MyMeteoCal: RASD 2.0

Requirements Analysis and Specification Document

Authors:
Benedetto Vitale
Ettore Randazzo
Giacomo Scolari
Professor:
Raffaella Mirandola
Course:
Software Engineering 2

January 24, 2015

Contents

1 Introduction

1.1 Description of the given problem

We want to develop a web-based application named MyMeteoCal, which wants to offer a new weather based online calendar for helping people scheduling their personal events avoiding bad weather conditions in case of outdoor activities. 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 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 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 the user 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) good-weather day (if any). In addition, a registered user should be notified by email (both for invitation and cloudy outdoor events alerts). We desire our app to provide the possibility to import and export a user's personal calendar. The system will also have to manage time consistency when creating an event by avoiding conflicts with existing events. The information associated to the weather of events should be updated periodically and then the system will have to notify outdoor event participants in case the forecast has changed.

1.2 Goals

We desire our app to provide the following features:

- A guest can register to the system.
- A guest, once registered, can log into the system.
- Users can see their own calendar whenever they want.
- Users can make their calendar public, thus allowing other users to see it.
- Users can create events related to their calendar.
- An event can be either public or private. If it is public it can be seen by any user.

- The creator of an event can invite other users to that event.
- Invited users are notified both by email and push notification and can accept or decline the invitation.
- The creator of an event can edit or delete it at will.
- An event contains weather forecast information (if available) regarding the place where it is scheduled.
- Users have the possibility to export/import their calendar.
- Whenever a bad weather condition regarding a certain outdoor event is forecast, users are notified and the closest good weather day is proposed.

1.3 Domain properties and assumptions

We suppose that these conditions hold in the analyzed world:

- Only people use MyMeteoCal.
- People cannot be in two distinct places at the same time.
- We assume that the weather forecasts are correct.
- An event occurs only in a single place.

1.4 Proposed system

We propose a web platform, allowing users to check their own calendar as well as any public calendar created by any other user.
Users will be able to create their own events, inviting any number of users, making it either public or private and selecting what they personally consider a bad weather situation for a given event. Creators will then be able to delete or modify a given event. Users will be able to import and export their calendar, and will be notified either by the web app and by email for other user's invitations or bad weather flags.

1.5 Identifying stakeholders

Our one and only stakeholder is our Software Engineering 2 professor. She wants us to work over an entire software development process step by step with the relative documentation,in order to make even an external person to the project understand its complete functioning; that's also why we have several deadlines which we aspire to respect.

1.6 Other considerations about the system

In addition to the given goals, we want MyMeteoCal to be:

- User-friendly, with an intuitive GUI, so that an average user is able to enjoy all the functionalities in the simplest possible way.
- As stable as possible: it has to grant some basic functions in a stable way. For example there has not to be any malfunctioning in creating a new event.

1.7 Glossary

These are the definitions of some words used in this document:

- Guest: A person who either is not registered or hasn't signed in yet.
- User: A person logged into the system.
- Event: Something happening in a specific place in a determined time span.
- Creator: A user who creates an event.
- Bad Weather: weather conditions considered not suitable for the specific event, so i.e. a bad weather flag concerning picnicking could not be so for a skiing activity.
- Good Weather: weather conditions considered suitable for a specific event;
- Calendar: A graphic representation of many years, displayed in months containing days, eventually containing events.

2 Actors Identifying

These are the actors concerning our system:

- Guest(already defined in the glossary).
- User(already defined in the glossary).

3 Requirements

Assuming that the domain properties, written in paragraph 1.3, hold, from the goals, written in paragraph 1.2, we can derive our requirements.

We write below, according to the goals, which are the necessary requirements:

1. *Registration of a person into the system:*

- The system has to provide a sign up functionality.

2. *Sign in of a person into the system:*

   - The system has to provide a sign in functionality.

3. *Operations related to the calendar:*

   - The system will offer to users the possibility to see their calendar whenever they want.
   - The system will allow users to make their calendar public or private.
   - Users can search through all the users to see a user's public calendar.
   - The system will offer to users the possibility of importing/exporting their calendar.

4. *Operations related to events:*

   - The system will offer to users the possibility of creating events related to their calendar.
   - The system will allow creators to make their events public or private.
   - Users are able to see any public event in any public calendar.
   - The system will offer to creators the possibility to invite other users.
   - The system will send a notification and an email to every invited user regarding a certain event, giving them the possibility to accept or refuse the invitation.
   - The system will have to notify the invited users both by email and push notification, plus, it will offer to invited users the possibility of accepting or declining the invitation.
   - The system will allow creators to delete or modify their events.

5. *Weather forecast information:*

   - The system will have to attach the weather forecast information(if available) related to the place in which events are scheduled as soon as they're created.
   - The system will have to update weather forecast information related to events every 12 hours, and if it is outdoor and it has changed it will have to notify its users of such change
   - The system will have to notify the creator if a bad weather condition is forecast three days before the start, proposing the closest good weather day available by push notifications and email.
   - The system will have to notify all the participants of an outdoor event if a bad weather flag is forecast one day before the start of the event.

3.1 Functional Requirements

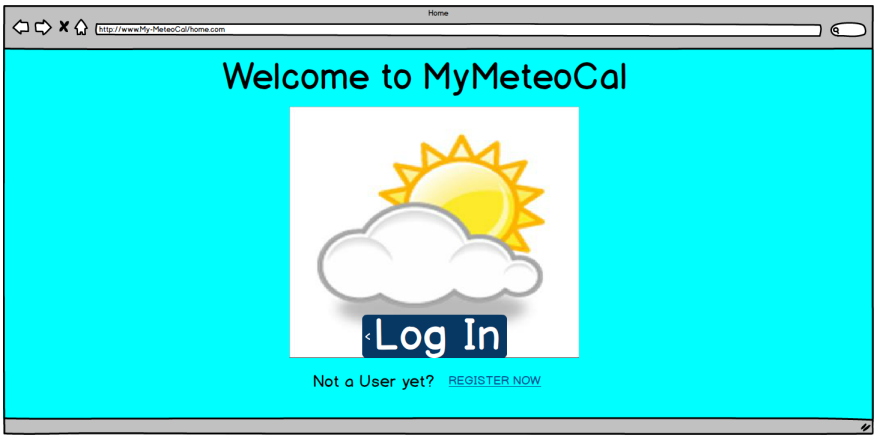
Once defined the main feature that MyMeteoCal has to offer, we then proceed to list the functional requirements related to each defined actor:

- Guest:
  - Sign up.
  - Sign in.
- User:
  - Sign out.
  - Look at their calendar, including its saved events.
  - Modify their profile information, including the possibility to set their calendar private (public by default).
  - Search for other users, and if one is selected, look at their public information (calendar and/or events).
  - Import/export their calendar.
  - Create an event.
  - Modify an event (including deletion).
  - Invite other users to an event created by he/she, sending push notification and emails to the targeted users.
  - See and accept or decline events' invitations.

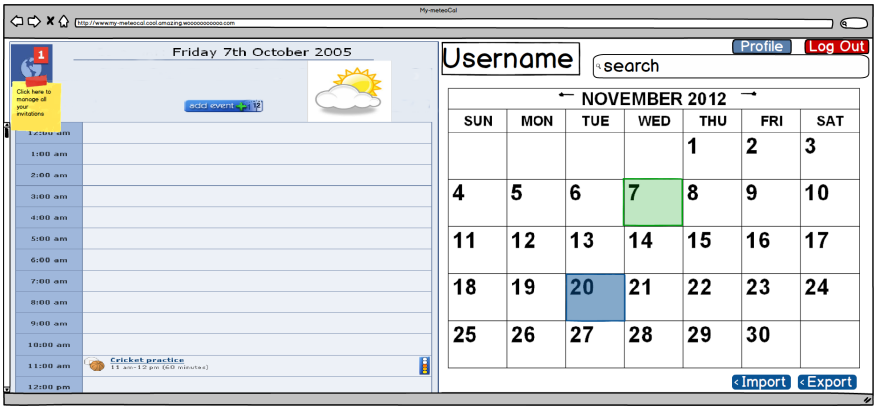
3.2 Non Functional Requirements

3.2.1 User Interface

The basic idea is offering a web-based user interface; the home web page will just offer to a guest two choices, registering or logging into the system, thus accessing the whole functionalities of our application presented before. In order to clarify our idea, we have created a mock-up of the main concept of the user interface that we have in mind:



Once the user is logged into the system, we have decided to create a sort of personal calendar wall for the user, in order to show him which are the days containing scheduled events (which will highlighted on the calendar), plus, he has the possibility to select a specific day by clicking on it. This operation will also show more informations about the selected day (which will be highlighted on the calendar in a different way from the one used for the days with scheduled events), as the scheduled events,their time span and the relative weather forecast. The wall also offer buttons that allow the user to manage his/her invitations in an easy way, export/import his/her calendar, logout, add an event and edit its personal profile. This is the main idea:



In particular, the button add event will lead to another page characterized by a format in which users will insert all the details related to the event, as the place in which is going to take place, other users that they want to invite, the kind of event, what they personally consider bad weather among many possible choices through a check-box inside the page and all the other possibilities stated in this document about the creation of events.

3.2.2 Documentation

During the development of our project, we will provide the following documents in order to organize our work step by step and in a clear way:

- *RASD*: Requirement Analysis and Specification Document,in order to understand the given problem and to analyse in a detailed way which are our goals and how to reach them defining requirements and specification in the clearest possible way.
- *DD*: Design Document, to define the real structure of our web application and its tiers.
- *JavaDoc comments in the source code*: to make anyone that wants to adapt the platform or do maintenance on it understand our code.
- *User Manual*: a guide to use MyMeteoCal.
- *Testing Document*: a report of our testing experience as users of another MyMeteoCal project.

3.2.3 Architectural Considerations

We will use J2EE platform with a database, in order to keep track of users and all the system related information.

This will also be remarked by the class diagram following in this document, that will represent the basic skeleton for our database, that will be explained in a more specific way through an ER Diagram in the DD.

In order to use MyMeteoCal an updated web browser and an internet connection are needed.

4 Scenarios Identifying

Here are some possible scenarios of MyMeteoCal:

- Jimmy is a businessman who needs to schedule his meetings in an efficient, compact and accessible way. He listens to a rumour on the streets of Milan, talking about this new web app called MyMeteoCal which appears to be exactly what he is looking for. He runs home and googles "MyMeteoCal", he enters to the site, quickly and recklessly registering and then signing in. He successively goes to his profile page and makes his calendar "private", because he cannot allow other people to see his meetings, and starts creating events, not inviting anyone, as he only needs a calendar to remember his own events.
- Lord Carlos is a PR for the famous "Drink&GetWasted" party-organizing organization, and he needs to create a new event for Saturday night. He logs in to MyMeteoCal and clicks on the next Saturday, clicks on create

event and inserts the info needed. As it is an indoor event, he doesn't need to specify any weather preference, so he just sets the hours of the events and he starts inviting as many users as he knows.

- Timmy is a mountain climber who wants to schedule an excursion to the Alps with his friends, so he creates a new event called "Fall from Mount Blanc", selecting its location, selecting the date and the time, inviting all his friends. He fears snowstorms so he wouldn't like to go there if it is forecast to snow, then he marks this event as "outdoor", and sets snow as bad weather.

- Timmy receives an email from MyMeteoCal, it warns him that a bad weather condition is forecast for the day he wanted to go climbing and that the next available day with a good weather is forecast to be two days after the selected date. He then sadly decides to modify his event changing its date.

- Lord Pectore doesn't remember what he has to do, so he logs in and checks his own calendar, he finds out to have a lot of free time, so, as he is secretly in love with another user, he looks for "ladyOscar", he selects the only result found and he checks her calendar, which happens to be public. He finds out she is having lunch at a restaurant in Manhattan on that day, so he quickly turns off his computer and takes the first flight available to Manhattan to see her (even though he's not been invited to, as we can see also bad behaviours can happen in this system).

- Lelouch Vi Britannia is a world-class very well known Nobleman, who has a lot of fans. He logs to the system and looks that there are some notifications for him. He opens the pop-up and reads all the events he's been invited to. He accepts every event he can, as he loves to be hailed, but unfortunately two of them coincide and the system tells him he can't accept the latter. So he decides to decline the last invitation.

- Cassandra is a Greek warrior not really accustomed to technology. She used to have all her scheduling in a stone table, but now she decides to use MyMeteoCal. She is too lazy to manually copy all her scheduling, so she decides to import it (accurately codifying her stone-tables to the format needed to import). She is also very frightened of information-loss, so any time she creates a new event she exports her calendar, to be sure to be able to restore it at any time.

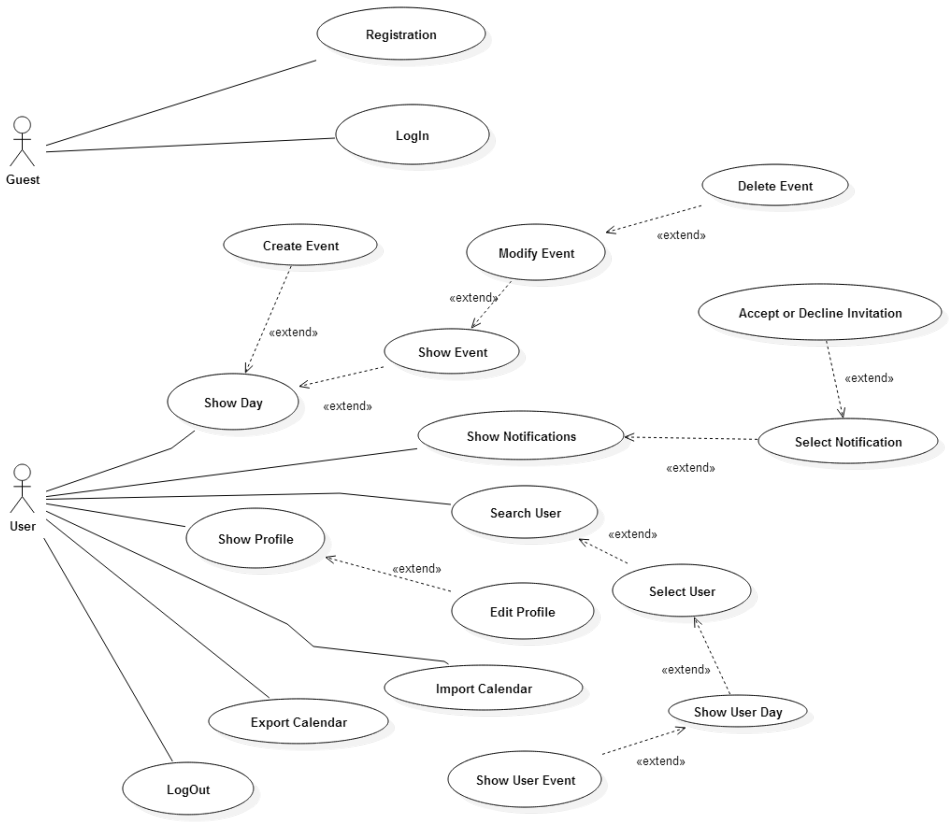
- Dart is a dark lord busy all the time planning to conquer the galaxy. He never checks MyMeteoCal for no reasons, so he systematically checks the mail. An email notifies him that he's been invited to Hutt Castle for a party. He then accepts it looking forward to that event.

5 System specification

We now proceed in providing the necessary information to describe our system in an appropriate way, through specific diagrams and their relative notation:

5.1 Use Case Diagram

From the scenarios described above and from the general analysis in this document,we have detected the possible use cases and represented them through the relative UML notation:



We also provide a specific description of the single use cases using the appropriate template:

UseCaseName	**Register**
Actors	Guest
Preconditions	The Guest is on the home page of the application
Flow of events	1. The Guest clicks on the registration link. 2. The system shows him the registration page. 3. The Guest insert the requested data in the relative input form. 4. The Guest clicks on the button confirm registration. 5. The system shows him/her a successful message.
Postconditions	The guest can log into the system with the declared credentials.
Exceptions	The requested data is not valid, so the registration fails and an error message is shown

UseCaseName	**Show Day**
Actors	User
Preconditions	The user must be on his/her home page
Flow of events	1. The user selects the month. 2. The system shows him the days in that month, highlighted on the calendar if they have an event scheduled. 3. The user selects a day. 4. The system shows him the selected day containing all the scheduled events and highlights it on the calendar as a selected day.
Postconditions	The user can see the selected day and is able to create a new event and see the scheduled events on that day,the day is also highlighted on the calendar as a selected day.
Exceptions	

UseCaseName	**Show Event**
Actors	User
Preconditions	The user is looking at his events on a specific day
Flow of events	1. The user selects an event from the list of events of that day. 2. The system shows him the selected event.
Postconditions	The user can see the information regarding the selected event, and if he is the creator he can modify it, delete it or invite other users.
Exceptions	

UseCaseName	**Create Event**
Actors	User
Preconditions	The User has selected a specific day on its calendar.
Flow of events	1. The User clicks on the button Create Event. 2. The system opens a new screen where the User can insert all the requested information in the relative forms. 3. The User inserts the requested information. 4. The system creates the events, adds it on the User's calendar and sends the invitations to the Users specified during the creation of the event. If it's available, the system also provides the weather forecast for the time in which the event is scheduled.
Postconditions	The new event has been created and it's shown on the User's calendar, the Users specified during the creation have also been invited to it and in case of outdoor event the weather forecast is shown(if available).
Exception	The User has inserted a past date and/or a time span that conflicts with another event, the operation is not completed and an error message is shown.

UseCaseName	**Modify Event**
Actors	User
Preconditions	The user is looking at the selected event and he is the creator of that event
Flow of events	1. The user clicks the "modify" button. 2. The system asks the user to modify some detail of the event. 3. The user modifies what he desires to modify and then clicks "Save Changes". 4. The system overrides the old event with the new info.
Postconditions	The info of the event is updated for the creator, and every other participant loses its participation and is invited again
Exceptions	The user tries to modify the date of the event inserting a date previous to the current Global Time. An error message is shown and the changes are not saved.

UseCaseName	**Delete Event**
Actors	User
Preconditions	The User is on the page to modify a specific selected event of which he/she is the creator.
Flow of events	1. The User clicks on the button Delete Event. 2. The system provides to delete the selected event and no longer lists it on every Users' calendar invited to that event.
Postconditions	The selected event no longer exists in the system and it's not shown on any Users' calendar any-more.
Exception	

UseCaseName	**Sign In**
Actors	Guest
Preconditions	The Guest is on the home page of the application.
Flow of events	1. The Guest clicks on the login button. 2. The system shows him the login page. 3. The Guest insert credentials. 4. The Guest submit the data. 5. The system shows him the home page with his/her calendar.
Postconditions	The Guest is logged into the system, thus becoming an user.
Exceptions	The credentials are not valid, the login fails and an error message is shown.

UseCaseName	**Show Notifications**
Actors	User
Preconditions	The User is in his/her home page.
Flow of events	1. The User clicks on the notifications button. 2. The system shows him all his/her notifications.
Postconditions	Notifications are shown and the User has the possibility of selecting one to obtain more information.
Exception	

UseCaseName	**Select Notification**
Actors	User
Preconditions	The User can see a list of notifications
Flow of events	1. The User selects a specific notification among the list. 2. The system shows him/her more detailed information about the event whose the notification is related.
Postconditions	More detailed information about the event related to the notification are shown by the system.
Exception	

UseCaseName	**Accept or Decline Invitation**
Actors	User
Preconditions	The User has selected a specific invitation among the list of notifications.
Flow of events	1. The User accepts (declines) the invitation by clicking on the relative button on the page. 2. The system adds him to(remove him from)the participants(the invited Users), removes the notification form the list and then updates the information related to the of the event.
Postconditions	The User is added to(removed from) the participants (the invited Users) and the list of Users of the specific event is updated.
Exception	The user has accepted an event that conflicts with another event, the invitation is not accepted and an error message is shown

UseCaseName	**Logout**
Actors	User
Preconditions	The User is in their homepage.
Flow of events	1. The User clicks on the button Logout. 2. The system shows him the Guest home page.
Postconditions	The User is now a Guest and he/she'll have to Login in order to become User again.
Exception	

UseCaseName	**Export Calendar**
Actors	User
Preconditions	The User is in their home page.
Flow of events	1. The User clicks on the button Export Calendar. 2. The system provides the download link for the calendar and the file is downloaded on the Client.
Postconditions	The Calendar file is located on the Client.
Exception	

UseCaseName	**Import Calendar**
Actors	User
Preconditions	The User is in their home page.
Flow of events	1. The User click on the button Import Calendar. 2. The system asks him for the path on the client of the file to be uploaded. 3. The User provides the path and the file is uploaded. 4. The system reads the file and merges the information with the existing online calendar.
Postconditions	The calendar file is uploaded and merged with the existing online Calendar.
Exception	• The uploaded file is of an unsupported format, the operations are stopped and an error message is shown. • There are some merging conflicts, the conflicting events are not added.

UseCaseName	**Show Profile**
Actors	User
Preconditions	The User is in their home page.
Flow of events	1. The User clicks on the button Profile. 2. The system shows him/her a page containing all his/her personal information.
Postconditions	The User can see all his/her personal information and eventually edit them.
Exception	

UseCaseName	**Edit Profile**
Actors	User
Preconditions	The User is on the page containing his/her personal information.
Flow of events	1. The User clicks on the button Edit Profile in the profile page. 2. The User can modify his/her personal informations in the relative forms, as well as set his/her calendar private (public by default). 3. The User submit the data. 4. The system updates the profile's information.
Postconditions	The User profile is updated with the information inserted by the User.
Exception	The inserted data is not valid, so the update fails and an error message is shown.

UseCaseName	**Search User**
Actors	User
Preconditions	The User is in their home page.
Flow of events	1. The User clicks on the search form. 2. The User inserts the username of the User he wants to seek. 3. The user submits the information. 4. The system shows them the results.
Postconditions	The results relative to the sought username are shown by the system.
Exception	A not valid string is sent, an error is shown.

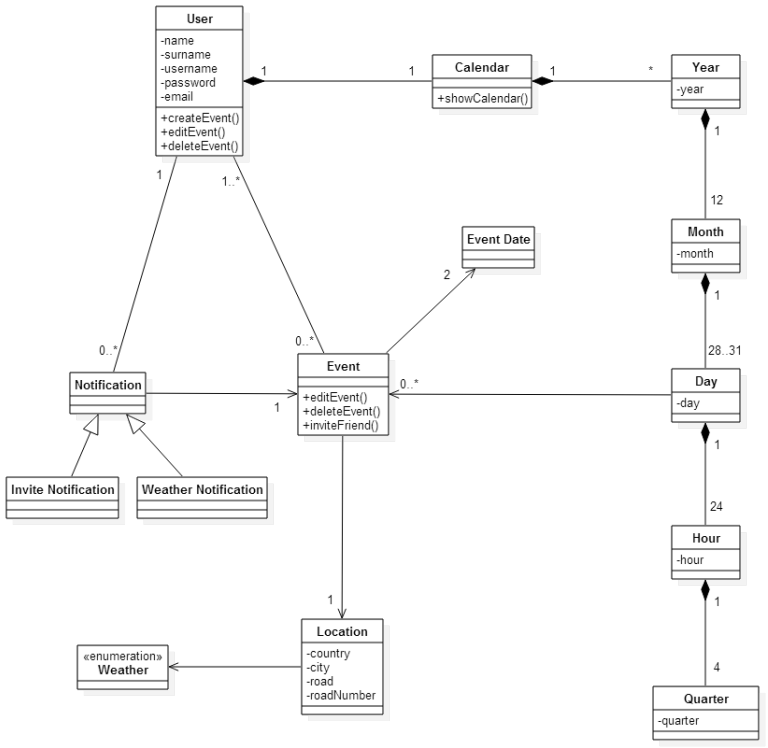
UseCaseName	**Select User**
Actors	User
Preconditions	The User can see a list of other Users.
Flow of events	1. The user clicks on the sought username among the results shown by the system. 2. The system shows them the home page of the selected User.
Postconditions	The User is on a page containing the calendar of the selected User.
Exception	

UseCaseName	**Show User Day**
Actors	User
Preconditions	The User has selected a User among a list of Users and their calendar is public.
Flow of events	1. The User clicks on a specific day on the selected User Calendar. 2. The system shows them the specific day with the scheduled events(if any).
Postconditions	The selected day with relative information is shown by the system.
Exception	

UseCaseName	**Show User Event**
Actors	User
Preconditions	The User has selected a specific day in which events are scheduled on another User calendar.
Flow of events	1. The User selects a specific event on the selected day. 2. The system shows him a new screen in which all the information related to the event are shown.
Postconditions	The selected event is opened in another screen with all the specific information related to it.
Exception	The event is private (and the User cannot see what it is about), the User cannot expand the event.

5.2 Class Diagram

Now that we have defined our use cases in a detailed way, we also provide the basic structure of our application, expressed by the following Class Diagram:

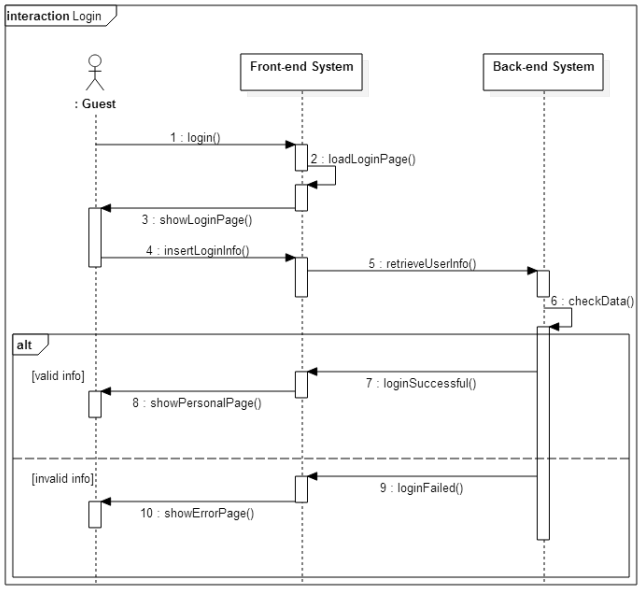


We would like to stress one important concept regarding this diagram: it is a conceptual representation of the system, useful to understand what we are wanting to do. In particular, the specification that a Calendar is going to have Years, which are going to have 12 Months and so on up to the Quarter of hours, and the fact that a Day is going to see some Events, is all a conceptual explanation of how we want to represent our Calendar. It is highly unlikely that this is going to be the actual class-planning diagram (mainly for the gigantic amount of useless Entities that *this* would cause. As a matter of fact, we are completely ignoring the subdivision made here for the Calendar in our Alloy model, as it would be useless.

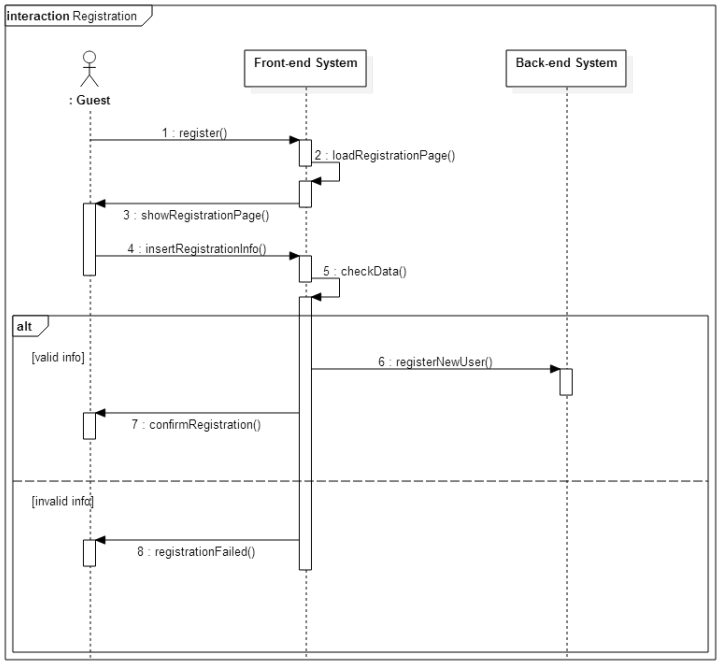
5.3 Sequence Diagrams

In order to clarify many dynamics of our application, we also provide the sequence diagrams of the most relevant operations in MyMeteoCal :

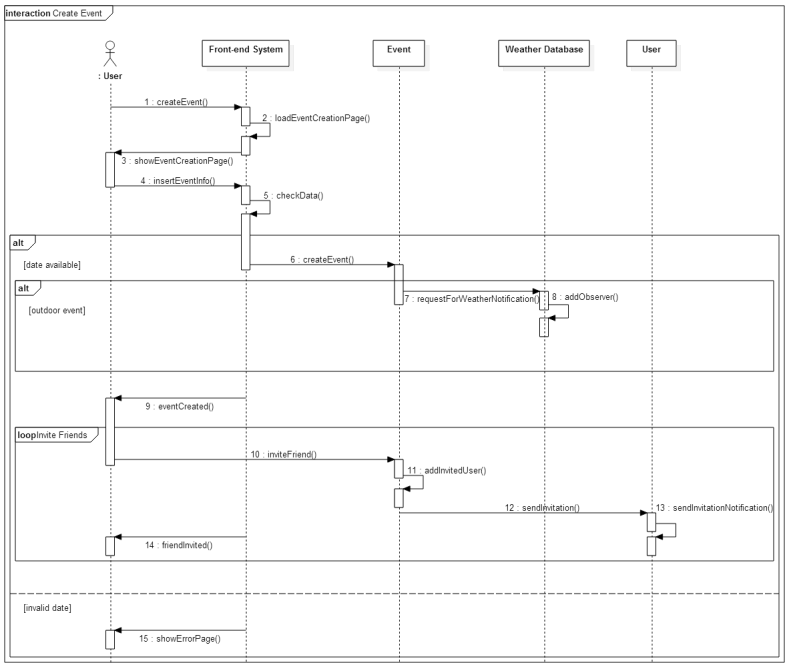
- **LogIn**



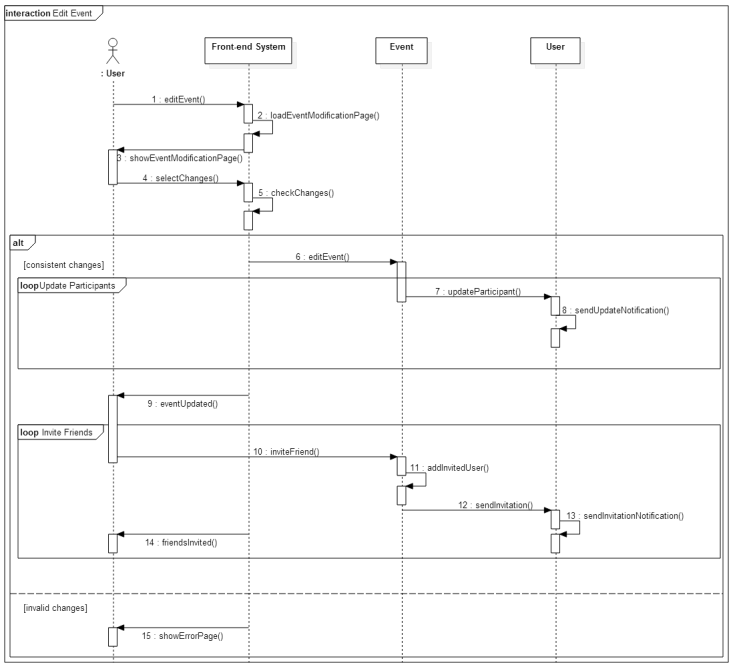
- **Registration**



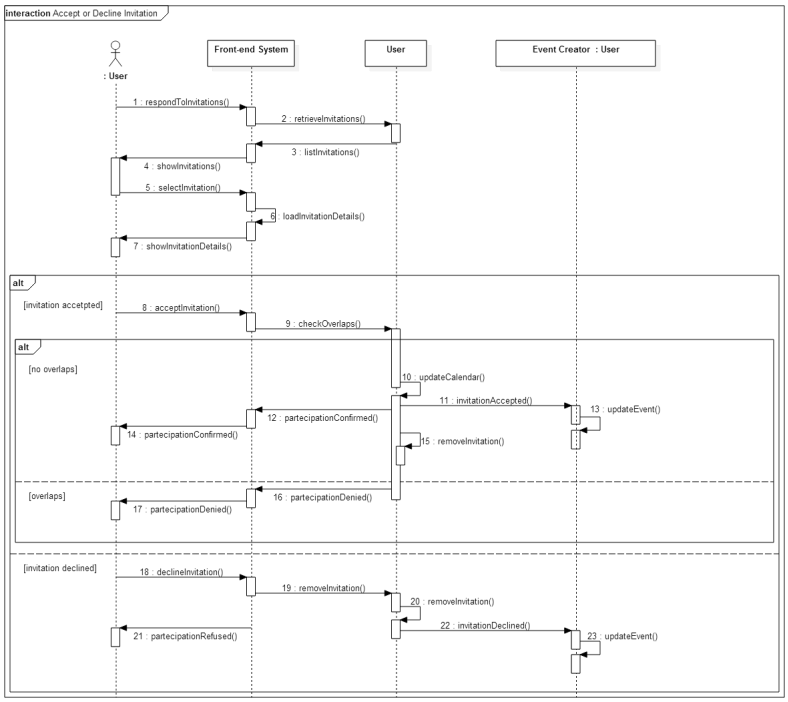
- **Create Event**



- **Edit Event**


interaction Edit Event
: User
Front-end System
Event
User
1 : editEvent()
2 : loadEventModificationPage()
3 : showEventModificationPage()
4 : selectChanges()
5 : checkChanges()
alt
[consistent changes]
6 : editEvent()
loop Update Participants
7 : updateParticipant()
8 : sendUpdateNotification()
9 : eventUpdated()
loop Invite Friends
10 : inviteFriend()
11 : addInvitedUser()
12 : sendInvitation()
13 : sendInvitationNotification()
14 : friendsInvited()
[invalid changes]
15 : showErrorPage()

- **Accept/Decline invitation**



6 Alloy

In this paragraph we are going to use Alloy Analyzer in order to evaluate our Class Diagram to check consistency.

6.1 Model

First, we show you our model, trying to be as clear and intuitive as possible. Even though it has already been said, we want to stress it again here: The Calendar is not divided here in tinier partitions of Years, Months and so on, because in our real implementation that is most likely not going to be the case, and for what is concerning alloy, our Calendar representation is useless to know. Therefore, there is only the Calendar here, and as you will see it is *more* than enough

```
module MyMeteoCal

// SIGNATURES

sig User{
    hasCalendar: Calendar,
```

```
    hasEvent: set Event,
    hasNotification: set Notification
}

abstract sig Notification{
    notificationOf: User,
    regardingEvent: Event
}

sig InviteNotification extends Notification{
}

sig WeatherNotification extends Notification{
}

sig Event{
    createdBy: User,
    hasPartecipant: some User,
    hasInvitedUser: set User,
    hasLocation: Location,
    hasStartingDate: EventDate,
    hasEndingDate: EventDate
}

sig Calendar{
    calendarOf: User
}

sig EventDate{
    isBefore: set EventDate
}

sig Location{
    hasWeather: lone Weather
}

sig Weather{
}

// FACTS

fact coherentRelationsProperty {
    // if there is an undirected relation, they must be consistent
    all u: User, c: Calendar | u in c.calendarOf iff c in u.hasCalendar
    all u: User, e:Event | u in e.hasPartecipant iff e in u.hasEvent
    all u: User, n:Notification | u in n.notificationOf iff n in u.hasNotification
```

```
}

fact noUselessLocationsProperty {
    // if there exists a location, it must be related to some event
    all l:Location | one e:Event | l in e.hasLocation
}

fact noUselessWeatherProperty {
    // if there exists a weather, it must be related to some location
    all w:Weather | one l:Location | w in l.hasWeather
}

fact noUselessEventDateProperty {
    // if there exists an EventDate, it must be related to some Event, even more than 1
    all ed: EventDate | some e:Event | (ed in e.hasStartingDate || ed in e.hasEndingDate)
}

fact creatorInvitedAswellProperty {
    // the creator of an event is also a partecipant of it
    all u: User, e:Event | u in e.createdBy implies u in e.hasPartecipant
}

fact invitedNotificationConsistencyProperty {
    //there is at most one inviteNotification for each event for a given User
    all u: User, e:Event | no disj n1,n2 : InviteNotification |
        (n1 in u.hasNotification && e in n1.regardingEvent &&
        n2 in u.hasNotification && e in n2.regardingEvent)
    //a User is invited iff he has an invitation regarding that event
    all u: User, e:Event | u in e.hasInvitedUser iff (some n: InviteNotification |
        (n in u.hasNotification && e in n.regardingEvent))
    //if a User is invited, then he is not partecipating yet
    no u:User, e:Event | (u in e.hasInvitedUser && u in e.hasPartecipant)
    //if a User is partecipating, then he has no invitations regarding that event,
    //but that should follow the statements said before.
    //this should be enough to ensure one cannot invite himself

}

fact timePrecedenceProperty {
    //every date is either before or after another date (no same dates exist)
    all disj ed1, ed2 : EventDate | (ed1 in ed2.isBefore || ed2 in ed1.isBefore)
    //transitive closure
    all ed1, ed2, ed3 : EventDate | ((ed3 in ed2.isBefore && ed2 in ed1.isBefore)
         implies ed3 in ed1.isBefore)
    //if an user has two events, they don't overlap
    no  disj u: User, e1, e2 : Event | e1 in u.hasEvent && e2 in u.hasEvent &&
```

```
        (some ed1e1, ed1e2, ed2e2: EventDate |
            (ed1e1 in e1.hasStartingDate || ed1e1 in e1.hasEndingDate)
             && ed1e2 in e2.hasStartingDate && ed2e2 in e2.hasEndingDate &&
            (ed1e1 in ed1e2.isBefore && ed2e2 in ed1e1.isBefore))
    //two events of a same user cannot start togheter
    all disj e1,e2: Event, u: User |( (e1 in u.hasEvent && e2 in u.hasEvent) implies
        no ed:EventDate | (ed in e1.hasStartingDate && ed in e2.hasStartingDate))
    //if a is before b, then b is not before a
    no ed1, ed2 : EventDate | ed1 in ed2.isBefore && ed2 in ed1.isBefore
    //if we have two eventdates related to the same event, the starting date is before the e
    //it also ensures, combined with the upper constraints, that a starting date
    //is not also an ending date
    all e:Event | all ed1, ed2: EventDate |
         (ed1 in e.hasStartingDate && ed2 in e.hasEndingDate) implies ed2 in ed1.isBefore
}

// ASSERTIONS

assert NoSelfInvitation {
    // a user cannot invite themself
    no u: User | some e: Event | u in e.createdBy && u in e.hasInvitedUser
}

check NoSelfInvitation

assert NoInvitesIfPartecipating {
    // if u is partecipating then he is not invited (anymore)
    no u: User | some e: Event | u in e.hasPartecipant && u in e.hasInvitedUser
}

check NoInvitesIfPartecipating

assert NoInstantEvent {
    // an event cannot end at the same time it starts
    no e: Event | some ed: EventDate | ed in e.hasStartingDate && ed in e.hasEndingDate
}

check NoInstantEvent

assert NoTimeLeapingEvent {
    // an event cannot end before it starts
    no e: Event, ed1, ed2 : EventDate | ed1 in e.hasStartingDate && ed2 in e.hasEndingDate &
        ed1 in ed2.isBefore
}

check NoTimeLeapingEvent
```

```
// PREDICATES

pred showAsManyInvitedAsUsers(){
    // it must be inconsistent
    #Event=1
    #Event.hasInvitedUser = #User
}

run showAsManyInvitedAsUsers for 3 but exactly 2 User

pred showSmallWorld(){
    #Event=1
    #Event.hasPartecipant=#User
}

run showSmallWorld for 3

pred showOneEvent(){
    #Event=1
    #Weather=1
    #Event.hasInvitedUser = 1
}

run showOneEvent for 2

pred showNoInvitationWorld() {
    all e : Event | #e.hasPartecipant = 1 && # e.hasInvitedUser = 0
}

run showNoInvitationWorld for 3 but exactly 2 User, exactly 3 Event, 0 WeatherNotification

pred showNoEventWorld() {
    #Event=0
}

run showNoEventWorld for 2 but exactly 2 User

pred showNoOverlapWorld(){
    #Event= 2
    all e : Event | #e.hasPartecipant = 1 && # e.hasInvitedUser = 0
    #User=1
}

run showNoOverlapWorld for 4 but exactly 4 EventDate
```

6.2 Result

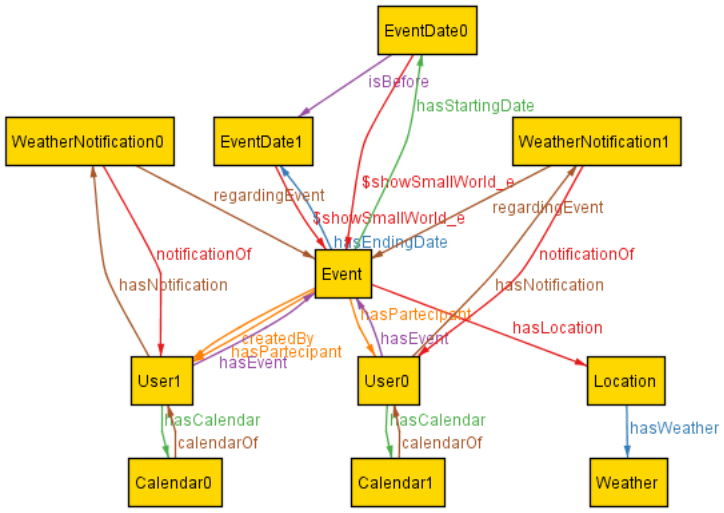
This is the result of the execution of the model:

10 commands were executed. The results are:
#1: No counterexample found. NoSelfInvitation may be valid.
#2: No counterexample found. NoInvitesIfPartecipating may be valid.
#3: No counterexample found. NoInstantEvent may be valid.
#4: No counterexample found. NoTimeLeapingEvent may be valid.
#5: No instance found. showAsManyInvitedAsUsers may be inconsistent.
#6: **Instance found.** showSmallWorld is consistent.
#7: **Instance found.** showOneEvent is consistent.
#8: **Instance found.** showNoInvitationWorld is consistent.
#9: **Instance found.** showNoEventWorld is consistent.
#10: **Instance found.** showNoOverlapWorld is consistent.

As expected, no counterexample was found for the assertions, and the predicate showAsManyInvitedAsUsers is inconsistent, as for a single Event the owner of it cannot be invited (because he is already participating). Instances were found for any other predicate, and by manually searching through them we haven't found any evident inconsistency.

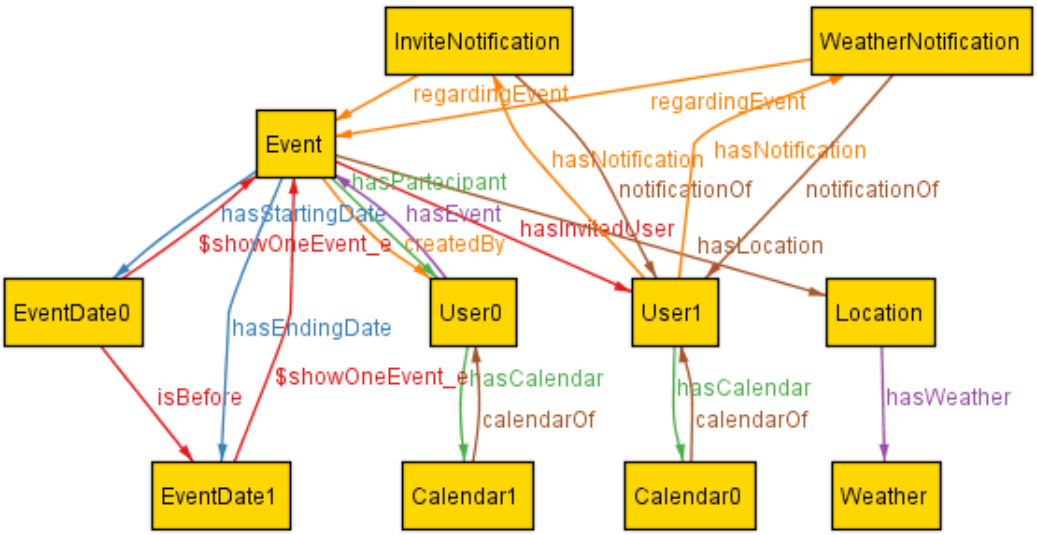
6.3 Worlds Generated

Now we quickly analyse some worlds generated (in particular one for each consistent predicate generated), starting from the most typical one.

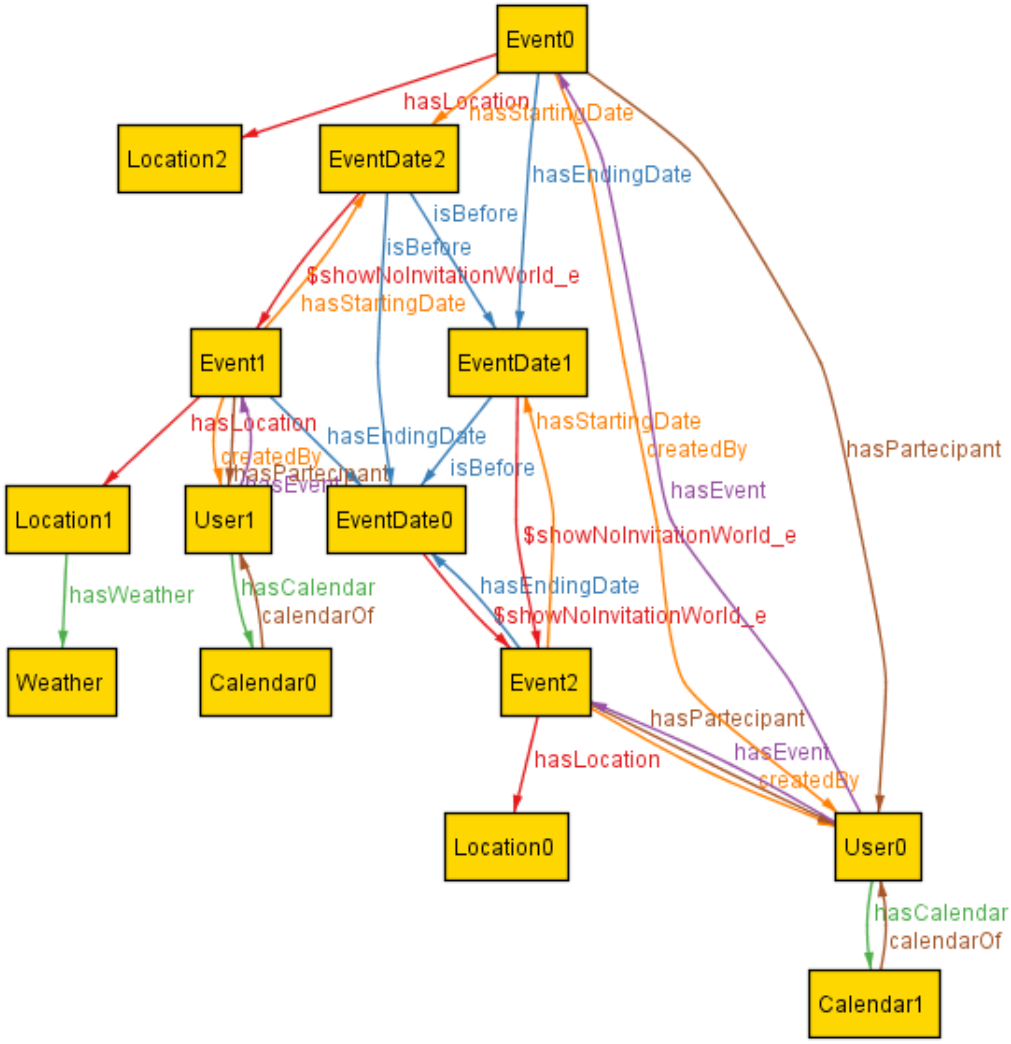


Some users can share the same Event, which is created by only one of them and is of course participating to it, the dates are Time consistent and the Event has one Location and one (optional) Weather info. Every User has his own

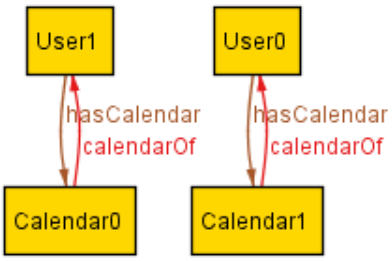
calendar and they can have some notifications, in this case only WeatherNotification because there are no other events and participating users cannot be invited again.



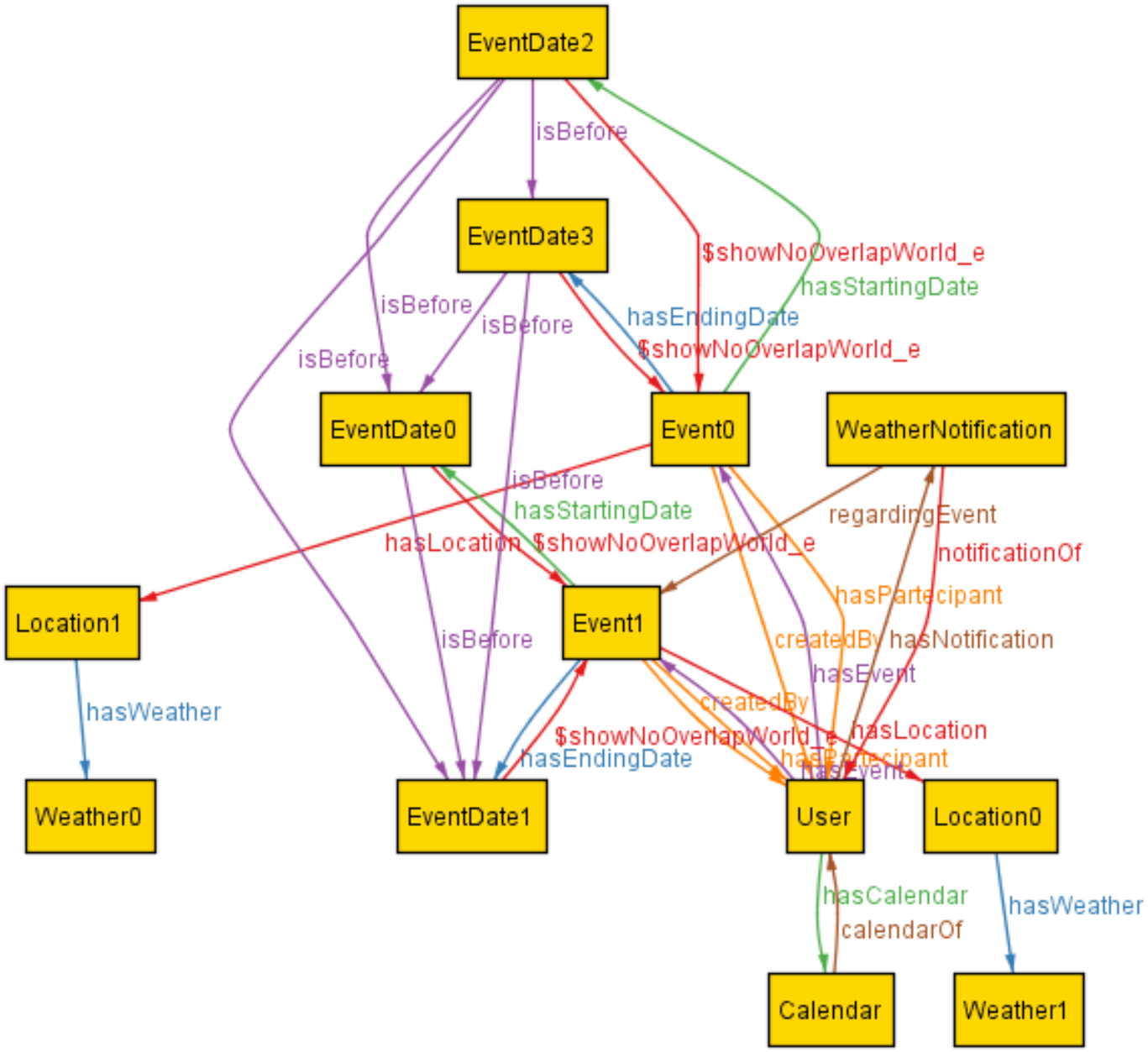
This image shows a world with only one event and two users. There is only one creator, who is also of course participating to that event, and one invited user who is not participating yet. Also, even if we will look at it clearly successively, the event has both starting and ending dates, and the starting date is before the ending date.



Here there is a world where every User only uses their events for themselves, without inviting anybody else. Notice also how both events 0 and 2 are made by User0. There are only three dates, and the two events happen one after the other, being consistent with time precedence, with the second event starting exactly when the first ends.



This is by far the most trivial case possible. It shows a world where no events exist (yet). We thought that if our system hasn't a consistent initial state, then it couldn't go to any other more complex state, so it was worth ensuring this case.



Last, the most critical point of all the model is put into scope: time consistency. Here we have a single User handling two events. This time we explicitly asked to have 4 Dates to see how the system would create the worlds, and they are all the same (on this part): The events cannot overlap. In this specific case Event0 is the first, as EventDate2 is before any other date, and it ends with EventDate3 which is only after EventDate2. Then Event1 occours, starting with

EventDate0 which is before the end, EventDate1.