FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Model based testing of cloud based social networks

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Software & Information Engineering

eingereicht von

Filip Rydzi
Matrikelnummer 1226452

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Dr. Christoph Dorn Ph.D
Mitwirkung: Dr. Alessio Gambi Ph.D

Wien, 24. August 2015

Filip Rydzi	Christoph Dorn

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Model based testing of cloud based social networks

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Filip Rydzi
Registration Number 1226452

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Dr. Christoph Dorn Ph.D
Assistance: Dr. Alessio Gambi Ph.D

Vienna, 24th August, 2015

Filip Rydzi

Christoph Dorn

Erklärung zur Verfassung der Arbeit

Filip Rydzi
Klzava 31, Bratislava, Slovakia

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 24. August 2015

Filip Rydzi

Danksagung

Ich möchte meinen Betreuern Dr. Alessio Gambi und Dr. Christoph Dorn für ihre Hilfe und Unterstützung bedanken. Sie waren immer zur Verfügung, um mir meine Fragen bereitwillig zu beantwortet und haben mir die immer richtige Weg gezeigt. Dank ihre konstruktive Kritik habe ich meine Bacherlorarbeit viel verbessert. Bei der Erstellung meiner Bachelorarbeit habe ich unter anderem mein Verständnis des Kontexts verbessert. Dafür bin ich den Herren Dr. Gambi und Dr.Dorn sehr dankbar, weil die jederzeit von mir verlangt haben, damit ich alles was ich erfunden habe mit aussagekräftige Argumente unterstütze.
Vielen Dank!

Acknowledgements

I wish to express my sincere thanks to Dr. Alessio Gambi and Dr. Christoph Dorn for their help and support. They were always available to willingly answer my questions and to show me the right way. Thanks to the constructive critique by Dr. Gambi and Dr. Dorn I was able to improve a lot of things in my bachelor's thesis. I have learned many new things during the creation of my thesis. Beside the new knowledge in the computer science field, I have learned to understand and explain the contexts better. For this I am very grateful to Dr. Gambi and Dr. Dorn, because they required from me all the time to support each part of my inventions with meaningful arguments.
Many thanks!

Kurzfassung

Diese Arbeit anwendet das Model based testing an eine soziale Netzwerk. Es wird ein Prozess der Umwandlung einer existierenden sozialen Netzwerk in der abstrakten sozialen Netzwerk vorgestellt. Auf der Grundlage der Umwandlungprozess, werden die ausführbare Testfälle von der abstrakten Testfälle, der vom Model der abstraken sozialen Netzwerk abgeleitet wurden, erstellt. Als erstes habe ich in meiner Arbeit die wichtigste Anwendungsfälle und Komponenten einer abstrakten sozialen Netzwerk identifiziert. Für jeden identifizierten Anwendungsfall habe ich die Benutzerrollen, die durch die Ausführung des Anwendungsfalls betroffen sind, festgestellt. Basierend auf dieser Information kann ein allgemeines Modell der abstrakten sozialen Netzwerk erstellt werden. Das allgemeine Modell habe ich mit human Architecture Description Language (hADL) erstellt. hADL ermöglicht eine Menge von Aktionen für jeden Kollaborationobjekt definieren, die von zur Verfügungstehenden Menschenkomponenten ausgeführt werden können. Die modellierte abstrakte soziale Netzwerk funktioniert einwandfrei aus der Sicht der festgelegten Anwendungsfälle. Die Funktionalität der abstrakten sozialen Netzwerk entspricht der Funktionalität von vielen modernen sozialen Netzwerken, dies ermöglicht, dass der vorgestellte Prozess, auf viele sozialen Netzwerken angewandt werden kann. Um abstrakte Testfälle zu generieren, wird das erstellte Modell als Eingabe für das Model based testing benutzt. Mit einer existierende Implementierung der sozialen Netzwerke kann man durch das Folgen des definierten Prozesses, die abstrakte Testfälle in die konkrete ausführbare Testfälle umwandeln. Die konkrete Testfälle kann man auf eine existierende soziale Netzwerk ausführen. Für das Testen habe ich mein eigenes soziale Netzwerk benutzt, diese Netzwerk läuft auf Cloud-Platform von Google. Basierend auf der Ergebnisse des Testens kann der Tester feststellen, ob die konkrete soziale Netzwerk der abstrakten sozialen Netzwerk entspricht.

Abstract

This thesis presents an application of model based testing to social network, and the process of mapping a concrete social network to the abstract social network. Based on this mapping concrete executable test cases can be created from the abstract test cases derived from the model of the abstract social network.
Firstly I identified the key use cases and components of an abstract social network, for each use case I identified, which user's roles should be affected by the execution of the use case. Based on this information a general model of an abstract social network can be created. I created the general model using hADL, which allows to define a set of actions for each collaboration object which can be executed by various human components. I assume that abstract social network works correctly in the view of the identified use cases. The functionality provided by the abstract social network matches the functionality of many existing modern social network, this allows the methodology to be applied to them. The model is used as input for model based testing to generate abstract test cases. Having a concrete implementation of a social network and following the process defined in this thesis, the abstract test cases are mapped to the concrete ones, which can be executed on the concrete social network. For the concrete social network, I used my own social network running on Cloud Platform by Google. Based on the results of the testing the tester can state whether the concrete social network matches the abstract social network.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Nowadays there are lots of people using many various social networks (SNs). There is also a huge number of new SNs, each with a specific aim, connecting people around the world. In order to satisfy the users, a Social Network (SN) has to provide some basic functionalities, but also persuade the users that their private content stays private and won't be available to any unauthorized entities. The aim of this thesis is to identify the key use cases of a SN and develop a methodology to prove that a given SN provides the specified functionalities by means of automated test cases generation.

1.2 Problem statement

The model of an abstract SN is constructed. This model identifies the key components of an abstract SN, as well as the user's roles participating in fulfilling various actions. The functionality of the modeled SN is considered to be correct. The test cases are derived from the given model using a Model based testing (MBT) approach.
The problem solved in my work is how to create the specific test cases (from the derived abstract test cases), which can be executed against a specific implementation of SN. To solve this problem are used the abstract test cases derived in this work. The next input is a specific SN. Based on the result of the testing it's possible to state, which changes should be done in the specific SN to match the abstract SN, which implies the correct functionality of specific SN in the view of the tested use cases.

1.3 Aim of the work

The aim of my work is to define a process, which enables to create concrete executable test cases from abstract test cases, derived using the MBT from a general model of SN.

The concrete executable test cases can be directly executed on a specific SN. Based on this testing the tester can state about the quality of the specific SN.

1.4 Methodological approach

1. Creating and modeling a abstract SN.
   To prove whether the concrete SN matches the abstract SN and based on the methodology to state whether the concrete SN works as expected, I need to design a general model of the abstract SN. Then it's possible to use this methodology for every SN, which can be mapped to the defined model.
   This model should identify and describe key components of abstract SN. These components will be shared between various users roles. The model should also define a set of actions, which can be performed by various user's roles on various components.

2. Identifying and modeling the key use cases.
   In the next step it's necessary to identify a list of key use cases of SN. Each of these use cases can be later tested on the specific SN. Because a SN offers an interaction between various users roles, and because we want to test for each use case whether only the authorized users roles are participating and interacting with other users roles and system components as expected, the very appropriate approach how to model such interaction is a sequence diagram. Therefore next step in the methodology is to create a sequence diagram for each use case, describing the interaction between user's roles and system components (of abstract SN).

3. Generating abstract test cases.
   Since we have the model of the abstract SN (from step 1) and a set of sequence diagrams, we know the message exchange between the system components, as well as the message exchange between the user's roles and system components. Cartaxo et. al. [2] presented a MBT approach to generate test cases from sequence diagrams. I slightly modified this approach and used it to generate the test cases. The detailed description of test case generation is explained in the Methodology Chapter 3.

4. Mapping the model of concrete social network to the abstract one.
   The test cases generated in the previous step are abstract and therefore not executable on the concrete SN. To execute the test cases on the concrete SN it's necessary to create the concrete executable test cases from the abstract ones. To create them, it's necessary to provide a model of the concrete SN. This model has to be mapped to the model of abstract SN. Based on the information provided by the mappings of the models, the derived abstract test cases can be mapped to the concrete executable test cases. Since the sequence diagrams of the abstract SN were used to derive test cases, it's needed to provide sequence diagrams of the

concrete SN. To create the sequence diagrams, we need to know all the participating components, therefore we need also to create a component diagram of the concrete SN.

5. Deriving concrete test cases.
   Based on the mapping from previous step, we create the concrete test cases, which will be executable on the concrete SN.
   The results of the testing show whether the concrete SN matches an abstract one and therefore shows if the concrete SN works correctly according to abstract SN. The results will be used to show, which improvements should be done in the concrete SN to obtain the correct functionality, which is described in the model of abstract SN.

The Figure 1.1 below describes the explained methodology in a simple diagram.

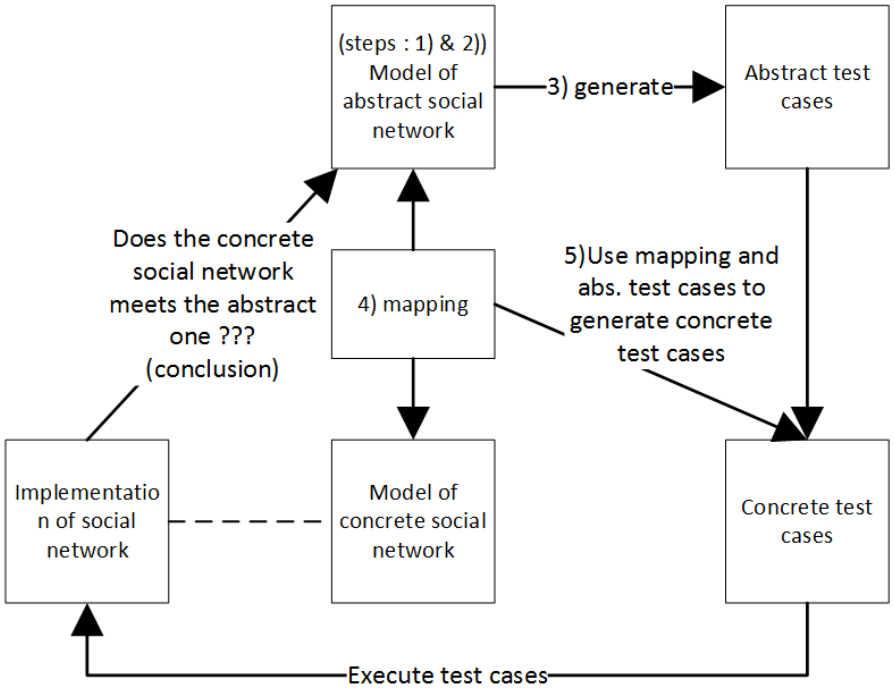


Figure 1.1: Diagram showing the explained methodology

1.5 Structure of the work

This thesis is structured into chapters as follows: Chapter 1 defines a problem being solved in my thesis, it also briefly explains the methodology used to address the problem.

Chapter 2 gives an overview of existing approaches, beside the ones used for this thesis, which could be applied also for the needs of this work and briefly explains the differences between them. Chapter 3 explains each step of the methodological approach, it presents the creation of the model of abstract SN, the process of abstract test case generation is also explained and used in this Chapter. In the Chapter 3 is presented the concrete SN used for execution of concrete test cases, as the mapping between the abstract and the concrete SN. Chapter 4 gives a process defining the steps to create the concrete executable test cases from the abstract ones, it presents the application of the defined process on two of the identified use cases. The last Chapter 5 evaluates the process defined in this Thesis, it also evaluates the results of the testing, based on the evaluation it concludes whether the tested concrete SN matches the introduced abstract SN.
The appendix support this thesis by applying the model based testing on other use cases and presents the implementation of the test cases.

CHAPTER 2

State of the art

2.1 Model based testing

MBT is a testing approach where the test cases are derived from a model of the System under test (SUT). Utting [18] provides a set of main stages involved in MBT:

1. Building an abstract model of the SUT
2. Validating the model
3. Generating abstract test cases from the model
4. Refining those abstract test cases into concrete executable tests

During the design phase of software creation, structural and behavioral models are constructed. Behavioral models enable to specify the interaction between components and how the system should react on various events, messages etc, therefore the behavioral models are better than structural models to generate test cases.
The following behavioral models are used to generated test cases: sequence diagrams [2] [3], state machine diagrams [7] [10] and activity diagrams [11] [6]. There are various algorithms to generate test cases from behavioral models. Shirole and Kumar [14] provide a categorization of the existing algorithms into the following categories:

- tree based test case generation - derives the test cases using the traversal, function minimization [8] etc.
  Li et al. [8] present an approach to generate the test cases using the sequence diagrams and Object Constraint Language (OCL) expressions. The first step is to construct a scenario tree. The nodes in the scenario tree represents the messages being exchanged between objects in sequence diagram. The edges are showing the order in which the messages are executed. Then the message paths are acquired from scenario tree. Then all relevant classes, class attributes and class operation

are needed to be identified. Based on the identified class information the OCL is used to describe pre- and postconditions of operations and invariants of the classes. The test cases are derived based on the OCL constraints and the acquired message paths. The derived test cases ensures message path coverage and pre- and post conditions coverage.

- graph based test case generation - derives the test cases using the Depth First Search (DFS) or Breadth First Search (BFS) traversal [2] etc.
  Cartaxo et al. [2] present an approach to generate the test from the sequence diagram. First, the sequence diagram is transformed into a Labeled Transition Systems (LTS). By traversing the LTS with DFS algorithm a set of paths is derived. Each derived path represents a single generated test case. This approach was original used for functional testing of mobile phone applications, they concentrate in the approach on the interaction between user and system. In my work I can benefit from this, because I am using black box testing.

- heuristic based test case generation - derives the test cases using ant colony optimization [9], genetic algorithm [5] etc.
  Li and Lam [9] presented an approach based on ant colony optimization to generate the test sequences automatically using the UML Statechart diagrams for state-based software testing. First the statechart diagram is converted into a directed graph, the nodes in the directed graph represent the states of the Statechart, the edges represent transitions between the states. Then they present in their work an algorithm to use the ants to search the directed graph and derive the test cases.
  Doungsa-ard et al. [5] proposed an approach to generate test data from a state machine diagrams using the genetic algorithm. The generated test data are in the form of sequence of triggers, which cause the state transition in the state machine diagram. The more transitions are visited during the testing, the better the test data are. The problem here is to find such set of test data, which cause to visit the most transitions.

- hybrid test case generation - derives the test cases using test sequence generator (TESTOR) algorithm [13], path constraint solving, mutation analysis [1] etc.

  Pelliccione et. al. [13] presented a model based test sequence generator TESTOR. Testor takes behavioral models (sequence diagrams(inSD) and state machine diagrams) as input. Then Testor outputs the test sequences in the form of more complete sequence diagrams (outSD). The outSD contains messages from inSD supplemented by information from state machine diagrams.
  Bandyopadhyay and Gosh [1] presented an approach to generate test cases using the state machines, sequence and class diagrams. First the sequence diagram and class diagrams are used to construct Variable assigment graph (VAG). Then the information from state machine diagrams is used to extend the constructed VAG and build an extended VAG. The paths are selected from extended VAG, [1] use the structure coverage criteria to select the paths. Then the path constraints are

generated, these path constraints together with the information from class diagram is used to create Alloy script. The output of Alloy script are the derived JUnit test cases.

2.2 Related approaches

Tamisier et. al. [16] developed a collaborative model based testing approach. In their approach they use an intermediary layer to process different level of abstraction of the SUT. The intermediary is realized by a Test Bench Scripting Language (TBSL). The model of the system created in UML can be compiled into TBSL, based on the information derived from UML model TBSL allows to create the test sequences.
The experimental trials have shown how the presented collaborative model based testing approach support collaboration during the creation and execution of test sequences. Vertical collaboration allows the teams participating at product development to work together at setting the test sequences. This is allowed because all participating teams are using the same model. Each participating team can set the abstraction level on the model. If someone modifies the model, then all the test sequences are created again and are visible for all participants. The horizontal collaboration is the collaboration within the same team. The presented collaborative model based testing is used in the automotive industry.

2.3 Modeling collaborative structures

There are many existing approaches to model a collaboration of a system.
Nandi et. al. [12] presents Business Entity Definition Language (BEDL), which can be used with existing process-centric technologies like: BPMN and WS-BPEL. Business Entities (BE) are key elements of BEDL. The BEDL enables to describe the BE in XML. The BE have four main components: information model, lifecycle, access policies and notifications. Information model is a set of value, key pairs defined in XML. The lifecycle is modeled as a finite state machine. The access policies component defines, which of the CRUD operations on attribute values defined in information model, can be invoked by which of the participating roles e.g. role X has a capability to update some attribute value. The access policies also define an execution policy, the execution policy means a state transition in the lifecycle model. The notification component enables external parties to subscribe for CRUD events.
They provide in the article an example how business entities defined in BEDL and business processes defined in WS-BPEL can work together. The business processes can invoke various operations on BE:

- request metadata about BE types
- request to access data from a BE instance
- request to query on BE instances

- etc.

For the needs of my thesis I concentrated on the access policies component of BE. As mentioned it enabled to define a CRUD policies for participating roles (including humans), but it's not very practical to use with SN, because there are many big components (e.g. shared artifacts) in the SN, the big components includes many attributes, so I would need to create a access policy for each attribute in the component of SN.
Teruel et. al. present Collaborative Systems Requirements Modelling Language (CSRML) in their work [17]. CSRML is an extension of i* to model Computer Supported Cooperative Work (CSCW) system. The roles defined in CSRML are used to carry out a set of related tasks. The actor playing a role participates in a task. A task can be individual (one role participating), collaboration, coordination, communication (more roles are participating). The CSRML is concentrated on collaboration between users, but does not provide an approach to model system components on which the users are collaborating.
Sungur et. al. [15] present hADL to model collaborative structure. hADL distinguish between human components, collaboration objects and collaboration connector. Human components represents the collaborating user roles. Collaboration objects are the components of the system, the existing human components collaborate through collaboration objects. Collaboration connector helps the human component and collaboration object to fulfill its role. hADL enables to define a set of actions on collaboration object, human component and collaboration component. The actions of various components can be connected with collaboration wire. The collaboration wire between action A_1 of component C_1 and action A_2 of component C_2 has the following semantic: C_1 has the capability to invoke operation A_2 of component C_2, C_1 invokes A_2 by calling A_1.
Since hADL is expressive enough to model a collaboration in SN and it provides a collaboration-centric component-connector view [15], it can be simple mapped to a component-connector diagram of the concrete SN therefore I chose hADL for my thesis.

CHAPTER 3

Methodology

3.1 Creating the abstract social network

In this Section I explained how to create a model of abstract SN based on the given description.
The first part provides a simple description of social network. The second part uses information provided in first part to create a model of abstract SN, this part also shows how the abstract social network can be modeled in hADL.

3.1.1 Description of the abstract social network

I decided to create this description in such a way, that the model based on this description created in next subsection can be applicable for most modern SNs.
Description:
Every user can create a post, each post can have a description and it must contain at least one photo. During creation of a post the user has to choose a subset of his friends, for which this post will be visible. Each user, who can see a post (let's call this user post viewer), can either like or dislike a photo within the post. The post viewer can give only one like(dislike) per photo. The post viewer can also comment at photo within a post. The post viewer has the capability to see all likes, dislikes, comments of each photo within a post. The user, who created a post (let's call him post owner) can update, delete his post, or comments photos within his post, but he cannot like (dislike) photo within his post. Post owner should always be notified, if his post was liked, disliked or commented by a post viewer. The post viewers should be notified if there is a new post, which is visible for them.
If a user(A) want to be a friend with another user(B), then A has to send friend request to B. As soon as B accepts friend request by A, this friend request is deleted and friendship between A and B is established. There can be always at most one friendship (friend

request) between each two users. The user B should always be notified if he receives some friend request. The user A should be notified if B has accepted his friend request.

3.1.2 Modeling the abstract social network

To model an abstract SN, it is necessary to identify the following:

- key components of a SN
- a set of user's roles collaborating at components
- for each component identify
  - a set of actions, which can be performed on a component by other components(including users)
- for each user's role identify
  - a set of capabilities of user's role, the user can interact with other components only by means of these capabilities
- link the user capabilities with the appropriate component's actions

Since we have all these information we can create the model. The hADL model describe the collaboration between participating human components and collaboration components. I can benefit from it, because I need to model the user's roles (as human components in hADL), and system components (as collaboration component in hADL). Applying the above process on the given description I identified the following

- components:
  - Post (PostEntity represents the Post's properties)
  - Photo
  - Like
  - Dislike
  - Comment
  - Friendrequest
  - Friendship
- user's roles
  - Post owner
  - Post viewer
  - Receiver - representing a user, who sent some friend request(s)
  - Sender - representing a user, who received some friend request(s)

– Friend - representing a user, who is a friend with at least one other user

Please note, that a user can have more roles. Let's consider a user, who creates a post and has a friend, the known roles of this user are post owner, friend.
Following the process above I identified for each user's role a set of capabilities, for each component a set of actions and link the user capability with the appropriate component's action.
Using hADL I created the collaboration structures shown in Figures 3.1 and 3.2. All the identified components are modeled as collaboration components, the identified user's roles are modeled as human components. Each user capability is linked with the appropriate component action using the collaboration wire.
Many modern SN support the notification mechanism, therefore I decided to add this mechanism into the abstract model. The users are notified about some changes on the collaborating components. The notifications are realized as hADL messages.
Except the human components and collaboration components, there are other components called collaboration connectors (CreateFriendship connector, CreateFR connector, Like/Dislike connector, Post spam checker, Notification connector) in the collaboration structure. Collaboration connectors helps the components to fulfill its roles [15]. Let me explain the aim of collaboration connector on simple example: Consider CreateFriendship connector. This connector has a read capability at friend request. Everytime when a receiver accepts a friend request, the CreateFriendship connector can see it, creates the new friendship from the friend request, removes the friend request and creates a friendship notification, which notifies the sender of the friend request, that the friend request was accepted.

3.2 Identifying the key use cases of abstract social network

In this Section there is a list of identified key uses cases of abstract SN. I will use these use cases later to generate the concrete tests, therefore it's important to choose such use cases, which I want to test later.
I need to identify use cases, which includes more human components, because I want to test, whether all participating human components are affected as expected by a given use case.

The identified key use cases of "Post" feature:

- creating a post - post is created by post owner and should be visible to selected post viewers
- update a post - post owner updates his post, these changes should be visible for post viewers
- delete a post - post owner deletes his post, this post shouldn't be visible anymore for post viewers

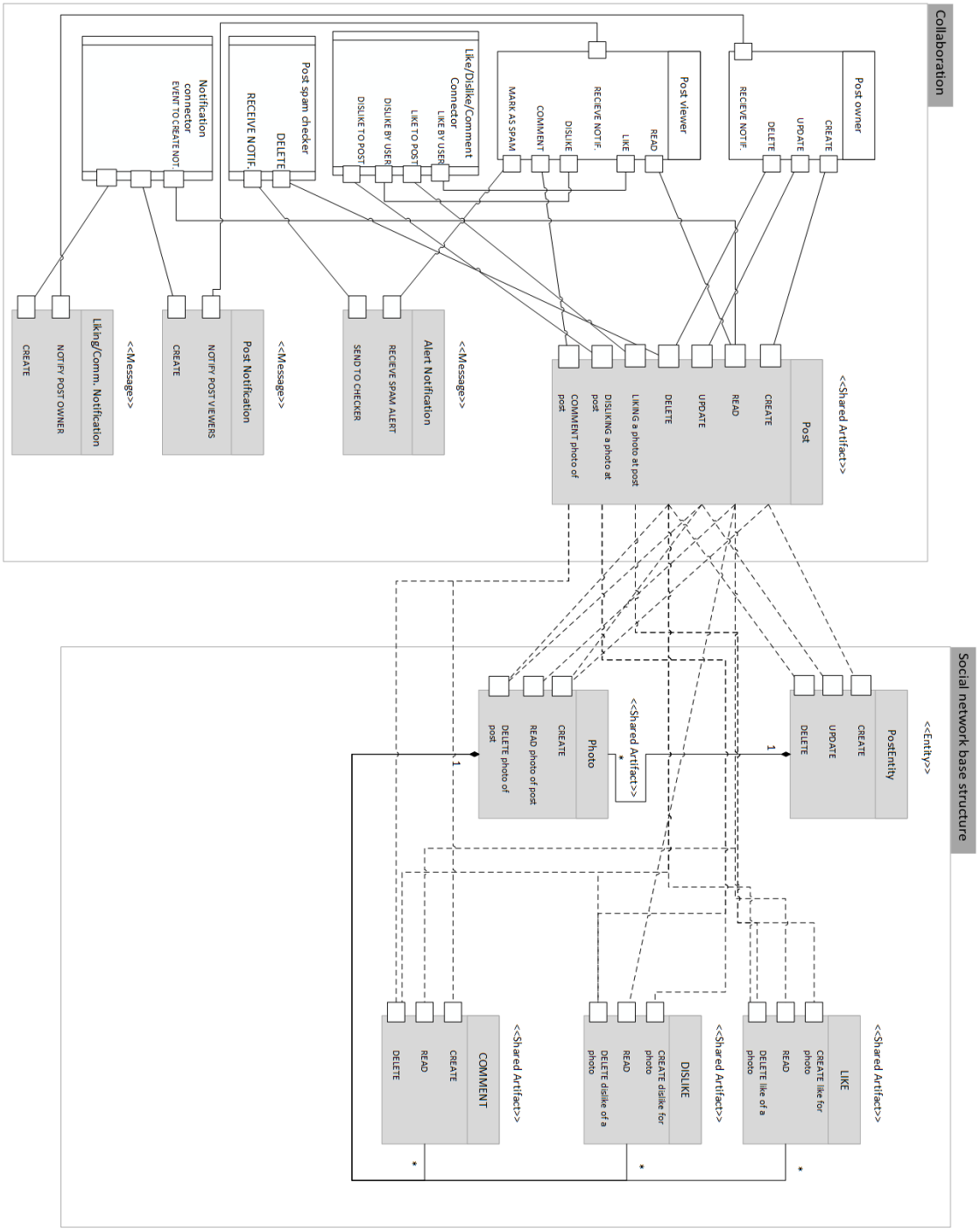


Figure 3.1: hADL diagram showing the collaboration at post with various user's roles

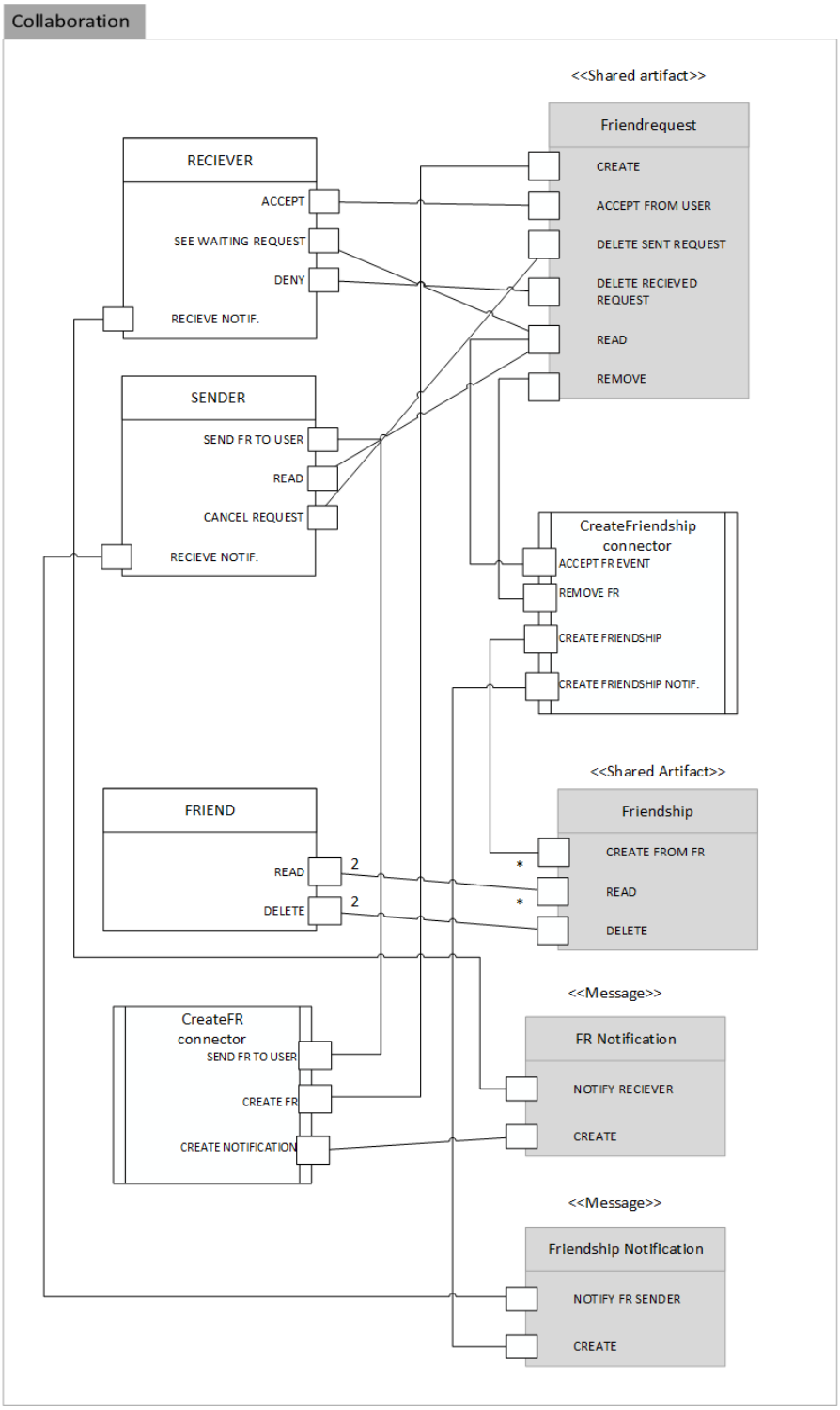


Figure 3.2: hADL diagram showing the collaboration at friend requests & friendships with various user's roles

- like a photo within post - like is visible for post viewers and post owner
- dislike a photo within post - dislike is visible for post viewers and post owner
- comment a photo within post - comment is visible for post viewers and post owner

The identified use cases of "Friendship and Friend request" feature:

- accepting a friend request - new friendship between sender and receiver should be created, friendship is visible for sender and receiver
- sending a friend request - new friend request is sent from sender to receiver, friend request is visible for sender and receiver
- deleting a friend request - friend request is not visible for sender, receiver anymore
- deleting a friendship - friendship is not visible for friends anymore

3.3 Modeling the identified use cases

Because I want to test, whether all participating roles are affected as expected by identified use case. I should identify all participating roles, as well as the expected interaction for each use case. The chosen approach how to describe interaction between participating roles is sequence diagram. For the creation of the sequence diagrams I used the created hADL diagrams shown in Figures 3.1 and 3.2.
I decided to choose the "like a photo within post" and "accepting a friend request" use cases to illustrate the proposed methodology on concrete examples, because these use cases are modeled in most complex sequence diagrams. These sequence diagram are shown in Figures 3.4 and 3.3. All the other sequence diagrams are shown in Appendix.

3.3.1 Like a photo within post

If a post viewer wants to like a photo, he needs to have a read capability on the photo. This read capability is implied by the by read capability on Post, because if a Post viewer has a read capability on a post, then he has the read capability on all its Photos, Likes, Dislikes, Comments. This implication is because of the substructure wires from read action on Post to read actions on Photo, Likes, Dislikes, Comment. Every time a post viewer likes a photo, then the like action is directed to Like/Dislike Connector. Under certain conditions (specified in sequence diagram) the connector invokes the "Liking a photo at post" action (on the Post), then the substructure wire redirect it to "create like for photo" action on Collaboration object Like and creates a new Like. The Notification connector has a read capability on Post, therefore he "sees" if there is a new Like and creates a Liking/Comm. Notification to notify the Post owner.
The sequence diagram 3.4 is created based on this description.

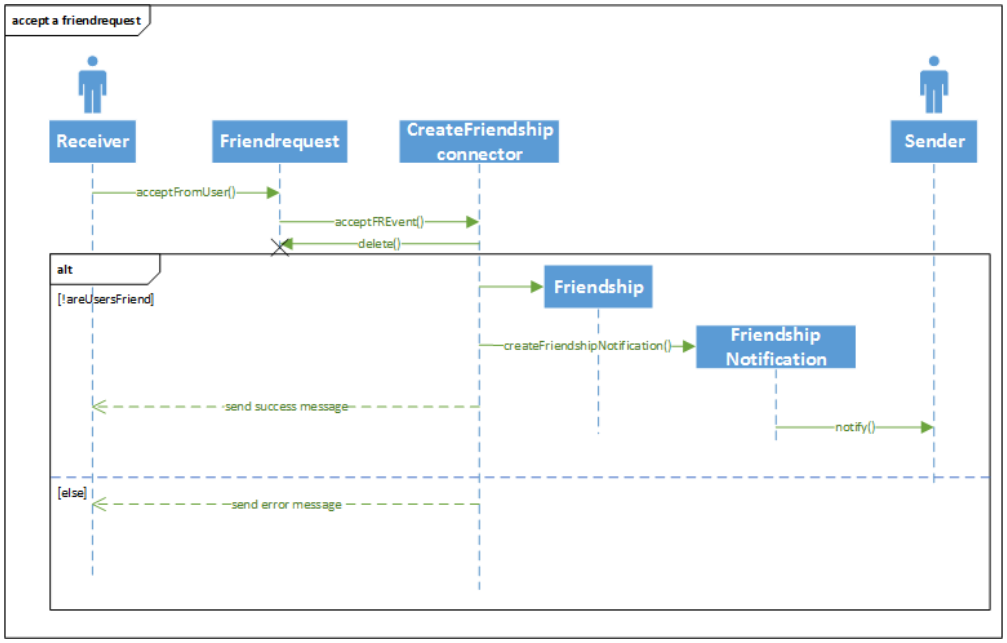


Figure 3.3: sd for accepting friend request

3.3.2 Accepting a friend request

To accept a friend request, a receiver needs to have a read capability on friend request. The read capability is specified by the "see waiting request" action on the side of Human component receiver and by "read" action on the side of Collaboration object Friendrequest. Every time the receiver accepts a Friendrequest, the CreateFriendship connector "sees" it (as Accept FR event), because the CreateFriendship connector has the read capability on the Friendrequest. Then under certain conditions (specified in sequence diagram) the CreateFriendshipConnector removes the Friendrequest(Remove FR action), creates a new Friendship based on the Friendrequest (create friendship action) and creates a Friendship Notification (create friendship notification action) to notify the Sender of Friendrequest, that the Receiver has accepted Friendrequest.
The sequence diagram 3.3 is created based on this description.

3.4 Generating abstract test cases

Based on the sequence diagrams created in the previous Section it's possible to create the abstract test cases.
One of many possible approaches how to generate test cases is the Test Case Generation by means of UML Sequence Diagrams and Labeled Transition System [2]. I chose this approach because it ensures the path coverage of the model used to generate test cases and that is exactly what I want, because the test cases ensuring the path coverage imply

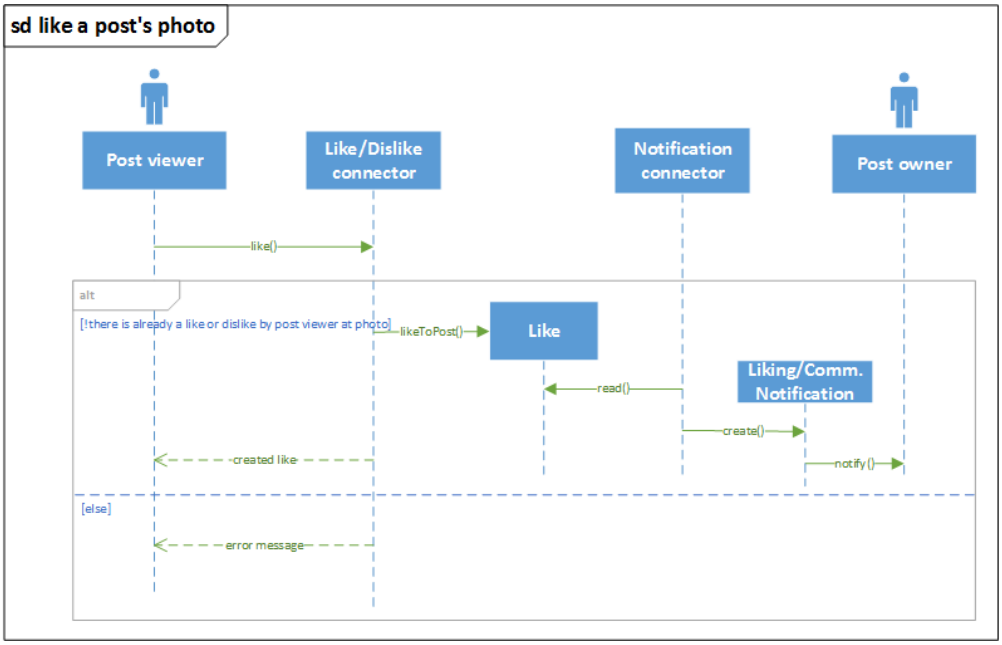


Figure 3.4: sd for liking a photo within a post

that I can determine for each input whether it affects all involved entities as expected. To use this approach it's necessary to follow these steps:

1. obtain LTS model for functional testing : The process of obtaining the LTS I used for my thesis, is the same as described in [2] paper, except the step 4).
   Modified step 4): For each message (in the order it appears) of the sequence diagram, if the message is:

   a) From User to System or from User to User must be create create a transition with the annotation "user input" and a transition where the label is the proper message content. All transitions following the transition annotated as "user input" describe how the system can behave after user input.

   b) From System to User or from System to System - must be create a transition with the label "expectedResults" and a transition where the label is the proper message content. All transitions containing the "expected result" annotation are the expected behavior of the abstract system following user input under given preconditions as well as initial conditions.

2. deriving test cases : To generate test case it's necessary to identify all paths from the LTS. A path can be obtained, using the DFS method, by traversing an LTS starting from the initial state. Each path will result in a test case.

To summarize the LTS used in my thesis.

An LTS is a 4-tuple S = (Q, A, T, q0)

- Q is a finite, nonempty set of states;
- A is a finite, nonempty set of labels (denoting actions);
- T, the transition relation; $T \subseteq Q \times A \times Q$
- q_0 is the initial state

Syntax of transition labels description:
$[roleA : actionName|actionName : roleA|actionName]$ $[annotation]$
$RoleA : actionName$ - denotes that roleA interacts directly with the system through an action actionName (e.g. user input, the initiator of the action is roleA)
$actionName : RoleA$ - denotes that system interacts directly with the roleA through an action actionName (the system notifies roleA about some result)
$actionName$ - action actionName is performed intern in the system (the only participating role is the system itself)

The described approach [2] outputs the set of test cases. A test case is a pair of input and expected output. The test case should give some feedback about the tested feature, but we can only benefit from this feedback, if the test case was executed when the system was in the correct state. Therefore we need for each test case to define a set of preconditions, as well as the set of post conditions.
As already mentioned it's tested here, not only if the system behaves as expected after execution of a test case, but also if all participating roles are affected as expected, therefore it's necessary to identify and describe for each test case all participating roles. To obtain this information we can reference the appropriate model from Section 3.3.

To generate an abstract test case we need to follow these steps:

1. Provide description of tested feature
2. Provide description of all participating roles (referencing a sequence diagram from Section 3.3)
3. Define preconditions
4. Obtain LTS model from UML sequence diagram
5. Obtain paths by traversing the LTS with DFS
6. Derive test cases resulting from paths
7. Define post conditions

In the subsections below follows the generation of abstract test cases for the features: "Liking a photo within post" and "Accepting a friend request". The abstract test cases generated for another features are listed in Appendix.

3.4.1 Liking a photo within post

The tested feature: **"Liking a photo within post"**
Brief description of feature: user denoted as post viewer creates a like for a photo within a post
Description of participating roles:

- like creator: user who is a post viewer and who creates a like
- post viewer: user who can see a post, post viewer has to be different from like creator

Preconditions:

- like creator exists
- post viewer exists
- post exists

Labeled transition system:
The LTS obtained to test this feature is shown in Figure 3.5.
Paths obtained from LTS
The paths are shown in table 3.1.

Table 3.1: path table obtained from LTS in Figure 3.5

Number	**Path**
1	Post viewer: like() (user input), conditions, There is no like AND no dislike by post viewer at photo error message: Post viewer (expected result)
2	Post viewer: like() (user input), conditions, else, likeToPost() (expected result), read(expected result), create(expected result), notify(): Post owner (expected result)

Resulting abstract test cases:
The test cases are shown in tables 3.2 and 3.3. These test cases are derived from path table 3.1.

Table 3.2: Abstract test case 1

Initial condition	There is already a like or dislike by post viewer at photo
User input	**Expected result**
Post viewer: like()	error message: Post viewer

Post conditions:

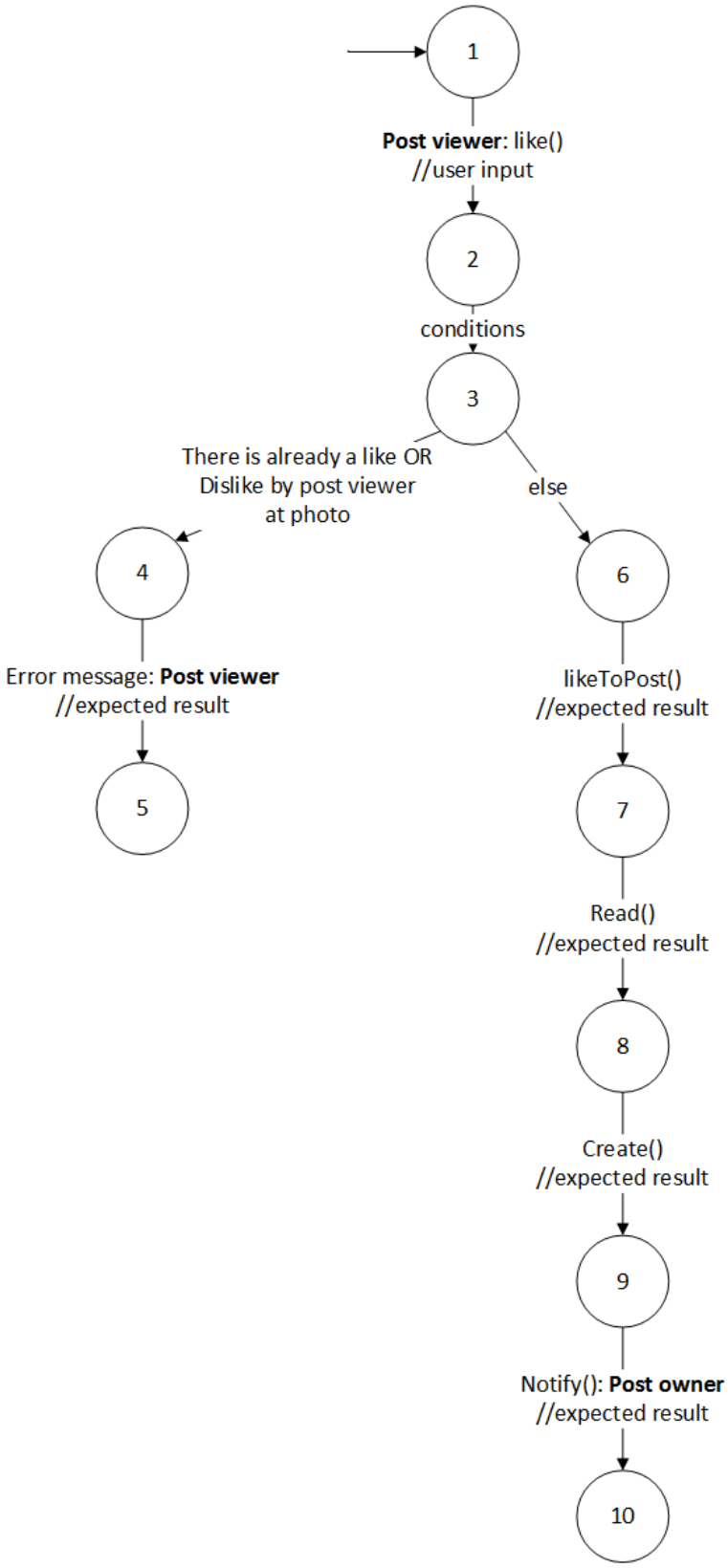


Figure 3.5: LTS derived from sequence diagram 3.4

Table 3.3: Abstract test case 2

Initial condition	There is no like and no dislike by post viewer at photo
User input	**Expected result**
Post viewer: like()	likeToPost(), read() , create(), notify(): Post owner, created like: Post viewer

- like creator exists
- post viewer exists
- post exists

3.4.2 Accepting a friend request

The tested feature: **"Accepting friend request (FR), creation of corresponding friendship"**
Brief description of feature: user denoted as receiver received a friend request from a user denoted as sender, receiver accepts the friend request, so the friendship between receiver and sender should be created.
Description of all participating roles:

- receiver: is a user of the system, who received some friend request
- sender: is a user of the system, who sent some friend request

Preconditions:

- sender sent a friend request to receiver
- friend request exists - From the hADL diagram in 3.2 we know that friend request is shared artifact between human components receiver and sender
- sender exists
- receiver exists
- sender is different from receiver

Labeled transition system:
The LTS obtained to test this feature is shown in Figure 3.6.

Paths obtained from LTS:
The paths are shown in table 3.4.

Abstract test cases resulted from path table:
The test cases are shown in tables 3.5 and 3.6. These test cases are derived from path

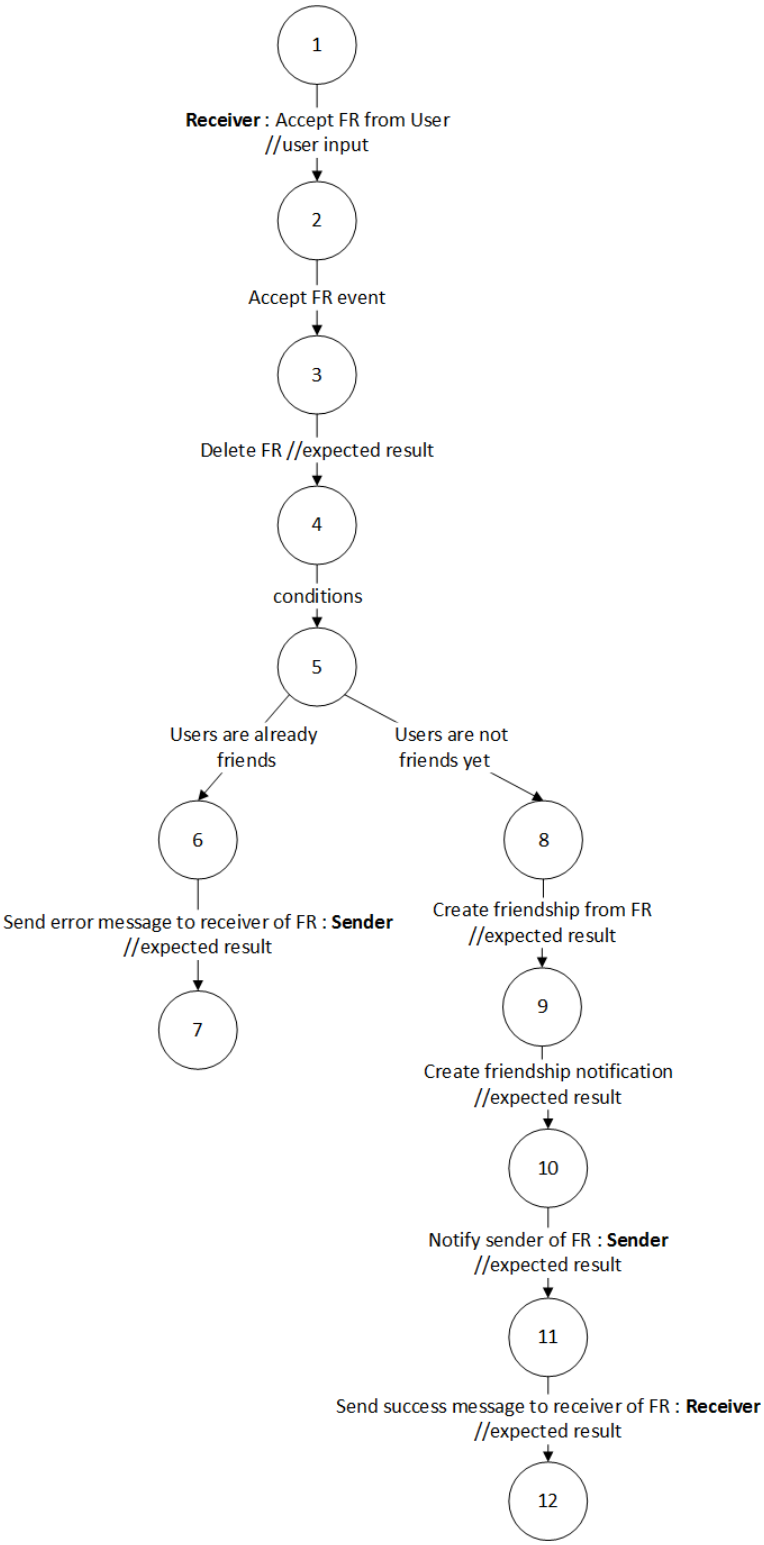


Figure 3.6: LTS derived from sequence diagram 3.3

Table 3.4: path table obtained from LTS in Figure 3.6

Number	Path
1	accept FR from User, accept FR event, delete FR, conditions, Users are already friends, expected result, send error message to receiver of FR
2	accept FR from User, accept FR event, delete FR, conditions, User are not friends yet, create,friendship from FR, create friendship notification, notify sender of FR, send success message to receiver of FR

table 3.4.

Table 3.5: Abstract test case 1

Initial condition	Users are already friends
User input	**Expected results**
Accept FR from User	delete FR, Show error message to receiver of FR

Table 3.6: Abstract test case 2

Initial condition	Users are not friends yet
User input	**Expected results**
Accept FR from User	delete FR, create friendship from FR, create friendship notification, notify sender of FR, send success message to receiver of FR

Post conditions:

- friend request has been deleted
- sender exists
- receiver exists
- sender and receiver are different

3.5 Mapping the model of concrete social network to the abstract one

In this Section I need to provide diagrams of concrete social network (cSN). Because I want to map the abstract test cases generated in Section 3.4 to the concrete ones, which can be executed on the implemented cSN, I need to provide sequence diagrams of cSN, these sequence diagrams will be mapped to the abstract ones and based on this mapping it's possible to derive the concrete test cases. In order to create sequence diagrams of a

system, I need to identify components of the system. To present all components I used the component diagram. Based on this component diagram I can create the sequence diagrams of cSN.

3.5.1 Modeling the concrete social network

The implementation of the backend (modeled in the component diagram 3.7) consists of two layers. The first layer is the Endpoint layer. This layer is an entry point for each incoming request from client. In this layer there is implemented the authentication and authorization of the request's owner. The authentication is implemented using Facebook API as OAuth2.0 provider. If the user is authenticated and authorized to perform an operation, then the request is send to the second layer (Service layer), otherwise an Exception is thrown. The Service layer connects to a MySQL database provided by Google Cloud Platform and performs the desired operation. The results are returned through the Endpoint layer to the client.
Each component xEndpoint (where x $\in$ Likes, Friendrequest, ...) includes at least the following classes: xEndpoint (part of the Endpoint layer), xService (part of the Service layer), x (representing the entity).
Each of the xService class implements the CRUD (ICreatex, IReadx, IUpdatex, IDeletex) interfaces. If a component xEndpoint needs another component yEndpoint to fulfill its operation, then xEndpoint uses a I[C|R|U|D]Y interface to invoke a method of yService. e.g. If a user wants to delete his account, he needs to delete all his posts. Therefore the UserEndpoint component requires the IRemoveSimplePost implemented by SimplePost-Service.
The whole backend runs on Google App Engine. The Google Cloud Endpoints are used as communication endpoints between client and backend.
The component diagram of the backend implementation of cSN is presented in Figure 3.7. For the implementation of the backend refer the Appendix Chapter B.

Description of the cSN

The cSN tested in my thesis provides almost the same functionalities as the abstract SN described in Section 3.1.1. There is only one difference:

If the post has more than one photo, the post viewer:

- can like only one photo per post
- can not dislike a photo within this post

3.5.2 Create a mapping between the hADL and component diagram

Dorn and Taylor [4] presented a Architecture mapping process. A software-to-collaboration mapping consist of

- a set of xADL elements (see component diagram 3.5.1)

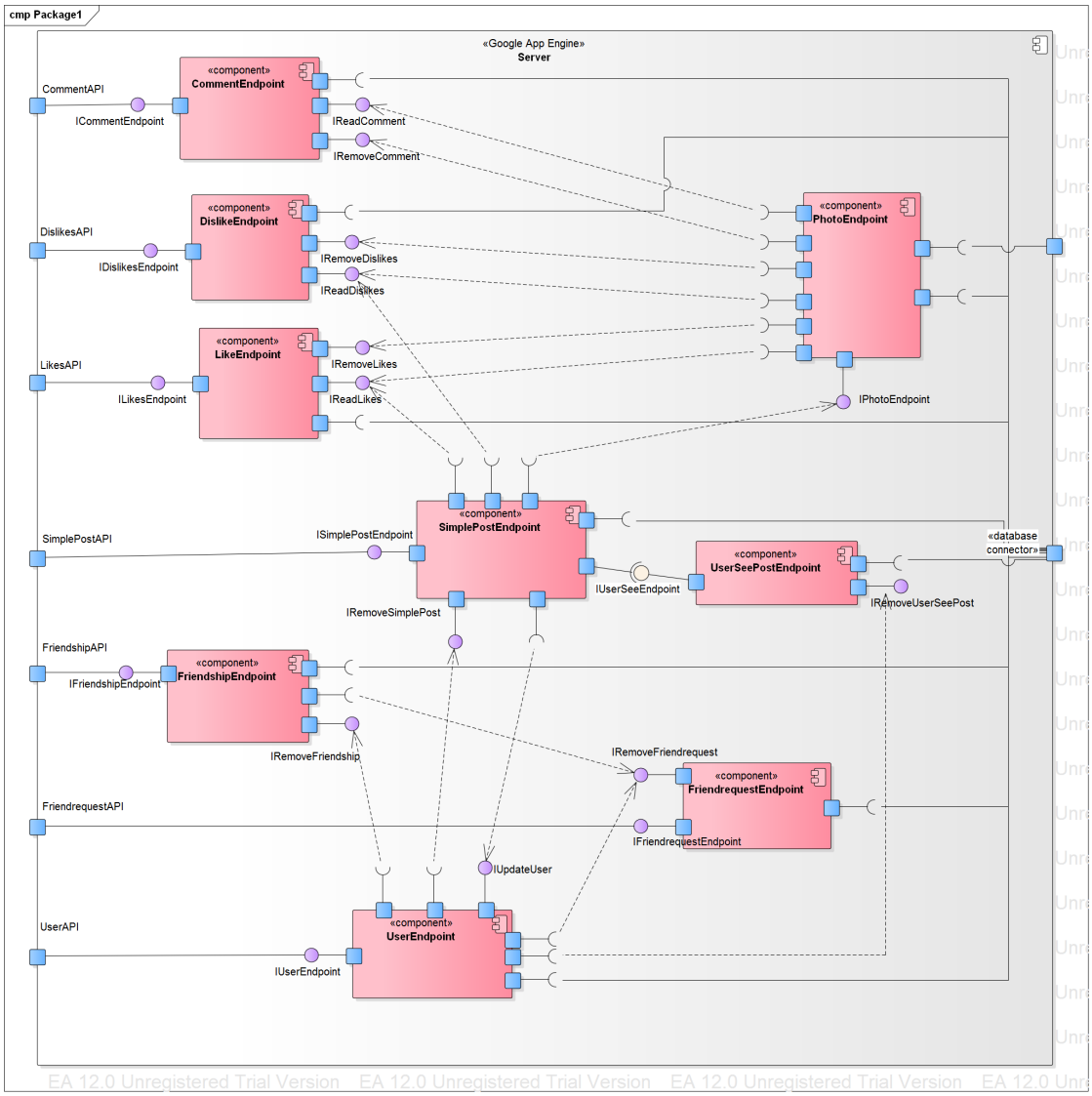


Figure 3.7: component diagram of cSN

- a set of hADL elements (see hADL diagram 3.4)
- a set of Interlock Point pairs: identifies exactly one xADL interface (component diagram interface) and exactly one hADL collaboration action
- the mapping type: determines how many instances of the xADL elements may map to how many instances of the hADL elements. For the mapping I am providing in my thesis holds the following: Each hADL element can be mapped to many instances of xADL (component diagram).

Based on this information to create a mapping I need to identify the component diagram's interfaces and map them to the corresponding hADL diagram's actions.
The resulting mapping is shown in Figure 3.8.

3.5.3 Create a mapping between hADL diagram and system class diagram

During my research as I have been creating the mapping presented in Section 3.5.4, I noticed that the sequence diagrams contain except the system component, some of the system entities. Therefore in order to create a mapping between the sequence diagrams I need to add a mapping between system entities and hADL elements. This mapping will also include the mapping of hADL human components to corresponding entities of the system. The creation of this mapping is based on the semantic of system entities and hADL elements.
The created diagram is shown in Figure 3.9

3.5.4 Create a mapping between sequence diagram of concrete implementation (xSD) and sequence diagram of abstract social network (hSD)

The whole mapping process presented in this Section is divided into two parts:

1. Mapping the interacting components
2. Mapping the messages

To know which message of xSD is mapped to which one in hSD it's necessary to know, which interacting component of xSD is mapped to which one in hSD.

Mapping the interacting components

1. Identify all COMPONENTS presented in xSD. The component diagram 3.5.1 is used to determine which of the interacting components in xSD are the components of the system.
2. Based on the mapping between hADL and component diagram 3.5.2, find the abstract "neighbors" (of COMPONENTS) used in hSD.
3. Identify all ENTITIES presented in xSD. The class diagram 3.5.3 is used to determine which of the interacting components are the entities of the system.
4. Based on the mapping between system's class diagram and hADL, find the abstract "neighbors" (of ENTITIES) used in hSD.
5. From the ENTITIES identify all ROLES presented in xSD (the following holds $ROLES \subseteq ENTITIES$).
   Since the following is known:

- hSD describes an operation of the abstract sytem
- hSD is derived from hADL 3.1.2
- hADL is a collaboration model, which means that hADL contains all participating human components

We can derive that all human components needed for an operation are used (as roles) in hSD. The corresponding operation in xSD should contain the corresponding concrete roles.

6. Find the abstract "neighbors" (of ROLES) used in hSD.

Mapping the messages

It's hard to formulate a procedure, which will output the correct mapping for each corresponding hSD and xSD pairs, because let's consider the following : Having two xSD, although the both describe the same operation, the order of the messages in first xSD does not need to be the same as the second one. Both of these xSDs should be mapped to the same hSD, which is hard when we can have more "different" xSD, which should be mapped to the same hSD.
The optimal rule you have to follow during mapping the messages is to follow the mapping of components and semantic of the messages.
I formulate a set of steps, which should output the mapping of messages. The procedure below iterates through all messages in xSD and map them to the corresponding messages from hSD.

```
1 for each message M in xSD {
2   identify message endpoints (components, entities, roles) xA, xB
    (meaning xA invokes message M of xB)
3   map xA, xB to corresponding hA, hB (component, entity, role)
    from hSD, if there are no corresponding components/entities/roles
    go to next message
4   find the shortest path P through unused messages from hA to hB
5   mark all messages in P as used
6   map all messages in P to M
7 }
```

Applying the process from section 3.5.4 to xSDs

The used operator $\approx$ means "is mapped to".

- **Accepting a friend request**
  Applying the above process to map hSD 3.3 to the corresponding xSD shown in upper part of Figure 3.10˙
  COMPONENTS = FriendshipEndpoint, IRemoveFriendrequest (Notice that IRemoveFriendrequest is an interface implemented by FriendrequestService, which is

part of the FriendrequestEndpoint component. See the description provided in the Section 3.5.1), CloudSQL
ENTITIES = Friendship, User
ROLES = User (in the role of Receiver)
finding the abstract neighbors of COMPONENTS/ENTITIES/ROLES

- FriendrequestEndpoint, since we need only the remove Friendrequest capability, and FriendrequestService implements IRemoveFriendrequest, therefore $IRemoveFriendrequest \approx Friendrequest$
- $FriendshipEndpoint \approx CreateFriendshipConnector$
- $User \approx Receiver$
- $Friendship \approx Friendship$

The resulted mapping is shown in Figure 3.10.

- **Liking a photo within post**

  COMPONENTS = LikesEndpoint, PhotoEndpoint, DislikesEndpoint, CloudSQL
  ENTITIES = User
  ROLES = User
  finding the abstract neighbors COMPONENTS/ENTITIES/ROLES

  - $LikesEndpoint \approx Like/DislikeConnector$
  - PhotoEndpoint has no direct abstract neighbor, but according to the semantic of sequence diagram, it can be mapped to Like/DislikeConnector
  - $DislikesEndpoint \approx Like/DislikeConnector$
  - $User \approx PostViewer$

  The resulted mapping is shown in Figure 3.11.

The other mappings are listed in Appendix.

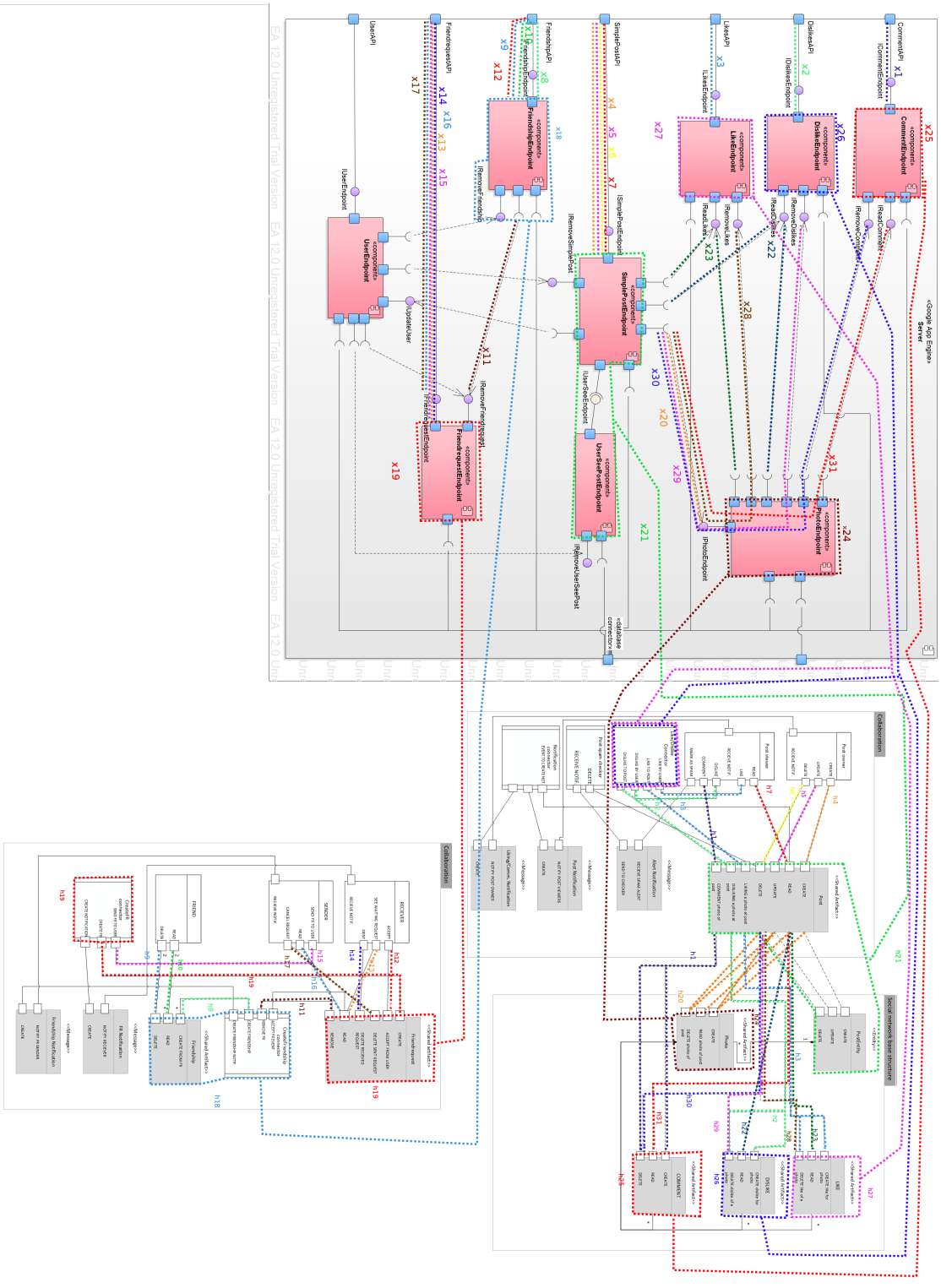


Figure 3.8: mapping between hADL diagram 3.1.2 and component diagram3.5.1

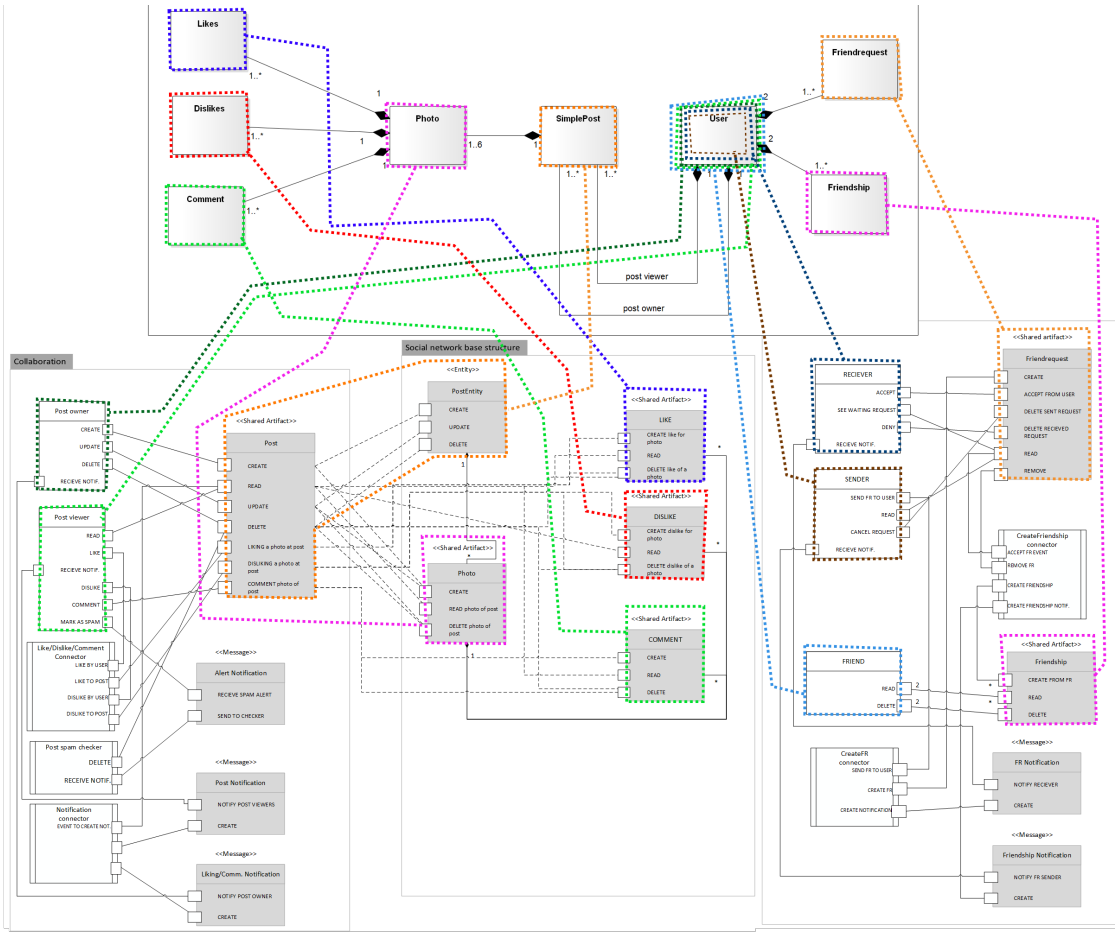


Figure 3.9: mapping between hADL diagram 3.1.2 and class diagram

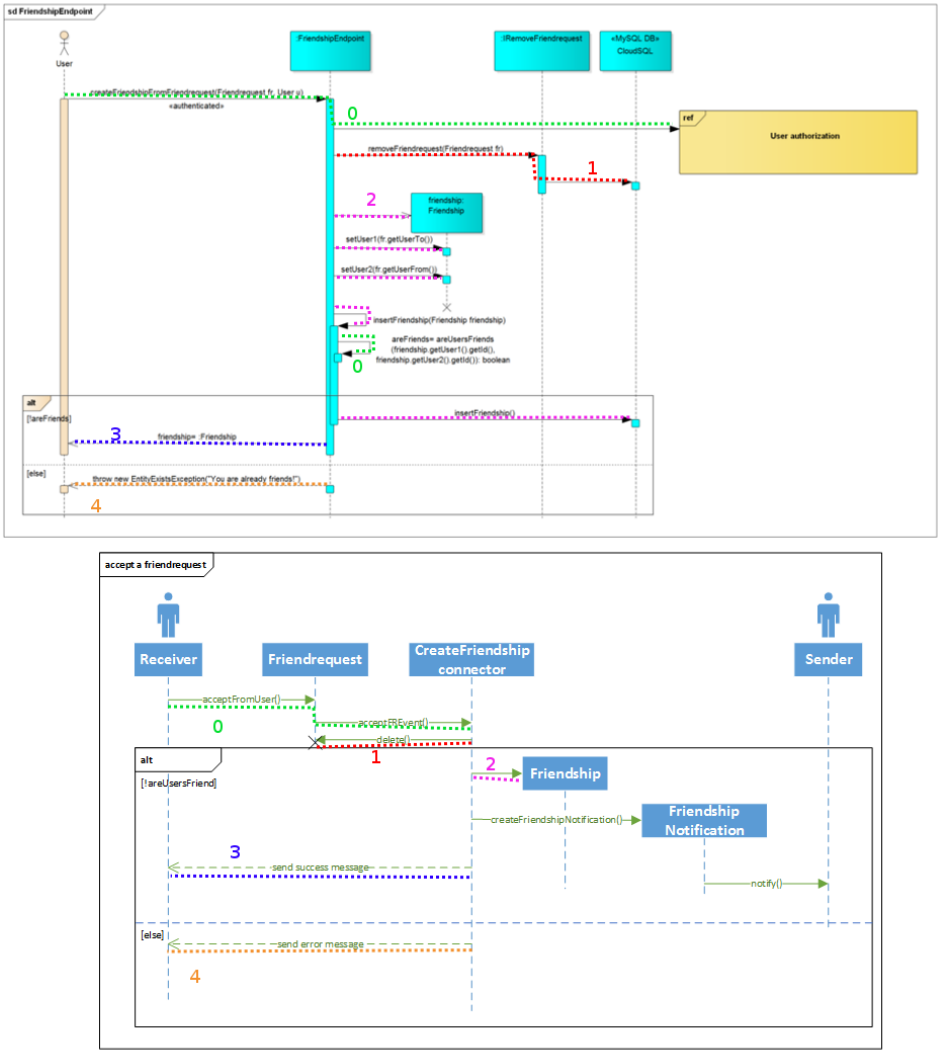


Figure 3.10: The resulted mapping for accept friend request (create friendship) diagram

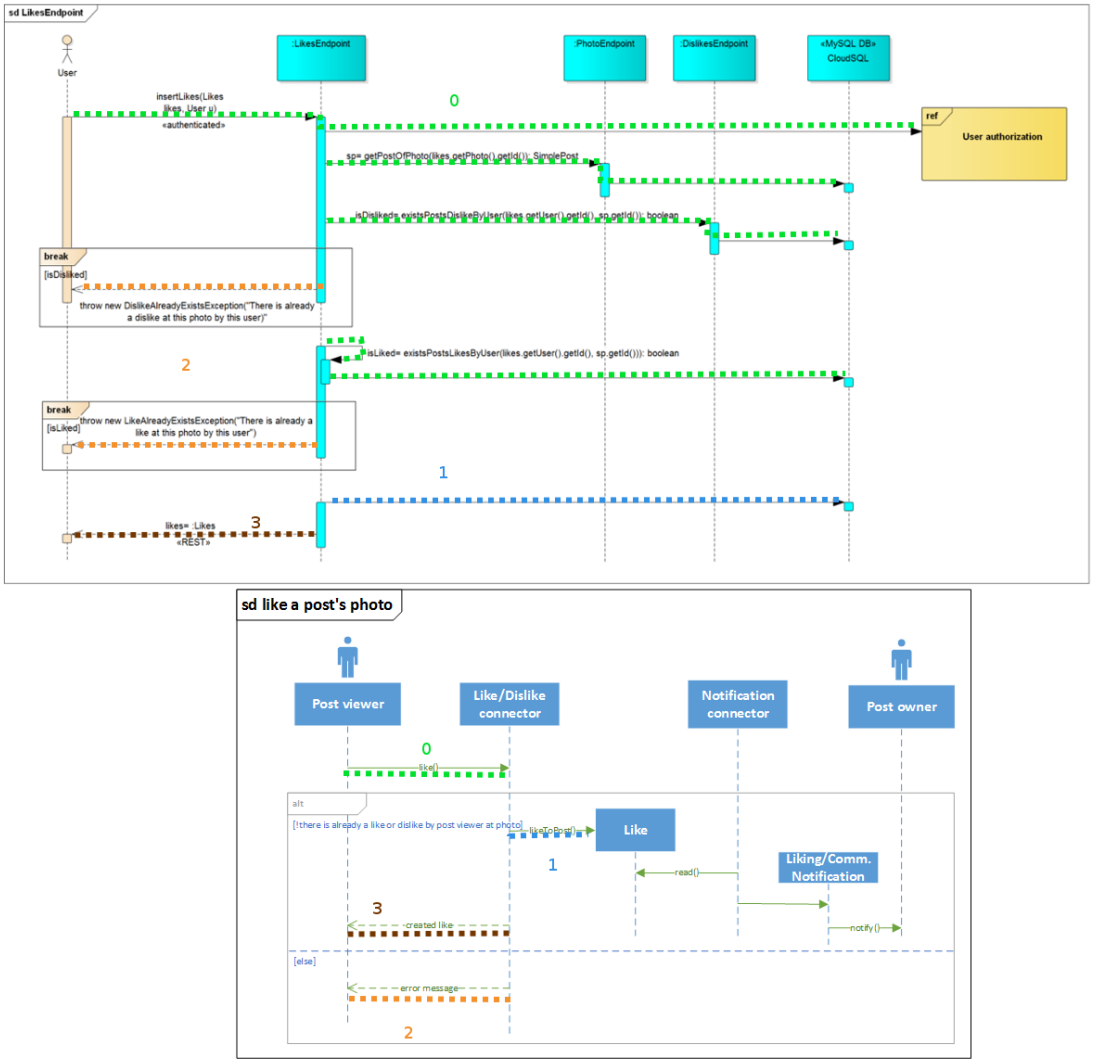


Figure 3.11: The resulted mapping for liking a photo within post feature

CHAPTER 4

Model based testing

In the previous Chapter we obtain enough information to map the abstract test cases generated in Section 3.4 to the concrete ones, which can be executed on the concrete implementation. This Chapter defines a process that derives the concrete executable test cases from the abstract ones.

4.1 Generating concrete test cases and showing expected system results

1. Bring the system in the state, for which it meets the preconditions.
   Map the preconditions given in the abstract test cases to the concrete ones, which can be shown for the concrete implementation.
   Create the concrete preconditions for the system, which implies that the preconditions hold.

2. Use the mapping between xSD and hSD to derive concrete test cases from the abstract ones 3.4

```
1 concreteTestCases = abstractTestCases
2 for each atc in concreteTestCases {
3 M = set of messages invoked in atc
4 for each m_i from M {
5   based on the mapping find n in xSD which
    corresponds to m_i in hSD
6   replace m_i by n
7  }
8 }
9 return concreteTestCases
```

3. Map the initial conditions from abstract test case, to the concrete initial condition, which can be used for concrete test cases.

```
1 inputSet = ∅
2 initialCon = set of all atoms
  (one simple condition without any logical operator)
  used in one initial cond.
3 for each atom C in initialCon {
4   find all messages M in hSD,
    which results influenced C
5   for each message m_i in M {
6     based on mapping xSD-hSD find
      message n in xSD, which is mapped to m_i
7     create the input I for n,
      for which n returns the initial condition
8     inputSet = inputSet ∪ I
9   }
10 }
11 return inputSet
```

4. Show for each test case whether the system returned the result as expected. Since I am using black box testing, I am interested only in messages, which interacts direct with the user.

```
1 for each test case T_i from concreteTestCases{
2  identify the messages M in the expected result of T_i,
   which were sent direct back to user
3  identify roles aR, participating in T_i
   //from abstract test case
4  use the class diagram mapping to map all roles in aR
   to concrete roles R
5  for each message m_j in M {
6    if m_j returns some object O,
7      show that O was created as expected
8      show that O was returned to correct role
9      for every role R, which should be
       affected by O, show that O affects R
       as expected
10     create role uR, so that O shouldn't be visible for uR,
       show that uR has no access to O
11   //"the show part" can be done using assertions,
     //possible with the use of another operations,
     //for which we assume, that they are working correctly
12 }
13 }
```

5. Show for each test case that post condition holds after execution

Map the postconditions given in the abstract test cases to the concrete ones, which can be shown for the concrete implementation.
This can be done using the appropriate mappings created in Section 3.5.

4.2 Applying the process from section 4.1 to abstract test cases generated in section 3.4

All the following test cases test the Service layer of backend implementation. Referencing the architecture description in subsection 3.5.1 I assume that each request incoming to Service layer is authenticated and authorized. Therefore I assume for each test case, that the owner of the request is authenticated and authorized to perform an operation.
Because the presented JUnit test cases test the Service layer, which is only accessible from backend (not as requests from remote client), therefore all JUnit test cases are executed on the server.

4.2.1 Accepting a friend request

1. Bring the system in the state, for which it meets the preconditions.

   a) sender sent a friend request to receiver

   b) friend request exists

   c) sender account exists

   d) receiver account exists

   e) sender and receiver have to be different

   To ensure preconditions c) and d), I need to map the sender/receiver to the corresponding entities of the system. Since I know that (from the class diagram mapping 3.5.3) both sender and receiver are mapped to User entity, which represents users of the system. The preconditions say that sender and receiver exist and we know that before executing each test case, we have a empty DB, therefore I need to create sender and receiver.
   The code below creates the sender and receiver in the system.

```
EntityManager mgr;
//initialization of mgr
UserService userService = new UserServiceImpl(mgr);

User sender = new User("senderName", "sender@somemail.com",
    "pass123");
User receiver = new User("receiverName",
    "reciever@somemail.com", "pass234");
sender = userService.insertUser(sender);
receiver = userService.insertUser(receiver);
```

Table 4.1: Concrete test case 1

Initial condition	Users are not friends yet
User input	**Expected results**
createFriendshipFromFriendrequest (Friendrequest fr, User u) - mapping 0	removeFriendrequest(fr) - mapping 1, throw new EntityExistsException("You are already friends!") - mapping 4

Table 4.2: Concrete test case 2

Initial condition	Users are not friends yet
User input	**Expected results**
createFriendshipFromFriendrequest (Friendrequest fr, User u) - mapping 0	removeFriendrequest(fr) - mapping 1, create friendship from FR (mapping 2 includes declaration and initialization of corresponding Friendship entity), insertFriendship(friendship) - mapping 2, return friendship - mapping 3

To ensure preconditions a) and b) we need to know how to create and send friend request from sender to receiver. To create Friendrequest object, we need to know the system's entity, to which friend request is mapped (from class diagram mapping created in Section 3.5.3 we know that friend request ≈ Friendrequest). Based on these information, we can create the Friendrequest in the system.

```
FriendrequestService friendrequestService = new
    FriendrequestServiceImpl(mgr);

Friendrequest f1 = new Friendrequest(sender, receiver);
friendrequestService.insertFriendrequest(f1);
```

2. Use the mapping shown in Figure 3.10 to derive concrete test cases from the abstract ones generated in Section 3.4.2

   The resulting concrete test cases are shown in tables 4.1 and 4.2.

3. Map the initial conditions from abstract test case, to the concrete initial condition, which can be used for concrete test cases.

   Test case1:
   initialCon = Users are already friends
   C = Users are already friends
   C was used in alt operator [!areUsersFriends, else]

C was influenced only by result of M = *areSenderReceiverFriends()*
m_1 = areSenderReceiverFriends()
n = areUsersFriends()
Create the input I for *areUsersFriends()*, for which *areUsersFriends()* returns the initial condition (Users are already friends).
This is the case if and only if there exists a Friendship between Sender and Receiver, so we need to create Friendship.

```
FriendshipService friendshipService = new
    FriendshipServiceImpl(mgr);

Friendship f = new Friendship(sender, receiver);
friendshipService.insertFriendship(f);
```

Test case2:
initialCon = Users are not friends yet
C = Users are not friends yet
C was used in alt operator [!areUsersFriends, else]
C was influenced only by result M = *areSenderReceiverFriends()*
m_1 = areSenderReceiverFriends()
n = areUsersFriends()
Create the input I for *areUsersFriends()*, for which *areUsersFriends()* returns the initial condition (Users are not friends yet).
This is the case if and only if there exists no Friendship between Sender and Receiver, since we execute every test case with empty DB and we didn't created a Friendship for this scenario, we don't need to do anything.
inputSet = $\emptyset$

4. Show for each test case whether the system returned the result as expected.

   Test case1:
   user input = createFriendshipFromFriendrequest(Friendrequest fr, User u)
   M = throw new EntityExistsException("You are already friends!")
   aR = Sender, Receiver. *This is known from the description of abstract test case.*
   R = User sender, User receiver. *This is known from aR and 3.5.3*
   This is what is needed to be shown:

   - *Show that O was returned to corresponding role*: EntityExistsException is returned to sender (1).

   //anotate the test case with
   @Test(expected = EntityExistsException.class)

   Test case2:
   user input = createFriendshipFromFriendrequest(Friendrequest fr, User u)

M = return friendship
aR = Sender, Receiver. *This is known from the description of abstract test case.*
R = User sender, User receiver. *This is known from aR and 3.5.3*
O = friendship
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: friendship was returned to receiver(1).
- *Show that O was created as expected*: (Since we know from step 3) friendship is Friendship between User sender and User receiver(2).
- *For every role R, which should be affected by O, show that O affects R as expected*: friendship must be visible for both sender and receiver (3).
- *Create role uR*: create User unauthorized and show that friendship is not visible for unauthorized (4).

```
/*
 * (1)
 */
Friendship friendship =
    friendshipService.createFriendshipFromFriendrequest(f1);

/*
 * (2)
 */
assertTrue((friendship.getUser1().equals(receiver) ||
    friendship.getUser1().equals(sender))
          && (friendship.getUser2().equals(receiver) ||
              friendship.getUser2().equals(sender)));

/*
 * (3)
 */
assertTrue(friendshipService.listFriendshipsOfUser(sender.getId())
   .getItems().contains(friendship));
assertTrue(friendshipService.listFriendshipsOfUser(receiver.getId())
   .getItems().contains(friendship));

/*
 * (4)
 */
User unauthorized = new User("unauth", "unauth");
unauthorized = userService.insertUser(unauthorized);

assertFalse(friendshipService.listFriendshipsOfUser(
     unauthorized.getId()).getItems().contains(friendship));
```

---

5. Show for each test case that post condition holds after execution.

   - friend request from preconditions has been deleted
     Since *friend request* ≈ *Friendrequest*, I need to show that the Friendrequest is no more in DB.

     ```
     assertTrue
         (friendrequestService.getFriendrequest(f1.getId()) ==
         null);
     ```

   - sender from pre condition exists
     Referencing the class diagram mapping created in Section 3.5.3, I know that Sender is mapped to User sender. I need to show that the User sender is in DB.

     ```
     assertTrue
         (userService.getUser(sender.getId()).equals(sender));
     ```

   - receiver from pre condition exists
     Using the same principle as in "sender from pre condition exists"

     ```
     assertTrue
         (userService.getUser(receiver.getId()).equals(receiver));
     ```

   - sender is different from the receiver

     ```
     assertFalse (sender.equals(receiver));
     ```

   The complete derived JUnit test cases are available in Appendix's Section B.2.1.

4.2.2 Liking a photo within post

- Use the mapping shown in Figure 3.11 to derive concrete test cases from the abstract ones generated in Section 3.4.1
  The resulting concrete test cases are shown in tables 4.3 and 4.4.

- Map the initial conditions from abstract test case, to the concrete initial condition

  initialCon = There is already a like by post viewer at photo, There is already a dislike by post viewer at photo.
  Since there are two *atoms* in the *initialCon* we need to execute the test case 4.3 two times. Each time one *atom* in *initialCon* is set to true.

- Show for each test case whether the system returned the result as expected

Table 4.3: Concrete test case 1

Initial condition	There is already a like or dislike by post viewer at photo
User input	**Expected result**
insertLikes(Likes likes, User u) - mapping 0	throw new DislikeAlreadyExistsException ("There is already a dislike at this photo by this user") - mapping 2 OR throw new LikeAlreadyExistsException ("There is already a like at this photo by this post") - mapping 2

Table 4.4: Concrete test case 2

Initial condition	There is no like and no dislike by post viewer at photo
User input	**Expected result**
insertLikes(Likes likes, User u) - mapping 0	insert Like to DB - mapping 1, return likes; - mapping 3

Test case1:
initial condition = There is already a like by post viewer at photo
user input = insertLikes(Likes likes, User u)
M = throw new LikeAlreadyExistsException("There is already a like at this photo by this post")
aR = like creator, post viewer. *This is known from the description of abstract test case.*
R = User likeCreator. *There is no need to create User postViewer, because exception should be thrown before it could be shown that Like likes is visible to postViewer.*
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: LikeAlreadyExistsException is returned to the post viewer (comment creator) (1).

//anotate the test case with
@Test(expected = LikeAlreadyExistsException.class)

Test case2:
initial condition = There is already a dislike by post viewer at photo user input = insertLikes(Likes likes, User u)
M = throw new DislikeAlreadyExistsException("There is already a dislike at this photo by this post")

aR = like creator, post viewer. *This is known from the description of abstract test case.*
R = User likeCreator. *There is no need to create User postViewer, because exception should be thrown before it could be shown that Like likes is visible to postViewer.*
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: DislikeAlreadyExistsException is returned to the post viewer (comment creator) (1).

//anotate the test case with
@Test(expected = DislikeAlreadyExistsException.class)

Test case3:
user input = insertLikes(Likes likes, User u)
M = return likes
aR = like creator, post viewer. *This is known from the description of abstract test case.*
R = User likeCreator, User postViewer
O = likes
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: likes was returned to the post viewer(like creator)(1).
- *Show that O was created as expected*: likes is Like at photo of a post(2).
- *For every role R, which should be affected by O, show that O affects R as expected*: likes must be visible for post viewers of likes' post (3).
- *Create role uR*: Create a User unauthorized, who is not a post viewer and show that he cannot create likes at post, which is not visible for him (4) - I show this in the Test case 4.

Test case4:
user input = insertLikes(Likes likes, User u)
M = throw new UnauthorizedDisLikeException("Unauthorized like create");
aR = non-post viewer
R = User nonPostViewer
This is what is needed to be shown:

- UnauthorizedDisLikeException was returned to User nonPostViewer

The complete derived JUnit test cases are available in Appendix's Section B.2.2.

The procedure and results of applying the process on other features are shown in Appendix.

CHAPTER 5

Critical reflection

In this Chapter I present the results of the concrete test cases created in Section 4.2. For each failed test case I provide the description how I find the reason of the failure. I also add suggestions how to prevent the failure. I evaluate here the presented process 4.1 and conclude whether my social network meets the abstract model.

5.1 Evaluation of the testing

In this Section I provide information about changes, which should be done to improve the implementation. These information are based on the results of the execution of the test cases derived in Section 4.2

The next subsections shows the test cases, which discover some failures and describe the reason of the failure. In the subsections 5.1.1, 5.1.3 and 5.1.4 I made some small modifications on the test cases and have observed how the system behaves.

5.1.1 Sending a friend request

Although the test case was successful and didn't discover any failure. I would like to know what happens if the Users sender and receiver are already friends, this is the case if and only if there is a Friendship between User sender and User receiver and User sender (receiver) sends Friendrequest to receiver (sender). Since there is no sense to create Friendrequest if the users are already friends, the system shouldn't allow this.
The test case, which answer this question is nearly the same as concrete Test case 2 in A.1, there is only one difference: there is a need to add a initial condition, saying that there is friendship between sender and receiver.

Creating the initial condition

```
/*
     * creating friendship between sender and receiver
     */
    Friendship friendship = new Friendship(sender, receiver);
    friendship = friendshipService.insertFriendship(friendship);
```

The result of this test case shows that, it's possible to create Friend request between Users, who are already friends.

Reason of this failure:
Since the concrete test cases are derived from the abstract one A.1. I looked at the sequence diagram, which was used to create A.1. The sequence diagram A.1 doesn't contain any fragment, which handles the behavior, if the users are already friends. Therefor the reason for this failure is an inaccuracy of the sequence diagram.

5.1.2 Updating and deleting a post

The test cases, where I removed some photos of the SimplePost and invoked the update operation have shown a failure. The reason of the failure was that the Photos weren't found in the Cloud storage. Since one of the preconditions for these test cases was that the post must exists, therefore I need to create a post before each test case. The information how to create a post is in the mapping A.12. The problem is that the createPhoto() operation is mapped only to the insertPhoto() operation, which insert the Photo metadata (id, path to photo, ...) to the DB, but the createPhoto() operation should be also mapped to the insertPhoto() operation, which inserts the photo (as binary object) to the Cloud storage.
The same was also a reason for the failure at deleting a post test cases.
But if I ignore the part that the photos should also be added to the Cloud storage, then the test cases didn't show any failure, so I can derive that the interaction between the client and database works as expected for updating and deleting a post.

5.1.3 Liking a photo within post

Referencing the description 3.5.1, I would like to know how the system behaves if a post viewer tries to like more photos within one post. Referencing the mapping shown in Figure 3.11 it doesn't seem, that the system will forbid it.
I created the JUnit test case B.2.11, which should answer the question.
The implementation of the test case is the same as Test case 3 in 4.2.2, the only differences are:

- create SimplePost containing more photos (1)
- create one more Like (2)

This test case has shown, that the system allows to create like to multiple photos within one post by one post viewer, which violates the description 3.5.1.

Reason of this failure By analyzing the mapping shown in Figure 3.11 was the failure easy to expect. The reason for this failure is an inaccuracy in design. Since the implementation was created according to the designed models, it follows that neither the implementation is able to handle the described behavior correct.

5.1.4 Disliking a photo within post

Referencing the description 3.5.1, I would like to know how the system behaves if a post viewer tries to dislike a photo within post, which contains more photos. Referencing the mapping A.21 it doesn't seem, that the system will forbid it.
I created the JUnit test case B.2.12, which should answer the question.
This test case has shown, that the system allow to create dislike at photo within one post with more photos, which violates the description 3.5.1.

Reason of this failure By analyzing the mapping shown in Figure A.21 was the failure easy to expect. The reason for this failure is an inaccuracy in design. Since the implementation was created according to the designed models, it follows that neither the implementation is able to handle the described behavior correct.

5.2 Evaluation of the process used for test case generation

The test cases generated by the process of test cases generation used in this thesis have shown some failures, but the process itself has its own limitations.

- Abstract test cases doesn't show how the system should behave if some unauthorized role want to perform operation.
  The sequence diagrams 3.3 used to generate the abstract test cases show only the interaction between authorized roles and the system. Therefore I needed to add the line 10 in step 4 in 4.1, which shows that unauthorized roles haven't access to objects.

- The sequence diagrams of the abstract social network 3.3 are too simple.
  If we want, that the test cases show more failures, then the sequence diagrams should be more detailed, meaning that they should describe more possible interactions, then we will obtain more paths from the sequence diagram which implies more test cases. But if the sequence diagram would be much detailed then it cannot be mapped to many concrete implementations.

The methodology that I developed to generate test cases from collaboration models helped me in finding some bugs and limitations of my implementation of a social network. The developed methodology has also shown its own limitations, which lead to that, that the testing process wasn't able to find all existing bugs.

Bibliography

[1] A. Bandyopadhyay and S. Ghosh. Test input generation using uml sequence and state machines models. *Proceedings of the International Conference on Software Testing Verification and Validation, 2009.*, pages 121 – 130, 2009.

[2] E.G. Cartaxo, F.G.O. Neto, and P.D.L. Machado. Test case generation by means of uml sequence diagrams and labeled transition systems. pages 1292–1297, Oct 2007.

[3] T.T. Dinh-Trong, S. Ghosh, and R.B. France. A systematic approach to generate inputs to test uml design models. In *Software Reliability Engineering, 2006. ISSRE '06. 17th International Symposium on*, pages 95–104, Nov 2006.

[4] Christoph Dorn and Richard N. Taylor. Coupling software architecture and human architecture for collaboration-aware system adaptation. pages 53–62, 2013.

[5] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart. Test data generation from uml state machine diagrams using gas. *International Conference on Software Engineering Advances, ICSEA 2007*, page 47, 2007.

[6] Hyungchoul Kim, Sungwon Kang, Jongmoon Baik, and Inyoung Ko. Test cases generation from uml activity diagrams. In *Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing, 2007. SNPD 2007. Eighth ACIS International Conference on*, volume 3, pages 556–561, July 2007.

[7] Y.G. Kim, H.S. Hong, D.-H. Bae, and S.D. Cha. Test cases generation from uml state diagrams. *Software, IEE Proceedings* -, 146(4):187–192, Aug 1999.

[8] Bao-lin Li, Zhi-shu Li, Li Qing, and Yan-Hong Chen. Test case automate generation from uml sequence diagram and ocl expression. In *Computational Intelligence and Security, 2007 International Conference on*, pages 1048–1052, Dec 2007.

[9] H. Li and C. Lam. An ant colony optimization approach to test sequence generation for state-based software testing. *Fifth International Conference on Quality Software, 2005. (QSIC 2005)*, pages 255 – 262, September 2005.

[10] Li Liuying and Qi Zhichang. Test selection from uml statecharts. In *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 31. Proceedings*, pages 273–279, 1999.

[11] Chen Mingsong, Qiu Xiaokang, and Li Xuandong. Automatic test case generation for uml activity diagrams. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 2–8, New York, NY, USA, 2006. ACM.

[12] P. Nandi, D. Koenig, S. Moser, R. Hull, V. Klicnik, S. Claussen, M. Kloppman, and J. Vergo. Introducing business entities and the business entity definition language (bedl). *Data4BPM*, April 2010.

[13] P. Pelliccione, H. Muccini, A. Bucchiarone, and F. Facchini. Testor: Deriving test sequences from model-based specifcations. *CBSE*, 3489:267 – 282, 2006.

[14] Mahesh Shirole and Rajeev Kumar. Uml behavioral model based test case generation: A survey. *SIGSOFT Softw. Eng. Notes*, 38(4):1–13, July 2013.

[15] C.Timurhan Sungur, Christoph Dorn, Schahram Dustdar, and Frank Leymann. Transforming collaboration structures into deployable informal processes. 9114:231–250, 2015.

[16] Thomas Tamisier, Hind Bouzite, Christophe Louis, Yves Gaffinet, and Fernand Feltz. Model based testing for horizontal and vertical collaboration in embedded systems development. In Yuhua Luo, editor, *Cooperative Design, Visualization, and Engineering*, volume 5738 of *Lecture Notes in Computer Science*, pages 293–296. Springer Berlin Heidelberg, 2009.

[17] MiguelA. Teruel, Elena Navarro, Víctor López-Jaquero, Francisco Montero, and Pascual González. Csrml: A goal-oriented approach to model requirements for collaborative systems. In Manfred Jeusfeld, Lois Delcambre, and Tok-Wang Ling, editors, *Conceptual Modeling – ER 2011*, volume 6998 of *Lecture Notes in Computer Science*, pages 33–46. Springer Berlin Heidelberg, 2011.

[18] Mark Utting. Position paper: Model-based testing.

APPENDIX A

Appendix

A.1 Sending a friend request

A.1.1 Abstract part

The sequence diagram is shown in Figure A.1.

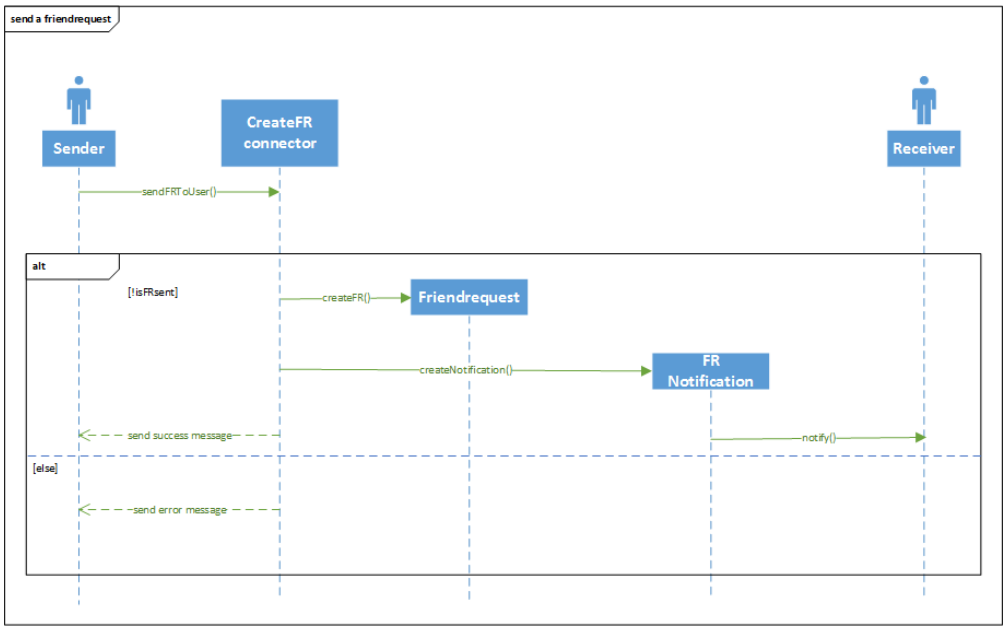


Figure A.1: sd for sending friend request

Test case

The tested feature: **"Sending a friend request"**
Brief description of feature: user denoted as sender sends a friend request to a user denoted as receiver
Description of participating roles:

- sender: is a user of the system, who sent some friend request
- receiver: is a user of the system, who received some friend request

Preconditions:

- sender exists
- receiver exists
- sender and receiver are different

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.2.
Paths obtained from LTS
The paths are shown in table A.1

Table A.1: path table obtained from LTS in Figure A.2

Number	Path
1	User input, send FR to User, conditions, Such Friend request already exists, expected result, show error message to sender
2	Create FR, create FR Notification, notify receiver, send success message to sender

Resulting abstract test cases:
The test cases are shown in tables A.2 and A.3. These test cases are derived from path table A.1.

Table A.2: Abstract test case 1

Initial condition	Friend request between sender and receiver already exists
User input	**Expected results**
Send FR to User	Show error message,to sender

Post conditions:

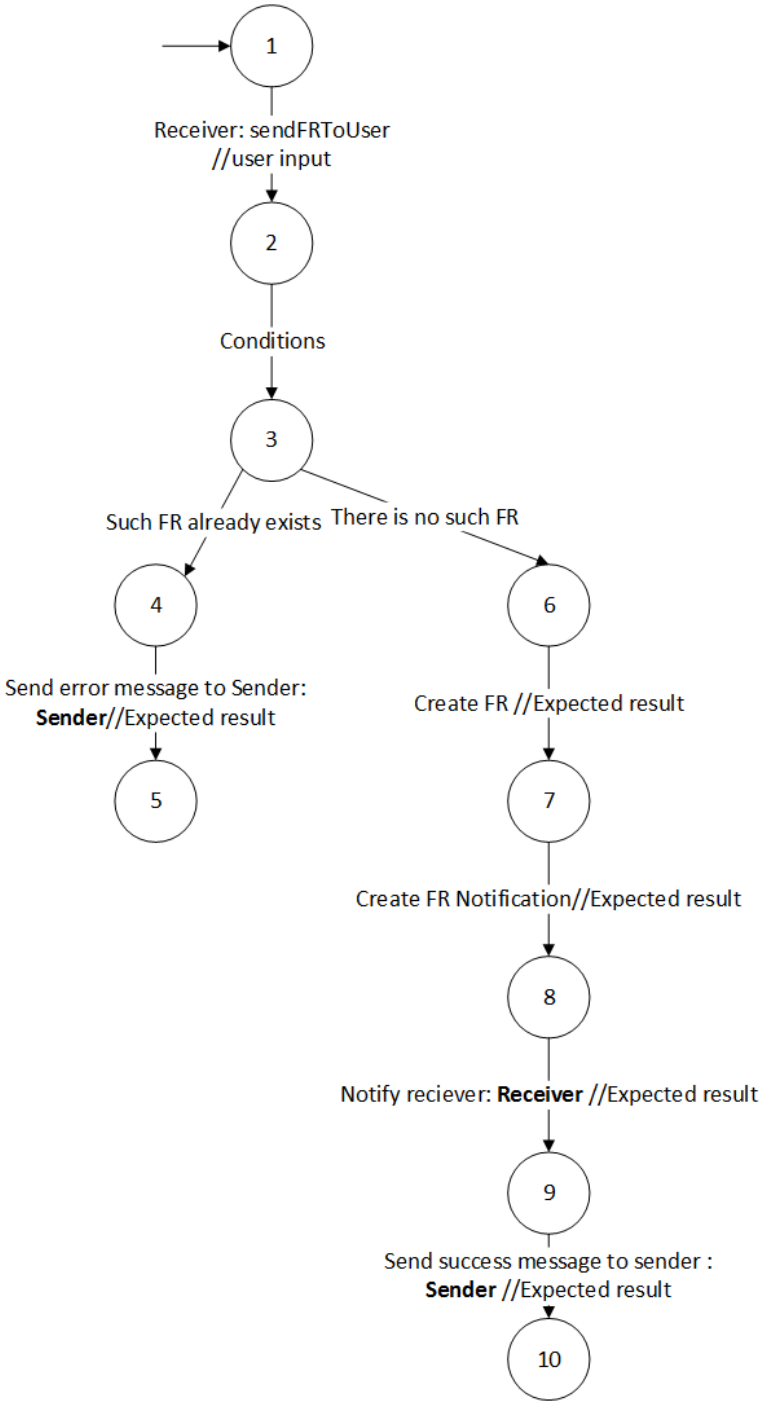


Figure A.2: LTS derived from A.1

Table A.3: Abstract test case 2

Initial condition	There is no friend request between sender and receiver
User input	**Expected results**
Send FR to User	Create FR, create FR Notification, notify receiver, send success message to sender

- sender exists
- receiver exists
- sender and receiver are different

A.1.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.3.

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping shown in Figure A.3 to derive concrete test cases from the abstract ones generated in Section A.1
  The resulting concrete test cases are shown in tables A.4 and A.5.

Table A.4: Concrete test case 1

Initial condition	Friend request between sender and receiver already exists
User input	**Expected result**
insertFriendrequest(Friendrequest f, User u) - mapping 0	throw new EntityExistsException ("Friendrequest has been already sent!"); - mapping 3

Table A.5: Concrete test case 2

Initial condition	There is no friend request between sender and receiver
User input	**Expected result**
insertFriendrequest(Friendrequest f, User u) - mapping 0	insertFriendrequest(f) - mapping 1, return f - mapping 2

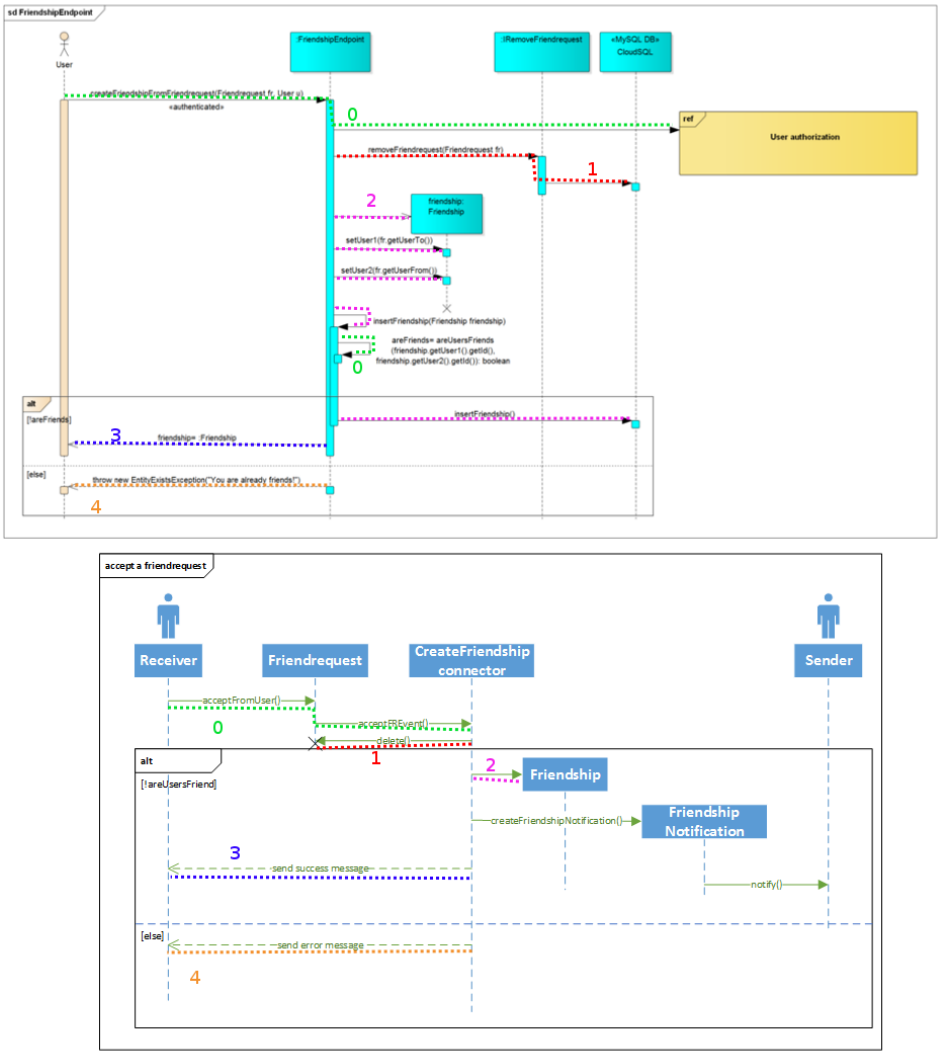


Figure A.3: The resulted mapping for sending a friend request feature

- Show for each test case whether the system returned the result as expected

  Test case1:
  user input = insertFriendrequest(Friendrequest fr, User u)
  $M =$ throw new EntityExistsException ("Friendrequest has been already sent!");
  aR = Sender, Receiver. *This is known from the description of abstract test case.*
  R = User sender, User receiver
  This is what is needed to be shown:

  - *Show that O was returned to corresponding role*: EntityExistsException is returned to sender (1).

  //anotate the test case with
  @Test(expected = EntityExistsException.class)

  Test case2:
  user input = insertFriendrequest(Friendrequest f, User u)
  M = return f
  aR = Sender, Receiver. *This is known from the description of abstract test case.*
  R = User sender, User receiver
  O = f
  This is what is needed to be shown:

  - *Show that O was returned to corresponding role*: f was returned to sender(1).
  - *Show that O was created as expected*: f is Friendrequest between User sender and User receiver(2).
  - *For every role R, which should be affected by O, show that O affects R as expected*: f must be visible for both sender and receiver (3).
  - *Create role uR*: create User unauthorized and show that f is not visible for unauthorized (4).

The complete derived test cases are available in B.2.3.

A.2 Delete a friend request

A.2.1 Abstract part

The sequence diagram is shown in Figure A.4.

Test case

The tested feature: **"Delete a friend request"**
Brief description of feature: user denoted as sender (receiver) removes sent (received) friend request
Description of participating roles:

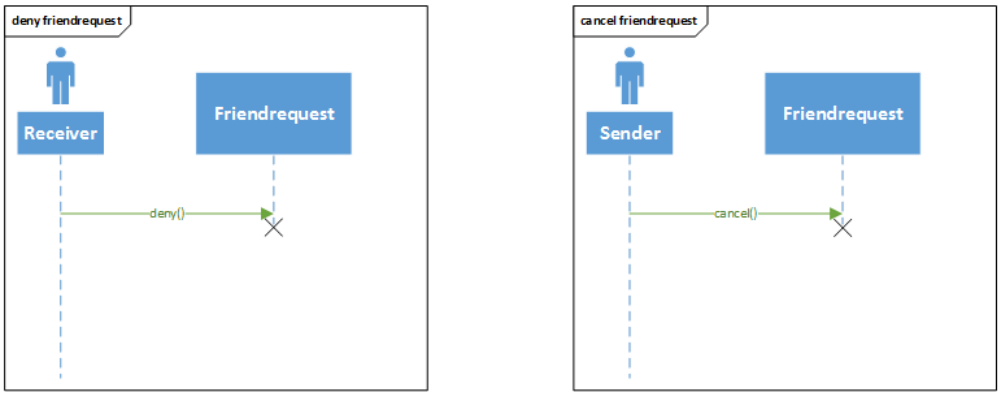


Figure A.4: sd for deleting friend request

- sender: is a user of the system, who sent some friend request
- receiver: is a user of the system, who received some friend request

Preconditions:

- sender exists
- receiver exists
- sender and receiver are different
- there is a friend request between sender and receiver

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.5.
Paths obtained from LTS
The paths are shown in tables A.6 and A.7.

Table A.6: path table obtained from first LTS in Figure A.5

Number	Path
1	User input, deny received FR, expected result, FR is deleted

Table A.7: path table obtained from second LTS in Figure A.5

Number	Path
1	User input, cancel sent FR, expected result, FR is deleted

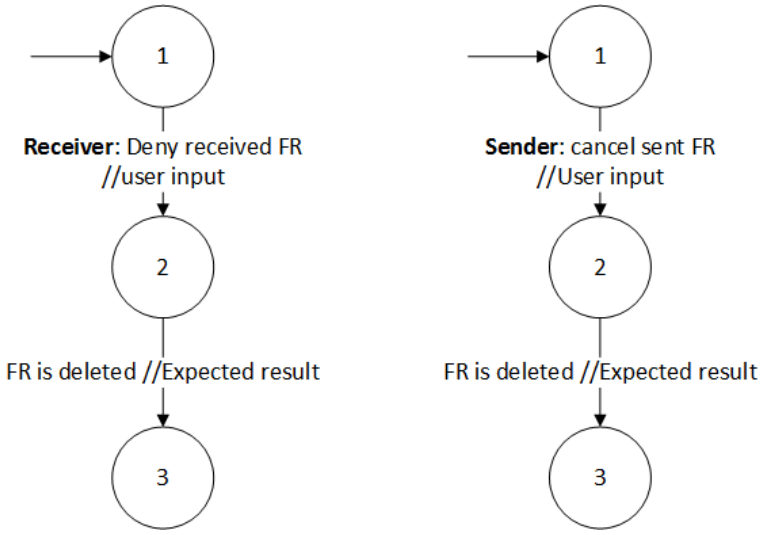


Figure A.5: LTS derived from A.4

Resulting abstract test cases:
The test cases are shown in tables A.8 and A.9. These test cases are derived from path tables A.6 and A.7.

Table A.8: Abstract test case 1

Initial condition	
User input	**Expected results**
Deny received FR	FR is deleted

Table A.9: Abstract test case 2

Initial condition	
User input	**Expected results**
Deny received FR	FR is deleted

Post conditions:

- sender exists
- receiver exists
- sender and receiver are different
- there is no friend request between sender and receiver

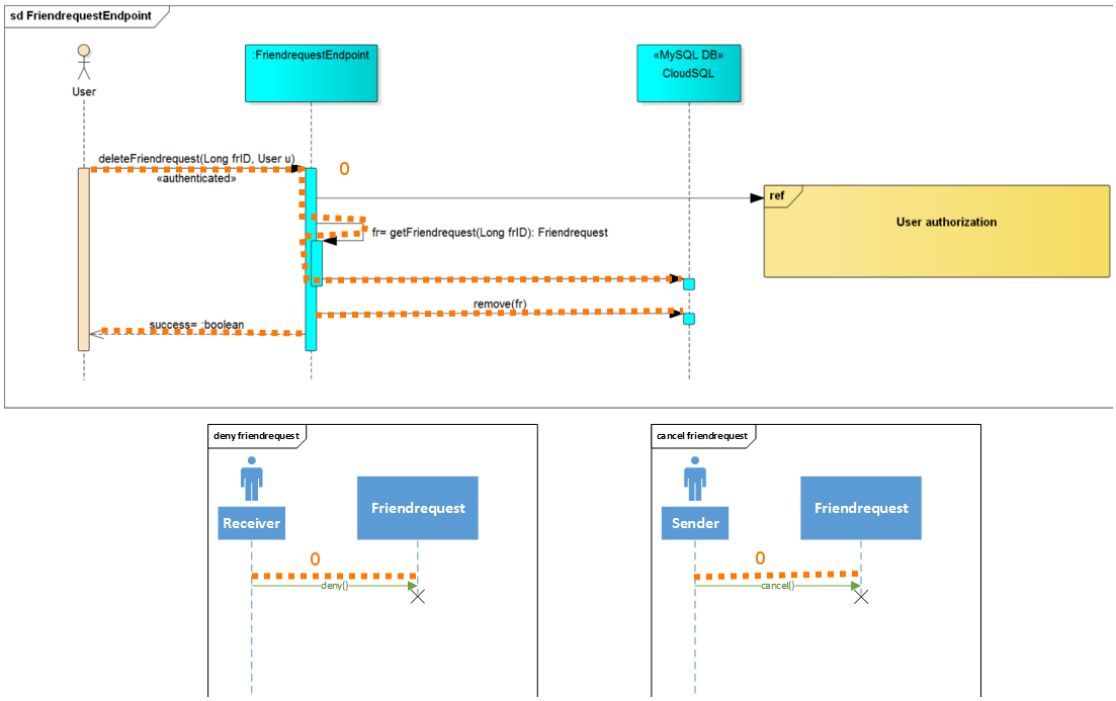


Figure A.6: The resulted mapping for delete a friend request feature

A.2.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.6.

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping shown in Figure A.6 to derive concrete test cases from abstract ones generated in Section A.2
  The resulting concrete test cases are shown in tables A.11 and A.11.

Table A.10: Concrete test case 1

Initial condition	
User input	**Expected result**
User receiver: removeFriendrequest(Long id, User u) - mapping 0	friend request is deleted

- Show for each test case whether the system returned the result as expected

Table A.11: Concrete test case 2

Initial condition	
User input	**Expected result**
User sender: removeFriendrequest(Long id, User u) - mapping 0	friend request is deleted

Test case1:
user input = removeFriendrequest(Long id, User u). *This is input from User receiver.*
$M = \emptyset$. *In this case there is no message returned back the user.*
aR = Sender, Receiver. *This is known from the description of abstract test case.*
R = User sender, User receiver
This is what is needed to be shown: In this cases there was created no object O, but we need to show, that Friendrequest was deleted as expected.

- friend request is not visible anymore

Test case2:
user input = removeFriendrequest(Long id, User u). *This is input from User sender.*
The same as for the test case 1.

If a user of the system wants to delete a friend request, he needs to invoke removeFriendrequest(Long id) method, it doesn't matter, whether he was User sender or User receiver, therefore the both concrete Test cases can be executed as one JUnit test.

The complete derived test cases are available in B.2.4.

A.3 Delete a friendship

A.3.1 Abstract part

The sequence diagram is shown in Figure A.7.

Test case

The tested feature: **"Delete a friendship"**
Brief description of feature: user denoted as friend removes friendship
Description of participating roles:

- friend: is a user of the system, who has a friendship with another user

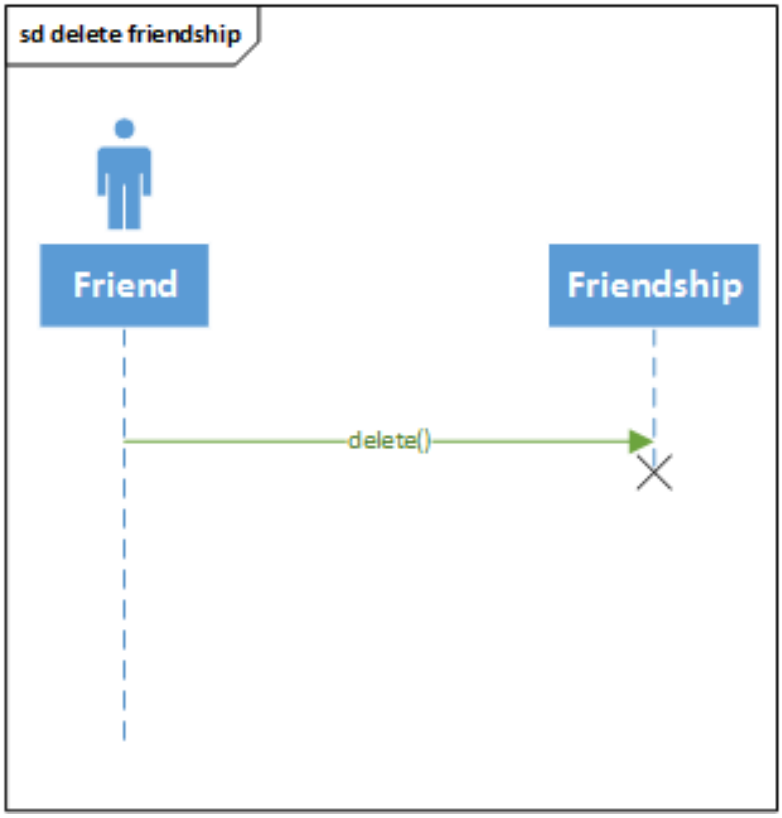


Figure A.7: sd for deleting friendship

Preconditions:

- 2 friends exists
- the friends are different
- there is a friendship between 2 friends

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.8.
Paths obtained from LTS
The paths are shown in table A.12.

Table A.12: path table obtained from LTS obtained in Figure A.8

Number	**Path**
1	User input, delete friendship, expected result, friendship is deleted

Resulting abstract test cases:
The test case is shown in table A.13. The test case is derived from path table A.12.

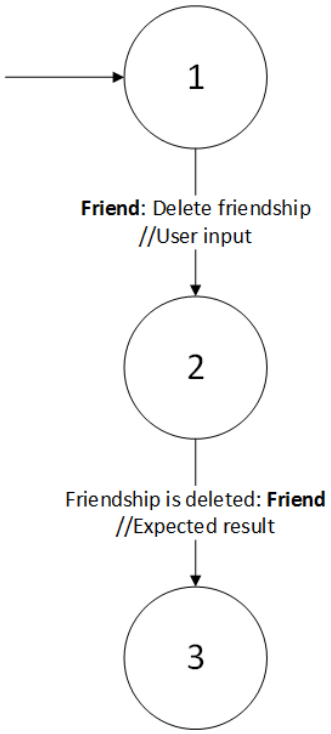


Figure A.8: LTS derived from A.7

Table A.13: Abstract test case 1

Initial condition	
User input	**Expected results**
Delete friendship	FR is deleted

Post conditions:

- 2 friends exists
- the friends are different
- there is no friendship between 2 friends

A.3.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.9.

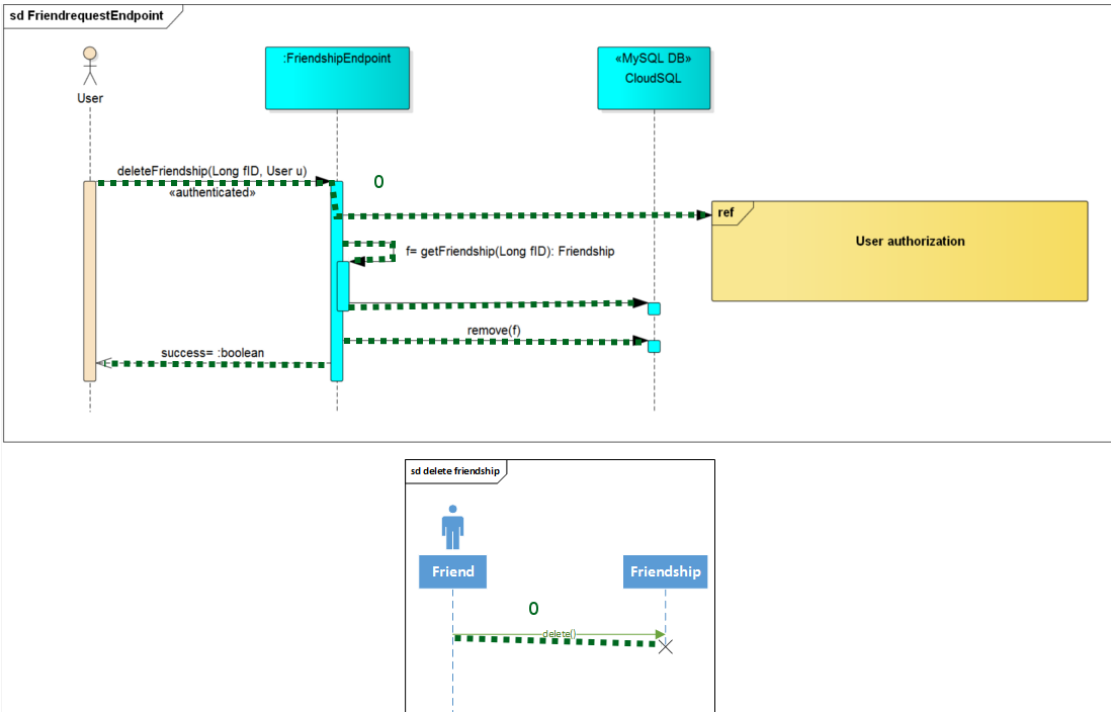


Figure A.9: The resulted mapping for delete a friendship feature

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping shown in Figure A.9 to derive concrete test cases from abstract ones generated in Section A.3.
  The resulting concrete test cases are shown in tables A.14.

Table A.14: Concrete test case 1

Initial condition	
User input	**Expected result**
User sender: removeFriendship(Long id, User u) - mapping 0	friend request is deleted

- Show for each test case whether the system returned the result as expected.

  Test case1:
  user input = removeFriendship(Long id, User u)
  $M = \emptyset$. *In this case there is no message returned back the user*
  aR = Friend A, Friend B. *This is known from the description of abstract test case.*
  R = User friendA, User friendB

This is what is needed to be shown: *In this cases there was created no object O, but we need to show, that Friendship was deleted as expected.*

- friendship is not visible anymore (1)

The complete derived test cases are available in B.2.5.

A.4 Creating a post

The sequence diagram is shown in Figure A.10.

A.4.1 Abstract part

Test case

The tested feature: **"Creating a post"**
Brief description of feature: user denoted as post owner creates a new post (containing at least one photo), the created post should be visible for users denoted as post viewers
Description of participating roles:

- post owner: user who creates a post
- post viewer: user who sees a post

Preconditions:

- post owner exists
- post viewers exist

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.11.
Paths obtained from LTS
The paths are shown in table A.15.

Table A.15: path table obtained from LTS in Figure A.11

Number	**Path**
1	Post owner: createPost() (user input), Post owner: createPhoto() (user input)*, created post (expected result): Post owner, read (expected result), createNotification() (expected result), notify() (expected result): Post viewer

Resulting abstract test cases:
The test case is shown in table A.13. The test case is derived from path table A.15.
Post conditions:

- post owner exists

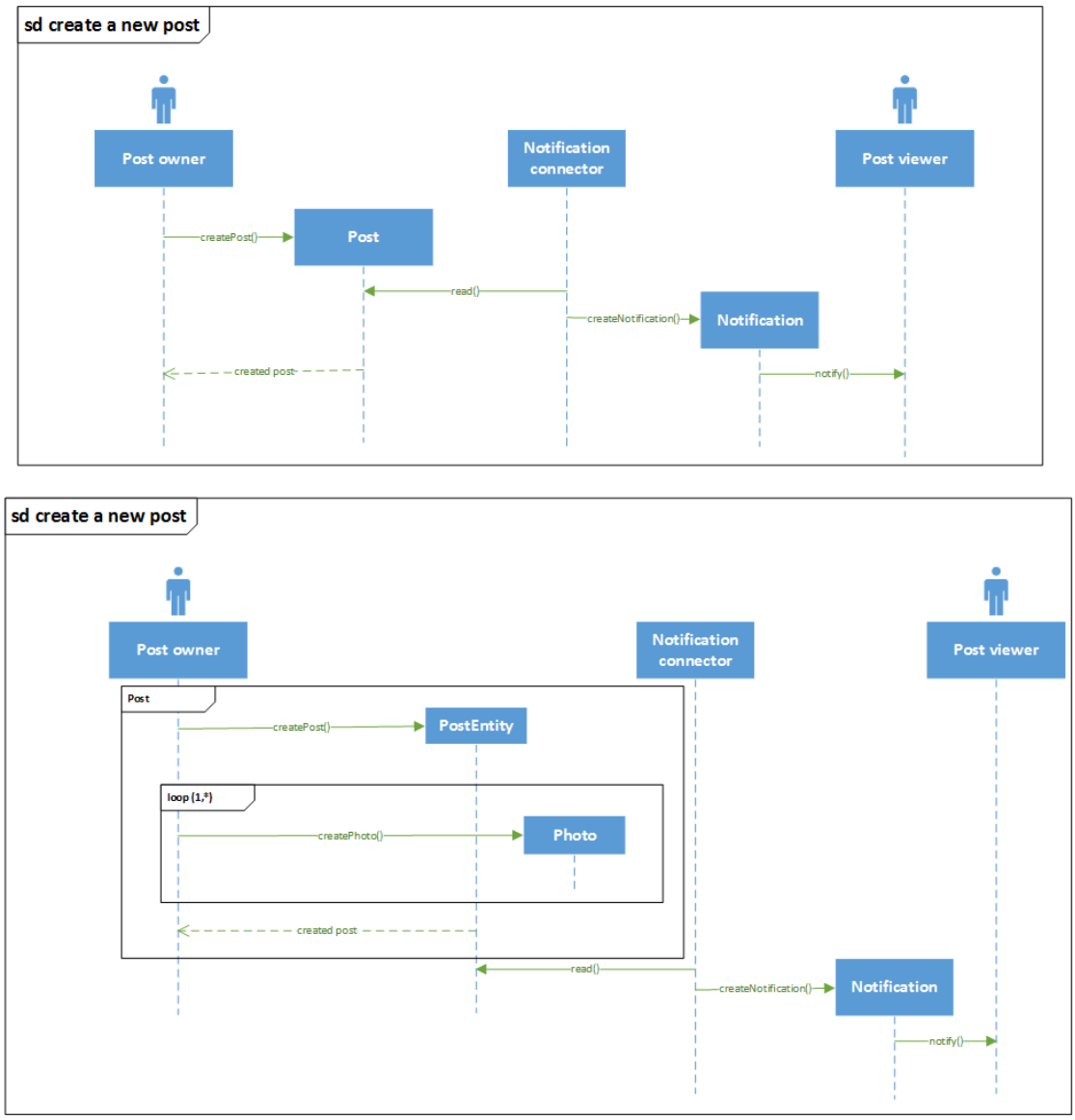


Figure A.10: sd for creating a post

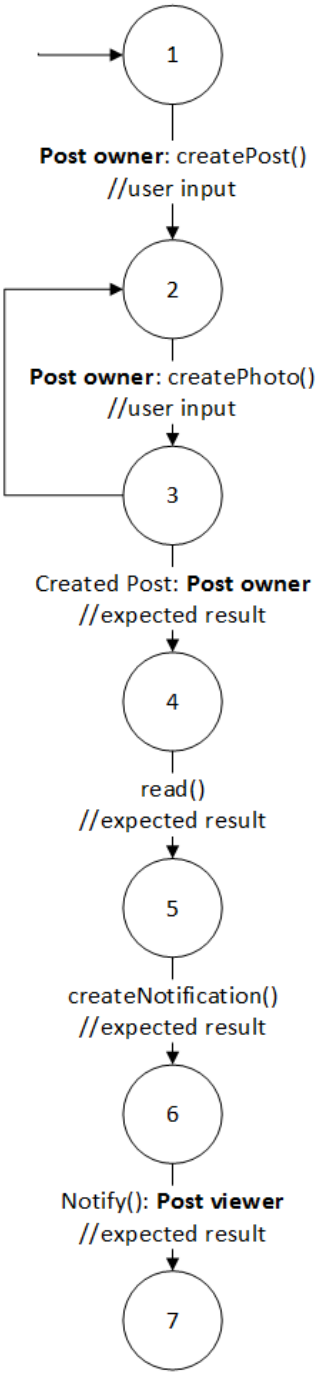


Figure A.11: LTS derived from A.10

Table A.16: Abstract test case 1

Initial condition	
User input	**Expected result**
Post owner: createPost() (user input), Post owner: createPhoto() (user input)*	created post (expected result): Post owner, read (expected result), createNotification() (expected result), notify() (expected result): Post viewer

- post exists
- post viewers exist

A.4.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.12.

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping shown in Figure A.12 to derive concrete test cases from abstract ones generated in Section A.4.
  The resulting concrete test cases are shown in tables A.17.

Table A.17: Concrete test case 1

Initial condition	
User input	**Expected result**
createSimplePost(SimplePost sp), - mapping 0 insertPhoto(Photo p)* -mapping 1	return post - mapping 2

- Show for each test case whether the system returned the result as expected

  Test case1:
  user input = createSimplePost(SimplePost sp, User u)
  M = return post
  aR = post owner, post viewer. *This is known from the description of abstract test case.*
  R = User postOwner, User postViewer.
  O = post
  This is what is needed to be shown:

  – *Show that O was returned to corresponding role*: post was returned to postOwner(1).

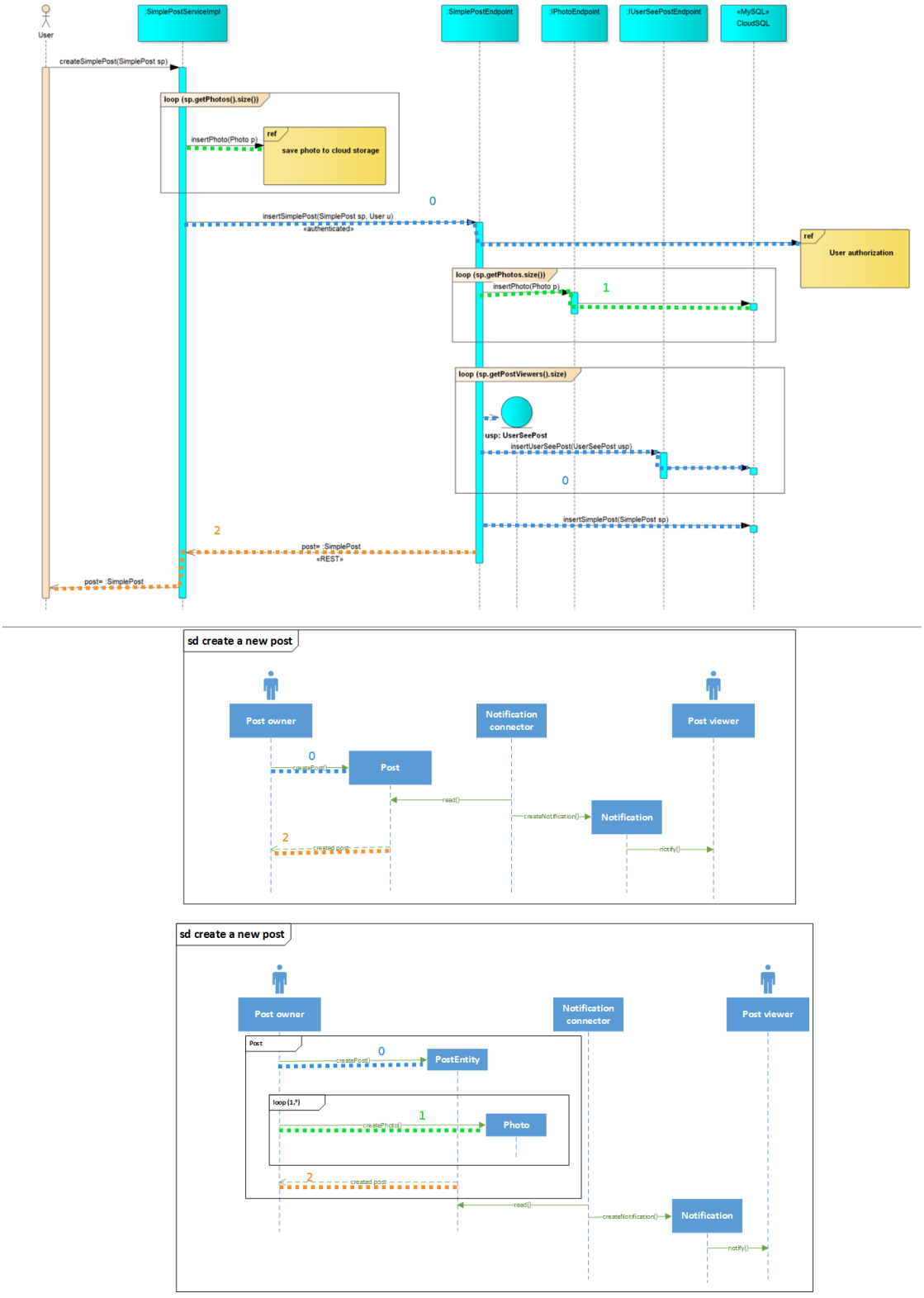


Figure A.12: The resulted mapping for creating a post feature

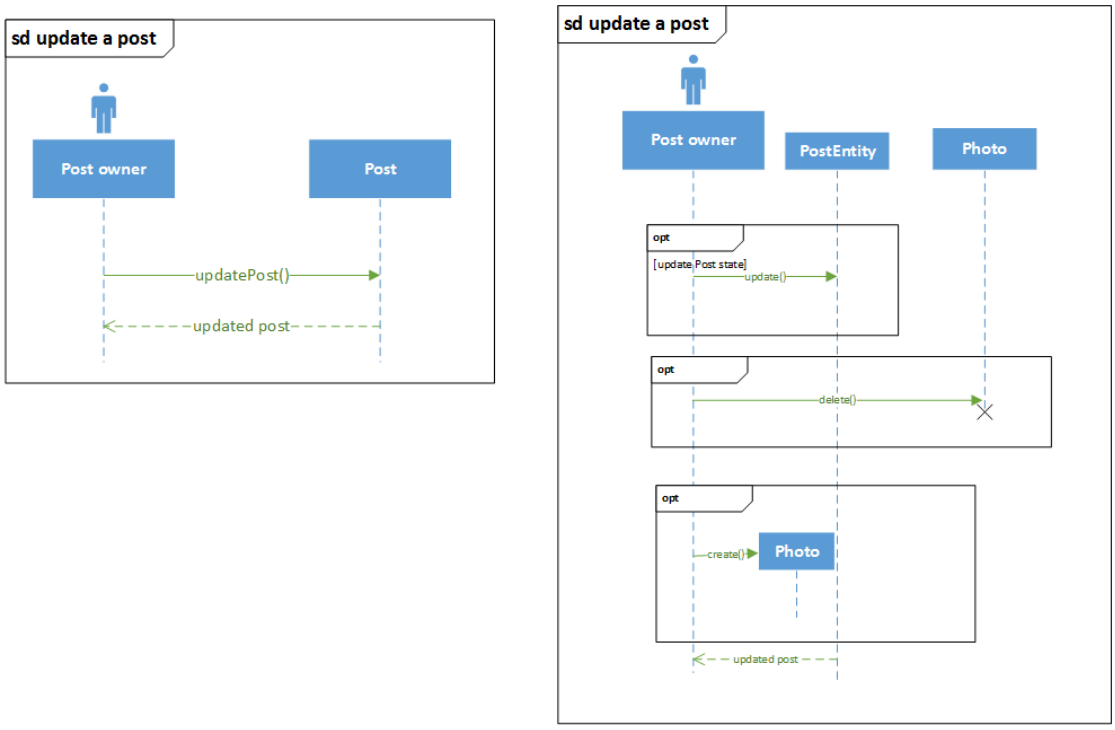


Figure A.13: sd for updating a post

- *For every role R, which should be affected by O, show that O affects R as expected*: post must be visible for all post viewers and for post owner (2).
- *Create role uR*: Create User unauthorized, who is not post viewer and not post owner, show that post is not visible for unauthorized (3).

The complete derived test cases are available in B.2.6.

A.5 Updating a post

The sequence diagram is shown in Figure A.13.

A.5.1 Abstract part

Test case

The tested feature: **"Updating a post"**
Brief description of feature: user denoted as post owner updates his post, changes in the post should be visible for post viewers
Description of participating roles:

- post owner: user who creates a post
- post viewer: user who sees a post

Preconditions:

- post owner exists
- post viewer exists
- post exists

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.14.
Paths obtained from LTS
The paths are shown in table A.18.

Table A.18: path table obtained from LTS in Figure A.14

Number	**Path**
1	Post owner: delete() (user input), updated post: post owner (expected result)
2	Post owner: delete() (user input), Post owner: create() (user input), updated post: post owner (expected result)
3	Post owner: update() (user input), Post owner: delete() (user input), updated post: post owner (expected result)
4	Post owner: update() (user input), Post owner: delete() (user input), Post owner: create() (user input), updated post: post owner (expected result)
5	Post owner: update() (user input), updated post: post owner (expected result)
6	Post owner: update() (user input), Post owner: create() (user input), updated post: post owner (expected result)
7	Post owner: create() (user input), updated post: post owner (expected result)

Resulting abstract test cases:
The test cases are shown in table A.19. The test cases are derived from path table A.18.
Post conditions:

- post owner exists
- post viewer exist

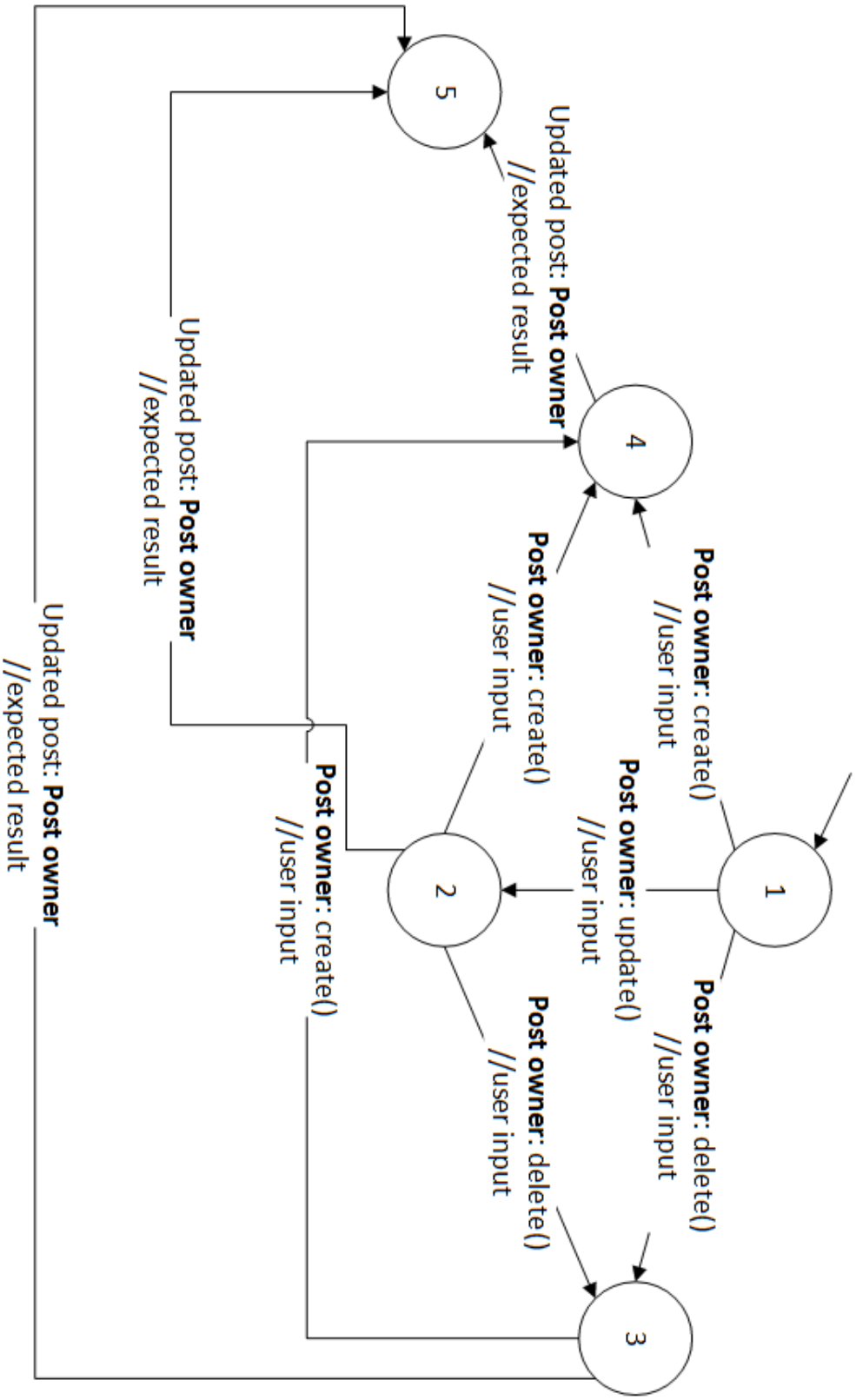


Figure A.14: LTS derived from A.13

Table A.19: Abstract test cases

Initial condition	
User input	**Expected result**
Post owner: delete()	updated post: post owner
Post owner: delete() Post owner: create()	updated post: post owner
Post owner: update(), Post owner: delete()	updated post: post owner
Post owner: update(), Post owner: delete(), Post owner: create(),	updated post: post owner
Post owner: update()	updated post: post owner
Post owner: update(), Post owner: create()	updated post: post owner
Post owner: create()	updated post: post owner

- post is updated

A.5.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.15.

Applying the process from 4.1 to abstract test cases 3.4

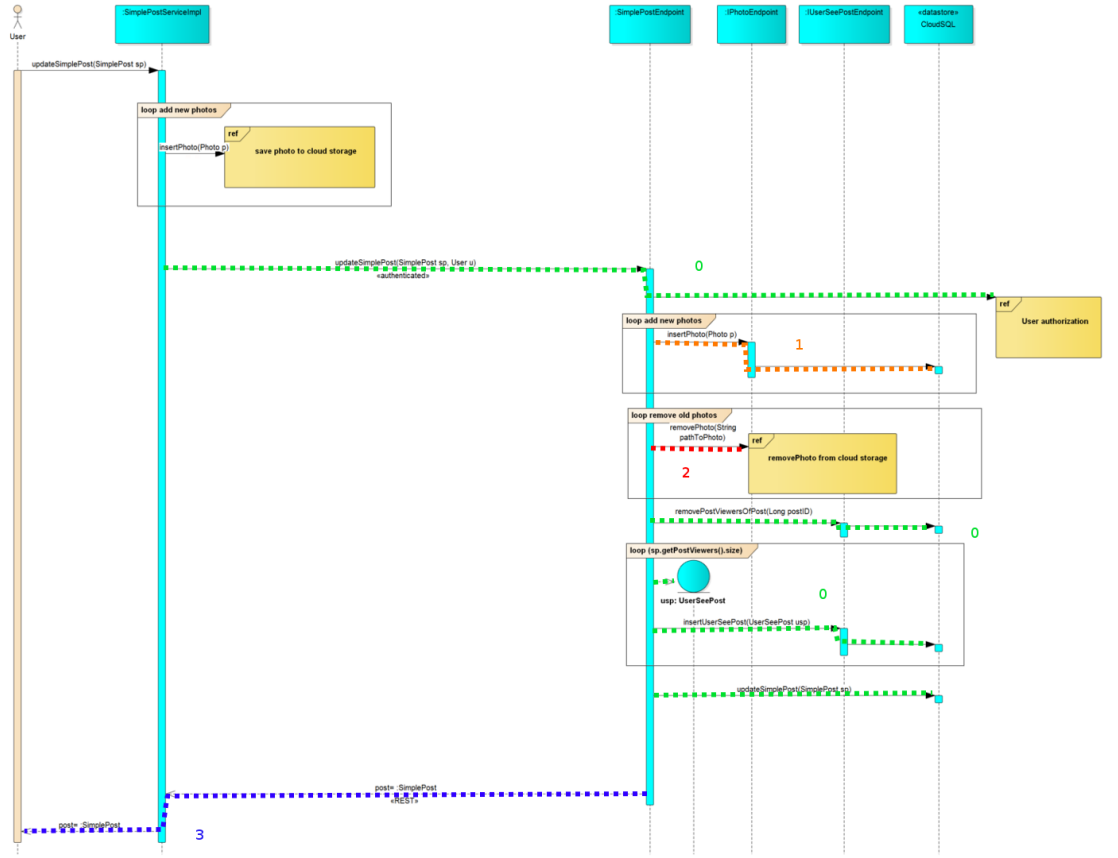
- Use the mapping shown in Figure A.12 to derive concrete test cases from abstract ones generated in Section A.4.
  The resulting concrete test cases are shown in table A.17.

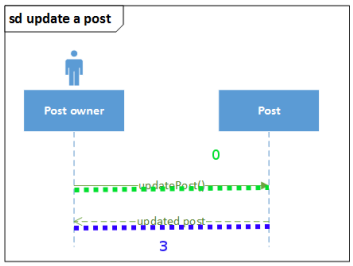
- Show for each test case whether the system returned the result as expected

  For all test cases:
  user input = updateSimplePost(SimplePost sp, User u) M = return post aR = post owner, post viewer. *This is known from the description of abstract test case.*
  R = User postOwner, User postViewer O = post

  This is what is needed to be shown:

  - *Show that O was returned to corresponding role*: post was returned to postOwner(1).
  - *For every role R, which should be affected by O, show that O affects R as expected*: Changes in post must be visible for all post viewers and for post owner (2).





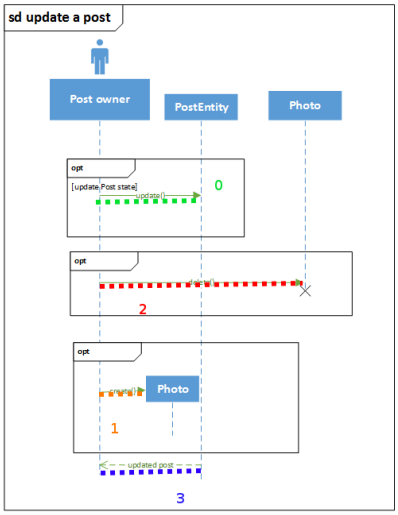


Figure A.15: The resulted mapping for updating a post feature

Table A.20: Concrete test cases

Initial condition	
User input	**Expected result**
User removes some photos (mapping 2) from SimplePost and invokes updateSimplePost(SimplePost sp, User u) - mapping 0	return post - mapping 3
User removes (mapping 2) and add (mapping 1) some photos from/to SimplePost and invokes updateSimplePost(SimplePost sp, User u) -mapping 2	return post - mapping 3
User updates the state (description) of SimplePost, deletes some photos (mapping 2) from SimplePost and invokes updateSimplePost(SimplePost sp, User u) -mapping 2	return post - mapping 3
User updates the state of SimplePost, removes (mapping 2), add (mapping 1) some photos from/to SimplePost and invokes updateSimplePost(SimplePost sp, User u) -mapping 2	return post - mapping 3
User updates the state (description) of SimplePost and invokes updateSimplePost(SimplePost sp, User u) - mapping 2	return post - mapping 3
User updates the state of SimplePost, add (mapping 1) some photos to SimplePost and invokes updateSimplePost(SimplePost sp, User u) -mapping 2	return post - mapping 3
User adds (mapping 1) some photos to SimplePost and invokes updateSimplePost(SimplePost sp, User u) -mapping 2	return post - mapping 3

– *Create role uR*: Create User unauthorized, who is not post viewer and not post owner, show that post is not visible for unauthorized (3).

The complete derived test cases are available in B.2.7.

A.6 Deleting a post

The sequence diagram is shown in Figure A.16.

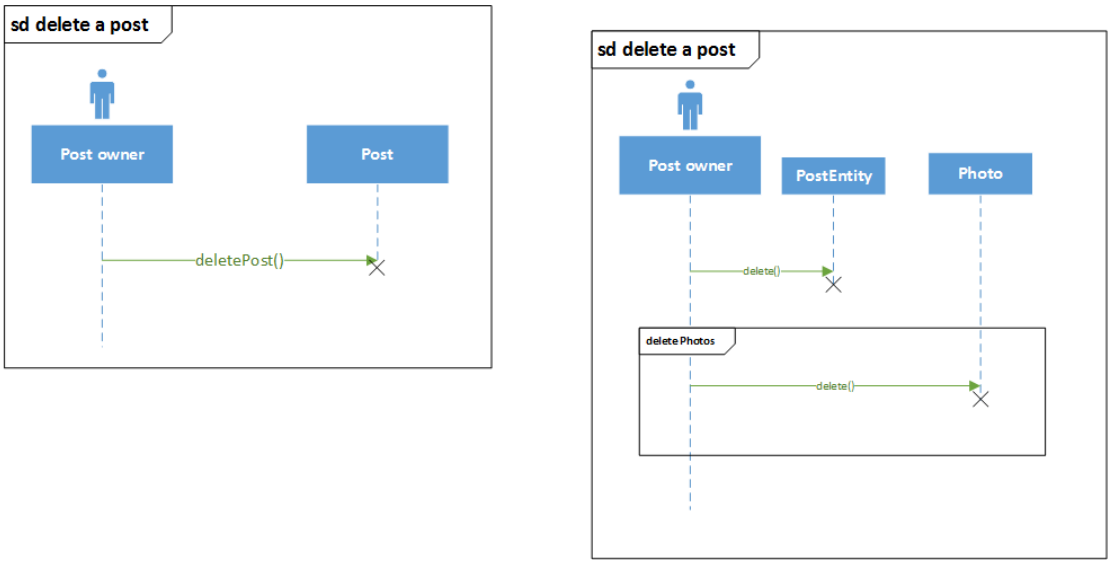


Figure A.16: sd for deleting a post

A.6.1 Abstract part

Test case

The tested feature: **"Deleting a post"**
Brief description of feature: user denoted as post owner deletes his post, the deleted post shouldn't be visible for post viewers and post owner anymore
Description of participating roles:

- post owner: user who creates a post
- post viewer: user who sees a post

Preconditions:

- post owner exists
- post viewers exist
- post exists

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.17.
Paths obtained from LTS
The paths are shown in table A.18.
Resulting abstract test cases:
The test cases are shown in table A.19. The test cases are derived from path table A.21.
Post conditions:

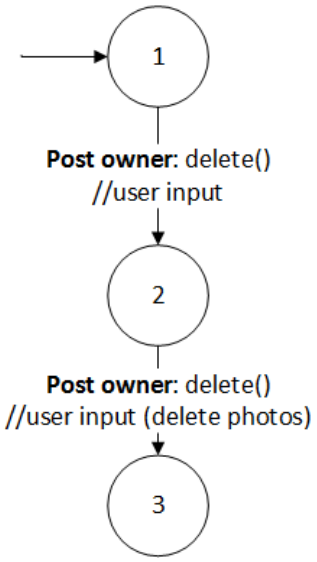


Figure A.17: LTS derived from A.16

Table A.21: path table obtained from LTS in Figure A.17

Number	Path
1	Post owner: delete() //user input, Post owner: delete() //user input (delete photos)

Table A.22: Abstract test case

Initial condition	
User input	**Expected result**
Post owner: delete() Post owner: delete() (delete photos)	

- post owner exists
- post viewer exist
- post is deleted

A.6.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.18.

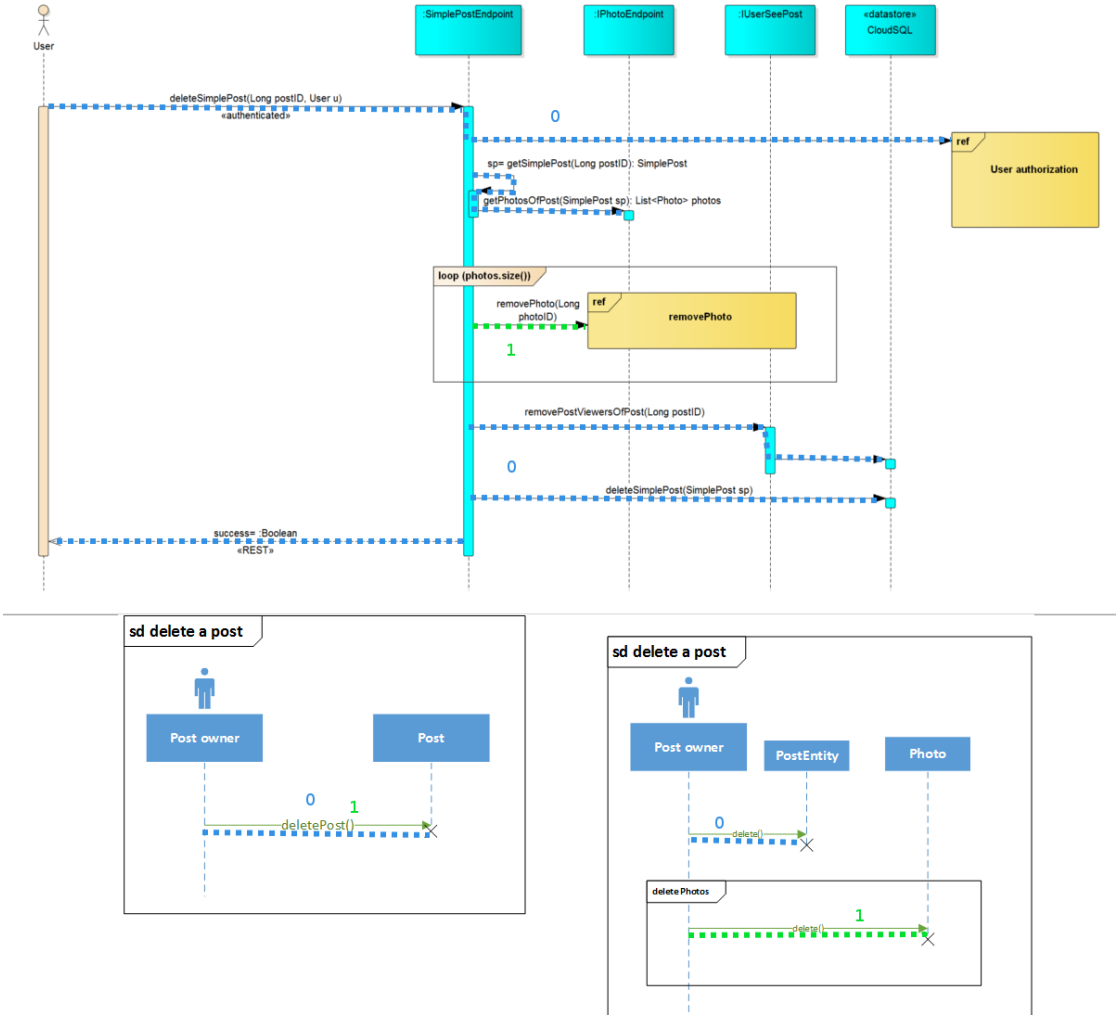


Figure A.18: The resulted mapping for deleting a post feature

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping shown in Figure A.18 to derive concrete test cases from abstract ones generated in Section A.6.
  The resulting concrete test cases are shown in table A.23.

Table A.23: Concrete test case

Initial condition	
User input	**Expected result**
deleteSimplePost(Long id, User u)	simple post with ID = id is deleted

- Show for each test case whether the system returned the result as expected.

  Test case1:
  user input = deleteSimplePost(Long id, User u)
  $M = \emptyset$. *In this case there is no message returned back the user.*
  aR = post owner, post viewer. *This is known from the description of abstract test case.*
  R = User postOwner, User postViewer.
  This is what is needed to be shown: In this cases there was created no object O, but we need to show, that post was deleted as expected.

  - post is not visible anymore

The complete derived test cases are available in B.2.8.

A.7 Disliking a photo within post

A.7.1 Abstract part

The sequence diagram is shown in Figure A.19.

Test case

The tested feature: **"Disliking a photo within post"**
Brief description of feature: user denoted as post viewer creates a dislike for a photo within a post
Description of participating roles:

- dislike creator: user who is a post viewer and who creates a dislike
- post viewer: user who can see a post, post viewer has to be different from dislike creator

Preconditions:

- dislike creator exists

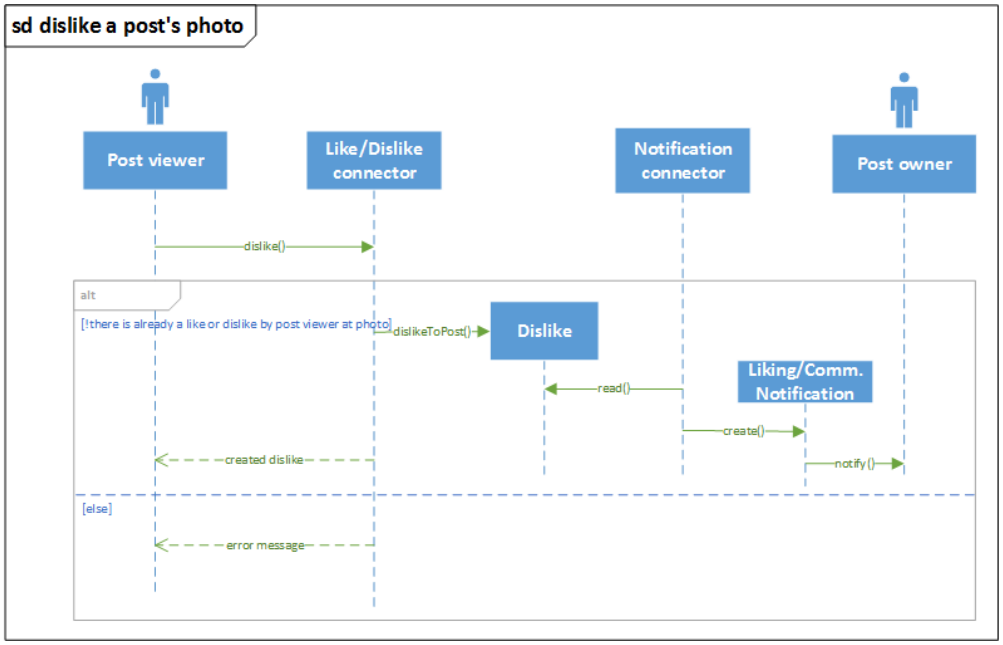


Figure A.19: sd for disliking a photo within a post

- post viewer exists
- post exists

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.20.
Paths obtained from LTS
The paths are shown in table A.24.

Table A.24: path table obtained from LTS in Figure A.20

Number	Path
1	Post viewer: dislike() (user input), conditions, There is already a like OR dislike by post viewer at photo error message: Post viewer (expected result)
2	Post viewer: dislike() (user input), conditions, else, dislikeToPost() (expected result), read(expected result), create(expected result), notify(): Post owner (expected result)

Resulting abstract test cases:
The test cases are shown in tables A.25 and A.26. These test cases are derived from path table A.24.
Post conditions:

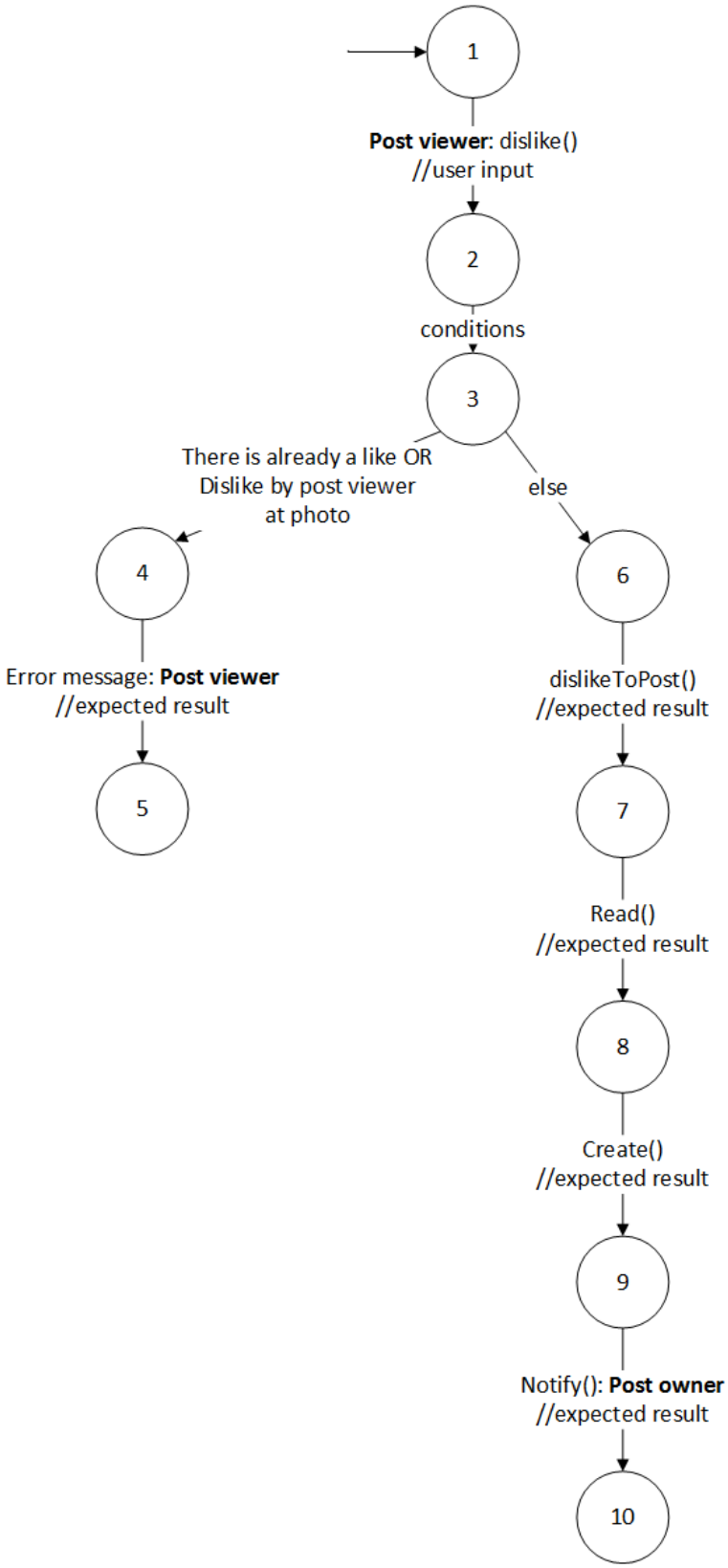


Figure A.20: LTS derived from A.19

Table A.25: Abstract test case 1

Initial condition	There is already a like or dislike by post viewer at photo
User input	**Expected result**
Post viewer: dislike()	error message: Post viewer

Table A.26: Abstract test case 2

Initial condition	There is no like and no dislike by post viewer at photo
User input	**Expected result**
Post viewer: dislike()	dislikeToPost(), read(), create() notify(): Post owner, created dislike: Post viewer

- post exists
- dislike creator exists
- post viewer exists

A.7.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.21.

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping in Figure A.21 to derive concrete test cases from the abstract ones generated in Section A.7
  The resulting concrete test cases are shown in tables A.27 and A.28.

Table A.27: Concrete test case 1

Initial condition	There is already a like or dislike by post viewer at photo
User input	**Expected result**
insertDislikes(Dislikes dislikes, User u) - mapping 0	insert dislike to DB - mapping 2, return dislikes; - mapping 3

- Map the initial conditions from abstract test case, to the concrete initial condition *initialCon* = There is already a like by post viewer at photo, There is already a dislike by post viewer at photo Since there are two *atoms* in the *initialCon* we need to execute the test case A.27 two times. Each time one *atom* in *initialCon* is set to true.

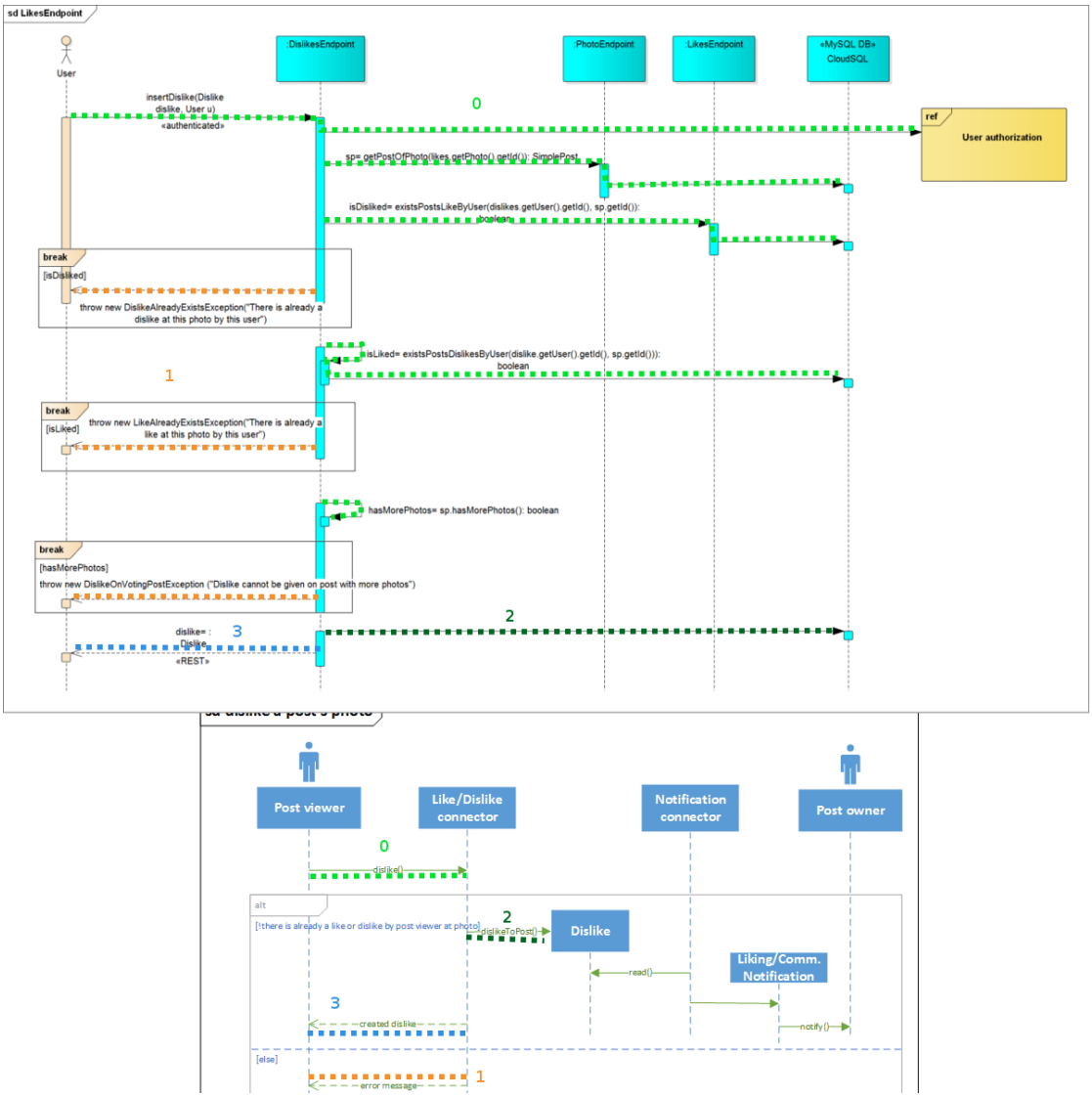


Figure A.21: The resulted mapping for disliking a photo within post feature

Table A.28: Concrete test case 2

Initial condition	There is no like and no dislike by post viewer at photo
User input	**Expected result**
insertDislikes(Dislikes dislikes, User u) - mapping 0	insert dislike to DB - mapping 2, return likes; - mapping 3

- Show for each test case whether the system returned the result as expected

  Test case1:
  initial condition = There is already a like by post viewer at photo
  user input = insertDislikes(Dislikes dislikes, User u)
  M = throw new LikeAlreadyExistsException("There is already a like at this photo by this post")
  aR = dislike creator, post viewer. *This is known from the description of abstract test case.*
  R = User dislikeCreator. *There is no need to create User postViewer, because exception should be thrown before it could be shown that Like likes is visible to postViewer.*
  This is what is needed to be shown:

  – *Show that O was returned to corresponding role*: LikeAlreadyExistsException is returned to the post viewer (comment creator) (1).

  //anotate the test case with
  @Test(expected = LikeAlreadyExistsException.class)

  Test case2:
  initial condition = There is already a dislike by post viewer at photo
  user input = insertDislikes(Dislikes dislikes, User u)
  M = throw new DislikeAlreadyExistsException("There is already a dislike at this photo by this post")
  aR = like creator, post viewer. *This is known from the description of abstract test case.*
  R = User likeCreator. *There is no need to create User postViewer, because exception should be thrown before it could be shown that Like likes is visible to postViewer.*
  This is what is needed to be shown:

  – *Show that O was returned to corresponding role*: DislikeAlreadyExistsException is returned to the post viewer (comment creator) (1).

  //anotate the test case with
  @Test(expected = DislikeAlreadyExistsException.class)

Test case3:
user input = insertDislikes(Dislikes dislikes, User u)
M = return dislikes
aR = dislike creator, post viewer. *This is known from the description of abstract test case.*
R = User likeCreator, User postViewer
O = dislikes
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: dislikes was returned to the post viewer(dislike creator)(1).
- *Show that O was created as expected*: dislikes is Dislike at photo of a post(2).
- *For every role R, which should be affected by O, show that O affects R as expected*: dislikes must be visible for post viewers of dislikes' post (3).
- *Create role uR*: create a User unauthorized, who is not a post viewer and show that he cannot create dislikes at post, which is not visible for him (4) - I show this in the Test case 4.

Test case4:
user input = insertLikes(Likes likes, User u)
M = throw new UnauthorizedDisLikeException("Unauthorized like create");
aR = non-post viewer
R = User nonPostViewer
This is what is needed to be shown:

- UnauthorizedDisLikeException was returned to User nonPostViewer

The complete derived test cases are available in B.2.9.

A.8 Commenting a photo within post

A.8.1 Abstract part

The sequence diagram is shown in Figure A.22.

Test case

The tested feature: **"Commenting a photo within post"**
Brief description of feature: user denoted as post viewer leaves a comment at photo within a post
Description of participating roles:

- comment creator: user who is a post viewer and who creates a comment

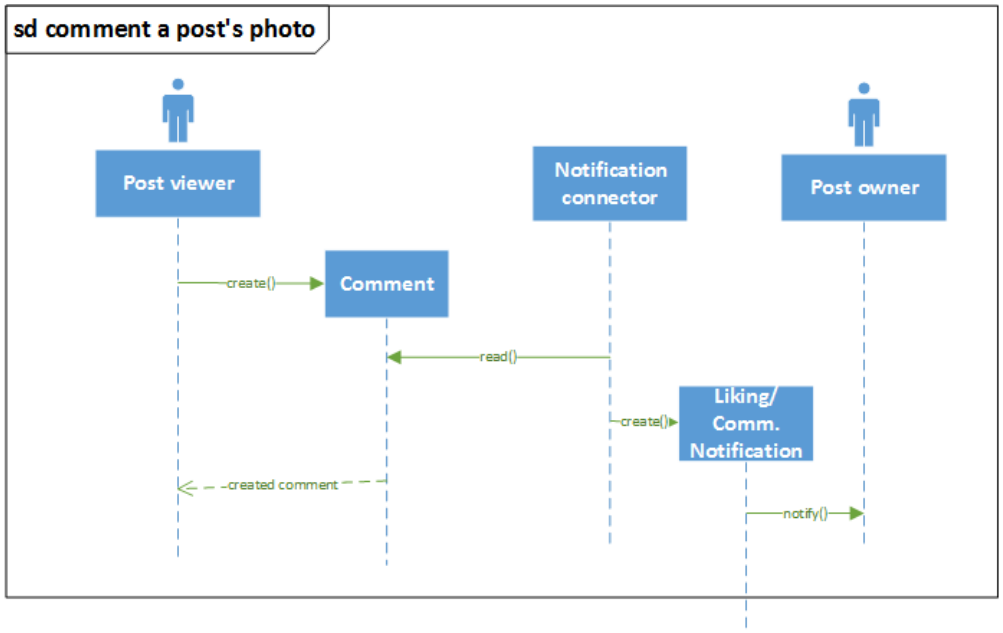


Figure A.22: sd for comment at photo within a post

- post viewer: user who can see a post, post viewer has to be different from comment creator

Preconditions:

- comment creator exists
- post viewer exists
- post exists

Labeled transition system:
The LTS obtained to test this feature is shown in Figure A.23.
Paths obtained from LTS
The paths are shown in table A.29.

Table A.29: path table obtained from LTS in Figure A.23

Number	Path
1	Post owner: create() (user input), read() (expected result), create() (expected result), notify() (expected result), created comment: Post viewer (expected result)

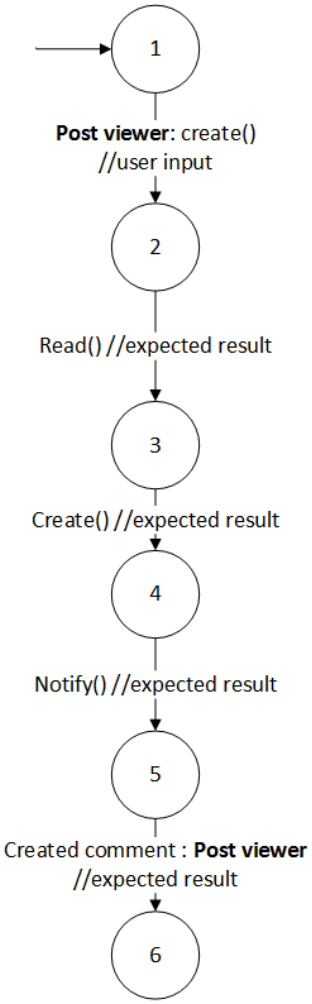


Figure A.23: LTS derived from A.22

Resulting abstract test cases:
The test case is shown in tables A.30. The test case is derived from path table A.29.

Table A.30: Abstract test case 1

Initial condition	
User input	**Expected result**
Post owner: create()	read(), create(), notify(), created comment: Post viewer

Post conditions:

- comment creator exists

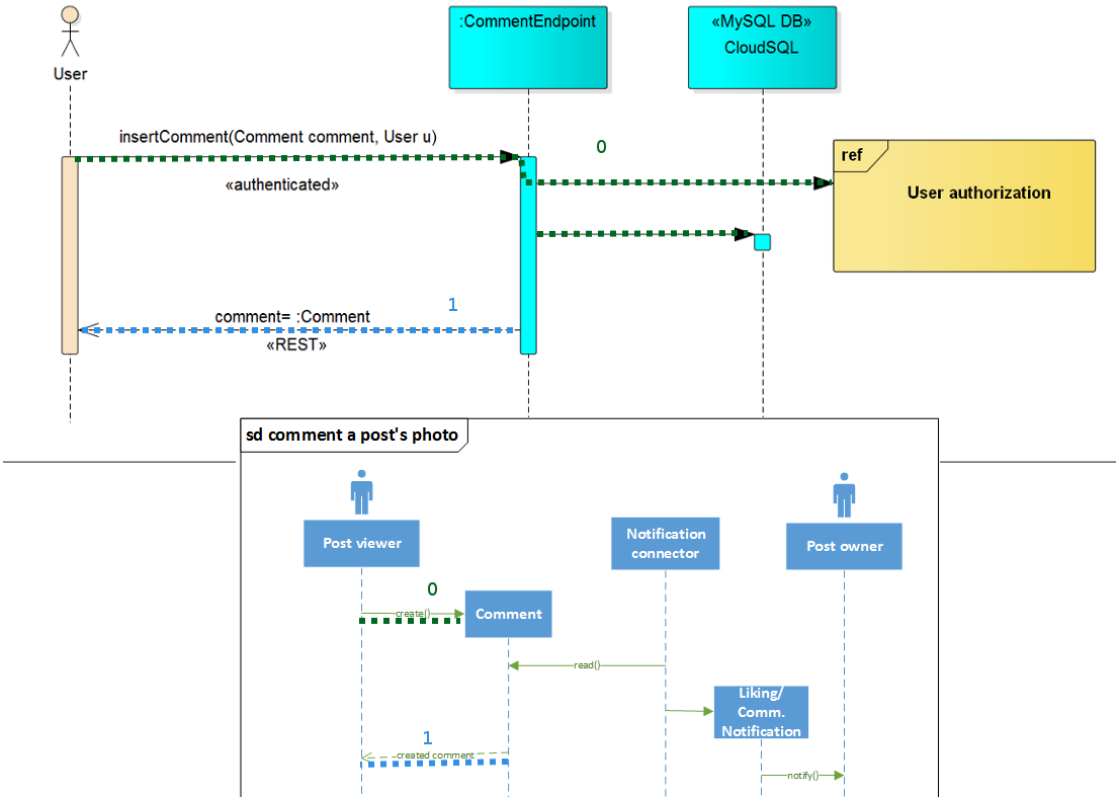


Figure A.24: The resulted mapping for commenting a photo within post feature

- post viewer exists
- post exists

A.8.2 Concrete part

Mapping between xSD and hSD

The resulted mapping is shown in Figure A.24.

Applying the process from 4.1 to abstract test cases 3.4

- Use the mapping shown in Figure A.24 to derive concrete test cases from the abstract ones generated in Section A.8
  The resulting concrete test case is shown in table A.31.
- Show for each test case whether the system returned the result as expected

Table A.31: Concrete test case

Initial condition	
User input	**Expected result**
insertComment(Comment c, User u) - mapping 0	return c;

Test case 1:
user input = insertComment(Comment c, User u)
M = return comment
aR = comment creator, post viewer. *This is known from the description of abstract test case.*
R = User commentCreator, User postViewer
O = comment
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: comment was returned to the post viewer(comment creator)(1).
- *Show that O was created as expected*: comment is Comment at photo of a post(2).
- *For every role R, which should be affected by O, show that O affects R as expected*: comment must be visible for post viewers of comment' post (3).
- *Create role uR*: create a User unauthorized, who is not a post viewer and show that he cannot create comment at post, which is not visible for him (2) - I show this in the Test case 2.

Test case 2:
user input = insertComment(Comment c, User u)
M = some Exception should be thrown
aR = comment creator, post viewer. *This is known from the description of abstract test case.*
R = User commentCreator, User postViewer
This is what is needed to be shown:

- *Show that O was returned to corresponding role*: exception was returned to the post viewer(comment creator)(1).

The complete derived test cases are available in B.2.10.

APPENDIX B

Implementation

B.1 Concrete social network

The source code of the concrete social network is created by the author of this thesis and is private.

B.2 JUnit Test cases

This section presents the implementation of derived concrete test cases.

B.2.1 Accepting a friend request

Test case 1:

```
@Test(expected = EntityExistsException.class)
  public void
      test_createFriendshipFromFriendrequest_shouldThrowEntityExistsException()
      {

    // creating preconditions
    User sender = new User("senderName", "sender@somemail.com",
        "pass123");
    User receiver = new User("receiverName",
        "reciever@somemail.com", "pass234");
    sender = userService.insertUser(sender);
    receiver = userService.insertUser(receiver);
    Friendrequest f1 = new Friendrequest(sender, receiver);
    friendrequestService.insertFriendrequest(f1);

    //initial condition
    Friendship f = new Friendship(sender, receiver);
```

```
    friendshipService.insertFriendship(f);

    /*
     * user input
     * expected result (1) EntityExistsException should be thrown
        here
     */
    friendshipService.createFriendshipFromFriendrequest(f1);


    // post conditions
    assertTrue(friendrequestService.getFriendrequest(f1.getId()) ==
       null);
    assertTrue(userService.getUser(sender.getId()).equals(sender));
    assertTrue(userService.getUser(receiver.getId()).equals(receiver));
    assertFalse(sender.equals(receiver));

  }
```

Test case 2:

```
  @Test
  public void test_createFriendshipFromFriendrequest_shouldOK() {

    // creating preconditions
    User sender = new User("senderName", "sender@somemail.com");
    User receiver = new User("receiverName",
       "reciever@somemail.com");
    sender = userService.insertUser(sender);
    receiver = userService.insertUser(receiver);
    Friendrequest f1 = new Friendrequest(sender, receiver);
    friendrequestService.insertFriendrequest(f1);

    /*
     * user input
     */
    Friendship friendship =
       friendshipService.createFriendshipFromFriendrequest(f1);

    /*
     * what is needed to be shown part (1)
     */
    assertTrue((friendship.getUser1().equals(receiver) ||
       friendship.getUser1().equals(sender))
        && (friendship.getUser2().equals(receiver) ||
           friendship.getUser2().equals(sender)));

    /*
     * what is needed to be shown part (2),(3)
```

```
     */
    assertTrue(friendshipService.listFriendshipsOfUser(sender.getId()).
    getItems().contains(friendship));
    assertTrue(friendshipService.listFriendshipsOfUser(receiver.getId()).
    getItems().contains(friendship));

    /*
     * what is needed to be shown part (4)
     */
    User unauthorized = new User("unauth", "unauth");
    unauthorized = userService.insertUser(unauthorized);

    assertFalse(friendshipService.listFriendshipsOfUser(unauthorized.getId()).getItems(

    // post conditions
    assertTrue(friendrequestService.getFriendrequest(f1.getId()) ==
        null);
    assertTrue(userService.getUser(sender.getId()).equals(sender));
    assertTrue(userService.getUser(receiver.getId()).equals(receiver));
    assertFalse(sender.equals(receiver));
  }
```

B.2.2 Liking a photo within post

Test case 1

```
@Test(expected = LikeAlreadyExistsException.class)
  public void
      test_insertLikes_shouldThrowLikeAlreadyExistsException() throws
      DislikeAlreadyExistsException, LikeAlreadyExistsException,
      UnauthorizedDisLikeException {

    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User likeCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
```

```
    photos.add(photo1);
    simplePost1.setPhotos(photos);

    List<User> postViewers = new ArrayList<User>();
    postViewers.add(likeCreator);
    simplePost1.setUserCanSeePost(postViewers);

    postOwner = userDAO.insertUser(postOwner);
    likeCreator = userDAO.insertUser(likeCreator);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    /*
     * creating initial condition
     */
    Likes likes1 = new Likes();
    likes1.setPhoto(photo1);
    likes1.setUser(likeCreator);

    likes1 = likesDAO.insertLikes(likes1);

    /*
     * user input
     */
    Likes likes2 = new Likes();
    likes2.setPhoto(photo1);
    likes2.setUser(likeCreator);

    /*
     * LikeAlreadyExistsException should be thrown here
     */
    likes2 = likesDAO.insertLikes(likes2);
  }
```

Test case 2

```
  @Test(expected = DislikeAlreadyExistsException.class)
  public void
      test_insertLikes_shouldThrowDislikeAlreadyExistsException()
      throws DislikeAlreadyExistsException,
      LikeAlreadyExistsException, UnauthorizedDisLikeException {

    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User likeCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
```

```
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    simplePost1.setPhotos(photos);

    List<User> postViewers = new ArrayList<User>();
    postViewers.add(likeCreator);
    simplePost1.setUserCanSeePost(postViewers);

    postOwner = userDAO.insertUser(postOwner);
    likeCreator = userDAO.insertUser(likeCreator);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    /*
     * creating initial condition
     */
    Dislikes dislikes1 = new Dislikes();
    dislikes1.setPhoto(photo1);
    dislikes1.setUser(likeCreator);

    dislikes1 = dislikesDAO.insertDislikes(dislikes1);

    /*
     * user input
     */
    Likes likes2 = new Likes();
    likes2.setPhoto(photo1);
    likes2.setUser(likeCreator);

    /*
     * DislikeAlreadyExistsException should be thrown here
     */
    likes2 = likesDAO.insertLikes(likes2);
  }
```

Test case 3

```
@Test
  public void test_insertLikes_shouldOK() throws
      DislikeAlreadyExistsException, LikeAlreadyExistsException,
      UnauthorizedDisLikeException {
    /*
     * creating preconditions
```

```
 */
User postOwner = new User("user1","email1");
User postViewer = new User("user3", "email123");
User likeCreator = new User("user2","email2");

SimplePost simplePost1 = new SimplePost();
simplePost1.setDescription("post1");
simplePost1.setPostedBy(postOwner);

Photo photo1 = new Photo();
photo1.setDescription("photo1");
photo1.setPath("gg");

List<Photo> photos = new ArrayList<Photo>();
photos.add(photo1);
simplePost1.setPhotos(photos);

List<User> postViewers = new ArrayList<User>();
postViewers.add(likeCreator);
postViewers.add(postViewer);
simplePost1.setUserCanSeePost(postViewers);

postOwner = userDAO.insertUser(postOwner);
likeCreator = userDAO.insertUser(likeCreator);
postViewer = userDAO.insertUser(postViewer);
simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

/*
 * user input
 */
Likes likes1 = new Likes();
likes1.setPhoto(photo1);
likes1.setUser(likeCreator);

/*
 * what is needed to be shown part
 */
likes1 = likesDAO.insertLikes(likes1); // (1) like was returned
   to post viewer

/*
 * likes1 is Like at photo of a post
 * (2)
 */
assertTrue(likes1.getPhoto().equals(photo1));

/*
 * (3) likes1 must be visible for post viewers
 */
```

```
    List<SimplePost> postVisibleForPostViewer = (List<SimplePost>)
        simplePostDAO.listSimplePostsForUser(postViewer.getId()).getItems();

    for(SimplePost sp : postVisibleForPostViewer) {
      if(sp.getPhotos().contains(photo1)) {
        List<Likes> likesOfPhoto = (List<Likes>)
            likesDAO.listLikesOfPhoto(photo1.getId(), 0,
            0).getItems();
        assertTrue(likesOfPhoto.contains(likes1));
      }
    }

    /*
     * showing post conditions
     */
    assertTrue(simplePostDAO.getSimplePost(simplePost1.getId()).equals(simplePost1));
        // (1)
    assertTrue(userDAO.getUser(likeCreator.getId()).equals(likeCreator));//
        (2)
    assertTrue(userDAO.getUser(postViewer.getId()).equals(postViewer));//
        (3)

  }
```

Test case 4

```
@Test(expected = UnauthorizedDisLikeException.class)
  public void test_insertLikes_shouldThrowUnauthorizedDisLike()
      throws DislikeAlreadyExistsException,
      LikeAlreadyExistsException, UnauthorizedDisLikeException {

    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User nonPostViewer = new User("user2","email2");
    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    simplePost1.setPhotos(photos);
```

```
    postOwner = userDAO.insertUser(postOwner);
    nonPostViewer = userDAO.insertUser(nonPostViewer);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    Likes likes1 = new Likes();
    likes1.setPhoto(photo1);
    likes1.setUser(nonPostViewer);

    /*
     * user input
     * UnauthorizedDisLikeException should be thrown here
     */
    likes1 = likesDAO.insertLikes(likes1);
  }
```

B.2.3 Sending a friend request

Test case 1

```
@Test(expected = EntityExistsException.class)
  public void
      test_insertFriendrequest_shouldThrowEntityExistsException() {

    User sender = new User("senderName", "sender@somemail.com");
    User receiver = new User("receiverName",
        "reciever@somemail.com");

    /*
     * creating precondition (1), (2), (3)
     */
    sender = userService.insertUser(sender);
    receiver = userService.insertUser(receiver);

    /*
     * creating initial condition
     */
    Friendrequest f1 = new Friendrequest(sender, receiver);
    friendrequestService.insertFriendrequest(f1);

    /*
     * user input
     * EntityExistsException should be thrown here
     */
    Friendrequest friendrequest = new Friendrequest(sender,
        receiver);
    friendrequest =
        friendrequestService.insertFriendrequest(friendrequest);
```

```
  List<Friendrequest> allFriendrequests = (List<Friendrequest>)
      friendrequestService
        .listFriendrequest().getItems();

  assertTrue(allFriendrequests.contains(f1));
}
```

Test case 2

```
@Test
public void test_insertFriendrequest_shouldOK() {

  User sender = new User("senderName", "sender@somemail.com");
  User receiver = new User("receiverName",
      "reciever@somemail.com");

  /*
   * creating precondition (1), (2), (3)
   */
  sender = userService.insertUser(sender);
  receiver = userService.insertUser(receiver);

  /*
   * no operation neccessary to create initial condition
   */

  /*
   * user input
   */
  Friendrequest friendrequest = new Friendrequest(sender,
      receiver);
  friendrequest =
      friendrequestService.insertFriendrequest(friendrequest);//
      "what needs to be shown" part(1)

  /*
   * "what needs to be shown" part (2)
   */
  assertTrue(friendrequest.getUserFrom().equals(sender) &&
      friendrequest.getUserTo().equals(receiver));

  /*
   * "what needs to be shown part" (3)
   */
  assertTrue(friendrequestService.listFriendrequestsFromUser(sender.getId()).
    getItems()
      .contains(friendrequest));
```

```
    assertTrue(friendrequestService.listFriendrequestsToUser(receiver.getId()).
      getItems()
        .contains(friendrequest));

    /*
     * "what is needed to be shown" part (4)
     */
    User unauthorized = new User("unauth", "unauth");
    unauthorized = userService.insertUser(unauthorized);

    assertTrue(!friendrequestService.listFriendrequestsFromUser(
    unauthorized.getId()).getItems().contains(unauthorized) &&
        !friendrequestService.listFriendrequestsToUser(
        unauthorized.getId()).getItems().contains(unauthorized));

    /*
     * showing post conditions
     */
    assertTrue(userService.getUser(sender.getId()).equals(sender));
    assertTrue(userService.getUser(receiver.getId()).equals(receiver));
  }
```

B.2.4 Delete a friend request

Test case 1

```
  @Test
  public void test_deleteFriendrequest_shouldOK() {
    User sender = new User("user1", "email1");
    User receiver = new User("user2", "email2");

    /*
     * creating preconditions
     * (1) (2) (3)
     */
    sender = userService.insertUser(sender);
    receiver = userService.insertUser(receiver);

    /*
     * creating preconditions
     * (4)
     */
    Friendrequest f1 = new Friendrequest(sender, receiver);
    friendrequestService.insertFriendrequest(f1);

    /*
     * user input
```

```
     */
    friendrequestService.removeFriendrequest(f1.getId());

    List <Friendrequest> allFriendrequests = (List<Friendrequest>)
        friendrequestService.listFriendrequest().getItems();

    /*
     * What needs to be shown (1), and post condition (4)
     */
    assertFalse(allFriendrequests.contains(f1));

    /*
     * showing post conditions
     */
    assertTrue(userService.getUser(sender.getId()).equals(sender));//(1)
    assertTrue(userService.getUser(receiver.getId()).equals(receiver));//(2)
    assertFalse(sender.equals(receiver));//(3)

}
```

B.2.5 Delete a friendship

Test case 1

```
@Test
public void test_deleteFriendship_shouldOK() {
    /*
     * creating preconditions
     * (1) (2)
     */
    User friendA = new User("user1", "email1");
    User friendB = new User("user2", "email2");

    friendA = userService.insertUser(friendA);
    friendB = userService.insertUser(friendB);

    /*
     * creating preconditions
     * (3)
     */
    Friendship friendship = new Friendship(friendA, friendB);
    friendship = friendshipService.insertFriendship(friendship);

    /*
     * user input
     */
    friendshipService.removeFriendship(friendship.getId());
```

```
    /*
     * What needs to be shown
     * (1)
     * and Post condition (3)
     */
    List<Friendship> allFriendships = (List<Friendship>)
        friendshipService.listFriendship().getItems();

    assertFalse(allFriendships.contains(friendship));

    /*
     * showing post conditions
     * (1) (2)
     */
    assertTrue(userService.getUser(friendA.getId()).equals(friendA)
        && userService.getUser(friendB.getId()).equals(friendB));

  }
```

B.2.6 Creating a post

Test case 1

```
@Test
  public void test_createSimplePost_shouldOK() {

    User postOwner = new User("user1","email1");

    /*
     * simple post creation
     */
    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setLikesCount(0l);
    photo1.setPath("path1");

    Photo photo2 = new Photo();
    photo2.setDescription("photo2");
    photo2.setLikesCount(0l);
    photo2.setPath("path2");

    User postViewer1 = new User("user2", "email2");
```

```
User postViewer2 = new User("user3", "email3");
User unauhtorized = new User("unauth","unauthmail");

List<Photo> photos = new ArrayList<Photo>();
photos.add(photo1);
photos.add(photo2);

List<User> postViewers = new ArrayList<User>();
postViewers.add(postViewer1);
postViewers.add(postViewer2);

simplePost1.setPhotos(photos);
simplePost1.setUserCanSeePost(postViewers);

/*
 * creating preconditions (1) (2)
 */
postOwner = userDAO.insertUser(postOwner);
postViewer1 = userDAO.insertUser(postViewer1);
postViewer2 = userDAO.insertUser(postViewer2);
unauhtorized = userDAO.insertUser(unauhtorized);

/*
 * user input
 */
simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

/*
 * "what is needed to be shown" part
 * (2) - post must be visible for all post viewers
 */
List<SimplePost> postViewer1Posts = (List<SimplePost>)
    simplePostDAO.listSimplePostsForUser(postViewer1.getId()).getItems();
List<SimplePost> postViewer2Posts = (List<SimplePost>)
    simplePostDAO.listSimplePostsForUser(postViewer2.getId()).getItems();

assertTrue(postViewer1Posts.contains(simplePost1));
assertTrue(postViewer2Posts.contains(simplePost1));

/*
 * (2) - post must be visible for post owner
 */
List<SimplePost> postOwnerPosts = (List<SimplePost>)
    simplePostDAO.listSimplePostsOfUser(postOwner.getId()).getItems();

assertTrue(postOwnerPosts.contains(simplePost1));

/*
 * (3) - post cannot be visible for unauthorized
```

```
     */
    List<SimplePost> unauthorizedPosts = (List<SimplePost>)
       simplePostDAO.listSimplePostsForUser(unauhtorized.getId()).getItems();

    assertFalse(unauthorizedPosts.contains(simplePost1));

    /*
     * showing post conditions
     * (2) - post exists is implied by the part above)
     * (1)
     */
    assertTrue(userDAO.getUser(postOwner.getId()).equals(postOwner));

    /*
     * (3)
     */
    assertTrue(userDAO.getUser(postViewer1.getId()).equals(postViewer1));
    assertTrue(userDAO.getUser(postViewer2.getId()).equals(postViewer2));

  }
```

B.2.7 Updating a post

Test case 1

```
@Test
  public void test_updateSimplePost1_shouldOK() {
    User postOwner = new User("user1", "email1");

    /*
     * simple post creation
     */
    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setLikesCount(0l);
    photo1.setPath("path1");

    Photo photo2 = new Photo();
    photo2.setDescription("photo2");
    photo2.setLikesCount(0l);
    photo2.setPath("path2");

    User postViewer1 = new User("user2", "email2");
```

```
User postViewer2 = new User("user3", "email3");
User unauhtorized = new User("unauth", "unauthmail");

List<Photo> photos = new ArrayList<Photo>();
photos.add(photo1);
photos.add(photo2);

List<User> postViewers = new ArrayList<User>();
postViewers.add(postViewer1);
postViewers.add(postViewer2);

simplePost1.setPhotos(photos);
simplePost1.setUserCanSeePost(postViewers);

/*
 * creating preconditions (1) (2) (3)
 */
postOwner = userDAO.insertUser(postOwner);
postViewer1 = userDAO.insertUser(postViewer1);
postViewer2 = userDAO.insertUser(postViewer2);
unauhtorized = userDAO.insertUser(unauhtorized);
simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

/*
 * UPDATE PART
 * BEGIN
 */
/*
 * remove photo
 */
simplePost1.getPhotos().remove(photo1);

/*
 * user input
 */
simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
/*
 * UPDATE PART
 * END
 */

/*
 * what is needed to be shown (2) - post must be visible for all
    post
 * viewers
 */
List<SimplePost> postViewer1Posts = (List<SimplePost>)
   simplePostDAO.listSimplePostsForUser(
    postViewer1.getId()).getItems();
```

```
    List<SimplePost> postViewer2Posts = (List<SimplePost>)
        simplePostDAO.listSimplePostsForUser(
         postViewer2.getId()).getItems();

    assertTrue(postViewer1Posts.contains(simplePost1));
    assertTrue(postViewer2Posts.contains(simplePost1));

    /*
     * (2) - post must be visible for post owner
     */
    List<SimplePost> postOwnerPosts = (List<SimplePost>)
        simplePostDAO.listSimplePostsOfUser(
         postOwner.getId()).getItems();

    assertTrue(postOwnerPosts.contains(simplePost1));

    /*
     * (3) - post cannot be visible for unauthorized
     */
    List<SimplePost> unauthorizedPosts = (List<SimplePost>)
        simplePostDAO.listSimplePostsForUser(
         unauhtorized.getId()).getItems();

    assertFalse(unauthorizedPosts.contains(simplePost1));

    /*
     * showing post conditions
     * (1)
     */
    assertTrue(userDAO.getUser(postOwner.getId()).equals(postOwner));

    /*
     * (2)
     */
    assertTrue(userDAO.getUser(postViewer1.getId()).equals(postViewer1));
    assertTrue(userDAO.getUser(postViewer2.getId()).equals(postViewer2));
    /*
     * (3)
     */
    assertTrue(simplePostDAO.getSimplePost(simplePost1.getId()).
    equals(simplePost1));

  }
```

The other test cases are the same only the update part is other:

Test case 2

```
/*
```

```
     * UPDATE PART
     * BEGIN
     */
    /*
     * remove photo
     */
    simplePost1.getPhotos().remove(photo1);
    /*
     * add photo
     */
    Photo photo3 = new Photo();
    photo3.setDescription("photo3");
    photo3.setLikesCount(0l);
    photo3.setPath("path3");

    simplePost1.getPhotos().add(photo3);

    /*
     * user input
     */
    simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
    /*
     * UPDATE PART
     * END
     */
```

Test case 3

```
/*
     * UPDATE PART
     * BEGIN
     */
    /*
     * remove photo
     */
    simplePost1.getPhotos().remove(photo1);
    simplePost1.setDescription("new description");

    /*
     * user input
     */
    simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
    /*
     * UPDATE PART
     * END
     */
```

Test case 4

```
/*
     * UPDATE PART
     * BEGIN
     */
    /*
     * remove photo
     */
    simplePost1.getPhotos().remove(photo1);
    simplePost1.setDescription("new description");
    /*
     * add photo
     */
    Photo photo3 = new Photo();
    photo3.setDescription("photo3");
    photo3.setLikesCount(0l);
    photo3.setPath("path3");

    simplePost1.getPhotos().add(photo3);
    /*
     * user input
     */
    simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
    /*
     * UPDATE PART
     * END
     */
```

Test case 5

```
/*
     * UPDATE PART
     * BEGIN
     */

    simplePost1.setDescription("new description");
    /*
     * user input
     */
    simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
    /*
     * UPDATE PART
     * END
     */
```

Test case 6

```
/*
     * UPDATE PART
     * BEGIN
     */
    /*
     * remove photo
     */
    simplePost1.setDescription("new description");
    /*
     * add photo
     */
    Photo photo3 = new Photo();
    photo3.setDescription("photo3");
    photo3.setLikesCount(0l);
    photo3.setPath("path3");

    simplePost1.getPhotos().add(photo3);
    /*
     * user input
     */
    simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
    /*
     * UPDATE PART
     * END
     */
```

Test case 7

```
/*
     * UPDATE PART
     * BEGIN
     */
    /*
     * add photo
     */
    Photo photo3 = new Photo();
    photo3.setDescription("photo3");
    photo3.setLikesCount(0l);
    photo3.setPath("path3");

    simplePost1.getPhotos().add(photo3);
    /*
     * user input
     */
    simplePost1 = simplePostDAO.updateSimplePost(simplePost1);
    /*
```

```
     * UPDATE PART
     * END
     */
```

B.2.8 Deleting a post

Test case 1

```
@Test
public void test_deleteSimplePost_shouldOK() throws IOException,
    GeneralSecurityException {

  User postOwner = new User("user1", "email1");

  /*
   * simple post creation
   */
  SimplePost simplePost1 = new SimplePost();
  simplePost1.setDescription("post1");
  simplePost1.setPostedBy(postOwner);

  Photo photo1 = new Photo();
  photo1.setDescription("photo1");
  photo1.setLikesCount(0l);
  photo1.setPath("path1");

  Photo photo2 = new Photo();
  photo2.setDescription("photo2");
  photo2.setLikesCount(0l);
  photo2.setPath("path2");

  User postViewer1 = new User("user2", "email2");
  User postViewer2 = new User("user3", "email3");
  User unauhtorized = new User("unauth", "unauthmail");

  List<Photo> photos = new ArrayList<Photo>();
  photos.add(photo1);
  photos.add(photo2);

  List<User> postViewers = new ArrayList<User>();
  postViewers.add(postViewer1);
  postViewers.add(postViewer2);

  simplePost1.setPhotos(photos);
  simplePost1.setUserCanSeePost(postViewers);

  /*
```

```
     * creating preconditions (1) (2) (3)
     */
    postOwner = userDAO.insertUser(postOwner);
    postViewer1 = userDAO.insertUser(postViewer1);
    postViewer2 = userDAO.insertUser(postViewer2);
    unauhtorized = userDAO.insertUser(unauhtorized);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);


    /*
     * user input
     */
    simplePostDAO.removeSimplePost(simplePost1.getId());

    /*
     * what is needed to be shown:
     */
    assertFalse(simplePostDAO.listSimplePost().getItems().contains(simplePost1));

    /*
     * showing post conditions:
     * Post condition (3) is shown in the assert above
     * (1), (2)
     */
    assertTrue(userDAO.getUser(postOwner.getId()).equals(postOwner));
    assertTrue(userDAO.getUser(postViewer1.getId()).equals(postViewer1));
    assertTrue(userDAO.getUser(postViewer2.getId()).equals(postViewer2));

  }
```

B.2.9 Disliking a photo within post

Test case 1

```
@Test
  public void test_insertDislikes_shouldOK() throws
     LikeAlreadyExistsException, DislikeAlreadyExistsException,
     UnauthorizedDisLikeException {
    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User postViewer = new User("user3", "email123");
    User dislikeCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
```

```
simplePost1.setPostedBy(postOwner);

Photo photo1 = new Photo();
photo1.setDescription("photo1");
photo1.setPath("gg");

List<Photo> photos = new ArrayList<Photo>();
photos.add(photo1);
simplePost1.setPhotos(photos);

List<User> postViewers = new ArrayList<User>();
postViewers.add(dislikeCreator);
postViewers.add(postViewer);
simplePost1.setUserCanSeePost(postViewers);

postOwner = userDAO.insertUser(postOwner);
dislikeCreator = userDAO.insertUser(dislikeCreator);
postViewer = userDAO.insertUser(postViewer);
simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

/*
 * user input
 */
Dislikes dislikes1 = new Dislikes();
dislikes1.setPhoto(photo1);
dislikes1.setUser(dislikeCreator);

/*
 * what is needed to be shown part
 */
dislikes1 = dislikesDAO.insertDislikes(dislikes1); // (1)
   dislike was returned to post viewer

/*
 * dislikes1 is Dislike at photo of a post
 * (2)
 */
assertTrue(dislikes1.getPhoto().equals(photo1));

/*
 * (3) dislikes1 must be visible for post viewers
 */
List<SimplePost> postVisibleForPostViewer = (List<SimplePost>)
   simplePostDAO.listSimplePostsForUser(postViewer.getId()).getItems();

for(SimplePost sp : postVisibleForPostViewer) {
  if(sp.getPhotos().contains(photo1)) {
    List<Dislikes> dislikesOfPhoto = (List<Dislikes>)
        dislikesDAO.listDislikesOfPhoto(photo1.getId(), 0,
```

```
            0).getItems();
        assertTrue(dislikesOfPhoto.contains(dislikes1));
      }
    }

    /*
     * showing post conditions
     *
     */
    assertTrue(simplePostDAO.getSimplePost(simplePost1.getId()).equals(simplePost1));
        // (1)
    assertTrue(userDAO.getUser(dislikeCreator.getId()).equals(dislikeCreator));//
        (2)
    assertTrue(userDAO.getUser(postViewer.getId()).equals(postViewer));//
        (3)

  }
```

Test case 2

```
@Test(expected = DislikeAlreadyExistsException.class)
  public void
      test_insertDislikes_shouldThrowDislikeAlreadyExistsException()
      throws LikeAlreadyExistsException,
      DislikeAlreadyExistsException, UnauthorizedDisLikeException {

    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User dislikeCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    simplePost1.setPhotos(photos);

    List<User> postViewers = new ArrayList<User>();
    postViewers.add(dislikeCreator);
    simplePost1.setUserCanSeePost(postViewers);
```

```
    postOwner = userDAO.insertUser(postOwner);
    dislikeCreator = userDAO.insertUser(dislikeCreator);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    /*
     * creating initial condition
     */
    Dislikes dislikes1 = new Dislikes();
    dislikes1.setPhoto(photo1);
    dislikes1.setUser(dislikeCreator);

    dislikes1 = dislikesDAO.insertDislikes(dislikes1);

    /*
     * user input
     */
    Dislikes dislikes2 = new Dislikes();
    dislikes2.setPhoto(photo1);
    dislikes2.setUser(dislikeCreator);

    /*
     * DislikeAlreadyExistsException should be thrown here
     */
    dislikes2 = dislikesDAO.insertDislikes(dislikes2);
  }
```

Test case 3

```
@Test(expected = LikeAlreadyExistsException.class)
  public void
      test_insertDislikes_shouldThrowDisdislikeAlreadyExistsException()
      throws LikeAlreadyExistsException,
      DislikeAlreadyExistsException, UnauthorizedDisLikeException {

    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User dislikeCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");
```

```
    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    simplePost1.setPhotos(photos);

    List<User> postViewers = new ArrayList<User>();
    postViewers.add(dislikeCreator);
    simplePost1.setUserCanSeePost(postViewers);

    postOwner = userDAO.insertUser(postOwner);
    dislikeCreator = userDAO.insertUser(dislikeCreator);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    /*
     * creating initial condition
     */
    Likes likes1 = new Likes();
    likes1.setPhoto(photo1);
    likes1.setUser(dislikeCreator);

    likes1 = likesDAO.insertLikes(likes1);

    /*
     * user input
     */
    Dislikes dislikes2 = new Dislikes();
    dislikes2.setPhoto(photo1);
    dislikes2.setUser(dislikeCreator);

    /*
     * LikeAlreadyExistsException should be thrown here
     */
    dislikes2 = dislikesDAO.insertDislikes(dislikes2);
  }
```

Test case 4

```
@Test(expected = UnauthorizedDisLikeException.class)
  public void test_insertLikes_shouldThrowUnauthorizedDisLike()
      throws DislikeAlreadyExistsException,
      LikeAlreadyExistsException, UnauthorizedDisLikeException {

    /*
     * creating preconditions
     */
    User postOwner = new User("user1","email1");
    User nonPostViewer = new User("user2","email2");
    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
```

```
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    simplePost1.setPhotos(photos);

    postOwner = userDAO.insertUser(postOwner);
    nonPostViewer = userDAO.insertUser(nonPostViewer);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    Dislikes dislikes1 = new Dislikes();
    dislikes1.setPhoto(photo1);
    dislikes1.setUser(nonPostViewer);

    /*
     * user input
     * UnauthorizedDisLikeException should be thrown here
     */
    dislikes1 = dislikesDAO.insertDislikes(dislikes1);
  }
```

B.2.10 Commenting a photo within post

Test case 1

```
@Test
  public void test_insertComment_shouldOK() {
    /*
     * Creating preconditions
     */
    User postOwner = new User("user1","email1");
    User postViewer = new User("user3", "email123");
    User commentCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
```

```
photos.add(photo1);
simplePost1.setPhotos(photos);

List<User> postViewers = new ArrayList<User>();
postViewers.add(commentCreator);
postViewers.add(postViewer);
simplePost1.setUserCanSeePost(postViewers);

postOwner = userDAO.insertUser(postOwner);
commentCreator = userDAO.insertUser(commentCreator);
postViewer = userDAO.insertUser(postViewer);
simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

/*
 * user input
 */
Comment comment = new Comment();
comment.setPhoto(photo1);
comment.setUser(commentCreator);
comment.setTimestamp(System.currentTimeMillis());

/*
 * what is needed to be shown part
 */
comment = commentDAO.insertComment(comment); // (1) comment was
    returned to the post viewer(comment creator)

/*
 * comment is Comment at photo of a post
 * (2)
 */
assertTrue(comment.getPhoto().equals(photo1));

/*
 * comment must be visible for post viewers of comment' post (3)
 */
List<SimplePost> postVisibleForPostViewer = (List<SimplePost>)
    simplePostDAO.listSimplePostsForUser(postViewer.getId()).getItems();

for(SimplePost sp : postVisibleForPostViewer) {
  if(sp.getPhotos().contains(photo1)) {
    List<Comment> dislikesOfPhoto = (List<Comment>)
        commentDAO.listNewCommentsOfPhoto(photo1.getId(), 0,
        0).getItems();
    assertTrue(dislikesOfPhoto.contains(comment));
  }
}

/*
```

```
     * showing post conditions
     */
    assertTrue(simplePostDAO.getSimplePost(simplePost1.getId()).
    equals(simplePost1)); // (1)
    assertTrue(userDAO.getUser(commentCreator.getId()).equals(commentCreator));//
        (2)
    assertTrue(userDAO.getUser(postViewer.getId()).equals(postViewer));//
        (3)
  }
```

Test case 2

```
@Test(expected = Exception.class)
  public void test_insertComment_shouldThrowException() {
    /*
     * Creating preconditions
     */
    User postOwner = new User("user1","email1");
    User postViewer = new User("user3", "email123");
    User commentCreator = new User("user2","email2");

    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    simplePost1.setPhotos(photos);

    List<User> postViewers = new ArrayList<User>();
    postViewers.add(postViewer);
    simplePost1.setUserCanSeePost(postViewers);

    postOwner = userDAO.insertUser(postOwner);
    commentCreator = userDAO.insertUser(commentCreator);
    postViewer = userDAO.insertUser(postViewer);
    simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

    /*
     * user input
     */
    Comment comment = new Comment();
    comment.setPhoto(photo1);
    comment.setUser(commentCreator);
```

```
    comment.setTimestamp(System.currentTimeMillis());

    /*
     * The Exception should be thrown here
     */
    comment = commentDAO.insertComment(comment); // (1) comment was
        returned to the post viewer(comment creator)

}
```

B.2.11 Multiple likes within one post

Test case

```
    SimplePost simplePost1 = new SimplePost();
    simplePost1.setDescription("post1");
    simplePost1.setPostedBy(postOwner);

    Photo photo1 = new Photo();
    photo1.setDescription("photo1");
    photo1.setPath("gg");

    Photo photo2 = new Photo();
    photo2.setDescription("photo1");
    photo2.setPath("gg");

    List<Photo> photos = new ArrayList<Photo>();
    photos.add(photo1);
    photos.add(photo2);
    simplePost1.setPhotos(photos);

    ...
    /*
     * user input
     */
    Likes likes1 = new Likes();
    likes1.setPhoto(photo1);
    likes1.setUser(likeCreator);

    Likes likes2 = new Likes();
    likes2.setPhoto(photo2);
    likes2.setUser(likeCreator);

    likes1 = likesDAO.insertLikes(likes1);
    likes2 = likesDAO.insertLikes(likes2); // the exception should
        be thrown here
```

B.2.12 Dislike a photo within post containing multiple photos

Test case

```
@Test(expected = Exception.class)
  public void test_insertDislikes_shouldThrowExpection() throws
      LikeAlreadyExistsException, DislikeAlreadyExistsException,
      UnauthorizedDisLikeException {
     /*
      * creating preconditions
      */
     User postOwner = new User("user1","email1");
     User postViewer = new User("user3", "email123");
     User dislikeCreator = new User("user2","email2");

     SimplePost simplePost1 = new SimplePost();
     simplePost1.setDescription("post1");
     simplePost1.setPostedBy(postOwner);

     Photo photo1 = new Photo();
     photo1.setDescription("photo1");
     photo1.setPath("gg");

     Photo photo2 = new Photo();
     photo2.setDescription("photo1");
     photo2.setPath("gg");

     List<Photo> photos = new ArrayList<Photo>();
     photos.add(photo1);
     photos.add(photo2);
     simplePost1.setPhotos(photos);

     List<User> postViewers = new ArrayList<User>();
     postViewers.add(dislikeCreator);
     postViewers.add(postViewer);
     simplePost1.setUserCanSeePost(postViewers);

     postOwner = userDAO.insertUser(postOwner);
     dislikeCreator = userDAO.insertUser(dislikeCreator);
     postViewer = userDAO.insertUser(postViewer);
     simplePost1 = simplePostDAO.insertSimplePost(simplePost1);

     /*
      * user input
      */
     Dislikes dislikes1 = new Dislikes();
     dislikes1.setPhoto(photo1);
     dislikes1.setUser(dislikeCreator);
```

```
    dislikes1 = dislikesDAO.insertDislikes(dislikes1); // the
        exception should be thrown here

  }
```