Softw Syst Model (2008) 7:67–83
DOI 10.1007/s10270-006-0039-3

REGULAR PAPER

Producing robust use case diagrams via reverse engineering of use case descriptions

Mohamed El-Attar · James Miller

Received: 2 March 2006 / Revised: 6 October 2006 / Accepted: 22 November 2006 / Published online: 23 January 2007
© Springer-Verlag 2007

Abstract In a use case driven development process, a use case model is utilized by a development team to construct an object-oriented software system. The large degree of informality in use case models, coupled with the fact that use case models directly affect the quality of all aspects of the development process, is a very dangerous combination. Naturally, informal use case models are prone to contain problems, which lead to the injection of defects at a very early stage in the development cycle. In this paper, we propose a structure that will aid the detection and elimination of potential defects caused by inconsistencies present in use case models. The structure contains a small set of formal constructs that will allow use case models to be machine readable while retaining their readability by retaining a large degree of unstructured natural language. In this paper we also propose a process which utilizes the structured use cases to systematically generate their corresponding use case diagrams and vice versa. Finally a tool provides support for the new structure and the new process. To demonstrate the feasibility of this approach, a simple study is conducted using a mock online hockey store system.

1 Introduction

Use case modeling [37], since it was introduced in the early 1990s by Ivar Jacobson has been constantly gaining wide acceptance by analysts, designers, testers and other stakeholders of a project. Use case modeling can be used to drive the design phase, the testing phase [28] and can be utilized for managerial purposes such as effort estimation [3] and business modeling [26]. The success experienced by use case modeling is chiefly because it is very simple to use to effectively describe the functional requirements of a system. Another attractive aspect of use case modeling is that it contains a small diagrammatic notational subset and a large degree of natural language. This allows all stakeholders of a project to understand the use case model, even those who are not technically equipped, which in turn will ensure that all stakeholders have a common understanding and agreement upon the capabilities and features of the system.

The large degree of informality contained in the use case descriptions often causes use case descriptions to be inconsistent with their corresponding use case diagrams. Moreover, inconsistencies may reside within the use case descriptions themselves. In a use case driven approach [25] such as the Rational Unified Process (RUP) [29,30,39], use case models are used to produce other UML artifacts such as activity and sequence diagrams [21,22,25,32,38,41,50]. Hence, it is important to invest in producing high quality use case models that will yield the production of other high quality UML artifacts. Consistency is a key quality attribute of use case models. Ensuring the consistency between use case descriptions and their corresponding diagrams requires a great deal of discipline from analysts, which seldom exists. Moreover, producing consistent use case models has been chiefly dependant on the experience of analysts. It is common knowledge that the expertise of analysts in industry varies significantly. Often, junior analysts are required to develop use case models, which will be highly

Communicated by Prof. August-Wilhelm Scheer.

M. El-Attar (✉) · J. Miller
STEAM Laboratory, Department of Electrical and Computer Engineering, University of Alberta, Edmonton, AB, Canada
e-mail: melattar@ece.ualberta.ca

J. Miller
e-mail: jm@ece.ualberta.ca

vulnerable to inconsistencies. The produced inconsistent use case models may potentially lead to the production of low quality software systems. Therefore, it is essential to devise a structure that will aid the production of consistent use case descriptions. It is also important that this structure can be used to ensure the consistency between the use case descriptions and their corresponding diagrams; while maintaining the ability of these diagrams to be understandable to all stakeholders, including "non-technical" stakeholders.

In this paper we propose a structure to assist with the description of use cases. The proposed use case description structure is called **S**imple **S**tructured **U**se **C**ase **D**escriptions (SSUCD). SSUCD serves as a guideline for authors in producing their use cases. Moreover, the SSUCD form will allow the use case descriptions to be machine readable. In this paper we devised a technique named **R**everse **E**ngineering of **U**se **C**ase **D**iagrams (REUCD), which will systematically generate use case diagrams from use cases that are described in the SSUCD form. Use case diagrams are developed at a much higher level of abstraction than use case descriptions. Hence, use case diagrams can be accurately reverse engineered from use case descriptions using REUCD. REUCD extracts limited information from use case descriptions to generate use case diagrams. Figure 1 shows an overview of SSUCD and REUCD. The REUCD process may also be reversed, whereby the use case diagram is initially developed and used to systematically generate 'skeletons' for the use case descriptions. Details about the events occurring in the use cases can then be manually completed. This concept is discussed in further detail in Sect. 4. However, the theme throughout this paper will be aimed at initially composing use case descriptions then systematically generating the diagrams from them, since use case descriptions contain all the information required to produce a complete use case diagram. In this paper, we present our featured tool SAREUCD (**S**imple **A**utomated **REUCD**), which will automate the REUCD process and increase the speed and accuracy of its application.

The remainder of this paper is organized as follows; in Sect. 2 we present a brief introduction to use case modeling and various use case authoring styles and structures. Section 2 also describes why inconsistencies exist in use case models and what their potential consequences are. Section 3 presents a detailed description of the proposed use case description structure SSUCD. Section 4 provides an overview of how use cases described in the SSUCD form can be used to systematically produce use case diagrams. An overview of the tool SAREUCD is presented in Sect. 5. Section 6 discusses the design of the SSUCD modeling language and its adherence to the quality principles that are expected to exist in any modeling language. In Sect. 7 we present a simplified Online Hockey Team Store system to demonstrate the application of SSUCD, REUCD and SAREUCD. Finally, Sect. 8 concludes and discusses future work.

2 Background

2.1 Brief introduction to use case modeling

This section presents a brief overview of use case modeling. A use case model contains three main components. Firstly, the use case diagram, which contains two main components: use cases and actors. Use cases are depicted as ovals, and serve as a visual summary of the services that are offered by the system to its actors. Actors are classes of users who can benefit from services offered by the system. A use case diagram also shows the relationships that exist between the use cases and the actors and between the use cases themselves. Secondly, the use case descriptions. The use case descriptions essentially describe the flow of events required to occur in order for an actor to benefit from a service offered by

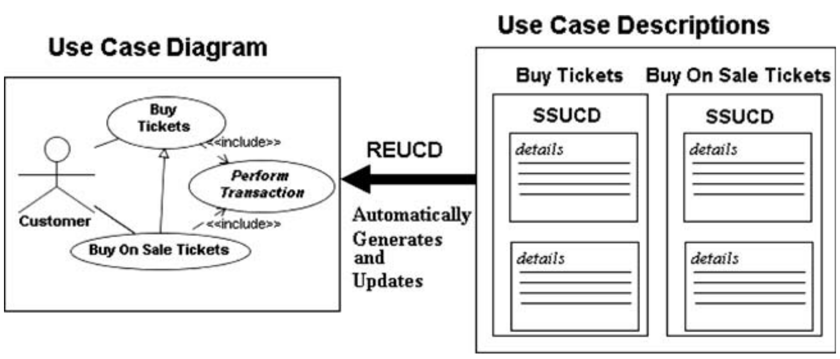


Fig. 1 The application of the REUCD process to systematically generate use case diagrams from descriptions

the corresponding use case [12]. There must be one use case description corresponding to each of the use cases depicted in a use case diagram. Traditionally, descriptions are authored in textual form. Lastly, a use case model makes use of a glossary component. The glossary is not explicit to use case models; it is used for other activities during development. The glossary will contain the explanations of common terms used throughout a project. The remainder of this paper will assume a basic understanding of the fundamental notations utilized in use case modeling.

2.2 Use case authoring styles

There have been many different approaches to authoring use case descriptions. Each approach is devised to describe use cases at different levels of detail and structure. For example, use cases can be described using a single short paragraph. Caution must be exercised while describing use cases in such form as this approach tends to assume that other stakeholders have a great degree of domain knowledge, which is not always the case. On the other hand, use cases can be described using a "full blown" approach that mentions every possible detail. Some approaches structure the use case descriptions very carefully, while others do not incorporate any structure. Johansson [27] analyzed and discussed problems that arise when attempting to construct a use case model and write the corresponding descriptions of use cases for a weather station system. The problems were principally caused by a lack of guidelines for authoring use case descriptions. The paper concludes by urging for guidelines for use case modeling.

- A number of authors have developed such guidelines, principally: Achour et al. [8] examined two different types of guidelines; styling guidelines (SGs) and content guidelines (CGs). The styling guidelines were mainly derived from current best practices such as those presented in [15,16]. The styling guidelines are used to improve the quality of use case structures. On the other hand, content guidelines were mainly derived from linguistics, artificial intelligence and previous experiences in applying Case Grammars to requirements analysis. The content guidelines are used to indicate the expected contents of use cases. The authors present an evaluation of their work which comprises seven hypotheses, three of which are related to CGs and the remaining four are related to SGs. The experimental procedure utilized 69 software engineers who had professional experience and participated in a half day presentation on use case authoring and modeling. The results of the study conclude that use case authoring guidelines generally improve the quality of the descriptions. The authors emphasize that even though authoring guidelines helped, they rarely lead to perfect use cases. Therefore, the authors suggest that the use case descriptions should be checked whenever quality is an issue.
- Firesmith [19] describes a broader range of guidelines. These guidelines fall into the following categories: modeling tools and languages; modeling externals; modeling use cases; modeling use case paths; and general guidelines.
- A number of use case authoring guidelines have been devised to capture the requirements for special types of systems. Anderson et al. [6] described styles of documenting business rules in use cases. Constantine [17] and Biddle et al. [9] described styles that lead to 'essential' use cases. Rebecca [45] has also promoted a conversational style of authoring use cases.
- Cockburn [16] described a set of eighteen different styles of writing use case descriptions, collected while working upon various projects, matching one by Jacobson [25]. The authoring style proposed by Jacobson has heavily popularized due to its adoption by RUP. The work presented by Bittner and Spence [10] further developed this style of use case authoring. However, this style in its current state experiences several limitations:

a) The authoring style lacks the required amount of structure to allow the use case descriptions to be machine readable, which will impede the systematic process of:
   - Generating use case diagrams from the descriptions.
   - Generating the 'skeletons' of use case descriptions from the diagrams.
   - Verifying the consistency between the use case diagram and the descriptions.

b) There is no mechanism available to:
   - Declare *generalization* relationships between use cases.
   - Declare *generalization* relationships between actors.
   - Declare *abstract* use cases.
   - Declare *abstract* actors.
   - Declare that a use case *implements* an *abstract* use case..
   - Allow an *extension* use case to reference the *base* use case it *extends*.

The structure proposed in this paper can be used to overcome the limitations outlined above. The key feature of SSUCD is that it will ensure the consistency between the use case descriptions and their corresponding diagrams.

2.3 Maintaining the readability of use case descriptions

The core feature behind the popularity of use case models is the great deal of unstructured natural language that the use case descriptions contain. The informality contained in use case descriptions makes it accessible by stakeholders who are not familiar with common programming jargon and acronyms. Customers are often not technical specialists and thus the informality in use case descriptions allows them to read and review the use case descriptions and provide feedback. Unstructured natural language is an indispensable component of use case descriptions. Unfortunately, it is impossible to formally analyze use case descriptions that are completely composed of unstructured natural language. Use case descriptions become highly vulnerable to poor quality attributes such as inconsistencies, incorrectness and incompleteness. Use case descriptions can be formally analyzed only if they adhere to a formal structure. However, describing use cases using only formal constructs will greatly reduce their readability and make them inaccessible to many stakeholders. Therefore, a tradeoff exists between the amount of unstructured natural language and formal constructs that use case descriptions may contain. The SSUCD structure provides a hybrid solution to this problem. The SSUCD structure contains a very limited set of formal constructs, the minimal amount required, while allowing analysts the flexibility and liberty of using as much unstructured natural language as possible. The SSUCD structure will allow a great deal of formal analysis to be performed on the descriptions while retaining their readability. Further structure can be added to the SSUCD descriptions in order to generate other types of UML artifacts. For example, the SUCD (Structured Use Case Descriptions) form [50] adds more formality and structure to SSUCD descriptions to allow the systematic generation of Activity diagrams.

2.4 The problem with inconsistencies in use case models

Many researchers have determined that inconsistencies in a use case model have harmful consequences. Inconsistencies can exist in use case models in various forms. The consequences of an inconsistency depend on the form that the inconsistency exists in:

- Anda et al. [5] outlined a taxonomy of the core categories of defects in use case models. They demonstrated that inconsistencies are a key category of defects which severely hampers the overall quality of a use case model. The consequences of the stated forms of inconsistencies were also outlined, which has shown to affect every aspect of the development process; from producing wrong, missing and inaccurate functionalities to producing systems that are difficult to test.
- Chandrasekaran [14] has explained that inconsistencies in a use case model are generally symptomatic of one of two problems; Firstly, the use case model might be handling concepts that are not defined or understood properly. Secondly, there maybe an ambiguity in the domain model.
- Lilly [34] and Bittner et al. [10] outlined a number of inconsistencies that explicitly exist in use case diagrams as well as other types of inconsistencies that may exist throughout the use case model. These authors have also explained the harmful consequences of these inconsistencies. For example, in [34], it is shown that an inconsistent system boundary may cause designers to implement the behavior of entities external to the system. This in turn requires more effort from the development team than is actually required, causing the project to fall behind schedule and go over budget.
- Ambler [2] warns that inconsistencies in use case models are usually a sign of missing or vague information. The literature has repeatedly shown that teams often fail due to a lack of details in the use case model rather than too much detail [10]. Ambler also warns that too many inconsistencies may cause the use case model to become "out-of-date" and therefore becoming useless. Therefore, use case models need to remain consistent to be effective in the development process.
- Consistency has always been a sought after as an essential quality attribute for use case models [4,8, 19,24,31,41]. Reviewing use case models is a highly recommended practice [7,31,42], used to assure their quality by assuring that they possess a great deal of consistency.
- Other researchers have devised techniques to incorporate and ensure consistency in the use case models being developed. McCoy [36] introduces a tool, which provides a template for modelers to input information about their use cases into a repository. The template ensures consistency during entry of the information. Achour et al. [8] compiled a set of styling and content guidelines to improve the quality of the use case descriptions. A number of these

guidelines are either directly or indirectly aimed at ensuring consistency within the use case descriptions. Butler et al. [13] introduced the concept of refactoring use case models. Butler explained that refactoring can improve the consistency of the use case models. Ren et al. [40] has developed a tool that implements the refactoring concepts presented in [13].

It can be concluded that it is desirable to minimize inconsistencies within use case models. Using the SSUCD structure and the REUCD process ensures that the use case descriptions and their diagram(s) are consistent with each other. For example, if the descriptions state that a certain use case is associated with a certain actor, then it will be ensured that an association relationship in the use case diagram will be depicted linking the given use case with the given actor. The SSUCD structure and the REUCD process do not however ensure that inconsistencies, present in the segments that are written in unstructured natural language, will be detected or eliminated. These segments require domain expertise to verify their consistency. For example, if a use case states that a theatre's seating capacity is 1200 while another use case states that the given seating capacity is 1400, then this type of inconsistency requires manual inspection (or review) to be detected and eliminated.

2.5 Inconsistencies: a closer look

An obvious argument at that point would be: if the heart of the use case model is in the descriptions while the diagrams only serve as a visual roadmap then:

> Why bother with the use case diagram? Why not just use the use case descriptions only to drive the development process? In such a case, when only the descriptions are considered, then ensuring the consistency between the use case descriptions and their diagrams is not important!

Even though this argument might be valid for very trivial systems, there still remain several problems if only the use case descriptions are considered. If the system is very complex, then a use case description might span over five pages to be adequately described it [10]. In such case, if a team member wanted to know the actors that are associated with a given use case, it would be more efficient and accurate to simply look up this information in the diagram rather than going through several pages of text. Use case diagrams are able to provide an overview of a system at a glance; while examining a set of use case descriptions cannot. Therefore, stakeholders might be misled about the general purpose of the system if the use case diagram did not accurately represent the descriptions. Hence, use case diagrams remain an indispensable component of use case models, and therefore if a use case diagram does not have an accurate representation of the descriptions, then this will lead to the design of a faulty system. Moreover, since the SSUCD structure allows use case descriptions to be machine readable, a great deal of information can be automatically extracted from the descriptions, such as the use cases that are *included* by a given *base* use case.

3 Simple structured use case description (SSUCD)

In this section we describe our proposed structure (SSUCD) for writing use case descriptions. Use cases described using the SSUCD structure contains four main sections, these are: (a) Use Case Name, (b) Associated Actors, (c) Description, (d) Extension Points and Extended Use Cases. With the exception of the "Description" section, these sections utilize a handful of keywords to embed the required structure. All keywords are written in uppercase for readability purposes. The "Description" section on the other hand is populated using natural language to allow for maximum flexibility and expressiveness by use case authors. Other sections can be added to cater to specific needs; the additional sections must be contained as subsections of the "Description" section. There have many templates presented in the literature for describing use cases [15,23,24,31,35,42]. The structured sections incorporated by SSUCD are the common sections found in many templates presented in the literature.

3.1 A brief introduction to the elements of SSUCD

For a fully detailed reference guide to SSUCD and its syntax, we refer interested readers to [49]. The subsequent sections will briefly present the structural elements of SSUCD and how they are used to map use case descriptions to diagrams (see Table 1), which is further illustrated using the Online Hockey Store System presented in Sect. 7.

(a) Use case name section:
The "Use Case Name" section states characteristic properties about a given use case. This section starts with the label "Use Case Name:".
Structural elements and keywords:
a) The name of the use case:
The "Use Case Name" section must state the name of the use case.
b) If the use case is *abstract*:

Table 1 Quality principles that should be present in modeling languages

Simplicity	The language does not contain any unnecessary complexity
Uniqueness	There are no overlapping features or redundant ones
Consistency	The language elements and features allow the required goals to be met
Seamlessness	The ability to generate code from the models
Reversibility	Changes at any point in the development can be propagated back to the models
Scalability	The ability to model large and small systems
Supportability	The ability for humans to utilize the language and the availability of tool support
Reliability	The language promotes the development of reliable software
Space economy	Models produced must be concise, showing the required information without clustering the view

This is stated using the keyword ABSTRACT. If the use case is not *abstract* then this keyword is omitted. On the other hand, if the use case implements an *abstract* use case, then this is stated using the keyword IMPLEMENTS followed by the name of the *abstract* use case. Similarly, if the use case does not implement any *abstract* use cases, then this keyword is omitted.
c) If the use case specializes other use cases:
This is stated using the keyword SPECIALIZES followed by the name of the parent use case. If the use case does not have any parents, then this keyword is omitted.
Mapping information and examples:
a) The name of the use case:
The name stated in the "Use Case Name" section must have a use case symbol (an oval) in the diagram with a matching name (see Fig. 2).
b) Abstract use cases and their implementation:
The name of an *abstract* use case is displayed in *italic* font in the diagram. A use case implementing an *abstract* use case creates a *generalization* relationship arrow in the diagram, originating from the implementing use case and directed towards the *abstract* use case (see Fig. 3).
c) Generalization between use cases:
The use case name as *specialized* by a child use case creates a *generalization* relationship link between the involved use cases, originating from the child use case and directed towards the parent use case (see Fig. 4).
(b) Associated actors section:
Actors are associated with use cases to perform the described behavior and to achieve a certain goal. Actors can be associated with use cases for various reasons. Each use case must specify the actors that are involved with it. The "Associated Actors" section is used to list the involved actors with only commas separating them.
Mapping information and example:
Actors listed in this section must have an association relationship link connecting the use case and the corresponding actors in the diagram (see Fig. 5).
(c) Description section:
The "Description" section contains the core behavior of the use case. As mentioned earlier, the "Description" section is intentionally designed to be populated using natural language to allow use case authors utmost flexibility with respect to describing their use cases. Another reason is to minimize the amount of learning required by the users of SSUCD. If an author needs to add a new section, the new section is simply written using natural language as part of the "Description" section.
Structural elements and keywords:
There is only one keyword in this section which states that the given use case *includes* another use case. An *include* relationship is stated using the keyword INCLUDE followed by the name of the *inclusion* use case enclosed in angled brackets "INCLUDE < *inclusion* use case name>".
Mapping information and example:
An INCLUDE statement present in the "Description" section of a use case creates an *include* relationship link originating from the *base* use case and directed

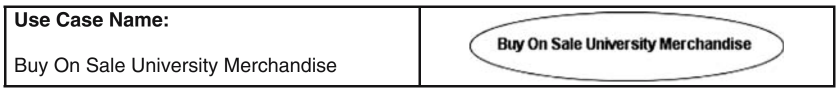
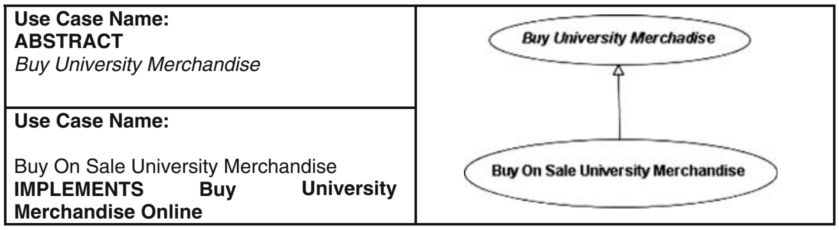

Use Case Name:
Buy On Sale University Merchandise

Fig. 2 Use case name and its representation


Use Case Name:
ABSTRACT
Buy University Merchandise

Use Case Name:
Buy On Sale University Merchandise
IMPLEMENTS Buy University Merchandise Online

Fig. 3 Abstraction and implementation in use cases and their representation

Use Case Name: Buy University Merchandise
Use Case Name: Buy On Sale University Merchandise **SPECIALIZES Buy University Merchandise**

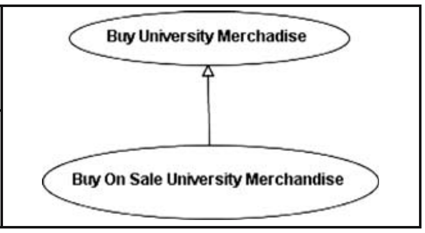


Fig. 4 Generalization between use cases its representation

Example:

Use Case Name: Enroll New Member
Associated Actors: Librarian, Member

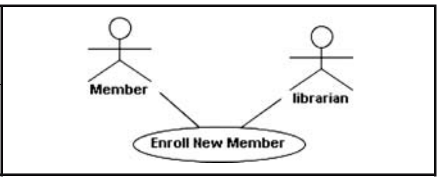


Fig. 5 Associations between use cases and actors and its representation

Use Case Name: *Enroll New Member*
Description: *… before a new member can be enrolled,* **INCLUDE <Authenticate User>** *must be performed to authenticate the staff …*

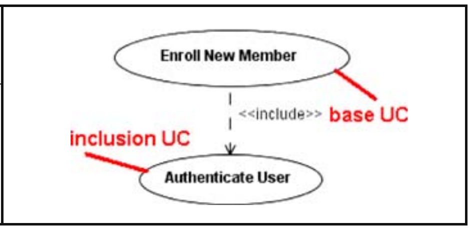


Fig. 6 The *include* relationship represented in the use case description body

towards the *inclusion* use case stated in the INCLUDE statement (see Fig. 6).

(d) Extension points section and extended use cases section:

The "Extension Points" section lists all the public extension points that belong to the given use case. Although there are two types of extension points; public and private, only public extension points appear on the use case diagram. Hence, private extension points can be described using natural language within the Description "section" without the need to add further structure. The "Extended Use Cases" section lists all the use cases that the given use case *extends*.

Structural elements and keywords:

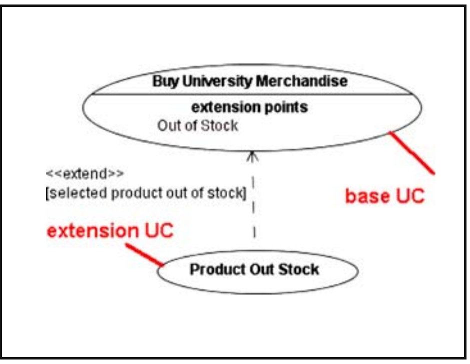
- The extension points section
  Base use cases that are extended should not have any knowledge of their *extension* use cases. *Base* use cases only provide public extension points for *extension* use cases to specify the locations where the extended behavior will be inserted. This is because *base* use cases are expected to be complete even without the incorporation of the *extension* use cases. Public extension points listed under an "Extension Points" section are separated using carriage return.
- The Extended Use Cases Section
  Conversely, *extension* use cases are expected to have knowledge of the *base* use cases they *extend*. The "Extended Use Cases" section lists the *base* use cases that the given use case *extends*. An *extended* use case is stated using the keyword "Base UC Name:" followed by its name. If an *extension* use case *extends* a base use case at a given public extension point, the extension point is stated using the keyword "Extension Point:" followed by the name of the extension point. Therefore, using the "Extension Point" construct is optional since stating a public extension point for a given *extend* relationship is optional. If a condition needs to be set for an *extend* relationship, this is stated using the keyword "IF" followed by the condition written in natural language. Specifying a condition for an *extend* relationship is optional. Hence, using the "IF" construct is also optional (see Fig. 7).

3.2 Formalizing the SSUCD structure grammar

It is essential for the grammar and constructs of the SSUCD structure to be formalized. Formalizing the SSUCD structure will provide a strict guideline to use case authors when composing use case descriptions, so that there is no disagreement or ambiguity as to what is allowed and what is not. The grammar of the SSUCD structure is defined in E-BNF and can be located at [49].

Fig. 7 The *extend* relationship represented in the use case description body

Example:

Use Case Name: Buy University Merchandise Online	
Extension Points: **out of stock**	
Use Case Name: Product Out of Stock	
Extended Use Cases: **Base UC Name:** Buy University Merchandise Online **Extension Point:** out of stock **IF** selected product is out of stock	

4 Consistency and mapping rules between use case descriptions and diagrams

In this section we will introduce the REUCD (Reverse Engineering of Use Case Diagrams) process which is used to systematically map SSUCD's structural constructs to diagrammatic notations that form use case diagrams. This systematic process is automated using the tool SAREUCD (see Sect. 5), which will ensure the consistency and speed of the process.

The process of generating use case diagrams from use case descriptions and vice versa is analogous to generating complete and accurate UML class diagrams from code and generating code structures from UML class diagrams. The reason UML class diagrams cannot be used to generate complete programs is because they act as a visual summary of a program's static structure. UML class diagrams are at a higher level of abstraction compared to code. On the other hand, a complete program will contain more than enough details required to generate complete and accurate UML class diagrams.

Use case descriptions (analogous to code) contain far more details than use case diagrams (analogous to class diagrams). Use case diagrams are at a higher level of abstraction than the descriptions. Therefore, given a set of use case descriptions, a complete and accurate use case diagram can be systematically produced (see Fig. 8a). However, if modelers choose to create use case diagrams manually first, which is often the case; a 'skeleton' of the use case descriptions can be systematically produced (see Fig. 8b). Detailed descriptions of the use case are later added manually by analysts to 'flesh out' the generated 'skeletons'. After the use case descriptions are complete, an updated version of the use case diagram can be systematically generated. Users of SSUCD and REUCD will not be burdened with performing these transformations since they will be carried out by the tool SAREUCD.

Consistency rules and mapping concepts between use case description structures and use case diagrams are described in detail at [49]. We encourage interested readers to review the referenced document for further details.

5 Tool support using SARUECD

Tool support is essential for the effective application of the REUCD process. For a highly complex software system, the corresponding use case model may contain up to four hundred use cases. Use cases are not sorted in any chronological order. Relationships linking use cases with other use cases and actors are also not sorted in any fashion either. Therefore, performing the REUCD process for such a system manually is a very cumbersome task that is error prone. Even for a relatively smaller use case model, one that contains twenty use cases, the application of the REUCD process is still vulnerable to mistakes.

The tool SAREUCD (Simple Automated REUCD) supports the generation of use case diagrams from use case descriptions and vice versa. In order to generate use case diagrams from use case descriptions, SAREUCD is loaded with a use case description file. SAREUCD parses through the descriptions of all the given use case descriptions and actors and generates a file containing the corresponding use case diagram. The use case diagram is generated in XML in order to be viewable by most UML modeling tools. However, since the format of the generated XML files generated by UML modeling tools vary, the XML files generated by SAREUCD is only viewable by MagicDraw 10.5. Conversely, in order to generate use case description 'skeletons', SAREUCD is loaded with use case diagram file. The use case diagram can be generated by a UML modeling tool. The use case diagram must be in XML format; however this is not an issue since almost all UML modeling tools generate information about their models in XML format. Upon parsing the diagram or description files, the properties of the given use case model is displayed (see Fig. 9).

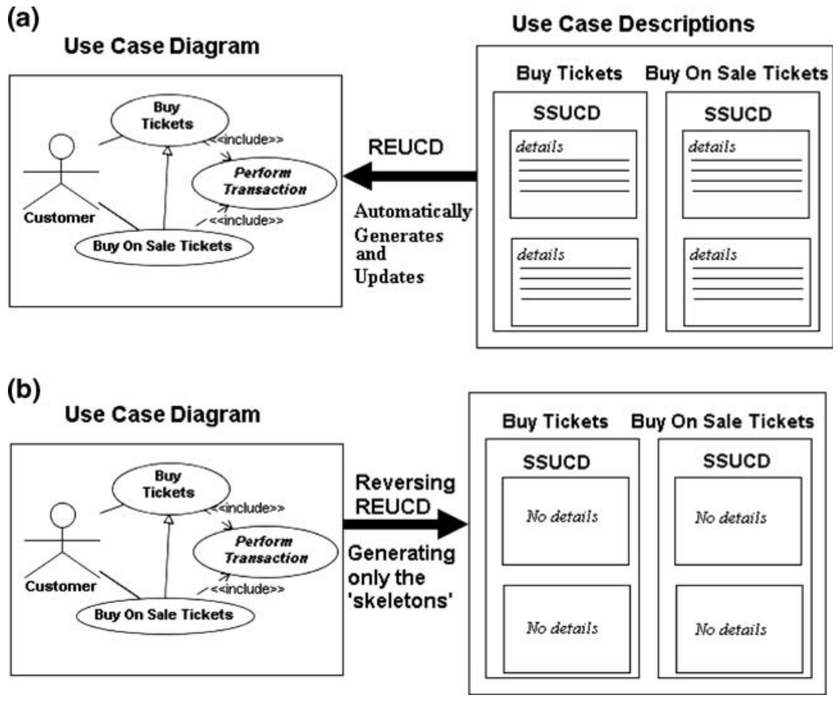
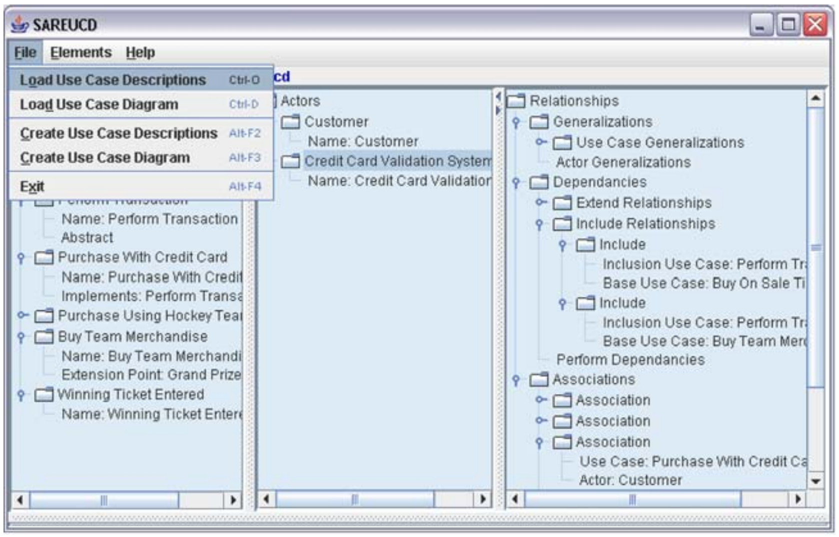


Fig. 8 Systematically generating use case diagrams from descriptions and description skeletons from diagrams


SAREUCD
File Elements Help
Load Use Case Descriptions Ctrl-O
Load Use Case Diagram Ctrl-D
Create Use Case Descriptions Alt-F2
Create Use Case Diagram Alt-F3
Exit Alt-F4

```
Name: Perform Transaction
Abstract
Purchase With Credit Card
  Name: Purchase With Credit
  Implements: Perform Transa
Purchase Using Hockey Tea
Buy Team Merchandise
  Name: Buy Team Merchandi
  Extension Point: Grand Prize
Winning Ticket Entered
  Name: Winning Ticket Entere
```

```
cd
Actors
  Customer
    Name: Customer
  Credit Card Validation System
    Name: Credit Card Validation
```

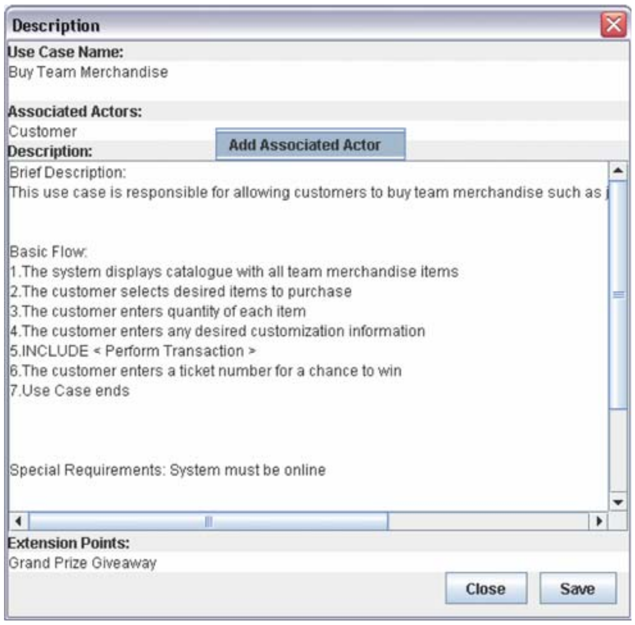
```
Relationships
  Generalizations
    Use Case Generalizations
    Actor Generalizations
  Dependancies
    Extend Relationships
    Include Relationships
      Include
        Inclusion Use Case: Perform Tr
        Base Use Case: Buy On Sale Ti
      Include
        Inclusion Use Case: Perform Tr
        Base Use Case: Buy Team Merc
    Perform Dependancies
  Associations
    Association
    Association
    Association
      Use Case: Purchase With Credit Ca
      Actor: Customer
```

Fig. 9 A screenshot of SAREUCD after transforming the descriptions to an object model

It is unpractical to require or expect use case authors to spend a great deal of time and effort learning the syntax of SSUCD and its consistency and mapping rules, especially since use case authors often have a business background rather than a technical background. Even if the syntax of SSUCD and its mapping rules were understood, creating the use case descriptions and their diagrams manually is error prone. Authors may inject many syntactical and inconsistency errors in the descriptions. It is highly desirable to reduce the time and effort spent learning SSUCD. Therefore, SAREUCD provides a simple GUI interface for creating and editing use case descriptions (see Fig. 10). This saves authors the time to learn many keywords and other syntax rules.

6 SSUCD modeling language design

Languages are designed to achieve a purpose, whether it is to create programs or models. In this paper we presented SSUCD, a modeling language used for analytical purposes. To create a high quality modeling language,



Use Case Name:
Buy Team Merchandise

Associated Actors:
Customer

Description:
Brief Description:
This use case is responsible for allowing customers to buy team merchandise such as

Basic Flow:
1.The system displays catalogue with all team merchandise items
2.The customer selects desired items to purchase
3.The customer enters quantity of each item
4.The customer enters any desired customization information
5.INCLUDE < Perform Transaction >
6.The customer enters a ticket number for a chance to win
7.Use Case ends

Special Requirements: System must be online

Extension Points:
Grand Prize Giveaway

Fig. 10 A use case description

certain quality principles must be considered and embedded into the modeling language. The presence of these quality attributes in any modeling language is essential to its usability and its adoption for widespread use. The literature has provided many guidelines for constructing languages. Paige et al. [51] presented guidelines and quality principles specifically for modeling languages. These quality principles are summarized in Table 1. This section will discuss the design of the SSUCD modeling language and its adherence to these quality principles.

Simplicity

The fundamental purpose of SSUCD is to ensure the consistency between the use case diagrams and their descriptions. All language constructs are designed with this goal in mind. If there are any segments in the use case descriptions that are not reflected in the diagrams, then they are not structured. Instead, they are populated using natural language, which is the original and most flexible method of authoring. The following is a summary of the entire list of constructs provided by SSUCD, sorted by their corresponding section, and how they affect the presentation and the consistency between the descriptions and the diagrams (see Table 2):

All language constructs are aimed towards achieving the ultimate goal of consistency between the descriptions and the diagrams. There are no constructs in SSUCD that do not contribute towards this goal. Furthermore, the grammar indicates that many sections, such as the "Extension Points", can be entirely omitted if not required, which further simplifies the authoring task.

Table 2 A summary of SSUCD's language constructs and their purposes

Section	Keyword	Diagram representation
Use case name	ABSTRACT	*Abstract* use cases appear in italic font in the diagrams
	SPECIALIZES	Results in the creation of generalization relationship links in the diagrams
	IMPLEMENTS	Results in the creation of generalization relationship links in the diagrams
	The name of the use case in natural language	A use case with the given name is shown in the diagram
Description	INCLUDE	The INCLUDE statement can be embedded within the text, and it will result in the creation of an *include* relationship link in the diagram
Extended use cases	Base UC name	Results in the creation of an *extend* relationship link in the diagrams
	Extension point	Optional to the user. The extension point name is displayed on the *extend* relationship link
	IF	Optional to the user. The condition is displayed on the *extend* relationship link
Extension points	The name of a public extension point	Results in the display of an extension point within the oval of the given use case in the diagram

Uniqueness

As shown above in Table 2, there are no overlapping features provided within SSUCD. All language features serve a unique purpose and are vital to ensure consistency, hence there is also no redundant features.

Consistency

Using the list of features provided by SSUCD, use case authors can ensure the consistency between the use case descriptions and the diagrams. This is evident by the ability of the tool SAREUCD to generate use case diagrams from the descriptions and vice versa.

Seamlessness

This quality is intended for design modeling languages that are required to provide an easy and direct transition to code. Hence, this quality is not directly applicable to SSUCD since it is an analytical modeling language that is not intended to show a solution which results in code, but rather to provide an analytical view of what the system is required to do. However, for the purposes of SSUCD, it can be shown from Table 2 that SSUCD constructs can be mapped directly to the notation of use case diagrams. Therefore, for a given set of descriptions, there is no complex computation required to develop their corresponding diagram.

Reversibility

The REUCD process utilizes the SSUCD constructs to systematically generate use case diagrams from the descriptions. Moreover, the REUCD process can also be reversed to systematically produce use case description (skeletons) from use case diagrams. Both these process (forward and reverse) are automated using the tool SAREUCD. Reversibility of SSUCD is discussed in great detail at [49].

Scalability

Use case modeling, in its current form, is used to model very large systems. However, these large models suffer from very poorly written descriptions since they are embedded with numerous inconsistency errors. SSUCD does not impede or hinder the production of use case descriptions. In fact, it provides an interface that guides the author while developing the descriptions. SSUCD, along with SAREUCD, encourages the author to consider aspects that would normally be ignored if the use cases were to be authored in a traditional fashion. For example, if a given *base* use case *includes* another *inclusion* use case, SSUCD requires the authors to consider where exactly in the behavior of the *base* use case will the behavior of the *inclusion* use case be performed. With regards to the additional effort required for creating syntactically correct descriptions, and avoiding the injections of human errors, SAREUCD eliminates this problem in two ways; it guides the authoring process to prevent the injection of errors and it performs all the required syntax checks to notify the user of any existing errors and how to correct them. Therefore, it can be argued that using SSUCD can only help the scalability of use case modeling as a functional requirements elicitation technique. The simplicity of SSUCD allows it be also used to model small systems as well. This is evident by the Online Hockey Store System case study presented in Sect. 7, which can be considered a relatively simple system.

Supportability

Perhaps the most important quality principle; if users do not have the adequate support to be able to use the language, then the language is useless. Users of any language, whether it is a modeling language or a programming language require tool support to help them produce models and programs. Tool support provided by SAREUCD is essential to the usability of SSUCD and the execution of the REUCD process. As mentioned previously, it is unrealistic to expect users of SSUCD of review its E-BNF syntax specifications in order to use it. SAREUCD was designed to perform most of the duties directly related to adopting SSUCD when describing use cases, whereby users need to be only concerned with writing the use cases instead of worrying about adhering to syntax rules.

Reliability

Producing reliable software is the principal objective of SSUCD. SSUCD ensures that the two major components of use case models are consistent. Consequently, understandability of use case models will significantly improve, which is vital to the success of a project that utilizes some form of a use case driven development process. Reliability and the cost of inconsistencies are discussed in great detail in Sects. 2.4–2.6.

Space economy

SSUCD's structural elements exist only within the use case descriptions. Visually, the presence of SSUCD within the textual descriptions is only in the form of a handful of English keywords. Therefore, the size of the use case descriptions in large will visually remain the same whether or not they were structured with SSUCD. Viewing the descriptions with SAREUCD further enhances their readability since SAREUCD hides a large

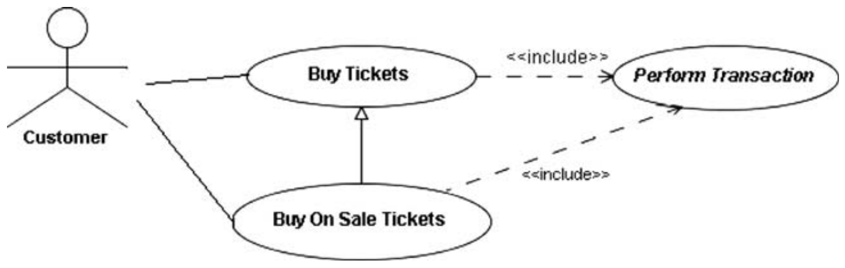


Fig. 11 The use case diagram after three use case descriptions are read

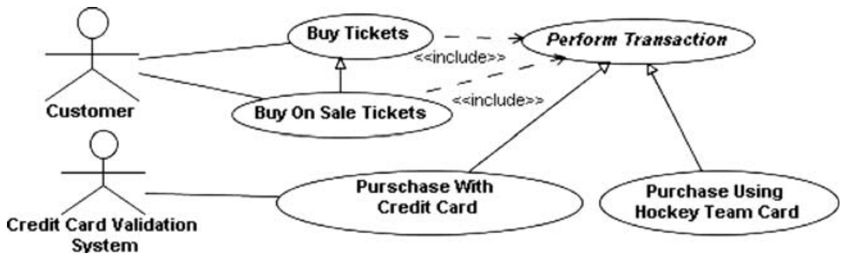

Customer
Buy Tickets
Perform Transaction
<<include>>
<<include>>
Buy On Sale Tickets
Credit Card Validation System
Purschase With Credit Card
Purchase Using Hockey Team Card

Fig. 12 The use case diagram after five use case descriptions are read

subset of SSUCD's keywords and structure to present the descriptions in a more natural form (see Fig. 10).

7 Online hockey team store system case study

The following case study is used to demonstrate how use case descriptions are presented in the SSUCD form and to demonstrate the application of the REUCD process. This case study will also illustrate the concepts, described in Sects. 3 and 4, to systematically generate use case diagrams from use case descriptions using the REUCD process. The case study is about a simplified Online Hockey Team Store system. The presented system is simplified for clarity, yet complex enough for the purposes of demonstrating the SSUCD structure and the REUCD process.

The system allows customers to purchase tickets for upcoming hockey games. To buy a ticket, a customer needs to choose the game he/she would like to attend from the team's online calendar. The customer selects the desired section in an area where he/she would like their tickets to be along with the quantity of tickets requested. Upon retrieval of this information, the system will search the database for the requested tickets. If the tickets are available, the customer is prompted to either accept or reject the offered seats. If the customer accepts the offered seats, the customer is then directed to a billing page where the purchase transaction can take place. Otherwise, if the tickets are not available, the customer is informed about the unavailability and then requested to submit another search for tickets. Occasionally, tickets for certain games in certain sections of the hockey arena may go on sale. Unlike regular priced tickets, a customer may purchase a maximum of six on sale tickets. The system also allows customers to purchase team merchandise such as hockey jerseys, sticks, and pucks. When choosing a merchandise item, the customer may provide customization requests for an extra cost. Available customization options depend on the type of item. For example, if the item was a hockey jersey, the customer may choose to have his/her name sewed on the jersey along with their favorite number. Meanwhile, if the item was a steel pen, the customer may have a name (or other words) engraved on the pen. To boost merchandise sales, a customer may enter a ticket number while purchasing merchandise for a chance to win a grand prize. A customer may purchase tickets and team merchandise using a credit card or a team hockey card. If the customer chooses to purchase using a credit card, an external credit card authorization system is utilized to verify the validity of the given credit card information. Meanwhile, if the customer chooses to purchase using a team hockey card, the customer is requested to enter a PIN. The system internally verifies the PIN with the associated hockey team card to approve the transaction. For any purchase, the customer is requested to enter billing information. The billing information is used for market survey purposes and delivery of tickets and team merchandise. Billing information would include the customer's name, phone number and address.

This simplified system contains seven use cases and two actors. The formal use case and actor descriptions are presented below. For illustrative purposes, the evolution of the use case diagram is shown below (see Figs. 11–13).

Actors:

1) Actor: Customer

Brief Description: This actor may purchase hockey tickets at regular price or on sale. This actor may also purchase team merchandise. The actor will be requested to pay using a credit card or a team hockey card.

2) Actor: Credit Card Validation System

Brief Description: This actor ensures the validity of a given a credit card number and an expiry date.

Use Cases:

1) Use Case Name: Buy Tickets

Associated Actors: Customer

Descritpion:

Preconditions:

At least one game and one seat is available

Brief Description: This use case is responsible for allowing customers to purchase as many tickets as they need in any section.

Basic Flow:

The system presents the different sections that exist in the arena and the price for a single seat in each section. The customer then enters information about the required tickets and submits order request. The system searches for the required tickets and prompts the Customer to accept or reject the offered seats. The customer accepts to purchase ticket and INCLUDE <Perform Transaction> to complete the transaction.

Alternative Flows:

- If tickets not available, the system notifies the Customer that the requested tickets are unavailable and the use case restarts.
- If the tickets were rejected, the system notifies the Customer that the cancellation has been confirmed and the use case restarts

Postconditions: If tickets are issued, these seats become unavailable for future Customers

2) Use Case Name: Buy On Sale Tickets SPECIALIZES Buy Tickets

Associated Actors:

Customer

Description:

Preconditions: At least one game and one seat is available

Brief Description: This use case is responsible for allowing Customers to purchase a maximum of six on sale tickets.

Basic Flow:

The system presents the different sections that exist in the arena and the price for a single seat in each section. The Customer then indicates interest to purchase on sale tickets and submits an order to request tickets. The system retrieves the information about the required tickets and searches for them. The system prompts the Customer to accept or reject the offered seats. The Customer accepts to purchase tickets and INCLUDE <Perform Transaction> is performed to complete the transaction.

Alternative Flows:

- If the tickets not available, the system notifies the Customer that the requested tickets are unavailable and the use case restarts.
- If the Customer requested too many tickets, the system notifies the Customer that the requested number tickets exceed the maximum allowed of six and the use case restarts.
- If the tickets were rejected, the system notifies the Customer that the cancellation has been confirmed and the use case restarts

Postconditions: If tickets are issued, these seats become unavailable for future Customers

3) Use Case Name:

ABSTRACT

Perform Transaction

Brief Description: *This use case is responsible for allowing customers to pay for their selected items*

Preconditions: *At least one ticket is requested for purchase*

Postconditions:

If tickets are issued, these seats become unavailable for future customers

If merchandise is sold, the merchandise database is updated

4) Use Case Name:
Purchase With Credit Card
IMPLEMENTS Perform Transaction

Associated Actors: Customer, credit card validation system
Preconditions: At least one item is requested for purchase
Brief Description: This use case is responsible for allowing Customers to pay for their selected items using a credit card
Basic Flow:
The system requests Customer to enter billing information. The Customer then enters the billing information and selects to pay using a credit card. Upon entering and submitting the credit card information, the given credit card is validated by the Credit Card Validation System and a receipt is printed.
Alternative Flows:

- If the credit card information is incorrect, the system notifies the Customer that the credit card information is incorrect and requests the Customer to enter the credit card information once again.

Postconditions: If tickets are issued, these seats become unavailable for future Customers

5) Use Case Name:
Purchase Using Hockey Team Card
IMPLEMENTS Perform Transaction

Associated Actors: Customer
Preconditions:
At least one item is requested for purchase
Customer has a hockey team card with a set PIN
Brief Description: This use case is responsible for allowing Customers to pay for their selected items using a preauthorized payment plan setup on their hockey team card
Basic Flow:
The system requests Customer to enter the billing information. The Customer then enters the billing information and selects to pay using a hockey team card. The Customer then enters team hockey team card number and PIN. The system verifies the card number and PIN and prints a receipt.
Alternative Flows:
If the card information is invalid the system notifies the Customer that the hockey card information is incorrect and requests the Customer to enter the hockey card information once again.
Postconditions: If tickets are issued, these seats become unavailable for future Customers

6) Use Case Name:
Buy Team Merchandise
Associated Actors:
Customer
Brief Description: This use case is responsible for allowing customers to buy team merchandise such as jersey, hockey sticks, mugs and other collectibles
Basic Flow: The system displays catalogue with all team merchandise items. The Customer then selects the desired items to purchase, the desired quantity and any desired customization information. The INCLUDE <Perform Transaction> use case is performed to complete the transaction. The Customer finally enters a ticket number (if one is available) for a chance to win.
Extension Points:
Grand Prize Giveaway

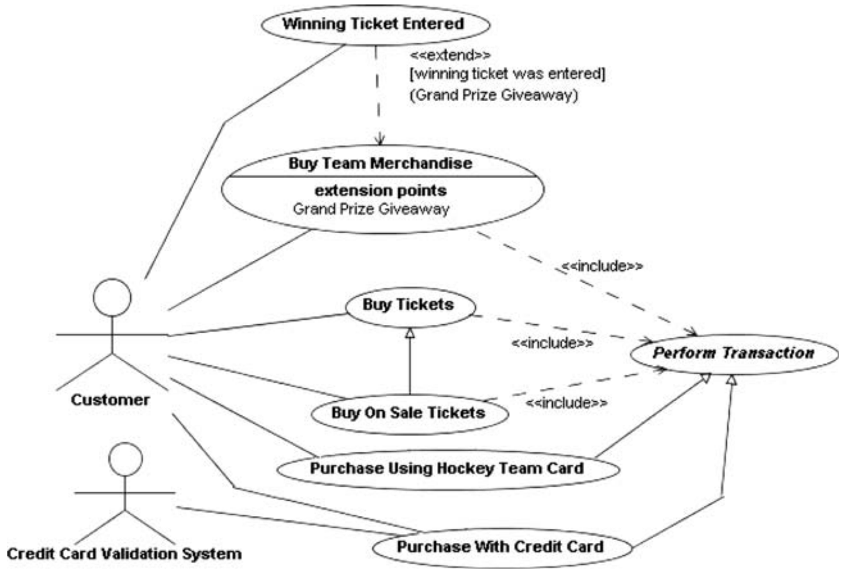


Fig. 13 The use case diagram after all use cases and actors are read

7) Use Case Name: Winning Ticket Entered
Associated Actors: Customer **Extended Use Cases:** Base UC Name: Buy Team Merchandise Extension Point: Grand Prize Giveaway IF the winning ticket was entered **Brief Description:** This use case is responsible for the situation where a winning ticket was entered. **Basic Flow:** If the winning ticket was entered, the system notifies the Customer that they won the grand prize. The Customer enters phone number for a service representative to call

The final use case diagram (Fig. 13) was systematically generated despite the descriptions containing very limited structure. Therefore, as discussed in Sect. 6, the ultimate goal was achieved by providing the minimal amount of structure without adding unnecessary complexities. The use case descriptions file and the XML file representing the diagram may be found at [49].

8 Conclusion and future work

A quality use case model improves every aspect of the development cycle. There are several quality attributes that should exist in every use case model. A use case model needs to be precise and unambiguous so that all stakeholders would have a common understanding of the capabilities and constraints of the system. A use case model needs to be analytical and should not contain any assumptions about the design or implementation. An analytical use case model should only describe what a system should do. Another essential quality attribute is consistency, which is the focal point of this paper. Many researchers and practitioners warn about the harmful consequences of inconsistencies in use case models. Inconsistencies can negatively affect every aspect of the development cycle as well as the stakeholders. Relying on heuristics and experience to manually detect inconsistencies can be cumbersome, error prone and requires a great deal of expertise to be effective. Such expertise is often not readily available. Use case descriptions require necessary structure to allow the automated generation of accurate use case diagrams and the detection of inconsistencies that may exist between them, while allowing analysts the liberty and flexibility to describe a system using natural language, which will make the model easy to comprehend. The added structure will also allow use case descriptions to be machine readable. Consequently, this will allow tools (such as SAREUCD) to systematically extract a great deal of information from the descriptions.

This paper proposes a structure (SSUCD) for describing use cases. The structure serves as a guideline to use case authors. The SSUCD form along with the REUCD process enables the systematic generation of use case diagrams and ensures consistency between the descriptions and their diagrams. The generated diagrams will be complete and provide an accurate representation of the use case descriptions. This process is automated by SAREUCD. The REUCD process may also be reversed, where the use case diagram is constructed before the use case descriptions. In that case, SAREUCD can automatically generate use case description ‘skeletons’ from use case diagrams. Analysts will then need to manually fill in the details of each use case description. After filling the details, SAREUCD can detect any inconsistencies between the diagrams and the descriptions and notify the analysts about these inconsistencies.

Future work can be directed towards developing a semi-systematic approach that will convert SSUCD use cases into SUCD use cases [50]. The SUCD structure contains for more constructs and hence has a greater set of features. SUCD provides analysts with the capability of precisely describing the workflows underlying use cases and automatically generating the corresponding UML Activity Diagrams. Future work can also be directed towards incorporating SAREUCD into a mainstream UML modeling which allows the widespread use of SSUCD and REUCD.

References

1. Adolph, S., Bramble, P.: Patterns for Effective use Cases. Addison-Wesley (2002)
2. Ambler, S.: http://www.agilemodeling.com/essays/when IsAModelAgile.htm
3. Anda, B., Dreiem, H., Sjøberg, D., Jørgensen, M.: Estimating software development effort based on use cases—experiences from industry. Submitted to UML’2001 (Fourth International Conference on the Unified Modeling Language)
4. Anda, B., Sjøberg, D., Jørgensen, M.: Quality and understandability in use case models. In: Lindskov Knudsen, J. (ed.) 15th European Conference Object-Oriented Programming (ECOOP), pp. 402–428. Springer, Budapest (2001)
5. Anda, B., Sjøberg, D.I.K.: Towards an inspection technique for use case models. In: Proceedings of the 14th International Conference on Software engineering and Knowledge Engineering, pp. 127–134 (2002)
6. Anderson, E., Bradley, M., Brinko, R.: Use case and business rules: styles of documenting business rules in use cases. Addendum to the Object-oriented programming, systems, languages, and applications conference (1997)
7. Armour, F., Miller, G.: Advanced Use Case Modeling. Addison-Wesley (2000)
8. Ben Achour, C., Rolland, C., Maiden N.A.M., Souveyet, C.: Guiding use case authoring: results of an empirical study. In: Proceedings IEEE Symposium on Requirements Engineering. IEEE Comput. Soc, Los Alamitos (1999)
9. Biddle, B., Noble, J., Tempero, E.: Essential use cases and responsibility in object-oriented development. In: Proc. of 25th CRPITS, Vol. 24, Issue 1 (2002)
10. Bittner, K., Spence, I.: Use Case Modeling. Addison-Wesley, MA (2002)
11. Boehm, B.: Software Engineering Economics. Prentice-Hall, Englewood Cliffs (1981)
12. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison-Wesley (1999)
13. Butler, G., Xu, L.: Cascaded refactoring for framework evolution. In: Proceedings of 2001 Symposium on Software Reusability, pp. 51–57. ACM Press, (2001)
14. Chandrasekaran, P.: How use case modeling policies have affected the success of various projects (or how to improve use case modeling). Addendum to the 1997 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (1997)
15. Cockburn, A.: Writing Effective Use Cases. Addison-Wesley (2000)
16. Cockburn, A.: Goals and use cases. J. Object-Oriented Program. **10**(5), (1997)
17. Constantine, L.L.: Essential modeling: use cases for user interfaces. ACM Interact. **2**(2), 34–46 (1995)
18. Fagan, M.E.: Design and code inspections to reduce errors in program development. IBM Systems J. **15**(3), 182–211 (1976)
19. Firesmith, D.G.: Use case modeling guidelines. In: Proceedings of Technology of Object-Oriented Languages and Systems – TOOLS 30. IEEE Comput. Soc, Los Alamitos (1999)
20. Gilb, T., Graham, D.: Software Inspection. Addison-Wesley, Reading (1993)
21. Gomaa, H.: Designing Software Product Lines with UML. Addison Wiley Professional (2004)
22. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison Wiley (2000)
23. Harwood, R.J.: Use case formats: Requirements, analysis, and design. J. Object-Oriented Program. **9**(8), 54–57 (1997)
24. Jaaksi, A.: Our Cases with use cases. J. Object-Oriented Program. **10**(9), 58–64 (1998)
25. Jacobson, I. et al.: Object-Oriented Software Engineering. A Use Case Driven Approach. Addison-Wesley (1992)
26. Jacobson, I., Ericsson, M., Jacobson, A.: The Object Advantage. ACM Press (1995)
27. Johansson, A.: Confusion in writing use cases. In: Proc. of the 2nd International Conference on Information Technology for Application (ICITA 2004)
28. Kaner, C., Bach, J., Pettichord, B.: Lessons Learned in Software Testing. Wiley, New York (2003)
29. Kroll, P., Kruchten, P.: The Rational Unified Process made Easy: a Practitioner’s Guide to the RUP. Addison-Wesley (2003)
30. Kruchten, P.: The Rational Unified Process: an Introduction, 2nd edn. Addison Wesley Longman Inc. (1999)
31. Kulak, D., Guiney, E.: Use Cases: Requirements in Context. Addison-Wesley (2000)
32. Larman, C.: Applying UML Patterns: an Introduction to Object-Oriented Analysis and Design and the Unified Process, 2nd edn. Prentice Hall (2001)
33. Leffingwell, D., Widrig, D.: Managing Software Requirements: a Unified Approach. Addison-Wesley (2000)
34. Lilly, S.: Use case pitfalls: top 10 problems from real projects using use cases. In: Proceedings of TOOLS USA ’99. IEEE Computer Society (1999)
35. Mattingly, L., Rao, H.: Writing effective use cases and introducing collaboration cases. J. Object-Oriented Program. **11**(6), 77–79, 81–84, 87 (1998)
36. McCoy, J.: Requirements use case tool (RUT). In: Companion of the 18th Annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 104–105 (2003)
37. OMG. UML 2.0 Infrastructure Specification. http://www.omg.org/docs/ptc/03–09–15.pdf (2003)
38. Overgraad, G., Palmkvist, K.: Use Cases Patterns and Blueprints. Addison-Wesley (2005)
39. Rational Software: Rational Unified Process Version 2002.05.00 (2002)
40. Ren, S., Butler, G., Rui, K., Xu, J., Yu, W., Luo, R.: A prototype tool for use case refactoring. In: Proc. of the 6th Inter-

national Conference on Enterprise Information Systems, pp. 173–178. Porto (2004)
41. Rosenberg, D., Scott, K.: Use Case Driven Object Modeling with UML. Addison-Wesley (1999)
42. Schneider, G., Winters, J.: Applying Use Cases—A Practical Guide. Addison-Wesley (1998)
43. Shull, F., Rus, I., Basili, V.: How perspective-based reading can improve requirements inspections. IEEE Comput. **33**(7), 73–79 (2000)
44. Sommerville, I.: Software Engineering, 5th edn. Addison-Wesley (1996)
45. Wirfs-Brock, R.: Designing Scenarios: Making the Case for a Use Case Framework. Smalltalk Report, Nov.–Dec., 1993. SIGS Publications
46. Wohlin, C., Korner, U.: Software faults: spreading, detection and costs. Softw. Eng. J. **5**(1), 33–42 (1990)
47. Yourdon, E.: Structured Walkthroughs. Prentice-Hall (1989)
48. McBreen, P.: Use Case Inspection List. Last accessed May 2006. http://www.mcbreen.ab.ca/papers/QAUseCases.html
49. STEAM Laboratory website at the University of Alberta: Last updated May 2006. http://www.steam.ualberta.ca/main/research_areas/SSUCD.htm
50. El-Attar, M., Miller, J.: AGADUC: towards a more precise presentation of functional requirement in use case models. In: Proc. 4th ACIS International Conference on Software Engineering, Research, Management & Applications (SERA 2006). Seattle (2006)
51. Paige, R.F., Ostroff, J.S., Brooke, P.J.: Principles for modeling language design. Infor. Softw. Technol. **42**(10), 665–675 (2000)

Author's biography

Mohamed El-Attar is Ph.D. candidate (Software Engineering) at the University of Alberta and a member of the STEAM laboratory. His research interests include Requirements Engineering, in particular with UML and use cases, object-oriented analysis and design, model transformation and empirical studies. Mohamed received a B.Eng. in Computer Systems from Carleton University. Contact him melattar@ece.ualberta.ca

James Miller received the B.Sc. and Ph.D. degrees in Computer Science from the University of Strathclyde, Scotland. During this period, he worked on the ESPRIT project GENEDIS on the production of a real-time stereovision system. Subsequently, he worked at the United Kingdom's National Electronic Research Initiative on Pattern Recognition as a Principal Scientist, before returning to the University of Strathclyde to accept a lectureship, and subsequently a senior lectureship in Computer Science. Initially during this period his research interests were in Computer Vision, and he was a co-investigator on the ESPRIT 2 project VIDIMUS. Since 1993, his research interests have been in Software and Systems Engineering. In 2000, he joined the Department of Electronic and Computer Engineering at the University of Alberta as a full professor and in 2003 became an adjunct professor at the Department of Electrical and Computer Engineering at the University of Calgary. He is the principal investigator in a number of research projects that investigate verification and validation issues of software, embedded and ubiquitous computer systems. He has published over one hundred refereed journal and conference papers on Software and Systems Engineering (see http://www.steam.ualberta.ca for details for recent directions); and currently serves on the program committee for the IEEE International Symposium on Empirical Software Engineering and Measurement; and sits on the editorial board of the Journal of Empirical Software Engineering. Contact him jm@ece.ualberta.ca