Knowledge Transformation from Task Scenarios to View-based Design Diagrams

Nima Dezhkam and Kamran Sartipi
Dept. Computing and Software, McMaster University, Hamilton, ON. L8S 4K1, Canada
{*dezhkan, sartipi*}*@mcmaster.ca*

Abstract

A large body of research in software requirement engineering domain has been dedicated to enhancing the structure of task scenarios using scenario schemas and predefined structures. However, less attention has been paid to the application of schemas in extracting design knowledge from scenarios. In this paper, we propose a schema-based technique to extract the design knowledge embodied in the text of scenarios and represent them using multi-view design diagrams. In this context, we define a framework and a scenario syntax that allow for generating a set of structured scenarios that cover the requirements of a software system. We define a novel scenario schema to parse the informal text of scenarios and populate an objectbase to maintain the design knowledge building blocks. Consequently, a set of guidelines are defined to incrementally build design diagrams for software views such as data and function. As a case study, the design diagram generation for a restaurant system is presented.

KEYWORDS: Knowledge; Transformation; Scenario; Schema; Design; Multiple Views; Object base.

1 Introduction

Scenario-based knowledge extraction from requirements has attracted significant attention within the requirement engineering field [13]. Scenarios are represented in a variety of formal and informal methods ranging from simple text and graphical media to relational algebra [6]. In this paper, we define a scenario as "*a structured narrative text describing a system's requirements in terms of system-environment interactions at business rule level*". Scenarios are considered as easy-to-use and effective means in different phases of software engineering process, such as: requirement elicitation and analysis, design representation, code development, testing, and maintenance [11, 8, 10, 14]. A wide range of research in knowledge extraction from software requirements attempt to investigate: the enhancement of scenario generation by using scenario schemas or pre-defined structures [3, 17]; scenario analysis and knowledge extraction [2]; and design-related document generation [15, 16].

In this paper, we introduce a novel technique to transform the knowledge from scenarios into well-formed design diagrams in two views of data and function. In this technique scenarios are generated using domain knowledge and in conformance with a regular expression syntax that imposes a structure to the scenario representation. The proposed approach allows us to reuse the domain knowledge and business rules within the scenarios through a scenario template knowledge base. Further, the generated structured scenarios are parsed using a novel *scenario schema* to populate an objectbase of design related entities and dependencies. The populated objectbase serves both as a data source during the design diagram construction and as a valuable electronic asset of design knowledge to be analyzed, augmented, and used during the maintenance phase of the software system. Finally, the information in the objectbase is used to create standard diagrams, such as Entity-Relationship diagram (ER) for data view, and function diagram for function view.

The contributions of this paper include: i) a framework to transform the structured knowledge of the scenarios into view-based design diagrams; and ii) a novel schema that allows for decomposing the scenarios into an objectbase of design-related entities and dependencies. As a case study, the design diagram generation for a fast-food restaurant system is presented.

2 Related work

The proposed approach in this paper relates to the literature for capturing and representing knowledge from task scenarios for various purposes. We present several approaches and discuss their similarities and contrasts with our work.

Anton and Potts [1] discuss different representations of scenarios in object oriented software engineering and requirements engineering. Jarke et al. [9] present a review on approaches to scenario-based requirement engineering and research issues.

Lamsweerde et al. [5] introduces KAOS methodology that supports requirements extraction from high-level goals, and assigns objects and operations to the various agents in a system. Their meta-model has similarities with our schema, however our approach aimed at extracting design diagrams after capturing the requirements.

In [6] a formal representation of scenarios using tabular expression is introduced in order to simplify the tasks of scenario validation, verification, and integration. In [3] a schema for semantic model of scenarios is defined to help requirement refinements. Leite et al. [7] aid the process of scenario construction and management by structuring scenarios using a conceptual model along with a form-oriented language. However, in addition to requirement elicitation and validation, our framework transforms the generated structured scenarios into design diagrams. Damas et al. [4] propose tool-supported techniques to generate behavior models from end-user scenarios, whereas we extract design diagrams from scenarios. Hufnagel et al. [16] present a scenario-driven object oriented requirements analysis to support design of a system. This approach does not define a scenario schema and also it is methodology dependent. In [12] a method for modular representation of the scenarios is proposed that supports the reusability of the scenarios in different design contexts. This approach is similar to ours in the sense that it attempts to define a structure for the scenarios.

Overall, the significance of our approach is that we generate scenarios using semi-structured templates and transform the knowledge within the text of scenarios into design relevant knowledge using guidelines that provide a repeatable and view-based design reconstruction process.

3 Proposed framework

In this section, we discuss the steps for transformation of the knowledge embodied in the text of scenarios into design knowledge represented by two views data and function of a software system. These steps are presented using the framework of Figure 1. In a nutshell, the proposed framework generates a set of structured scenarios and uses a schema to parse these scenarios into ingredients of the view-based design representations. The proposed framework consists of three stages, as follows.

3.1 Stage 1: scenario generation

This stage consists of generating a set of structured text-based scenarios that conform with a regular expression syntax. To facilitate scenario generation and controlling the format and vocabulary of the generated scenarios, a predefined set of domain-specific templates can be utilized.

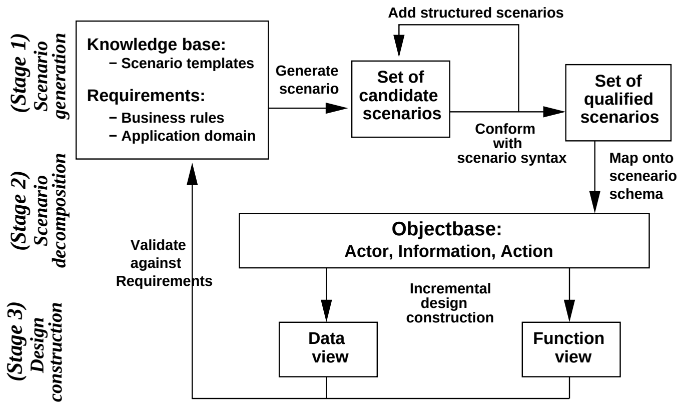

(Stage 1) Scenario generation
(Stage 2) Scenario decomposition
(Stage 3) Design construction
Knowledge base:
– Scenario templates
Requirements:
– Business rules
– Application domain
Generate scenario
Set of candidate scenarios
Add structured scenarios
Conform with scenario syntax
Set of qualified scenarios
Map onto sceneario schema
Objectbase:
Actor, Information, Action
Validate against Requirements
Incremental design construction
Data view
Function view

Figure 1. The proposed design construction framework from scenarios.

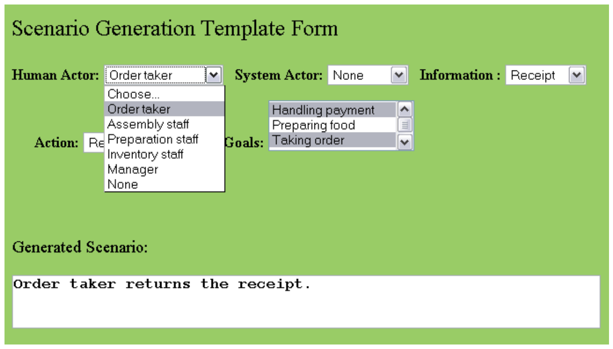
Consequently, at the end of this stage a set of qualified scenarios are produced that cover a part or the whole of the system requirements.

Scenario structure. We define a structure for scenarios that is imposed by the regular expression syntax in Figure 2 and the semantics that are defined by the application domain's business rules. In this scenario syntax, *Actor, Action*, and *WorkingInformation* are the entity-types and action-types that will be defined in Section 3.2. Each scenario consists of a sequence of one or more *Actors*, *Actions*, and *Working Information*, each of which can have between zero or more *Constraints*. In this form we can generate syntactically correct scenarios which will be further decomposed to populate the objectbase in Section 3.2 and generate design diagrams in Section 3.3.

Scenario templates. In order to facilitate generation of structured scenarios and reuse of the captured domain knowledge and vocabulary, the proposed framework leverages a tool to populate a knowledge-base of scenario templates which are organized to store the structured scenarios in a specific application domain. This allows a software engineer to assemble scenarios using a repository of domain-specific vocabulary that is maintained for a software domain. Figure 3 illustrates a sample scenario template form for a fast-food restaurant system. This form consists of fields such as: Actor, Information, and Action, where each field possesses a vocabulary of corresponding business terms. The generated scenario at the bottom of the form is a proper assembly of the terms selected from these fields.

$$Scenario \;:\; \{\mathit{Actor} + \{\mathit{Constraint}\}^{0..N}\}^{1..N} + \{\mathit{Action} + \{\mathit{Constraint}\}^{0..N}\}^{1..N} + \{\mathit{Working\ information} + \{\mathit{Constraint}\}^{0..N}\}^{1..N}$$

Figure 2. Regular expression syntax for scenario generation, where "+" and "0..N" represent composition and range, respectively.


Scenario Generation Template Form

Human Actor: Order taker | System Actor: None | Information : Receipt

Choose...
Order taker
Assembly staff
Preparation staff
Inventory staff
Manager
None

Action: Re

Goals:
Handling payment
Preparing food
Taking order

Generated Scenario:

```
Order taker returns the receipt.
```

Figure 3. Scenario generation template form for a fast-food restaurant system.

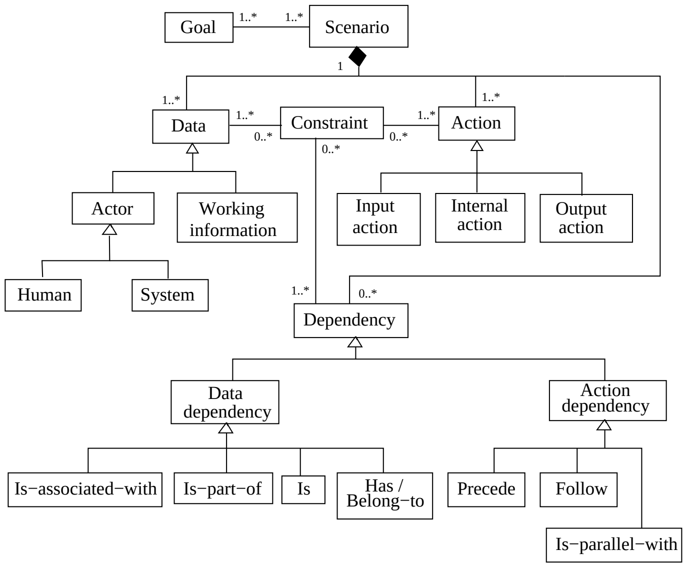


Figure 4. Scenario Schema to parse a scenario and populate an objectbase.

3.2 Stage 2: scenario decomposition

In this stage, the qualified scenarios are mapped onto the proposed scenario schema in Figure 4 which allows us to parse the structured scenarios and generate instances of classes Goal, Actor, Working information, Action, and their corresponding dependencies that are defined in the scenario schema. The generated instances incrementally populate an objectbase of design knowledge that is used to generate design-related diagrammatic representations.

Using the class diagram representation of the scenario schema in Figure 4 the texts of the structured scenarios are parsed and the resulting instances of the classes are stored in the objectbase. The objectbase is represented as a group of columns, where each column stores the instances of a class in the scenario schema that belong to different scenarios. In other words, a scenario (as a row) in the populated objectbase consists of the instances of the relevant classes of the scenario schema that are stored in different columns, and a unique *index* that identifies the scenario.

As shown is Figure 4, in our model every instance of the *Scenario* class is composed of one or more instances of *Actor*, *Working information*, and *Action* classes, and zero or more instances of *Dependency* class. Every Action, Actor, and Working information is associated with zero ore more *Constraint*s. Moreover, every *Scenario* instance is associated with one or more instances of *Goal* class. In the rest of this section the classes of the proposed scenario schema are introduced along with examples from a fast-food restaurant system domain.

Goal: represents the reasons and the desired effects for which the subject system has been produced and used. A goal can be *functional* which corresponds to performing a task, or *objective* which refers to achievement of a quality for the system. Examples of goals in a fast-food restaurant system are as follows: handling payment (functional), preparing food (functional), and shortening order preparation time (objective).

Actor: an actor is a "human" or a "system" or a "component of a system" that interacts with other actors during the execution of the scenarios. Examples of actors in a restaurant system include: order taker (human), raw material supplier (system), or food assembly station (component of a system).

Action: an action is an activity that is performed by an actor during the execution of the scenarios. Generally, an action manipulates an instance of *Working information*. Actions can be categorized into three different types of *Input*, *Internal*, and *Output*, based on the scope of the system. Examples include: taking order (input), computing the price of an order (internal), and delivering food (output).

Working information: refers to the information that is manipulated (exchanged, transported, communicated, operated on, stored, etc.) by the scenario's actor during the execution of the scenario's actions. Examples are: customer's order, raw material, menu item, and item price.

Dependency: refers to a binary relationship between two instances of the classes *Actor, Action*, or *Working information*. When needed, the multiplicity of the participants in a dependency should be mentioned in the dependency instance. In such a case, the dependency can be represented by a quadruple with the multiplicity of each participant proceeding it.

In our schema, a dependency can be of type *Data dependency* or *Action dependency*. Data dependency can be one of the following subtypes: *Is*, e.g., "order taker *Is* an employee"; *Is-associated-with*, e.g., "every menu item *Is-associated-with* a recipe" (or (1,menu item,1, recipe)); *Has*, e.g., "every menu item *Has* a name"; *Belong-to*, that is the inverse[1] of *Has*, e.g., "an ID *Belongs-to* an employee"; *Is-part-of*, e.g., "a kitchen *Is-part-of* a restaurant".

Action dependency can be one of the following subtypes: *Precede*, e.g., "order payment *Precedes* order delivery"; *Follow*, that is the inverse of *Precede*, e.g., "order preparation *Follows* order taking". *Is-parallel-with*, e.g., "sending order to assembly station *Is-parallel-with* sending order to preparation station".

The proposed scenario schema in Figure 4 includes a *Constraint* class that associates any quantifiable constraint to *Data*, *Action*, and *Dependency* classes. Examples of different types of constraints include: *capacity*, *value range*, *ordinal*, *timing*, *privilege*, etc. As an example, a restaurant system may have "*younger than 10*" as a *constraint* associated with *actor* of some scenario, in order to perform a specific *action* such as "offering kids deal".

3.3 Stage 3: design construction

In this section, we discuss the guidelines that transform the contents of the objectbase obtained in Stage 2 into design diagrams. Entity-Relationship (ER) and function diagrams are the most intuitive and relevant diagrams that can be directly extracted from the objects within the objectbase and represent data view and function view of the system, respectively.

[1]For some dependencies, their inverse dependencies are also included in the schema to facilitate back tracing of dependencies in the objectbase.

Data view. The following guidelines specify the generation of ER diagrams from the objectbase:

Data view step I. Extract all instances of *Actor*, *Working information*, and *Data dependency* classes from the object base and apply the following rules on them:

1. Instances of *Actor* and *Working information* are candidate entities/attributes.
2. Instances of *Is* dependency imply generalization and inheritance relationships, i.e., A *Is* B, means A is sub-entity of B, or B is super-entity of A.
3. Instances of *Is-associated-with* dependency imply candidate association relationships.
4. Instances of *Has* and *Belong-to* dependencies are used to identify the attributes of the entities, i.e., A *Has* B (or B *Belongs-to* A) means B is an attribute of entity A.
5. Instances of *Is-part-of* dependency imply candidate decomposition relationships.
6. Candidate entities/attributes that never appear on the right-hand side of a *Has* dependency (or left-hand side of a *Belong to*) dependency are entities and not attributes.
7. Candidate entities/attributes that appear on either side of a *Is*, *Is-associated-with*, or *Is-part-of* relationship are considered as entities.
8. Candidate entities/attributes that appear on the left-hand side of a *Has* dependency (right-hand side of a *Belong to* dependency) are considered as entities.
9. Candidate entities/attributes that appear on the right-hand side of a *Has* dependency (or left-hand side of a *Belong to* dependency) and do not apply in any of the rules vi-viii, are considered as the attributes of the entity on the other side of that dependency.

Data view step II. Depict every entity by a rectangle, every attribute of an entity as a bubble connected to it and label them by their names. Every relationship between two entities can be represented by a line connecting them. Label every relationship according to the type of dependency it came from, e.g., "is", "is-part-of", etc.

Function view. Function view of a system is well represented by function diagrams. The following guideline specifies the generation of function diagrams from the objectbase.

Function view step I. Extract all instances of *Action*, *Action dependency*, and *Constraint* classes from the object base and apply the following rules on them:

1. Instances of *Action* class are the functions.
2. Instances of the *Follow* and *Precede* dependencies determine the time-order of execution of the functions. To simplify the diagram generation, transform all the *Precede* dependencies to *Follow*, i.e., for all functions f_1 and f_2, change $f_1 Precede f_2$ to $f_2 Follow f_1$.

3. The participants of a *Is-parallel-with* dependency must be executed concurrently.
4. The condition(s) for a function to follow another is determined by the *Constraint*s related to the function, actor, and working information in the corresponding scenario that the "following" appears.

Function view step II. Generate $Follow^+$ relationship (the transitive-closure of the *Follow*), i.e., $f_1 Follow^+ f_2$ means there exists a set of functions g_i where, f_1 *Follow* g_1, g_1 *Follow* g_2, ..., g_n *Follow* f_2.

Function view step III. Sort the functions in ascending order based on the number of the functions they follow, i.e., based on the number of times they appear on the left hand side of a *Follow* relationship.

Function view step IV. Start from the beginning of the sorted list, depict the first function (name A) with a square and label it by its name. List all the functions that *Follow* A. If the list contains only one function (name B), depict B and connect A to B with an arrow. If the followers list contains more than one function (name B, C, ...), then a choice condition has occurred. If there are any pair of functions (name B and C) in the list that have an *Is-parallel-with* dependency, connect A to B and C with arrows and an *AND* bubble. Otherwise the functions are connected using an *OR* bubble. Next, all arrows are labeled with the triggering condition(s) obtained in rule "4" above. Finally, remove A from the list and repeat *Function view step IV*, until the list is empty.

The functions in the function view correspond to the actions performed by the actors in the system, and can be considered as candidate methods of classes in the detailed design of the system. Also, the sequence and the AND and OR relationships between the functions reflect the design decisions that should be considered during the implementation phase of the system.

The above guideline can be semi-automated. User involvement is required in cases of conflicts or inconsistencies, such as duplicate usage of actor or action names in different roles, etc. In such cases user can be prompted to perform manual resolution.

The realization of the scenario to design transformation will be presented as a case study in the next section.

4 Case study: Fast-food Restaurant System

In this section, the results of applying the proposed framework to the case of a fast-food restaurant system is presented.

4.1 Stage 1: scenario generation

The following scenarios that conform with the proposed scenario syntax in Figure 2 were generated using our proprietary scenario generation tool. Note that for simplicity in demonstration, the following scenarios demonstrate little interactions and few conditions.

• Scenario #1: *"Order taking station computes and reports the price of the orders."*
• Scenario #2: *"Order taking station sends the paid orders to assembly station."*
• Scenario #3: *"Order taker logs into the OT station using ID and password."*
• Scenario #4: *"Order taker initiates orders."*
• Scenario #5: *"Order taker adds and removes (edit) menu items of an unpaid order."*
• Scenario #6: *"Order taker enters the amount of money received from the customer (cash-in) to OT station."*
• Scenario #7: *"Order taker defers the payment of orders."*
• Scenario #8: *"Order taker reviews the orders."*
• Scenario #9: *"Order taker calls-back unpaid orders."*
• Scenario #10: *"Order taker returns the change (and receipt) for the order."*
• Scenario #11: *"Order taker sends the cash exceeding cash limit to the cash safe."*
• Scenario #12: *"Order taker logs out from his/her ID."*

4.2 Stage 2: scenario decomposition

At this stage, the scenarios were mapped onto the proposed scenario schema to instantiate different class instances and the resulting instances are stored in the objectbase. Table 1 presents a part of the objectbase that is populated with instances of *Data* and *Action* and five *Dependency* classes from Scenarios #1 to #10 above.

4.3 Stage 3: design construction

In this stage we followed the guideline presented in Section 3.3 to construct the diagrams for data and function views.

Data view. Candidate entities/attributes are stored in different *Data* columns (i.e., $Actor|_{System}$, $Actor|_{Human}$, and $Working\ information$) of the objectbase. Similarly, the dependencies among these candidates are stored in the objectbase (under *Is, Belong-to, ...* columns). A part of the the ER diagram for the restaurant system (i.e., order taker component), constructed using the guideline for **Data view** is shown in Figure 5.

Function view. The list of extracted functions sorted by $Follows^+$ relationship is shown in Table 2. Also, the extracted dependencies between these actions are stored in different *Action dependency* columns of the objectbase. The function diagram for the order-taker component of the restaurant system (constructed using the guideline for **Function view**) is shown in Figure 6.

Table 1. A part of the objectbase created from the scenarios #1 to #10.

$Index$	$Actor\|_{System}$	$Actor\|_{Human}$	$Working\ information$	$Action\|_{Input}$	$Action\|_{Internal}$	$Action\|_{Output}$
1	OT Station	-	order,price	-	compute price	report price
2	OT Station, ASM station	-	paid order	-	-	send paid order to ASM station
3	-	order taker,OT station	ID&password	-	login to system	-
4	-	order taker	order	-	initiate order	-
5	-	order taker	menu item,unpaid order	-	add/remove menu item	-
6	-	order taker,OT station	cash-in	enter cash-in	-	-
7	-	order taker	order	-	defer payment	-
8	-	order taker	order	-	review	-
9	-	order taker	unpaid orders	-	call-back	-
10	-	order taker	change/receipt	-	-	return change/receipt

Index	Is-associated-with	Belong-to	Is-part-of	Follow	Precede
1	-	(price,order)	(report price, compute price)	(report price, compute price)	-
2	-	-	(1,paid order,1,order)	(send paid order to ASM station, report price), ...	-
3	-	(ID&password,order taker)	-	-	(login to system, send paid order to ASM station), ...
4	(1,order taker,n,customer order)	-	-	(initiate order, login to system)	(initiate order, compute price)
5	(n,menu item,1,order)	-	-	(edit order, initiate order), ...	(edit Order, compute price), ...
6	-	(cash-in,order)	-	(enter cash-in, report price), ...	(enter cash-in, send paid order to ASM station), ...
7	-	-	-	(defer payment, edit order), ...	-
8	-	-	-	(review orders, login to system)	-
9	-	-	(1,unpaid order,1,order)	(call-back unpaid orders, login to system)	(call-back unpaid orders, enter cash-in), ...
10	-	(change/receipt,order)	-	(return change/receipt, enter cash-in), ...	(return change/receipt, send paid order to ASM station)

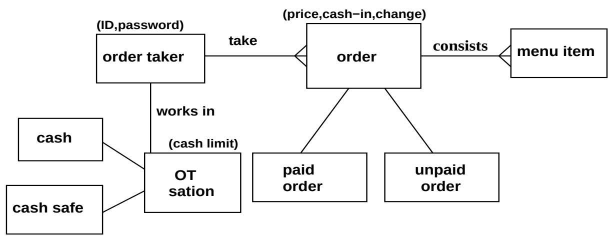


Figure 5. Generated Entity-Relationship diagram for the order taking component.

The generated design diagrams and the existing knowledge in objectbase will enable us to extract other design diagrams such as class diagram of the system. Figure 7 illustrates the complete class diagram of the restaurant system. This diagram is obtained from the ER diagram of the system that was generated in the proposed framework. The space limitation of the paper does not allow us to provide the required guidelines.

5 Discussion and conclusion

In this paper, we presented a systematic and semi-automatic approach for transforming the design knowledge within task scenarios onto a set of design diagrams. We proposed a framework with three major stages of scenario generation, scenario decomposition, and design construction.

Table 2. List of actions in order taking component and corresponding to *Follow* relation.

Index	Action	Follows$^+$
1	Login using ID & password	-
2	Logout the system	1
3	Review orders	1
4	Initiate order	1
5	Call-back unpaid orders	1
6	Edit orders	1,5
7	Compute price	1,5,6
8	Report price	1,5,6,7
9	Defer order payment	1,5,6,7,8
10	Enter cash-in	1,4,5,6,7,8
11	Return change & receipt	1,4,5,6,7,8,10
12	Send order to assembly station	1,5,6,7,8,10,11
13	Send excess cash to cash safe	1,4,5,6,7,8,10,11,12

The task scenarios are structured by the means of a regular expression syntax and can be reused through a knowledge-base of scenario templates. A scenario schema has been proposed as the core of the approach that allows us to decompose scenarios into design entities and dependencies as the means to populate an objectbase. The generated objectbase would maintain the building blocks that allow the

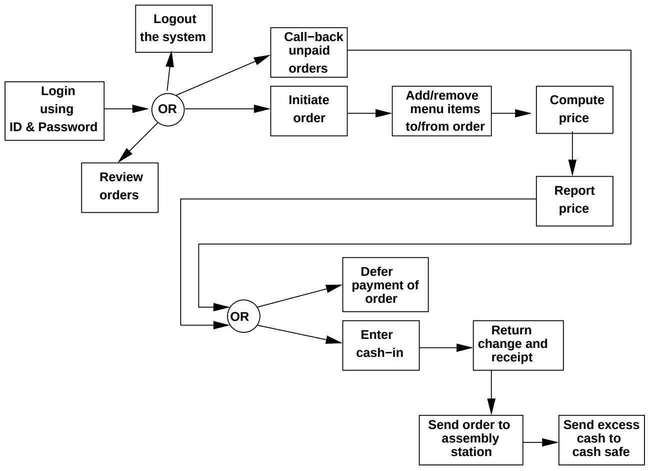

Logout the system
Call−back unpaid orders
Login using ID & Password
OR
Initiate order
Add/remove menu items to/from order
Compute price
Review orders
Report price
Defer payment of order
OR
Enter cash−in
Return change and receipt
Send order to assembly station
Send excess cash to cash safe

Figure 6. Generated function diagram for order taking component.

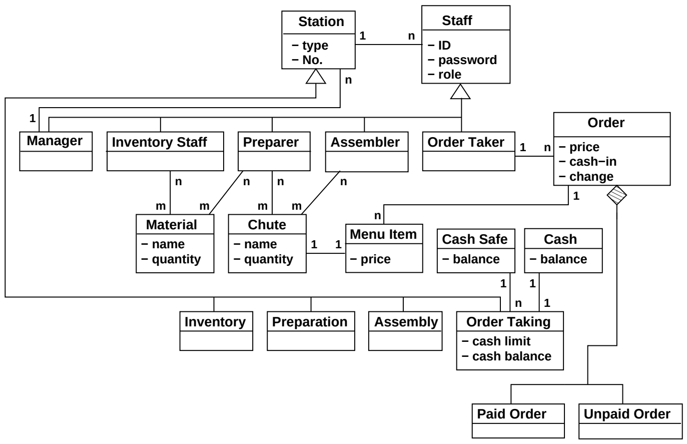


Figure 7. Generated class diagram of the whole restaurant system.

engineer to generate the design diagrams for two views of the software system using common-practice modeling.

We compared the constructed Entity Relationship diagram in Figure 5 with the similar diagram generated for the same restaurant system by a software engineer. In this comparison 6 out of 8 entities for the order taking component were the same in both diagrams which indicates a promising result. The proposed technique provides a disciplined and structured approach to requirement-to-design transformation process within the knowledge engineering field. The proposed scenario schema provides a clear understanding of the major building blocks of the software system's functional entities and their relationships. The populated objectbase serves both as a data source during the design diagram construction and as a valuable electronic asset of design knowledge to be analyzed, augmented, and used during the maintenance phase of the software system. Specifically, the objectbase can be mined to extract more general design decisions that is not feasible by a human-based analysis.

References

[1] A. Anton and C. Potts. Representational framework for scenarios of system use. In *Requirements Engineering*, volume 3, pages 219–241, 1998.

[2] L. Chung and K. Cooper. A knowledge-based cots-aware requirements engineering approach. In *SEKE '02: Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 175–182, New York, NY, USA, 2002. ACM Press.

[3] C.Potts. Scenic: A strategy for inquiry-driven requirements determination. In *Proc. RE'99: International Symposium on Requirements Engineering*, Limerick, Ireland, June, 1999.

[4] C. Damas, B. Lambeau, and P. Dupont. Generating annotated behavior models from end-user scenarios. *IEEE Trans. Softw. Eng.*, 31(12):1056–1073, 2005.

[5] R. Darimont, P. Massonet, and A. Van Lamsweerde. KAOS: An Environment for Goal-Driven Requirements Engineering. *In Proceedings of the ICSE'98*, pages 1–2, 1998.

[6] J. Desharnais, R. Khedri, and A. Mili. Representation, validation and integration of scenarios using tabular expressions. *Journal of Formal Methods in Software Development. Special issue on tabular expressions*, 2002.

[7] J. C. S. do Prado Leite, G. D. S. Hadad, J. H. Doorn, and G. N. Kaplan. A scenario construction process. *Requirements Engineering*, 5(1):38–61, 2000.

[8] Haumer, P. Pohl, and K. Weidenhaupt. Requirements elicitation and validation with real world scenes. In *IEEE Transactions on Software Engineering 24*, pages 1036–1054, 1998.

[9] M. Jarke, T. X. Bui, and J. M. Carroll. Scenario management: An interdisciplinary approach. *Requir. Eng.*, 3(3/4):155–173, 1998.

[10] E. Nasr, L. McDermid, and G. Bernat. Eliciting and specifying requirements with use cases for embedded systems. *In Proceedings of the 7th International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'2)*, pages 350–357, January 2002.

[11] B. A. Nuseibeh and S. M. Easterbrook. Requirements engineering: A roadmap. *In A. C. W. Finkelstein (ed) "The Future of Software Engineering ". (Companion volume to the proceedings of the ICSE'00)*, 2000.

[12] J. Ralyte. Reusing scenario based approaches in requirement engineering methods: Crews method base. *In REP'99*, pages 305–309, 1999.

[13] A. Sutcliffe. Scenario-based requirements engineering. *In Proceedings of the International Conference on Requirements Engineering (RE'03)*, pages 320– 329, 2003.

[14] A. G. Sutcliffe. Scenario-based requirements analysis. *Requirements Engineering Journal*, 3(1), 1998.

[15] Y. E. Tsai, H. C. Jiau, and K.-F. Ssu. Scenario architecture - a methodology to build a global view of oo software system. In *COMPSAC*, pages 446–451, 2003.

[16] W. Wang, S. Hufnagel, P. Hsia, and S. M. Yang. Scenario driven requirements analysis method. In *Proceedings of the Second International Conference on Systems Integration*, pages 446–451, 1992.

[17] H. H. Zhang and A. Ohnishi. A transformation method of scenarios from different viewpoints. In *APSEC 2004*, pages 492–501, 2004.