POLITECNICO DI MILANO

Department of Elettronic, Information ad Biomedical Engineering

Master Degree course in Computer Science Engineering

METEOCAL
Requirement Analysis and Specification Document
(RASD)

Instructor: Prof. Elisabetta Di Nitto

Authors:

Marco Arnaboldi matr. 824306

Andrea Luigi Edoardo Caielli matr. 836789

Accademic Year 2014–2015

Contents

Abstract

The main task of this document is to give a specification of the requirements that our system has to fulfil adopting the IEEE-STD-830-1993 standard for RASD documentation . It also introduces the functional and non-functional requirements via UML diagrams and a high level specification of the system. In the last part of this document it presents the formal model of the specification using Alloy analysis.

The information in this document are intended for the stakeholders and the developers of the project. For the stakeholders this document presents a description useful to understand the project development, meanwhile for the developers it's an useful way to show the matching between the stakeholders' requests and the developed solution.

Chapter 1

Introduction

In this chapter, it will be displayed the main purpose of the project, a general description of the problem and a glossary which explains how to interpret the terms that will be used from now on.

1.1 Purpose

The system is a software called MeteoCal. It's a weather based on-line calendar for helping people scheduling their own events, avoiding bad weather condition in case of outdoor activities.

1.2 Scope

Users, once registered, should be able to create, delete and update events. An event should contain information about when and where the event will take place, whether the event will be indoor or outdoor. During event creation, any number of registered users can be invited. Only the organizer will be able to update or delete the event. Invited users can only accept or decline the invitation. Whenever an event is saved, the system should enrich the event with weather forecast information (if available). Moreover, it should notify all event participants one day before the event in case of bad weather conditions for outdoor events. Notifications are received by the users when they log into the system. A user A should be allowed to make his/her calendar public, that is, visible to all other registered users. These last ones will see all the time slots in which A is busy but without seeing the details of the corresponding events, unless they have been defined as public. Events can be defined as public or private by their owners, upon creation. If an event is public, all the registered users can see its details, including the corresponding participants. In case of bad weather conditions for outdoor events, three days before the event, the system should propose to its creator the closest (in time) sunny day (if any).

1.3 Stakeholders

If the MeteoCal project is intended as an academic exercise the stakeholder is the professor who gave us the project. In this case the stakeholder expects to receive a working software that fulfils the requests and the documentation about requirements analysis, design, development, testing, project reporting and a final presentation. The purpose of the stakeholder is to evaluate the ability of the students and level of comprehension of the subject. On the other hand, if MeteoCal is intended as a real business project the stakeholder is the "X software house". Its needs are the same of the previous case concerning the documentations, but the application has to be more complete and user friendly due the market competition. In this case the purpose of the stakeholder is to sell the product and maximize its profit.

1.4 Glossary

To avoid misunderstanding and ambiguity, the following definitions and acronyms here listed represent the meaning that the developers associated to these common words in the context of the project:

- **Guest**: a person that isn't enrolled or logged in the system (likely it's his first visit).
- **User**: a person that is logged in the system.
- **Event**: description of an activity in a certain date.
- **WFI**: acronym for "Weather Forecast Information".
- **WeatherBot**: it's a neologism to define the service which provides the WFI.
- **Organizer**: an user that creates an event.
- **Invitation**: the request to attend an event sent from the event's organizer to an user.
- **Feedback**: the reply to an invitation. It is a yes or no response plus an optional comment.
- **Alert**: a system internal message to warn an user about invitation, bad WFI or feedback.
- **Favourites**: set of users defined by an user A, to speed up the invitation phase.

Chapter 2

Overall Description

This chapter defines: the aim of the product, a summary of the major functions that should be provided, the user characteristics, the constraints and the assumptions over the domain, the actors involved and a list of non-functional requirements.

2.1 Product Perspective

The proposed solution is a partially independent web application, because the calendar management is self contained but the WFI are provided from an external service. As already said the client will be built as a web application. In this way the users will be able to access via Internet, using a common browser (e.g. Chrome). The reasons of this choice will be discussed in a following sections.

2.2 User Characteristics

The application has no specific user target, since everyone with a basic web knowledge will be able to benefit from the service. The user has registered because he needs an application that gives him the opportunity to schedule his commitments and invitations and at the same time keep an eye on the weather condition for his outdoor events.

2.3 Domain Assumption

In the given description of the problem there are some ambiguities. Some integrative hypothesis had been added to solve them:

- *User identification*. The main issue is to attest the uniqueness of the user. The solution is provided by the e-mail address that is by definition unique.
- *User Research*. The research could be made by e-mail or by nickname only between users.
- *Ubiquity*. An user could have overlapped events scheduled in his calendar. Example: user A has an events E1 from 4.00 pm to 6.00 pm and another event E2 from 5.00 pm to 6.00 pm.
- *Invitation deadline*. When an invitation is sent it lasts until the beginning of the related event. After that the invitation is invalid.
- *Feedback*. To an invitation follows at most one feedback.
- *Smart organization*. An organizer can organize only events that are not overlapped.
- *Self invitation*. An organizer can't invite himself to his events.
- *Invitation sending*. An organizer can invite users during the event creation phase and when the event is saved, until the event's start.
- *WFI correctness*. The WFI that are received from the WeatherBot are 100% correct.

- *Lone organizer*. An event can be organized by only one user.
- *Calendar Unicity*. One an user can have only one calendar.
- *No second thoughts*. If the answer to an invitation is no, the user can not change it in a second time.
- *No sneak in event*. An user can participate to an event only if he's the creator or has received an invite.

2.4 Constraints and General Assumption

The main constraint lies in the obligation to use the J2EE platform and the Enterprise Java Beans (EJB) in the application development. It can be assumed that users will have a browser and an Internet connection. About other interfaces, MeteoCal needs a DBMS (DataBase Management System) to store the persistent data. A possible improvement of the system cold be made by providing an access to a SMTP server to send confirmation e-mail for the registration and to notify the alerts.

2.5 Actors and Functionalities

The actors of the MeteoCal system are:

- **A1 Guest**: a guest is a person who is not already enrolled into the service. The only two functionalities that a guest is enable to do are the registration (for the first time) and the log in.
- **A2 User**: an user is a person who is already enrolled. He is enable to:
  - Check his calendar
  - Search for other users
  - Create events

  - Manage invitation
  - Manage personal info
  - Read alerts

- **A3 Organizer**: It is a specification of user. An user become an organizer when he creates an event. He is enable to:
  - Invite to his own events
  - Modify and delete his own events
    Note: modification, cancellation and invitation are functionalities that pre-request an event creation. This concept will be better explained in chapter 3.

- **A4 WeatherBot**: It is the weather forecast server provider. Its only functionality is to provide the WFI to the system.

2.6 Non Functional Requirements

Some non functional requirements have been found in order to fulfil some qualities and a correct operation flow that the system should provide:

- *User friendliness*. The use interface has to be as simple and intuitive as possible. This because the system doesn't aspect an user with a wide previous knowledge of it.
- *Portability*. The client has to be compatible to all the major hardware and software platform on the market, this is accomplished using the web application solution presented early.
- *Performance*. To supply suitable service, the system has to be reactive and able to answer to a high number of also simultaneous requests. Because of this the interaction between the client and the server has to be reduced to a minimum, in order to no overload the net.

- *Reliability*. The system should be able to guarantee the service 24/24, 7/7. The server farm should allow every day maintenance without compromise the functionality. However the system doesn't cover a critical function, so brief unavailability could be acceptable.

- *Data integrity, consistency and availability*. Data have to be always accessible, so the system has to provide always an access to them in normal condition. They also have to be duplicate in order to avoid data losses in case of system fault. For instance the use of RAID, mirroring and backup could present a valid solution.

- *Security*. To ensure that private events, calendars and sensible data remain private, the system should offer different security measures. First of all all the account will be protected by the couple personal e-mail plus personal password. Secondly, since the system allows the insertion of data, such data should be filtered in order to avoid vicious or unwanted modification in the data base (e.g. SQL injection). At last to protect the on-line transmission of data an HTTPS protocol could be used. However this last proposal should be delayed to a future implementation because the HTTPS protocol needs the purchase of a certificate signed by recognized certification authority.

Chapter 3

Functional Requirements and Scenarios

This chapter presents the functional requirements of the system, already introduced in the previous chapter functionalities. At the beginning they will be presented in natural language as scenarios, then from these scenarios a higher level description will be abstracted. This last section will present the use-cases in the form of UML and sequence diagrams.

3.1 Scenarios Natural Language Description

Has an example, the story of Aldo is shown.

3.1.1 System Registration

Aldo is new in the system and wants to register. He clicks on the sign in button and inserts his data. Then he submits the request to the system. If the registration is successful Aldo will be redirected to a log in page, otherwise an error message will be displayed.

3.1.2 Log in

Aldo, who is now registered, wants to log in. He clicks on the login button and he's asked to insert his own e-mail and password. If the combination is correct he can enter the system, otherwise an error will appear.

3.1.3 Profile Management

Aldo is now logged in. He decide to change his nickname in the public profile. He clicks on his profile link and then changes the nickname field. After he has completed the changes he clicks on the apply button to submit the changes.

3.1.4 Create an Event

It's Monday and Aldo wants to schedule his Saturday's picnic in his calendar. he clicks on the create event link, then he inserts the information about the even (when, where ...) and submit the requests. The system saves his event in his calendar and adds the WFI.

3.1.5 Invitation

Aldo wants to invite his friend Bruno (that is already a user) to his picnic. He click on the event, a menu appears and he clicks on invite. He inserts in the search engine the nickname or the email of his friend one at time. After the system gives back the results of the research he selects the friend among the results by clicking on their names. Then the system sends an invitation. Aldo logs out.

3.1.6 Invitation acceptance

Bruno logs in the system and checks his calendar. He notices an alert. He clicks on it and more details are shown. It is an invitation to a picnic from

Aldo. He accepts the invitation by clicking on the yes button and adds a little comment.

3.1.7 Check Calendar

On Friday Aldo logs in again. He notices two alert. He clicks on the first one, that is the acceptance from Bruno to his picnic event. The other one is a system warning about bad weather condition for Saturday and suggests him to move the event on Sunday.

3.1.8 Modify Event

Aldo decides to move his picnic on Sunday. He clicks on the picnic event on the calendar and than modifies from the just appeared menu. He changes the date to Sunday and applies the change. When Bruno will log in, he will receive an alerts about the modification.

3.2 Functional Requirements Scheme Description

In this section the requirements shown above, will be presented in a more formal way, using also UML state and sequence diagrams. The follow requirements are deducted both domain assumption and system goals. They are the bridge that will permit the achievement of the goal considering the domain assumptions.

3.2.1 Use Case Diagram

The Figure 3.1 gives an overview on the main actors involved in the system and the functionalities that they are able to execute.

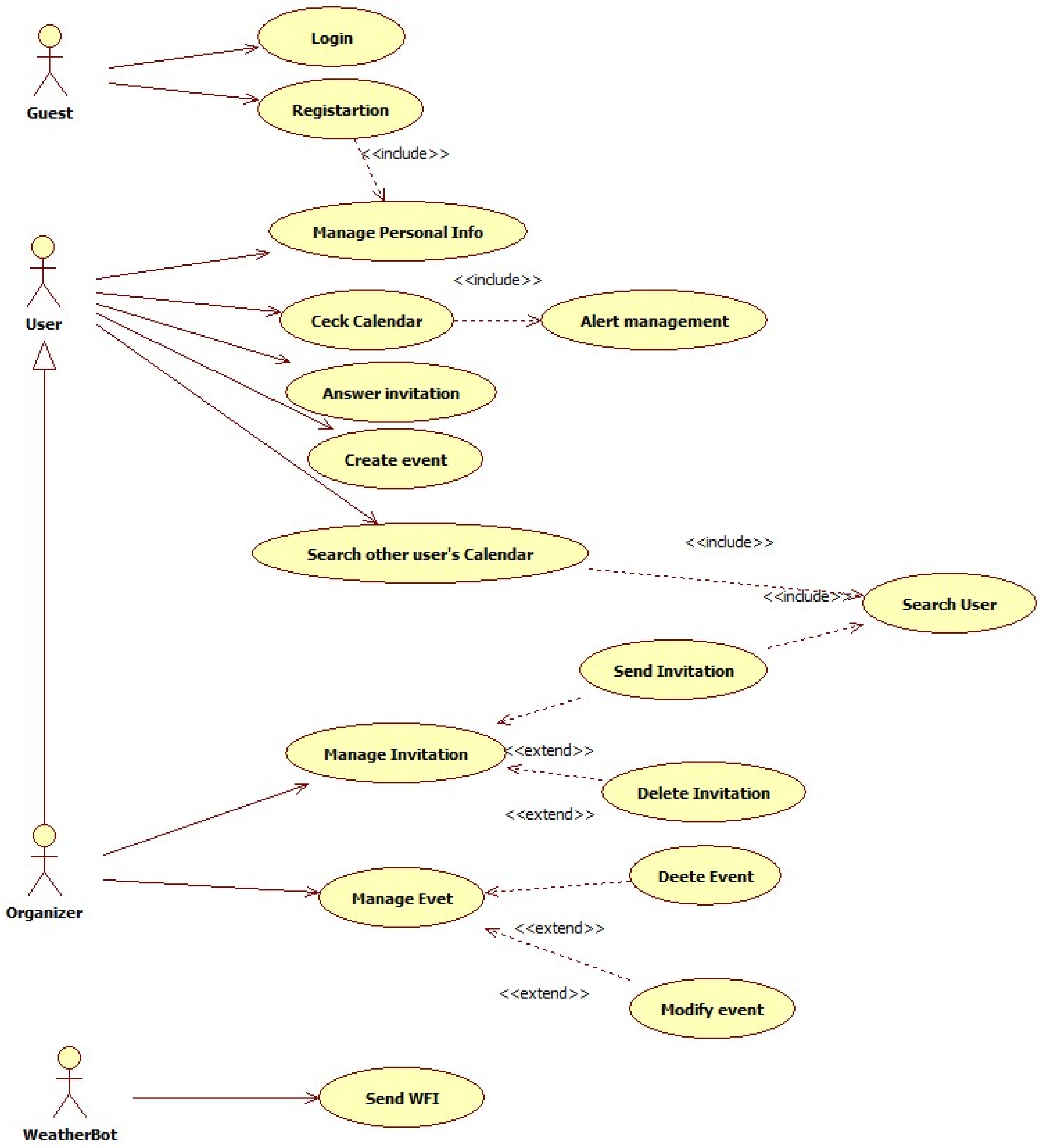


Figure 3.1: Use Case Diagram for MeteoCal

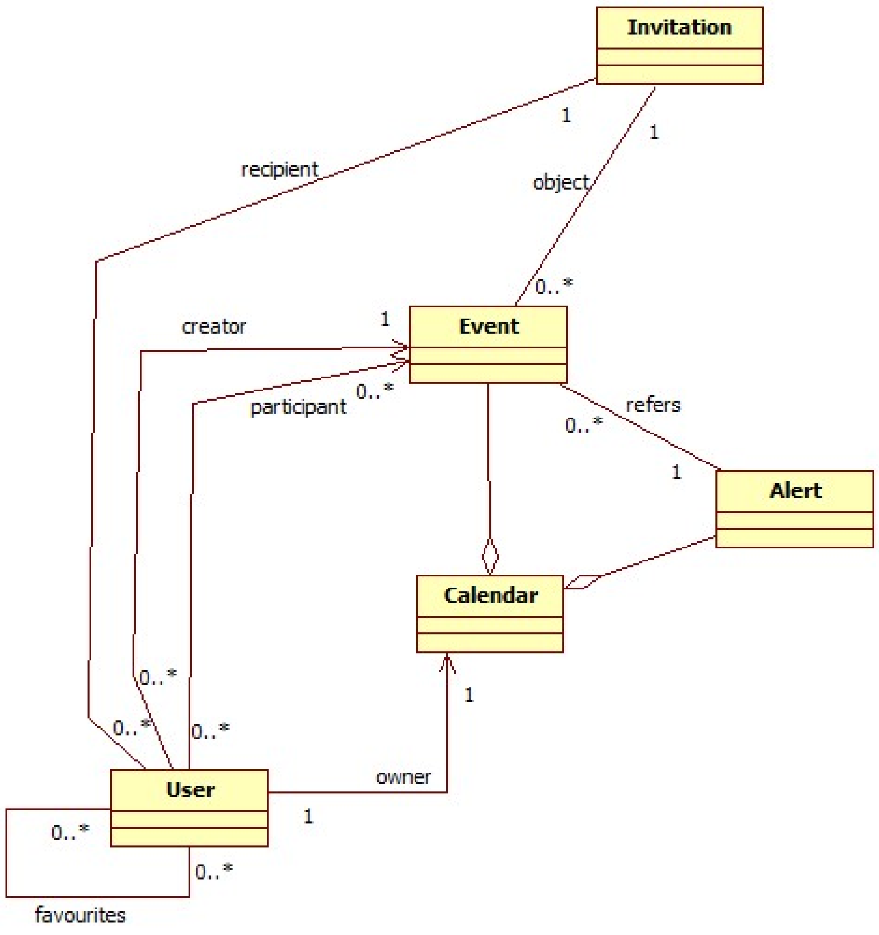


Figure 3.2: Class diagram for MeteoCal

3.2.2 UML Class Diagram

The Figure 3.2 describes a first proposal of entities and their relation.

The UML class diagram shown in Figure 3.2 has some cycles mainly caused by the concept of Calendar. These cycles are wanted because in this first project phase the Calendar entity is created only to collect all the events and alerts of a certain user to provide a faster access to them and an easier implementation.

3.2.3 Registration

Actors	Guest
Preconditions	The guest had never before registered into the system
Execution Flow	1. The guest accesses at the registration page 2. The system requires to insert name, surname, nickname, email and password 3. The system verifies the unicity of the email address 4. The system confirms the registration 5. The guest clicks on the confirmation link 6. The system notifies the registration and sends the user to the profile management page 7. The system generates an empty calendar for the new user
Postconditions	The guest is now a user, he is enrolled in the system and he is logged in
Exceptions	The email is not unique; the guest doesn't complete the activation clicking on the link

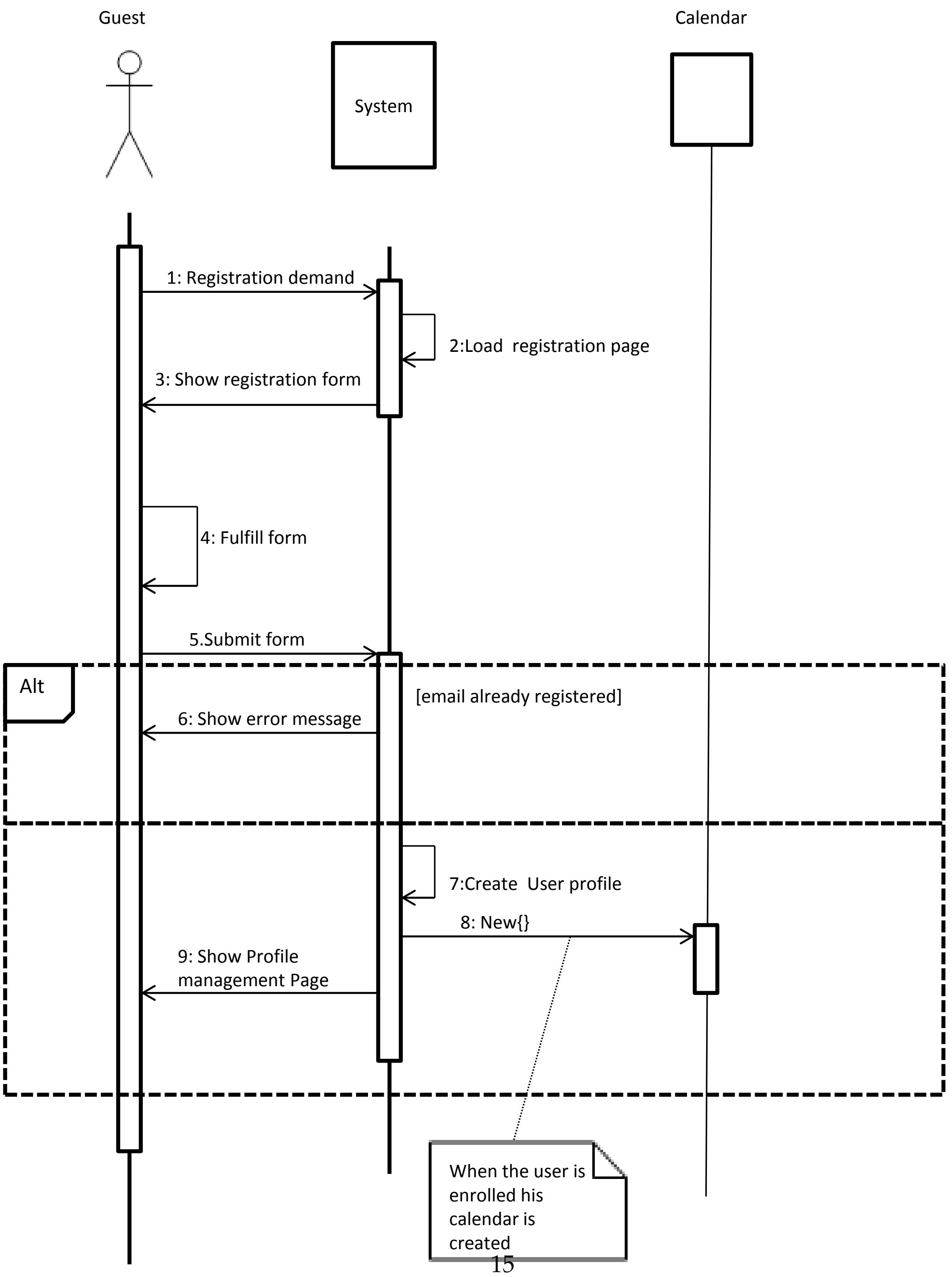


Figure 3.3: Sequence diagram for the registration

3.2.4 Login

Actors	Guest
Preconditions	The guest has already a working account (has successfully completed the registration)
Execution Flow	1. The guest arrives to the web-site and clicks on the "login" button 2. He inserts the combination of e-mail and password 3. The system checks the couple 4. The guest is redirect to his calendar homepage and becomes an user 5. The system controls the WFI for all the scheduled events and updated them 6. The system checks and updates Alerts and Incoming invitations
Postconditions	The guest is now a user, he is logged in and capable to use all the functionality of the system
Exceptions	The couple e-mail password is incorrect: the user cannot log in

Login

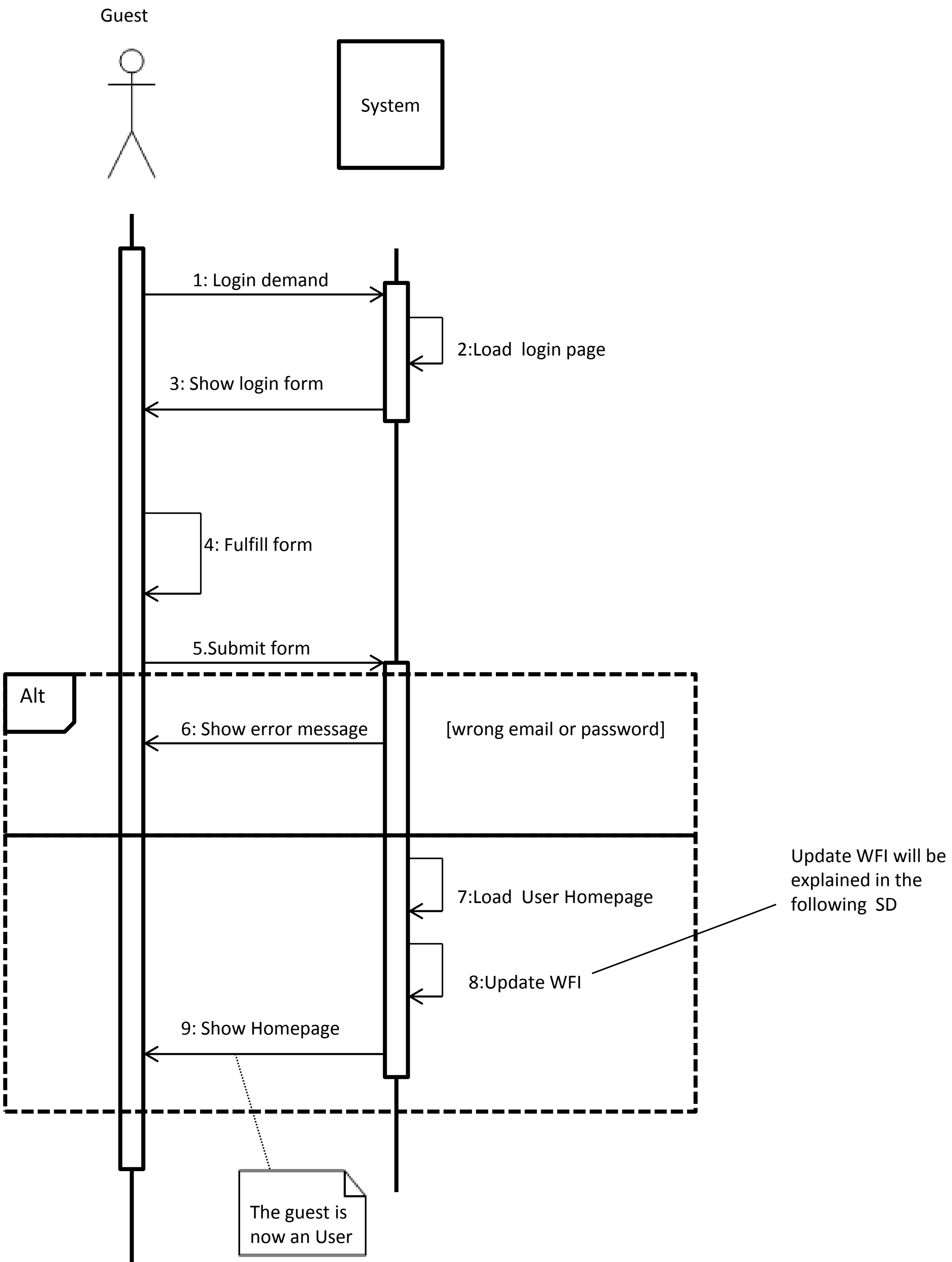


Figure 3.4: Sequence diagram for the login

3.2.5 Profile Management

Actors	Users
Preconditions	The user is logged into the system
Execution Flow	1. The user via his calendar page chooses to the change profile option 2. The system shows at the users the modification form 3. The user modifies the chosen fields 4. The user submit the modifications 5. The system update the user's info
Postconditions	The user's profile is modified
Exceptions	The user doesn't submit the modifications; the new modifications are invalid. In both cases the new modifications are not applied

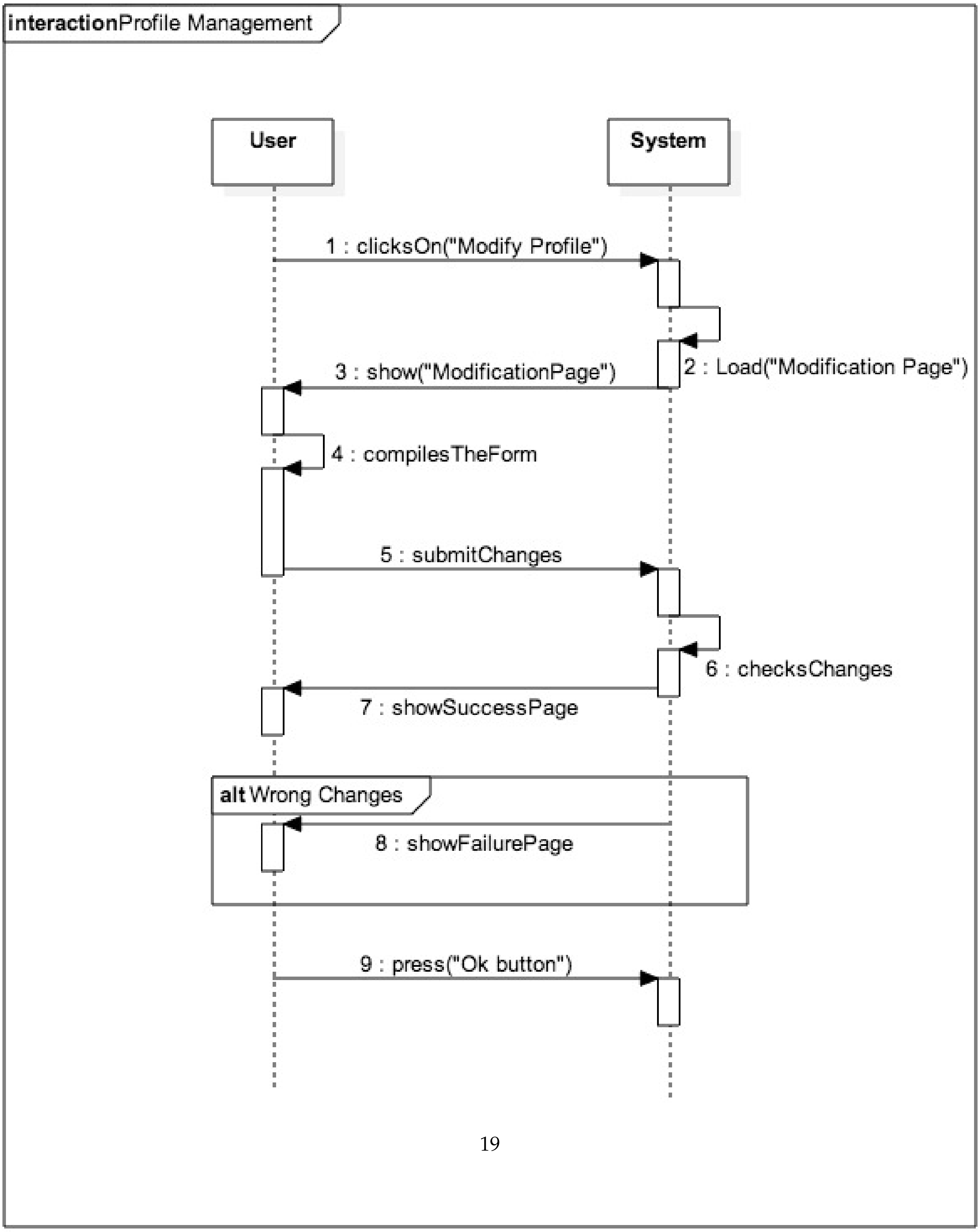


Figure 3.5: Sequence diagram for the profile management

3.2.6 Create Event

Actors	User
Preconditions	The user is logged into the system
Execution Flow	1. The user select a day from his calendar 2. From the just appeared menu he clicks on "Create Event" 3. He inserts the informations about the event 4. He indicates if it's an outdoor or an indoor event [default:indoor] 5. He indicates if the event is public of private [default:private] 6. The user submits his event 7. The system controls the WFI for the new event and updated them (if available) 8. An initial invitation phase starts (for detail on Invitation see Invitation Sending paragraph)
Postconditions	The user now has a new event scheduled in his calendar and he is the organizer of that event
Exceptions	Event's hour or location are missing: the event is not created. The user doesn't confirm the event creation: the event is not created

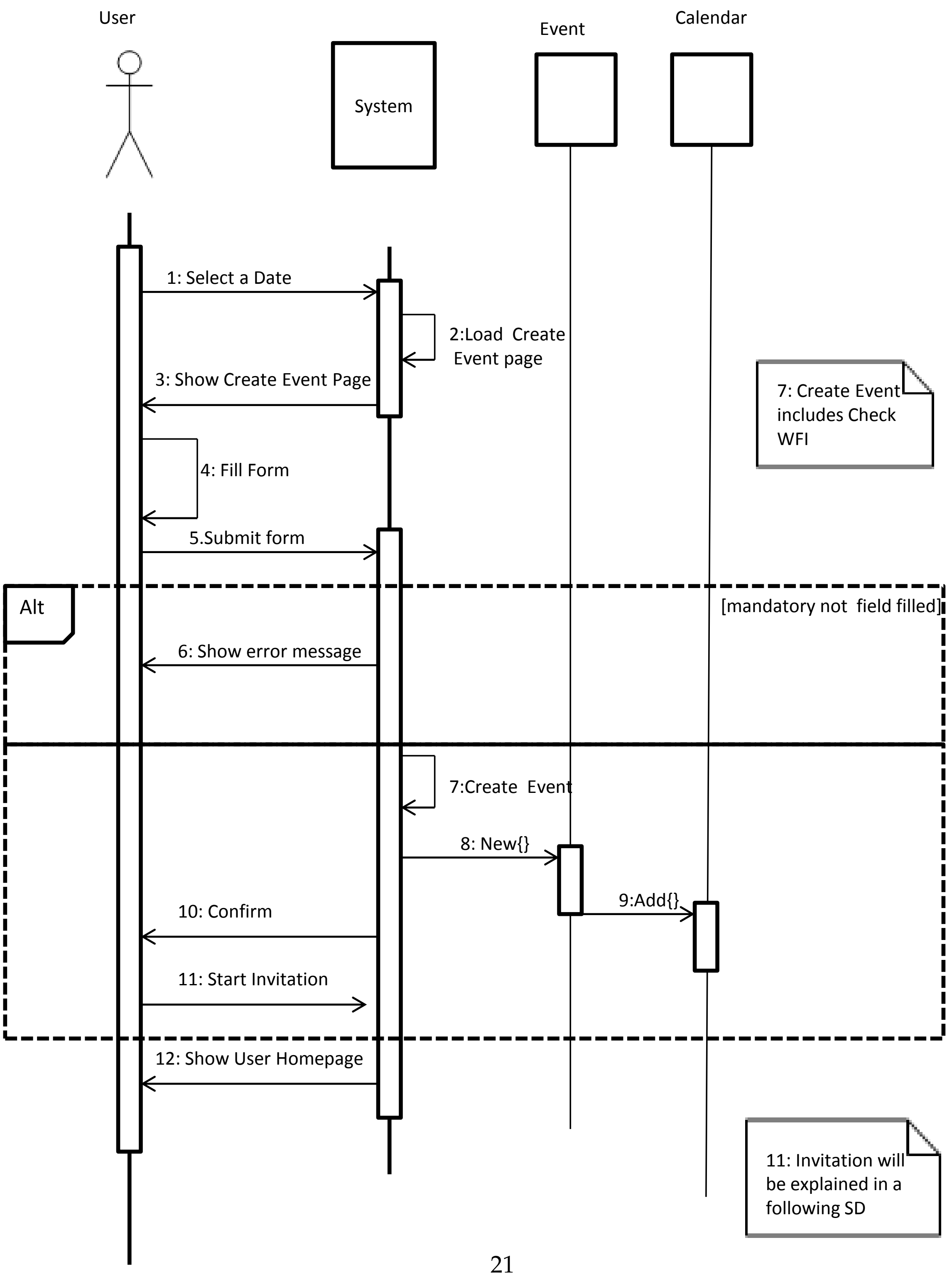


Figure 3.6: Sequence diagram for the event creation

3.2.7 Modify Event

Actors	One organizer A
Preconditions	A is an organizer of the even E and is logged in
Execution Flow	1. A clicks on E 2. The system shows the event and the management interface 3. A clicks on the modify button 4. The system shows the modification interface 5. A modifies the fields 6. A click on the submit button 7. The system update the calendar infos 8. The system generate alerts notifications for the users that participate at E 9. The system redirect A to the calendar page
Postconditions	The E's infos are updated and alerts notifications are generated
Exceptions	The user doesn't submit the modifications; the new modifications are invalid. In both cases the new modifications are not applied

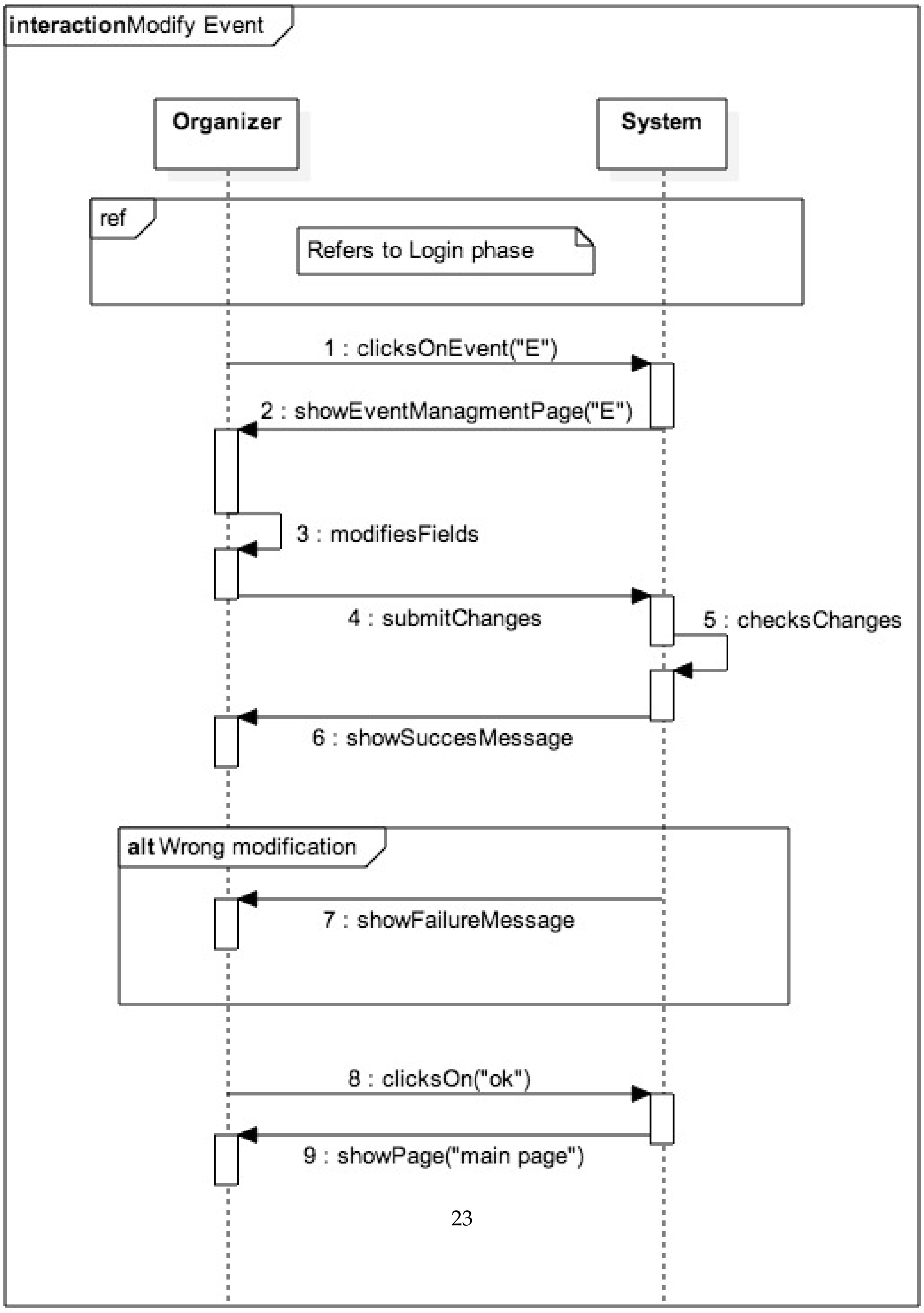


Figure 3.7: Sequence diagram for the event modification

3.2.8 Invitation Sending

Actors	One organizer A and an user B
Preconditions	A is the organazier of the even C. A has never before invited B to his event C. A is logged in
Execution Flow	1. A clicks on C 2. The system shows the event and the management interface 3. A clicks on invite link 4. The system shows the invitation interface 5. A insert B nickname or email in the user search engine 6. The system show the users that correspond with the inserted parameters 7. A finds B and clicks on him, he can also clicks on other users 8. A clicks on the invite button 9. The system generates an invitation for the selected users
Postconditions	Users B and the eventually selected users receive an invitation notification
Exceptions	The user A doesn't identify the user B that is looking for

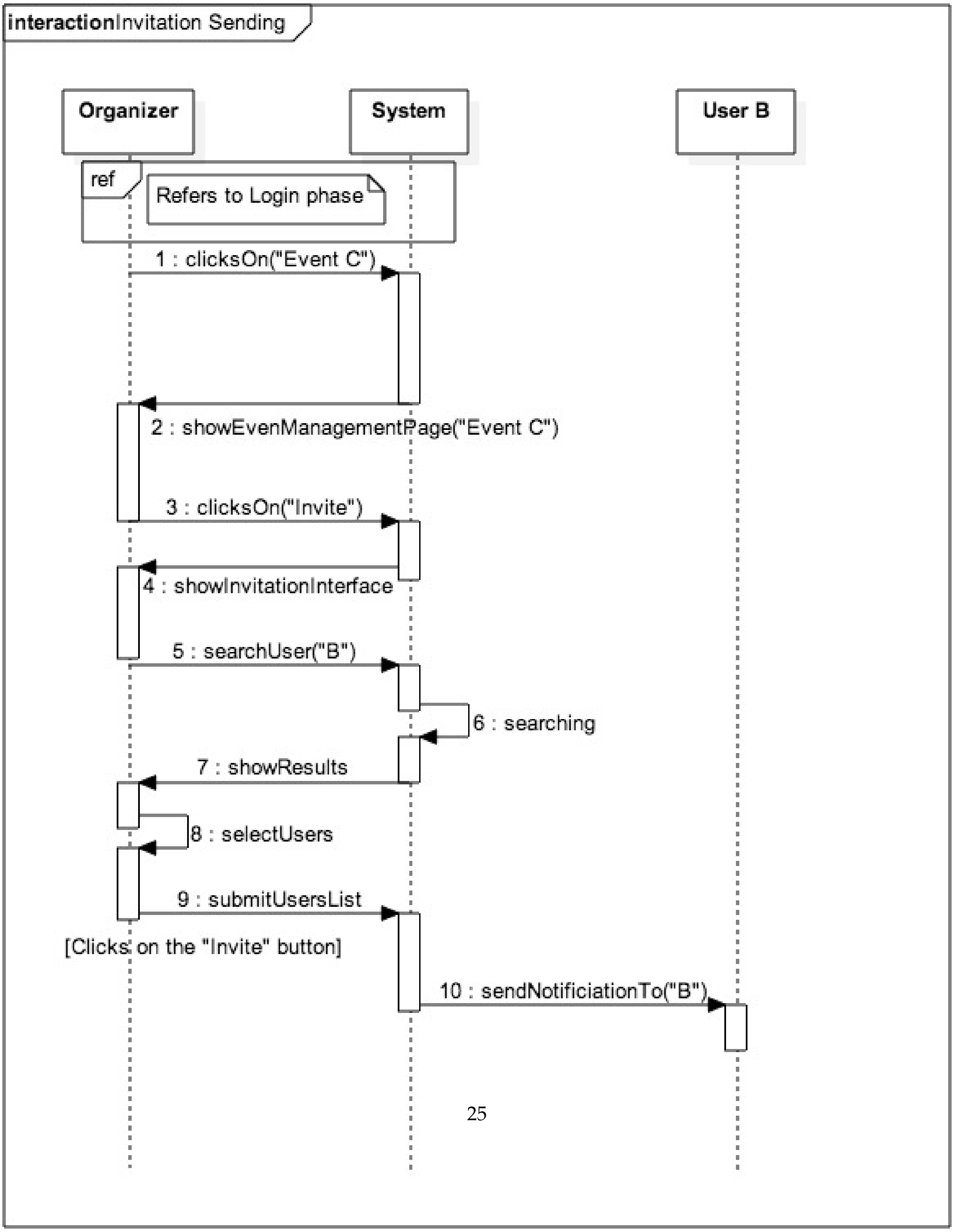
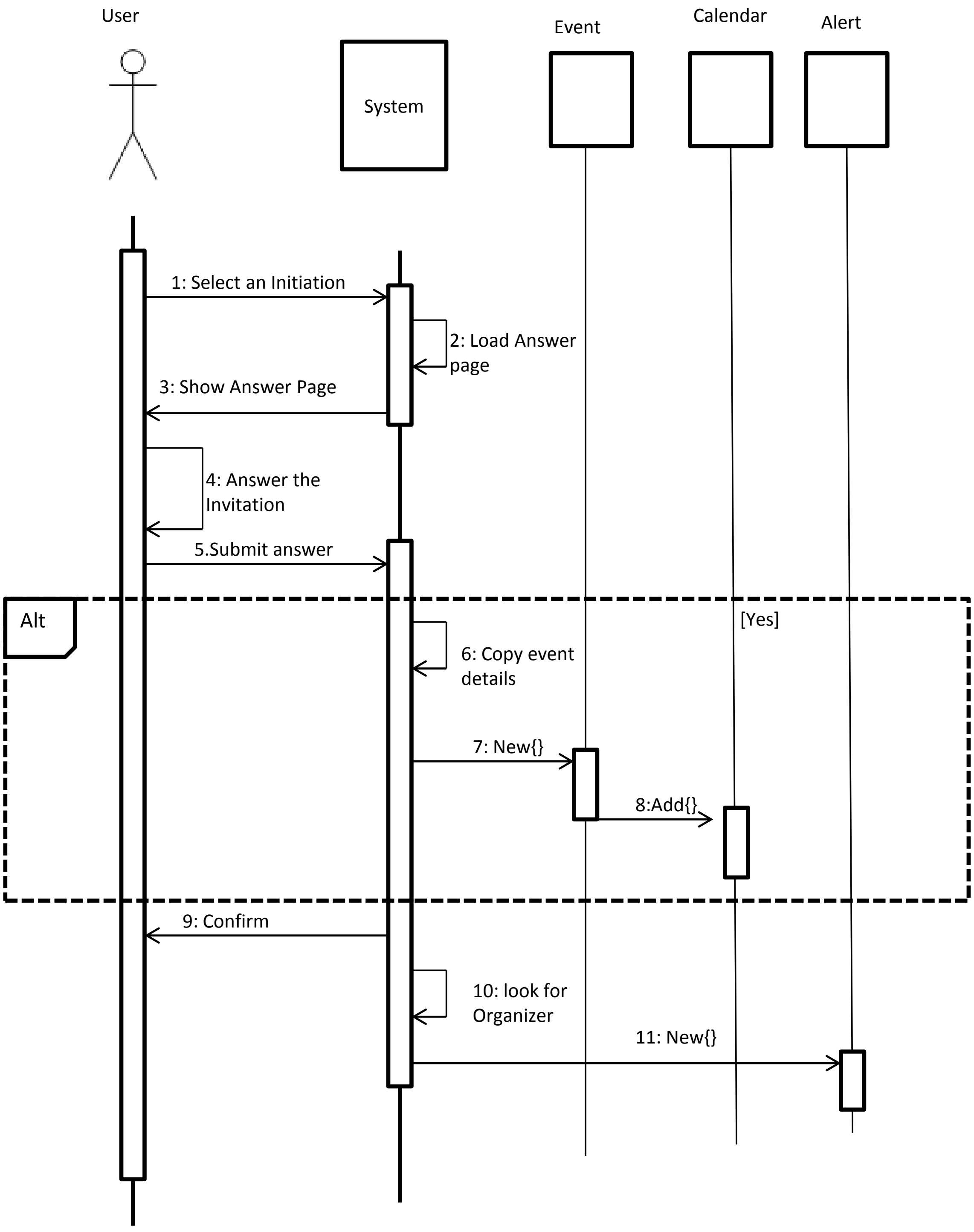


Figure 3.8: Sequence diagram for the invitation sending

3.2.9 Invitation Acceptance

Actors	One user A
Preconditions	A is already logged in and has received an invitation to an event E
Execution Flow	1. The user selects "manage invitation" from his calendar 2. The user selects an incoming invitation 3. A selects the yes or no check-box and can add a comment 4. A submits his answer 5. The system creates and sends a feedback alert to the event's organizer 6. The system adds E to A's calendar 7. The system controls in the A's calendar the WFI for the new event and updated them (if available) 8. Eventual Alerts are shown
Postconditions	A now has E in his calendar
Exceptions	A doesn't submit the answer: the event is not added and no feedback is sent to organizer



NOTE: Event and Calendar refers to the User, Alert to the organizer

Figure 3.9: Sequence diagram for the invitation acceptance

3.2.10 Alert management

Actors	One user A
Preconditions	A is already logged in
Execution Flow	1. A clicks on the "Alert Notification" icon 2. The system shows the list of the alert (the unread ones are highlighted) 3. A clicks on the "Details" button of an alert 4. The system show the details about the selected alert (if the alert was previously highlighted, then it becomes normal) 5. A clicks on the "Delete" button of an alert 6. The system delete the alert
Postconditions	If the alert was previously highlighted ant it is selected, then it becomes normal. If the alert is deleted then it is no longer in A's calendar
Exceptions	A has no alerts: no action possible

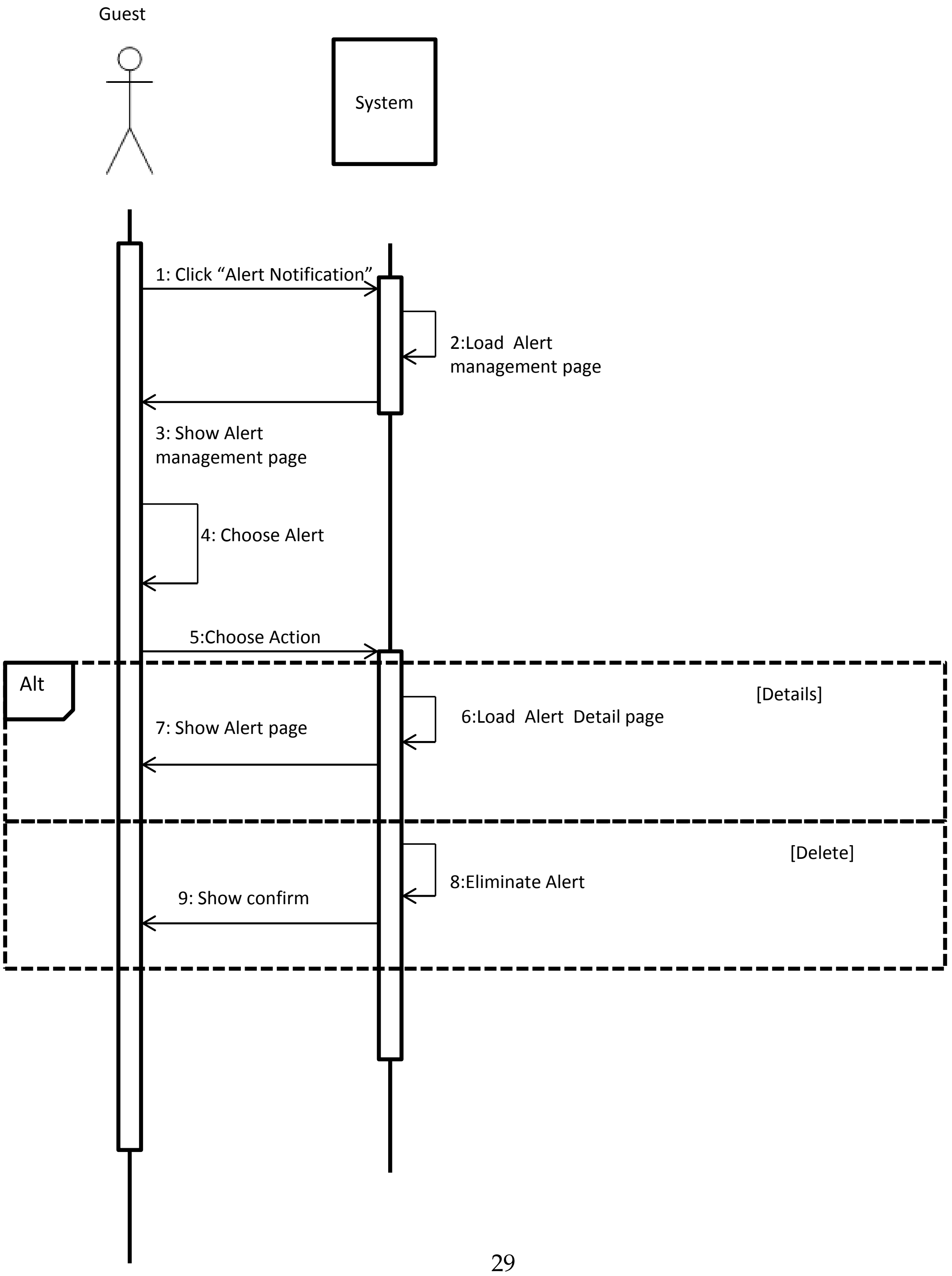


Figure 3.10: Sequence diagram for the alert management

3.2.11 Check Calendar

Actors	One user A
Preconditions	A is logged in
Execution Flow	1. It's the combination between "Alert Management" and "Invitation acceptance" 2. A can navigate in his calendar 3. A can change calendar view
Postconditions	None
Exceptions	None

3.2.12 WFI Control

Actors	WeatherBot
Preconditions	The user A is already logged and has at least an event E scheduled
Execution Flow	1. The system looks for A's event E 2. The system selects E and checks if it's an outdoor event 3. If so the system sends to the WeatherBot the time and the location of E 4. WeatherBot checks WFI for that when and where 5. WeatherBot returns the WFI to the system 6. The system updates the WFI for E in A's calendar 7. Eventual Alerts are created and shown
Postconditions	E's WFI are now up to date
Exceptions	E is not an outdoor event: no update is started. WeatherBot hasn't WFI for that when and where: no update is done

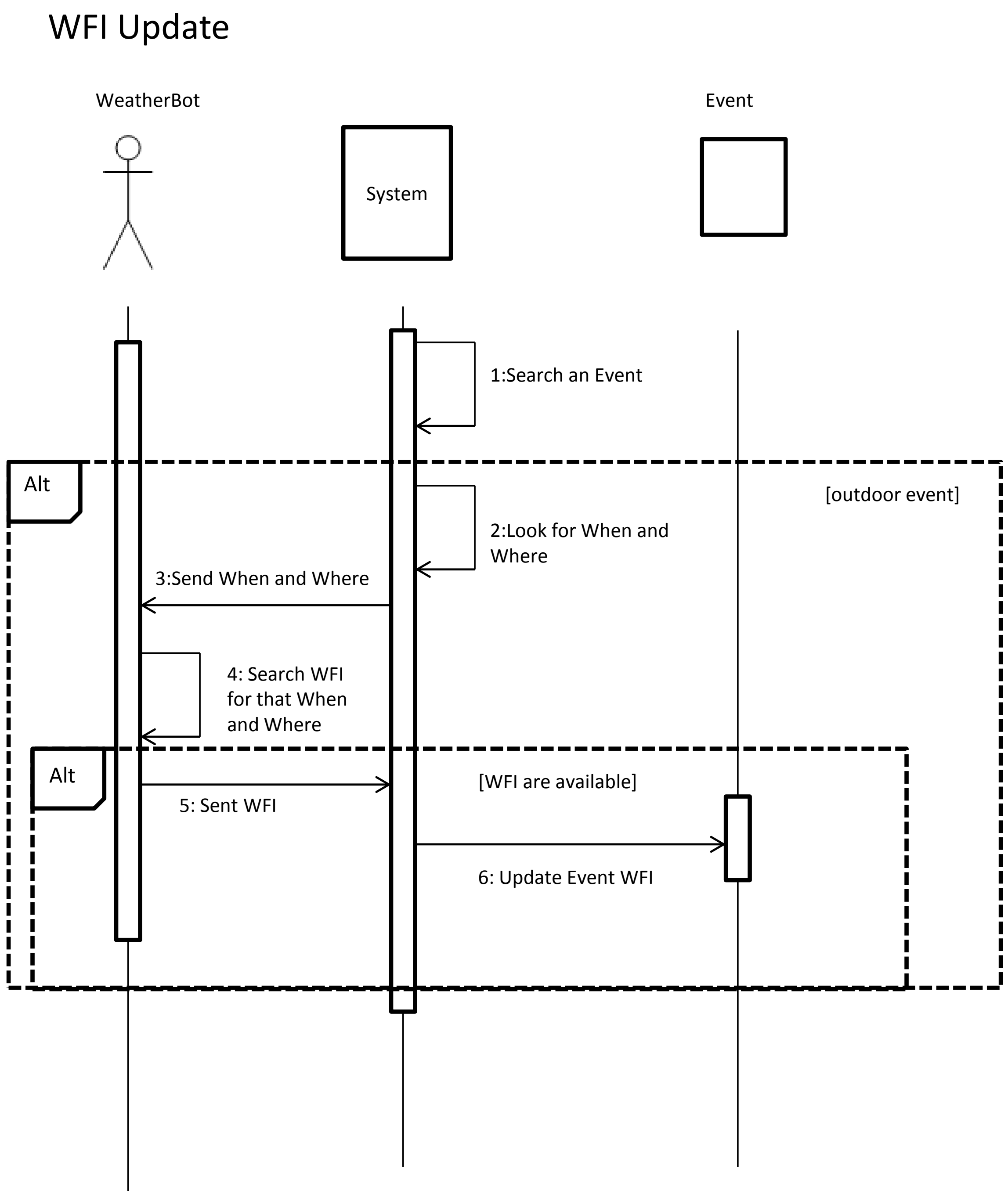


Figure 3.11: Sequence diagram for the WFI control

3.3 Entities Behaviour

In this section the behaviour of some entities presented in Figure 3.2 is exposed using UML state chart diagrams.

3.3.1 Invitation Class

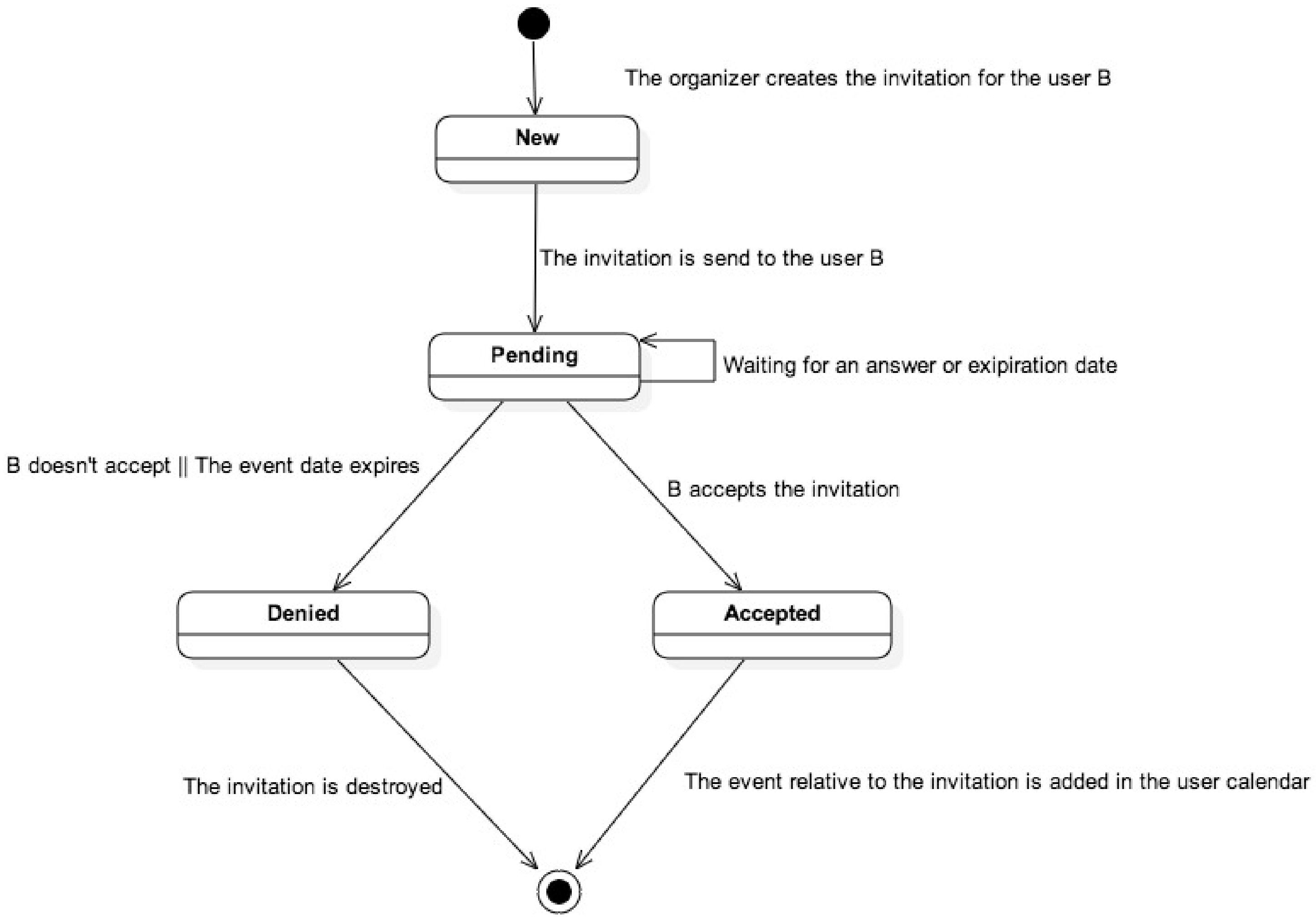

The organizer creates the invitation for the user B
New
The invitation is send to the user B
Pending
Waiting for an answer or exipiration date
B doesn't accept || The event date expires
B accepts the invitation
Denied
Accepted
The invitation is destroyed
The event relative to the invitation is added in the user calendar

Figure 3.12: The behaviour of the Invitation class presented via UML state diagram

3.3.2 Event Class

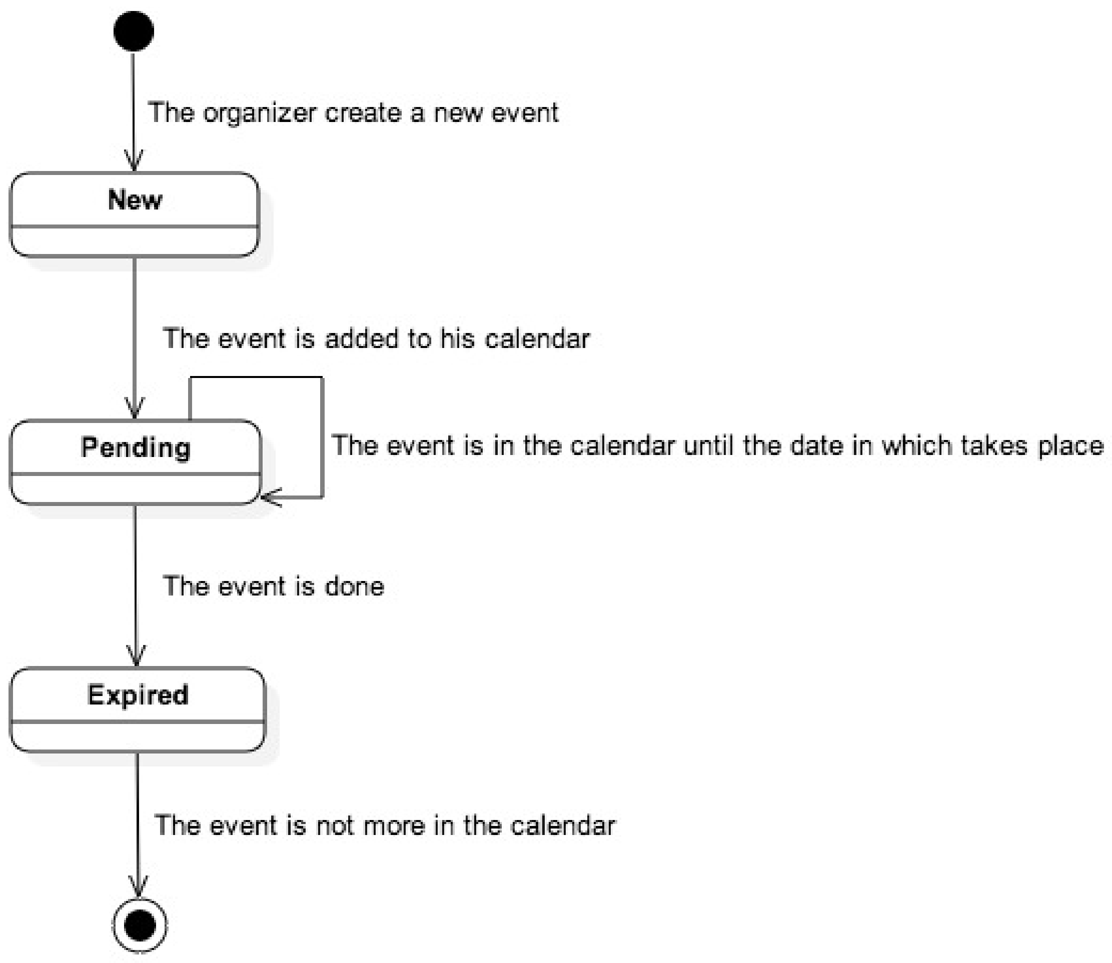

The organizer create a new event
New
The event is added to his calendar
Pending
The event is in the calendar until the date in which takes place
The event is done
Expired
The event is not more in the calendar

Figure 3.13: The behaviour of the Event class presented via UML state diagram

Chapter 4

Alloy Modelling

In this chapter the consistency of the proposed Class Diagram will be tested via Alloy Analyzer. The report that follow is composed by the code used to describe the model and an example of world generated by our predicates.

4.1 Alloy Code

Here the code used is presented.

```
module meteoCal

/*** Class declaration ***/

sig GenericText{}

sig User {
  email: one GenericText,
  favourites: set User
x

sig Event {
  creator: one User,
  partecipants: set User,
  --where: one GenericText,
  --when: one GenericData,
  --wfi: one WeatherForecast
}

sig Invitation {
  recipient: one User,
  object: one Event
  --- the sender can be recovered from the event
}

sig Calendar{
  owner: one User,
  events: set Event,
    alerts: set Alert
}

sig Alert{
  reference: one Event,
}

-- there are also other attributes but are not relevants for
    this representation

/*** DEFINITION OF THE CONSTRAINTS ***/

--each email has to be unique
fact emailUnicity{
  no disj u1, u2: User | u1.email = u2.email
}
--each user has only one calendar
fact calendarUnicity{
  no disj c1, c2: Calendar | c1.owner = c2.owner
}
--each user must have a calendar
```

```
fact oneUserOneCalendar {
  no u: User | all c: Calendar| u != c.owner
}
--an user cannot have himself as favourite
fact noSelfFavorite{
  no u: User | u in u.favourites
}
--an event is in the calendar only if the owner is the
    organizer or a partecipant
fact noFloatingEvent{
  all e: Event | all c: Calendar | e in c.events iff c.owner in
      e.partecipants or c.owner = e.creator--u in
      e.partecipants implies e in c.events and c.owner = u
}
--all the events are contaned in calendars
fact noEventOutOfCalendar {
  all e: Event | some c: Calendar | e in c.events
}
--an user cannot have two favourites who are the same user
fact noDoubleFavouries{
  all u1, u2, u3: User | u2 in u1.favourites and u3 in
      u1.favourites implies u2 != u3
}
--a person cannot be invited twice
fact noDoubleInvitation {
  all u: User | no disj i1, i2: Invitation | i1.recipient = u
      and i2.recipient = u and i1.object = i2.object
}
-- only the organizer of an event can invite to that event
fact onlyOrganizerCanInvite {
  all e: Event | no i: Invitation | e.creator = i.recipient and
      i.object = e
}
 --each partecipanr had to be invited previously
fact aPartecipantHadAnInvite {
 all e: Event | all p: e.partecipants | one i: Invitation | p
     = i.recipient and i.object = e
}
--all the alerts are contaned in calendars
fact noAlertOutOfCalendar {
 all a:Alert | some c: Calendar | a in c.alerts
}
--all the alerts in a calendar are only on the events in the
    calendar
fact alertsOnlyOnMyEvents{
 all c:Calendar | all a:c.alerts | a.reference in c.events
}
-- each organizer has in his calednar the events that he
    creates
fact organizerEvent {
 all e: Event | one c: Calendar | c.owner = e.creator and e in
     c.events
}




/*** FUNCTION ***/

-- invitations returns the set of invitations of which the
    event passed as param is the object
fun invitation [e: Event]: set Invitation {
 {i:Invitation | e = i.object}
}
-- organizedEvents returns the set of events whose creator is
    the same as the param
fun organizedEvents[u: User] : set Event {
 {e: Event | e.creator = u}
}
--partecipantEvents returns the set of events in which the
    user passed as param is partecipant
```

```
fun partecipantEvents[u: User] : set Event {
  {e: Event | u in e.partecipants}
}


/*** ASSERTION ***/
--nobody can invite himself
assert noSelfInvitation{
  no i: Invitation | i.recipient = i.object.creator
}
check noSelfInvitation for 3
--nobody can partecpiate an event without a previous invitation
assert noSneakInEvent {
  all e: Event | #e.partecipants <= #invitation[e]
}
check noSneakInEvent for 3
--all the events in the calendare are there because the owner
    of the calendar is either
-- the organizer or a partecipant
assert noGhostEvent {
  all c: Calendar | #c.events = #organizedEvents[c.owner] +
      #partecipantEvents[c.owner]
}
check noGhostEvent for 3

/*** PREDICATES ***/
--is a way to ensure a number of significant entities
pred showEvent {
  #User > 1 and #Event > 2
}
run showEvent for 4

pred show {


}
run show {} for 3
```

4.2 Alloy Response

Here the alloy response to the model is shown.

```
5 commands were executed. The results are:
   #1: No counterexample found. noSelfInvitation may be valid.
   #2: No counterexample found. noSneakInEvent may be valid.
   #3: No counterexample found. noGhostEvent may be valid.
   #4: Instance found. showEvent is consistent.
   #5: Instance found. show is consistent.
```

4.2.1 Alloy World

Here some examples generated by the description of the model is displayed.

World 1

In Figure 4.1 is shown the world generated by Alloy Analizer via the execution of the predicate "showInvite". In this predicate the number of users and events is respectively constrained to be greater than 1 and 2. in this way the Analizer generates a significant populated world. It could be assume that the world generated under the imposed constraints is consistent.

Word 2

In Figure 4.2 is shown the world generated by Alloy Analizer via the execution of the predicate "show". In this predicate there are no constraints. The predicate generates a world that respects the given constraints where each entities has at maximum 3 instances.

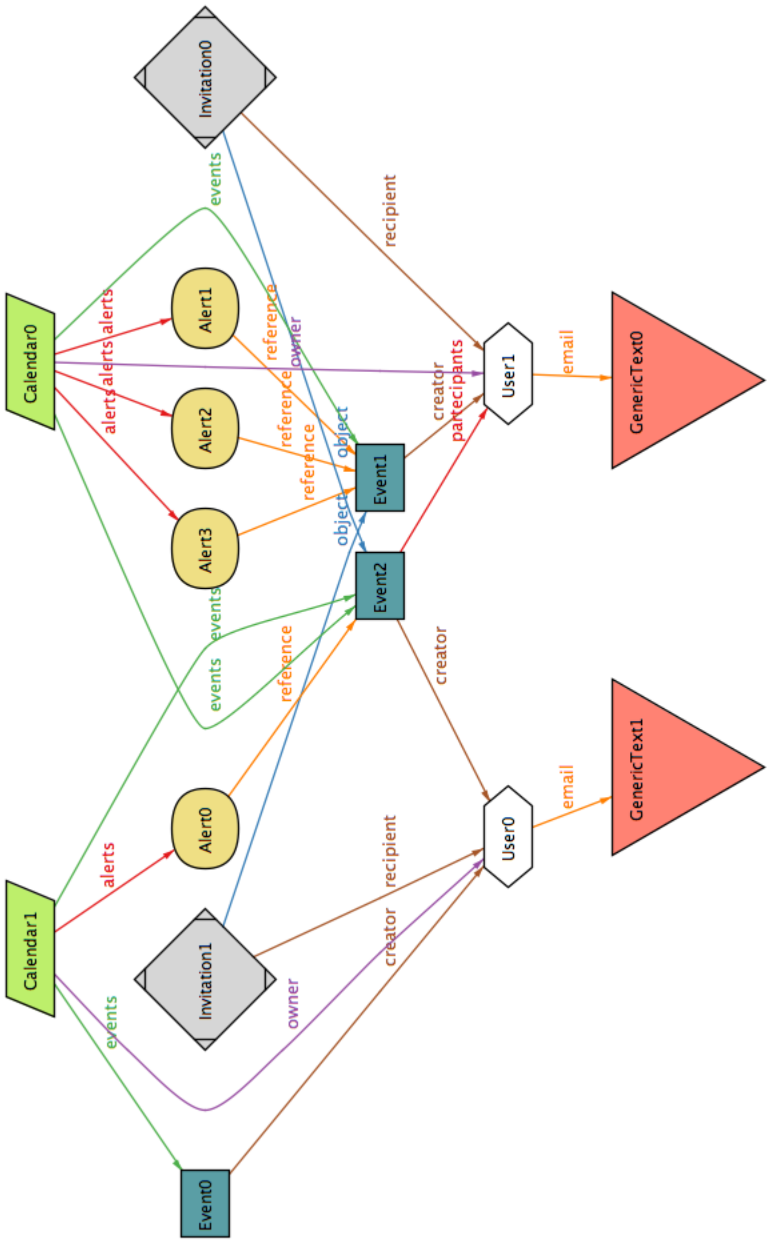


Figure 4.1: World created from the predicate showInvite

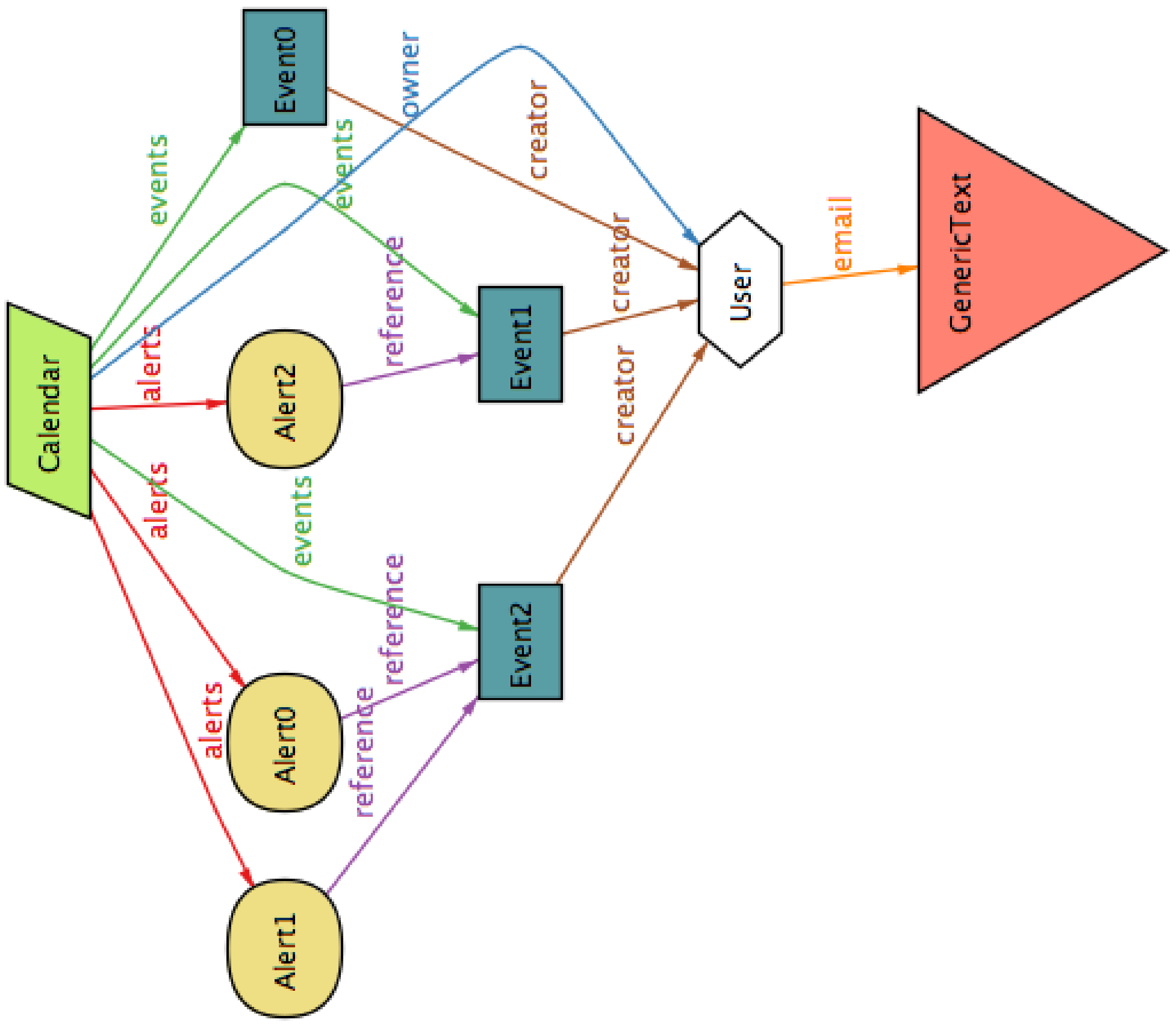


Figure 4.2: World created from the predicate show

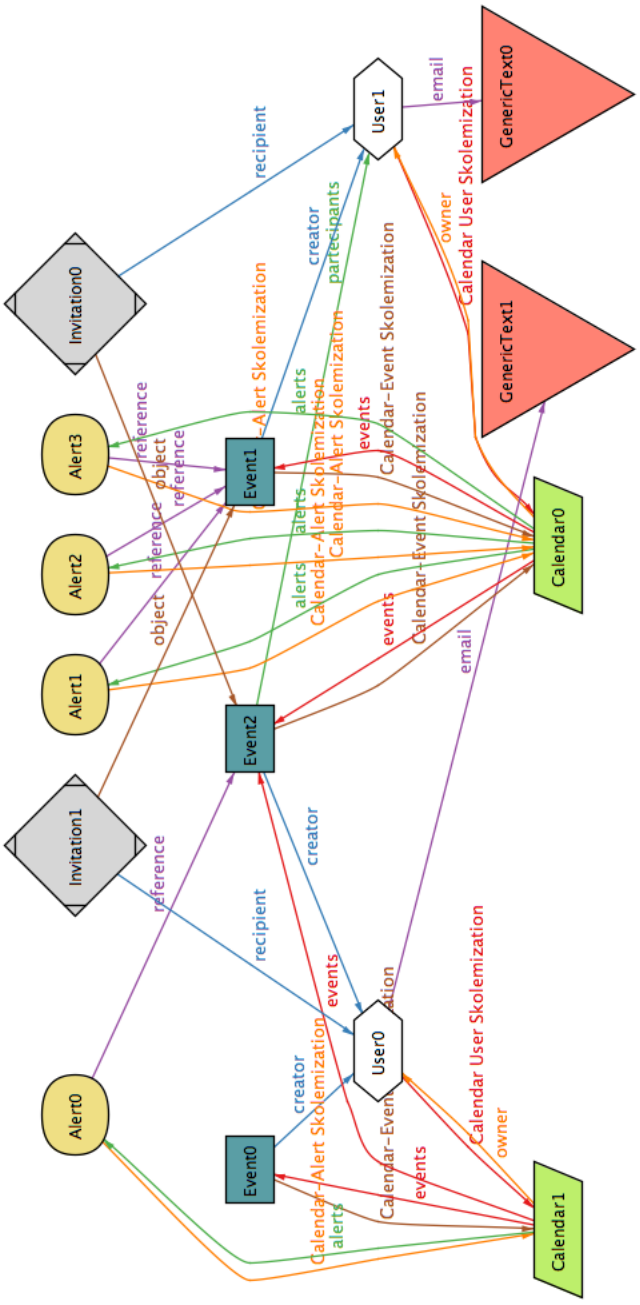


Figure 4.3: World created from the predicate showInvite with skolemization option enabled

World 1 bis

in Figure 4.3 is shown the same world presented in 4.2.1, but in this case the skolemization option is enabled.

4.3 Alloy Result Evaluation

Observing the generated world in Figure 4.3, the existence of some skolemization functions, that are disabled in the model presented in Figure 4.1, can be extrapolated. One first conclusion could be simply done from the Alloy code presented. It's likely that some "fact" over the calendar behaviour has inside unnecessary existential quantifiers over the variable "c" (that refers to Calendar). But the need to introduce this quantifiers is directly consequence of the model representation that is been chosen to adopt. So it's safe to say that the skolem function are also consequences of the cycles introduced in paragraph concerning the UML Class Diagram. This proves the hypothesis made in the chapter about the data redundancy, showing that the same relation can be obtained without the use of the class Calendar. In spite of that is still useful to regroup connected data and to a simpler visual comprehension, at least in this first part of the project. During the following project phases the elimination of these redundancies could be taken into account.

Chapter 5

Other info

This chapter contains information about the used tools and the hours of work by the members of the working group.

5.1 Working hours

Date	Arnaboldi's hours	Caielli's hours
2014/10/30	2h	2h
2014/11/06	5.30h	5.30h
2014/11/07	3.30h	3.30h
2014/11/8	3.30h	3.30h
2014/11/9	—-	2h
2014/11/10	2h	—-
2014/11/12	3h	4h
2014/11/13	4h	4.30h
2014/11/14	3.30h	—-
Total	27h	25h

5.2 Tools

In this first requirements study phase the following tools were used:

- LaTeX and TeXMaker editor
- Alloy Analizer
- starUML