Use of Tabular Analysis Method to Construct UML Sequence Diagrams

Margaret Hilsbos and Il-Yeol Song
College of Information Science and Technology
Drexel University,
Philadelphia, PA 19104
USA
{mhilsbos, song}@drexel.edu

Abstract. A sequence diagram in UML is used to model interactions among objects that participate in a use case. Developing a sequence diagram is complex; our experience shows that novice developers have significant difficulty. In earlier work, we presented a ten-step heuristic method for developing sequence diagrams. This paper presents a *tabular analysis method* (TAM) which improves on the ten-step heuristic method. TAM analyzes the message requirements of the use case, while documenting the resulting analysis in a tabular format. The resulting table is referenced to build the sequence diagram. This process aids novice modelers by separating the problem analysis from the learning curve of a modeling tool. Building sequence diagrams with the systematic approach of TAM facilitates consistency with the use case model and the class model. We found that developers effectively developed sequence diagrams using TAM.

1 Introduction

A sequence diagram is a type of interaction diagram, which is used in UML to depict a set of messages between objects which participate in a use case [2, 13]. Objects in a sequence diagram are typically instances of classes, and the messages passed between objects invoke operations of the receiving classes [12]. If we accept as axiomatic that the elements of a sequence diagram should be consistent with their corresponding elements in the other diagrams of the system model, then there we need straightforward construction methods which help the modeler achieve this consistency.

While seemingly intuitive, methods for constructing a sequence diagram have not been discussed much in literature. Our experience shows that novice developers have significant trouble in understanding and developing sequence diagrams. Most UML books simply explain the notations and semantics and present pre-built sequence diagrams. Some authors provide simple guidelines for developing sequence diagrams. We found that those simple guidelines are not sufficient for many novice developers. Most research activities on sequence diagrams have focused on real time systems [6, 17], simulation [4] or behavior-driven analysis and design. Very few authors even mentioned possible methods, processes or steps that could be used to develop effective sequence diagrams.

Li [7] proposed using a parser to semi-automatically translate use case steps into "message records" which can be used to construct a sequence diagram. The parser produces a tabular listing of classes, objects, and operations, based on the syntactic structure of each sentence in the use case steps. The modeler can then apply this information to create the sequence diagram. Li's method relies on first "normalizing" the expression of the use case steps to a somewhat rigorous grammatical model. This normalization depends on a modeler's English-language skills. Furthermore, although normalization of the vocabulary and expressions of use cases theoretically may be beneficial, it may not always be feasible in real-world projects because the use case is primarily for communication with users, not for input to a computer program.

Song [18] introduced a heuristic-based approach (Figure 1) to constructing sequence diagrams. The technique instructs the modeler to pull appropriate elements from the prerequisite model artifacts (the use case description and the analysis class diagram), and induces some consistency in the resulting model. Our paper proposes an enhancement to the heuristic approach of [18]. Similarly to [7], our proposed method results in a tabular listing of message records. However, while [7] requires "normalization" of the use case steps, our method proposed here requires only that the use case be clear and unambiguous, and that ultimately agreement between the use case and the sequence diagram can be confirmed.

The enhanced method, referred to as the *tabular analysis method* (TAM), consists essentially of reordering the steps of Song's ten-step heuristic (referred to as the "original heuristic"), sequentially applying the reordered steps to each action in the use case activity flow, and documenting the resulting analysis in a tabular format. The tabular data can then be used as a reference to build the diagram relatively quickly in a modeling tool. Some advantages of the tabular method over the original heuristic are:

- The step-by-step process of the TAM is defined in more detail, which should be helpful to novice modelers to conceptualize and visualize the building process.
- With the TAM, a more comprehensive model is relatively easy to construct: the tabular format presents the modeler with all elements to be considered for an operation in an easy-to-read format, which encourages thoroughness in modeling parameters, constraints, etc. – resulting in a more semantically complete model with minimal added effort.
- Tool-independence: the tabular data could be exported to an XMI file (see [10]) and then uploaded into the modeling tool of choice, to create the model elements. A further conversion of the XMI file to an SVG file (see [11]) could fully automate the diagram creation from the table. While the theorized automation capability has not yet been developed, if this potential is realized, the tabular method presented here will have significant added value.

By using the method proposed in this paper, the modeler can express the analysis in a more commonly familiar tool (a word processor or spreadsheet program), and then reference the analysis worksheet while learning how to construct the elements in the

CASE tool. In other words, this method separates two analytic processes so that each can be more fully attended to by students or novice modelers.

1 Select the initiating actor and initiating event from the use case description.

2 Identify the primary display screen needed for implementing the use case. Call it the *Primary boundary object*.

3 Create a *use-case controller* (*primary control object*) to handle communication between the primary boundary object and domain objects.

4 If the use case involves any included or extended use case, create one *secondary control object* for each of them.

5 Identify the number of major screens necessary to implement the use case. Create one *secondary boundary object* for each of the major screens and create one *secondary control object* for each of them.

6 From the class diagram, list all *domain* classes participating in the use case by reviewing the use case description. If any class identified from the use case description does not exist in the class diagram, add it to the class diagram.

7 Use those classes just identified as block labels (Column names) in the sequence diagram. List classes in the following order: (1) The primary boundary stereotype, (2) The primary use case controller, (3) Domain classes (list in the order of access), and (4) Secondary control objects and secondary boundary objects in the order of access

8 Identify all problem-solving operations based on the following classifications:

8.1	Instance creation and destruction	8.2	Association forming
8.3	Attribute modification:		
8.3.1	Calculation;	8.3.2	Change States
8.3.3	Display or reporting requirements	8.3.4	Interface with external objects or systems

These problem-solving operations can be identified by:

- Identify verbs from the use case description
- Remove verbs used to *describe* the problem; select verbs used to solve the problem. We call these verbs *problem-solving verbs(PSVs)*.
- From the problem-solving verbs, select verbs that represent an automatic operation. We call these PSVs *problem-solving operations(PSOs)* and use them in the sequence diagram.

9 Rearrange the sequence of messages among the object classes based on any pre-existing design patterns, when possible.

10 Name each message and supply it with optional parameters. This can be done at design stage as well.

Fig. 1. The Ten-Step heuristics [18]

The rest of this paper is organized as follows. Section 2 discusses issues related to the consistency among UML elements that need to be maintained for the accurate sequence diagrams. Section 3 presents our TAM for constructing sequence diagrams with examples, and Section 4 concludes the paper.

2 Model Consistency

Sequence diagrams share model elements with use cases and class diagrams. In this section, we discuss consistency issues among use case models, class diagrams, and sequence diagrams.

2.1 Consistency with the Use Case Model

A sequence diagram represents the design for fulfilling the requirements expressed in a use case [5]. The use case elements which should be reflected in the sequence diagram are postconditions, actions, and related use cases (included, extending, and specialized). The postconditions in the use case description specify what state of the system must be true upon successful completion of the use case [5]. If the sequence diagram depicts all the behavior required for successful completion of the use case, it follows that each postcondition specified in the use case description must be achieved by some message in the set of sequence diagrams for that use case. Conversely, if the use case postconditions accurately define the system state, it follows that the use case description should identify as postconditions all final states resulting from execution of the use case behavior detailed by the sequence diagram.

Each action specified or implied in the use case description should be detailed in a corresponding message or set of messages in the sequence diagram. Depending on the clarity and completeness of the use case description text, the author of the sequence diagram may need to infer some of the operations. Song [18] presents several categories for identifying operations: instance creation and destruction, association forming, attribute modification, calculation, change of states, display or reporting, and interface with external objects or systems. Each action in the use case description will require one or more of these message types for its fulfillment.

2.2 Consistency with the Class Diagram

From the literature review, we have identified several areas where consistency between class diagrams and sequence diagrams can be easily confirmed: classes, operations, arguments, visibility between objects, and composition responsibility.

Classes. All entity classes used in a sequence diagram must appear in the Class Diagram. Conversely, if sequence diagrams are completed for <u>all</u> use cases within the project scope, all entity classes shown on the design class diagram must be used in at least one sequence diagram, with the exception of some or all abstract classes. Abstract classes *may* be shown in certain cases [9, 15]; but generally the receiver of a message is a concrete class - the lowest class in its hierarchy to which all instances addressed by the message could belong [15].

Operations. For an object to handle a message that it receives, it must have a conforming interface, which is defined in the receiver's class as an operation signature [12, 8]. Therefore, all messages shown on the sequence diagram must map to operations of the

receiving class in the class diagram. A temporary message name may be assigned before the class operation has been designed.

Arguments. A sequence diagram message may transfer information to the receiver as arguments. Arguments must represent information that is known to the sender, such as attribute values or constants. Depending on the intended precision of the model, the sequence diagram may not show all the relevant arguments [3]. However, some parameters should always be shown, such as an object or parameter that is being passed among multiple other objects [3]. Some practitioners choose not to show all (or even any) return messages [12]. Pender argues that it is worth the effort to model operations and returns completely, to avoid ambiguity [12].

Visibility (Relationships between Classes). In order for objects to exchange messages, the sending object must have a handle to the receiving object [15]. Another way of saying this is that the sender must have *visibility* to the receiver. Some authors state or imply that a message between two objects in a sequence diagram requires a permanent association (association, generalization, or aggregation) to be shown between the classes in the class diagram [1]. Others note that there are *four types of visibility* possible between objects – attribute visibility, parameter visibility, local visibility, and global visibility [5, 14] – and that only attribute visibility requires a permanent association [16]. Messages which rely on parameter, local, or global visibility to a class require a *temporary*, or *transient*, association between the classes [16]. A transient association is modeled on the class diagram as a *dependency* instead of an association, with an arrow depicting the direction of the dependency [5, 14]. To summarize, consistency between the class diagram and the sequence diagram requires that for each message in the sequence diagram, the class diagram depicts either a permanent association or a dependency, according to the type of visibility required, between the classes of the sender and receiver. Conversely, if an association depicted on the class diagram is never used in an interaction, then there must be an error in the model [1].

Composition Responsibility. If the class diagram depicts a whole-part (composition, or strict aggregation) relationship between two classes, then the whole should create the part [5, 14]. In the sequence diagram, this is depicted by first creating the composite (probably with a create() message from a controller), then using the composite as the sender for the create() message to the part.

3 Constructing Accurate Sequence Diagrams With TAM

In this section, we present our tabular analysis method (TAM) for constructing accurate sequence diagrams in a manner that enforces consistency with the use case, while promoting consistency with the class diagram as well. The TAM uses the ten-step heuristic introduced in [18] as a starting point, and applies it methodically to each step in the use case description. The TAM takes the procedures expounded in [18] as follows:

- A system sequence diagram (SSD) is constructed first, treating the system as a "black box" and modeling only the actions visible to the actor. These actions are called system events [5].
- Each system event may be documented in one detailed sequence diagram; or details for multiple system events may be combined in one sequence diagram.
- A separate sequence diagram will be constructed for each included or extending use case.
- A separate controller is used for the base use case, and each included or extending use case, and functions as a "connector" between the sequence diagrams.
- Actors communicate with boundary objects, which communicate with controllers, which communicate with the entity objects. Normally, actors do not communicate directly with controller or entity objects.

The TAM uses a tabular format called Sequence Analysis Table (SAT) which captures the list of use case actions by adding columns for source and receiving objects, message names and parameters. Figure 2 shows a condensed version of the empty template. The table used here can also be thought of as a condensed version of the Tabular Notation described in the UML 2.0 specification [13]. A Sequence Analysis Table is created for the primary use case and each included or extending use case. The overall process of the TAM can be summarized as follows:

- From the use case description, create a *system sequence analysis table (SSAT)* to create a system sequence diagram. Note that each line here is an input system event from an actor to a system or an output from the system to an actor.
- Expand each line of SSAT in such a way that each system event or output can be broken into multiple messages that can be represented in a sequence diagram. Into detailed sequence analysis table.
- Each included or expanding use case description results in a different detailed SAT.
- Create a sequence diagram from the detailed sequence analysis table.

Step #	Use case action	Message Name	Parameters	Constraints	Sender	Receiver

Fig. 2. The Template for Sequence Analysis Table (SAT)

3.1 Getting Started – Designing the System Sequence Diagram (SSD)

In this section, we discuss how to create a system sequence diagram in the TAM. The first step is to copy the actions from the use case description document (UCD) to the empty template. Each input from an actor to the system (called a system event) and an output from the system to the actor forms a row in SAT. Next, for each row, enter a short

message name that describes the primary communication. Depending on the quality of the UCD steps, some editing may be required at this point.

The next step is to identify sending and receiving objects as the initiating actor and a boundary object representing the system user interface. Evaluate each subsequent action as "from the actor to the system" or "from the system to the actor". For "system" put "BO"(meaning a boundary object that represents the system being modeled) in the "Sender" or "Receiver" column. It is not necessary to name boundary objects yet.

An example Use Case Description of the use case "Process Rents" for a Video Rental System (VRS) case study is shown in Figure 3 (only the Main Success Scenario is shown). The resulting System Sequence Analysis Table is shown in Figure 4; and the resulting SSD diagram is shown in Figure 5. Refer to [18] for the problem statement, the use case diagram, and the class diagram of the VRS case study.

Actor Action	**System Action**
1. Staff starts "New Rental" mode on POS, if POS is not already in the correct mode.	
2.The staff verifies the customer's status by their ID card or number.	3. System finds and displays the customer's information.
	4. INCLUDE Get Overdue Fees. Any overdue items are displayed with tape information, due date, and late fee amount due.
5. The staff records new rental items by scanning the bar codes.	6. System determines rental price and due date and displays with title, date and price for each item.
7. On completion of last item entry, the staff indicates to the POS terminal that the rental is complete.	8. System calculates tax and total rental fee, and records with date out. Late fees determined in prior step are added to the balance due.
9. Staff accepts payment from customer for the balance due.	10. INCLUDE Pay Fees.
	11. IF payment is successful, on-hand inventory is reduced by one for each rented item, and receipt is printed.
12. Staff hands items and receipt to customer and concludes transaction.	

Fig. 3. The Main Success Scenario of Use Case "Process Rents" of VRS Use Case Description

3.2 Defining the Sequence Diagram (SQD) Details

In this section, we show how to build the detailed sequence diagram in the TAM. At this point, the template contains a single line for each system event between the Actor and the System, as shown in Figure 4. Based on the ten-step heuristic, this section shows how to decompose the single interaction into multiple messages at the detailed level, as shown in Figure 6.

A detailed sequence diagram must show the interactions *within* the system, between various objects, as well as the parameters and constraints relevant to the messages. The

following steps describe a methodical approach to completing the table. *In the resulting table, there will be one line for each message shown on the sequence diagram.* That means, in completing the detailed information, it will be necessary to *insert lines in the table* wherever multiple messages are required to implement a use case step – which will be true for *almost all* of the use case steps. The steps described below are illustrated in Figure 6 through Figure 8 for the "Enter Customer" system event of the VRS use case.

S#	Use case action	Message Name	Parameters	Constraints	Sender	Receiver
1	Staff starts "New Rental" mode on POS, if POS is not already in the correct mode.	start_rental			Staff	BO
2	Staff verifies the customer's status by their ID card or number.	enter_customer			Staff	BO
3	System finds and displays the customer's information.	display_customer			BO	Staff
4	INCLUDE Get Overdue Fees. Any overdue items are displayed with tape information, due date, and late fee amount due.	display_overdue			BO	Staff
5	The staff records new rental items by scanning the bar codes.	enter_rental_items		*[until end_rental received]	Staff	BO
6	System determines rental price and due date and displays with title, date and price for each item.	display_item_data		[enter_rental_items received]	BO	Staff
7	On completion of last item entry, the staff indicates to the POS terminal that the rental is complete.	end_rental			Staff	BO
8	System calculates tax and total rental fee, and records with date out. Late fees determined in prior step are added to the balance due.	display_total_due			BO	Staff
9	Staff accepts payment from customer for the balance due.	n/a			Customer	Staff
10	INCLUDE Pay Fee.	enter_payment			Actor: Staff	BO
11	IF payment is successful, on-hand inventory is reduced by one for each rented item, and receipt is printed.	print_receipt		payment is successful	BO	Staff
12	Staff hands items and receipt to customer and concludes transaction.	n/a		payment is successful	Staff	Customer

Fig. 4: Resulting System Sequence Analysis Table (SSAT) for System Sequence Diagram of the VRS Example

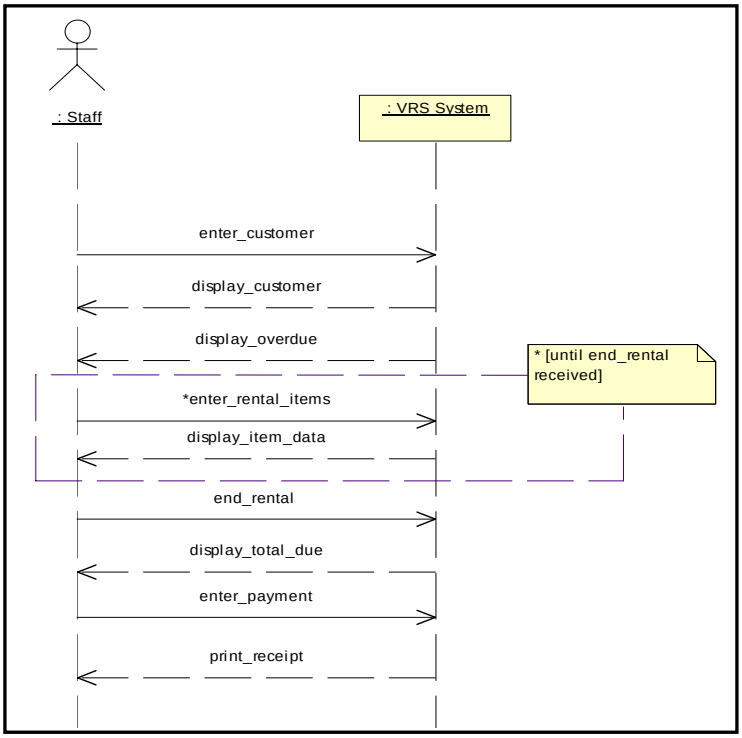

Fig. 5. Resulting System Sequence Diagram (SSD) for use case "Process Rents"

Specify details for initial lines.

1. Name the Primary Boundary Object, and replace all instances of “BO” in the table with the selected name.
   - In VRS example: RentalWindow.
   - This is analogous to step 2 of the original heuristic.
2. Add a use case controller (CO) to the *Receiver* column for each Included or Extending Use Case. The detailed steps for each included or extending use case will be documented in a separate Sequence Analysis Table.
   - In VRS example, the controllers are named: for “INCLUDE Get Overdue Fees” - AccountHandler; for “INCLUDE Pay Fees” – PaymentHandler.
   - This is analogous to step 4 of the original heuristic.
3. Identify constraints: everywhere a qualifying word such as “if” appears in a use case step, there should be a constraint for one or more messages.

- Initially, we suggest to address only the "Main Success Scenario" of the use case. However, to *fully complete* the set of sequence diagrams for the use case, the modeler must verify that all alternatives are documented. For example, in the VRS case there is a qualifier "If payment is successful" in the main success scenario. The alternative path – payment is not successful – is addressed in the "Other Successful Scenarios" and "Unsuccessful Scenarios" of the use case description. Initially, and for this paper, the alternative paths will not be developed.

Expand the table by adding lines for required messages - decompose each use case action as follows:

4. Add the lines for communication between the primary boundary object to and from the appropriate controller. This incorporates step 3 of the original heuristic.
5. Identify the problem-solving operations as described in [18] Heuristic step 8.
6. Identify the message parameters (data) and the classes to which the corresponding attributes belong. This step corresponds to step 6 of the original heuristic.
   - If the sequence diagram is intended to be completed exhaustively and precisely, *all* parameters should be determined, by consulting the analysis class diagram for entity classes and attributes which satisfy the semantics of the use case step.
   - If a less exhaustive documentation is sufficient, specify only the most important parameters. Alternatively, a group of attributes may be named, as in "customer_data" rather than listing customer name, address, etcetera. Always identify *all entity classes* which will be involved in each use case step.
   - If the need for additional entity classes is discovered, add the new classes to the class diagram, with the required attributes.
   - If the need for additional attributes for an entity class is discovered, add the new attributes to the class in the class diagram.
7. For each entity class identified in step 6, insert a line in the table, between the lines just added for communication with controllers. Add the *object* representing the entity class in the *Receiver* column, using instance notation (e.g., :Customer) if an instance is appropriate (almost always the case – if in doubt, assume an instance is required). The receivers should be listed in the order they will be addressed.
8. For messages which are procedure calls, the *Sender* for each new line is typically the *Receiver* of the prior line. Thus the *Sender* in the first inserted line is the controller of the use case step. However, this rule will not always result in the correct allocation of responsibility. See [5] for discussion of allocating responsibility to objects; in proceeding through the following steps, the modeler should re-arrange the Senders and Receivers as necessary to assign responsibilities correctly.
9. Name the messages and distribute the parameters among the lines pertaining to the step. This step corresponds to step 10 of the original heuristic.
   - Some parameters to be displayed must be calculated. Insert a line for each calculation; identify the attributes needed for the calculation and list as parameters.

Step #	Use case action	Message Name	Parameters	Constraints	Sender	Receiver
1	Staff starts "New Rental" mode on POS, if POS is not already in the correct mode.	start_rental			Actor: Staff	Rental Window
1.1		start_rental			Rental Window	Rental Handler
1.2		request_cust_id			Rental Handler	Rental Window
2	Staff verifies the customer's status by their ID card or number.	enter_customer	cust_ID		Actor: Staff	Rental Window
2.1		get_cust	cust_ID		Rental Window	Rental Handler
2.2		get_cust	cust_ID		Rental Handler	Customer
2.3			customer_ data		Customer	Rental Handler
2.4			customer_ data		Rental andler	Rental Window
3	System finds and displays the customer's information.	display_customer	cust_ID		Rental indow	Actor: Staff

Fig. 6. Expansion of use case step 2 of Figure 4 for communication with controllers.

- Verify that every parameter to be displayed is either retrieved from a class or calculated based on attributes retrieved from a class, and that all of the classes involved are listed.
- Indicate iteration of a message with a * (example: *get_item_info).
- Be sure to identify and add to the table, any messages with the same *Sender* and *Receiver* (such as calculations).
- Figure 7 shows how the steps to this point have been applied for the first two use case steps of the table. The expansion of use case step 2 to add communication with controllers and the entity class Customer is outlined in bold.

10. Review the constraints originally entered for the use case steps, and copy these as necessary into the new lines for the added messages.

Throughout the analysis, identify any clarifications needed, or missing or implied steps in the use case. Insert rows and add steps as needed; highlight changes to the use case steps in order to go back and update the UCD for consistency later. If it is apparent that additional major screens will be required, create a secondary boundary object and secondary controller for each major screen that is needed in addition to the main screen for the use case. Insert lines in the table as needed for passing messages from the primary controller to each secondary boundary object.

Step #	Use case action	Message Name	Parameters	Constraints	Sender	Receiver
1	Staff starts "New Rental" mode on POS, if POS is not already in the correct mode.	start_rental			: Staff	: Rental Window
1.1		start_rental			: Rental Window	: Rental Handler
		create_rental			: Rental Handler	: Rental
1.2		request_cust_id			: Rental Handler	: Rental Window
2	Staff verifies the customer's status by their ID card or number.	enter_customer	cust_ID		: Staff	: Rental Window
2.1		get_cust	cust_ID		: Rental Window	: Rental Handler
2.2		get_cust	cust_ID		: Rental Handler	: Rental
2.3		get_cust	cust_ID		: Rental	: Customer
2.4			customer_data		: Customer	: Rental
2.5			customer_data		: Rental	: Rental Handler
2.6			customer_data		: Rental Handler	: Rental Window
3	System finds and displays the customer's information.					
4	INCLUDE Get Overdue Fees. Any overdue items are displayed with tape information, due date, and late fee amount due.	get_overdue	cust_ID		: Rental Window	: Account Handler
4.1		display_overdue	tapeInfo, dueDate, lateFeeDue		: Account Handler	: Rental Window
4.2		display_customer	customer_data, overdue_data		: Rental Window	: Staff

Fig. 7. Detailed Sequence Analysis Table for *Enter Customer* system event

At this point, the table is complete for the *main success scenario of the primary use case*. Similar tables should be constructed for the included and extending use cases. If it is desired to create a generic sequence diagrams which includes all scenarios, messages and qualifications can be added as necessary by inserting rows in the table, to include the information covered in the "Other Successful Scenarios" and "Unsuccessful Scenarios" sections of the UCD.

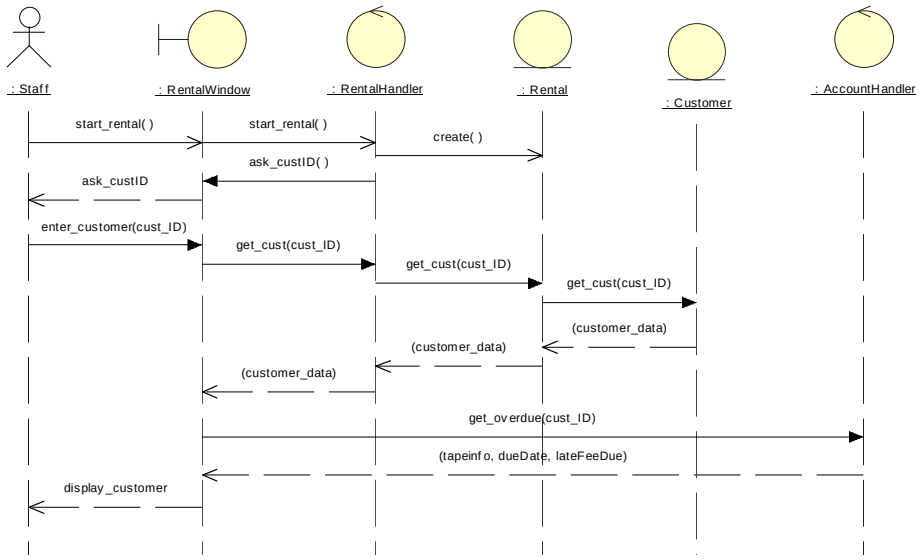

Fig. 8. Sequence Diagram for "Enter Customer" system event

Once satisfied that the table represents all data required for the sequence diagram, the modeler can create the diagram by referring to the table. Since the classes have already been modeled, the modeler merely drags the required classes into the sequence diagram. The classes should be ordered in the diagram as proposed in step 7 of the original heuristic. Then the message names, parameters, and qualifications can be copied and pasted from the table into operations in the modeling tool. The messages should be shown on the diagram in the correct relative time order. As mentioned in the Introduction, we envision a future capability to export a table developed with this method into an XMI file which can then be imported into any XMI-compatible tool, thus further simplifying the diagram creation.

4 CONCLUSION

This paper has proposed the tabular analysis method (TAM) for constructing accurate sequence diagrams.. The proposed method is rigorous and, applied as envisioned, results in a thorough modeling of operation elements. As such it may be considered tedious, but has the advantage of separating model analysis from the vagaries of tool usage. Therefore, it may prove ideal for application in the following circumstances:

- learning environments;
- situations where there is a need for comprehensive sequence diagrams; and

- situations where there may be multiple modeling tools in use, or tool selection is not complete at the time modeling needs to begin.

In addition, the tabular format used in this method is anticipated to be adaptable to automated model interchange in accordance with the OMG specifications for XML Metadata Interchange [10] and UML 2.0 Diagram Interchange [11]. Successful development of conversion scripts to realize this automation will enhance the applicability of the tabular approach, to the point where even experienced modelers may find it useful for quickly documenting interaction sequences.

References

1. Ambler, S. W. (2002) "UML Modeling Style Guidelines" retrieved 8/24/2003 from http://www.modelingstyle.info/classDiagram.html
2. Ambler, S. W. (2003) "UML 2 Sequence Diagram Overview" retrieved 8/24/2003 from http://www.agilemodeling.com/artifacts/sequenceDiagram.htm
3. Chonoles, M. J.; Schardt, J.A. (2003). *UML 2 for Dummies*. Wiley.
4. Kabajunga, C and Pooley, R.. Simulating UML sequence diagrams. *UK Performance Engineering Workshop*, UK PEW 1998, pages 198—207, July 1998.
5. Larman, C. (2002) *Applyng UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.* 2nd ed., Prentice Hall PTR.
6. Li, X. and Lilius, J. Timing Analysis of UML Sequence Diagrams. *UML'99, The Unified Modeling Language. Beyond the Standard*. October 28-30, 1999.
7. Li, L. (2000). "Translating Use Cases to Sequence Diagrams", Automated Software Engineering, 2000. Proceedings ASE 2000. 11-15 Sept. 2000. pp. 293 – 296.
8. Litvak, B., Tyszberowicz, S. and Yehudai, A. (2003) "Behavioral Consistency Validation of UML diagrams", In *Proceedings of the First International Conference on Software Engineering and Formal Methods (SEFM'03)*.
9. Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall.
10. Object Management Group. (May 2003). "XML Metadata Interchange (XMI) Specification Version 2.0" retrieved 03/17/2004 from http://www.omg.org/cgi-bin/doc?formal/2003-05-02.
11. Object Management Group.. "UML 2.0 Diagram Interchange draft adopted specification." Retrieved 03/17/2004 from http://www.omg.org/cgi-bin/apps/do_doc?ptc/03-07-03.pdf
12. Pender, T. (2003). *UML Bible*. Wiley.
13. Object Management Group. (August 2003). "UML 2.0 Superstructure Final Adopted specification." Retrieved 9/12/2003 from http://www.omg.org/cgi-bin/doc?ptc/2003-08-02 .
14. Oestereich, B. (2002). *Developing Software with UML: Object-Oriented Analysis and Design in Practice.* Addison-Wesley.
15.. Page-Jones, M. (2000). *Fundamentals of Object-Oriented Design in UML*. Addison-Wesley.
16. Rumbaugh, J., Jacobson, I., and Booch, G. (1999). *The Unified Modeling Language ReferenceManual.* Addison-Wesley.
17. Seemann, J. and Wvg, J. (1998). Extension of UML Sequence Diagrams for Real-Time Systems, In Proc. *International UML Workshop*, Mulhouse, June 1988.
18. Song, I. (2001). "Developing Sequence Diagrams in UML." In *Proc. of 20th International Conference on Conceptual Modeling (ER2001)*, Yokohama, Japan, 2001, pp. 368-382.