A Multi-level Methodology for Developing UML Sequence Diagrams

Il-Yeol Song, Ritu Khare, Yuan An, and Margaret Hilsbos

The iSchool at Drexel, Drexel University,
3141, Chestnut Street, Philadelphia, PA 19104, USA
songiy@drexel.edu, rk84@drexel.edu, yan@ischool.drexel.edu,
mhilsbos@drexel.edu

Abstract. Although the importance of UML Sequence Diagrams is well recognized by the object-oriented community, they remain a very difficult UML artifact to develop. In this paper we present a multi-level methodology to develop UML Sequence Diagrams. Our methodology is significant in three aspects. First, it provides a multilevel procedure to facilitate ease of the development process. Second, it makes use of certain patterns to ensure the validity of SQDs. Third, it uses consistency checks with corresponding use-case and class diagrams. Throughout the steps of the method we present rules and patterns demonstrating correct and incorrect diagramming of common situations through examples. The purpose of this study is to serve as a reference guide for novice sequence diagram modelers. This methodology is particularly useful for novice practitioners who face challenges in learning the process of SQD development.

1 Introduction

Sequence Diagrams (SQDs) are one of the important dynamic modeling techniques in the UML. An SQD visualizes interactions among objects involved in a use case scenario. Although several methods have been proposed to develop an SQD, the development of SQDs remains a very difficult part of the object oriented development process. The development process is very intricate. As new objects and messages are identified, the diagram gets more packed and complicated. Also, at every step, multiple factors, such as which object to choose, which message to assign to what object, and what patterns to use for message passing, need to be taken care of simultaneously. As a result, the modeler very often ends up making mistakes in the diagram, and making an SQD which is inconsistent with other UML artifacts. Hence, we are motivated to develop an easy-to-use and practical method for SQD development.

In our earlier work, we presented a ten-step method (Song, 2001) for developing SQDs based on use case descriptions and a class diagram. In this paper, we extend our earlier work as follows: First, we re-organize the steps into three levels and each level is further divided into several stages so that we can focus on one issue at a time. Second, we add guidelines and patterns using correct and incorrect examples. Third, we provide consistency checks between an SQD and use case and class diagrams. This method brings forth the recommended visual patterns and warns against mistakes committed by SQD developers. The purpose of this study is to serve as a reference

Q. Li et al. (Eds.): ER 2008, LNCS 5231, pp. 114–127, 2008.
© Springer-Verlag Berlin Heidelberg 2008

guide for novice SQD modelers. In this paper, we use UML 2.0 notation to present SQDs. For the notations of SQDs in UML 2, we refer Ambler (2008b) or Larman (2004).

The rest of the paper is organized as follows. Section 2 presents the research methods used and the related literature review. Section 3 describes the process of multi-level methodology to develop SQDs. Section 4 concludes our paper.

2 Research Setting and Related Literature Review

In this paper, we have come up with both correct and incorrect patterns of SQDs, as well as guidelines. The guidelines were tested and examples were collected for the past five years of teaching SQDs in a graduate class. Incorrect patterns have been found on the basis of our observation of the mistakes students make in SQD assignments to develop SQDs. Our subjects include students from different backgrounds including computer science, information science, psychology, biosciences, biomedical, and business. Students found our guidelines usable and effective.

There is a decent amount of research related to specification of semantics of the SQD. Xia and Kane (2003) present an attribute grammar based approach on defining the semantics of UML SQD to make it easily convertible to code. While both this paper and the work by Aredo (2000) prepare a framework for defining the semantics of an SQD to create a shared understanding across a design team, the problem of designing an SQD still remains unresolved especially for novices.

Baker et al. (2005) address the problem of automatically detecting and resolving the semantic errors that occur in SQD scenarios. The method proposed in this paper is claimed to be successful in detecting semantic inconsistencies in industrial case studies. Our method, however, takes a preventive action to deal with the semantic inconsistencies by basing itself on the commonly occurring valid patterns in SQD and avoiding the frequently committed mistakes by novices.

Li (2000) presents a parser that semi-automatically translates use case steps into message records that can be used to construct a sequence diagram. The work is based on syntactic structure of standardized sentences of use-case description. Although this work provides useful rules for novices, e.g. converting a use case to “message sends”, it does not avoid common mistakes made by novices.

Other recent works on SQDs include use of SQDs for code generation, generation of SQDs through reverse engineering of code, and finding reusable SQDs from existing artifacts. Rountev and Connell (2005), and Merdes and Dorsch (2006) present reverse engineering techniques to extract an SQD from a program. Another interesting work ‘REUSER’ by Robinson and Woo (2004) automatically retrieves reusable SQDs from UML artifacts.

Our review shows that the research effort for developing an easy-to-use method for developing SQDs is still rare and far from satisfactory. Hence, the task of creating this artifact remains challenging to novices, and they continue to commit errors. We propose a multi-level methodology in order to develop the artifact in an incremental manner. We refer to the work by Bist, MacKinnon, and Murphy (2004) for guidelines in drawing SQDs.

3 A Multi Level SQD Development

An Overview

In this section we describe our proposed methodology to develop an SQD. The most significant portion of this work is that it offers a multi-level way to develop SQDs. Instead of considering multiple design issues at each step of SQD development, we propose focusing on one major issue at each level of the process to make best possible use of knowledge at every level. We begin designing in terms of the objects first which form the building blocks of an SQD. After the objects are well arranged, responsibilities are assigned to them in the next level. In the last level, the visual pattern of the SQD obtained from previous level is analyzed to make further modifications and produce the final version of the SQD. Furthermore, each level is divided into stages to focus on just one sub-issue at each stage, and to further simplify the overall development process. A multi-level development process also offers the following in a systematic manner:

- Maintenance of consistency with other UML artifacts (use-case and class diagrams).
- Correctness of the SQD by making use of certain rules and visual patterns.
- Warnings to stay away from frequently committed mistakes in drawing SQDs.

In this paper, we deal with the UML use-case descriptions that represent one main success scenario and zero or more included use cases. It is important to maintain consistency between the SQDs and other diagrams of the system model. The SQD depends on classes identified in the domain model. Therefore, before beginning to construct an SQD, the use cases should have been identified and use case descriptions generated for the use cases assigned to the current development iteration, and an analysis class diagram (domain model) constructed.

Fig. 1 shows the three-level development process. Table 1 summarizes the whole methodology. The three abstract levels are: the object framework level, the responsibility assignment level, and the visual pattern level. Each level comprises multiple stages.

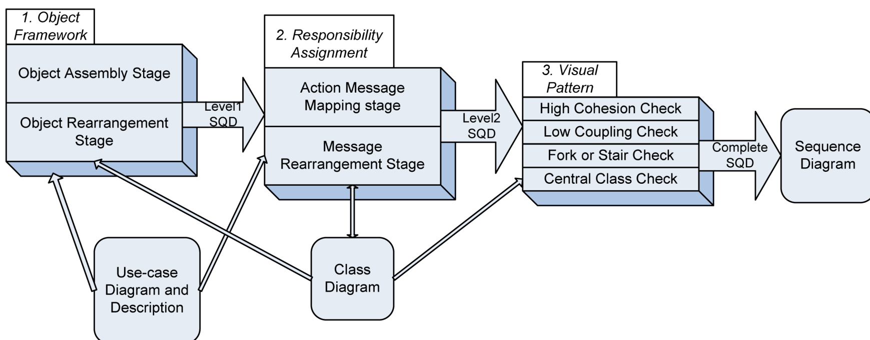

Fig. 1. A Component Diagram of Multi-level SQD Development

Table 1. The Steps of Multi-level SQD Development

(BO: Boundary object; CO: Control Object; EO: Entity Object)

1. Object Framework Level*:* Identify the building participants which constitute the basic framework of an SQD.
1.1 ***Object Assembly Stage*****:** Identify the actor, primary BO, primary CO, secondary CO(s), secondary BO(s) for the SQD.
1.2 ***Object Rearrangement Stage*****:** Rearrange the classes (and also the actor) in the following order: Actor, Primary BO, Primary CO, EOs (list in the order of access), and Secondary COs and Secondary BOs in the order of access.

2. Responsibility Assignment Level*:* Assign correct responsibilities to each object.
2.1 ***Action-Message Mapping Stage***: Map every *automated* action in the use-case description to a message in the SQD. Each message would fall under one of the following categories: Instance creation and destruction, Association forming, Attribute modification, Attribute access, Input/Output/Interface, and Integrity-constraint checking.
2.2 ***Message Re-arrangement Stage***: Perform arrangement checks such as: making sure that each message is traceable to the primary actor through activated objects, giving meaningful names to each message, checking consistency of SQD with class diagram, removing any unnecessary return messages, and checking for continuity of focus of control.

3. Visual Pattern Level*:* Apply final checks based on the visual patterns illustrated by the SQD.
3.1 ***High Cohesion Check*****:** Make sure that the responsibilities assigned to a class are related, and there exists a high cohesion within a class.
3.2 ***Low Coupling Check*****:** Re-arrange messages from one class to another class to reduce coupling.
3.3 ***Fork or Stair Check*****:** Choose between the "fork" and the "stair" pattern depending on the relationship between classes, and taking into account the pros and cons of both patterns.
3.4 ***Central Class Check*****:** It should be kept in mind that the class, which looks central in the class diagram, is likely to send most messages to other classes in the SQD.

3.1 The Object Framework Level

In this level, we identify the building blocks that constitute the framework of an SQD.

3.1.1 Object Assembly Stage: Following Are the Steps to Be Followed in This Stage

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 p*rimary boundary object*.
3. Create a *primary control object* to handle communication between the primary boundary object and domain objects. It is not always necessary to have a *control object (CO)* between the *boundary object (BO)* and the *entity object (EO)*. A *BO* can directly pass message to an *EO*, if the message is simple and requires no manipulation.
4. If the use case involves any included or extended use case, create one *secondary CO* for each of them. UML 2.0 introduces specific notation for connecting sequence diagrams. The following method has previously been suggested, and may be simpler to apply, at least until modeling tools "catch up" to the UML 2 notation.

As shown in Fig. 2, use a separate CO for the supporting use case; show the supporting CO on the base use case SQD, with messages to and from indicating the flow of control (Song, 2001).

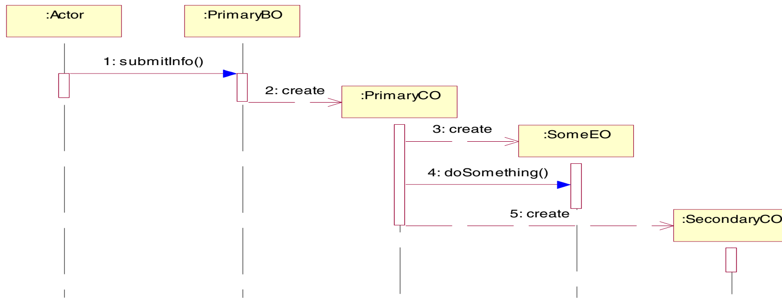

Fig. 2. The use of a Control Object for an Inclusion Use Case

5. Identify the number of major screens necessary to implement the use case. The following cases represent the situations that require creation of a new secondary BO*:*
   - A new window needs to be opened for user's input and the contents of the original window need to be kept visible
   - A new window only handles a sub-flow and the original window may proceed with the sequence regardless of the operations in the secondary window.

   Also, create a secondary CO 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. These classes become the EOs.

3.1.2 Object Rearrangement Stage

Use the classes just identified as participant names in the SQD. In a logical sequence of actions, tasks begin with an actor interacting with an interface (BO). The BO then passes control to a CO that has resources to carry the required actions, which then passes control to relevant EOs, and so on. Hence, list the actor and the classes in the following order: Actor, Primary BO, Primary CO, EOs (list in the order of access), Secondary BOs, and Secondary COs in the order of access.

3.2 Responsibility Assignment Level

Responsibility assignment refers to the determination of which class should implement a message, and which class should send the message. It is important to assign the correct set of responsibilities to each object because they become operations of corresponding objects in the design stage UML artifacts such as design-class diagrams. A message in an SQD is assigned to the class at the target of the message. For example, the message 4 *doSomething()* will be implemented as an operation in class

:SomeEO in Fig. 2. There are several guidelines that can be followed when assigning responsibility to classes. In this paper we follow the GRASP (General Responsibility Assignment Software Patterns) guidelines described by Larman (2004).

3.2.1 Action-Message Mapping Stage

Each action specified or implied in the use case description should have a corresponding message(s) in the SQD. Depending on the degree of completeness of the use-case description text, the author of the SQD may need to infer some of the operations. The messages are identified through the following procedure:

- Identify verbs from the use-case description.
- Remove verbs that describe the problem. Select verbs that solve the problem and call them *problem-solving verbs* (PSVs).
- From the PSVs, select the verbs that represent an *automatic* operation or a manual operation by the actor. We call these PSVs *problem-solving operations* (PSOs) and use them as messages in the SQD.

Larman (2004) uses three types of postconditions: *Instance creation and destruction*, *Association forming*, and *Attribute modification*. In this paper, we treat them as PSO categories. Here, we add three more PSO categories: *Attribute access, Input/Output/Interface,* and *Integrity-constraint checking*. *We use these six PSO categories to identify messages from a use case description. These six types of PSOs can also be used in identifying messages that are necessary but not explicit in the use case descriptions.*

A. Instance Creation and Destruction: The "Creator" pattern suggests rules for determining which object should send an object creation message (Larman, 2004). Class B should have the responsibility to send *create()* message to A in the following cases: B aggregates A objects; B records instances of A objects; B closely uses A objects; or B has the initializing data that will be passed to A when it is created. Often, the controller will have the initializing data, but an entity class will be assigned the responsibility when it is closely associated with the new object as in the first four cases. The UML 2 notation suggests that a created object should be placed at the creation height in the diagram, which Ambler (2008b) refers to as "direct creation".

Fig. 3 shows the correct SQD with the direct creation of object :SomeEO.

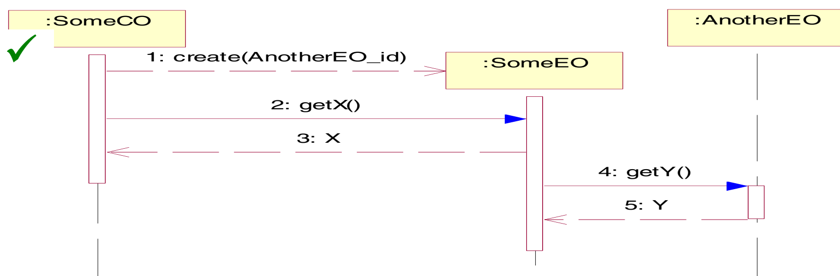

Fig. 3. Correct Object Creation at the Creation Height

B. Association Forming: If there is an association between two classes, then at least one of the SQDs must include a message that forms this association. If a depicted association is never supposed to be used at all, then there must be an error either in the class diagram or the SQD (Ambler, 2008b). Associations can be formed by creating the object with the appropriate parameters or by updating the appropriate parameter in the object. The association must be formed before other operations, which require visibility from the sender to the receiver, can be performed. Fig. 3 shows an example where the association is formed between :SomeEO and :AnotherEO by the parameter (AnotherEO_id) being passed to :SomeEO at creation. This makes the *getX()* message possible.

C. Attribute Modification (set/compute/convert): *For each postcondition that causes a state change, there should be a message. The messages change the value of attributes such as deposit_amount(), calc_subscription_charge(),* and *convert_cm_to_inch(). Any message that sets a value, computes a value, or converts one unit to another belongs to this message type.*

D. Attribute Access (get/find/compare/sort): This type of message reads values of attributes. Any message that gets a value, finds a value, compares values, and sorts values belongs to this message type.

A frequent mistake of novice developers is to try to update an attribute of a read-only class in the use case. We call such a class as a *reference class,* which refers to an entity class that just provides information to a use case, and that should not be updated by any interaction. Fig. 4 illustrates a case where the modeler did not understand the roles of the class :PricingPolicy. Fig. 4 is a portion of a sequence diagram submitted by a student for use case called "Add Paid Subscription" in a subscription automation system. This example is incorrect because :PricingPolicy is a class that stores the pricing rules. :PricingPolicy may be updated in a maintenance transaction, but not by a customer transaction of adding a new subscription. A tip-off is the class name. Any class name including "policy", "rule", or "template" is probably a reference class for any interaction except the use case to specifically update that class. This example also demonstrates the value of clear class names. Fig. 5 shows a correct depiction of the same interaction.

***E. Input/Output/Interface*:** This type of messages is used (a) to input data, (b) to display output, generate report, or to save a data to a storage, and (c) to interface with external objects or systems.

Interaction with an external system is shown by a message from a CO to the BO of the external system. Some messages to be included in an SQD aren't mentioned anywhere in the use-case description; a designer of an SQD needs to make decisions regarding these messages. Entire communication between a BO and a CO is based on the designer's judgment. The GRASP "Controller" pattern stipulates that, for interactions requiring any manipulation or coordination, actor inputs are transferred from the interface (BO) to a CO. Fig. 6 is an example of an incorrect use of an entity class to send a message that should be sent by the controller. Fig. 7 shows the corrected version, sending a message from a CO to an external system.

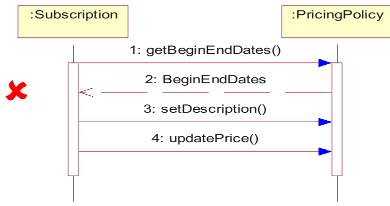

Fig. 4. Incorrect, updating a *reference* class

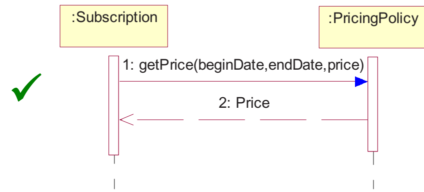

Fig. 5. Correct, getting information from a *reference* class

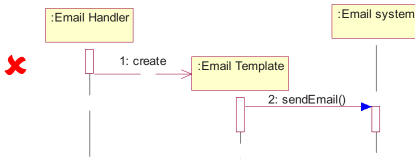

Fig. 6. Incorrect- Entity communicating with an external system

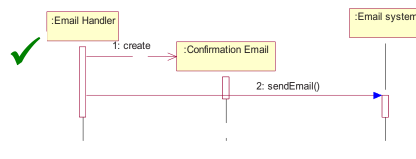

Fig. 7. Correct – Controller communicating with an external system

A common BO pattern is as follows. An actor creates a BO and enters some data. The BO creates a necessary CO and transfers data to it. After the CO completes whatever processing it is responsible for (which may include calling other COs), the CO returns some value to the calling BO. The BO displays some information for the actor. Fig. 8 shows an incorrect message sequence between an actor and the BO, and Fig. 9 shows an example of the correct use of a BO.

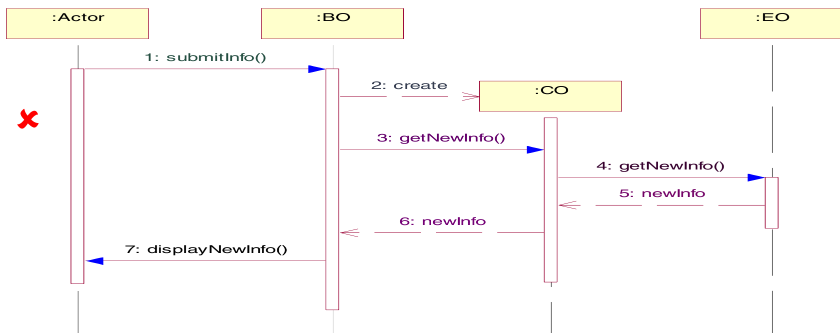

Fig. 8. An incorrect sequence of messages between a user and a window

It should be noted that a BO may access an EO directly, but this is only appropriate when the interaction is very simple, e.g. a retrieval of values from a single class, or an update to a single class with no calculations. Fig. 10 shows an example of this pattern.

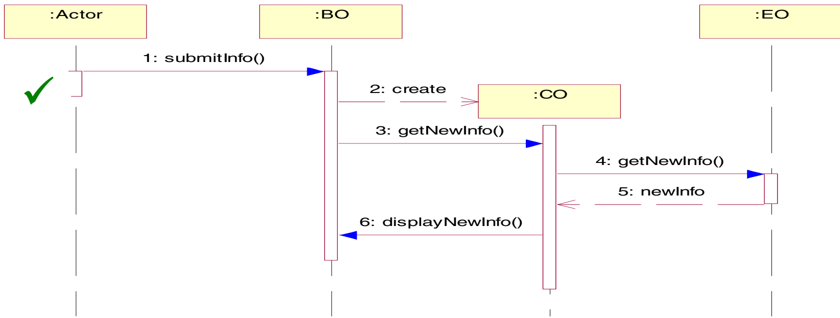

Fig. 9. A correct Sequence of messages

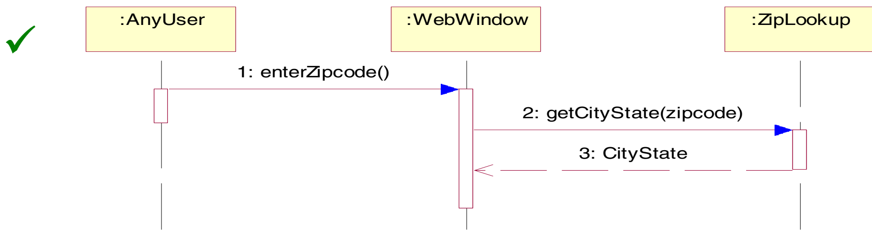

Fig. 10. An example of valid direct communication between BO and EO

F. Integrity-constraint (IC) Checking**:** Another message type in an SQD is an IC checking operation. Checking a complex integrity constraint usually requires passing of multiple messages among objects. Examples include validating a user input or computing a discount amount based on the current order and customer credit rating.

3.2.2 Message Rearrangement Stage

After the Action-Message mapping stage, an SQD is generated; but it still requires manipulation and re-arrangement of messages among objects. Perform the following to rearrange the messages in the SQD.

1. *Make sure that each message is traceable to the primary actor through activated objects.* The actor interacts with a BO. The BO transfers information to and from other objects via COs. At no time can an object initiate a message without first being activated by another object which is already activated, except for a BO which is activated by a message from the actor (Pooly and Steven, 1999). The exception to this is active objects, which are beyond the scope of this paper. Fig. 11 is an example of an invalid SQD, where the :PaymentHandler initiates a message without first being activated. Fig. 12 shows a correct version of the same diagram, where the :PaymentHandler is first activated by the :SubscriptionHandler.

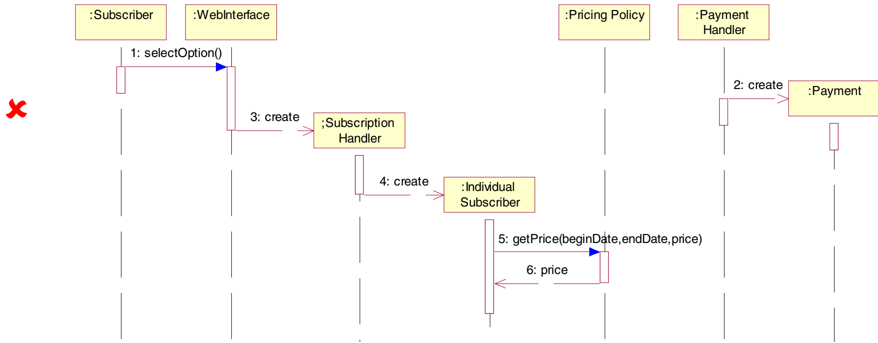

Fig. 11. Incorrect – Payment Handler was never activated

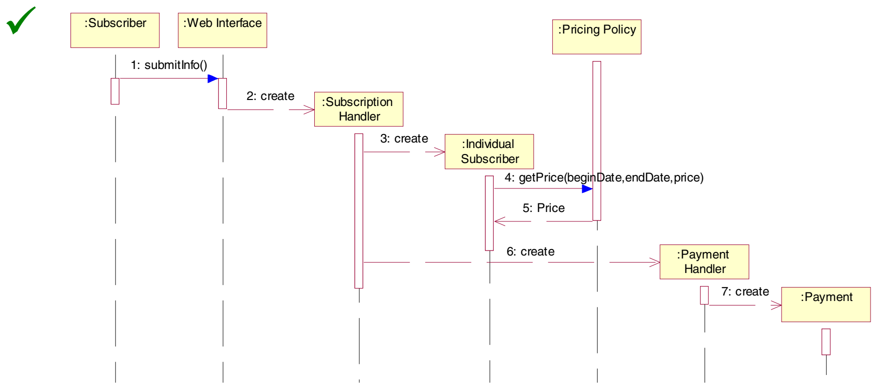

Fig. 12. Correct – Payment Handler is created

2. *Name each message with meaningful names.* Message names should clearly communicate what is being requested. For example, if a message to :Client is getting the email address, the message *getEmailAddress()* is more descriptive than *getEmail().*

Supply each message with optional parameters. The SQD will not necessarily show all the relevant attributes as message arguments (Chonoles and Schardt, 2003). Some parameters, however, should be shown, such as an object or parameter that is being passed among multiple other objects. (Ambler 2008b; and Chonoles and Schardt, 2003). The items to verify with respect to message arguments are:

- Each depicted or implied argument represents either an input value or an attribute of some class or a class in the class diagram. Specified parameters which represent attributes or classes should match their depiction in the class diagram.
- The sender of the message containing the argument has visibility to the value or attribute(s) used in the arguments.

3 Check the SQD for consistency with the Class Diagram. All entity classes used in an SQD must appear in the class diagram. Conversely, if SQDs are completed for all use cases within the project scope, all entity classes shown on the design class diagram must be used in at least one SQD, with the following caveats:

- This is not necessarily true for abstract classes.
- In many projects, SQDs will not be generated for the entire set of use cases. In that case, the modeler should mentally verify that any remaining concrete entity classes will be used by the yet-to-be-modeled SQDs.

4. *Check if return messages are implied or required*. It is not always necessary to show returns on SQDs (Arlow and Neustadt, 2002; Larman, 2004). Returns should be shown only when showing them makes the drawing more understandable (Ambler, 2008b; and Fowler 2000). Some rules that help to make this determination are:

- When a message implies the return, such as *getPrice(),* it is not necessary to show the return.
- When complex processing results in a new value that is returned to a calling routine, the return should be shown.
- Ambler(2008b) suggests "If you need to refer to a return value elsewhere in your SQD, typically as a parameter passed in another message, then indicate the return value on your diagram".
- Returns usually point from right to left, but not always; messages normally point from left to right, but not always. Therefore it is important to use the correct notation for clarity.

5. *Check for the correctness of focus of control*. The focus of control is also referred to as a method activation box (OMG 2003) or method-invocation box (Ambler, 2008a). The focus of control shows the time during which the object is active, or has control of the interaction. If an object receives a message (message no. 2 in Fig. 13) that needs to return a value to a calling class (:PaymentWindow), the focus of control for the calling class should be continuous as the object is just waiting for a response. The focus of control should remain active till a return message (message no. 3) is received from :PaymentHandler. Fig. 13 shows incorrect focus of control and a wrong return notation. A corresponding correct diagram is not shown due to limited space.

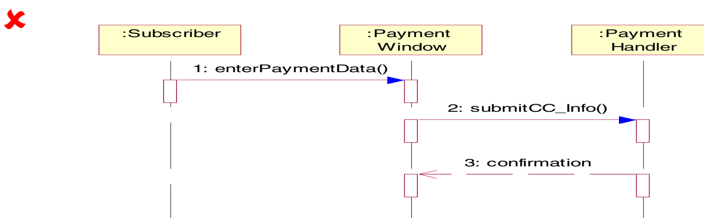

Fig. 13. Incorrect – broken focus of control; returns shown incorrectly

3.3 Visual Pattern Level

3.3.1 High Cohesion Check

"High Cohesion" stipulates that the responsibilities of a class should be closely related and should not be diverse. In Fig. 14, :PaidSubscriber is sending those messages to :Subscription that have nothing to do with the job of being a subscriber, i.e.

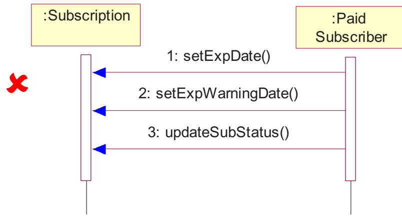

Fig. 14. An SQD with Low Cohesion

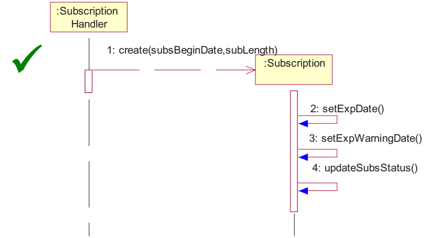

Fig. 15. An SQD with High Cohesion

:PaidSubscriber is handling the attributes which are irrelevant to its function, causing poor cohesion. A better solution is shown in Fig. 15, where the CO :SubscriptionHandler provides the data to :Subscription, which then calculates and sets the attributes itself.

3.3.2 Low Coupling Check

"Low Coupling" is a design goal to assign responsibilities such that coupling is reduced to the extent possible while observing "High Cohesion" and other guidelines. Use the following guidelines to achieve low coupling:

1. Ensure that the recipient object of a message and the parameter in a message is either part of the state of the sending object; passed as a parameter to the method sending the new message; or returned from a previous message sent within the current method (Law of Demeter; Rowlett, 2001).
2. To send a message, use a source class that is already coupled to the target class.
3. Introduce a CO if many messages are being passed between two classes. In this way, the CO can coordinate among multiple objects.
4. Make sure a parameter is not passed again and again in multiple messages.

3.3.3 Fork or Stair Check

Application of the aforementioned guidelines results in a visual pattern to the SQD, which is descriptively called a "fork" or "stair" pattern (Jacobson, 1992). Once constructed, a message sequence can have a very noticeable visual pattern resembling either a "fork" (Fig. 16) or a "stair" (Fig. 17). This effect is more than just appearance; it presents an overall indication of how responsibilities are assigned. An SQD will probably exhibit both patterns, depending on the relationships of the classes. A "fork" structure is recommended when messages could change the order of message sequences or when there is a central object that controls many messages as in the case of enforcing an integrity constraint. Interactions of control objects frequently show a "fork" pattern. This pattern helps in reuse, error recovery, and maintenance (Rowlett, 2001). A "stair" structure is recommended when there is no central object in the SQD or when messages have strong connections among objects based on relationships such as a temporal relationship (e.g, order – invoice – delivery – payment) or an aggregation hierarchy.

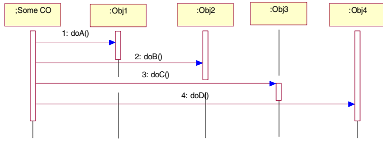

Fig. 16. The Fork Pattern

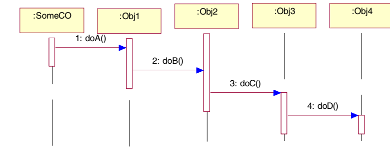

Fig. 17. The Stair Pattern

3.3.4 Central Class Check

Chonoles and Schardt (2003) presents the notion of "central class" concept. They suggest identifying a central class for a use case, and note that this class will probably do a large part of the work in the interaction. For example, if a use case involves the classes shown in Fig. 18, it can be seen that Obj2 is the central class – it has the shortest access route (least hops) to all the other classes in the interaction. With this observation, if the modeler looks at the finished SQD (Fig. 19), he would notice a "fork" structure beginning from Obj5. It might be an indication that responsibilities are incorrectly assigned (Obj5 has the most difficult access to the other classes). On the other hand, a fork structure emanating from Obj2 would not be surprising.

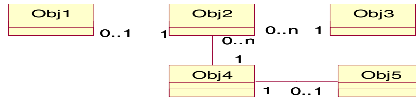

Fig. 18. Identifying the central class Obj2

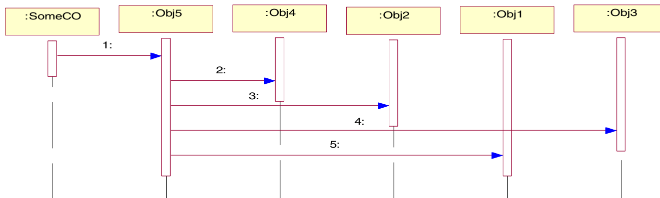

Fig. 19. Likely not a good pattern for the class diagram in Fig. 18

4 Conclusion

In this paper, we have presented a multi-level development methodology for developing SQDs in UML. Our research is motivated by the need of providing a practical method with easy-to-use guidelines for novice SQD developers. We have included guidelines and common visual patterns in SQDs, highlighting the frequently committed mistakes by novices. The guidelines were tested and examples were collected for the past five years of teaching SQDs in a graduate class. The students found our guidelines usable and effective. In future, we will perform a formal study to measure the number and nature of mistakes they make at each level of the methodology.

References

1. Ambler, S.W.: UML 2 Sequence Diagram Overview (2008a), http://www.agilemodeling.com/artifacts/sequenceDiagram.htm
2. Ambler, S.W.: UML Sequence Diagramming Guidelines (2008b), http://www.agilemodeling.com/style/sequenceDiagram.htm
3. Aredo, D.B.: Semantics of UML sequence diagrams in PVS. In: UML 2000 Workshop on Dynamic Behavior in UML Models, Semantic Questions, York, UK (2000)
4. Arlow, J., Neustadt, I.: UML and the Unified Process: Practical Object-Oriented Analysis and Design. Addison-Wesley Professional, Boston (2002)
5. Baker, P., Bristow, P., Jervis, C., King, D., Mitchell, B., Burton, S.: Detecting and resolving semantic pathologies in UML sequence diagrams. In: 10th European Software Engineering Conference, pp. 50–59. ACM, New York (2005)
6. Bist, G., MacKinnon, N., Murphy, S.: Sequence diagram presentation in technical documentation. In: 22nd Annual International Conference on Design of Communication: The Engineering of Quality Documentation, pp. 128–133. ACM, New York (2004)
7. Chonoles, M.J., Schardt, J.A.: UML 2 for Dummies. Wiley, Hoboken (2003)
8. Fowler, M.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley Professional, Boston (2000)
9. Jacobson, I.: Object Oriented Software Engineering: A Use Case Driven Approach. Addison-Wesley Professional, Boston (1992)
10. Larman, C.: Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall PTR, Upper Saddle River (2004)
11. Li, L.: Translating Use Cases to Sequence Diagrams. In: 15th IEEE International Conference on Automated Software Engineering, Washington, DC, pp. 293–296 (2000)
12. Merdes, M., Dorsch, D.: Experiences with the development of a reverse engineering tool for UML sequence diagrams: A case study in modern java development. In: 4th International Symposium on Principles and Practice of Programming in Java, pp. 125–134. ACM, New York (2006)
13. Object Management Group. UML 2.0 Superstructure Final Adopted specification (2003), http://www.omg.org/cgi-bin/doc?ptc/2003-08-02
14. Pooley, R., Stevens, P.: Using UML: Software Engineering with Objects and Components. Addison-Wesley, Harlow (1999)
15. Robinson, W.N., Woo, H.G.: Finding Reusable UML Sequence Diagrams Automatically. IEEE Software 21(5), 60–67 (2004)
16. Rountev, A., Connell, B.H.: Object naming analysis for reverse-engineered sequence diagrams. In: 27th International Conference on Software Engineering, pp. 254–263. ACM, New York (2005)
17. Rowlett, T.: The Object-Oriented Development Process: Developing and Managing a Robust Process for Object-Oriented Development. Prentice Hall, Upper Saddle River (2001)
18. Song, I.-Y.: Developing Sequence Diagrams in UML. In: 20th International Conference on Conceptual Modeling, pp. 368–382. Springer, London (2001)
19. Xia, F., Kane, G.S.: Defining the Semantics of UML Class and Sequence Diagrams for Ensuring the Consistency and Executability of OO Software Specification. In: 1st International Workshop on Automated Technology for Verification and Analysis, Taipei, Taiwan (2003)