Journal of Information Technology and Computer Science
Volume 5, Number 2, August 2020, pp. 221-234
Journal Homepage: www.jitecs.ub.ac.id

Challenges in Developing Sequence Diagrams (UML)

Tri A. Kurniawan[1], Lam-Son Lê[2], Bayu Priyambadha[1,3]

[1] Software Engineering Lab., Faculty of Computer Science, Universitas Brawijaya, Indonesia
[2] Enterprise Software & Processes Group, Faculty of Computer Science & Engineering, HCMC University of Technology, Vietnam
[3] Laboratory of Software Systems and Software Engineering, Interdisciplinary Graduate School of Agriculture and Engineering, University of Miyazaki, Japan
{triak@ub.ac.id, lam-son.le@alumni.epfl.ch, bayu@earth.cs.miyazaki-u.ac.jp}

Received 01 July 2020; accepted 28 July 2020

Abstract. During the object-oriented software design phase, the designers have to describe the dynamic aspect of the system under development through the most common interaction diagram variant in UML 2.0, i.e. sequence diagrams. Some novice designers, including undergraduate and postgraduate students, suffer from making inappropriate models due to insufficiently detailed guidance required to develop such sequence diagrams. This paper classifies some potential mistakes which are likely performed by such novice designers, and discusses the corresponding corrections. We summarized such mistakes based on our long experiences in teaching software modeling classes as well as software analysis and design classes. There were classified twenty-one potential mistakes with respect to the syntactical and semantical correctness of the developed models. It is concluded that novice designers have to be aware and take into account the identified mistakes in such a way they can produce correct sequence diagrams.

1 Introduction

Unified Modeling Language (UML) is widely used in software development practices [1] [2] [3] [4]. A recent survey placed Python and Java, regarded as object-oriented programming languages, at the top rank of programming languages [5], which, in turn, requires UML for software modeling at the software analysis and design phases. As such, UML becomes a de-facto modeling language for object-oriented software systems. Since its first release in 1997, Object Management Group (OMG) maintains eleven UML specifications, including the latest release, i.e. version 2.5.1[1]. UML provides several diagrams categorized into two major kinds of diagram types, i.e. structure diagrams and behavior diagrams [6]. Structure diagrams represent the static aspect of the system under development, while behavior diagrams abstract the dynamic aspect. Sequence diagrams are widely used to describe such a dynamic aspect, which represent the time ordering of messages interchanged between objects [6] [7].

Based on our experiences in teaching software analysis and design classes as well as software modeling classes in undergraduate and/or postgraduate degrees for more than ten years, we observed some mistakes that are likely made by the

[1] Website: http://www.omg.org/spec/UML/2.5.1

students. Most students, as the novice software designers, encounter some challenges in abstracting interaction flows into a representative sequence diagram as the realization of a use case scenario. They may focus on the class diagram as an important diagram in object-oriented design [2]. General guidances of applying the sequence diagrams they acquire from many works of literature are not sufficient [8]. Further, there are different explanations in applying UML diagrams in some cases between one book to the others. As such, the students suffer from developing incorrect sequence diagrams, which is, in turn, modeling invalid systems.

In some cases, the created models differ from the developed systems. Concerning the software quality, the correct analysis and design models will determine the product quality of the subsequent phases, i.e. implementation and testing. The correctness of software models is determined by syntactic quality, semantic quality, and pragmatic quality [9] [10]. In regard to incorrect sequence diagrams, they may not conform to the syntax of the UML sequence diagram (i.e. syntactically incorrect) as well as violate the problem domain (i.e. semantically incorrect). There exist some attempts to guide teaching software modeling and UML [11] [12] [13], as well as dozens of UML guidance books, e.g. [2] [7] [14] [15]. However, to our knowledge, no publication focuses on describing the potential mistakes that are likely carried out by novice software designers, particularly in developing UML sequence diagrams.

This paper classifies some potential mistakes which are commonly carried out by the students and discusses corresponding corrections in order to develop valid sequence diagrams. We summarize such possible mistakes by observing the student's works along with their class and project assignments, laboratory tasks, final examinations, and thesis. Also, we extended our observation on some other works presenting sequence diagrams, which were available in some literatures and websites. We recommend an appropriate correction at each mistake mainly based on the UML specifications [6], Jacobson's objectory approach [16], and UML books [2] [7] [15], as well as the software model correctness concept, i.e. syntactic and semantic quality [9] [10]. Given such classification, novice designers, including students, can be assisted to avoid creating invalid sequence diagrams in their works.

The rest of this paper is structured as follows. Section 2 briefly discusses the fundamental of the sequence diagram in UML 2.0. Section 3 introduces challenges in developing sequence diagrams as well as related discussions. Finally, Section 4 describes the conclusion.

2 Sequence Diagram Fundamental in UML 2.0

In UML 2.0, there are seven concrete diagrams and one abstract diagram to represent the behavioral aspect of a system under development [6]. Such an abstract diagram refers to the interaction diagram in which four concrete diagrams inherit from, including sequence and communication diagrams. Among all the others, the sequence diagram is the most common variant. Sequence diagram concerns describing the time ordering of messages interchanged between objects [6] [7].

A sequence diagram is developed based on a particular use case scenario that encompasses both basic and alternative flows [15]. As such, a valid and detailed scenario becomes crucial information used to construct interaction between objects involved in a particular use case. Elements of a sequence diagram include actor, objects, message, lifeline, execution occurrence, and frame, as illustrated in Fig. 1. Further, we may add a reply message, combined fragment, and creation message if needed. All objects are placed at the top of the diagram along the X-axis. The new

object is placed on the right side of the other object which initiates further interaction. Messages are placed along the Y-axis, going down to represent an increasing time. A complete sequence diagram is outlined in a frame whose name represents the use case being described.

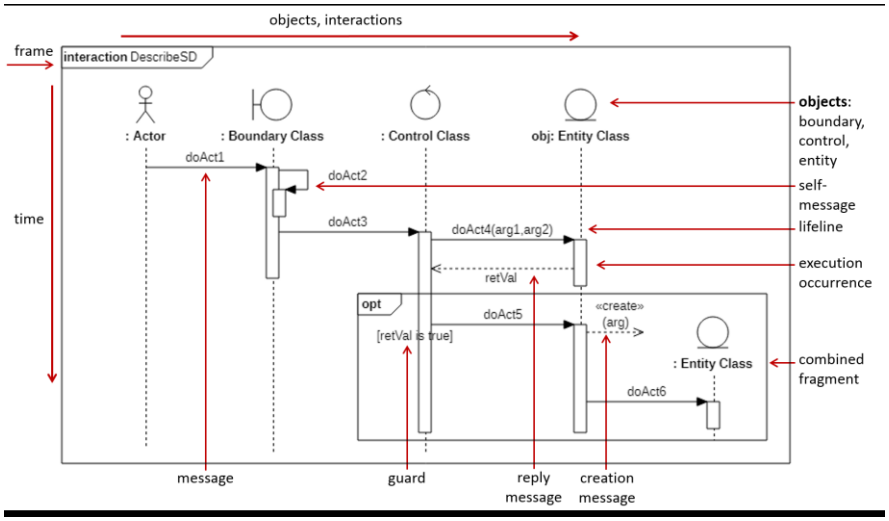

Fig. 1. Common elements of a sequence diagram

Objects are classified into three stereotypes, i.e. boundary, entity, and control, initially introduced by Jacobson [15] [16], even OMG specifies such objects in general [6]. A boundary (or interface) object manages communication between actors and the system under development. Each actor requires its own boundary object for its action on the system [16]. The boundary objects are identified from the use case scenario. An entity object represents persistent information handled by the system, which usually corresponds to a particularly relevant concept or thing in real life. It is obvious to identify entity objects from the use case scenario derived from the problem domain. It is noted that one use case represents one specific function in the software system. In such a function, a control object manages the interaction/logical flow of one use case scenario (one software function). It plays a glue between boundary and entity objects [15]. In a sequence diagram, we may have more than one boundary and entity objects. However, one control object per use case is recommended.

Decorating object in the diagram is essential for better understanding. Each object is labeled at the bottom by its corresponding class or actor. The colon at each label represents an instance of a class or actor. Any name placed before the colon indicates the specified object name. We may a particular name at each object as required. Each object has its own lifeline describing the time-line for a process or the life of the object during a sequence [6] [14]. On a lifeline, there exist execution occurrences (or focus of controls [2] [7]) denoting the start or the end events of execution, i.e. sending or receiving messages.

Messages convey information interchanged between related objects. The synchronous messages are commonly used in a sequence diagram, instead of asynchronous ones. Messages, as shown di Fig. 1, are examples of synchronous messages using a filled arrowhead [6]. Once a message sent from the first object to the second object, there exists a method invocation at the second object by the first

object. As such, the message label denotes a method name at the receiving object, which must be in a verb form. A self-message is similar to such synchronous messages with a special case in which the method invocation occurs in the same object. Creation messages are used for object instantiations, which correspond to a constructor method in a class. A reply message delivers information replied by the receiving object, which is labeled by the value being returned. Such a reply message must be placed instantaneously below a related message, if needed. Some methods have a return value, and some others are not. If a method has a return value, then the diagrams must show it as a reply message; otherwise, it needs not to be drawn.

Combined fragments represent a set of object interactions according to a particular condition represented by its operator. Such fragments may improve the readability of the diagram. The operators include **alt** (alternatives, representing an XOR behavior choice), **opt** (option, denoting a choice behavior based on the only true condition), **loop** (loop, describing an iterative behavior), and **ref** (reference, depicting a go-to another sequence diagram behavior) [2] [6]. A guard may involve in a certain operator, which is a basis for evaluating the subsequent interaction flow.

Concerning object interactions, there should be a pattern to follow such that change complexity [16] and object responsibilities [2] can be proportionally managed. We construct the following four rules adapted mainly from Rosenberg's work [15] with some adjustments based on Larman's work [2]:

- **Rule 1**: *Actor objects can send messages only to boundary objects*. Such messages (i.e. synchronous) represent instructions to be performed by the system.
- **Rule 2**: *Boundary objects can send messages to control and actor objects*. Messages (i.e. synchronous) sent to control objects denote instructions to be performed by such control objects. Messages sent to actor objects represent information to the actor objects, which may then affect their behaviors.
- **Rule 3**: *Entity objects can send messages to control objects as well as to entity objects*. Messages sent to control objects specify a return value of a received message from such control objects, i.e. reply messages. Messages delivered to other entity objects denote instructions to be performed by such objects according to the responsibility patterns, e.g. Creator, Information Expert [2].
- **Rule 4**: *Control objects can send messages to boundary, entity, and control objects*. Messages (i.e. synchronous) sent to boundary objects represent instructions to boundary objects for displaying particular information. Messages (i.e. synchronous) delivered to entity objects denote instructions to be performed by such entity objects. If the interaction flow is too complicated, we may delegate some responsibilities from a control object to another control object by sending a message (i.e. synchronous).

3 Challenges and Discussion

We classify and discuss the following mistakes, which are likely carried out by the students. We intend to support such novice designers to get their skills upgraded. For the shake of clarity, we illustrate a particular case with a snapshot diagram for just highlighting the main issue being discussed.

3.1 Sequence diagrams implement MVC pattern

It is often assumed that the MVC (model-view-controller) pattern is implemented in the software design if we model such software using sequence diagrams. In fact, sequence diagrams do not relate to the MVC pattern since they have different

concepts. As sequence diagram modeling relies on use case scenario [2], it refers to Jacobson's objectory approach [15] [16]. Such an approach classifies objects into three stereotypes, i.e. entity, control, boundary, which is then known as the ECB (entity-control-boundary) approach. There exist object interaction rules in ECB [15], which differ from MVC. Further, the control object in ECB is part of the domain layer or business logic and coordinates works requested by actors through the boundary object. In comparison, the controller object in MVC is part of the UI layer and manages UI interactions.

3.2 Diagram uses a class representation instead of an object

Some designers do not put a colon at the objects' label for representing objects. As such, the labels denote classes, instead of objects. Based on the definition of the sequence diagram, there should be a set of objects put on the top of the diagram. Fig. 2a illustrates two interchanging classes, which is the wrong representation. Correcting such a mistake, a colon '**:**' should be put before the name of the class representing an object, as shown in Fig. 2b. We may also define a specific name of the object by putting such a name before the colon, e.g. `obj:Registrant`.

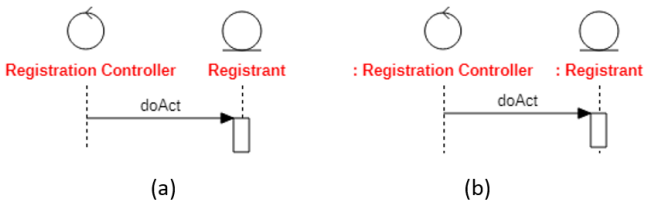

Fig. 2. Interaction of objects: (a) wrong representation; (b) correction

3.3 Diagram does not include all related involving objects

Each sequence diagram should depict all related objects within a use case scenario for both main and alternative flows. Every noun introduced in such a scenario becomes an object candidate in the diagram. This setting requires a detailed, complete, and valid use case scenario. Once such a good scenario is available, we need to improve our robustness analysis [15] [16] if this kind of mistake occurs.

3.4 Class naming of involving object is not in noun form

Class, especially an entity, must represent a set of shared characteristic objects/things that occur in the problem domain and/or solution domain. As such, the name of the class should be in a noun form, which is valid in a particular context. Fig. 3a depicts an interaction between a control object (i.e. an instance of `Registration Controller` class) and an entity object (i.e. an instance of `Registration` class). In the registration process, the valid entity to be created is the `Registrant`, as shown in Fig. 3b, represents the one who will be accepted as a member, while `Registration` denotes the process itself.

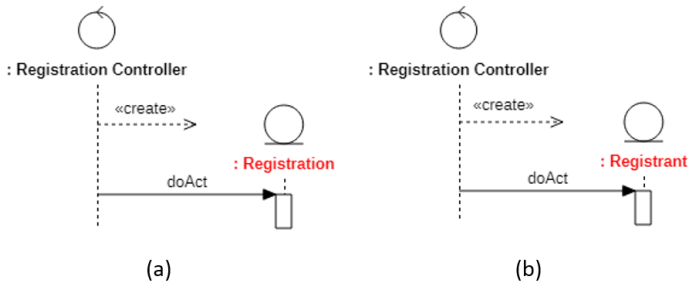

Fig. 3. Class naming convention in a noun form: (a) wrong representation; (b) correction

3.5 Object performs inappropriate responsibility

One crucial task in object design is how to assign responsibilities properly to each object or class [2]. Such a task determines the coupling and the cohesion levels, which in turn affect the software quality. Fig. 4a illustrates validating the registration form from any unacceptable data, e.g. blank fields, data type violation, which is performed by the control object requested by the boundary object as part of the `doAct2` message. As form validation is more relevant to the boundary object to perform, it is more suitable to assign such a responsibility to the boundary object, as shown in Fig. 4b. This change will improve the cohesion level of the design in the boundary object. If there is a change in validation rule w.r.t. form changes, we only need to adjust one relevant place, i.e. boundary object. This approach should be appropriately applied in all objects, including boundary, control, and entity.

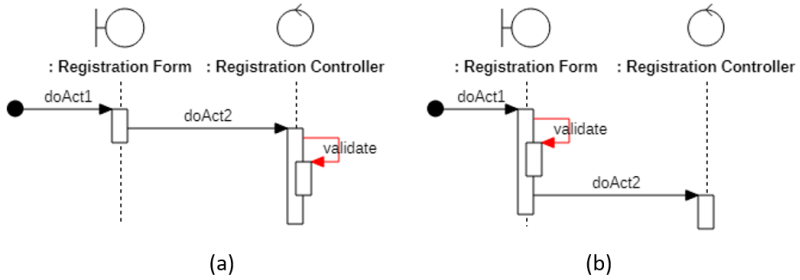

Fig. 4. Object responsibilities: (a) wrong representation; (b) correction

3.6 Actor sends a message to the controller object directly

By the rules mentioned above, actor objects can send messages only to boundary objects. It is unacceptable to put messages from actor objects directly to control objects. Some novice designers are pushed to do such a mistake due to some constraints associated with the implementation framework they use, as illustrated in Fig. 5a. Correcting such a mistake, we have to put a boundary object in between, as shown in Fig. 5b.

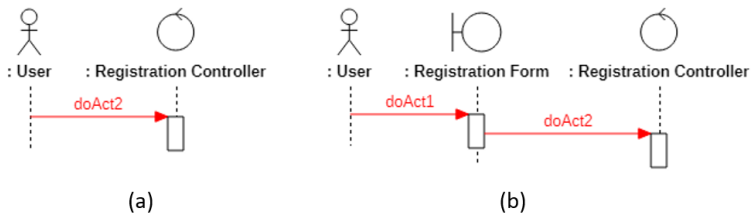

Fig. 5. Actor object interactions: (a) wrong representation; (b) correction

3.7 The message is not in an instructional verb form

Messages (synchronous) denote method invocations at the receiving objects requested by the sending objects. As such, messages must be labeled in the instructional verb forms representing what the objects have to do. Further, messages will be method names in their respective classes when we develop a complete class diagram. This kind of mistake includes labeling messages in noun forms, in a phrase describing a task sequence or what a process does. Correcting this mistake, we have to specify a clear compact instruction word in verb form to label a message, e.g. `validate`, `save`, `calculateBalance`. Fig. 6a shows a message sent from an actor object describing a task sequence performed by such an actor. The message does not specify what the boundary object has to perform. Fig. 6b illustrates the more specific message consisting of an instruction to the receiving object.

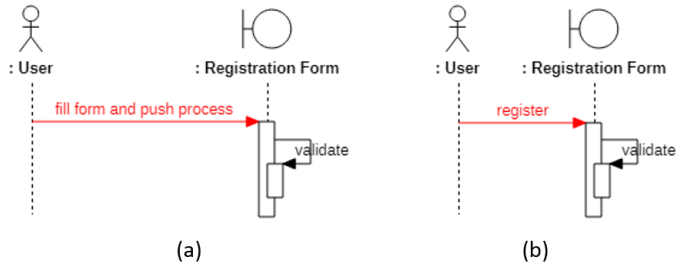

Fig. 6. Messages in intructional verb forms: (a) wrong representation; (b) correction

3.8 Messages from actor objects represent the actor's activities

Some novice designers put messages sent from actor objects representing the activities performed by such actors. According to the specification, a message in sequence diagrams represents a method invocation of the receiving object. As such, messages should not constitute any activity carried out by the sending object. Fig. 7a illustrates two incorrect messages sent by the actor object since they represent the actor object's activities, i.e. `fill username`, `fill password`. In this case, the only correct message is `submit`, which must be handled by the boundary object, as shown in Fig. 7b.

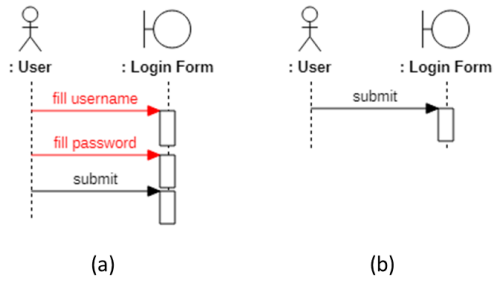

Fig. 7. Message invocation: (a) wrong representation; (b) correction

3.9 The same boundary object handles messages from various actors in different interactions

Novice designers often think a boundary object can be used in many different use case scenarios since such a boundary object conveys substantially similar information to its actor objects. In the implementation stage, such an approach may be valid. We, however, have to be careful if we deal with modeling since each boundary object has its own corresponding generated events associated with their relevant boundary object elements, such as buttons, links. Correcting this mistake, we should have dedicated boundary objects at each use case scenario or interaction.

3.10 Return result, if required, from a message is missing

A synchronous message may require a return value from the receiving objects. This value should be explicitly modeled in the diagram since such a value will be considered in the subsequent interaction flows. Fig. 8a depicts a message without a return value, although it requires. Given this model, we may be confused to understand the next interaction flows. Fig. 8b outlines a more understandable model by explicitly showing the return value required by the message `getStatus`. However, we can also represent the result value or return value included in the message, e.g. `getStatus:status` [2] [6], without using a reply message.

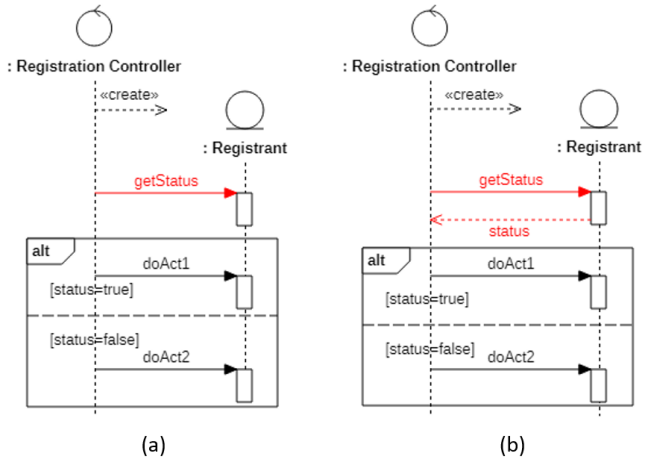

Fig. 8. Return result representation: (a) wrong representation; (b) correction

3.11 Reply message does not have specific information or value

As the reply message delivers a return value of a particular synchronous message, such a return value should be put on the label of the reply message. It is essential information required by the subsequent interaction flows, as described in Section 3.10. Fig. 9a illustrates an unspecific return value, which is, in turn, difficult to deal with. Fig. 9b shows a specific return value, i.e. `aDate`, such that it can be used as a specific guard in the subsequent combined fragments, as discussed in Section 3.10.

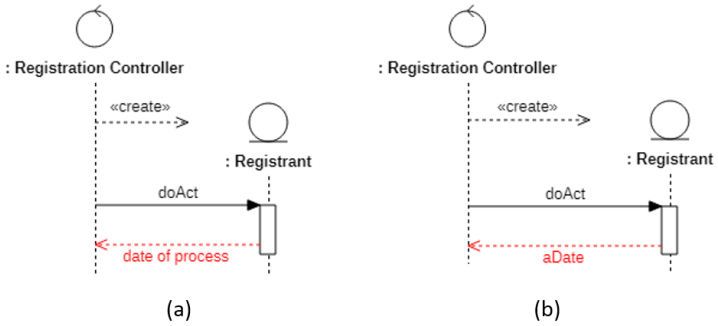

Fig. 9. Reply message representation: (a) wrong representation; (b) correction

3.12 High coupling interaction exists between two objects

High software quality can be achieved by creating a software design that has low coupling and high cohesion. These two design quality metrics influence each other. If we increase the design cohesion, we will get low design coupling. Fig. 10a depicts a high coupling interaction between two objects since there exist several single method invocations engaged by the control object for setting the attributes after entity object creation. We can improve it through a parameterized object creation message, as shown in Fig. 10b. It is noted that inadequate object responsibility assignments will also affect the coupling level [2].

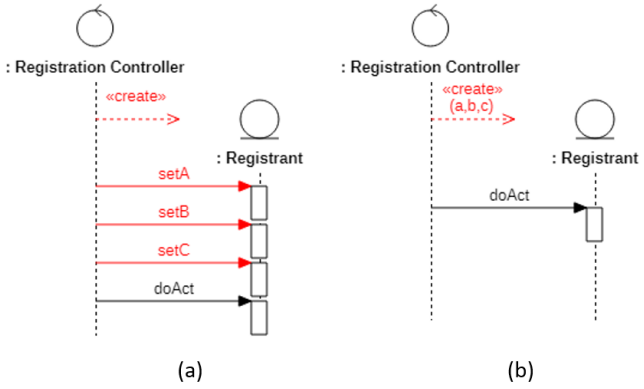

Fig. 10. High coupling interactions: (a) wrong representation; (b) correction

3.13 Entity object represents a particular table in the database

Sequence diagrams are developed based on the information described in use case

scenarios. In such scenarios, we do not deal with any detail within the system under development, including specific databases or tables. We just introduce all relevant objects which exist in the problem domain and involve in such scenarios. Every object in the sequence diagram represents a single object instantiated from a particular class. Fig. 11a shows an entity object `Registrant Table`, which is an inappropriate entity object due to it should be absent in the use case scenario. Fig. 11b illustrates the correct entity object `Registrant`, which relevant to the use case scenario. Further, it is noted that the entity object `Registrant` represents a single object registrant, while `Registrant Table` denotes a set of registrants.

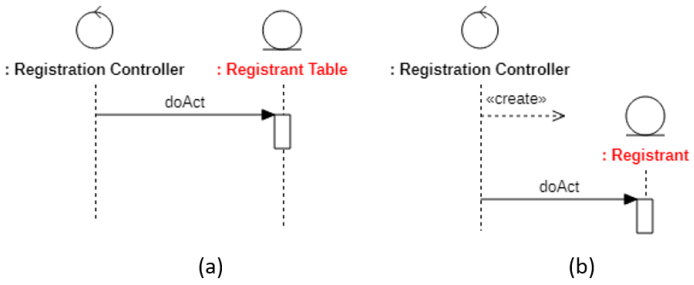

Fig. 11. Entity object representation: (a) wrong representation; (b) correction

3.14 Name of an entity object uses the word 'model'

Some novice designers put the word 'model' at the end of each entity object name. They assume such objects as models in MVC pattern in which sequence diagrams implemented such pattern, as described in Section 3.1. For example, we can use Fig. 11a to illustrate this mistake, i.e. `Registrant Model` instead of `Registrant Table`. As we abstract any related object that occurs in the problem domain and/or solution domain, such an object should represent something in such domains, i.e. `Registrant`, as shown in Fig. 11b.

3.15 Creation messages do not fit with the object representations

Creation messages are used for instantiating objects which do not exist at the time such messages will be invoked. As such, the instantiated objects should be represented once the creation messages sent. In sequence diagrams, all objects which are not instantiated will be represented at the top of the diagrams. For example, object `Registrant Table` in Fig. 11a already exists when the sequence of the interactions is started. While object `Registrant` in Fig. 11b will be instantiated once required using the creation message. If such a creation message occurs, the corresponding created object has to be represented at the point of such a message, not at the top of the diagrams.

3.16 Message parameters are sometimes missing

In some cases, messages (synchronous), including creation messages, have parameters that represent information to be manipulated within the invoked method at the receiving object. Such parameters should be explicitly introduced in the diagram such that we can recognize the messages correctly. However, some designers ignore the message parameters if required, and only focus on the message itself. For example, a message with parameters `doAct(arg1,arg2)` is more

recognizable than message `doAct()`, while parameters `arg1` and `arg2` are required by such message.

3.17 Alternative and optional fragments do not rely on specific guard information

Alternative and optional fragments require specific guard information to evaluate the subsequent interaction flows. Some novice designers omit such essential information on the diagram. Fig. 12a shows an optional combined fragment without a guard determining a true condition to execute `doAct`, which is, in fact, based on the reply message `status`. Fig. 12b presents the correct diagram with a specific guard, i.e. `status=true`.

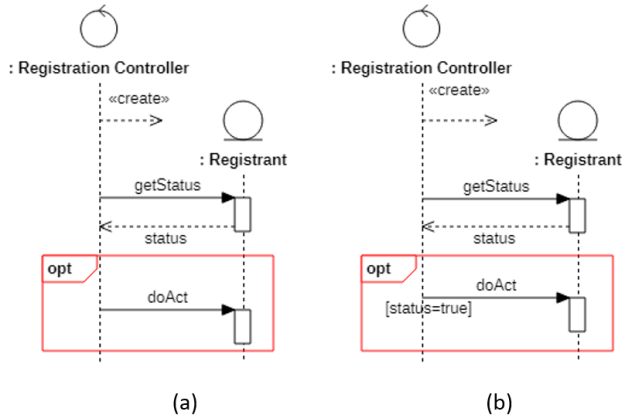

Fig. 12. Specific guard information: (a) wrong representation; (b) correction

3.18 Diagram does not represent all flows described in the use case scenario

Each sequence diagram represents the logical flows that occur in a particular use case scenario, including basic and alternative flows. In some cases, we only illustrate the basic flow in the diagram, ignoring entirely or partly the alternative flows. Correcting this, we have to draw all flows described in the use case scenario.

3.19 Diagram uses an inappropriate boundary object of the menu page

Novice designers likely start to draw a sequence diagram using a boundary object representing the menu page of the system under development. The sequence continues to another boundary object, which receives actual information from the actor object. In fact, we do not include any boundary object representing the menu page at the beginning of all sequence diagrams of a particular system. Fig. 13a illustrates an actor object which clicks a specific menu at the `Menu` page for displaying the `Registration Form` page. Such a boundary object of the `Menu` page is not required in the registration sequence. Also, this interaction violates Rule 2, in which two boundary objects interact with each other. Fig. 13b presents the actual boundary object which interacts with the actor object in the registration flow.

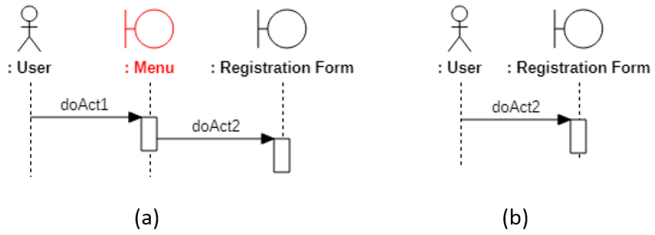

Fig. 13. Boundary object interactions: (a) wrong representation; (b) correction

3.20 Lifeline includes a disconnected execution occurrence

An execution occurrence in the lifeline occurs when an object sends a message to another object, indicating a method invocation. If such an invoked method returns a particular value, the execution occurrence has to be drawn until the point of the reply message. Novice designers are likely not aware of such a focus of control such that the execution occurrence is disconnected. Fig. 14a shows such a disconnected execution occurrence at the lifeline of the control object based on an invoked method `doAct1`. The replay message `retVal` does not point to execution occurrence. Fig. 14b illustrates the correct model. The execution occurrence spans from the start until the end of the events at the control object lifeline concerning the invoked method `doAct1`.

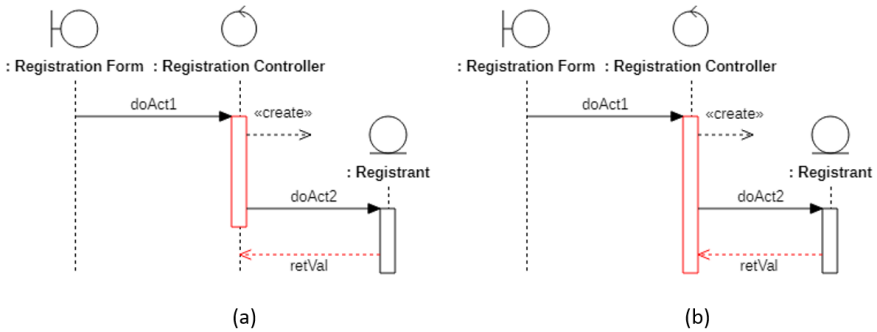

Fig. 14. An execution occurrence: (a) wrong representation; (b) correction

3.21 A sending object does not receive a reply message from the corresponding receiving object

A reply message occurs when an invoked method at the receiving object has a return value. Such a reply message should be drawn from the receiving object to the sending object. The sending object refers to the object that performs such method invocation. Fig. 15a illustrates the wrong diagram in which the entity object sends the reply message `retVal` to the boundary object in response to message `doAct2`. The boundary object, as the sending object, should receive a reply message from the control object, as the receiving object, in response to message `doAct2`. Similarly, the control object should receive a reply message from the entity object in response to the message `doAct3`. Further, Fig. 15a violates Rule 3 since the entity object directly sends a message to the boundary object. Fig. 15b depicts the correct diagram involving two reply messages which are sent in sequence to the corresponding

sending objects, respectively.

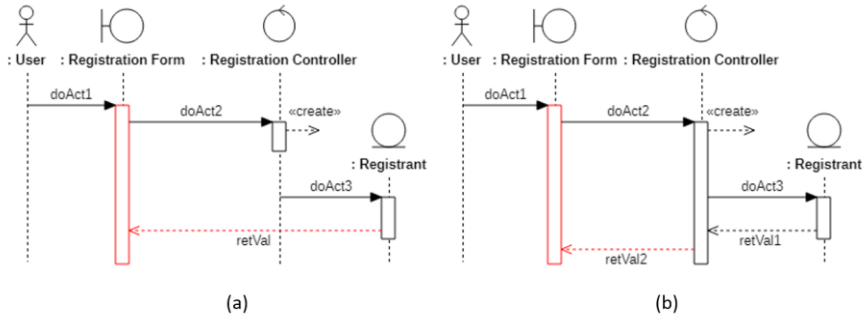

Fig. 15. Reply messages flows: (a) wrong representation; (b) correction

4 Conclusion

We have classified twenty-one potential mistakes, which are likely made by the students as novice designers in developing sequence diagrams based on the specification of UML 2.0 and related works of the literature. They have to be aware of such mistakes if they need to produce such sequence diagrams, which are syntactically and semantically correct. Building correct models become a critical task in software development practices as a way to understand the system under development in a particular aspect. Finally, high-quality software relies on its all related correct models describing various aspects of the system.

References

1. G. Engels, R. Heckel and S. Sauer, "UML --- A Universal Modeling Language?," in *Application and Theory of Petri Nets 2000: 21st International Conference, ICATPN 2000 Aarhus, Denmark, June 26--30, 2000 Proceedings*, M. Nielsen and D. Simpson, Editors, Springer Berlin Heidelberg, pp. 24-38. (2000)
2. C. Larman, Applying UML and Patterns, 3rd Edition, NJ: Prentice-Hall. (2005).
3. C. F. J. Lange, M. R. V. Chaudron, and J. Muskens, "In Practice: UML Software Architecture and Design Description," *IEEE Software,* vol. 23, no. 2, pp. 40-46. (2006)
4. T. A. Kurniawan, D. S. Rusdianto, A. H. Brata, F. Amalia, A. Santoso, and D. I. N. R. P. Raharjo, "An Exploratory Study of Requirements Engineering Practices in Indonesia – Part 2: Efforts, Processes and Techniques," *Journal of Information Technology and Computer Science,* vol. 5, no. 1, pp. 65-74. (2020)
5. S. Cass, "Top Programming Languages 2020," *IEEE Spectrum. (2020)*
6. OMG, OMG Unified Modeling Language (OMG UML) Version 2.5.1, Object Management Group.(2017)
7. G. Booch, J. Rumbaugh, and I. Jacobson, The Unified Modeling Language: User Guide, Addison Wesley. (1999)
8. I.-Y. Song, "Developing Sequence Diagrams in UML," in *International Conference on Conceptual Modeling*. (2001)

9. B. Unhelkar, Verification and validation for quality of UML 2.0 models, Wiley Online Library. (2005)
10. O. I. Lindland, G. Sindre, and A. Solvberg, "Understanding Quality in Conceptual Modeling," *IEEE Software,* vol. 11, no. 2, pp. 42-49. (1994)
11. T. Tamai, "How to Teach Software Modeling," in *The 27th International Conference on Software Engineering*. (2005)
12. L. Kuzniarz and M. Staron, "Best practices for teaching UML based software development," in *International Conference on Model Driven Engineering Languages and Systems*. (2005)
13. B. Westphal, "Teaching Software Modelling in an Undergraduate Introduction to Software Engineering," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. (2019)
14. A. Dennis, B. H. Wixom, and D. Tegarde, Systems Analysis and Design with UML Version 2.0: An Object-Oriented Approach, John Wiley & Sons, Inc. (2005)
15. D. Rosenberg and K. Scott, Use Case Driven Object Modeling with UML: A Practical Approach, Addison-Wesley. (1999)
16. I. Jacobson, Object-oriented software engineering: a use case driven approach, New York: ACM Press. (1992)
17. M. Grossman, J. E. Aronson, and R. V. McCarthy, "Does UML make the grade? Insights from the software development community," *Information and Software Technology,* vol. 47, no. 6, pp. 383-397. (2005).