CALENDIARY

Pattern application

OOANS: semestral paper

Date: 13.5.2018
Year: 1. Ing.
Academic year: 2017/2018
Exerciser: Ing. Lukáš Doubravský

Peter Berta
Gabriela Hózová

Obsah

1. Introduction

People usually forget about important matters when faced with large amount of different responsibilities. To reduce stress induced by keeping all the matters in order, people tend to use various notes, reminders or calendars. Even though everybody is more or less familiar with basic functions of calendars, diary-like applications bring more advanced features every day.

A calendar is a system of organizing days for social, religious, commercial or administrative purposes. Productivity levels vary from day to day, and modern tools are supposed to help us work better and faster. Calendars should help us focus on our most important tasks instead of showing us how much work we still need to do in a short time.

We can keep track of all responsibilities at any moment with these types of applications. Calendar can be accessed from any device, including smartphone, and we can always check upcoming tasks without needing to carry agenda with us. If we want to update our schedule while commuting to work, there is no problem. We can have a look at all our tasks right now.

When working on some team projects, we can share our schedules with other people.

In addition to that, people sometimes want to write some notes to specific days like in diary. In most calendars, it is usually not possible.

Great number of students tend to struggle with their time schedules and have overall bad time management. We want to help them by creating a calendar application with task management functionality.

In this paper, we design new application, so we are working on designing patterns without refactoring.

1.1. Project goal

In our project, we want to merge calendar features and diary features. Our application will be easy-to-use, intuitive calendar, where various notes can be added, events can be created, and users can share their schedules with other people. It should be innovative solution for planning and time management.

2. Project Specification

In this project, we design an application which connects functionality of a calendar and a diary. User will be able to create events and add reminders to them. During the day, user will be able to add any text or note the particular day, just like into a diary. Furthermore, there will be a possibility of evaluating the day by number. If the day was enjoyable, user can rate it with high value. Based on these records, application will be able to predict how enjoyable the next day will be, depending on planned events. Additionally, user will be able to display rating of past days and export them into compact and comprehensible form.

2.1. Architectural style

For this application, based on present state of analysis, we decided to choose client-server pattern. Furthermore, elements of object-oriented architecture will be present.

3. Functional Requirements

- Graph visualization
- Creating and editing events
- Filtering existing events
- Displaying the list of events after a successful search
- Creating and editing reminders for events
- Daily evaluation
- Account management
- User discovery - friend requests
- Creating snapshots to share calendar with other people

4. Modules

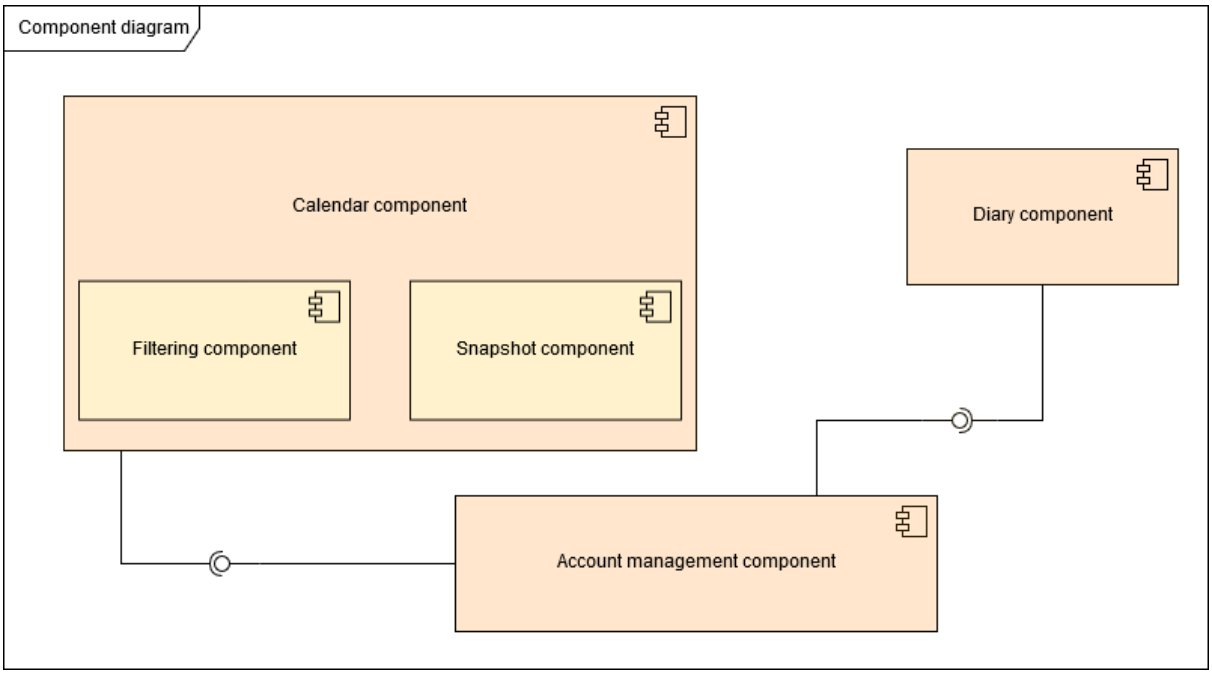

Figure 1: Component diagram

4.1. Calendar component

This module will handle all functionality concerning events and reminders. Users will have the possibility to create and edit events, add simple additional information like place, notes, tags or associates. Also, they will be able to set reminders for all their events and edit them later when necessary.

4.2. Filtering component

Part of this module is a component, which will take care of event filtering. When user has too many events in one day or week, it can be hard to find what he is looking for. For this purpose, there will be a possibility to filter out different events based on their name, tag, place or associates.

4.3. Snapshot component

Application will support a possibility to create a snapshot of chosen time window. This way, users are able to share their events of the day or week without giving access to their whole calendar. This module will handle snapshot creation as well as process request from users to see others' snapshots. Requested user can approve or deny access to their snapshots.

4.4. Diary component

Responsibility of this module will be to take care of users notes and evaluations of the day. From these records, this module will try to predict next day, or week based on events, which already occurred in the past.

4.5. Account management component

User will be able to create an account, which will bear his personal information and store all of his data. He can then sign in at different computer and all of his settings and data will be available there. Users can also add other users as a friend. They can be later referenced in events as associates to express, that this user is also attending certain event.

5. Use Cases

ID	Name	Description
UC01	Create event	User can create an event in calendar.
UC02	Delete event	User can delete existing event in calendar.
UC03	Add reminder to event	Reminder to event can be added to new or already existing event.
UC04	Display the week overview	User can make overview of one or more weeks.
UC05	Create snapshot	After confirmation of request to snapshot, snapshot is created.
UC06	Filter the events	User can filter the events according to time, type of event…
UC07	Predict mood	System can predict mood according to users rating and events.
UC08	Add note to day	Application has diary feature, so user can write his notes to specific days.
UC09	Rate day	User can rate every day and system than can predict user's mood to next days.
UC10	Authenticate	System authenticates user's login.
UC11	Request snapshot	User can request snapshot of someone's calendar.
UC12	Create user	After registration, new user in database is created.
UC13	Delete user	Administrator can delete users (flag as argument).
UC14	Manage event	CRUD use case.
UC15	Manage user	CRUD use case.
UC16	Sign in	Before seeing the calendar, user must sign in.
UC17	Sign up	If user is not registered, he must sign up.
UC18	Update event	User can edit event information.

UC19	Update user	User can edit information about himself.
UC20	View event	Details of events can be displayed.
UC21	View user	Details of users can be displayed.

Tab. 1: List of use case

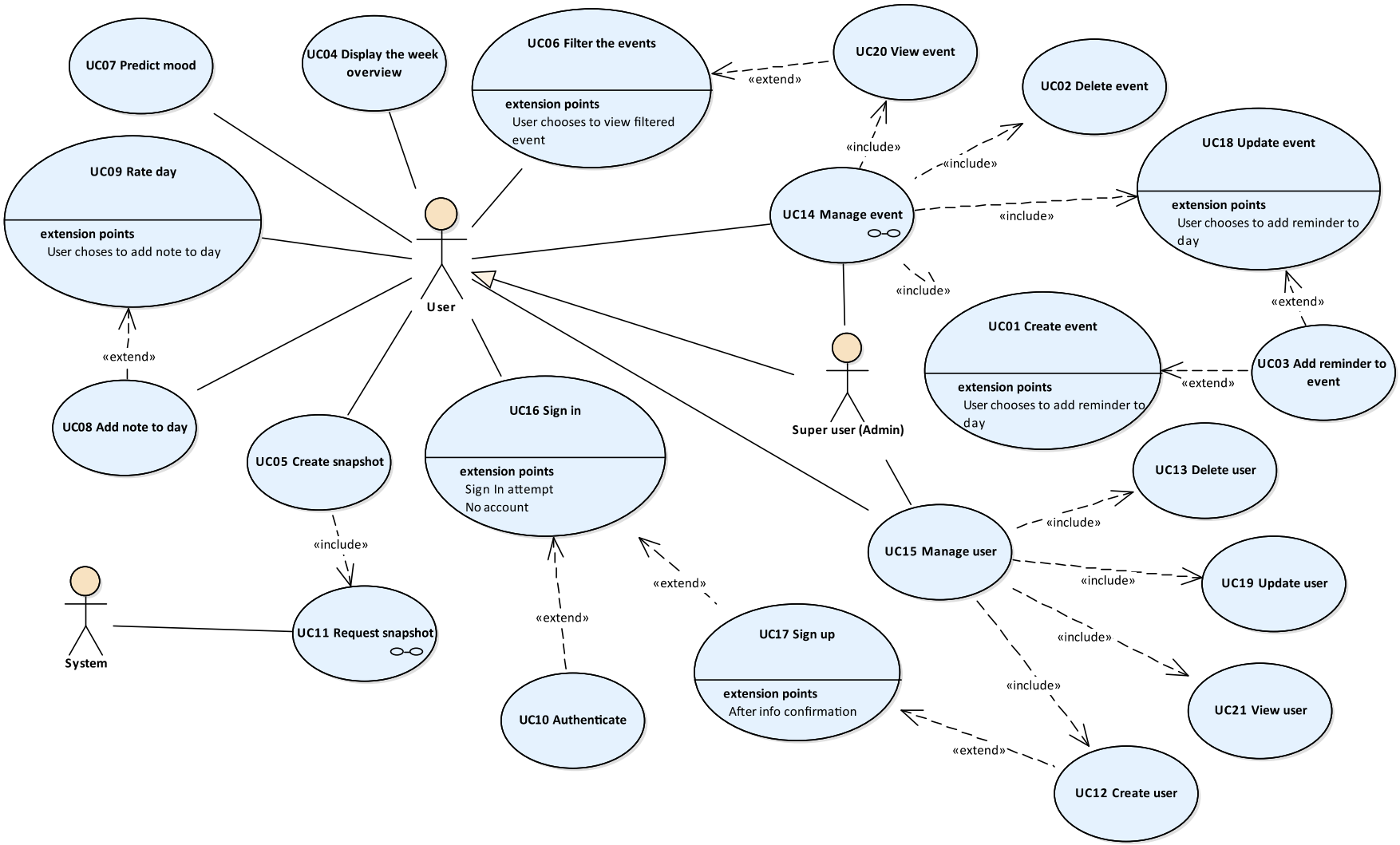

Figure 2: Use case diagram

5.1. UC01 Create an event

CHARACTERISTIC INFORMATION

Goal in Context: User wants to create an event in calendar.

Level: Primary task

Preconditions: User knows information about the event which is being created. The main screen is open.

Success End Condition: The event is created and is visible in calendar.

Failed End Condition: The event is not created or is not visible in calendar.

Primary Actor: User

Trigger: User wants to create an event.

MAIN SUCCESS SCENARIO

1. User chooses to create new event.
2. System shows form for new event.
3. User fills in information.
4. User confirms event creation.
5. System creates new event according to provided information.

EXTENSIONS

3a. User highlighted the day of the event.
3a1. System fills in the day of the event with the highlighted day by user.
3b. User can set up a reminder.
3b1. Continue with use case <UC03 Add a reminder to an event>
4a. Event with the same information already exists.
4a1. System prompts user to review event information.

SCHEDULE

Due Date: release 1.0

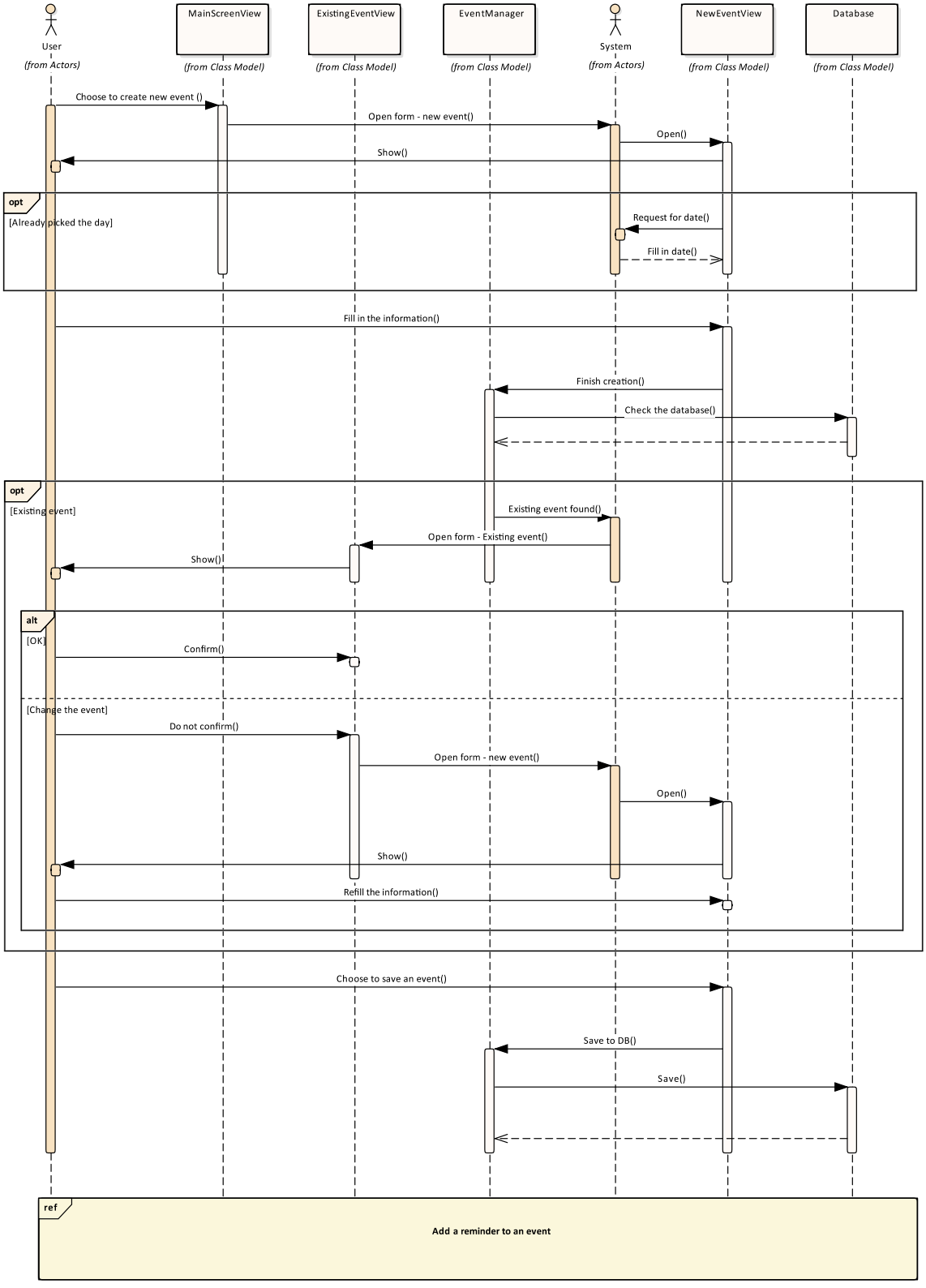

Figure 3:Sequence diagram of UC01 – Create an event

5.2. UC02 Delete an event

CHARACTERISTIC INFORMATION

Goal in Context: User wants to delete an event in calendar.

Level: Subfunction

Preconditions: User have created the event.

Success End Condition: The event is deleted and is not visible in calendar.

Failed End Condition: The event is not deleted and is visible in calendar.

Primary Actor: User

Trigger: User wants to delete an event.

MAIN SUCCESS SCENARIO

1. User chooses the existing event from calendar.
2. System shows the detail of the selected event.
3. User chooses the option to delete the event.
4. User confirms event removal.
5. System deletes the event from calendar.

EXTENSIONS

9a. User does not confirm event removal.

SCHEDULE

Due Date: release 1.0

5.3. UC03 Add a reminder to an event

CHARACTERISTIC INFORMATION

Goal in Context: User wants to add a reminder to event in calendar which has already been created.

Level: Subfunction

Preconditions: The event has to be created before adding a reminder to it.

Success End Condition: User gets the reminder before event starts.

Failed End Condition: User does not get the reminder before event starts.

Primary Actor: User

Trigger: User wants to add a reminder to event which has already been created.

MAIN SUCCESS SCENARIO

1. User chooses to add a reminder to an event.
2. System shows a form for editing event.
3. User fills in information about the reminder.
4. User confirms event editing.
5. System updates event information.

SCHEDULE

Due Date: release 1.0

5.4. UC06 Filter events

CHARACTERISTIC INFORMATION

Goal in Context: User wants to find specific events.

Level: Primary task

Preconditions: There have been some events already created.

Success End Condition: The events have been filtered according to user’s requirements.

Failed End Condition: The events have not been filtered according to user’s requirements.

Primary Actor: User

Trigger: User wants to find specific events.

MAIN SUCCESS SCENARIO

1. User chooses to filter events.
2. System shows a form for filtering.
3. User fills in the specific information.
4. System shows the events that suit the wanted criteria.

EXTENSIONS

4a. System shows message, that there are not any suitable events.

SCHEDULE

Due Date: release 2.0

5.5. UC04 Display the week overview

CHARACTERISTIC INFORMATION

Goal in Context: User wants to display the week overview of his events in calendar.

Level: Primary task

Preconditions: There have been already created some events. The main screen is open.

Success End Condition: The overview is displayed.

Failed End Condition: The overview is not displayed.

Primary Actor: User

Trigger: User wants to display the week overview of his events in calendar.

MAIN SUCCESS SCENARIO

1. User chooses to display the overview.
2. System shows form for displaying overview.
3. User picks start and end date.
4. System checks, if start date is before end date.
5. System displays overview of chosen date range.

EXTENSIONS

4a. Start date is after end date in calendar.
4a1. System return user to picking dates.

SCHEDULE

Due Date: release 2.0

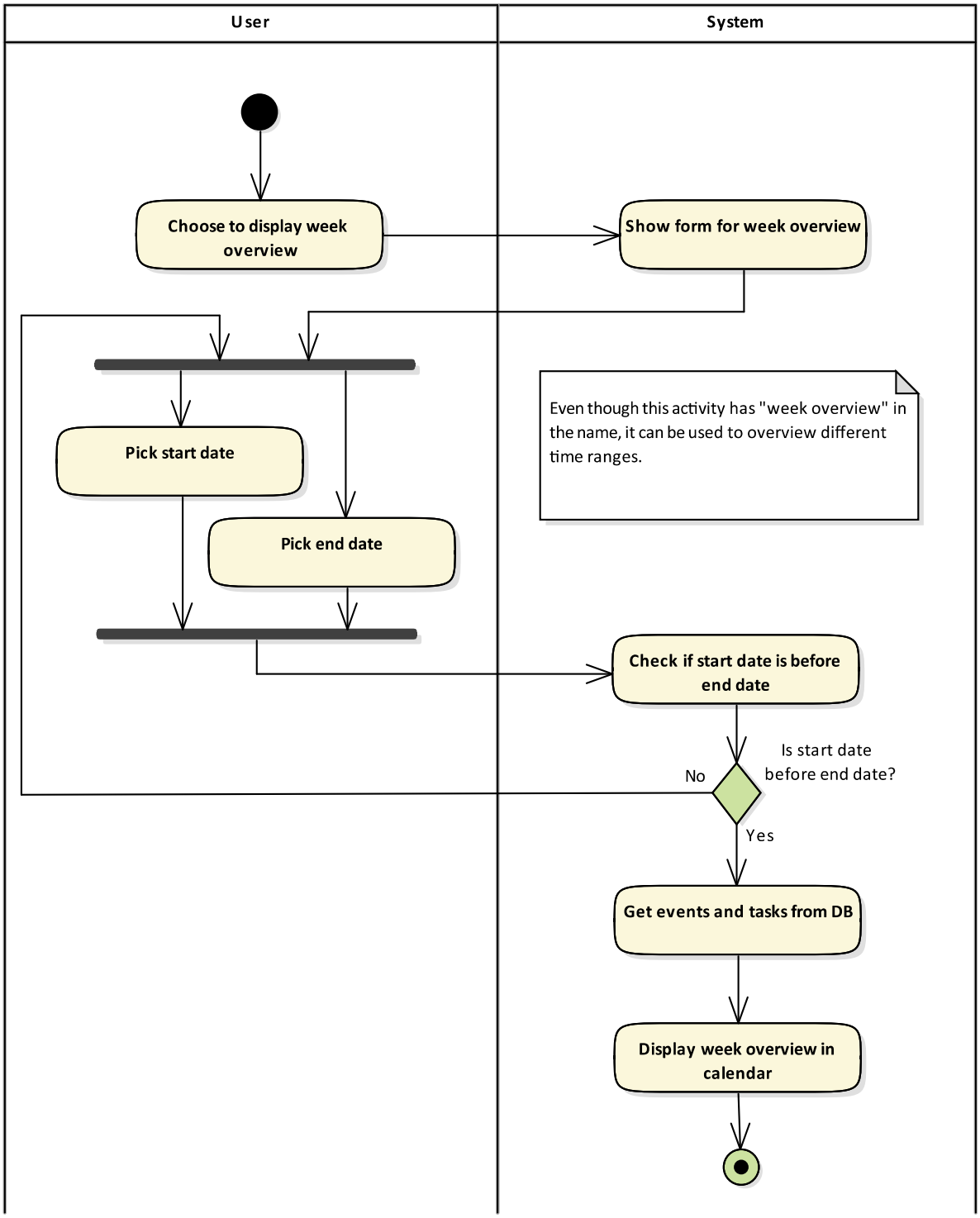

Figure 4: Activity diagram of UC04 – Display the week overview

5.6. UC06 Filter the events

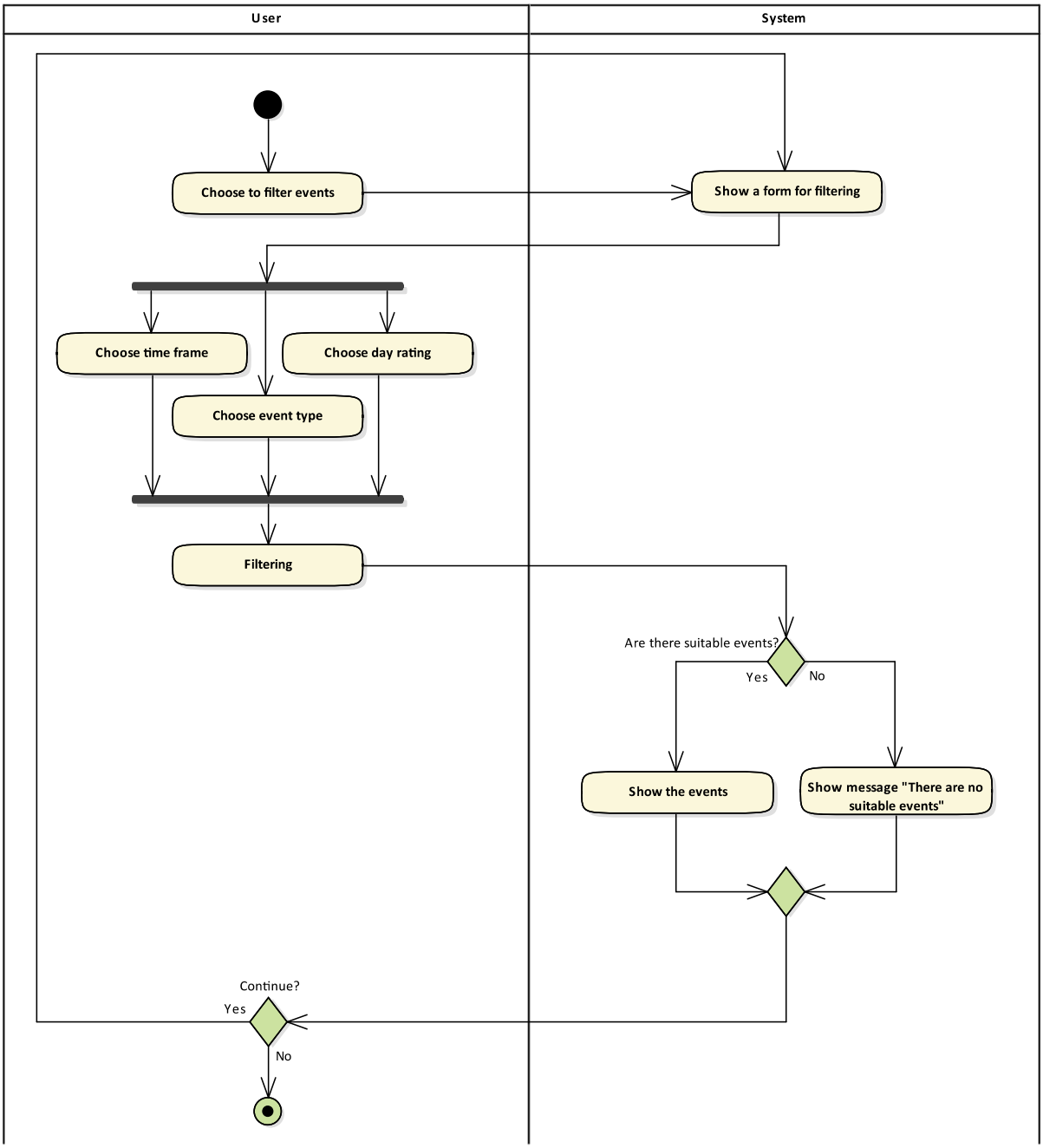

Figure 5: Activity diagram of UC06 – Filter the events

User can filter events by different criterions. They can be filtered by time, event type or rating assigned by the user. If there are no events to be displayed, a message “There are no suitable events” is shown. After that, another filtration can be performed, to get better results.

5.7. UC08 Add a note to day

CHARACTERISTIC INFORMATION

Goal in Context: User wants to add a note to day as he would write to the journal.

Level: Primary task

Preconditions: The main screen is open.

Success End Condition: The note has been added to day.

Failed End Condition: The note has not been saved.

Primary Actor: User

Trigger: User has experienced something interesting or just want to write something down.

MAIN SUCCESS SCENARIO

1. User chooses to add a note to a day.
2. System shows a form for adding a note.
3. User fills in information about the note.
4. User confirms note adding.
5. System saves the note to day.

EXTENSIONS

3a. User highlighted the day in the calendar view (main screen).
3a1. System fills in the day in form for adding with the highlighted day by user.

SCHEDULE

Due Date: release 1.0

5.8. UC11 Request snapshot

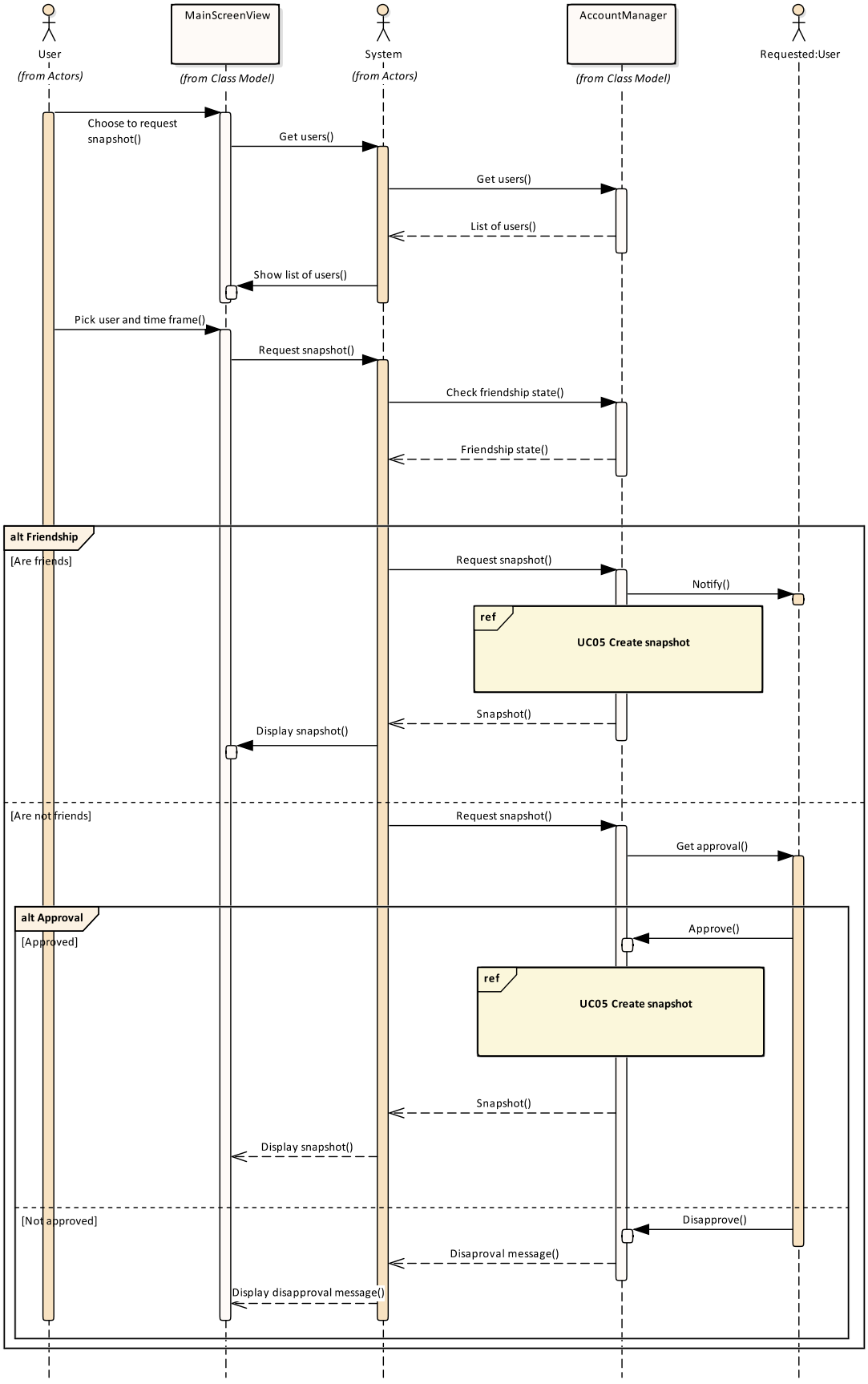

Figure 6: Sequence diagram of UC11 – Request snapshot

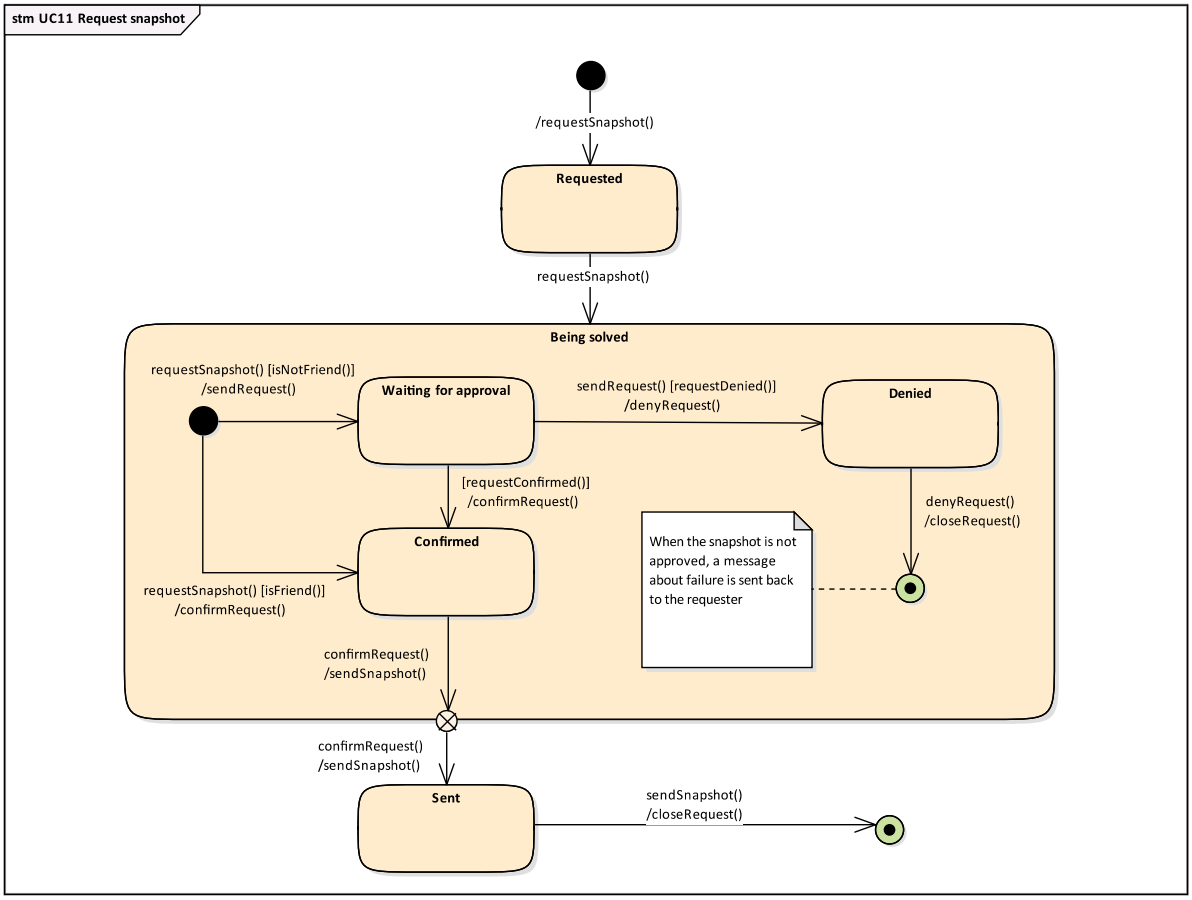

Figure 7: State diagram of element Snapshot

User can request to see other user's events from designated time frame. If these two people are friends, requested events are automatically packed into snapshot and send to the requestor. In case these users are not friends, requested user must approve of creating and sending the snapshot. After approval, snapshot is created and sent to the requestor. In the other case, message about denial is sent to the requestor.

5.9. UC14 Manage event

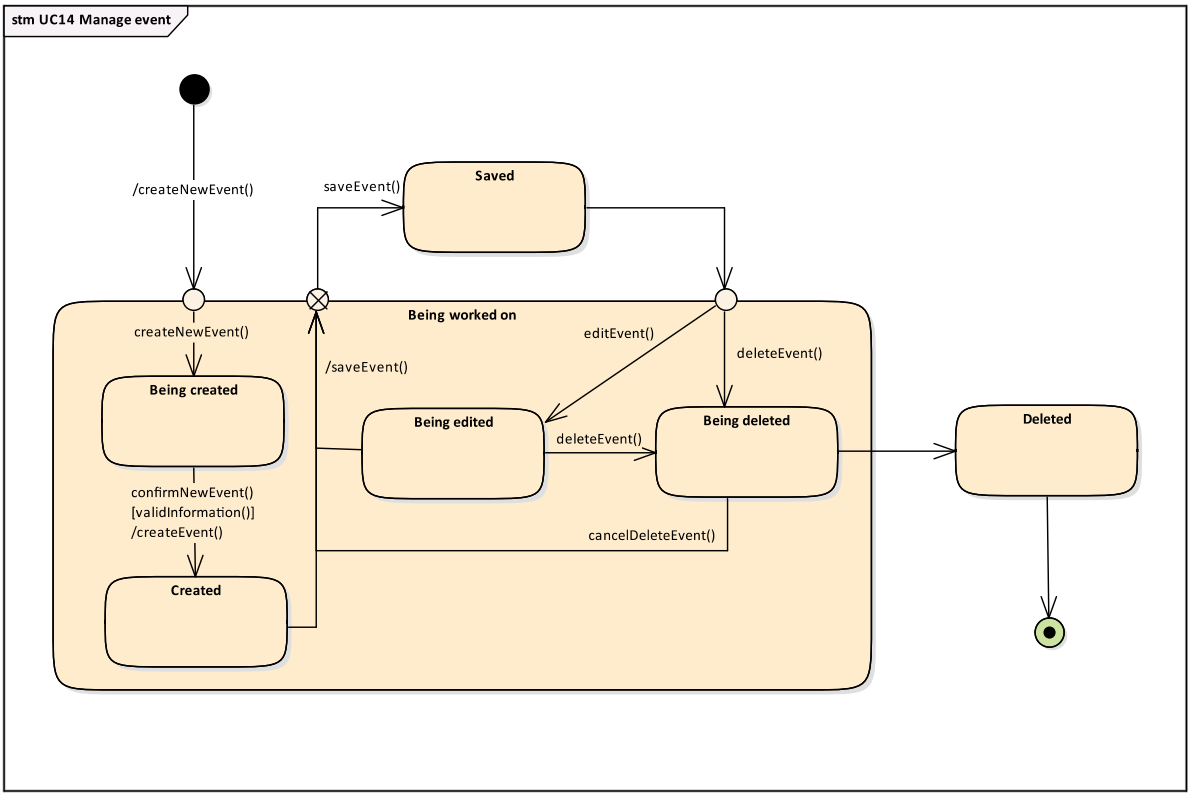

Figure 8: State diagram of element Event

Event can be created by a user. Upon deciding to create an event, a sample event is created, which holds temporary event data filled by user. When the event is completed, it is saved to the database. Event can be edited at any time. User can also delete an event.

5.10. UC18 Update event

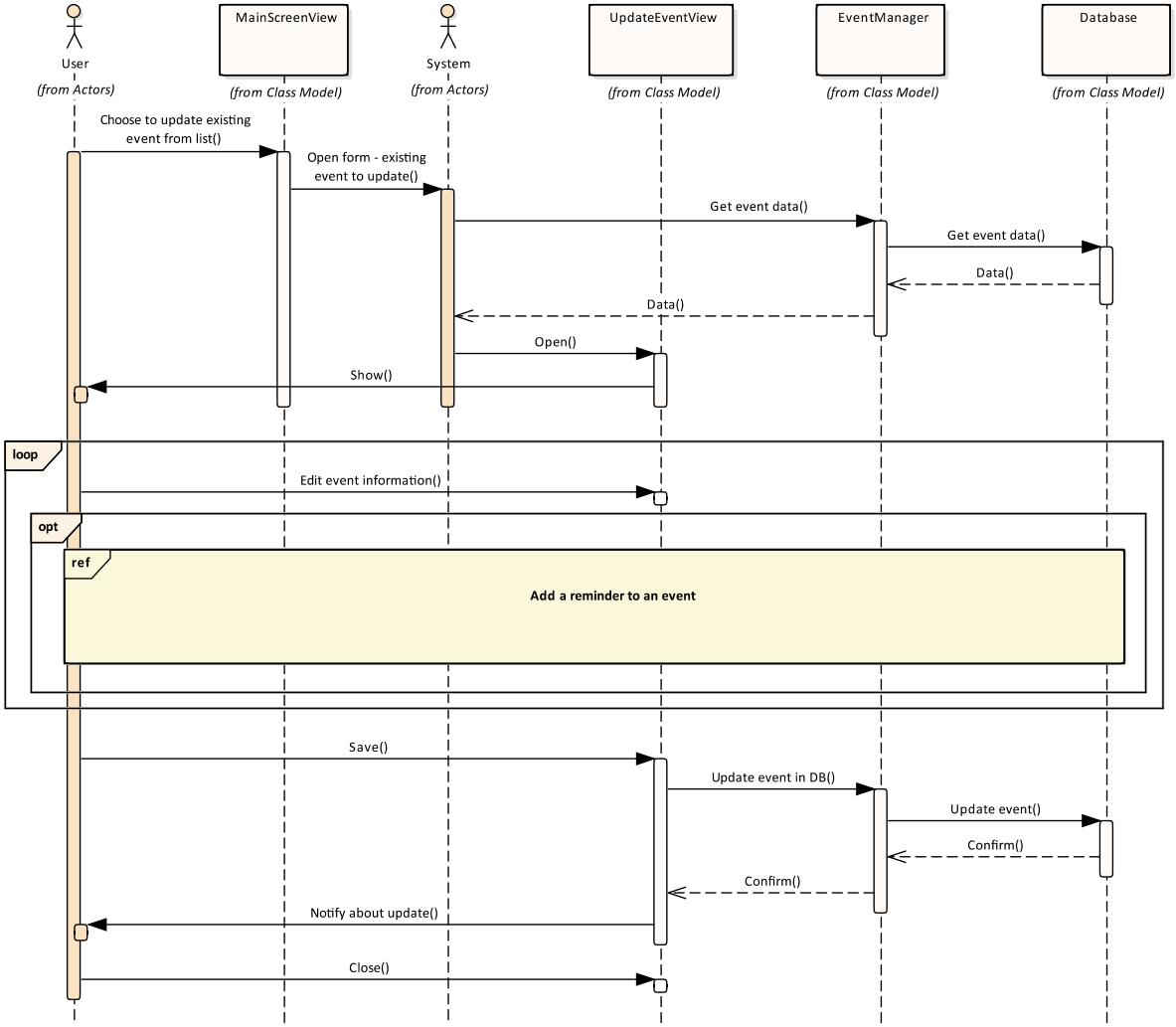

Figure 9: Sequence diagram of UC18 – Update event

User can decide to update an event at any time. He can add multiple reminders to one event. After user changes event data in the application, event in database is also updated.

6. Class Diagram

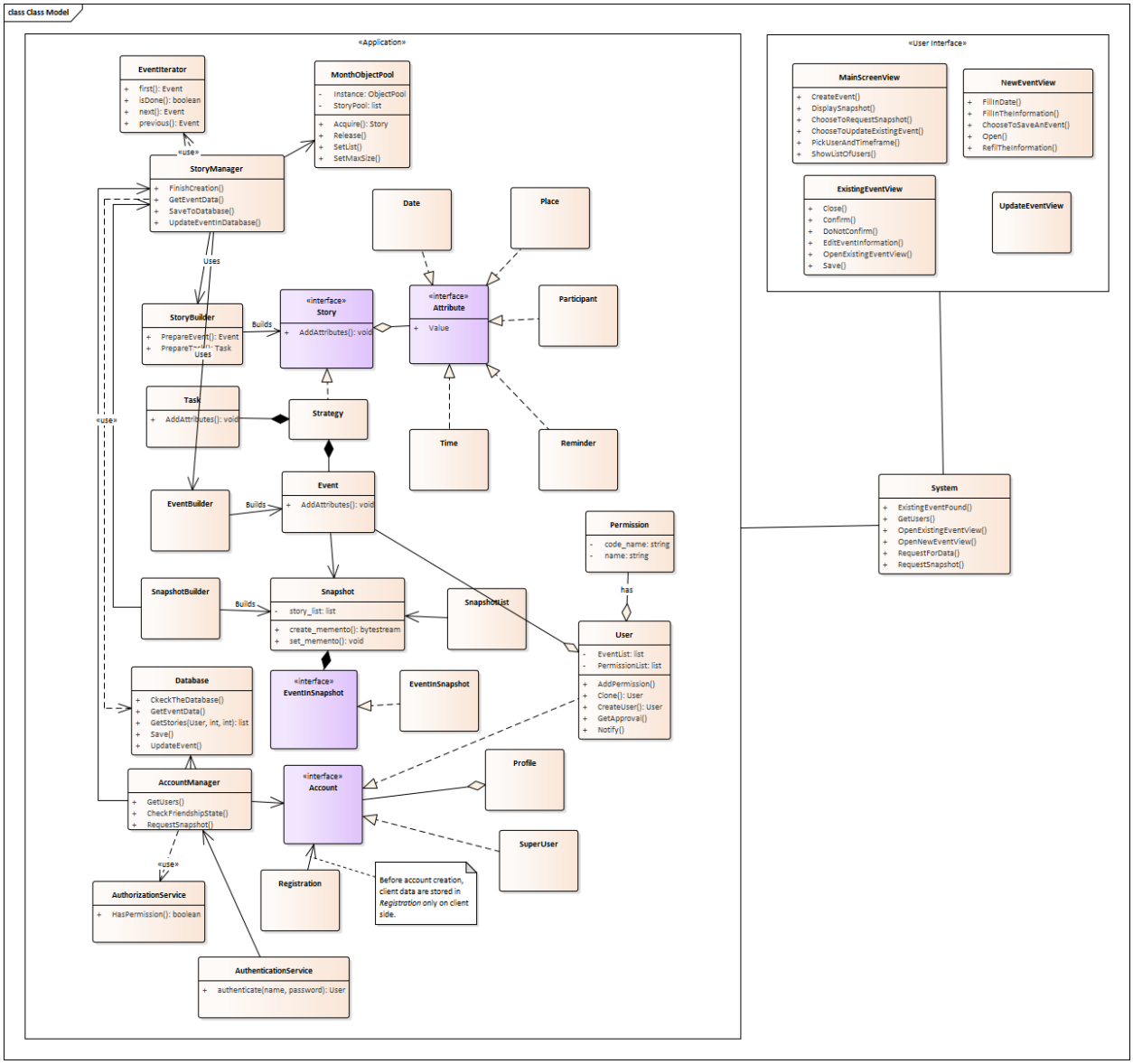

Figure 10: Class model of system CALENDIARY

7. Design patterns

7.1. Authorization/Authentication

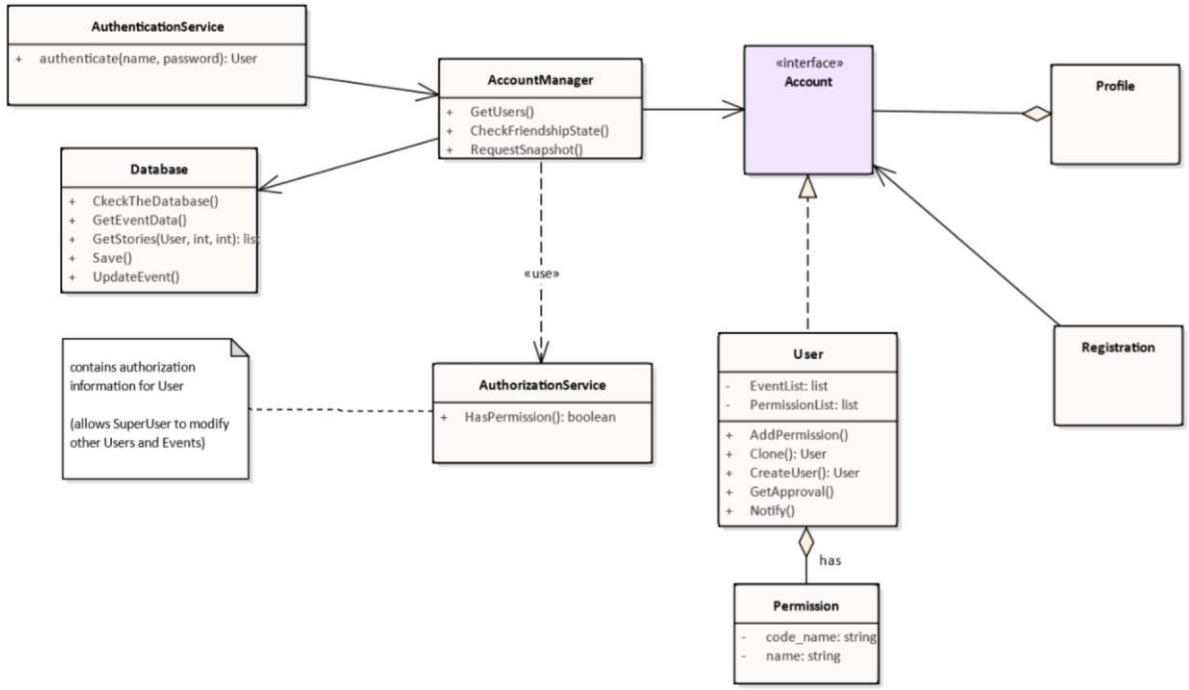

Figure 11: Class model of design pattern Authorization/Authentication

```
"""
Authorization and authentication of user.
"""

# User creation
from calendiary.models import User
user = User.CreateUser('gabca', 'gabca@fiit.isthebest', 'mamnovetenisky')

# User authentication
import calendiary.services.AuthenticationService
user = AuthenticationService.authenticate(username='john', password='secret')

if user is not None:
    pass
else
    pass


# User Authorization
from calendiary.services import Permission

permission = Permission.create(
    codename='can_publish',
    name='Can Publish Posts',
)

user.AddPermission(permission)

# Check for Authorization
from calendiary.services import AuthorizationService

permission_code = "can_do_sth"
if AuthorizationService.HasPermission(user, permission_code):
    pass
```

Figure 12: Source code of used design pattern Authorization/Authentication

Design patterns authorization and authentication are used in this project to authenticate user to do something. Each user has some permissions, so it has to be controlled.

7.2. Memento

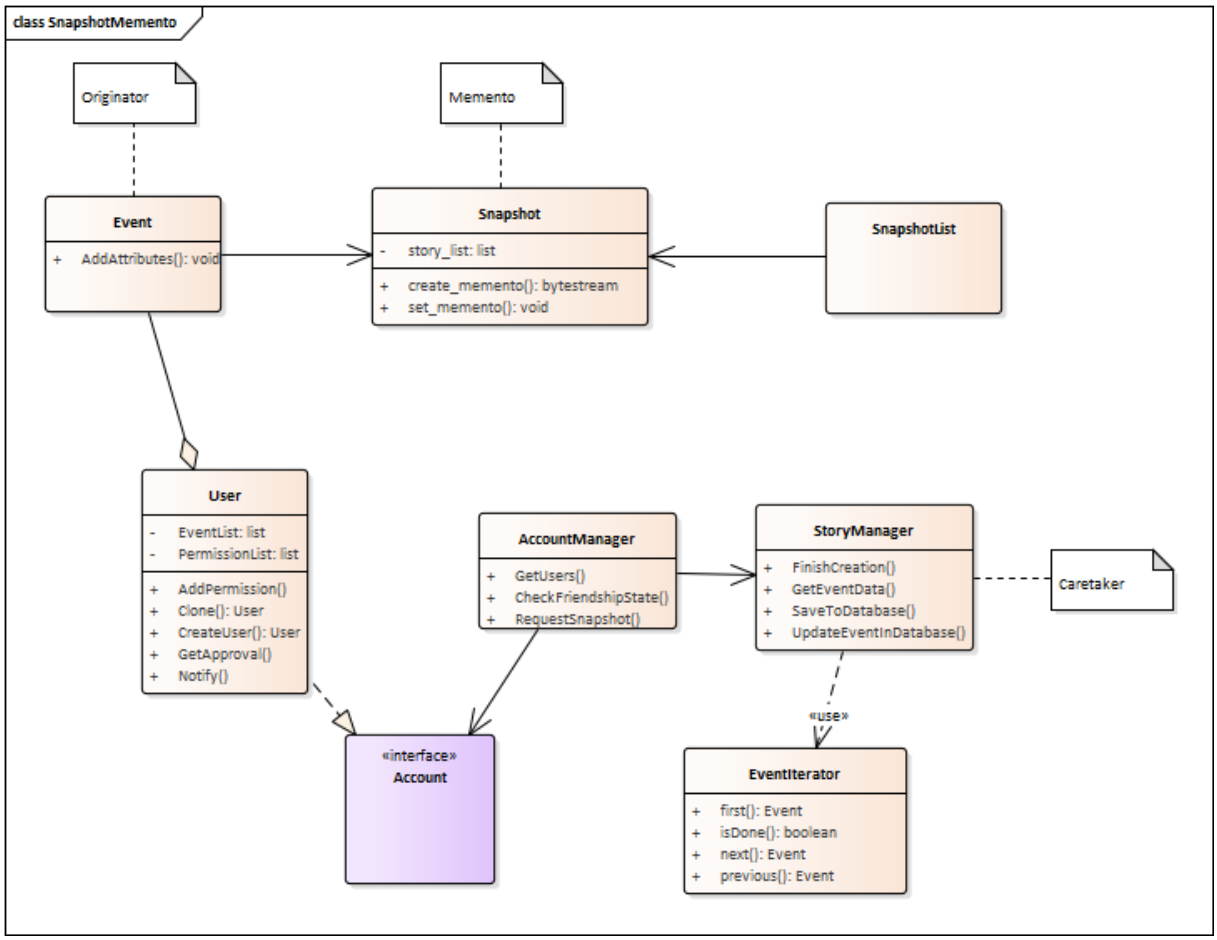

Figure 13: Class model of used design pattern Memento

```
"""
Creates a snapshot of certain stories to be sent.
This snapshot can be recovered by requestor/receiver and reviewed.
Represents a memento pattern.
"""

class Snapshot():
    """
    Includes stories to be sent.
    Also creates and recovers memento state.
    """

    def __init__(self, story_list):
        self._story_list = story_list
        self._memento = None

    def set_memento(self, memento):
        # Create memento object from byte stream and update from it
        recieved_memento = pickle.loads(memento)
        vars(self).clear()
        vars(self).update(recieved_memento)

    def create_memento(self):
        # Return snapshot of 'self'
        return pickle.dumps(vars(self))
```

Figure 14: Source code of used design pattern Memento

The Memento captures and externalizes an object's internal state so that the object can later be restored to that state. This pattern is used in the project to allow doing snapshots of selected time range. The object of memento keeps state of a calendar of one person and is passed to another person.

7.3. Object pool

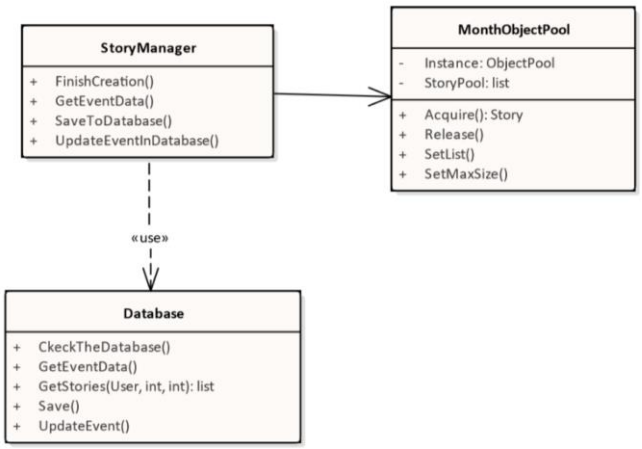

Figure 15: Class model of used design pattern Object pool

```
"""
Object pool pattern for storing stories from database.
"""

class ObjectPool:

    instance = None

    def __init__(self):
        if not ObjectPool.instance:
            self._story_pool = []
            ObjectPool.instance = self._story_pool
        else
            pass

    def _acquire(self):
        return self._story_pool.pop()

    def _release(self, story):
        self._story_pool.append(story)

    def _set_list(self, story_list):
        self._story_pool = story_list

    def _set_max_size(self, max_month):
        self._max_month = max_month



def sample():
    object_pool = ObjectPool()
    stories = DB.get_stories(user, month, year)
    object_pool._set_list(stories)
```

Figure 16: Source code of used design pattern Object pool

Object pools are used to manage the object caching. A client with access to an object pool can avoid creating new objects by simply asking the pool for one that has already been instantiated instead. In this system object pool is used to cache the objects of created instances of months. If person has opened month for example March, he can move forward on April and back on February. These all months will be saved in the pool.

7.4. Singleton

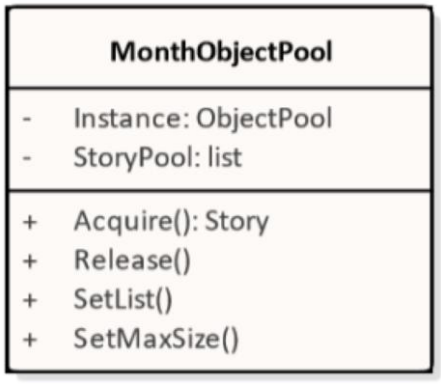

Figure 17: Class model of used design pattern Singleton

```
4
5   class ObjectPool:
6
7       instance = None
8
9       def __init__(self):
10          if not ObjectPool.instance:
11              self._story_pool = []
12              ObjectPool.instance = self._story_pool
13          else
14              pass
```

Figure 18: Source code of used design pattern Singleton

Application needs one, and only one, instance of an object. It is used for object pool, because there must be just one instance of it.

7.5. Prototype

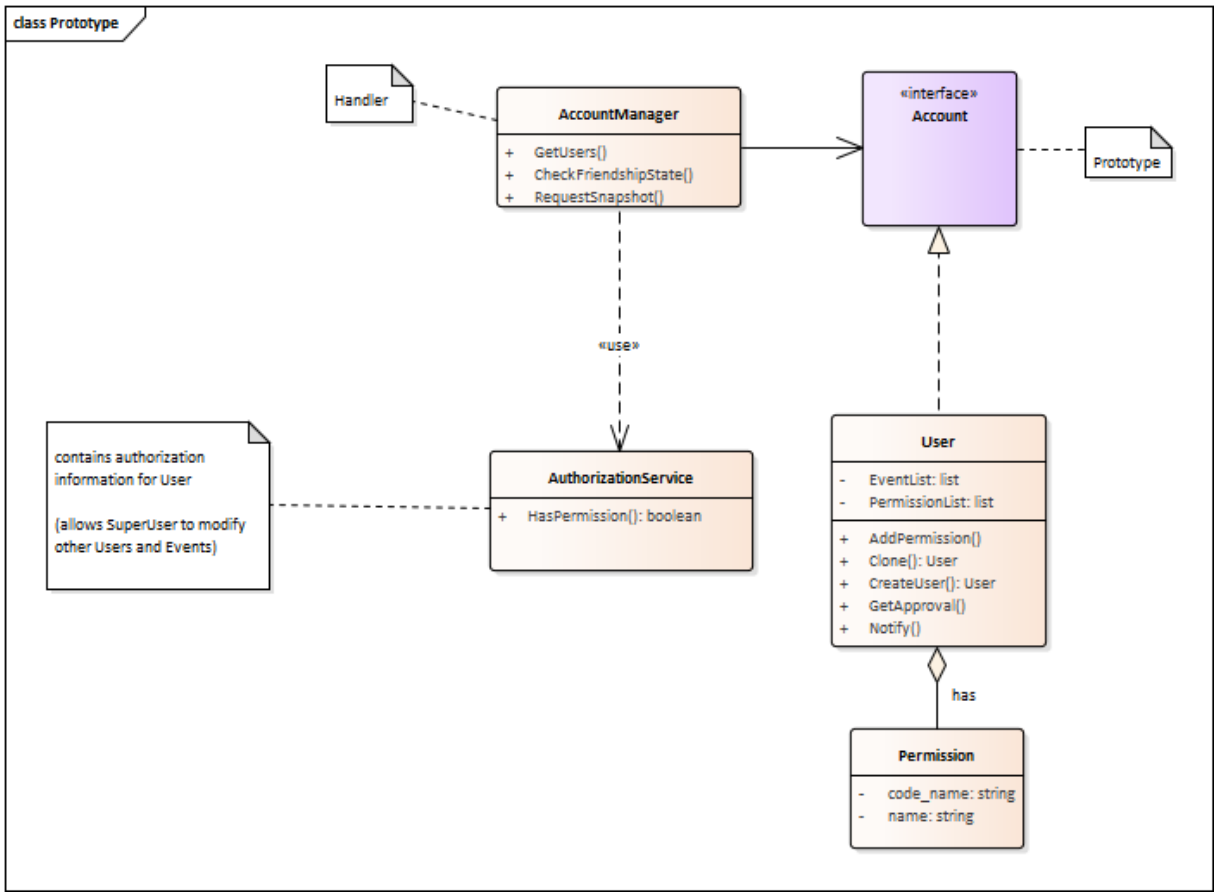

Figure 19: Class model of used design pattern Prototype

```python

import copy

class Prototype_User:
    """
    Example User to be copied.
    """

    def __init__(self):
        self._event_list = []
        self._permission_list = []

    def __AddPermission__(self, Permission):
        self._permission_list.append(Permission)


def main():
    prototype = User()
    prototype.__AddPermission__(sample_permission)

    # When creating New user
    new_user = copy.deepcopy(prototype)
```

Figure 20: Source code of used design pattern Prototype

The Prototype pattern specifies the kind of objects to create using a prototypical instance. In this case the design pattern prototype is used to create prototype of user and then add him the permissions.

7.6. Iterator

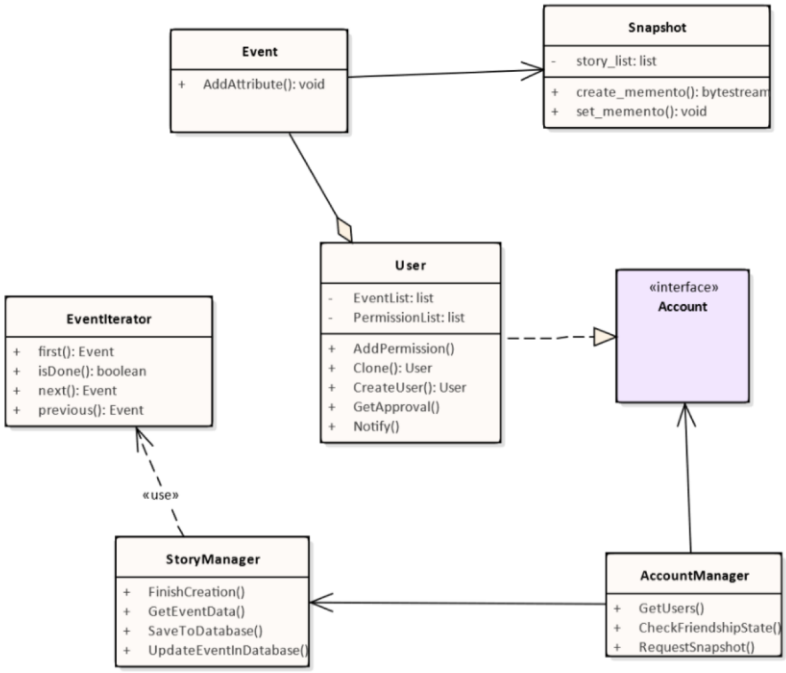

Figure 21: Class model of used design pattern Iterator

```
"""
Iterator of events which collects suitable events for new snapshot.
"""

import collections.abc


class EventIterator():
    """
    Implement the Event Iterator.
    """

    def __init__(self):
        pass

    def iter(self, list):
        self.__iter__(list);

    def __iter__(self, list):
        self._event_list = list;

    def __first__(self):
        if len(self._event_list) > 0:
            return self._event_list[0]

    def __is_done__(self):
        if self.__next__ is not None:
            return False
        else
            return True

    def __next__(self):
        if self._i < len(self._event_list):
            i = self._i
            self._i += 1
            return self._event_list[i]
        else:
            return
```

Figure 22: Source code of used design pattern Iterator part 1

```
31
32      def __next__(self):
33          if self._i < len(self._event_list):
34              i = self._i
35              self._i += 1
36              return self._event_list[i]
37          else:
38              return
39
40      def __previous__(self):
41          if self._i > 0:
42              i = self._i
43              self._i -= 1
44              return self._event_list[i]
45          else:
46              return
47
48
49  def main():
50      concrete_aggregate = ConcreteAggregate()
51      for _ in concrete_aggregate:
52          pass
53
```

Figure 23: Source code of used design pattern Iterator part 2

The Iterator provides ways to access elements of an aggregate object sequentially without exposing the underlying structure of the object. We use iterator to access objects of events to make list of events for snapshot.

7.7. Builder

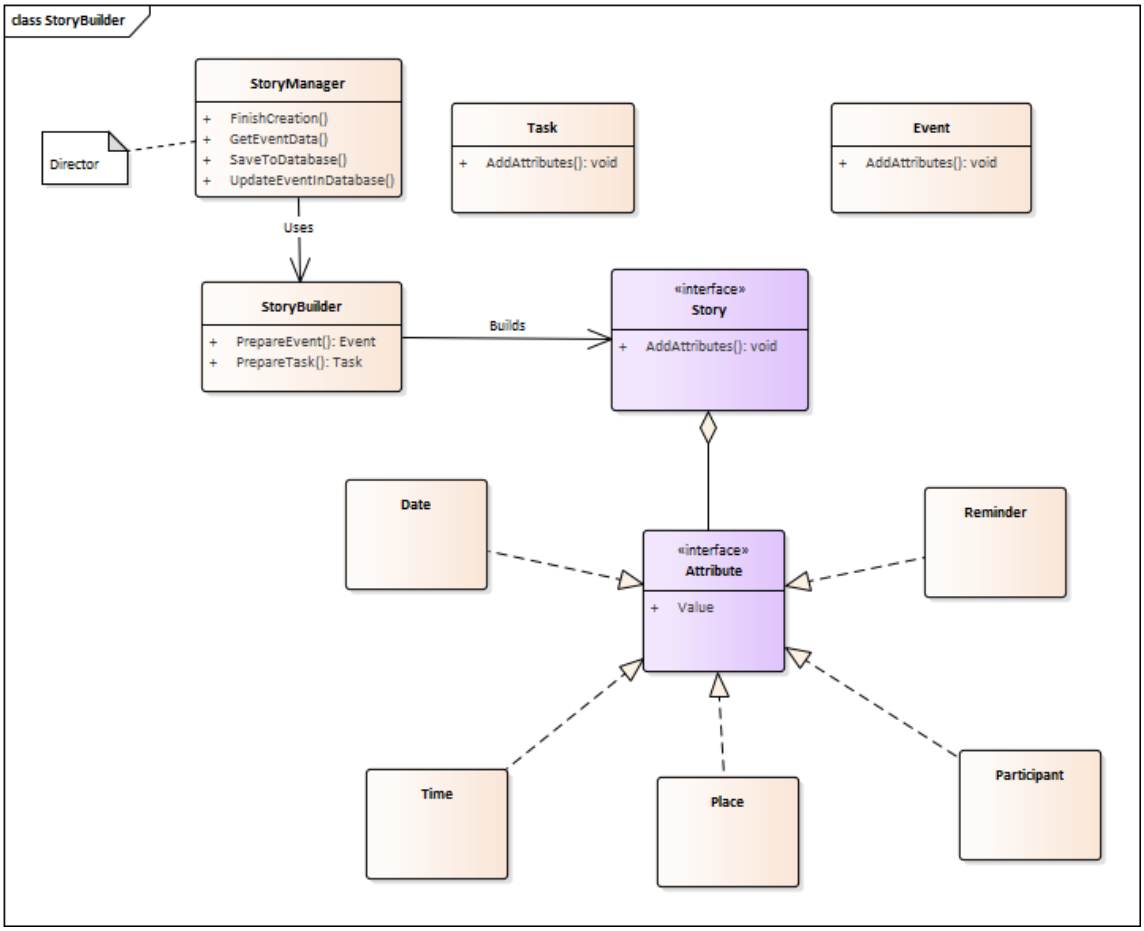

Figure 24: Class model of used design pattern Builder

```
"""
"""
import abc

class StoryManager:
    def __init__(self):
        self._builder = None

    def construct(self, builder):
        self._builder = builder
        self._builder._PrepareTask()
        # self._builder._PrepareEvent()


class StoryBuilder(ABC):
    def __init__(self):
        self.product = Story()

    @abc.abstractmethod
    def _build(self):
        pass

    # Build task out of available attributes
    def _PrepareTask(self):
        pass

    # Build event out of available attributes
    def _PrepareEvent(self):
        pass
```

Figure 25: Source code of used design pattern Builder

The Builder pattern separates the construction of a complex object from its representation so that the same construction process can create different representations. In our work, we use the design pattern builder to build a story. There are more options how the story can be – event or task.

7.8. Strategy

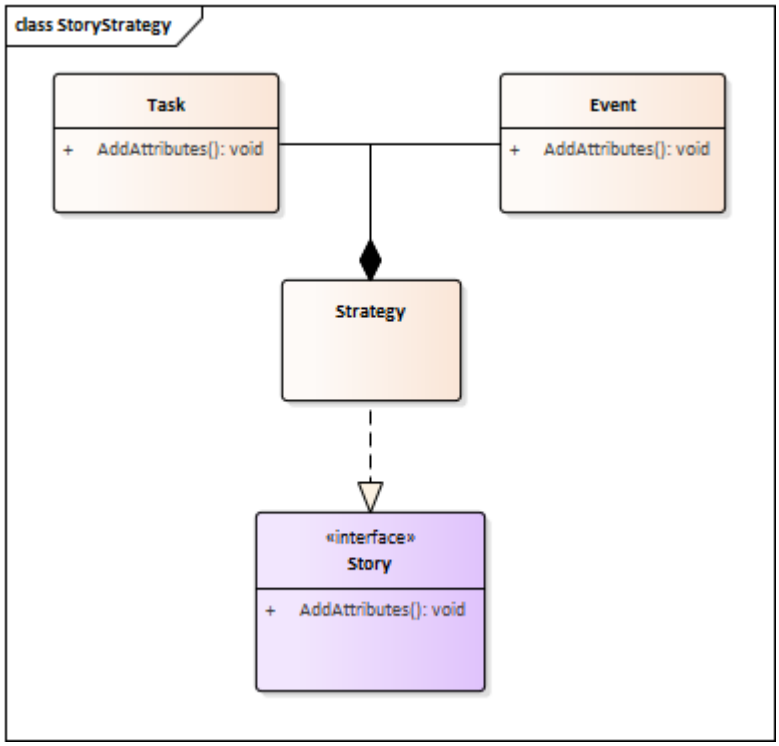

Figure 26: Class model of used design pattern Strategy

8. Distribution of work

Chapter	%
Introduction	Peter Berta – 50%
	Gabriela Hózová – 50%
Project Specification	Peter Berta – 50%
	Gabriela Hózová – 50%
Functional Requirements	Peter Berta – 50%
	Gabriela Hózová – 50%
Modules	Peter Berta – 50%
	Gabriela Hózová – 50%
Use Cases	Peter Berta – 50%
	Gabriela Hózová – 50%
Class Diagram	Peter Berta – 50%
	Gabriela Hózová – 50%
Design Patterns	Peter Berta – 50%
	Gabriela Hózová – 50%