SCENARIO-DRIVEN REQUIREMENTS ENGINEERING: METHOD AND TOOL

Meng Tian

A Thesis

in

The Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements for the Degree of
Master of Computer Science at
Concordia University
Montréal, Québec, Canada

February 2003

ABSTRACT

Scenario-Driven Requirements Engineering: Method and Tool

Meng Tian

Scenarios have been used in different areas such as military, economy, software engineering, human computer interaction, and theatrical arts. In software and user interface requirements, a scenario is a description of a person's interaction with a system. It describes what the user wants to do but does not describe how this is to be done. Scenario-based requirements engineering brings an integrated answer to the following questions: Who, What, Where, When, and Why. This thesis illustrates different meanings and describes different examples that illustrate the power of scenarios in different areas. In addition, a survey of the current existing tools for scenario-based requirement engineering is presented. In the second part, we propose a progressive, iterative and interleaved process of using scenario at different requirement engineering stages including elicitation, analysis and validation. This process model has been applied to our SUCRE (Scenario and Use Cases for User-Centered Requirement Engineering), which is an XML-based system for scenario-driven requirement engineering. SUCRE is a tool for working with scenarios. Within the SUCRE system, scenarios are stories that capture information about users and their tasks, including the context of use. In our SUCRE system, scenarios are stored in an XML-based database and described using XML notation. Besides the SUCRE system prototype, we discuss the XML-based structure of the scenario database we are developing. The paper concludes with a discussion on the evolution of the process model and the XML-based SUCRE system.

Acknowledgments

I wish to express my deepest appreciation to my supervisor Dr. Ahmed Seffah for providing me such a wonderful opportunity to work on this research area and for his assistance in the preparation of this manuscript. It is his supervision, encouragement, and support that have made this work possible.

I would like to thank Mrs. Carla De Waele for her help in developing part of our framework. My great gratitude is also extended to Dr. V. S. Alagar for his comments and continuous support.

My sincere thanks to my parents for their love and support. Many thanks to all my friends for their encouragement and help.

Contents

List of Figures

Number *Page*

List of Tables

Number *Page*

Chapter 1 – Scenario-Based Requirements Engineering: Terminology and Foundations

1.1 Origins

According to Moltke and Clausewitz [1], the concept of scenarios has already been in existence for more than 2000 years. The earliest outlines of possible scenarios, or what could be interpreted as such, were concerned with military survival and their aim of vanquishing the enemy. At that time, scenarios were primarily embodied as military strategic planning. For example, attack your enemy where he is weakest, build on your strength [1]. Then, in the early seventies, the scenario concept appeared in the field of economy. Specifically, it is the models of military strategic planning that entered the business environment. Presently, scenarios have been applied to widespread areas and disciplines such as software engineering, human computer interaction, digital arts, and organization operation.

1.2 General Definition and Example

The American Heritage College Dictionary gives the following definition of scenario [2]:

1) An outline of a dramatic or literary plot
2) A screenplay
3) An outline or model of an expected or supported sequence of events.

For example,

> *Joe is flying to Sydney. On the way to the airport, he found that he did not have enough money for a taxi. He went to the local ATM and identified himself. He specified that he wanted $100 from his savings account.*

According to Carroll, scenarios have characteristic elements: setting (context of the environment), agents or actors, sequences of actions and events, and goals. Every scenario involves at least one agent and at least one goal. The agent performs a sequence of activities to achieve a certain goal in the circumstances of the setting [3].

For the flying to Sydney scenario:

Setting: the setting is a taxi, with a person seated inside who is going to the airport. The setting also includes the amount of money at hand.

Agents or actors: Joe is the only agent in this example.

Goals: the goals are going to the airport and flying to Sydney. A subgoal is having money to pay the taxi driver.

Actions and events: taking a taxi is an action that facilitates the goal of going to the airport. Going to an ATM, identifying himself, and withdrawing $100 are all actions that facilitate the goal of paying the taxi driver.

Other definitions of scenarios are:

> *A **scenario** is a time-ordered sequence of object interaction to fulfill a specific need [30].*

> *A **scenario** is an instantiation of a generic task type, or a series of generic tasks linked by transitions. It specifies the characteristics of the group that should carry it out, and the social protocols which should be in place. It describes what the users should (try to) do at the requirements level, but not how they should do it at any of the lower levels of the framework [31].*

Scenarios can be used to help understand socio-technical systems [4, 5], to elicit and validate requirements [6, 7, 8], to perform behavioral analyses [6, 9], to analyze requirements [10, 11], to analyze software architecture [12], to understand requirements [13], and to develop initial OOA models [15]. Adopted under different circumstances, scenarios mean a different thing. Some authors define scenarios as sequences of events [14, 15], while other authors interpret it as behavior drawn from use cases [16]. In addition, different authors present scenarios in different forms. Some use tabular or diagrammatic notations to present scenarios [14, 15]. Others present scenarios as user interface storyboards [17] or natural language description [16].

The following sections of this chapter illustrate several usages of scenarios, and how scenarios are defined for these usages.

1.3 Scenarios in Software Engineering

In software engineering, scenarios have been used to help elicit requirements, analyze requirements, detect ambiguities in requirements, uncover missing features and inconsistencies among specified features, and verify and validate requirements [3]. In the following subsections, we will give examples to illustrate how scenarios are used in these aspects.

1.3.1 Scenarios in Requirement Elicitation

Scenarios are helpful in requirements elicitation as people can relate to these more readily than an abstract statement of what they require from a system. Furthermore, scenarios are particularly useful for adding details to an outline requirement description.

Usually, the requirement elaboration process involves three components as described by Colin Potts et al.: document requirements, discuss requirements, and evolve requirements [13]. As shown in Figure 1.1, it is an iterative process.

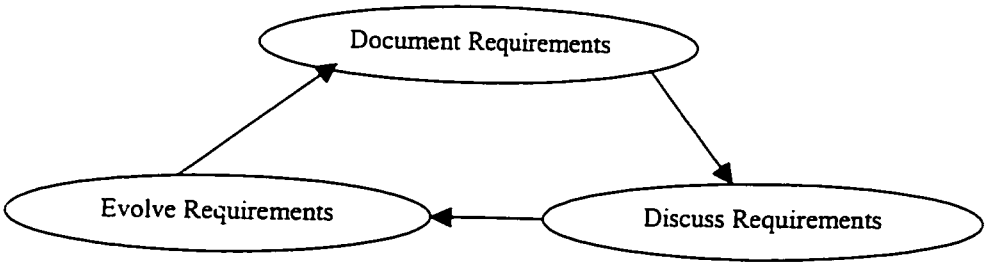

Figure 1.1 Iterative requirement elaboration process

Documenting requirements consists of gathering information from stakeholders, studying existing documents, and drafting requirement documentation. The requirement document contains domain knowledge of the system to be constructed, system constraints, background information, and existing documentation.

Discussing requirements involves presenting the generated requirements to stakeholders, collecting suggestions and opinions via questions, answers, and reasons, and finally, achieving the agreement. There are many ways to demonstrate the requirements such as

prototypes, text description accompanied by graphical visualization. In this stage, the discussion results need to be recorded for future reference and refinement.

Finally, according to the discussion results, evolve requirements by freezing a requirement or changing it, or adding more information into the requirements.

Markus and Uolevi [18] proposed a systematic approach, which manipulates the three components mentioned above. In addition, it uses prototype techniques to present the gathered requirements. The approach combines prototypes, use cases, and scenarios all together to elicit initial software requirements. Here is a definition of scenarios given by Markus and Uolevi [18]:

> *Scenarios are defined as narrative descriptions or stories in a specific context bound in time [19] or as specific instances containing descriptions of the environment, the context, the actors, and the actions with definite beginning and end points [20]. They are also presented as specific instances of use cases where a scenario describes a path of actions through a use case [21].*

The steps [18] of the approach are shown in Figure 1.2. It demonstrates the gradual process from "what a system does" to "how to accomplish the functionality in the system."

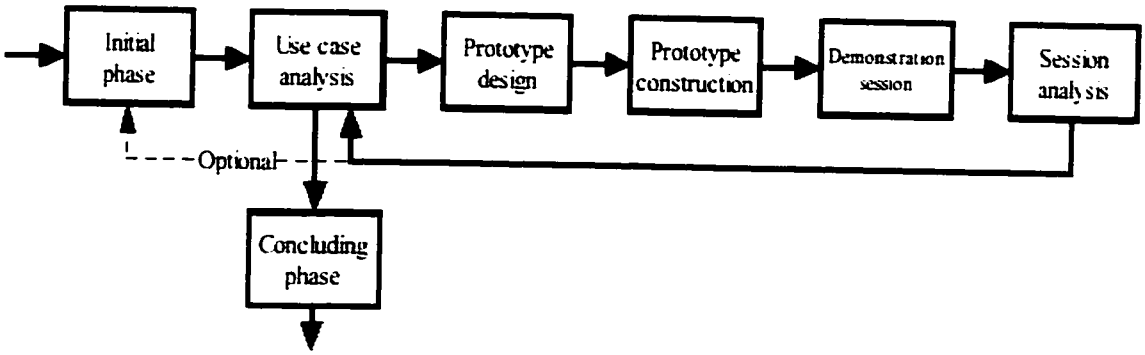

Figure 1.2 Iterative prototyping processes [18]

Initial phase: This corresponds to the document requirements component in Figure 1.1. This phase also identifies the user classes and the primary tasks that each user class needs to accomplish [18]. The use cases and scenarios can be documented to templates, which can be based on the ones presented by Kulak and Guiney [21]. This phase can be executed once, or iterated through the process, depending on whether major changes occur in the system definition [18].

Use case analysis: This involves analyzing the generated use cases and requirement documents to decide whether or not the prototyping process should be continued. If the goals of the prototyping process set in the initial phase are not achieved, or they are deemed unreasonable to prototype, the process iterations are aborted and the concluding phase is executed [18]. Otherwise, the use cases to prototype have to be selected.

Prototype design: In this stage, scenarios are generated based on the prototype context, which describes the prototype environment. Prototype structure design starts with a prototype skeleton. A prototype skeleton describes the problem and use case events [18].

It answers the "What" question – what the system does during the course of requirement elicitation. Scenarios are used to answer the "How" question – how to manipulate the system. A use case event could have many scenarios, as shown in Figure 1.3. All of these scenarios present design options and alternatives for a certain problem. According to Markus [18], the starting and end points of each scenario are determined in the prototype skeleton, so that each scenario can focus on a specific problem area and the common, overlapping parts in scenarios need not be designed multiple times.

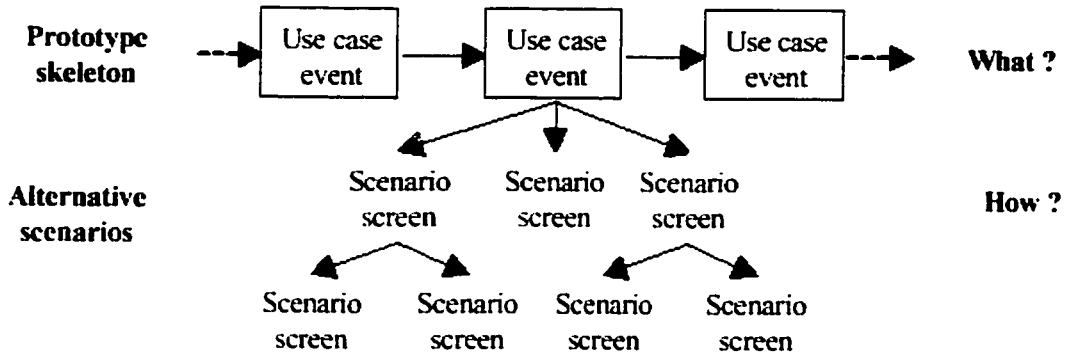

Figure 1.3 A tree-like scenario structure [18]

Prototype construction: Obviously this phase is to implement prototypes according to the prototype design. The prototypes should be able to link the scenario screens together so that an alternative scenario can be selected from any screen of the prototype.

Prototype demonstration session: This phase is the same as the discuss requirements component in Figure 1.1. First, the system analyst demonstrates to stakeholders how to use the prototype. Then, the stakeholders start the prototype over. The system analyst then stimulates discussion about each screen by asking questions, presenting the

scenarios, recording stakeholders' reactions, and documenting discussion and feedback for each possible solution.

Session analysis: The feedback about a specific scenario screen describes the suitability of the solution that is presented by the scenario to the problem that is presented by the use case. The results from the last phase are analyzed to evolve and refine use case events and scenarios of the requirements. As shown in Figure 1.2, the prototype may be used to serve as the basis for the iteration, beginning from the use case analysis phase.

Concluding phase: The output of this phase is a single document, which contains system description, the prototypes constructed, the problems prototyped, used use cases, scenarios, feedback recorded in the demonstration session, and other requirement documents. This document will be available for the following stages of the system development and future reference [18].

The proposed method is summarized in Table 1.1. This table illustrates the inputs, outputs, and tasks of each phase. The bolded letters in the Inputs column indicate new inputs to the process; otherwise, inputs are the outputs from previous phases.

Table 1.1 Input, output, tasks of each phase of the prototyping process [18]

Inputs	Phase	Outputs
Domain knowledge **Stakeholder vision** **Documents**	*Initial phase*	Prototype goals System context User classes Use case descriptions Scenarios Other requirements
Prototyping goals System context User classes Use case descriptions Other requirements **Analyst knowledge**	*Use case analysis*	Go / no-go decision Description of the selected use case
Description of the selected use case System context	*Prototype design* *1. Design context* *2. Select scenarios* *3. Design prototype structure* *4. Design screens*	Prototype context Scenarios to implement Screen descriptions Scenario storylines Prototype structure
Prototype context Description of the selected use case Scenarios to implement Screen descriptions **Possible previous version of prototype** **Multimedia objects**	*Prototype construction* *1. Construct screens* *2. Link screens* *3. Test prototype*	Use case prototype New multimedia objects
Use case prototype Scenario storylines **Stakeholders views** **Facilitator skill**	*Prototype demonstration session* *1. Introduction* *2. Prototype presentation* *3. Conclusion*	Session recordings
Session recordings	*Session analysis*	New use cases More detailed use cases New scenarios Other requirements
Prototyping goals System context Use case descriptions Other requirements	*Concluding phase*	Process report

This method has been used to experiment with a system that handles the mobile payments of car parking fees. As Markus et al. state: "A structured approach to scenarios, when accompanied by throwaway prototyping of the user interface, can provide a satisfactory expression and rigorous analysis of a user's functional requirements. [18]"

1.3.2 Scenarios in Requirement Validation

Scenarios can also be used to validate requirements. This section introduces a method and tool proposed by Sutcliffe et al. [8] to show how scenarios can help in validation. In this method and tool, scenarios are defined as:

> *One sequence of events that is one possible pathway through a use case. Many scenarios may be specified for one use case and each scenario represents an instance or example of events that could happen. Each scenario may describe both normal and abnormal behavior. [8]*

Scenarios can be seen as pathways through a specification of system usage and representation of the system behavior [8]. Thus, by inspecting the behavior of the future system, they enable validation.

The method, expressed in data flow diagram (DFD) format is shown in Figure 1.4:

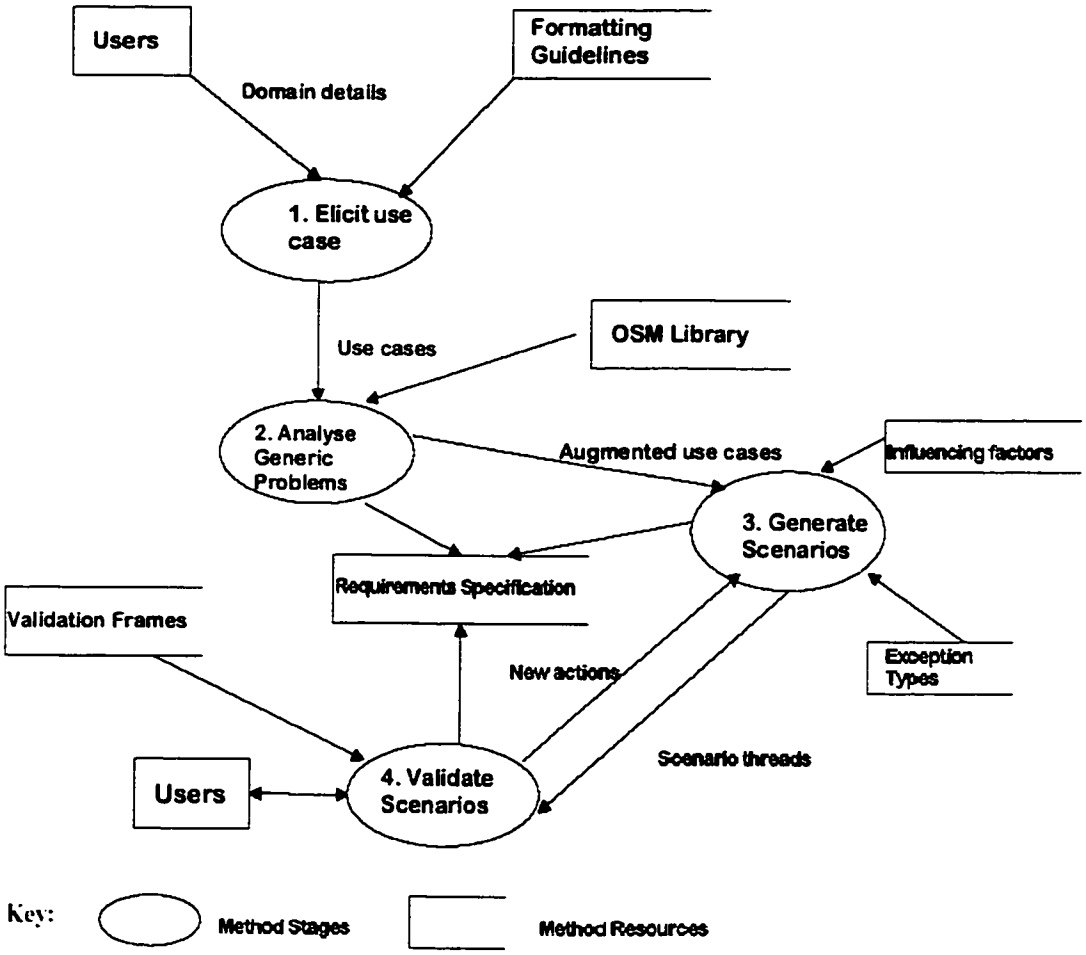

Figure 1.4 Method stages for scenario-based requirement engineering [8]

In general, the method is a 4-step process [8]:

1. Elicit and document use cases
2. Analyze generic problems and requirements

   A library of reusable, generic requirements attached to models of application classes, termed Object System Models (OSMs), is provided. Basically this step takes the use cases elicited in the first step and maps them to the appropriate generic application

classes in OSMs and then suggests high level generic requirements, attached to the classes as design rationale 'tradeoffs'.

3. Generate scenarios

   In this step, scenarios are generated by walking through each possible event sequence in the use case, and each pathway becomes a scenario. Therefore, for a certain use case, one or many scenarios could be generated for it. This step not only generates scenarios for normal behavior, but also for exceptional and error conditions.

4. Validate system requirements using scenarios

   Validation is accomplished with the help of a requirement management tool, called CREWS-SAVRE. More detailed information about this tool is introduced in Chapter 2 of this thesis. This step involves the interaction between the software engineers and the tool.

The rest of this section recites the case study in Sutcliffe [8] to show the role that scenarios play in step 3 and step 4. This case study is based on a security dealing system at a major bank in London. Securities dealing systems buy and sell bonds and gilt-edged stock for clients of the bank. The use case in our example includes the agreement on the price between dealer and buyer. Partial use case and the normal scenarios generated from the use case are shown in Figure 1.5. The difference between each scenario is the timing of event E40 that ends action 40, and whether action 40 or action 45 occurs. The paths for abnormal sequences are illustrated in Table 1.2, from which we can see that the generic requirement is the solution to the exception.

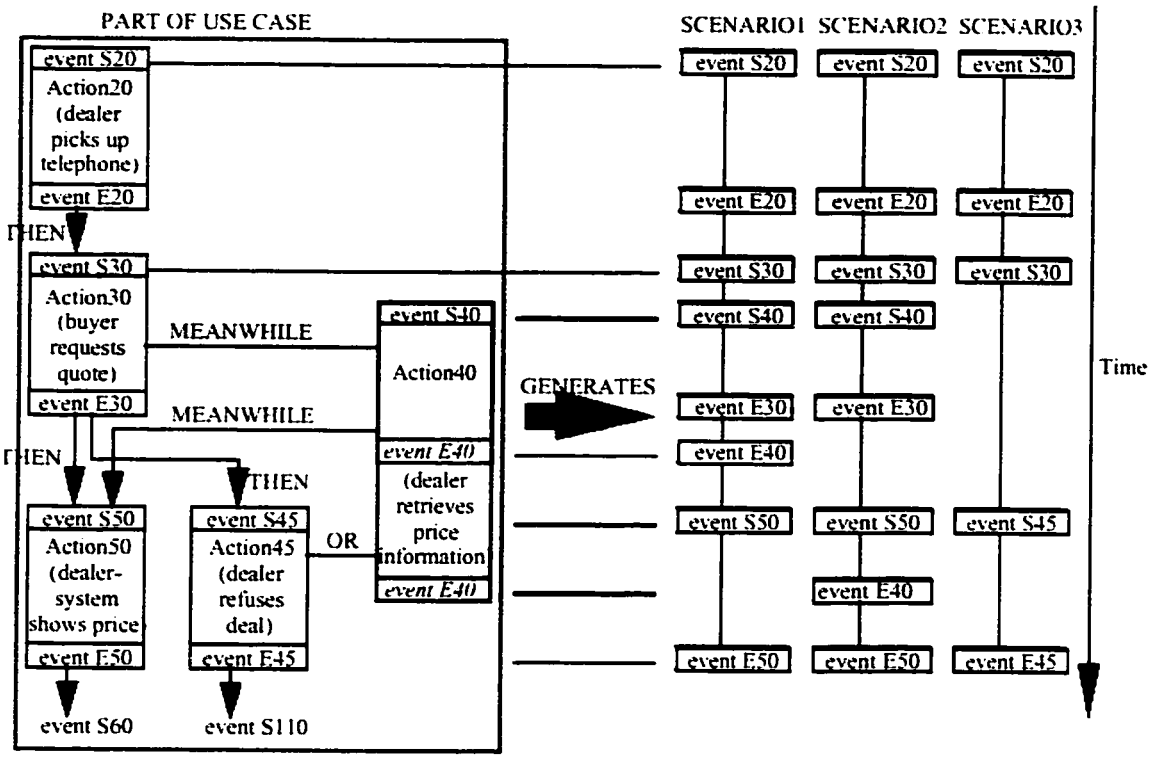

Figure 1.5 Three normal scenarios paths generated from a use case fragment [8]

Table 1.2 Exceptions and generic requirements for abnormal patterns [8]

Exception	Generic Requirement
event does not happen - omitted	time-out, request resend, set default
event happens twice (not iteration)	discard extra event, diagnose duplicate
event happens in wrong order	buffer and process, too early - halt and wait, too late - send reminder, check task
event not expected	validate vs. event set, discard invalid event
information - incorrect type	request resend, prompt correct type
incorrect information values	check vs. type, request resend, prompt with diagnosis
information too late (out of date)	check data integrity, date/time check, use default
information too detailed	apply filters, post process to sort/group
information too general	request detail, add detail from alternative source

CREWS-SAVRE provides two approaches to validate requirements [8]:

1) It presents each scenario to the user alongside the requirement documents, to enable user-led walkthrough and validation of system requirements. For example, the user explores a normal course event, the start of the action: "the dealer enters information into the dealer-system", and chooses the alternative course (exception) relevant to the selected event: "What if the information is incorrect?" If the alternative course is not handled in the requirement specification, one or more candidate generic requirements that are appropriate for the abnormal event are provided to the user. In our example, two generic requirements, 'the system shall check for data entry mistakes' and 'the system shall restrict possible data entry', are provided. The user should pick up one of the two options as a solution to the abnormal situation and add it to the requirement document.

2) According to Sutcliffe [8], the second approach automatically cross-checks a requirements document and a scenario using a collection of patterns which encapsulate 'good' socio-technical system design and requirement specification. The CREWS-SAVRE tool applies one or more validation frames to each event or event pattern in a user-selected scenario to detect the missing or incorrect system requirements [8]. Each validation frame specifies a pattern of actions, events, and system requirements. A validation frame contains two parts. The first part defines the pattern of events, action type, and agent type. The second part defines generic requirements needed to handle the event/action pattern. The frames categorize each requirement as a functional, performance, usability, interface, operational, timing, resource, verification, acceptance testing, documentation, security, portability,

quality, reliability, maintainability, or safety requirement. Hence, automatic requirements-scenario cross-checking is possible using patterns of event, agent and action types in the scenario and requirement types in the requirement document [8].

1.4 Scenarios in Human Computer Interaction (HCI)

Scenarios have been identified as an effective means for analyzing human computer interaction [23]. In addition, scenarios bring significant benefits in consensus among all stakeholders during requirement development for interactive systems [22].

In the HCI field, a widely accepted definition of scenario is:

> *Scenarios are stories – stories about people and their activities. Scenarios highlight goals suggested by the appearance and behavior of the system; what people try to do with the system; what procedures are adopted, not adopted, carried out successfully or erroneously; and what interpretations people make of what happens to them* [3].

In short, a scenario in HCI is rich in details about the context of use. Scenarios can be used in HCI design [3], deriving the UI prototype [24], and helping HCI research to redress the balance between generality and accuracy in theories [25].

1.4.1 Scenarios in UI Prototype

This section introduces an iterative, four-step process approach, which makes use of use cases, scenarios represented as sequence diagrams, and Colored Petri Nets (CPNs) to derive a prototype of the UI from scenarios [24]. This approach not only derives UI prototype from scenarios, but also helps generate a formal specification of the system.

The first step is to construct use cases for the system. For each use case, scenarios are acquired and presented in a form of sequence diagram. The sequence diagram consists of objects, time lines, and messages conveyed among different objects. Messages interchanged are enriched with constrains, which represent the UI information (e.g. in Figure 1.7, *Insert card* is message, *inputData(ATM.Insert_card)* is a constrain). Typical use case diagrams and sequence diagrams of ATM are shown in Figure 1.6, Figure 1.7 (succeed in login) and Figure 1.8 (fail to login).

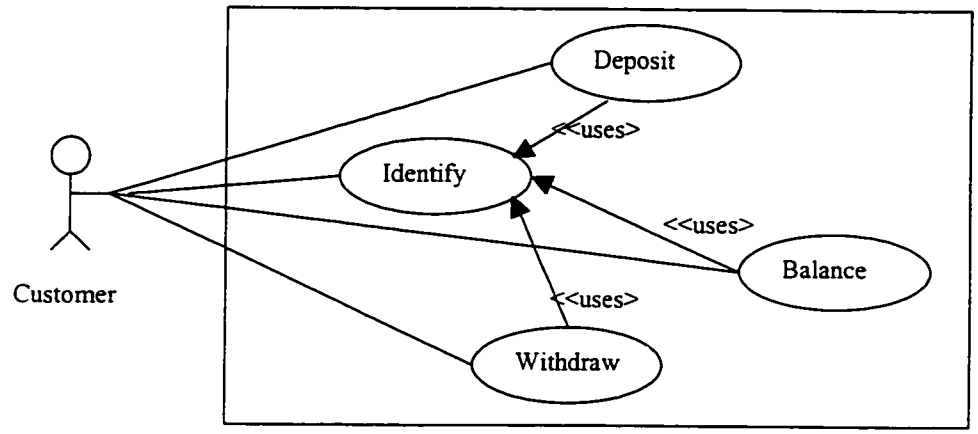

Figure 1.6 Use case diagram of ATM system [24]

Figure 1.6 also shows the convention of use case diagram in UML notation. Use cases are represented as ellipses, and actors are depicted as icons connected with solid lines to the use cases they interact with. One use case can call upon the services of another use case. Such a relation is called a *uses* relation and is represented by a directed solid line. The *extends* relation can be seen as a *uses* relation with an additional condition upon the call [24].

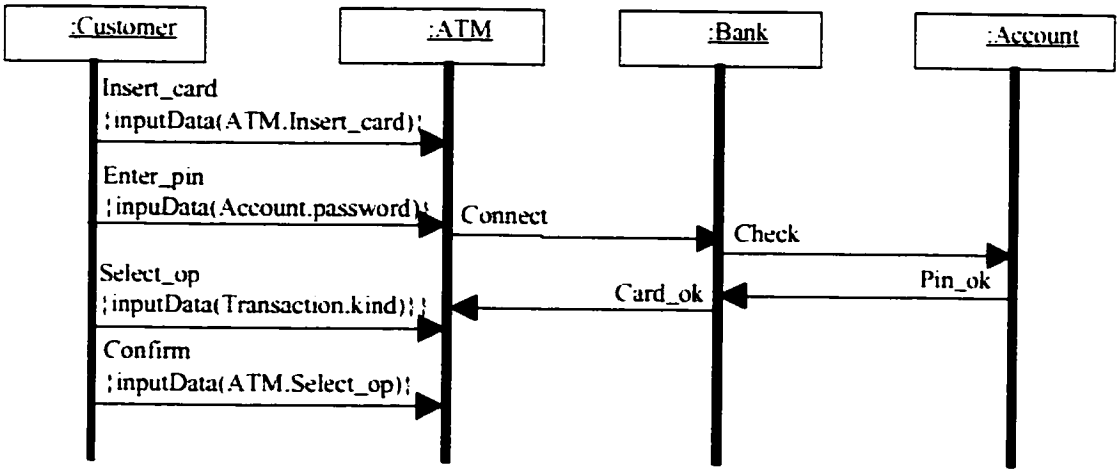

Figure 1.7 Sequence diagram (scenario) for successful login of use case Identify [24]

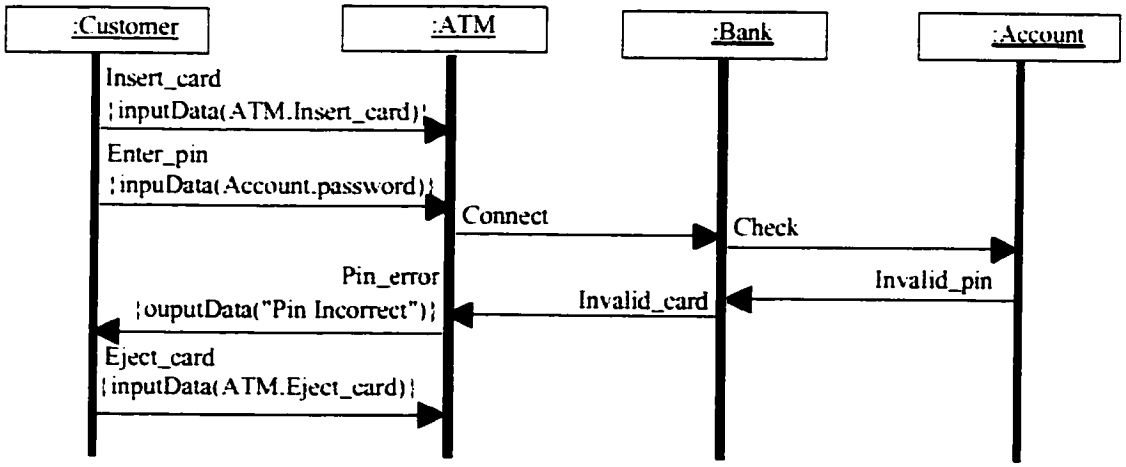

Figure 1.8 Sequence diagram (scenario) for error login of use case Identify [24]

In the sequence diagrams above, the horizontal dimension represents the objects, and the vertical dimension represents time. Horizontal solid arrows represent the messages from the lifeline of the object sender, to the lifeline of the object receiver. UI information is expressed in the curly braces following a message.

Once the UI constrains of the messages are specified in the sequence diagrams, they can be used to determine the corresponding widgets appearing in the UI prototype. According to Mohammed and Rudolf [24], widget generation adheres to a list of rules, which is based on the terminology, heuristics and recommendations found in IBM's guide to user interface design [26] and which includes the following eight items:

- *A button widget is generated for an inputData constraint with a method as dependency, e.g., Insert_card() {inputData(ATM.insert_card)} in Figure 1.7.*
- *An enabled textfield widget is generated in case of an inputData constraint with a dependency to an attribute of type String, Real, or Integer, e.g., Enter_pin() {inputData(Account.password)} in Figure 1.7.*
- *A group of radio buttons widgets are generated in case of an inputData constraint with a dependency to an attribute of type Enumeration having a size less than or equal to 6, e.g., Select_op() {inputData(Transaction.kind)} in Figure 1.7.*
- *An enabled list widget is generated in case of an inputData constraint with a dependent attribute of type Enumeration having a size greater than 6 or with a dependent attribute of type collection.*
- *An enabled table widget is generated in case of an inputData constraint with multiple dependent attributes.*
- *A disabled textfield widget is generated for an outputData constraint with a dependency to an attribute of type String, Real, or Integer.*
- *A label widget is generated for an outputData constraint with no dependent attribute, e.g., Pin_error() {outputData("Pin Incorrect")} in Figure 1.8.*
- *A disabled list widget is generated in case of an outputData constraint with a dependent attribute of type Enumeration having a size greater than 6 or with a dependent attribute of type collection.*
- *A disabled table widget is generated in case of an outputData constraint with multiple dependent attributes.*

The second step is to derive CPNs from both the use case diagram and all the sequence diagrams. The result is two kinds of CPNs: use case CPNs and scenario CPNs.

The next step is to merge all scenario CPNs for a given use case in order to produce an integrated CPN that models the behavior of a use case. This step is repeated for each use case. Then, these CPNs must be integrated with the directly generated CPN from the

given use case diagram to create a global CPN, which captures the behavior of the system. In the ATM case, the CPN for successful login scenario and the CPN for error login are grouped into one CPN for the use case 'Identify'. Repeat the same procedure to 'Withdraw', 'Deposit' and 'Balance' use cases, then combine all CPNs with the use case CPN generated in the second step to represent the functionality of an ATM system.

The last step is to generate a user interface prototype from the global CPN specification constructed in step 3. The prototype generation consists of five operations as detailed by Elkoutbi and Keller [27]: generating graph of transitions, masking non-interactive transitions, identifying UI blocks, composing UI blocks, and generating frames from composed UI blocks. The user interface prototype generated for ATM systems by this approach is shown in Figure 1.9.

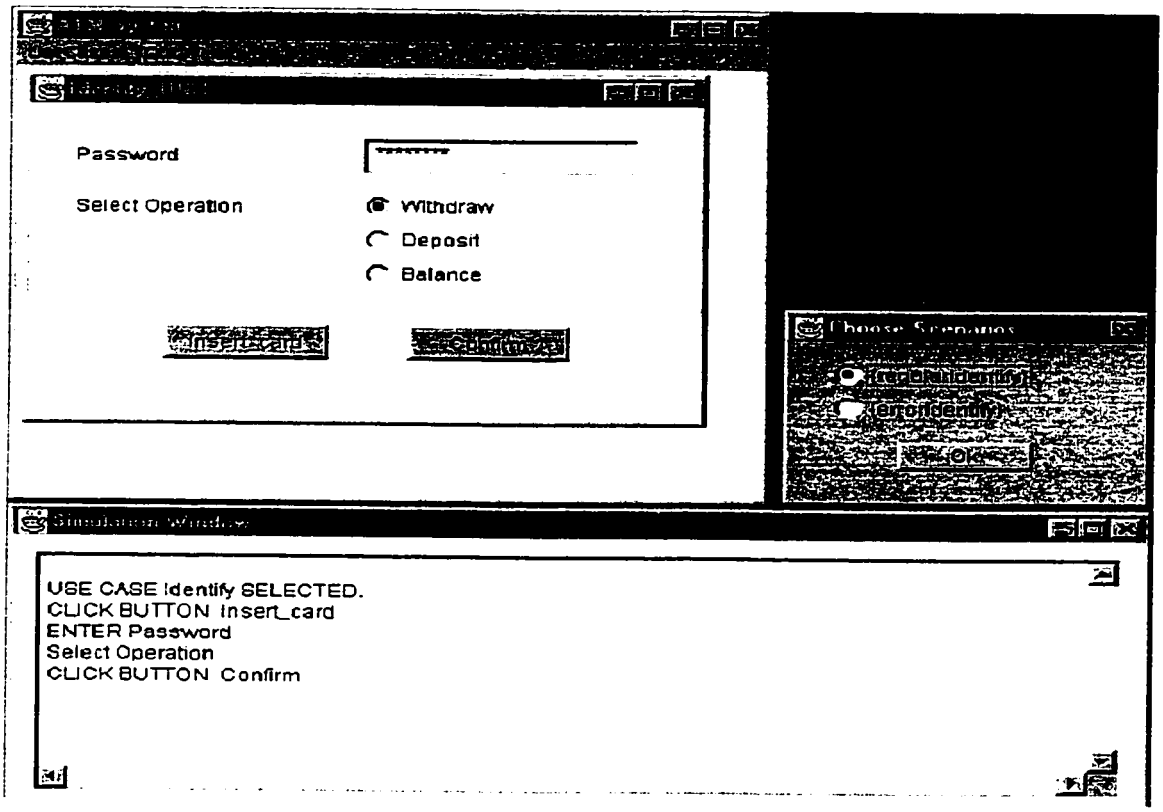

Figure 1.9 User interface prototype execution of ATM system

Choose the *Identify use case* from the UseCases menu. After pressing *Insert_card* button, the *Password* field and *Select Operation* radio buttons are enabled. Moreover, the *Simulation Window* functions as a log window to record actions performed and inform users of the reactions needed. Whenever the execution needs a decision since there may be several options/alternatives that exist, a dialog box will appear for scenario selection (e.g. regularIdentify and errorIdentify scenarios in Figure 1.9).

1.5 Scenarios in Theatrical Arts and Other Fields

1.5.1 Scenarios in Theatrical Arts

Scenarios are used widely in movies. They are used to tell a story and illustrate concretely how to perform the story. Richard Kostelanetz gives the following definition of scenario [28]:

> *If the standard theatrical script has dialogue interspersed with stage instructions and the standard work of music has notes and durational instructions written on staves in horizontal lines, an alternative script, by definition, offers other kinds of text, to induce radically different kinds of performance. ...Instead, there is not just one kind of alternative but several possibilities, such as general instructions for performance activities or a sequence of drawings (with or without words) or a collection of verbal lines to be spoken as the performer wishes, among other hypotheses. It is my polemical purpose to suggest that all these possibilities belong to a single category, which I call* ***Scenarios****.*

What role do scenarios play in theatrical art? In short, scenarios guide and direct the actors to perform. Richard presents us with a good explanation about the role scenarios play:

> *A script is the playwright's road map for the performers, telling them how to proceed. If the playwright provides his or her instructions in a familiar form, the performer are likely to drive straight to his or her destination without a pause. If, however, the map omits some instructions or has unconventional notations, or it marks a path forbidden to cars, or the route is full of one-way streets that proceed in the contrary directions, then the map will induce the travelers to make routes they had not experienced before, perhaps making perceptions they would have otherwise missed. Alternative scenarios serve a similar function in the lives of performing artists.*

To better understand Richard's explanations of a scenario in theatrical arts, a recited scenario example from his book follows [28]. The example is called 'NASOPLOSIVE CHANT for 2 or more chanters', which provides performance instructions to actors.

NASOPLOSIVE CHANT for 2 or more chanters

PERFORMACE INSTRUCTIONS

1. *Begin by setting up a steady, rhythmic pulse of about 2 or 3 beats a second; when all performers feel this pulse securely, go on to part I.*
2. *Begin part I by all saying the indicated sound ("NOOB") in unison, in a monotone, in the rhythm you have set up, one syllable per beat. Keep up a relatively continuous stream of sound as in a chant.*
3. *After a time (you be the judge of how long, or appoint a member of the performing group to decide), begin to leave short silences at random intervals (like commas), still keeping the beat, but no longer in unison. Gradually become more expressive in your pronunciation.*
4. *On a signal from an appointed member of the group, go on to the next section.*
5. *Begin part II in unison, as part I was performed, but make both syllables (NOOBA) fit into the time of the previous one. Follow steps 3 and 4. Change which syllable you accent at will.*
6. *Follow the same procedures for part III, IV, and V, making each syllable the same length as the syllables in the part II (in other words, keep the beat set up while adding a greater variety of syllable combinations).*
7. *Part VI is self explanatory. The piece ends when the performers and/or audience have had enough.*

1.5.2 Scenarios in Other Areas

Scenarios can be used in other areas such as prompting reuse of design patterns in business industries [29, 30], evaluating collaboration tools [31], and conducting business-planning [1].

1.6 Scenario in Our Research Work

The previous sections illustrated the different definitions of scenarios in different contexts. Primarily, our research work focuses on the software engineering area. We adopt Sutcliffe's definition of scenarios. Scenarios are defined as:

> *One sequence of events that is one possible pathway through a use case. Many scenarios may be specified for one use case and each scenario represents an instance or example of events that could happen. Each scenario may describe both normal and abnormal behavior. [8]*

In addition, we also consider scenarios as stories or narrative descriptions of the interactions between the system and the users and of user activities and system responses [18]. In our case, scenarios can be formalized using use cases. Furthermore, scenarios used in our work bear the characteristics mentioned by Carroll: setting (context of the environment), agents or actors, sequences of actions and events, and goals. Every scenario involves at least one agent and at least one goal. The agent performs a sequence of activities to achieve a certain goal in the circumstances of the setting [3].

1.7 Summary

This chapter documented the concept of scenarios, beginning with its origin. Then, it gave a couple of definitions of scenarios. Subsequently, this chapter illustrated different uses of scenarios in different areas such as software engineering, human computer interaction, movie arts, business planning, and business model reuse. For each use of the scenarios, a definition of scenarios for that use was introduced and an example was introduced to show how scenarios are used. Finally, the definition of scenarios used in our research work was given. The next chapter will discover the existing tools dealing with scenarios and illustrate the disadvantages of each tool.

Chapter 2 – Tools for Working with Scenarios

A lot of tools exist for requirement engineering in the industry. Some of them are information management and traceability tools such as DOORS developed by Telelogic Company [40], CaliberRM developed by Starbase Corporation [32] and Rational RequisitePro developed by Rational Software Company [33]. Some are CASE tools such as AxiomDsn developed by Structured Technology Group, Inc. for the modeling of the design of a software system [34] and EasyRM requirement manager developed by Cybernetic Intelligence GmbH for managing documentation such as glossary, requirement, and reference [35]. Among those tools, most of them have nothing to do with scenarios. As H. Zhu et al. concluded: "existing commercial tools for requirements engineering either do not support scenario analysis at all (e.g., DOORS) or only provide facilities for editing use cases (e.g., Rational's ROSE). [37]" In fact, there are only a few software tools that deal with scenarios in requirement engineering. In the following subsections, several tools of this kind are discussed.

2.1 An automated tool for requirements engineering

This tool was introduced by Hong Zhu and Lingzi Jin, at the School of Computing and Mathematical Sciences, Oxford Brookes University, Oxford, UK [37]. They presented this automated tool for scenario-driven requirements analysis in the NDRASS (Requirements Analysis Support System) system. Hong Zhu et al defined the notion of scenarios as a set of situations of common characteristics that might reasonably occur in the use of a system [37]. They also defined the characteristics of a scenario [37]:

User agents: *a scenario must have a specific type of user, or a set of types of users, that participates in the use of the system in the scenario, and/or a subset of the equipment in the environment system that is involved in the operation of the system.*
Use purpose: *a scenario is the set of situations when the user or users use the system with a specific goal or purpose.*
Operation condition: *a scenario must occur under certain operational conditions and certain states of the environment system.*

The NDRASS system has its own requirement definition language NDRDL-2, where ND stands for Nanjing University. This language defines the structure of requirement definition and description of scenarios. A parser for the NDRDL-2 language has been developed to translate the language into machine-recognizable internal form, which can also be processed by the automated tool. In this tool, scenarios are presented using multiple views – data flow diagram (DFD), entity relationship diagram (ERD), and state transition diagram (STD). However, these diagrams are slightly extended from the classic DFD, ERD and STD. This extension lies in that the requirement definition language NDRDL-2 defines the notation of DFD, ERD and STD. This tool contains four interactive and iterative activities to accomplish scenario analysis:

1) It identifies scenarios by analyzing the agents, goals, and conditions. It then describes scenarios in the notation defined by NDRDL-2. A scenario description contains six fields: the name and numbering of the scenario, a short description of scenario, extended DFD, extended ERD, and extended STD [37]:

<Scenario-Description>::=
Scenario [<Scenario-Number>:] <Scenario-name>;
[<Introduction>;] [ER-description>;]
[<DF-description>;] [<CF-description>]

2) It checks the consistency and completeness of a set of scenarios. NDRASS system consists of a set of automatic checkers, which provide automated tool support for checking consistency and completeness of the information and knowledge in the requirement definition [37]. These checkers are syntax checker, scenario/model consistency and completeness checker, model/knowledge consistency checker, and test adequacy checker [37].
3) It supports automatic synthesis of requirement models from a set of scenarios. The NDRASS system consists of a set of automatic synthesisers and generators, which provide automatic tools support for model generation [37].
4) It supports automatic validation of requirement definitions by analyzing the consistency between a set of scenarios and requirement models.

Since scenarios are presented using different views -- DFD, ERD and STD, it increases the inconsistency and ambiguity among scenarios, and the complexity of scenarios. In addition, both scenario and requirement descriptions use NDRDL-2 notation, which has not been generally accepted yet. It requires a lot of practical case studies be carried out in real industry.

2.2 PRIME-CREWS environment

PRIME-CREWS stands for PRocess Integrated Modeling Environment – Co-operative Requirements Engineering With Scenarios. PRIME-CREWS environment supports the use of scenarios in the construction of goal models and in the validation of the model [36]. In this environment, scenarios represent concrete examples of current and future system usage [36]. PRIME-CREWS implements an approach proposed by Peter Haumer

et al [36]. This approach bridges the gap between concrete examples of current system usage (scenarios) and the conceptual current-state models [36]. Current-state model partially defines the functionality and history of the existing system [36]. The requirements for the future system are based on the current-state model. Peter Haumer et al. proposed to capture observations of current system usage using rich media (e.g. video, picture, screen dumps, and speech) and to interrelate those captured observations (called Real World Scene) with the current-state model [36]. From this point of view, scenarios can also be understood as instances of use cases recorded in the form of real-world scenes. More precisely, they [36] proposed to relate the parts of the observations which have caused the definition of a goal or against which a goal was validated with the corresponding goal. Thereby an interrelation between the conceptual goal model and the recorded observations are established. The interrelations between the components of the current-state model and the corresponding parts of the observations provide the benefits for [36]:

- *Explaining and illustrating a conceptual model to, e.g., untrained stakeholders or new team members, and thereby improving a common understanding of the model;*
- *Detecting, analysing and resolving different interpretation of the observations;*
- *Comparing different observations using computed annotations based on the interrelations;*
- *Refining or detailing a conceptual model during later process phases.*

The PRIME-CREWS environment offers tools for multimedia management, for goal modeling and for visualization of various annotations based on the interrelations [36]. The support of requirement validations is realized through tools that enable the user to

view the relationships between goals and scenarios in various ways, and the tool to calculate a number of metrics [37].

PRIME-CREWS has its disadvantage. Innovative projects will benefit less from the tool when there is no precursor system. According to Haumer [36], the PRIME-CREWS environment implements an approach that bridges the gap between concrete examples of current system usage (scenarios) at the instance level and the conceptual current-state models at the type level. However, this approach does not equally apply for any kind of project. It is well suited for projects in which the functionality of the old system could be observed, and in which this functionality has to a large degree, been provided by the new system (even if the system implementation changes significantly due to, for example, technological progress) and/or in which observed shortcomings provide the basis for deriving new goals for the new systems.

2.3 CREWS-SAVRE Tool

The CREWS-SAVRE tool was implemented to support the 4-step method proposed by Sutcliffe et al. [8]. This method is for scenario-based requirement engineering that integrates with use case approaches to object oriented development. The steps of the method are demonstrated in Section 1.3.2 of this paper. This tool provides use case and scenario editing tools, a scenario generation tool, and semi-automatic validation of incomplete and incorrect system requirements. Scenario has been defined as a linear sequence of events on a path in a use case [8]. CREWS-SAVRE supports six main functions, which correspond to the architecture components shown in Figure 2.1 [8].

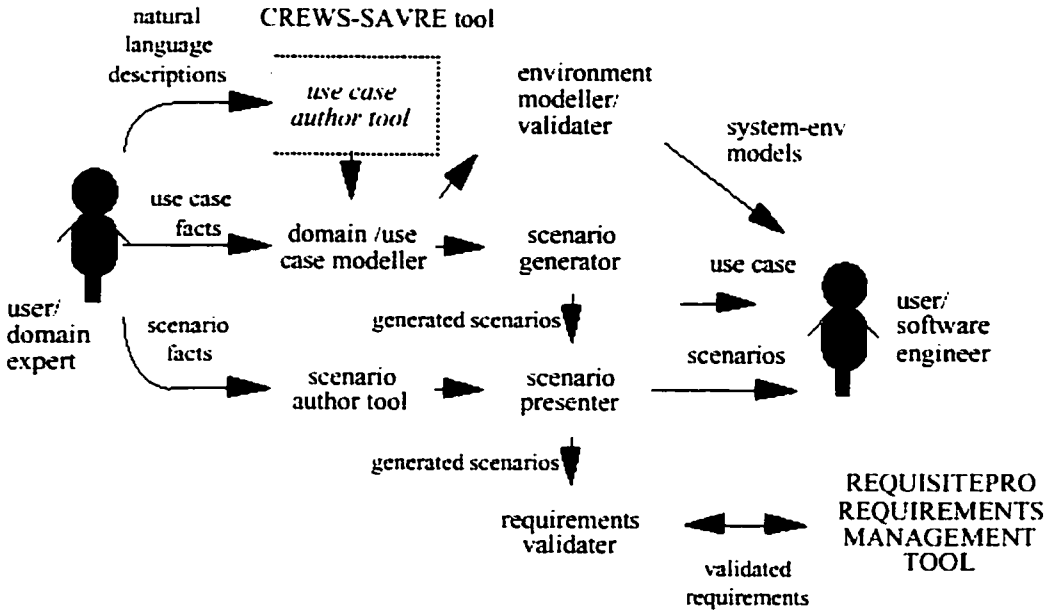

Figure 2.1 Overview of the CREWS-SAVRE tool architecture [8]

1. *Incremental specification of use cases and high-level system requirements (the domain/use case modeller supports method step 1 in Section 1.3.2);*
2. *Automatic generation of scenarios from a use case (scenario generator supports step 3 in Section 1.3.2);*
3. *manual description of use cases and scenarios from historical data of previous system use, as an alternative to tool-based automatic scenario generation (use case/scenario authoring component supports step 1 in Section 1.3.2);*
4. *Presentation of scenarios, supporting user-led walkthrough and validation of system requirements (scenario presenter supports step 4 in Section 1.3.2);*
5. *Semi-automatic validation of incomplete and incorrect system requirements using commonly occurring scenario event patterns (requirements validator supports step 4 in Section 1.3.2);*
6. *Guiding natural language authoring of use case specification.*

CREWS-SAVRE provides two ways to validate the system requirement: first, by applying one or more validation frames to each event or event pattern in a user-selected scenario to determine missing or incorrect system requirements. According to Sutcliffe et

al. [8], validation frames specify a pattern of actions, events and system requirements. Second, by presenting each scenario to the user alongside the requirement documents in order to enable user-led walkthrough and validation of system requirements [8]. In addition, exception events are also generated to help users identify the incompleteness of the requirement model in view of how the system will deal with the exception events [38].

Although this tool facilitates the requirement elicitation and validation, it has some shortcomings:

1) The user uses CREW-SAVRE's domain and use case modeller components to create use case specification.

> *First, for a domain, the software engineer specifies all actions in the domain, defines agents and objects linked to these actions, assigns types {e.g. communicative, physical} to each action, and specifies permissible action sequences. From this initial domain model, the user can choose the subset of domain actions which form the normal course of a use case. [8]*

From the creation of use case specification, we can see that it is software engineers who define agents and actions of a system, but not users. Software engineers preset the models and users can only choose from the preset models instead of creating the models themselves. Users are not fully involved in use case specification. An analogy will make this point clear. If you want to buy a fruit cheesecake from a grocery store and the store only offers two kinds of cheesecake – strawberry and cranberry, you can only choose from these two kinds. If you prefer blueberry or kiwi cheesecake, the store cannot fulfill your need. However, if you choose to make the cheesecake by

yourself, you can have any kind of cheesecake that you want. Some varieties may not be found in any store.

2) In this approach, the generated use case are mapped to the appropriate generic application classes in the OSMs (Object System Models) library, which contains reusable, generic requirements attached to models of application classes. Therefore, when the generic specification is recruited to the requirement specification, the resulting specification document is not application-specific, and may be too general to be efficient for later phases such as design.

3) CREWS-SAVRE provides a way to present each scenario to users together with requirement documents to validate system requirements. Requirement documents are a contract between users and software engineers. Users prefer to read the requirement documents written in natural language. However, designers usually write requirement documents in a precise language (e.g. modeling language), that is easier for them to understand and facilitates the design phase. Therefore, it is more difficult for users to understand the requirement documents. In addition, most users may be domain experts but not professionals in computers, and they may not have enough knowledge to understand the requirement documents written in a precise language.

2.4 Scenario Plus

Scenario Plus was developed by Ian Alexander. It generates and models goals. It also captures, verifies, animates and plays back scenarios [39]. Designed for use by stakeholders who want to describe their requirements to software developers, Scenario Plus consists of a set of add-on tools such as the use case toolkit, diagrams toolkit, and extensions toolkit to enable DOORS (Dynamic Object Oriented Requirements System) [40] to be used for scenario-based requirement elicitation and analysis [39]. Scenario Plus is also capable of generating test scripts.

DOORS is an information management and traceability tool, which was developed by Telelogic. As described in Telelogic's web site [40], requirements are handled within DOORS as discrete objects. Each requirement can be tagged with an unlimited number of attributes allowing for easy selection of subsets of requirements for specialist tasks. DOORS includes an on-line change proposal and review system that lets users submit proposed changes to requirements, including a justification. DOORS offers unlimited links between all objects in a project for full multi-level traceability. Impact and traceability reports as well as reports identifying missing links are available across all levels or phases of a project life cycle. Verification matrices can be produced directly or output in any of the supported formats including RTF for MS-Word, Interleaf and FrameMaker [40].

2.5 Summary

This chapter introduced several tools for requirement engineering, which deal with scenarios. Among them, some are research tools, and some are commercial tools. In addition, the disadvantages and shortcomings of each tool were illustrated in this chapter. In the next chapter, a requirement engineering tool will be proposed to cover most of the disadvantages mentioned here.

Chapter 3 – SUCRE: An XML-Based System for Documenting and Using Scenarios

3.1 Why XML?

XML (Extensible Markup Language) is a metalanguage used to define other languages [41]. Any form of data in a database can be described and manipulated by a language. Therefore, XML can be used to define the form of data [42]. XML is a markup language that specifies neither the tag set nor the grammar for that language [41]. It is a document type definition (DTD) that defines the grammar and tag set for a specific XML formatting. Simply speaking, DTD defines the way an XML document should be constructed [41].

XML, a language geared toward markup, with only the slightest restrictions, allows us to create as many tags as we like, and insert them wherever we want in our documents. Tags can be embedded in other tags, and parsers can be used to specifically pull out the information that immediately follows named tags. Therefore, it increases the flexibility of data formatting [43].

With XML, transfer of XML-based scenario can become standardized easily. Tags can be combined into Document Type Definitions (DTDs) to form a template of sorts, which can be used to standardize scenarios of different types. Using standard DTDs will not only make the scenarios consistent, but scenarios can be easily categorized and retrieved [43].

3.2 Overview of SUCRE system

SUCRE (Scenario and Use Cases for Requirements Engineering) is an XML-based system for scenario-driven requirement engineering. The SUCRE system aims to provide requirement engineers with a semi-automated tool to elicit, validate and share user requirements captured as scenarios. The SUCRE system has five components called Scenarios, User Archetypes, Usability Goals, Prototypes, and Use Cases. In its current state, the scenario component of the SUCRE system is being constructed. The scenario component offers different features such as scenario editing and scenario analysis.

As mentioned in Chapter 1, scenarios can have different meanings and can be documented using different formats such as tabular or diagrammatic notations, user interface storyboards or textual. In our SUCRE system, we consider a scenario as stories that can be formalized using use cases.

3.2.1 SUCRE Objectives

One of the most difficult problems in user requirement engineering is the communication gap that exists between different end-users, stakeholders, and software engineers. Software engineers tend to speak using technical terms while the stakeholders and end-users use natural language. As a result, the software engineers may not have a clear understanding of what is specifically needed in the system they are building. The customers and end-users would often be surprised when they notice that the final product does not fulfil their expectations [18].

There are several solutions to this problem. A well-known solution is to use prototypes. This presents something concrete that the stakeholders can react to [44]. Since scenarios allow different stakeholders to describe and review the problem in their own language instead of some abstract model, they are also a solution to the problem [45]. In Chapter 2, we have already investigated several tools working with both use cases and scenarios and illustrated their shortcomings. In our proposed SUCRE system, we combine prototype, scenarios and use cases in a single and comprehensive framework to avoid most of the shortcomings in other tools.

3.2.2 SUCRE Architecture

To bridge the gap between the natural language used by stakeholders and formal modeling language used by developers, we propose a progressive, iterative and interleaved process model of requirement engineering (as shown in Figure 3.1).

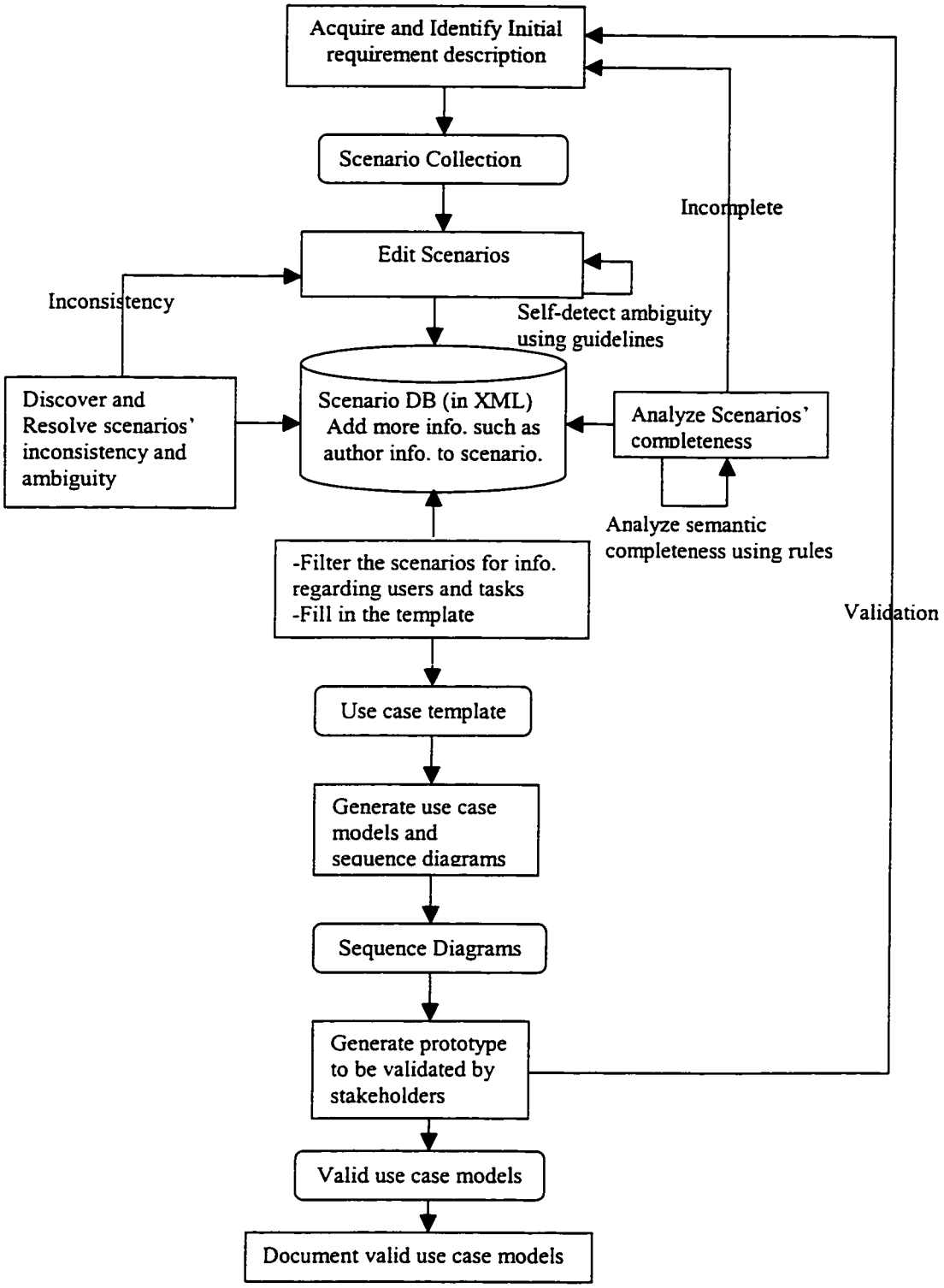

Figure 3.1 Architecture of scenario component in SUCRE system

Figure 3.1 shows that SUCRE is a tool support for eliciting requirements, where scenarios play an important role. This tool involves different kinds of actors, and each actor performs different tasks. The following sections describe Figure 3.1 in more detail.

3.2.3 Process Actors and Their Responsibilities

Our system users include end-users, stakeholders, usability experts, and requirement engineers. Among them, end-users, stakeholders, and usability experts are all responsible for eliciting scenarios. Stakeholders and end-users also validate the scenarios. Both usability experts and requirement engineers operate on scenarios and generate the use-case models, and finally, document use case models.

End Users

In theory, end-users are people who know what to build and what the system does. However, in the real world, users do not have clear and complete ideas about the desired system. Therefore, they need to work together with professionals who can help them in requirement elicitation. In our proposed system, end-users can work with usability experts to elicit requirements.

Usability Expert

In our system, usability experts are responsible for eliciting scenarios directly from end-users and stakeholders. In order to retrieve all necessary requirements in a comprehensive way, usability experts use several techniques such as interviews, questionnaires, and rapid prototypes to get feedback. After collecting a large set of scenarios, usability

experts will be responsible for cleaning up the scenarios, describing them in an XML form, and inputting clear and consistent scenarios into the scenario database.

Stakeholders

Stakeholders are people who may not use the system directly. However, their opinions will affect what the system does or how the system looks. For example, end-users, managers, engineers involved in maintenance, domain experts, and trade unions are all stakeholders.

Requirements Engineer

First, requirement engineers together with usability experts clean up scenarios to resolve the inconsistency and incompleteness. Then they extract information regarding users and tasks from scenarios and fill in the use case templates. Subsequently, requirement engineers generate use case models based on the templates and validate the generated use case models. Finally, requirement engineers document the use case models.

3.2.4 System Features

The proposed process model (as shown in Figure 3.1) involves different tasks. This section is going to explain each of them in detail.

Acquire and Identify Initial Requirements

This feature is to acquire the essential requirements from end-users and stakeholders, and present requirements in natural language descriptions of scenarios. Then it identifies a set

of scenarios by analyzing the settings, agents, and goals. Several approaches to identify scenarios are proposed in Jacobson et al. [16] and Maiden et al. [46]. The requirement contents should include purpose, scope and general constraints of the desired software, a brief description of the overall functionality of the system, functional requirements, non-functional requirements, and domain knowledge. Since end-users and stakeholders do not usually have software development knowledge, they may ignore some information, which is important for later requirement development. Furthermore, the information provided may not always be well organized. Therefore, in SUCRE, quite often the end-users and stakeholders will rely on the help of usability experts. Usability experts make use of methods such as interviews and questionnaires to retrieve the acquisition, and generate scenarios according to guidelines. These scenarios will then be useful, and easier for later analysis. According to Rolland, guidelines are of two types: style guidelines provide recommendation on the style of writing narrative prose; content guidelines advise the author on the expected contents of his/her prose [47]. Both guidelines have the form of plain text, which can be prompted to the usability expert when writing down scenarios. Some examples of guidelines are shown below [47].

***Style guidelines*:**
1- You should avoid the use of anaphoric references such as « he », « she », « it » « his »or « him ». Instead, you should use nouns defined in the use case glossary.
2- You should avoid the use of synonyms and homonyms. The same object or agent should be named identically throughout all texts.
3- You should avoid sentences with more than two clauses, each clause being composed of a subject, a verb and its complements.
4- You should mention explicitly your assumption when some action is done under certain conditions. For example you can use ''If <condition> then Action''.
5- You should mention explicitly the condition for stopping repeated actions. For example you can use ''Repeat <action> until <condition>''.

6- You should mention explicitly the co-occurrence of several actions. For example, you can use "While <action>, <co-occurring action>".
7- When you express an action, you should mention explicitly the agent which undertakes the action, and the object of the action. Thus prefer the active to the passive voice.
8- When you express an action of communication, you should mention in addition the source and the destination of the communicated object.
9- You should write sentences at the present tense, avoid the use of the negation, of adverbs and of modal verbs.

Contents guidelines:
The expected scenario prose is a description of a single course of actions. Alternative scenarios, interruptions or exceptional treatments are described separately. A course of actions typically describes sequentially ordered actions: actions from an agent to the system, system responses to the agent, communications between agents and possibly actions internal to the system. You should put your sentences in the order of the scenario history. The course of actions should be completed with the resulting end states. You should describe the course of actions you expect, not the actions which are not expected.

Edit Scenarios

After acquiring and identifying the requirements, we get a collection of scenarios. Each scenario will be described in an XML notation and stored in a scenario database. In addition to original scenarios, additional information such as the scenario's id, author, creation date and last modification date will also be saved in the scenario database. Before being saved in the database, the scenarios must be checked using the style and content guidelines (as mentioned above) to detect and resolve any ambiguity. In addition, each clause of scenarios needs to be described in Rolland's text structure [47], which are useful for later scenario analysis and use case model generation. However, this editing only needs to be done after scenarios are cleaned up. When scenarios are cleaned, nouns, verbs, adjectives, adverbs, prepositions, pronouns, and articles in scenarios are tagged. Adjectives, adverbs, and articles can be ignored and removed from the clauses. Nouns should have already replaced pronouns in the requirement acquisition and identification

step. Sequentially, subject, main verb, object, and complement required for structure constructions are extracted from the nouns, verbs and articles. Rolland's text structure will look different depending on the different ways of expression. For example, for a simple clause with subject, verb, and object, there are 11 structures to express it. Two of them look like: *S1: [[NG](Subject)*$_{Agent}$ *[Verb](Main Verb)*$_{Action}$ *[NG](Complement)* $_{Object}$ *](VG active)*$_{Action}$ for active action. *[[NG](Subject)*$_{Object}$ *[be Verb](Main Verb)*$_{Action}$*](VG passive)*$_{Action}$ for passive action.

The scenario database plays a very important role in our XML-based system. It functions as an information center and data repository to provide data in order to fulfill different requests. It stores all scenarios collected from users at the very beginning, refined scenarios, and categorized scenarios.

Clean Up Scenarios

The next step of our process model is to clean up the edited scenarios. First, it discovers inconsistencies and then resolves the inconsistencies. Second, it analyzes the completeness of scenarios. In our system, usability experts are responsible for the cleanup work. When this step is finished, we will get a collection of tagged nouns, verbs, adjectives, adverbs, prepositions, pronouns, and articles. In the following two subsections, the cleanup activities are explained in detail.

1) Discover Inconsistency. It is widely recognized that for a complicated software system, there may be a great number of scenarios. A scenario, in our case, is a particular

path through use cases. Several scenarios may annotate one use case and there may be overlaps among them. Thus it increases the probability of inconsistency among scenarios. Also, different stakeholders may provide conflicting requirements for the same functionality. In addition, end users and stakeholders express requirements in their own terms, according to their understandings, which can cause ambiguity in the terminology. Therefore, we need to discover any inconsistency and ambiguity.

A few approaches exist which we can adopt to discover and resolve any inconsistency. An approach, proposed by Keller et al., merges all XML-formatted scenarios annotating the same use case together to resolve the overlap among scenarios [24]. Rolland et al. proposed some rules to resolve possible linguistic ambiguities in the expression of scenarios [47]. In addition, our tool ScenarioTool (Chapter 4) is currently being developed in response to the problem of terminology ambiguity.

In our ScenarioTool, terminology ambiguity cleanup is a process of "find and replace". Usability experts parse the edited scenarios to find the hyponym, hypernym, and synonym, and then replace them with consistent, alternative ones from the glossary to reduce linguistic inconsistency. It sounds simple, but in practice this process involves a lot of natural language processing sub-processes and applies natural language processing tools. Table 3.1 gives a brief overview about the sub-processes and tools used. Section 3.3 will elaborate on this discussion. This process may need usability experts' interaction to clarify ambiguity. Consider a simple example,

Scenario1: The XYZ system contains a subsystem – the ABC system. The system is able to ...

Ambiguity: "The system" could be either, the XYZ system or the ABC system.

Solution: In this case, we need human interaction to help solve this ambiguity. We will prompt participant to clarify "The system" either to be XYZ or ABC.

This step plays an important role in the whole system because all other features are based on the cleaned scenarios. If the ambiguity and inconsistency are not resolved at this stage, the use case models and requirement specifications based on the edited scenarios will be confused, and not at all useful in the later stages. This step can be an iterative process. If the ambiguity is found, it will backtrack to describe scenarios and to resolve the conflicts.

Table 3.1 Brief overview about sub-processes and tools used

Sub-processes Description	Natural Language Processing Tool
Parse a given string to separate all punctuation from words	PreTokeniser
Parse a given string to reproduce it one sentence per line in a text file.	SentenceSeparator
Parse a given tagged string argument and returns a String array of all the nouns it finds. The tagged string was created using Qtag3.0[1] and that the tagging notation is that of Qtag3.0.	QtagNounFinder
Parse a given tagged string argument and returns a String array of all the verbs it finds. The tagged string was created using Qtag3.0 and that the tagging notation is that of Qtag3.0.	QtagVerbFinder
Given a word, WordNet[2] executable with the given word as argument, parse the WordNet output, and returns a String array of all synonyms of the word found by WordNet.	SynonymFinder
Given a word, WordNet executable with the given word as argument, parse the WordNet output, and returns a String array of all hyponyms[3] of the word found by WordNet.	HyponymFinder
Given a word, WordNet executable with the given word as argument, parse the WordNet output, and returns a String array of all hypernyms[4] of the word found by WordNet.	HypernymFinder
Given a word, WordNet executable with the given word as argument, parse the WordNet output, and returns the String definition of the given word found by WordNet.	DefinitionFinder

[1] Qtag 3.0 is a Part-Of-Speech tagger. It reads a text file, analyses it, and creates an output file identical to the input file except that all nouns, verbs, adjectives, etc. have been tagged.

[2] WordNet is a lexical database for the English language that was created by Princeton University. It contains synonyms, hypernyms, hyponyms, definitions, and more for many English words.

[3] One word is a hyponym of another if it has a more specific sense. For example, "digital computer is a hyponym of "computer".

[4] One word is a hypernym of another if it has a more general sense. For example, "machine " is a hypernym of "computer".

2) Analyze Scenario Completeness. It is likely that we did not collect sufficient scenarios for the potential software in the first run. Therefore, it will backtrack to the scenario acquisition and identification stage to cover the missing requirements. In addition, there may be incomplete semantics in scenario clauses and sentences. Rolland et al. [47] proposed applying rules to discover and resolve incomplete semantics. For example, completion rules state that if the instantiated linguistic pattern of a clause has a missing element, then the author is asked to provide the missing element [47]. Figure 3.2 gives an example of the completion rule, CO1. It denotes that for every action V there is an object O such that if an agent performs this action V on the object, but the agent element is missing, then the author will be asked to provide the missing element to make the clause complete. CO1 applies when an action does not have an agent [47]. ASK predicate indicates the need of action required from the author. All rules have a premise and a body separated by a <<→ >>. The premise defines the precondition for executing the body [47].

CO1 : ∀ V , ∃ O: Action (V) [Agent:? ; Object:O] →
ASK(« Complete : V by... (agent of the action) »)

Figure 3.2 An example of completion rule [47]

Filter Scenarios and Fill in Template

After cleaning up the scenarios, the next step is to analyze the edited scenarios to figure out the actors (agents) and tasks (actions) of the potential software, the relationship between them, and constraints. In other words, it is essential to catch the semantics of text, in order to fill the gap between the informal representation (scenarios) and the formal model of use cases. This step is based on the approach presented by Rolland [47],

who proposed using semantic patterns to capture the semantics of text within scenarios. This process involves linguistic analysis and semantic representations of scenarios. There exist several representations of semantics such as, Sowa' s conceptual graphs [49], or Montague' s grammar [48] and case grammar [47]. Since case grammar focuses on the notion of action that is central in the use case model, it becomes the best choice for our purpose. Case grammar represents semantics using a set of semantic cases such as agent, object, destination and source [47]. According to Rolland et al. [47], semantic cases define the semantic roles that the different elements of an action clause play with respect to the main verb. Patterns of semantic cases are called semantic patterns. Indeed, semantic patterns represent the meaning of a chunk of text [47]. For example, a general action semantic pattern looks like: *Action (V) [Agent; Object]*. The semantic pattern for "the customer sends the order" is *Action (send) [Agent: 'the customer'; Object: 'the order']*. Usually, the meaning of a text can be represented by different expressions, for example, "the customer sends the order", "the order is sent by the customer", "the sending of the order by the customer" and many more. These different expressions are called surface structure. Therefore, each semantic pattern can be expressed by several surface structures. During the course of filtering, semantic patterns function as templates to associate a semantic case (agents, object, destination, source) to different elements of the clauses and sentences. This addresses the semantics of the clauses and of the sentences. Semantic patterns define different semantics of the elements required in use case models such as actions, agents, objects, initial and final states of objects, the relationship between two objects, constraints and conditions, and the relationship between actions (sequence, concurrency and repetition). In turn, the semantics of use

cases' elements are expressed by semantic patterns. Take "the order is sent by the customer" as an example. As a communication action, this scenario clause is not complete in semantics. By applying the completion rule and by asking the requirement engineer, the new scenario clause will be "the order is sent by the customer to company X". The text structure describes the clause as follows:

[['order'](Subject)$_{Object}$['is sent'](Main Verb)$_{Communication}$['by customer'](Complement)$_{Agent-Source}$['to company X'](Complement)$_{Destination}$](VG passive)$_{Communication}$

The semantic pattern instance, generated from the above text structure by applying analysis rule, is:

Communication ('send') [Agent : 'customer' ; Object : 'order' ; Source : 'customer' ; Destination : 'company X']

This filter process involves the application of many rules. Rolland discussed these in detail [47]:

Analysis rules: identify action names, agents and objects.

Clarification rules: change the wording and remove possible ambiguities.

Completion rules: complete a semantic pattern not fully instantiated to accomplish the semantic completion of a clause.

Emergence rules: help generate abnormal scenarios. They are based on flow conditions and raise the question of what happens if the flow conditions do not hold.

For each scenario in the database, we can fill in the templates that detail the actors' profiles and their tasks. For our example, this template looks like Table 3.2.

Table 3.2 The templates that detail the actors' profiles and their tasks

Actors	Tasks	
Customer	Task Name	Send Order
	Task Input/Starting state	Order form is ready to be sent
	Task Output/Finishing state	Order has been sent
	Task dependencies Dependencies between several actions.	[B] Fill in order form Note: [B] stands for "before", indicates action happens before this action. [A] stands for "after" indicates actions happening after this action. [C] concurrency. [I] stands for iteration

Generate Use Case Models

After filtering scenarios, information about scenarios is recorded using the templates. We use semantic patterns to present scenarios and each semantic pattern defines different semantics of the elements required in use case models such as actions, agents, objects, initial and final states of objects, the relationship between two objects, constraints and conditions, and the relationship between actions (sequence, concurrency and repetition). Therefore, the semantics of the elements that are required in use case models are recorded in templates. The recorded information will be used to generate use case models.

However, we face a problem – how can we categorize several tasks into one use case? For example, actions – the user inserts his card in the ATM, the ATM checks if the card is valid. If the card is valid, then it does something. All these actions can be grouped into one use case – Identify. For the time being, the solution does need the interaction of a software engineer. First, we should group together all tasks corresponding to the same

actor. Then, the requirement engineers can decide which tasks should be in one use case. Once all this has been done, it is possible to generate the use case models.

According to Rolland, an important property of semantic patterns is univocity [47]. Compared to natural language, which provides many alternative ways to express the same knowledge, semantic patterns give a unique deep level representation to support the meaning of a chunk of text [47]. Therefore, when we use semantic patterns to present scenarios, this property helps reduce the possibility of inconsistency and resolves ambiguity.

Validate and Document Use Case Models

After use case models are generated, stakeholders will be responsible for validating these models and providing feedback. The feedback helps usability experts and requirement engineers to address the inconsistency and incompleteness of scenarios as well as to refine the use case models. As mentioned in Section 3.2.1, prototyping is a solution to bridge the gap between stakeholders and software engineers. Therefore, in order to make validation easier for stakeholders, we generate rapid prototypes and present them to the end-users or stakeholders for validation and evaluation. Mohammed and Kell proposed an approach of deriving sequence diagrams from scenarios and then generating prototypes from the sequence diagrams [24]. Since we have scenarios associated with the use cases in the generation of use case models, we can follow Kell's approach to create the sequence diagrams from scenarios, then generate prototypes from sequence diagrams for later validation. According to the advice and suggestions of stakeholders, existing

scenarios may need to be modified, and new scenarios can be added. In either case, a new iteration of the process model is about to start. The iteration goes on and on until both requirement engineers and users (end users and stakeholders) come to an agreement.

Once the use case models are generated and validated, it will be possible to document the valid use case models. As steps are completed, our scenarios in the database are refined progressively with respect to completeness and clarity. These scenarios are useful for documenting use case models. In addition, since the template records the requirement information, it is also a source that can be used to document use case models.

3.2.5 Advantages of SUCRE System

The following are the main advantages of our SUCRE system:

1) Our SUCRE system involves greater user participation and provides more accurate requirements. Instead of retrieving use case directly from users, we gather scenarios directly from users. In the latter case, it is the users rather than the developers who identify the agents and goals. During the course of acquisition, usability experts, who are not only domain experts but also play the role of requirement engineers, help users find their needs and guide them towards the appropriate direction. This is helpful for the later design phase. They can then organize the initial acquisition to a form that is easily understood by designers and software engineers. Moreover, usability experts are able to refine the scenarios according to the guidelines. Compared to the general requirements in Sutcliff [8], usability experts will help users to generate requirements specific to their potential software system. In this manner,

the acquisition achieved will be more accurate and fit the users better. An analogy can make this clear. Assuming a man wants to buy a suit, he can go to a suit store and pick out a suit that is supposedly his size from styles available in-store. He can then ask for minor modifications. Alternately, he can present his preferred style, one that came with the help of a designer, and ask the tailor to make it specifically for him. Comparatively, the latter case will provide the man with a better fitting suit.

2) Usability experts are people who have knowledge and experience about how to motivate users to talk about their needs, how to organize events such as interviews, and how to gather information from users. They also have techniques for grouping people together. Usability experts can be from different disciplines such as psychology, or from different computer research fields such as human center interaction (HCI). Therefore, the participation of usability experts makes requirement acquisition more efficient and effective. If users provide requirements all alone, it is very possible that they will miss and ignore information that could be requirements. In addition, users may not discover all of their needs on their own. It is very likely that the acquisition process will be repeated a number of times. This wastes both users' and software engineers' time. However, the expertise of usability experts can motivate the users to find their needs to a large extent, and help reduce the chance of repetition of requirement acquisition.

3) The end-result that the SUCRE system provides is specific to a certain system. It is more efficient than a general one for the design phase. Since the final requirement

specification is accurate and fits users better, it decreases the possibility of tracing back to requirement acquisition from the design stage or even the implementation stage. In addition, during the whole process of the SUCRE system, we keep checking inconsistency and incompleteness in scenarios, and between scenarios and models, to make sure that we will eventually achieve clear, correct, and complete use cases for designers to use.

4) Since we get the user fully involved, users are aware of the functionality of the potential system. Moreover, since users provide scenarios for steps necessary to perform a certain job, it is easier and faster for users to become familiar with the new system. In other words, the command structures are intuitive and cognitive to users. Therefore, training for the potential software in the future is easier.

5) In the validation stage, we use prototypes instead of asking users to read the specification documents. It is easier for users to understand the potential system and to provide us precise feedback to refine our scenarios and use cases.

6) The 'univocity' property of semantic patterns reduces the possibility of inconsistency and resolves ambiguity while semantic patterns are used to present scenarios.

3.3 Summary

This chapter demonstrated the objective, architecture, actors, and features of the proposed SUCRE system, which is an XML-based system for scenario-driven requirement

engineering. In addition, this chapter introduced what XML is and illustrated the reasons why XML has been chosen to present scenarios in the SUCRE system. In the next chapter, the prototype and main functionality of the SUCRE system will be presented. In addition, it will provide examples of DTDs and XML files, which are used to present scenarios in SUCRE's scenario database.

Chapter 4 – SUCRE Prototypes

Carla De Waele, an undergraduate student at Concordia University, developed part of the SUCRE prototype, which is called ScenarioTool. This tool implements the cleanup functionality – resolves the terminology inconsistency. In addition, another prototype is being developed to access the XML-based database of scenarios. This chapter summarizes these works.

4.1 ScenarioTool Architecture

4.1.1 Overview

Figure 4.1 illustrates the different packages that comprise ScenarioTool. A package usually consists of more than one class. The packages are organized in different layers. The first layer is the user interface. Each layer below the first layer, holds classes that represent a further abstraction from the user interface and become progressively more technical.

The Model-View-Controller (MVC) design pattern is adopted in the ScenarioTool architecture. Package GUI is the view, which manages the way the ScenarioTool is displayed. The packages in domain layer and technical services layer function as a model, which manages whatever data or values the tool uses. The package scenarioTool is the controller, which determines what happens when the user interacts with the ScenarioTool. It also delegates the look-and-feel-specific aspects to the model. The lines

connecting different packages represent the relationships that exist between the classes of the different packages. For example, some classes in the GUI package communicate directly with the classes in the scenarioTool package and vice versa. However, classes in the GUI package do not communicate directly with classes in either the Tools or SwingWorkers packages and vice versa.

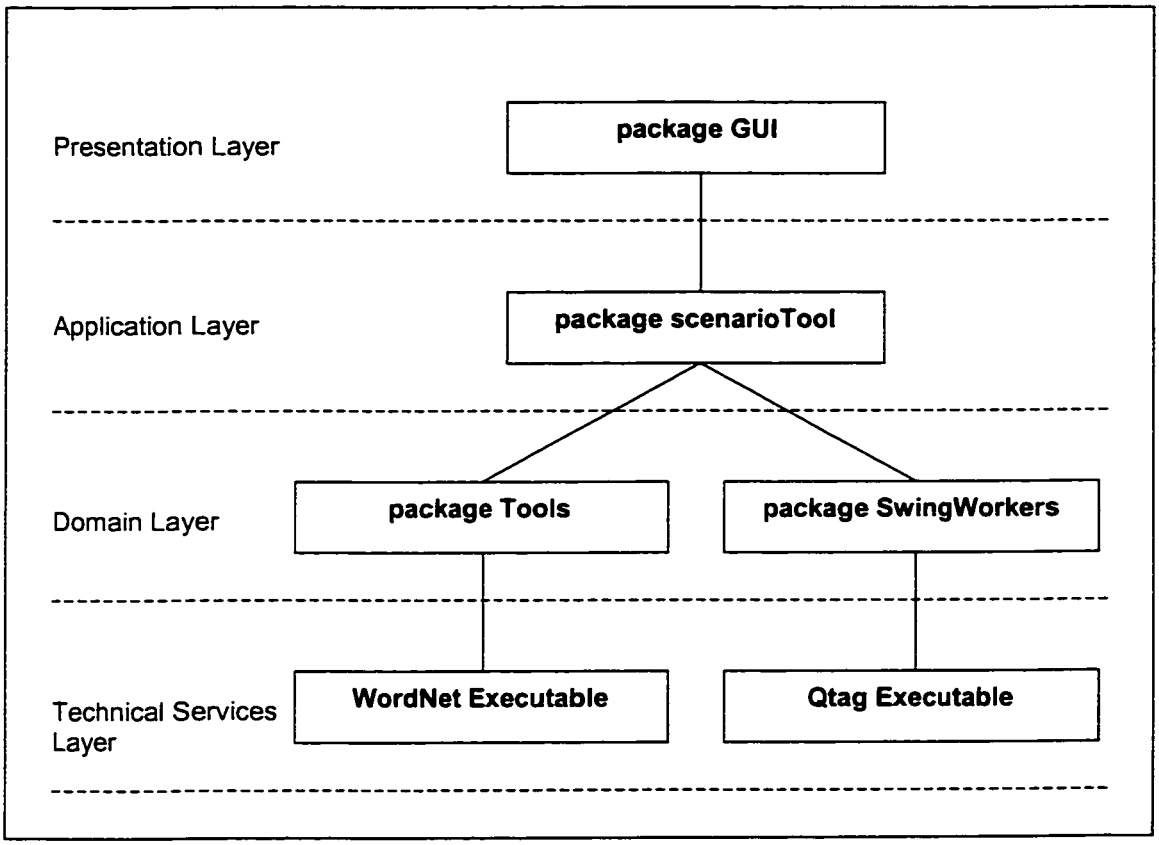

Figure 4.1 Overview of ScenarioTool's architecture

The following is a short description of each package:

1. **package GUI:** Its purpose is to display the panels, windows, and various components of the User Interface on the screen.

2. **package scenarioTool:** This is the main component of the software. It captures the interaction and acts as a controller for the whole program, providing indirection between the presentation layer and the lower layers.

3. **package Tools:** This is concerned with any natural language processing including parsing of text, as well as a connection to external natural language processing programs.

4. **package SwingWorkers:** This is concerned with any time-consuming natural language processing task. Its purpose is to fork separate threads for time-consuming tasks, so that the user's interaction with the GUI is not halted.

5. **Qtag Executable:** The directory called "qtag" contains the Qtag 3.0 executable jar file, which is a Part-Of-Speech (POS) tagger. It parses a given text file and outputs a tagged text file where each noun, verb, adjective, adverb, preposition, pronoun, and article is tagged.

6. **WordNet executable:** WordNet is an executable external program that must be downloaded separately. It is not, in any way, "contained" within the ScenarioTool system. You can think of it as a separate system that ScenarioTool talks to. However, if WordNet is not on the local computer, ScenarioTool won't work. When WordNet is downloaded, it automatically defines a path to itself in the computer's "path.exe" file, which allows ScenarioTool to invoke it from within any directory.

4.1.2 Detailed Architecture

Figure 4.2 illustrates the types of relationships that exist between the classes within a package and between the different packages.

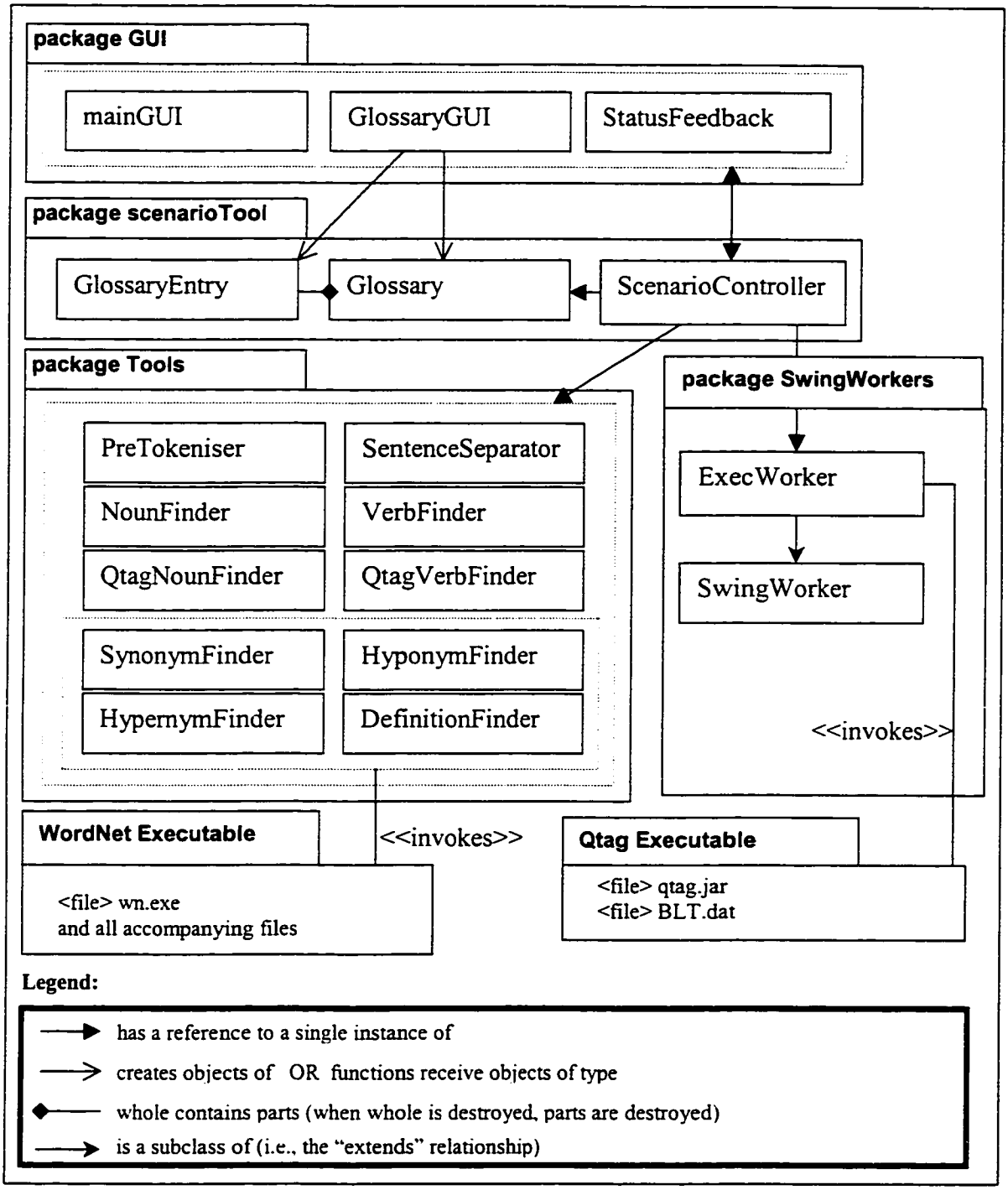

Figure 4.2 ScenarioTool's architecture

The following is a short description of each class that is depicted in Figure 4.2 and of the executables that ScenarioTool invokes:

- **package GUI**

  mainGUI:

  The main GUI displays the main frame and all of its components. It is created by ScenarioController at the start of the program and remains as a single instance throughout one ScenarioTool session. When created, it is given a reference to the single instance of ScenarioController. When it is closed, it uses its reference to ScenarioController to call the exit() function, which terminates the ScenarioTool session, and the program exits.

  GlossaryGUI:

  This is a secondary GUI window. It displays the frame that represents ScenarioTool's glossary. It is created by ScenarioController and exists as a single instance for as long as its "close" button is not pressed. When it is created, it is given a reference to the single instance of ScenarioController. GlossaryGUI may create some temporary objects of type Glossary or GlossaryEntry within the scope of its member functions. However, it does not hold any reference to persistent instances of either class.

StatusFeedback:

This is a secondary GUI window. ScenarioController creates it whenever a time-consuming task is in progress. It informs the user that she/he must wait for the system to perform its task.

- **package scenarioTool**

GlossaryEntry:

Each instance of this class represents an entry in a glossary. It only has three attributes. Namely, a String "term", a String "definition", and an ArrayList "synonymList". Only an instance of the Glossary class can create a *persistent* GlossaryEntry object.

Glossary:

Only one instance of this class can exist at any one time. However, during the course of a ScenarioTool session, it is possible that more than one instance is instantiated. It only has one attribute. Namely, an ArrayList of GlossaryEntry objects. So, a Glossary object can contain one or more GlossaryEntry objects. It creates its GlossaryEntry objects as a result of a request by ScenarioController.

ScenarioController:

This is the mediator class between all classes of the GUI package and those of other packages. Its purpose is to capture user inputs and act upon them. Therefore, it is the ActionListener, MouseListener, DocumentListener, ListSelectionListener, and

CaretListener for GUI components. Only one instance of ScenarioController exists for any one session of ScenarioTool.

ScenarioController creates two Glossary instances at the beginning of the program. One is called "glossary" and the other "PERMANENTglossary". The first is a working copy of the glossary, which is updated every time the user interacts with the GlossaryGUI instance. The second, "PERMANENTglossary", holds the last saved version of the glossary. It is initialized at the beginning of program with information from a text file that remains in permanent memory, "PERMANENTglossary.txt". When the user presses the "SAVE" button on the GlossaryGUI instance, the working copy of the glossary is copied into the PERMANENTglossary. When the user exits the ScenarioTool session, the PERMANENTglossary is written off into "PERMANENTglossary.txt".

ScenarioController creates single instances of PreTokeniser, SentenceSeparator, NounFinder, VerbFinder, QtagNounFinder, QtagVerbFinder, SynonymFinder, HypernymFinder, HyponymFinder, and DefinitionFinder. It therefore holds a reference to each of them, and calls upon them when it needs to. ScenarioController also creates instances of ExecWorker whenever necessary.

- **package Tools**

PreTokeniser:

This is a natural language processing tool. It has no attributes, only a single member function. This function parses a given string argument by separating all punctuation from words (i.e., putting white space between them), and returns the resulting string. Only one instance of PreTokeniser exists for any one ScenarioTool session. ScenarioController creates it.

SentenceSeparator:

This is a natural language processing tool. It has no attributes, only a single member function. This function parses a given string argument and reproduces it one sentence per line in a text file called "sentenceSeparated_output.txt". Only one instance of SentenceSeparator exists for any one ScenarioTool session. ScenarioController creates it.

NounFinder:

This is a natural language processing tool. It has no attributes, only a single member function. This function parses a given POS-tagged string argument and returns a String array of all the nouns it finds. It assumes the PennTreebank tagging notation. Only one instance of NounFinder exists for any one ScenarioTool session. ScenarioController creates it. NOTE: The instance of this class is currently not being used by the program. It was developed because initially, the program attempted to use the Brill Tagger as POS tagger, which outputs text in PennTreebank notation.

However, because of problems with the tagger, the Qtag 3.0 was used as POS tagger instead and it outputs text in a different notation.

VerbFinder:

This is a natural language processing tool. It has no attributes, only a single member function. This function parses a given POS-tagged string argument and returns a String array of all the verbs it finds. It assumes the PennTreebank tagging notation. Only one instance of VerbFinder exists for any one ScenarioTool session. ScenarioController creates it. NOTE: The instance of this class is currently not being used by the program. It was developed because initially, the program attempted to use the Brill Tagger as POS tagger, which outputs text in PennTreebank notation. However, because of problems with the tagger, the Qtag 3.0 was used as POS tagger instead and it outputs text in a different notation.

QtagNounFinder:

This is a natural language processing tool. It has no attributes, only a single member function. This function parses a given tagged string argument and returns a String array of all the nouns it finds. It assumes the tagged string was created using Qtag 3.0 and that the tagging notation is that of Qtag 3.0. Only one instance of QtagNounFinder exists for any one ScenarioTool session. ScenarioController creates it.

QtagVerbFinder:

This is a natural language processing tool. It has no attributes, only a single member function. This function parses a given tagged string argument and returns a String array of all the verbs it finds. It assumes the tagged string was created using Qtag 3.0 and that the tagging notation is that of Qtag 3.0. Only one instance of QtagVerbFinder exists for any one ScenarioTool session. ScenarioController creates it.

SynonymFinder:

This is a natural language processing tool. It has no attributes, only a single member function. Given a word, this function invokes the WordNet executable with the given word as argument, parses the WordNet output, and returns a String array of all synonyms of the word found by WordNet. Only one instance of SynonymFinder exists for any one ScenarioTool session. ScenarioController creates it.

HyponymFinder:

This is a natural language processing tool. It has no attributes, only a single member function. Given a word, this function invokes the WordNet executable with the given word as argument, parses the WordNet output, and returns a String array of all hyponyms of the word found by WordNet. Only one instance of HyponymFinder exists for any one ScenarioTool session. ScenarioController creates it.

HypernymFinder:

This is a natural language processing tool. It has no attributes, only a single member function. Given a word, this function invokes the WordNet executable with the given word as argument, parses the WordNet output, and returns a String array of all hypernyms of the word found by WordNet. Only one instance of HypernymFinder exists for any one ScenarioTool session. ScenarioController creates it.

DefinitionFinder:

This is a natural language processing tool. It has no attributes, only a single member function. Given a word, this function invokes the WordNet executable with the given word as argument, parses the WordNet output, and returns the String definition found by WordNet of the given word. Only one instance of DefinitionFinder exists for any one ScenarioTool session. ScenarioController creates it.

- **package SwingWorkers**

ExecWorker:

This is a subclass of SwingWorker. SwingWorker's `construct()` method is overridden. It invokes Qtag 3.0 to perform its task while the GUI remains operational.

SwingWorker:

This is a class written by the Java community. It is an abstract class. It can be sub classed to create a dedicated thread, which can be used to perform time-consuming tasks while keeping the GUI operational.

- **WordNet Executable**

  WordNet is a lexical database for the English language that was created by Princeton University. It can be downloaded at *www.cogsi.princeton.edu/~wn*. It contains synonyms, hypernyms, hyponyms, definitions, and more, for many English words.

  ScenarioTool makes a system call (Java's `Runtime.exec()`) in order to run the WordNet executable. WordNet needs its own application environment and therefore, Java's runtime environment is completely taken up by WordNet, putting ScenarioTool temporarily out of commission. ScenarioTool takes up Java's runtime environment again once WordNet exits, and the program becomes responsive once more.

- **Qtag Executable:**

  Qtag 3.0 is a POS tagger (see its README file under the directory called "qtag"). It reads a text file, analyses it, and creates an output file identical to the input file except that all nouns, verbs, adjectives, adverbs, prepositions, pronouns, and articles, have been tagged.

  ScenarioTool makes a system call (Java's `Runtime.exec()`) in order to run the Qtag Executable. Qtag needs its own application environment and therefore, Java's runtime environment is completely taken up by Qtag, putting ScenarioTool temporarily out of commission. ScenarioTool takes up Java's runtime environment again once Qtag exits, and the program becomes responsive once more.

4.1.3 Design Rationale

The following describes the rationale for ScenarioTool's current architectural design described in Section 4.1.1 and Section 4.1.2. Because ScenarioTool is designed for integration into a larger project, it was designed so that integration would require changes to the least amount of classes necessary.

ScenarioController is the centre of the system. It conveys messages between all GUI classes and core classes. The GUI displays the user interface. ScenarioController captures the user's input and delegates tasks to core classes. Then, the core classes return information to ScenarioController, and ScenarioController redirects this information back to GUI classes. In turn, GUI classes display the information to the user. The workflow is shown in Figure 4.3. So, the entire control of ScenarioTool is in the hands of ScenarioController. ScenarioController is also the creator of every other class instance in the system. Both of these properties (i.e., its centralized location and its role as system initiator) make ScenarioController the "spokesperson" for ScenarioTool. Once ScenarioTool is integrated into a larger project, any changes required will be limited to the ScenarioController class.

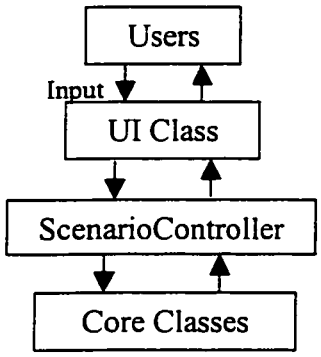

Figure 4.3 Workflow of ScenarioController

It is very likely that the ScenarioController class will become too bloated once ScenarioTool reaches its full potential. This can happen because the entire requirement elicitation process goes on when software engineers filter the edited scenario text for information regarding actors and their tasks, fill in templates of actor profiles, and filter these profiles to automatically produce use case maps. In that situation, more Controller classes can be created, each of which can handle a separate step in the entire requirements elicitation process. For example, one could add an "ActorController", which would be responsible for capturing the user's input from an entirely different set of GUI classes. This would delegate tasks to an entirely different set of core classes, and would ultimately control a different step in the requirements elicitation process – that of filling in templates of actor profiles. There would then be the need for a class that supervises all controllers, and that starts the system. Figure 4.4 explains this:

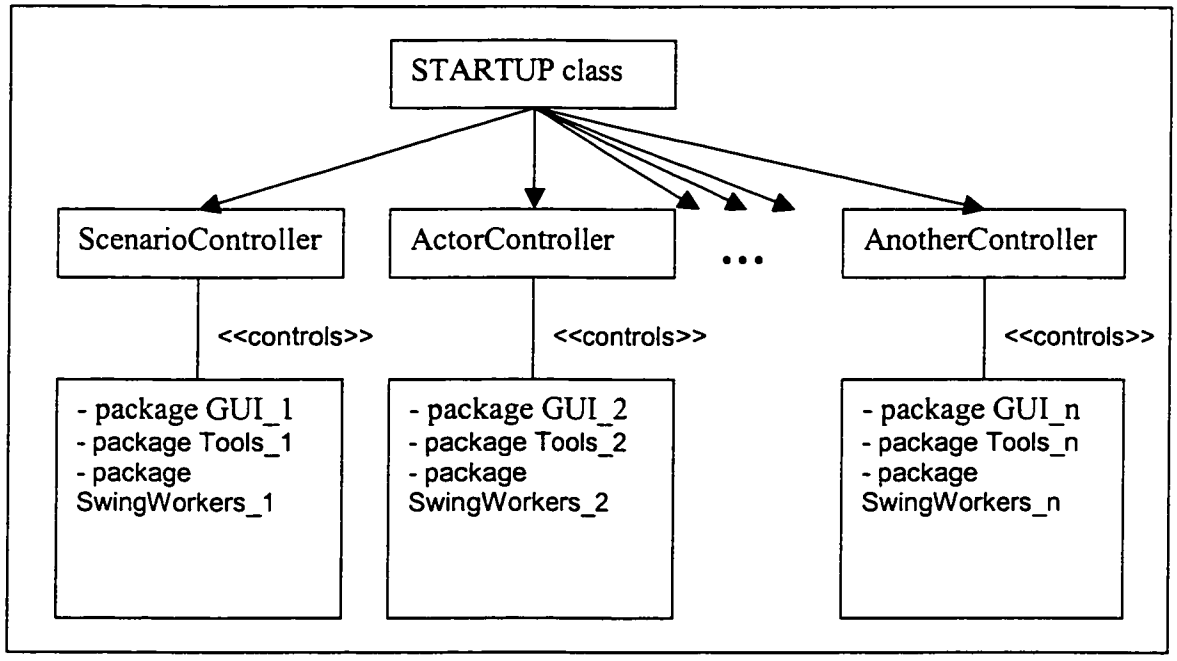

Figure 4.4 Future look of ScenarioTool's architecture

Finally, ScenarioTool's architecture is a layered one for the purpose of comprehension and maintenance. Separating the GUI classes from more domain-specific classes makes it easier for someone new to understand the function of each. It is also easier to maintain the software, because it may be easier to spot the location of a bug or to add classes.

4.1.4 Listing of Main Functionality

The following is a list that describes ScenarioTool's main functionalities. It does not include functionalities that were intended but not yet completed.

- Insert Text
- Cut, Copy, Paste Text Excerpts
- Select Word or Text Excerpt
- Find All Nouns
- Find All Verbs
- Find Definition of Highlighted Noun
- Find Definition of Highlighted Verb
- Find Definition of Selected Word
- Find Synonyms of Highlighted Noun
- Find Synonyms of Highlighted Verb
- Find Synonyms of Selected Word
- Find Hyponyms of Highlighted Noun
- Find Hyponyms of Highlighted Verb
- Find Hyponyms of Selected Word

- Find Hypernyms of Highlighted Noun
- Find Hypernyms of Highlighted Verb
- Find Hypernyms of Selected Word
- Find Synonyms Within Text of Selected Word
- Find Hyponyms Within Text of Selected Word
- Find Hypernyms Within Text of Selected Word
- Replace Synonym With Alternative Synonym
- Replace Hyponym With Alternative Hyponym
- Replace Hypernym With Alternative Hypernym
- Clear All Highlights
- Create/Maintain Glossary
    - Glossary: Add/Delete Term
    - Glossary: Add/Delete Synonym
    - Glossary: Add/Delete/Modify Definition
    - Glossary: Display List of Terms
    - Glossary: Display List of Synonyms
    - Glossary: Display Term Definition
    - Glossary: Save Glossary
    - Glossary: Open Glossary

4.2 SUCRE Prototype

The SUCRE prototype was developed in Java. As mentioned in Chapter 2, scenarios are stored in an XML-based database. The underlying document type definition (DTD) is given in Section 4.3.

4.2.1 User Interfaces

The main interface is split into two areas, as shown in Figure 4.5. The first area lists the five major features to be supported by the SUCRE system -- scenarios component, user archetype, usability goals, prototype, and UI model. The Scenario component is responsible for scenario editing. In its current state, the Scenario component is now being implemented. Area two is a working area.

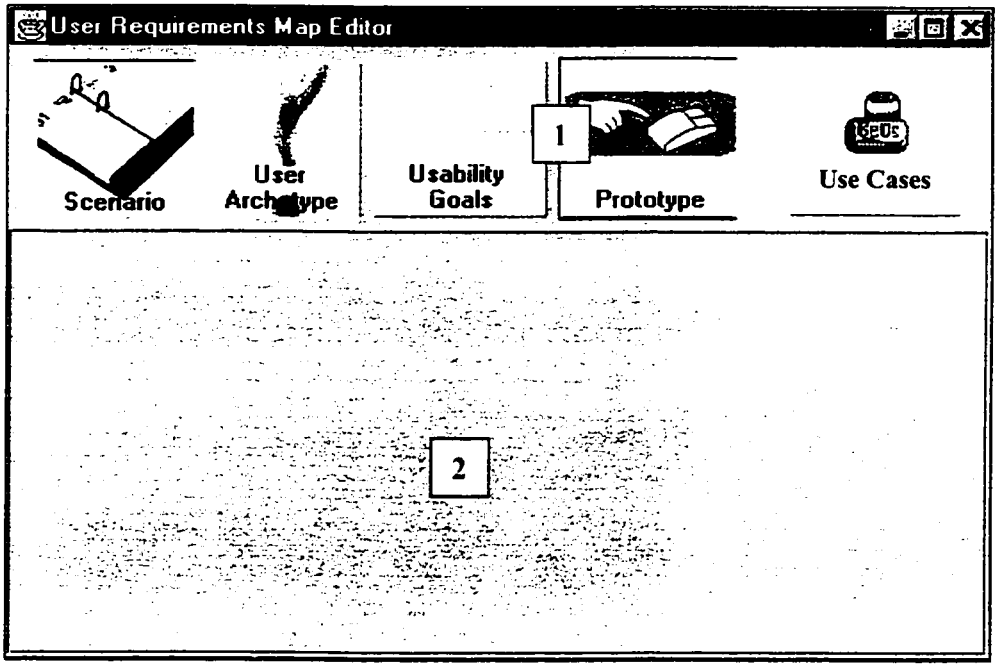

Figure 4.5 SUCRE UI prototype

Figure 4.6, Figure 4.7 and Figure 4.8 exemplify several functionalities such as, search a scenario with a given index, add and edit new scenarios, and others.

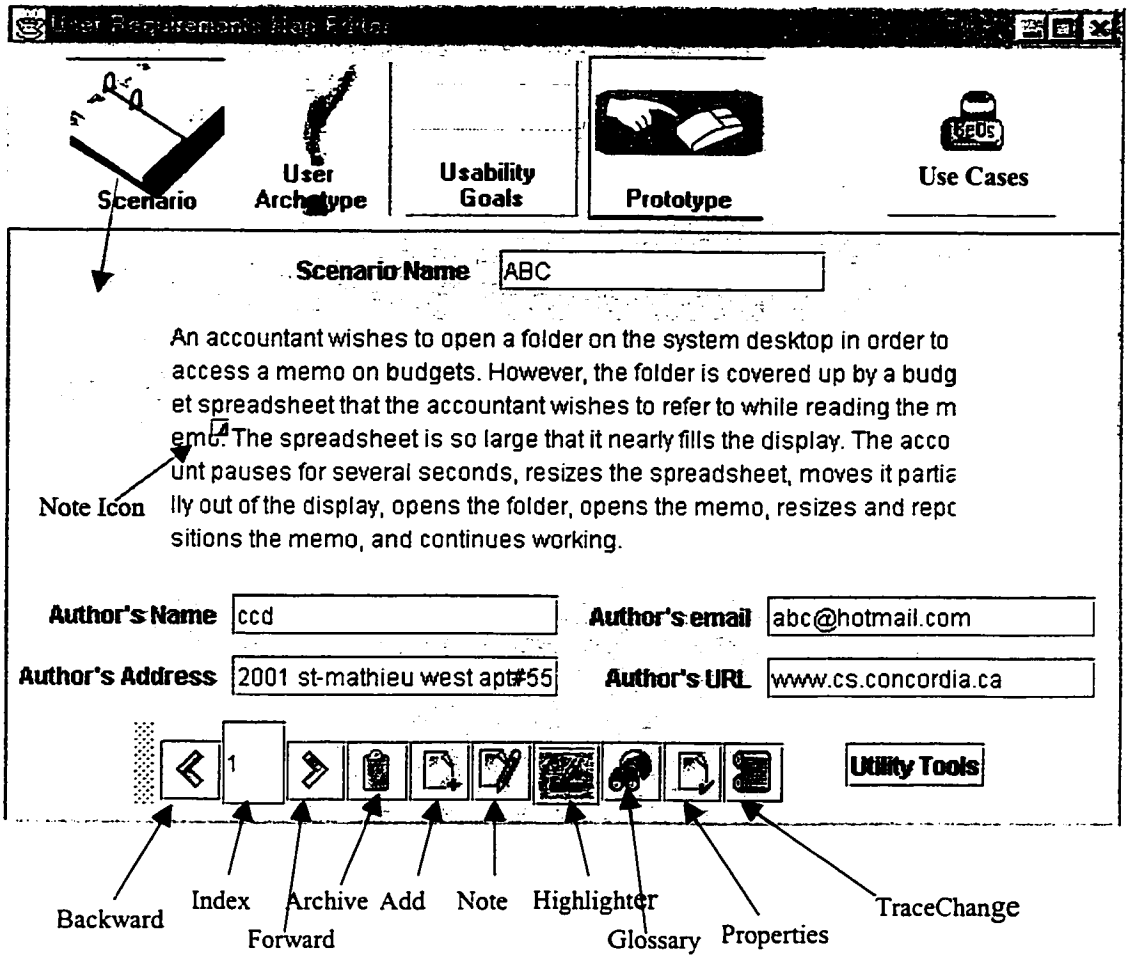

Figure 4.6 Choose scenario component

After choosing the Scenario icon in the interface, SUCRE automatically shows the scenario with index 1 in the UI (Figure 4.6).

4.2.2 List of Functionality

The functionalities that have been implemented include the following:

- The Index field displays the ID (e.g. integer) of the current scenario. Given a scenario index, search in the scenario database and display the matched scenario in the UI (Figure 4.6). By default, it is always the first scenario in the scenario database that gets displayed. The small Note Icon (Figure 4.6) in the text area indicates that a note exists for explaining certain words or phrases (e.g. note for word "memo" in Figure 4.7). It is the same as writing down notes in the margin of a book while reading. It facilitates scenario editing. The notes can be seen by pressing the small icon (Figure 4.7).

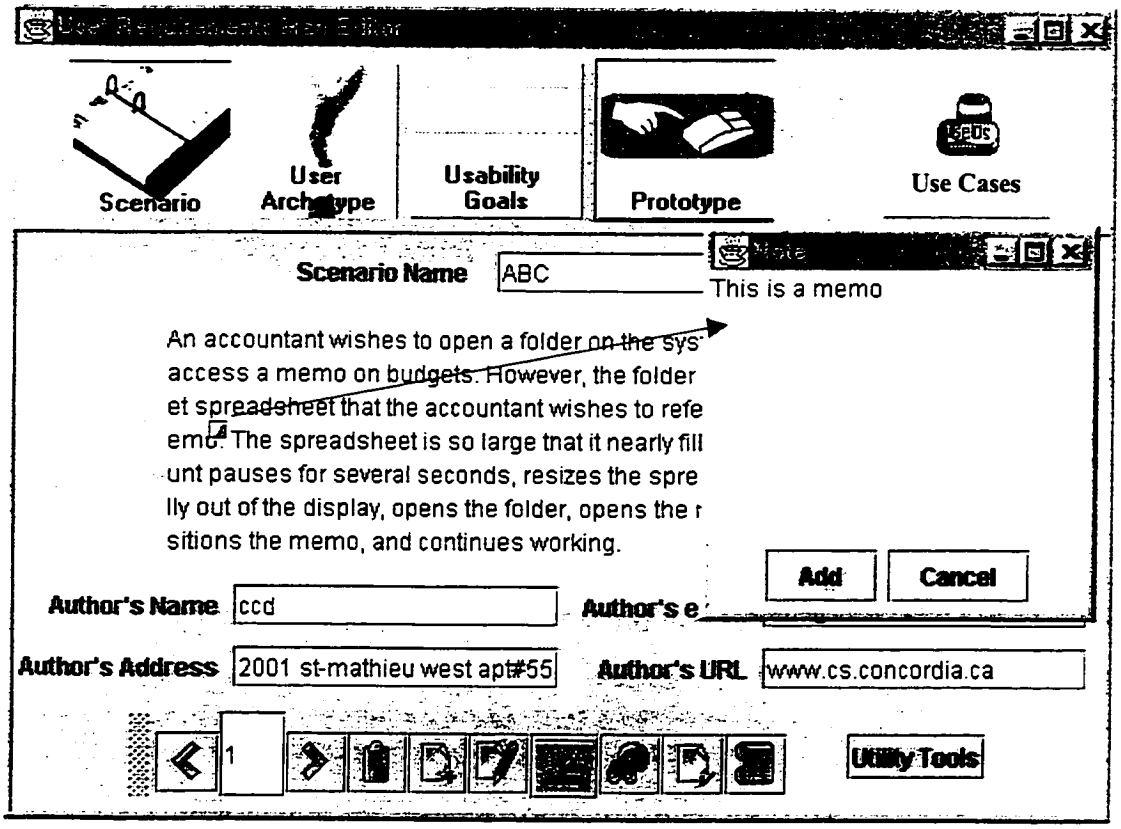

Figure 4.7 Note for the word "Memo"

- The Forward button retrieves the next scenario in the database. Usually, it is the one with the index (currentindex + 1).
- The Backward button retrieves the previous scenario in the database. Usually, it is the one with the index (currentindex - 1).
- It reads a scenario from a scenario database, and fills in the fields in the user interface as shown in Figure 4.6.
- It creates a new scenario. Users provide the information related to the scenario by filling in the fields in the UI. Then, the scenario is added into the scenario database (Figure 4.8).

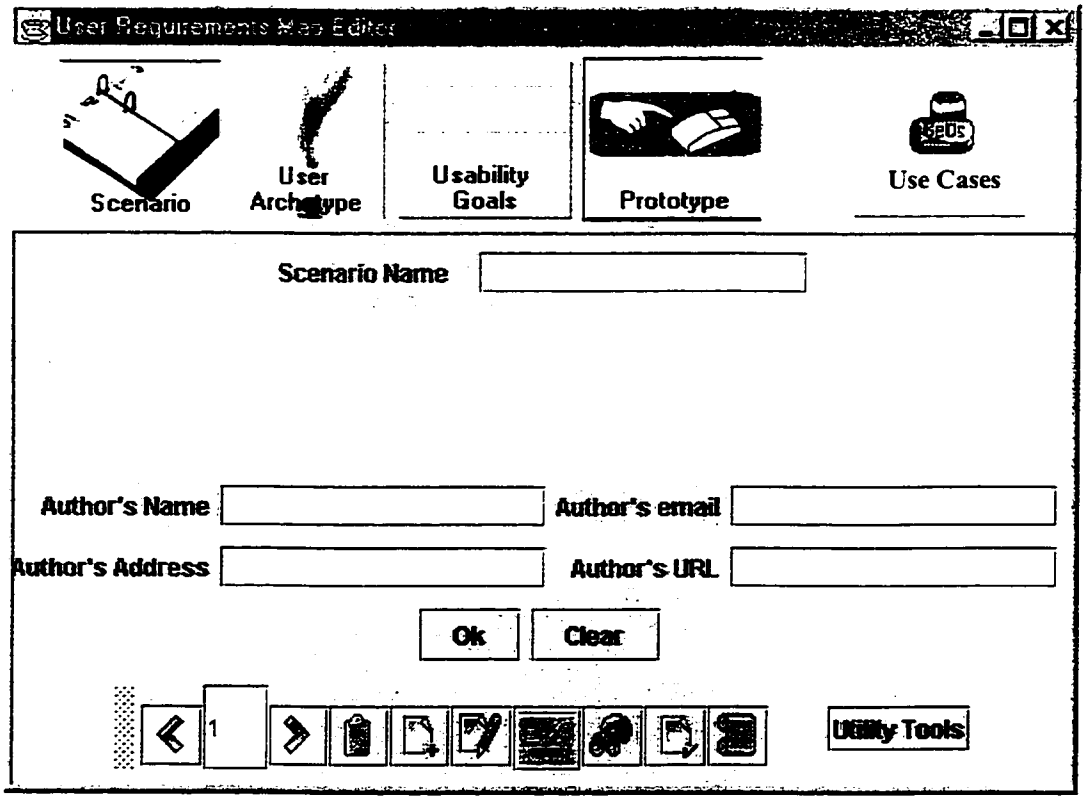

Figure 4.8 Add a new scenario

The following are the functionalities that need to be added to the UI in the future:

- Archive: Create scenario templates, which can be used in different applications such as ATM, bank. When creating scenarios for similar applications, use the template as a starting point. This is the same idea as the resume templates in MS Word.
- Add note: Highlight the text on which note is going to be added. Then edit the note and add a small note icon on the highlighted text for a certain scenario.
- Show property: show the property of a scenario such as the creation date, author information, and comments.
- Track change: track the change record of a given scenario.
- Utility Tools contains functionality such as print, a spell check, and so on.

All functionalities are grouped in a toolbar, which can be dragged and dropped anywhere (Figure 4.9).

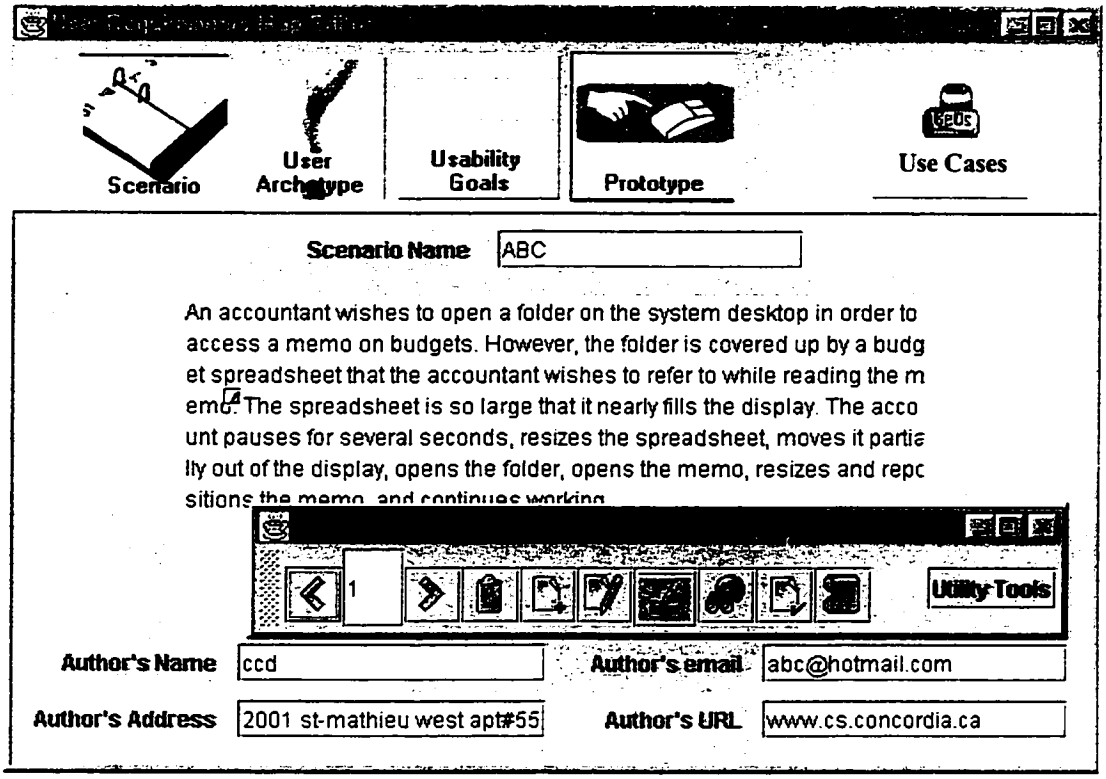

Figure 4.9 Toolbar containing all functionality

4.3 Underlying DTD and example XML file

This section demonstrates a document type definition (DTD) we developed. It gives an example of XML based scenario using this DTD.

4.3.1 DTD

<!-- ScenarioCollection.dtd -->

<!-- ScenarioCollection is the root of the document -->
<!ELEMENT ScenarioCollection (scenarios)+>

<!ELEMENT scenarios (ScenarioName, CreatedDate, ModifiedDate, Author+, Notes*, Highlighter*, BodyText)>
<!ATTLIST scenarios id ID #REQUIRED>

```
<!-- ScenarioName contains the name of the scenario -->
<!ELEMENT ScenarioName (#PCDATA)>

<!-- CreatedDate remembers the data when the scenario is created -->
<!ELEMENT CreatedDate (#PCDATA)>

<!-- ModifiedDate records the last modified data -->
<!ELEMENT ModifiedDate (#PCDATA)>

<!-- Author contains information about the author -->
<!ELEMENT Author (name, address, city, state, postalcode, country, email*, url*)>

<!-- name, address, email, url are all elements inside Author -->
<!ELEMENT name (#PCDATA)>
<!ELEMENT address (#PCDATA)>
<!ELEMENT email (#PCDATA)>
<!ELEMENT url (#PCDATA)>

<!-- Notes keep track of the notes added by the users, need to remember where it is and
what it is -->
<!ELEMENT Notes (NotePos, NoteText)+>
<!ELEMENT NotePos EMPTY>
<!ATTLIST NotePos ln CDATA #IMPLIED>
<!ATTLIST NotePos col CDATA #IMPLIED>
<!ELEMENT NoteText (#PCDATA)>

<!-- HighLighter remembers the position of the highlighted part -->
<!ELEMENT HighLighter (startpos, endpos)+>
<!ELEMENT startpos EMPTY>
<!ATTLIST startpos ln CDATA #IMPLIED>
<!ATTLIST startpos col CDATA #IMPLIED>
<!ELEMENT endpos EMPTY>
<!ATTLIST endpos ln CDATA #IMPLIED>
<!ATTLIST endpos col CDATA #IMPLIED>

<!-- BodyText keep the contents of the scenario -->
<!ELEMENT BodyText (#PCDATA)>
```

4.3.2 An Example of XML Document Using Underlying DTD

```
<?xml version="1.0"?>
<!DOCTYPE ScenarioCollection SYSTEM "ScenarioCollection.dtd">

<ScenarioCollection>
```

```
<scenarios id="1">
        <ScenarioName>ABC</ScenarioName>
        <CreatedDate>April-04-2002</CreatedDate>
        <ModifiedDate>April-29-2002</ModifiedDate>
        <Author>
                <name>ccd</name>
                <address>2001 st-mathieu west apt#555 Montreal Quebec</address>
                <email>abc@hotmail.com</email>
                <url>www.cs.concordia.ca</url>
        </Author>
        <Notes>
                <NotePos ln="20" col="50"></NotePos>
                <NoteText>This is a memo</NoteText>
        </Notes>
        <HighLighter>
                <startpos ln="10" col="15"></startpos>
                <endpos ln="18" col="25"></endpos>
        </HighLighter>
        <BodyText>
                An accountant wishes to open a folder on the system desktop in order to
                access a memo on budgets. However, the folder is covered up by a budget
                spreadsheet that the accountant wishes to refer to while reading the memo.
                The spreadsheet is so large that it nearly fills the display. The account
                pauses for several seconds, resizes the spreadsheet, moves it partially out
                of the display, opens the folder, opens the memo, resizes and repositions
                the memo, and continues working.
        </BodyText>
 </scenarios>
</ScenarioCollection>
```

4.4 Summary

This chapter presented the current work of the SUCRE system. Two prototypes are under development now – ScenarioTool and access interface to XML-based database of scenarios. This chapter then demonstrated the architecture and rationale of ScenarioTool and showed the developed access interface to the scenario database. It also illustrated functions which have already been accomplished and which will be developed in the future. At the end of this chapter, examples of DTD and XML file, which are used to present scenarios in SUCRE's scenario database, were presented.

Chapter 5 – Conclusion and Future Work

In this thesis, we reviewed different approaches for using scenarios. We investigated several tools for working with scenarios in requirement engineering. We discovered that few tools support the use of scenarios such as CREWS, CREWS-SAVRE and Scenario Plus. Among these tools, we deeply analyzed those that combine both scenarios and use cases. All these tools are based on the same scenario-based requirement process. First, they extract scenarios from use case models, then analyze the extracted scenarios. This approach assumes that use cases are written before scenarios. However, as known, stakeholders do not usually express their need in use cases. Instead, they prefer to use natural language. The aim to put stakeholders at ease, bridge the gap between stakeholders and requirement engineers, and overcome the shortcomings of existing tools motivates us to propose our SUCRE system. SUCRE combines use case and scenarios. In SUCRE, we start developing from scenario and then extract use cases. In other words, it lets stakeholders express their requirements like telling stories. Then, SUCRE analyzes the collected scenarios and generates the use cases. This approach involves the user directly and provides more accurate requirements. The help of usability experts saves requirement engineers' time and work. The gathered requirements are more consistent and complete. Finally, stakeholders have increased awareness of the system functionality, so less training will be required.

We proposed a scenario database to store our scenarios and adopted the most outstanding metadata management solution currently available – XML. This increases the flexibility and portability of our data.

5.2 Future work

The SUCRE prototype is the first part of a long-term ongoing research project. The expected benefit is to let software engineers be able to semi-automatically elicit a set of requirement from scenarios. To achieve this goal, the following questions need to be answered:

1) How can we extract the relationships among use cases from scenarios such as aggregation and association?
2) How can we improve the analysis of consistency and completeness of scenarios?
3) How can we improve the template used to generate use cases from scenarios?

Further to these questions, we need to improve our SUCRE system by developing more features.

References

[1] Ute von Reibnitz. (1998). *Scenario Techniques*. Hamburg, New York, McGraw-Hill Book Company GmbH.

[2] Margery S. Berube et al. (1993). *The American Heritage College Dictionary*. 3rd edition. Boston, New York, Houghton Mifflin Company.

[3] John M. Carroll. (2000). *Making Use Scenario-Based Design of Human-Computer Interactions*. London, England, The MIT Press Cambridge, Massachusetts.

[4] Morten Kyng. (1995). Creating Contexts for Design. In John M. Carroll, ed. *Scenario-based design: Envisioning work and technology in system development*, New York, John Wiley.

[5] Ken Eason, Susan Harker and Wendy Olphert. (1996). Representing Socio-Technical Options in the Development of New Forms of Work Organisation. *Europen Journal of Work and Organizational Psychology*, vol. 5, no. 3, pp. 399-420.

[6] Colin Potts, Kenji Takahashi and Annie I. Anton. (1994). Inquiry-Based Requirements Analysis. *IEEE Software*, vol. 11, no. 2, pp. 21-32.

[7] Alistair. G. Sutcliffe. (1997). A Technique Combination Approach to Requirements Engineering, In *Proceedings 3rd IEEE International Symposium on Requirements*

Engineering(RE'97), IEEE Computer Society Press, Annapolis, MD, January 05 - 08, 1997, pp. 65-74.

[8] Alistair G. Sutcliffe, Neil A. M. Maiden, Shailey Minocha, and Darrel Manuel. (1998). Supporting Scenario-based Requirements Engineering. *IEEE Transactions on Software Engineering,* vol. 24, issue 12, (December), pp. 1072-1088.

[9] Alistair Cockburn. (1995). *Structuring Use Cases with Goals*. [Internet] (1999) Available from <http://members.aol.com/acockburn/papers/usecases.htm> [Accessed April 5, 2003].

[10] Alistair G. Sutcliffe and Shailey Minocha. (2001). Scenario-Based Analysis of Non-Functional Requirements. In *The Seventh International Workshop on RE: Foundation for Software Quality (REFSQ'2001),* Interlaken, Switzerland, 2001, pp. 68-84.

[11] Alistair Sutcliffe. (1998). Scenario-Based Requirement Analysis. *Requirements Engineering Journal* vol 3, no. 1, pp. 48-65.

[12] Rick Kazman, Gregory Abowd, Len Bass, and Paul Clements. (1996). Scenario-Based Analysis of Software Architecture. *IEEE software,* vol 13, no. 6, (November), pp. 47-55.

[13] Colin Potts, Kenji Takahashi and Annie Anton. (1994). Inquiry-Based Scenario Analysis of System Requirements. *IEEE software,* vol 11, no. 2, pp. 21-32.

[14] Mitchell Lubars, Colin Potts and Charles Richter. (1993). Developing Initial OOA Models, In *Proceedings of the 15th international conference on Software Engineering, IEEE Computer Society Press*, Baltimore, Maryland, United States, 1993, pp. 255-264.

[15] Kenneth S. Rubin and Adele Goldberg. (1992). Object Behavior Analysis. *Communications of the ACM*, vol.35, no.9 (September), pp. 48-62.

[16] Ivar Jacobson. (1992). *Object-Oriented Software Engineering: A Use-case Driven Approach*, Addison-Wesley.

[17] John Karat and John L. Bennett. (1991). Using Scenarios in Design Meetings—A Case Study Example. In John Karat (ed.) *Taking Software Design Seriously: Practical Techniques for Human-Computer Interaction Design*, San Diego, CA, USA, Academic Press Professional, Inc. pp. 63-94.

[18] Markus Mannio and Uolevi Nikula. (2001). Requirements Elicitation Using a Combination of Prototypes and Scenarios. *Telecom Business Research Center, Lappeenranta University of Technology, Lappeenranta, Finland.* [Internet]. Available from < http://www.tbrc.fi/pubfile/TBRC_500000331.pdf> [Accessed April 5, 2003].

[19] Larry L. Constantine and Lucy A. D. Lockwood. (1999). *Software for Use: A Practical Guide to the Models and Methods of Usage-Centered Design.* Addison Wesley Pub Co.

[20] Karen McGraw and Karan Harbison. (1997). *User-Centered Requirements: The Scenario-Based Engineering Process.* Mahwah, New Jersey, Lawrence Erlbaum Associates, Inc.

[21] Daryl Kulak and Eamonn Guiney. (2000). *Use Cases – Requirements in Context*, Addison-Wesley Pub. Co.

[22] Scott P. Overmyer. (1999). The Use of Scenarios in Developing, Validating, and Specifying Requirements for Interactive Systems: A Case Study from a NASA Project. In *The Fifth International Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ'99),* Heidelberg, Germany, 1999, paper 9.

[23] Bonnie A. Nardi. (1992). The Use of Scenarios in Design. *ACM SIGCHI Bulletin* vol24, no. 4 (October), pp 13-14.

[24] Mohammed Elkoutbi and Rudolf K. Keller. (2000). User Interface Prototyping based on UML Scenarios and High-level Petri Nets [Internet]. Available from

<http://www.iro.umontreal.ca/~labgelo/Publications/Papers/atpn-2000.pdf>
[Accessed April 5, 2003].

[25] Richard M. Young and Phil Barnard. (1987). The use of Scenarios in Human-Computer Interaction Research: Turbocharging the Tortoise of Cumulative Science. ACM SIGCHI Bulletin vol. 17, no. SI, (May) pp. 291-296.

[26] IBM Incorporation. (1991). Systems Application Architecture: Common User Access – Guide to User Interface Design – Advanced Interface Design Reference.

[27] Mohammed Elkoutbi and Rudolf K. Keller. (1998). Modeling Interactive Systems with Hierarchical Colored Petri Nets. In *Proc. of 1998 Adv. Simulation Technologies Conference. Soc. for Comp. Simulation Intl. HPC98 Special session on Petri-Nets.* Boston, MA, April 1998. pp. 432-437.

[28] Richard Kostelanetz. (1980). *Scenarios: Scripts to perform.* Brooklyn, N.Y., Assembling Press Participation Projects Foundation, Inc.

[29] Alistair G. Sutcliffe. (1995). Requirements Rationales: Integrating Approaches to Requirements Analysis. In *Proceedings of Designing Interactive Systems (DIS'95), ACM Press,* New York, 1995, pp. 33-42.

[30] Peter Coad, David North and Mark Mayfield. (1995). *Object Models: Strategies, Patterns and Applications.* New Jersey, Prentice-Hall.

[31] Charles Sheppard. (1997). *Scenarios for Evaluation of Collaboration Tools.* [Internet]. Available from <http://zing.ncsl.nist.gov/nist-icv/documents/node7.html> [Accessed April 5, 2003].

[32] CaliberRM. (2002). In *Starbase Corporation.* [Internet]. Available from <http://www.borland.com/caliber/> [Accessed April 5, 2003].

[33] Rational RequisitePro. (2002). In *Rational Software Company.* [Internet]. Available from <http://www.rational.com/products/reqpro/index.jsp> [Accessed April 5, 2003].

[34] AxiomDsn. (1999) In *Structured Technology Group, Inc.* [Internet] February 21, 2003. Available from <http://www.stgcase.com/casetools/axiomdsn.html> [Accessed April 5, 2003].

[35] EasyRM. (2000). In *Cybernetic Intelligence GmbH.* [Internet]. Available from <http://www.easy-rm.com/> [Accessed April 5, 2003].

[36] Peter Haumer, Klaus Pohl, and Klaus Weidenhaupt. (1998). Requirements Elicitation and Validation with Real World Scenes. *IEEE Transaction on Software Engineering,* vol. 24, no. 12 (December), pp. 1036-1054.

[37] Hong Zhu and Lingzi Jin. (2002). Scenario Analysis in an Automated Tool for Requirements Engineering, *Journal of Systems and Software* vol 61, no. 2, (March), pp. 145-169.

[38] Neil A. Maiden, Shailey Minocha, Alistair G. Sutcliffe, Darrel Manuel, and Michele Ryan. (1999). A co-operative scenario based approach to acquisition and validation of system requirements: how exception can help! *Interacting with Computers* vol 11, no. 4 (April), pp. 645–664.

[39] Ian Alexander. (1997). In *Scenario Plus.* [Internet] May 10, 2002. Available from <http://www.scenarioplus.org.uk/> [Accessed April 5, 2003].

[40] DOORS. (2001). In *Telelogic Company.* [Internet] Available from <http://www.telelogic.com/products/doorsers/> [Accessed April 5, 2003].

[41] Brett McLanughlin. (2001). *Java and XML.* 2nd edition. Sebastopol, CA, O'Reilly & Associates.

[42] Charles Ashbacher. (2000). *Teach yourself XML in 24 hours.* Sams Publishing.

[43] Adrienne Tannenbaum. (2001). *Metadata Solutions Using Metamodels, Repositories, XML, and Enterprise Portals to Generate Information on Demand.* Addison-Wesley Professional.

[44] Kimmond, R.M. (1995). Survey into the acceptance of prototyping in software development. In *Proceedings from the sixth IEEE International Workshop on Rapid System Prototyping (RSP'95),* Chapel Hill, North Carolina, June 1995, pp. 147-152.

[45] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. (1998). Scenarios in System Development: Current Practice, *IEEE Software*, vol 15, no. 2, (March/April) pp. 34-45.

[46] Neil Maiden, Shailey Minocha, Keith Manning, and Michele Ryan. (1997). A Software Tool for Scenario Generation and Use. In *Proceedings of the Third International Workshop on Requirements Engineering: Foundation for Software Quality.* Barcelona, Spain, 1997.

[47] Colette Rolland and Camille Ben Achour. (1998). Guiding the Construction of Textual Use Case Specifications. *Data and Knowledge Engineering,* vol. 25, no. 3 (April), pp. 125-160.

[48] David R. Dowty, Robert E. Walls, and Stanley Peters. (1981). *Introduction to Montague Semantics.* Reidel-Klawer.

[49] John F. Sowa. (1984). *Conceptual Structures: Information Processing in Mind and Machine.* Addison-Wesley Systems Programming Series. Boston, MA, USA, Addison-Wesley Longman Publishing Co., Inc.