentPlan()
<<uses>>
TreatmentPlanUB
+CurrentPatientInfo(Name)
+CurrentPatientInfo(PhysicianName)
+CurrentTargetInfo(Name)
+CurrentTargetInfo(AbsorptionCoeff)
+CurrentTargetInfo(ScatteringCoeff)
+CurrentTargetInfo(ThresholdDose)
+CurrentTreatmentOption(Number)
+MRIImageSetNames
+SelectedMRIImageSetName
+completeDataItems()
+present()
+presentListOfMRIImageSets()
+presentListOfStructures()
+selectLinkMRIImageSet()
+selectMRIImageSet()
+selectDefineTarget()
+selectStructure()
+selectSetTreatmentOption()
+selectDoSimulation()
+selectTreatmentOption()
+selectPresentSimulationResult()
+selectGenerateTreatmentPlanReport()
MRIImageSetPresentUB
+SelectedMRIImageSet
+completeTracing()
+completeTreatmentDevicePosition()
+present()
<<uses>>
<<uses>>
TreatmentDeviceArrayUB
+CurrentTreatmentDeviceArray
+completeAddTreatmentDevice()
+present()
+selectAddTreatmentDevice()
<<uses>>
<<uses>>
SimulationResultUB
+CurrentSimulationResultImageSet
+present()
Use
Aggregate
Association

Figure 4.21: Boundary Class Diagram: Case Study

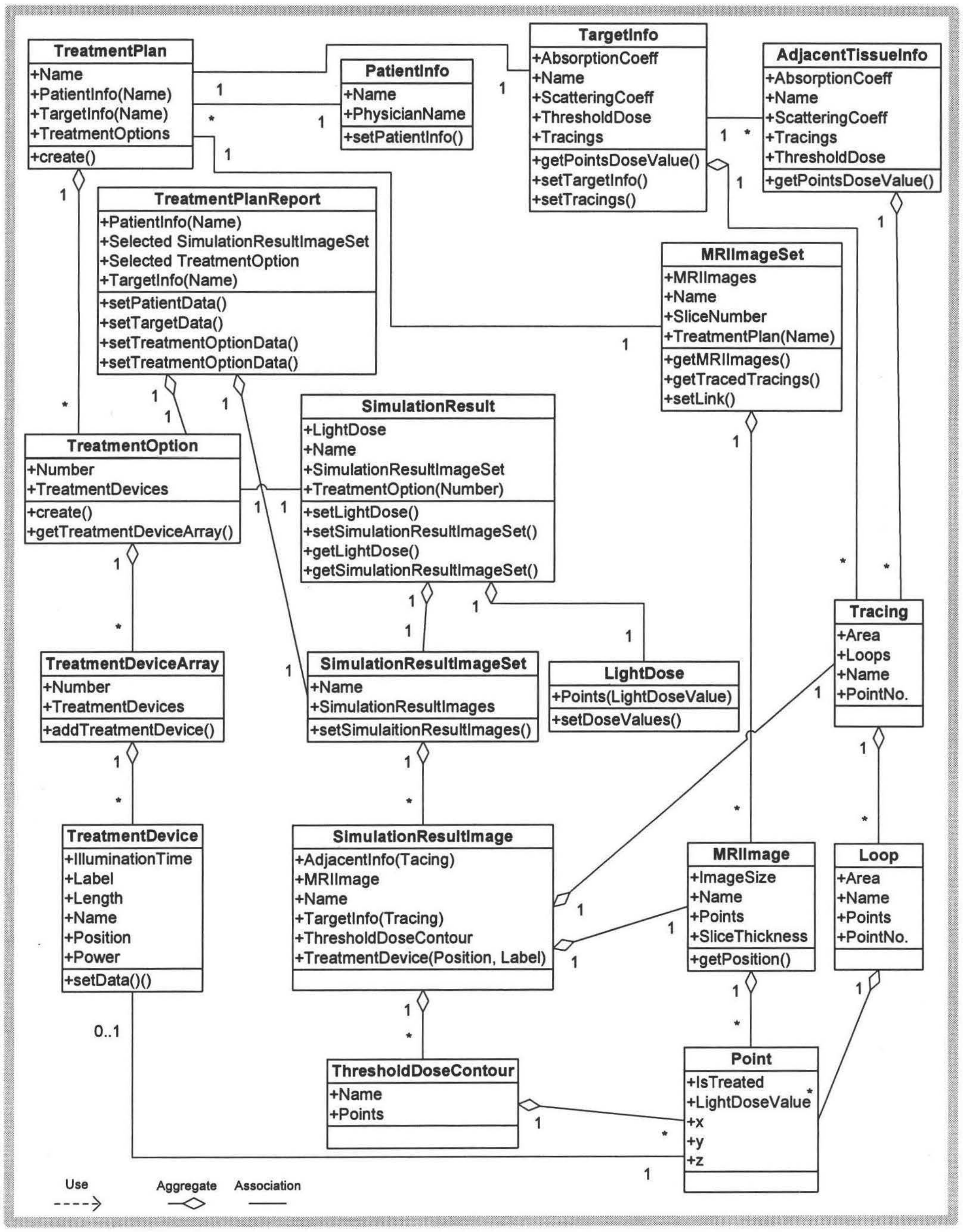


Figure 4.22: Final Domain class diagram: Case Study

Chapter 5

Conclusions and Future Work

This chapter draws the conclusion of this thesis, and suggests future research in this field.

5.1 Conclusion

We conclude this research with the following remarks.

- The User-Centric Software Requirements Specification (UCSRS) approach we developed is applicable to user-centric software systems. It provides an incremental and iterative user oriented engineering method to develop the software requirements.

- The well-defined processes of our UCSRS approach are developed in the spirit of an engineering method. They provide enough detail for people to adopt this approach. Also, the processes lend themselves to being implemented with the aid of software tools (see below).

- Users are kings of user-centric software. The user-centric SRS not only helps to ensure the right software systems are built, but should also help to raise acceptance and productivity.

- Requirements have levels, where user requirements are a subset of high level system requirements. In the pure software case, they are identical. Software requirements should be derived from the analysis of the user requirement (pure software case), they are what the system must have from the developer's point of view.

- Software requirements should just specify the externally perceivable behavior of the system, which are functions or task-specific constraints that directly produce these behaviors.

- In an object-oriented requirements analysis approach, the (functionality of) the system is decomposed into analysis classes. Analysis classes are the externally perceivable functional parts of the system. They are abstractions of the future subsystems or implementation classes. The analysis is the process we use to find "function and data" of the system and allocates them to various analysis classes.

- Software requirements specifications should not restrict the software design and future software evolution. This point is reflected both in user requirements and software requirements specifications. In user requirements, flexible and precise words are advocated, such as using *selection* and *present*, rather than *click and display.* In the SRS, words like button, menu, dropdownlist etc, should not appear in the user interface classes (boundary classes).

5.2 Future Work

There are several areas of this paper that can be extended in future research.

One area is to examine the management process of the SRS, for example, for each user task, the rough time duration and cost. Because the budget is also important to decide the success of a project.

The second area is to prototype the case study using a specific programming language such as MS Visual Basic.net 2005.

Moreover, a computerized toolset is necessary with the following functions: constructing the task scenario table, transforming scenario tables into use cases, constructing the data hierarchy, composing the data hierarchies, and transforming the final data hierarchy to a UML-like class diagram.

Finally, more case studies in various kinds of applications should be applied, to show that UCSRS has universal applicability to most user-centric software systems.

Bibliography

[1] J. R. Abrial, S. A. Schuman, and B. Meyer, *A Specification Language, in On the Construction of Programs.* Cambridge University Press, 1980.

[2] A. Alliance, "http://www.agilealliance.org/."

[3] M. D. Altschuler, T. C. Zhu, J. Li, and S. M. Hahn, "Optimized Interstitial PDT Prostate Treatment Planning with the Cimmino Feasibility Algorithm," *Medical Physics*, vol. 32, no. 12, pp. 3524–3536, 2005.

[4] S. Ambler, "Reduce Development Costs with Use-Case Scenario Testing," *Software Development*, vol. 3, no. 7, pp. 53–61, 1995.

[5] A. I. Anton, "Goal-Based Requirements Analysis," in *ICRE '96: Proceedings of the 2nd International Conference on Requirements Engineering (ICRE '96)*, (Washington, DC, USA), p. 136, IEEE Computer Society, 1996.

[6] A. Bahrami, *Object Oriented Systems Development.* McGraw-Hill/Irwin, 1998.

[7] S. C. Bailin, "Object-Oriented Requirements Analysis," in *Software Requirements Engineering*, vol. 2, pp. 334–355, IEEE Computer Society Press, 1997.

[8] W. BC., "Photodynamic Therapy for Cancer: Principles," *Canadian Journal of Gastroenterology*, vol. 16, no. 6, pp. 393–396, 2002.

[9] B. Beizer, *Software Testing Techniques.* International Thomson Computer Press, 1990.

[10] M. R. Blaha and J. R. Rumbaugh, *Object-Oriented Modeling and Design with UML.* Prentice Hall, 2004.

[11] B. Boehm, "A Spiral Model of Software Development and Enhancement," *Computer*, vol. 21, no. 5, pp. 61–72, 1988.

[12] G. Booch, *Object-Oriented Analysis and Design with Applications.* Addison Wesley, 1994.

[13] I. K. Bray, *An Introduction to Requirements Engineering.* Addison Wesley, 2002.

[14] B. Brudgge and A. H. Dutoit, *Object-Oriented Software Engineering: Using UML, Patterns and Java.* Prentice Hall, 2003.

[15] E. BSSC, "ESA PSSP-05-0-ISSUE 2: SOFTWARE PROJECT DOCUMENTS." European Space Agency, Feb. 1991.

[16] CADMIT, "PDT Treatment Planning Platform, Training Manual," 2006.

[17] S. E. I. Carnegie Mellon, "CMMI v1.1." http://www.sei.cmu.edu, 2002.

[18] J. M. Carroll, *Scenario-based Design: Envisioning Work and Technology in System Development.* John Wiley Sons, 1995.

[19] J. Chen and S. Chou, "An Object Oriented Analysis Technique Based on the UML." http://www.adtmag.com/joop/article.aspx?id=3666, June 2001.

[20] P. Coad and E. Yourdon, *Object-Oriented Analysis.* Prentice Hall, 1991.

[21] A. Cockburn, "Structuring Use Cases with Goals," *Humans and Technology HaT TR95.1*, 1995.

[22] A. Cockburn, *Writing Effective Use Cases.* Addison-Wesley Professional, 2000.

[23] R. Collard, "Test Design," *Software Testing and Quality Engineering*, vol. 1, no. 4, pp. 30–37, 1999.

[24] T. Corner, "Transitioning from Structured Analysis to Object-Oriented Design," in *Proceedings of the Fifth Washington Ada Symposium on Ada*, pp. 151–162, ACM Press, 1988.

[25] J. Crinnion, "Evolutionary Systems Development," *Plenum Press, New York*, 1991.

[26] L. R. A. Daniel R. Windle, *Software Requirements Using the Unified Process.* Prentice Hall, 2002.

[27] A. Dardenne, S. Fickas, and A. van Lamsweerde, "Goal-Directed Requirements Acquisition," *Science of Computer Programming*, vol. 20, no. 1-2, pp. 3–50, 1993.

[28] A. M. Davis, *Software Requirements: Objects, Functions and States, Second Edition.* Prentice Hall PTR, 1993.

[29] M. DeBellis, "User-Centric Software Engineering," *IEEE Expert*, vol. 10, no. 1, pp. 34–41, Feb. 1995.

[30] T. Demarco, *Structured Analysis and System Specification.* Prentice Hall PTR, 1979.

[31] A. Dix, I. Finlay, G. Abowd, and R. Beale, *Human Computer Interaction.* Prentice Hall, New Jersey, 1993.

[32] D. Dolmans, D. Fukumura, and R. Jain, "Photodynamic Therapy for Cancer," *Nature Reviews Cancer*, vol. 3, no. 5, pp. 380–387, 2003.

[33] M. Eva, *SSADM Version 4: A User's Guide.* Mcgraw Hill Book Co Ltd, 1991.

[34] M. Fagan, "Design and Code Inspections to Reduce Errors In Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 258–287, - 1976.

[35] FDA, "General Principles of Software Validation; Final Guidance for Industry and FDA Staff." http://www.fda.gov/cdrh/comp/guidance/938.html, Jan. 2002.

[36] D. C. Gause and G. M. Weinberg, *Exploring Requirements: Quality Before Design.* Dorset House Publishing Company, 1989.

[37] T. Grimm, "The Human Condition: A Justification for Rapid Prototyping," *Time Compression Technologies*, vol. 3, no. 3, May 1998.

[38] C. Haapala, "User Centric Development." http://www.stormingmedia.us/, 1994.

[39] D. Harel, "Statemate: A Working Environment for the Development of Complex Reactive Systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, 1990.

[40] C. Heitmeyer, "Using the SCR* Toolset to Specify Software Requirements," in *Proceedings of the Second IEEE Workshop on Industrial Strength Formal Specification Techniques*, ACM, Oct 1998.

[41] IEEE, *IEEE Standard For Software Reviews - IEEE Std 1028-1997.*

[42] IEEE, *IEEE Standard For Software Testing - IEEE Std 829-1998.*

[43] IEEE, *IEEE Standard Glossary of Software Engineering Terminology - IEEE Std 729-1983.*

[44] IEEE, "IEEE Recommended Practice for Software Requirements Specifications," *IEEE Standard 830-1998 Edition, New York, US*, June 1998.

[45] M. W. Inc, "Merriam-Webster's Online Dictionary." http://www.merriam-webster.com/cgi-bin/dictionary, 2007.

[46] M. Jackson, *System Development.* Prentice Hall, 1983.

[47] M. Jackson, *Software Requirements and Specifications.* Addison-Wesley Professional, 1995.

[48] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley Professional, 2000.

[49] I. Jacobson, *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley Professional, 1992.

[50] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process.* Addison-Wesley Professional, 1999.

[51] M. Keil and C. Erran, "Customer-Developer Links in Software Development," in *Communications of the ACM*, vol. 38(5), pp. 33–34, ACM Press, 1995.

[52] J. C. Kelly, J. S. Sherif, and J. Hops, "An Analysis of Defect Densities Found During Software Inspections," *Journal of Systems and Software*, vol. 17, no. 2, pp. 111–117, - 1992.

[53] G. Kotonya and I. Sommerville, *Requirements Engineering: Processes and Techniques.* Wiley, 1998.

[54] C. Larman, "The Use Case Model: What are the Processes?," *Java Report*, vol. 3, no. 8, pp. 62–72, 1998.

[55] L. A. Maciaszek, *Requirements Analysis and System Design.* Addison Wesley, 2005.

[56] K. L. McGrawand and K. Harbison, *User-centered Requirements: the Scenario Based Engineering Process.* Lawrence Erlbaum, 1997.

[57] S. J. Mellor and S. Shlaer, *Object Life Cycles: Modeling the World In States.* Prentice Hall PTR, 1991.

[58] B. Meyer, *Object-Oriented Software Construction.* Prentice Hall PTR, 2000.

[59] J. Mylopoulos, L. Chung, and E. Yu, "From Object-Oriented to Goal-Oriented Requirements Analysis," *Commun. ACM*, vol. 42, no. 1, pp. 31–37, 1999.

[60] NEGMA. http://www.prostatepdt.com/, 2007.

[61] OGI. http://omlc.ogi.edu/pdt/, 2007.

[62] OMG, "Unified Modeling Language 2.0 Specification." http://www.uml.org/, 2007.

[63] K. Orr, *Structured Requirements Definition.* K. Orr, 1981.

[64] G. Overgaard and K. Palmkvist, *Use Cases: Patterns and Blueprints.* Addison-Wesley Professional, 2004.

[65] C. Pahl, "Adaptive Development and Maintenance of User-Centric Software Systems," *Information and Software Technology*, vol. 46, no. 14, pp. 973–986, 2004.

[66] D. L. Parnas, "Tabular Representation of Relations," Tech. Rep. CRL Report 260, Telecommunications Research Institute of Ontario (TRIO), McMaster University, 1992.

[67] D. L. Parnas and J. Madey, "Functional Documents for Computer Systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, 1995.

[68] E. Programming, "Acceptance Test." http://www.extremeprogramming.org/.

[69] S. Robertson and J. Robertson, *Mastering the Requirements Process.* Addison Wesley Professional, 2006.

[70] C. Rolland, C. Souveyet, and C. B. Achour, "Guiding Goal Modeling Using Scenarios," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, Dec 1998.

[71] D. Ross, "Applications and Extension of SADT," *IEEE Computer*, vol. 18, no. 4, pp. 25–34, April 1985.

[72] W. W. Royce, "Managing the Development of Large Software Systems: Concepts and Techniques," *Technical Papers of Western Electronic Show and Convention (WesCon).*, vol. August, no. 1, pp. 25–28, 1970.

[73] RTCA and EUROCAE, "Software Considerations in Airborne Systems and Equipment Certification," Dec. 1992.

[74] J. Rumbaugh, "Getting Started: Using Use Cases to Capture Requirements," in *Software Requirements Engineering*, vol. 2nd, pp. 153–157, IEEE Computer Society Process, 1997.

[75] J. Rumbaugh, M. Blahaby, W. Lorensen, F. Eddy, and W. Premerlani, *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[76] E. Seidewitz and M. Stark, "Toward a General Object-Oriented Software Development Methodology," in *ACM SIGAda Ada Letters*, vol. VII, Issue 4, pp. 54–67, ACM Press, 1987.

[77] S. Shlaer and S. J. Mellor, *Object Oriented Systems Analysis: Modeling the World in Data.* Prentice Hall PTR, 1988.

[78] I. Sommerville, *Requirements Engineering: A Good Practice Guide.* John Wiley and Sons Ltd, 1997.

[79] SRI, "PVS Specification and Verification System." http://pvs.csl.sri.com/.

[80] A. Sutcliffe, "Scenario-Based Requirements Analysis," *Requirements Engineering Journal*, vol. 3, no. 1, pp. 48–65, 1995.

[81] R. H. Thayer and M. Dorfman, *Software Requirements Engineering, 2nd Edition.* Wiley-IEEE Computer Society Pr, 2000.

[82] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Guided Tour," in *RE '01: Proceedings of the 5th IEEE International Symposium on Requirements Engineering*, p. 249, IEEE Computer Society, 2001.

[83] A. van Lamsweerde, "Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice," *Requirements Engineering*, vol. 6, no. 11, pp. 4–7, 2004.

[84] A. Wassyng and M. Lawford, "Lessons Learned from a Successful Implementation of Formal Methods in an Industrial Project," in *FME 2003: International Symposium of Formal Methods Europe Proceedings*, vol. 2805 of *Lecture Notes in Computer Science*, pp. 133–153, Springer-Verlag, Aug 2003.

[85] B. Whelan, W. Whelean, and S. Davidson, "Treatment Planning Platform for Photodynamic Therapy: Architecture, Function, and Validation," *Optical Methods for Tumor Treatment and Detection: Mechanisms and Techniques in Photodynamic*, vol. 4612, pp. 85–92, 06 2002.

[86] K. E. Wiegers, *Software Requirements.* Microsoft Press, 1999.

[87] Wikipedia, "Joint Requirements Development." http://en.wikipedia.org/, 2007.

[88] Wikipedia, "Photodynamic Therapy." http://en.wikipedia.org/wiki/, 2007.

[89] Wikipedia, "Software Prototyping." http://en.wikipedia.org/wiki/, 2007.

Appendix A

The SRS Template

A.1 Introduction

The introduction provides an overview of the entire SRS. The overview contains the following parts: purpose of the document, scope of the software, definitions and acronyms, other documents referred by this document, and the overview of the structure of the SRS.

A.1.1 Purpose

This subsection includes the statement of the purpose of the SRS and the statement of the intended audience for the SRS

A.1.2 Scope of the Software

This subsection identifies the name of the software to be built, explains what the software (the system) will do, describes the objectives of the application of the software. Reference URD 1.2. Vision Statement

A.1.3 Definitions, Acronyms, and Abbreviations

This subsection provides definitions of all terms, acronyms, and abbreviations. Derived from URD 1.6 Glossary

A.1.4 References

This subsection lists all the other documents referenced which should include the title, date, publishing organization.
URD

A.1.5 Overview

This subsection describes what the rest of the SRS contains, and how it is organized.

A.2 Overall Description

This section of SRS provides a complete, abstract view of the software system to be built. It describes the general factors that affect the software product, which are defined in detail in section 3 of the SRS. Overall description includes six sections: product perspective, product functions, user characteristics, constraints, assumptions and dependencies, and apportioning of requirements.

A.2.1 Product Perspective

This subsection of the SRS demonstrates a perspective of the software and its environment with a system context diagram if it is stand alone, or a block diagram if it is a part of a larger system. Also, this subsection provides a general description of interfaces of the software system, including system interfaces, user interfaces, hardware interfaces, software interfaces, communication interfaces.
Reference URD 1.4 System Constraints, URD 2.1 System Context Diagram

A.2.2 Product Functions

This subsection of the SRS provides a summary of the major functions of the software to be built.
Reference URD 1.2. Vision Statement

A.2.3 User Characteristics

This subsection describes the general characteristics of the intended users of the software including the educational level, experience, and technical expertise.
Reference URD 2.1 Actors and their Profiles

A.2.4 General System Constraints

This subsection provides a general description of the constraints that will limit the development options.
Reference URD 1.4

A.2.5 Assumptions and Dependencies

This subsection lists each of the factors that affect the requirements stated in the SRS. These factors are not design constraints on the software but are, rather, any changes to them that can affect the requirements in the SRS. For example, an assumption may be that a specific operating system will be available on the hardware designated for the software. If, the operating system is not available, the SRS would then have to change accordingly.

A.2.6 Apportioning of Requirements

This subsection states requirements that may be delayed until future versions of the software.

A.3 Specific Requirements

This section of SRS specifies the results of our analysis work including the specific functional requirements, system constraints. All functions of the software system should be specified, together with the inputs and outputs of the software system. In our approach, this corresponds to all the class specifications.

The specifications in this section specify what we call the software requirements distinguishing from user requirements. They should be in a level of detail sufficient to enable designers to design a system accordingly, and tester to test that the system satisfies those requirements. Although these specifications are the definitions of the software solution of the problem according to the developers' view, they should be also externally perceivable by users, operators, and other external systems.

A.3.1 External Interfaces

This subsection presents details of hardware interfaces and communication interfaces. For various interfaces, included contents and formats can vary, they should contain the following:

- Name of item and purpose.
- Valid range, accuracy, and/or tolerance.
- Units, type.
- Timing.
- Relationship to other inputs/outputs.
- Data formats.
- Command formats.
- End messages.

A.3.2 Boundary Classes

This subsection specifies the detailed description of the boundary classes (user interfaces and system interfaces). Each class has a set of attributes and function, which are documented using specific specification language and standard form. The class specification template may reference Figure 3.34.

A.3.3 Domain Classes

This subsection specifies the detailed description of the domain classes.

A.4 Performance

This section specifies requirements about both static and dynamic quantifiable attributes of the systems, such as the number of simultaneous users to be supported, the number of transactions , the amount of data to be processed, response time and accuracy. All of them should be stated in measurable terms, such as "all the transaction shall be processed in less than 1 second."

Reference URD 1.4, URD 5.

A.5 Design Constraints

This section specifies the design constraints that are imposed by other standards, hardware limitations, operation limitations etc. For example, the software shall be a web based application. Reference URD 1.4, URD 5.

A.6 Reliability

This section specifies robustness-the degree to which a system can function correctly in different conditions (e.g. acceptable mean time to failure), safety-a measure of the absence of catastrophic consequences to the environment (e.g. consequence of system failure, if the backup systems are available), and security requirements of the system-the factors that protect the software from malicious access, use, modification, destruction, or disclosure (e.g. utilize certain cryptographical techniques, system log).

Reference URD 1.4, URD 5.

A.7 Maintainability

This section specifies the ability to change the system to deal with new technology or to fix defects. Reference

Reference URD 1.4.

A.8 Portability

This section should specify the ease attributes of porting the software to another hardware or software environments. This may include: percentages of components with host-dependent code and host-independent code; use of a proven portable language, use of a particular operating system.

Reference URD 1.4.

A.9 Legal

This section specifies the requirements concerned with licensing, regulation, and certificate issues. They may include the report format, data naming, accounting procedures and audit tracing. For example, in a payroll system, an audit trace requirement states that all changes to a payroll database must be recorded in a trace file with before and after values.

Reference URD 1.4

A.10 Other Requirements

This section specifies other requirements that are not mentioned in precious sections.

Appendix B

The Partial SRS of PDT Treatment Planning Software

Revision History

Name	Date	Reason	Version

B.1 Introduction

B.1.1 Purpose

This document will define all software requirements for the PDT Treatment Planning Software System. The intended readers include the users, developers, and testers.

B.1.2 Scope of the Software

The PDT Treatment Planning Software System will provide users the ability to make treatment plans for cancer patients who need a radical treatment.

Reference URD 1.2. Vision Statement

B.1.3 Definitions, Acronyms, and Abbreviations

MRIImageSet	=(Name, MRIImages, SliceNumber, TreatmentPlan(Name)) //A series of MRI Images for certain patient, usually the T2-weight series is used for treatment planning//.
MRIImage	=(Name, Points, SliceThickness, ImageSize) //A slice of image scanned by Magnetic resonance imaging (MRI) Machine.//
PatientInfo	=(Name, PhysicianName) //A patient whose treatment plan is produced by the treatment planning software.//
TreatmentOption	=(Number, TreatmentDeviceArray) //A treatment option is a set of treatment parameters.//
TreatmentDeviceArray	=(Name, TreatmentDevices, DeviceNumber) //An array of treatment devices//
TreatmentDevice	=(Name, Label, Length, Power, IlluminationTime, Position) //A light delivery source//

Derived from URD 1.6 Glossary

B.1.4 References

- "Recommended Practice for Software Requirements Specifications", IEEE standard 830-1998 edition, New York, US"
- User Requirements Document, URD
- Analysis Model Document, AMD

B.1.5 Overview

This document will present all the specific requirements for the PDT Treatment Planning Software System. Section 2 describe the general factors that affect the software and its requirements. Specific functional requirements are specified in section 3 with the form of class specifications, categorized for each class. System constraints are also specified in section 3. The user requirements documents (URD) and analysis model document (AMD) are maintained as attachments of this document.

B.2 Overall Description

B.2.1 Product Perspective

The PDT Treatment Planning Software System provides users a computerized tool that allows users to make a near optimal treatment plan for a specific patient.

B.2.1.1 System Interfaces

No external system interfaces are connected.

Reference URD 1.4 System Constraints, URD 2.1 System Context Diagram

B.2.1.2 User Interfaces

The user interface shall require a stand alone software environment.

B.2.1.3 Hardware Interface

No hardware interfaces are connected.

B.2.1.4 Software Interface

Finite Element Method computation software is required.

B.2.1.5 Communication Interface

No communication interfaces are required.

B.2.1.6 Memory

No specific memory requirements have been specified.

B.2.2 Product Functions

The major functions of the software are

- Allow users to simulate a treatment.
- Allow users to generate a treatment plan report.

- Allow users to reconstruct treatment simulation.

Reference URD 1.2. Vision Statement

B.2.3 User Characteristics

The users are familiar with medical image software.
Reference URD 2.1 Actors and their Profiles

B.2.4 General System Constraints

The software will be certified by FDA.
Reference URD 1.4

B.2.5 Assumptions and Dependencies

None.

B.3 Specific Requirements

This part shall use the template as stated in appendix A.

B.3.1 External Interfaces

No specific external interfaces.

B.3.2 Boundary Classes

B.3.2.1 C1000, MainUB

The MainUB class shall provide each user type the ability to navigate its central boundary objects.

B.3.2.1.1 C1000-F1, SelectMakeTreatmentPlan()

Input	**Type**	**Reference**
User selection	User event	-
Output	**Type**	**Uses**
Message of presenting a TreatmentPlanUB	Message	B.3.2.2.13

Requires

User is logged in.

Ensures

System shall perform the present function of a TreatmentPlanUB.

B.3.2.2 C2000, TreatmentPlanUB

The TreatmentPlanUB class shall provide the user the ability to navigate all the functions related to making a treatment plan, including *Enter PatientInfo & TargetInfo, Link MRIImageSet, Define Target, Set TreatmentOption, Do Simulation and Generate TreatmentPlanReport.*

B.3.2.2.1 C2000-A1, MRIImageSetNames

A TreatmentPlanUB shall maintain a set of available MRIImageSet names to be chosen.

Attribute	**Type**	**Reference**
MRIImageSetNames	A set of strings	B.3.3.2

B.3.2.2.2 C2000-A2, SelectedMRIImageSetName

A TreatmentPlanUB shall maintain a selected MRIImageSet name, the default value is first one of the MRIImageSetNames or null.

Attribute	**Type**	**Reference**
SelectedMRIImageSetName	String	-

B.3.2.2.3 C2000-A3, CurrentPatientInfo(Name)

A TreatmentPlanUB shall maintain a current PatientInfo(Name), the default value is null. The user shall be allowed to enter a string to change the value.

Attribute	**Type**	**Reference**
CurrentPatientInfo(Name)	String	-

B.3.2.2.4 C2000-A4, CurrentPatientInfo(PhysicianName)

A TreatmentPlanUB shall maintain a current PatientInfo(PhysicianName), the default value is null. The user shall be allowed to enter a string to change the value.

Attribute	**Type**	**Reference**
CurrentPatientInfo(PhysicianName)	String	-

B.3.2.2.5 C2000-A5, CurrentTreatmentOption(Number)

A TreatmentPlanUB shall maintain a CurrentTreatmentOption(Number), the default value is 1. The user shall be allowed to change the value.

Attribute	**Type**	**Reference**
CurrentTreatmentOption(Number)	Integer	-

B.3.2.2.6 C2000-A6, CurrentTargetInfo(Name)

A TreatmentPlanUB shall maintain a CurrentTargetInfo(Name), the default value is null. The user shall be allowed to change the value.

Attribute	**Type**	**Reference**
CurrentTargetInfo(Name)	String	-

B.3.2.2.7 C2000-A7, CurrentTargetInfo(AbsorptionCoeff)

A TreatmentPlanUB shall maintain a CurrentTargetInfo(AbsorptionCoeff), the default value is 0.0. The user shall be allowed to change the value.

Attribute	**Type**	**Reference**
CurrentTargetInfo(AbsorptionCoeff)	Real	-

B.3.2.2.8 C2000-A8, CurrentTargetInfo(ScatteringCoeff)

A TreatmentPlanUB shall maintain a CurrentTargetInfo(ScatteringCoeff), the default value is 0.0. The user shall be allowed to change the value.

Attribute	**Type**	**Reference**
CurrentTargetInfo(ScatteringCoeff)	Real	-

B.3.2.2.9 C2000-A9, CurrentTargetInfo(ThresholdDose)

A TreatmentPlanUB shall maintain a CurrentTargetInfo(ThresholdDose), the default value is 0.0. The user shall be allowed to change the value.

Attribute	**Type**	**Reference**
CurrentTargetInfo(ThresholdDose)	Real	-

B.3.2.2.10 C2000-F1, SelectLinkMRIImageSet()

Input	**Type**	**Reference**
User selection	User event	-
Output	**Type**	**Uses**
Message of presenting the list of MRIImageSetNames	Message	B.3.2.2.11

Requires

A TreatmentPlan is active in the system.

Ensures

System shall perform PresentListOfMRIImageSets function.

B.3.2.2.11 C2000-F2, PresentListOfMRIImageSets()

Input	**Type**	**Reference**
MRIImageSetNames	A set of strings	-
Output	**Type**	**Uses**
A presentation of the MRIImageSetNames	Presentation	-

Requires

-

Ensures

If the MRIImageSetNames is not null, the system shall present a list of available MRIImageSet names, and set SelectedMRIImageSetName=MRIImageSetNames[1];
Else the system shall present the error information and stop current function.

B.3.2.2.12 C2000-F3, SelectMRIImageSet()

Input	**Type**	**Reference**
User selection	User event	-
Output	**Type**	**Uses**
SelectedMRIImageSetName	String	-
Message of setting the link of the selected MRIImageSet	Message	B.3.3.2.4
Message of presenting an MRIImageSetPresentUB	Message	B.3.2.3.2

Requires

The system presents a list of available MRIImageSet names (Ref. B.3.2.2.11).

Ensures

The system shall set SelectedMRIImageSetName=the selected MRIImageSet Name.
If the selected MRIImageSet is reachable and readable, the system shall perform the setLink function (Ref. B.3.3.2.4) and the presenting an MRIImageSetPresentUB function (Ref. B.3.2.3.2).
Else if the retried times < 3, the system shall allow the user to retry current function or choose to stop current function.
Else the system shall notify user the error information and stop current function.

B.3.2.2.13 C2000-F4, Present()

Input	**Type**	**Reference**
-	-	-
Output	**Type**	**Uses**
A presentation of a TreatmentPlanUB	Presentation	-

Requires

-

Ensures

System shall present A3-A9 data items.
System shall present required function selections F1, F5-F10.

B.3.2.2.14 C2000-F5, CompleteDataItems()

Omitted.

B.3.2.2.15 C2000-F6, SelectDefineTarget()

Omitted.

B.3.2.2.16 C2000-F7, SelectSetTreatmentOption()

Omitted.

B.3.2.2.17 C2000-F8, SelectTreatmentOption()

Omitted.

B.3.2.2.18 C2000-F9, SelectPresentSimulationResuslt()

Omitted.

B.3.2.2.19 C2000-F10, SelectGenerateTreatmentPlanReport()

Omitted.

B.3.2.3 C3000, MRIImageSetPresentUB

The MRIImageSetPresentUB class shall provide the ability to present the selected MRIImageSet.

B.3.2.3.1 C3000-A1, SelectedMRIImageSet

A MRIImageSetPresentUB shall maintain an MRIImageSet to be presented, the default value is null.

Attribute	Type	Reference
SelectedMRIImageSet	MRIImageSet	-

B.3.2.3.2 C3000-F1, Present()

Input	**Type**	**Reference**
An MRIImageSet(Name)	String	-
Output	**Type**	**Uses**
Message of getting the MRIImages from the selected MRIImageSet	Message	B.3.3.2.5
Presentation of the selected MRIImageSet	Presentation	-
Requires		
The input MRIImageSet(Name) is not null.		
Ensures		
The system shall present the MRIImages[1] and the list of MRIImages of the selected MRIImageSet.		

B.3.3 Domain Classes

B.3.3.1 C1, TreatmentPlan

The TreatmentPlan class represents a treatment plan for a patient in the treatment planning software system. The system shall maintain a current TreatmentPlan for current patient.

B.3.3.1.1 C1-A1, Name

The system shall maintain a TreatmentPlan name for each TreatmentPlan.

Attribute	**Type**	**Reference**
Name	String	-

B.3.3.2 C2, MRIImageSet

The MRIImageSet class represents the MRI image sets of patients that are managed by the treatment planning software system. Each MRIImageSet includes a collection of MRIImages.

B.3.3.2.1 C2-A1, TreatmentPlan(Name)

The system shall maintain a TreatmentPlan name for each MRIImageSet, the default value is null.

Attribute	**Type**	**Reference**
TreatmentPlan(Name)	String	B.3.3.1.1

B.3.3.2.2 C2-A2, Name

The system shall maintain a name for each MRIImageSet.

Attribute	**Type**	**Reference**
Name	String	-

B.3.3.2.3 C2-A3, MRIImages

Each MRIImageSet contains a set of MRIImages.

Attribute	**Type**	**Reference**
MRIImages	MRIImage	-

B.3.3.2.4 C2-F1, SetLink()

Input	**Type**	**Reference**
Current TreatmentPlan(Name)	String	
Output	**Type**	**Uses**
-		
Requires		
Current TreatmentPlan(Name) is not null.		
Ensures		
TreatmentPlan(Name) of Current MRIImageSet is set.		

B.3.3.2.5 C2-F2, GetMRIImages()

The system shall retrieve all of the MRIImages of an MRIImageSet.

Input	**Type**	**Reference**
-	-	
Output	**Type**	**Uses**
A set of MRIImages	MRIImage	-
Requires		
-		
Ensures		
All of the MRIImages of an MRIImageSet are retrieved.		

B.4 Performance

No specific requirements.

B.5 Design Constraints

The application shall be designed to run on MS Windows.

Reference URD 1.4, URD 5.

B.6 Reliability

This system shall have no more than one failure per calendar week.

Reference URD 1.4, URD 5.

B.7 Maintainability

No specific maintainability requirements have been specified.

Reference URD 1.4.

B.8 Portability

No specific requirements required.

Reference URD 1.4.

B.9 Legal

This software must meet the FDA standard, "General Principles of Software Validation; Final Guidance for Industry and FDA Staff."

Reference URD 1.4

B.10 Other Requirements

No other requirements have been specified.