MOD 4D UML

Software Requirements Specification

for

Orchestra

Version 1.0 approved

Prepared by Cédric Cligniez, Antoine Malinet and Siqi Lei

Concordia University

May 23rd, 2018

Revision History

Name	Date	Reason For Changes	Version
Sequence diagram	8 May 2018	creation of Sequence diagram	1.0
Scenarios	8 May 2018	creation of 18 Scenarios	1.0
Use case diagram	8 May 2018	creation of Use case diagram	1.0
Class diagram	8 May 2018	creation of Class diagram	1.0
Class diagram	16 May 2018	update of Class diagram, added classes “Team”, “User”, changed dependencies to adapt new classes.	1.1
Implementation	18 May 2018	implementation	1.1
Sequence diagram	22 May 2018	to adapt to function names	1.1
Scenarios	22 May 2018	update 18 Scenarios to adapte the command prompt	1.1

1. Introduction

1.1. Purpose

The project intends to build an internal-use software to build a numeric environment to facilitate the management of musicians, instruments, teams, events of an Orchestra. The software can also be extended in the future to enable customers to view events details, buy online tickets for events, and merchandises related to the orchestra.

This report discusses several aspects of the development and use, especially the progress made during the several iterations of development for the Orchestra software.

1.2. Intended Audience and Reading Suggestions

The software is designed for an orchestra manager. One should learn how to interact with a console.

1.3. Project Scope

Instead of building graphical interface, we decided to develop a console application. The application uses serializable objects to store informations, and allows users to interact with the software with a console, that is to say type a specific number to select a choice among a menu.

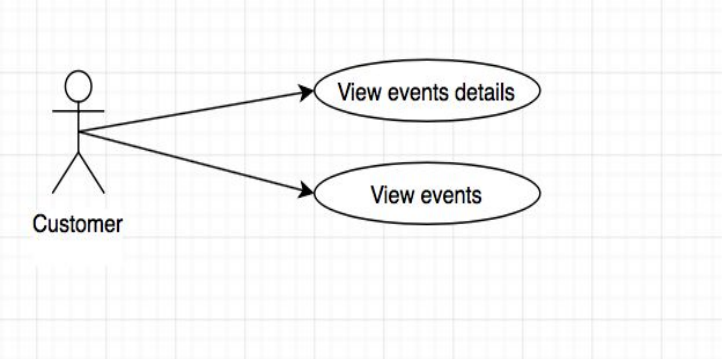
- ### Methodology

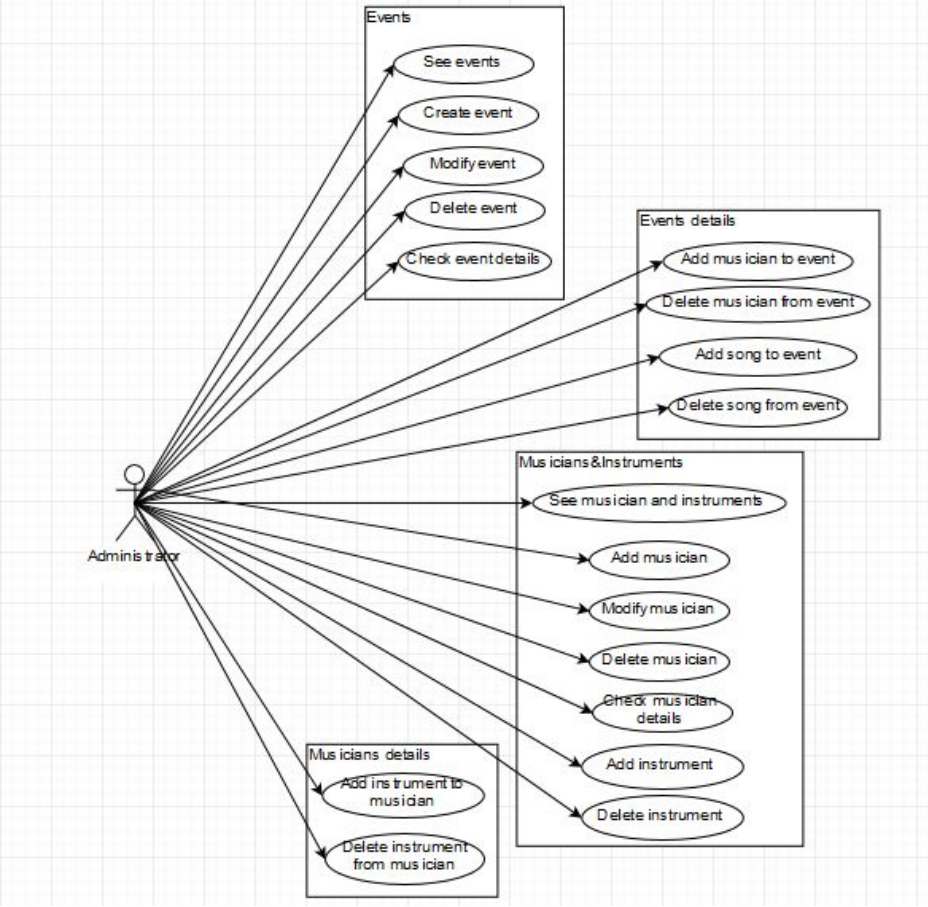
The agile methodologies we adopted in this project enable us to develop a tractable and plastic project in case of modifications in the requirements and design. By the time of the beginning of this report, we have already done 2 presentations orally and made the code running and functional, and had done 2 major versions of the code and some minors modifications, and successfully adjust the requirements following the time.

- ### Actors

In terms of actors, the initial list of actors established during the version 1.0 was judged sufficient to fully capture the range of users targeted by our project Orchestra. As such we have not brought any significant changes to our actors as we developed the version 1.1.

- **Use-case diagram**





1.4. References

http://draw.io
http://blog.csdn.net
http://tutorialspoint.com
https://www.oodesign.com

https://sourcemaking.com
http://www.runoob.com
Unnesessary inheritance: https://help.semmle.com/wiki/display/JAVA/Type+inheritance+depth
Software safety requirement instructions
:http://www.cs.nott.ac.uk/~pszcah/G53QAT/Report08/tdh06u-WebPage.html

Overall Description

1.5. Product Perspective

This is a new self-contained product, developed in order to propose an IT-based solution to handling the actors of an orchestra.

1.6. User Classes and Characteristics

Two types of people will use this product: administrators and customers. The first will have a lot more rights than the second, but only the second will be able to purchase tickets and merchandise. A login system would allow to identify the users.

1.7. Operating Environment

The application works on Windows 7, Mac OS Linux and further.

1.8. Assumptions and Dependencies

Although the operating environment of our project is any IDE that can run Java, we recommend the Eclipse or IntelliJ IDE, those are two user-friendly IDE easy to manipulate and follow up.

1.9. Budget

Since we don't have any experience in budget, here are the time spent estimation: about 12 hours for each team member, so about 36 hours for the whole group.

2. System Features

Things that we have successfully done :

1. Division of work items and check-in list to meet the course requirements for version 1.0 and 1.1
2. Features on check-list for version 1.0 and 1.1
3. The use cases, scenarios and sequence diagrams is reviewed and updated following the agile methodologies.
4. ASQ (After-Scenario Questionnaire) evaluations were done after the implementations.
5. Internal testing composed of unit-test and punctual manual tests.
6. External testing by oral presentation
7. Software Metrics and overall evaluation

Things that we would have liked to develop but have not implemented yet:

1. add conditions to modify objects to avoid duplicates;
2. Develop graphical interface instead of command line;
3. add customers log in.

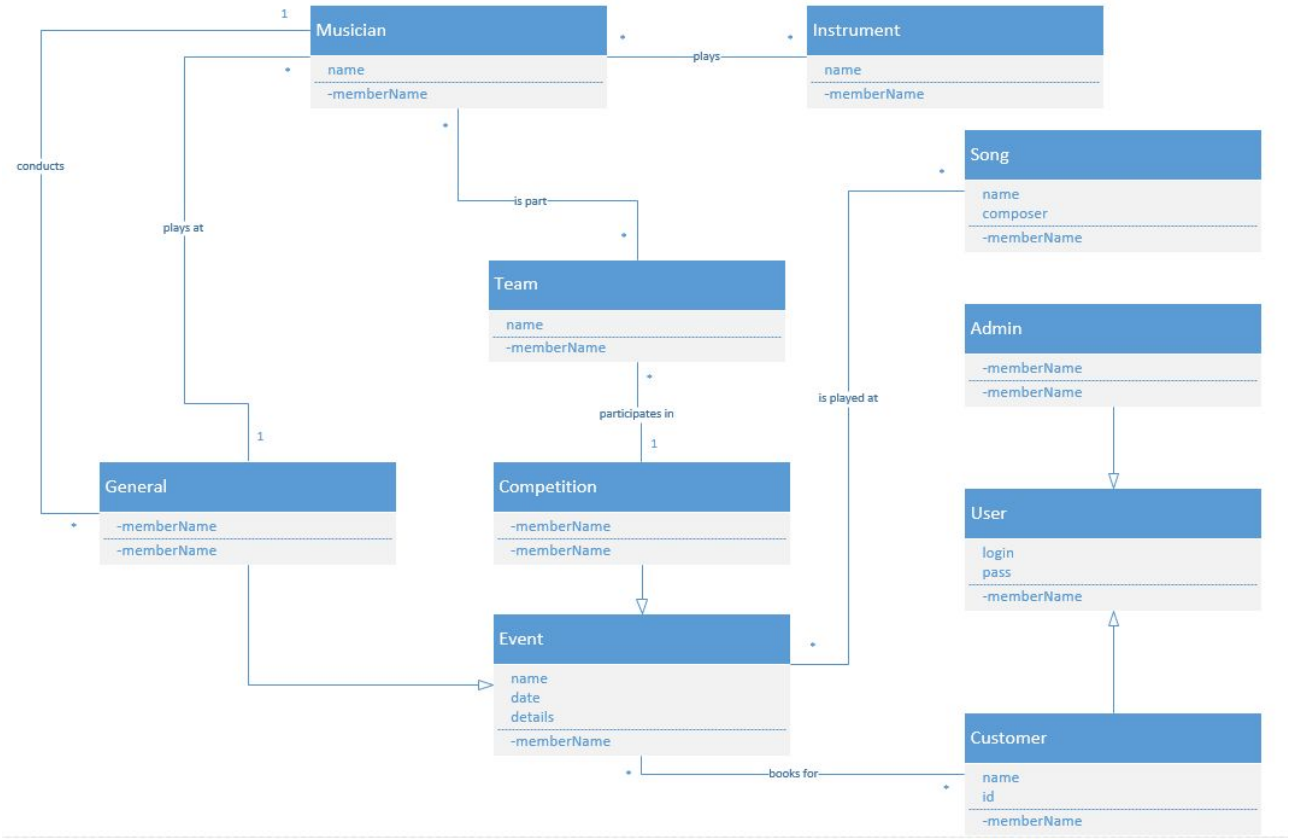
2.1. Project Features

2.1.1. Overall of the project

The product allows to save and navigate between different lists: musicians, instruments and events. As an administrator, you can view the details of each instance, add, delete, modify these instances, and link them to each other.

As a customer, you can book for an event after viewing them and its details (not implemented yet).

2.1.2. Project domain model



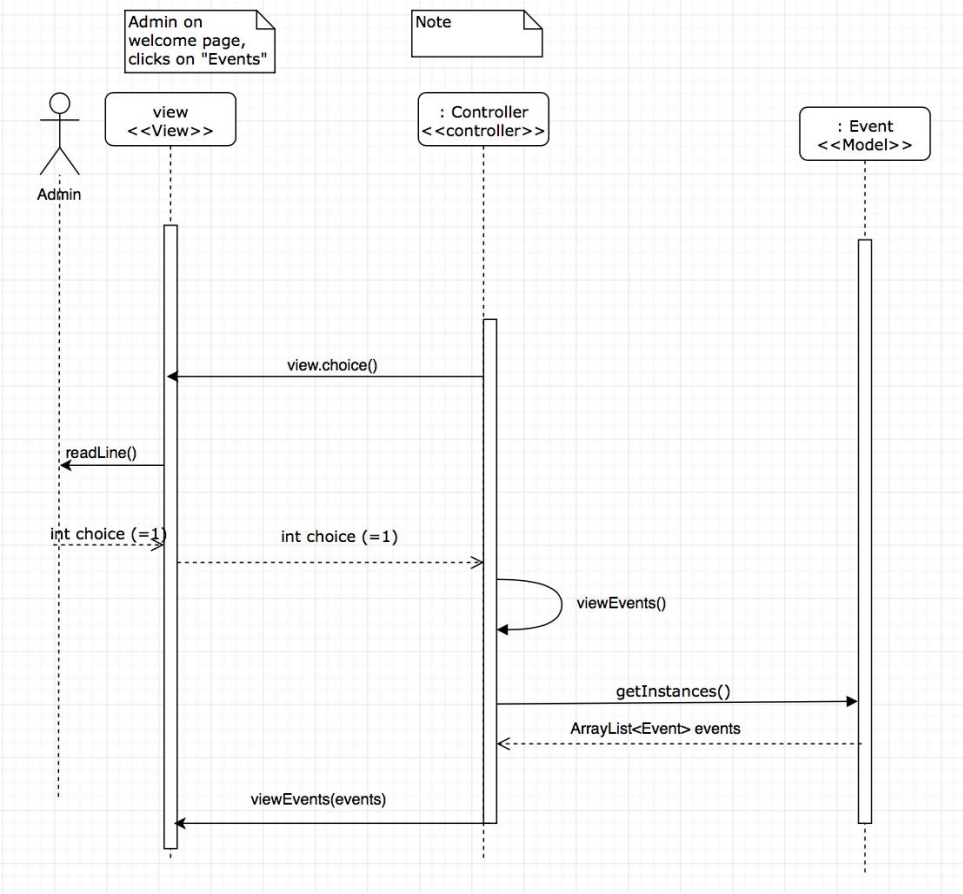
This class diagram does not represent the last version of the code. Actually it is a representation of how the code was before we implemented the **composite pattern**. It allowed to add teams and musicians to a team. It might not be the best choice to implement composite pattern in our project, however, since it is feasible and allows us to add a pattern to satisfy the course instructions, we still approve the idea of adding it for educational purpose .

2.1.3. Use Cases

Use case 1

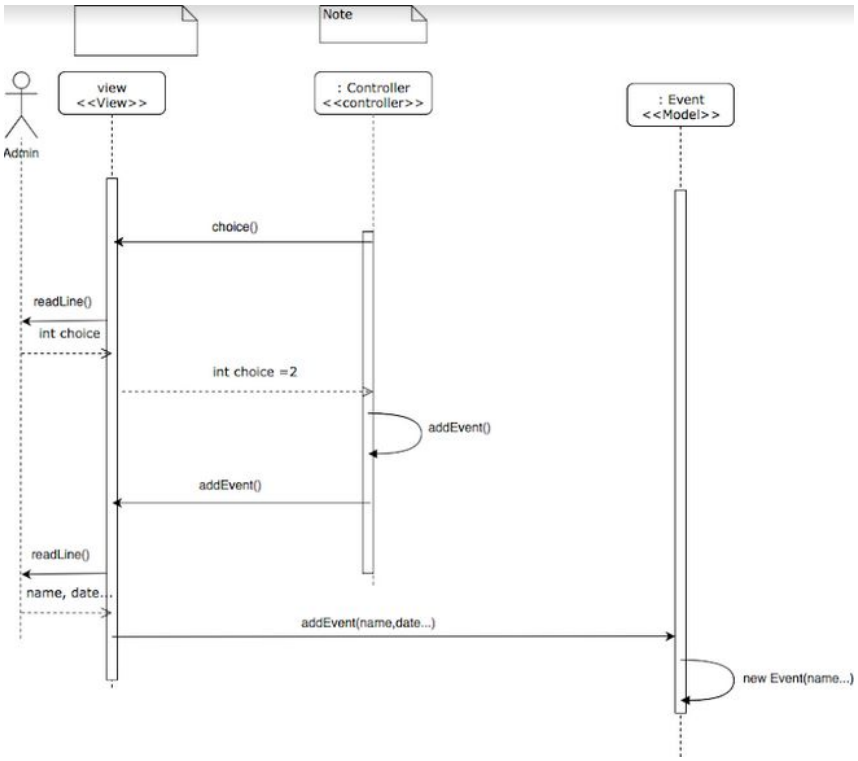
The administrator is on the event menu and chooses number 1 which displays the list of events.

Number	1	
Name	See Events	
Summary	Displays the Events	
Preconditions	Being on the Events page	
Postconditions	Being on the Events page	
Primary actor(s)	User	
Secondary actor(s)	Events Class	
Main Scenario	**Step**	**Action**
	1	System displays the Events



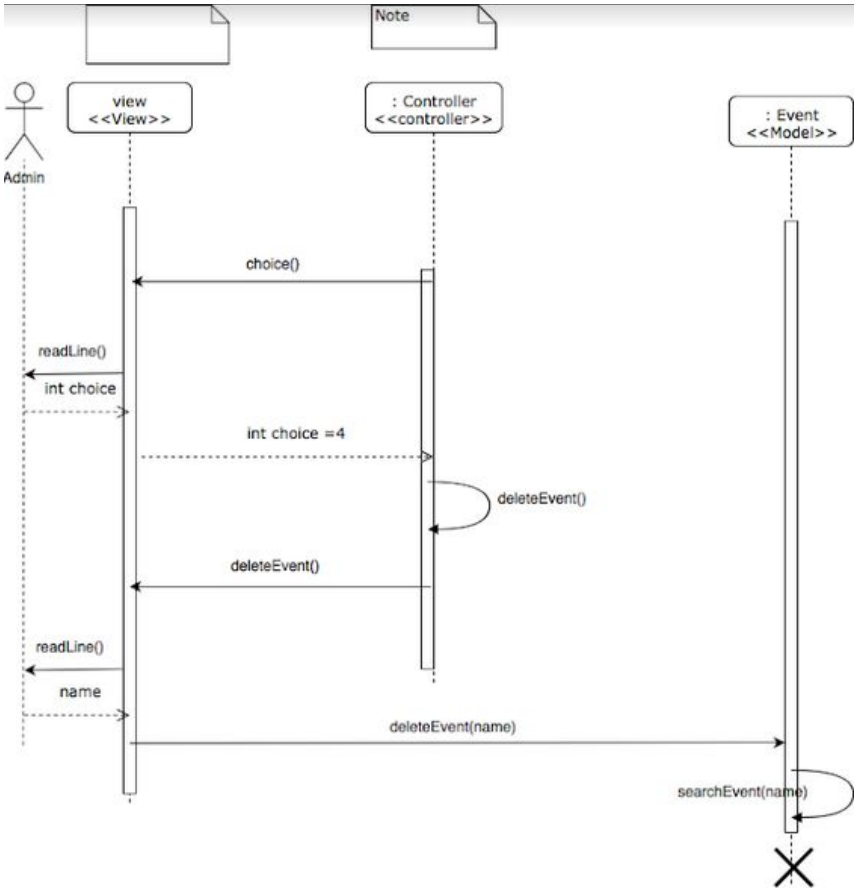
Use case 2

Number	2	
Name	Add Event	
Summary	The admin enters the necessary information to add an event	
Preconditions	Being on the events page, having logged in as admin	
Postconditions	An event has been added	
Primary actor(s)	Admin	
Secondary actor(s)	Events Class	
Main Scenario	**Step**	**Action**
	1	System displays form
	2	User enters the name of the event
	3	User enters the date of the event
	4	User enters the type of the event
	5	User enters the details of the event
	6	System saves the event
	7	System displays event page
Extensions	**Step**	**Branching Action**
	5a	User cancels
	5b	System displays event page
	6a	If the form has not been correctly filled, display error



Use case 3

Number	3	
Name	Delete Event	
Summary	Remove an event from the events list	
Preconditions	Being on the events page, having logged in as admin	
Postconditions	The event has been deleted	
Primary actor(s)	Admin	
Secondary actor(s)	Events Class	
Main Scenario	**Step**	**Action**
	1	System displays confirmation window
	2	User presses OK button
	3	System deletes the event
	4	System displays event page
Extensions	**Step**	**Branching Action**
	2a	User presses Cancel button
	2b	System displays event page

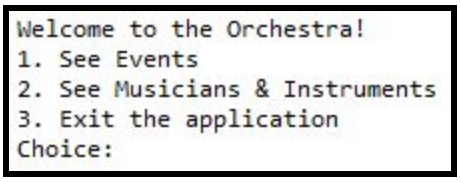


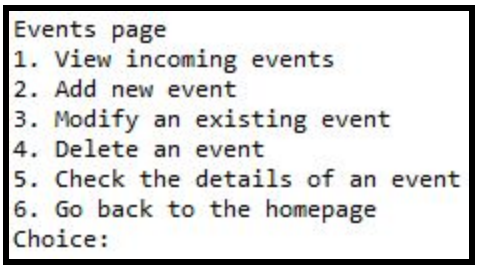
3. External Interface Requirements

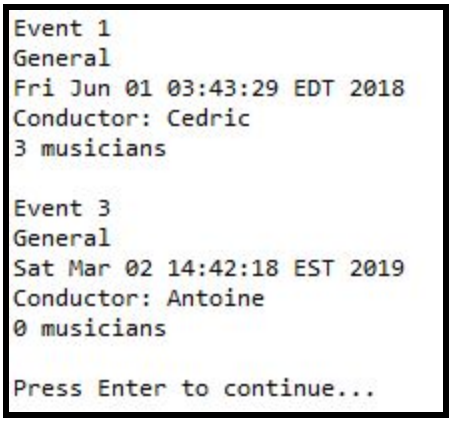
3.1. User Interface

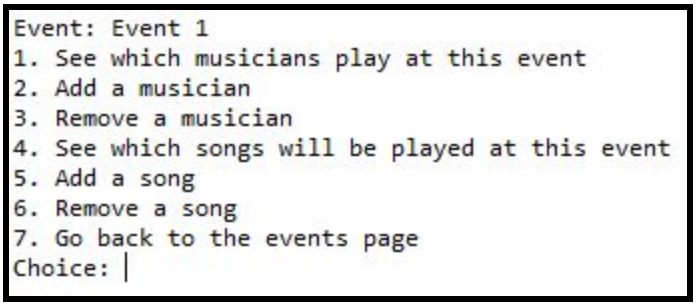
A mock-up was created before because we were supposed to code a GUI. You can find it in the first iteration report, and this work would have been used if we had designed a GUI.

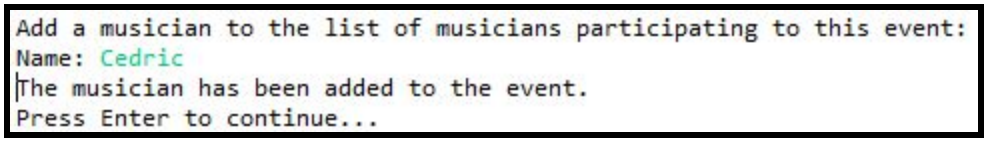
Since we chose to implement a console interface, menus are displayed with choices to make, and sometimes pieces of information to enter. This is like a standard and very simple command line interface. Here are key screenshots:

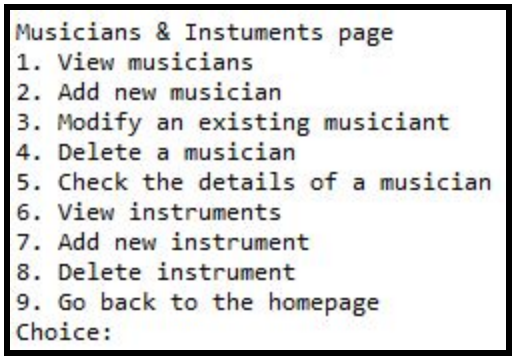












State diagram for the pages:

We used a system of states to tell the system which page the user was on, so that it can display the corresponding information and ask for the correct inputs. Here is a diagram showing how these states work:

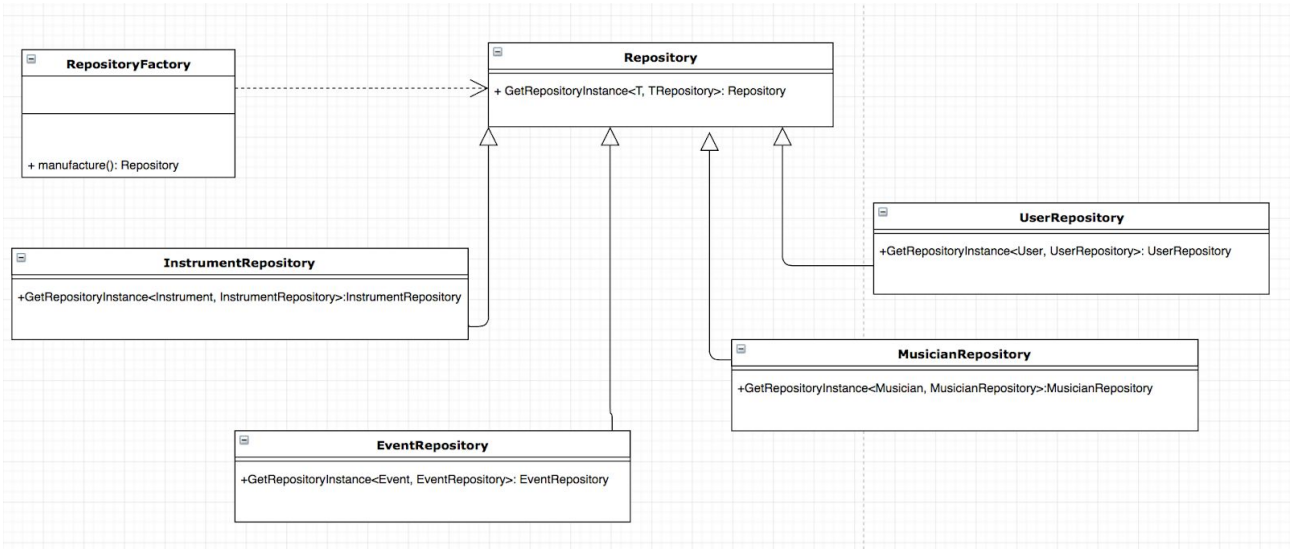
4. Other Nonfunctional Requirements

4.1. Performance Requirements

The performance requirements consist of the design of 3 patterns for our project. A software design pattern is a general, reusable solution to a commonly occurring problem within a given context in software design. For the project overall performance and coding quality, we suggest here factory pattern, composite pattern, singleton pattern and façade pattern that we could implement or have implemented in our project:

Factory pattern

Repository Factory produces repositories intended to store data, which is inherited by several different repositories, each one has similar but slightly different from the other. As you can see from the following diagram, Repository produces 4 types of different repositories for Event, Instrument, User and Musician. This pattern hasn't been coded, we used serialization instead.



Composite pattern

As explained in the 2.1.2, the composite pattern could be feasible and give us a good occasion to implement a tree structure. Even it's not the best idea to implement it, we still consider it for pedagogical purpose.

Singleton pattern

This pattern was used for the view, it allows only one running view at a time.

Façade pattern

This pattern should not be implemented in our project, though it could be of a good use for one single instance functions, such as add or delete that are common, but not for “add musician to an event” for example, which involves two classes.

We still did not code this pattern because we saved our created classes in static variables in each class, and it is impossible to overload static functions. (Especially from an interface, which can only recently contain static functions.)

4.2. Safety Requirements

Software System Safety optimizes system safety in the design, development, use, and maintenance of software systems and their integration with safety critical hardware systems in an operational environment. Not all of these can be rectified by the programmers, so here we’d like to list three types of majors safety requirements: software logic errors, software support errors, hardware failures.

- **software logic errors**

  We have considered thoroughly the exceptions provocated by user manipulations errors, which include wrong input value and types, absence of those inputs that may occurs during the project running process.

- **software support errors**

  The maintenance of the project should be considered once the software is commercialized and followed by programmers in case of technical issues.

- **hardware failures**

  The operating system constructions and running environment should be followed as mentioned in 1.7 and 1.8.

4.3. Security Requirements

Users passwords are hashed using SHA-256 and a checking function has been implemented. However, the login case was not implemented yet.

Conclusion

This project consists of the development of an internal used software for an Orchestra, and can be served as prototype for extension of more general Orchestra commercial need in the future, however, to achieve that we still need more investment on research, engineering and market investigation to understand more generic needs.

Since the project is intended for the course MOD4B UML and the course duration is short, the guidelines, instructions and our ambition for this project required intensive and progressive research and development. To achieve the project successfully, the key elements are team work, management planning, adherence to guidelines and clear targets. For setting clear goals and objectives, we tried to follow the principles named as above: Specific, Measurable, Attainable, Relevant, and Punctual. All team members have a clear understanding of these mentioned criteria and the necessary components of the software development lifecycle.

By the time we submitted the projects, we have already successful done :

8. Division of work items and check-in list for version 1.0 and 1.1
9. Features on check-list for version 1.0 and 1.1
10. The use cases, scenarios and sequence diagrams is reviewed and updated following the agile methodologies.
11. ASQ (After-Scenario Questionnaire) evaluations were done after the implementations.
12. Internal testing composed of unit-test and punctual manual tests.
13. External testing by oral presentation
14. Software Metrics and overall evaluation

As we suggested at the beginning of the report, the **agile methodologies** is adopted in order to develop a tractable project in order to facing of modifications in the analysis, requirements and design. The team collaborated through regular online communication and daily weekly meetings to stay informed of the progress and observe objectives and deadlines.

The project also provided us a very good opportunity to review the MVC model, especially the design and implementation through the Java language adaptation since it's very important for engineer to learn adaptation of language specificities.

The development process evolved through various project phases: Requirements, Analysis, Design, Implementation and Test, report and presentations. Every team member had the opportunity to participate in all the above phases and have very good understanding of the overall picture.

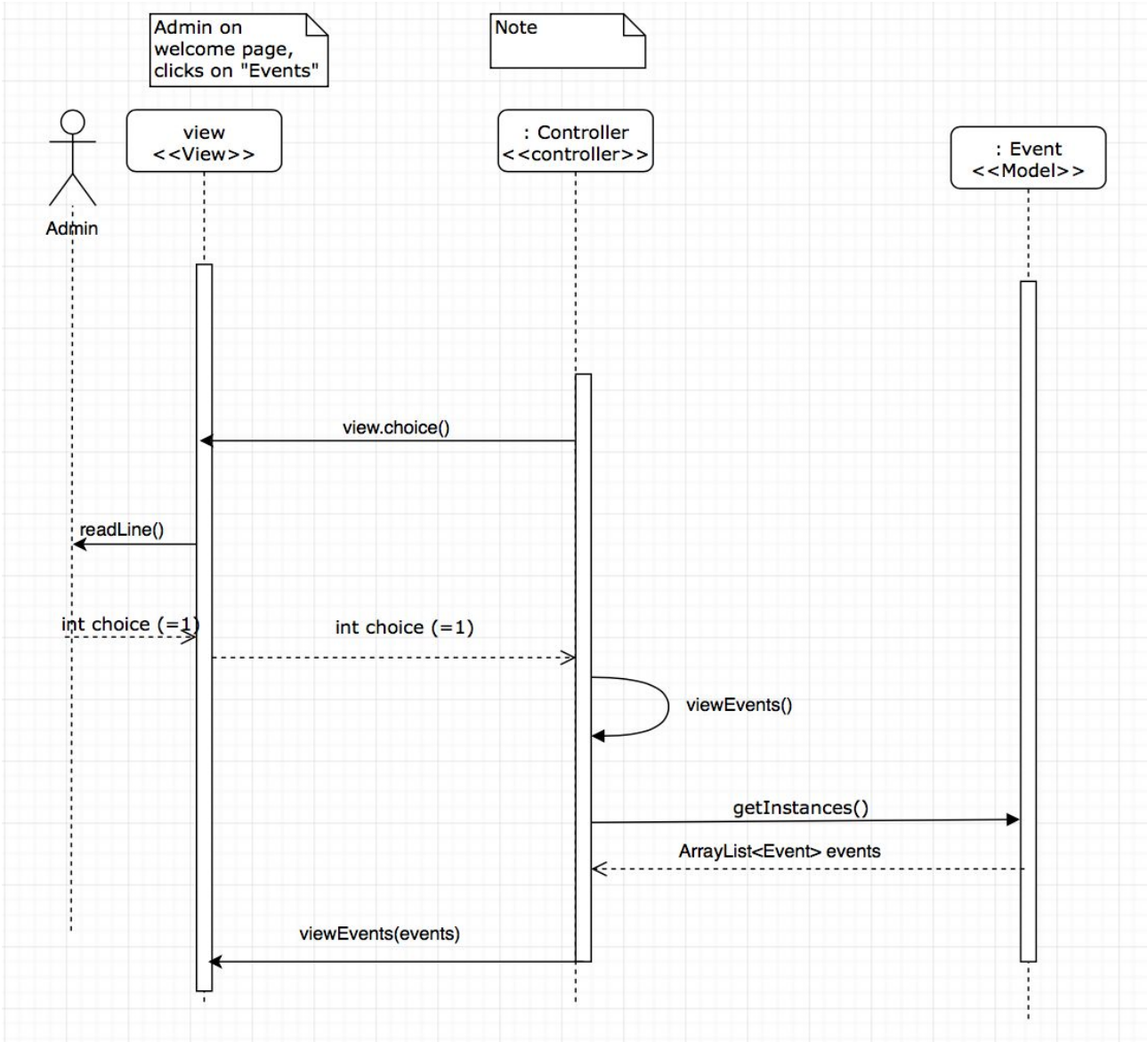
In regards to the team work, to face several technical and communication problems, we had learn the development of team collaboration methods and skills, and finally an outcome that meets the majority of our original targets.

Appendix A: Analysis Models

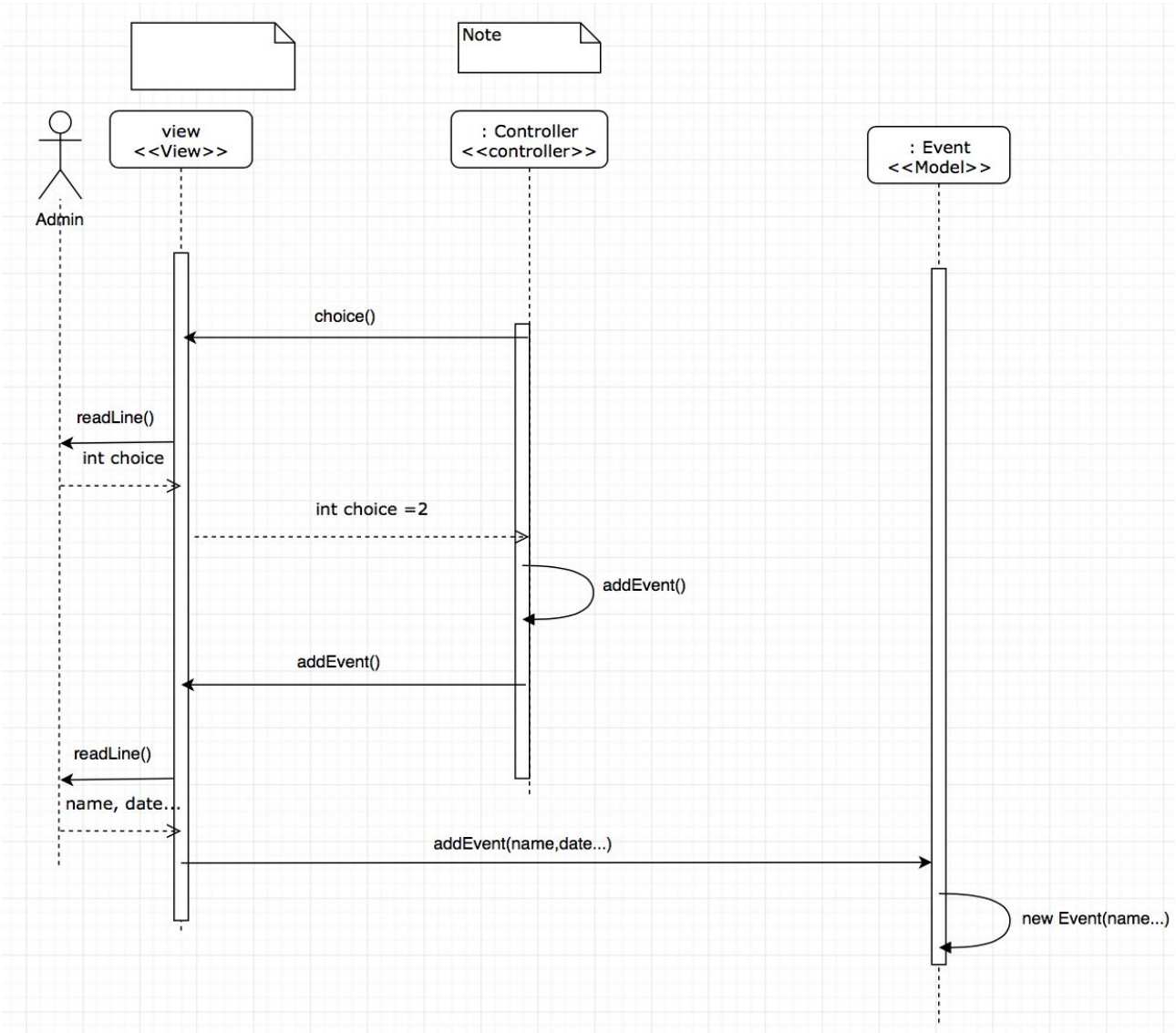
In this section, we will present you the **package diagram** and the most significant **sequence diagram** of our project.

- *Sequence diagrams :*

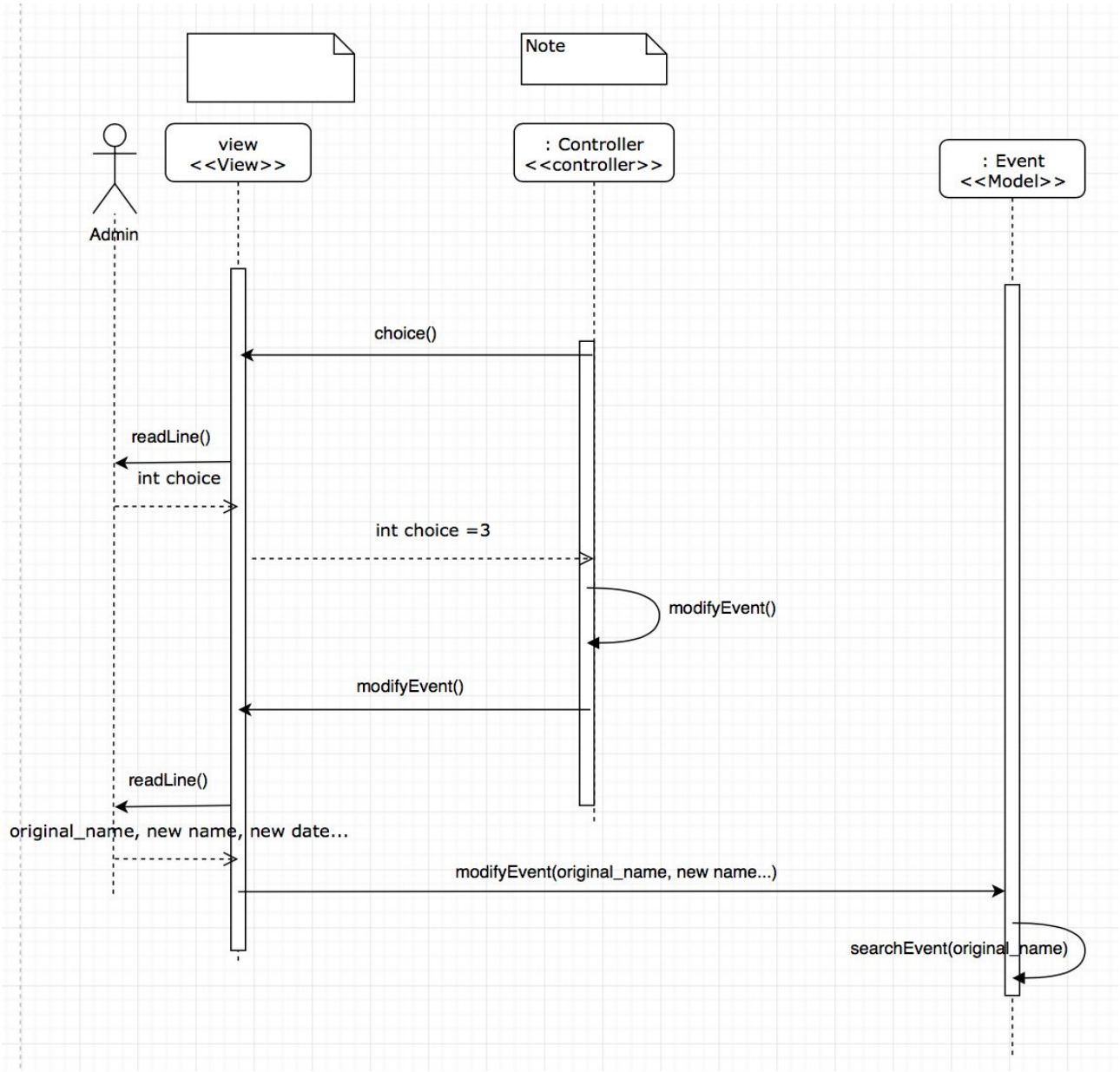
1. *ViewEvent*



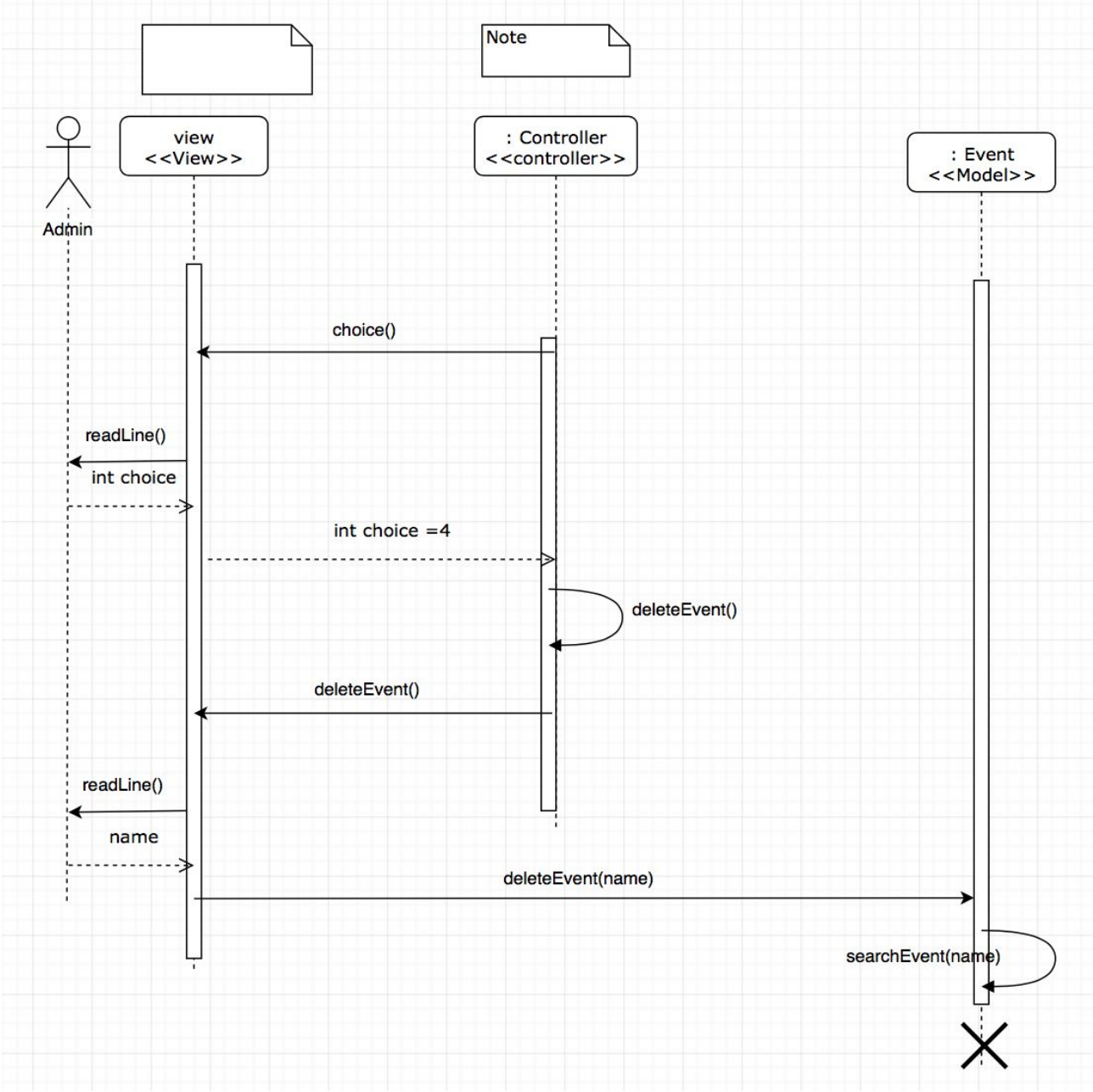
2. *AddEvent*


Note
Admin
view
<<View>>
: Controller
<<controller>>
: Event
<<Model>>
choice()
readLine()
int choice
int choice =2
addEvent()
addEvent()
readLine()
name, date...
addEvent(name,date...)
new Event(name...)

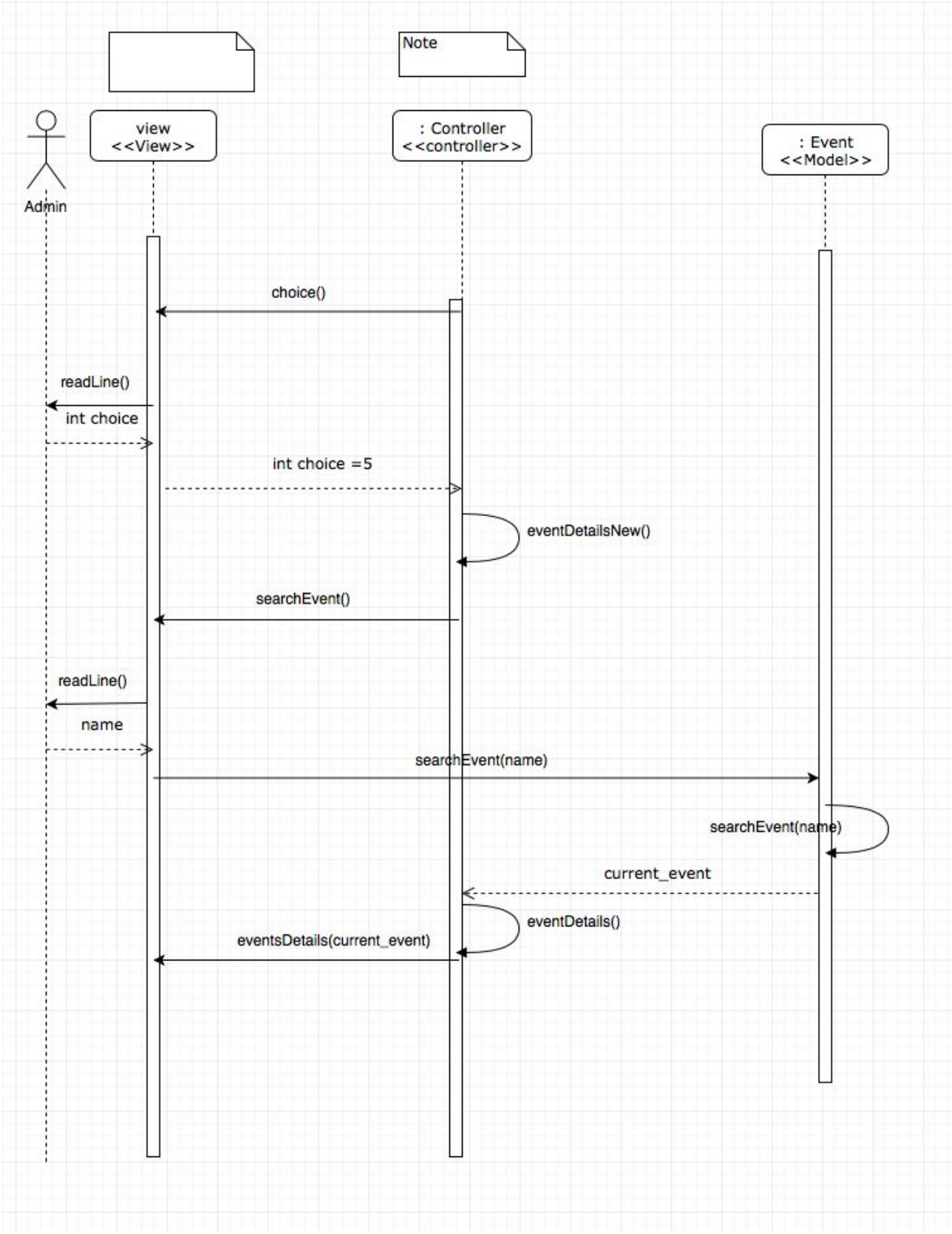
3. *ModifyEvent*



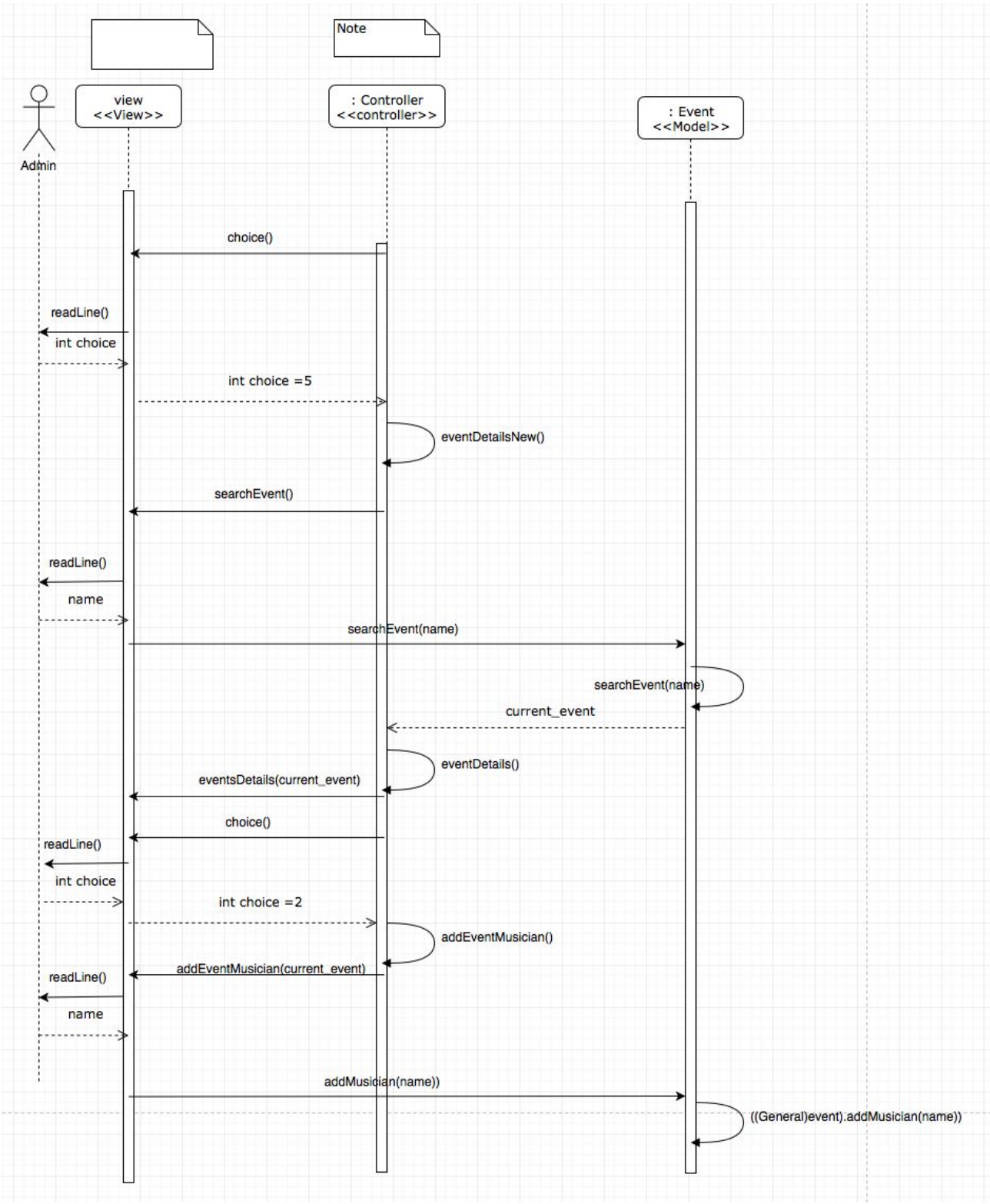
4. DeleteEvent



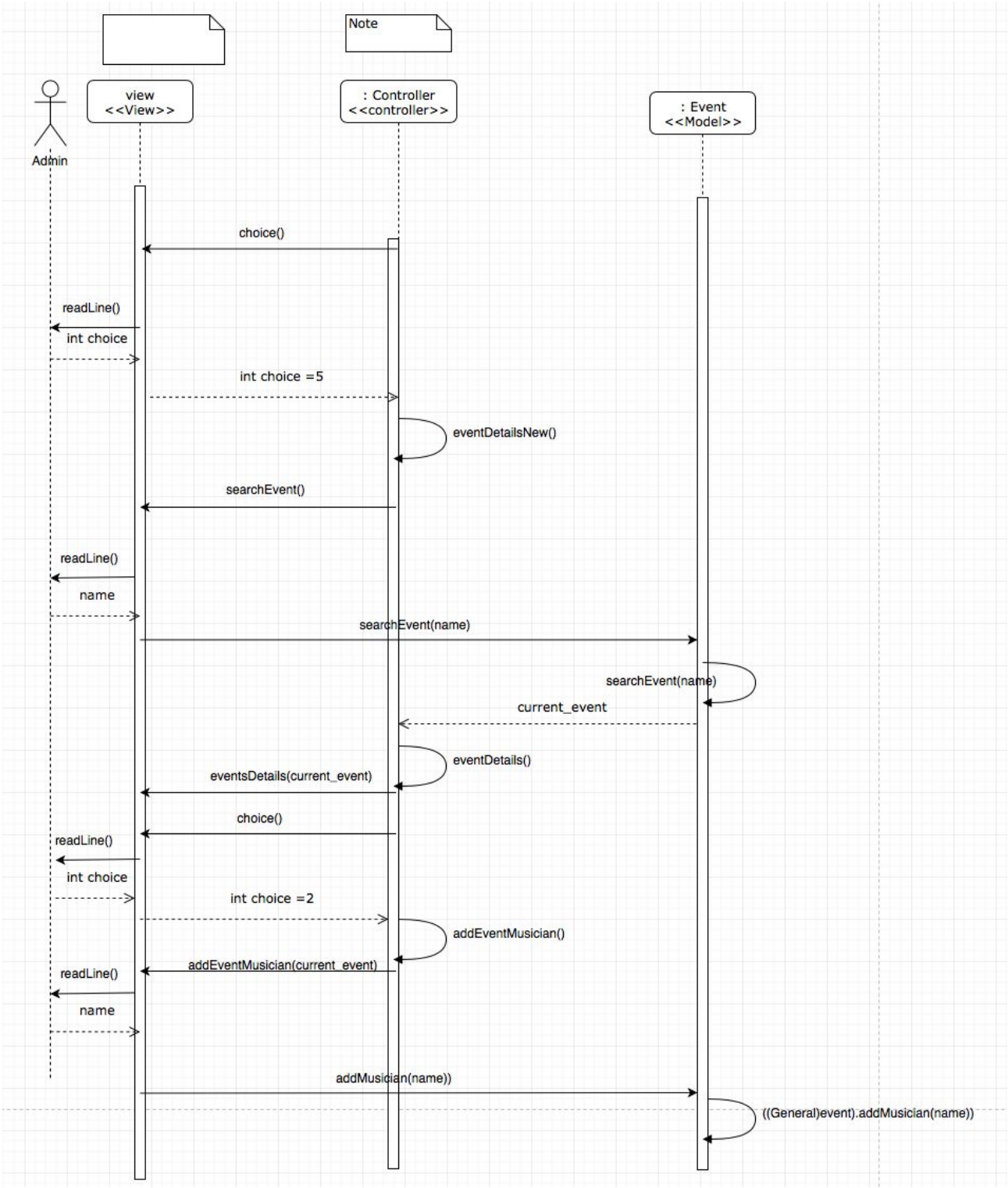
5. *CheckEventDetails*



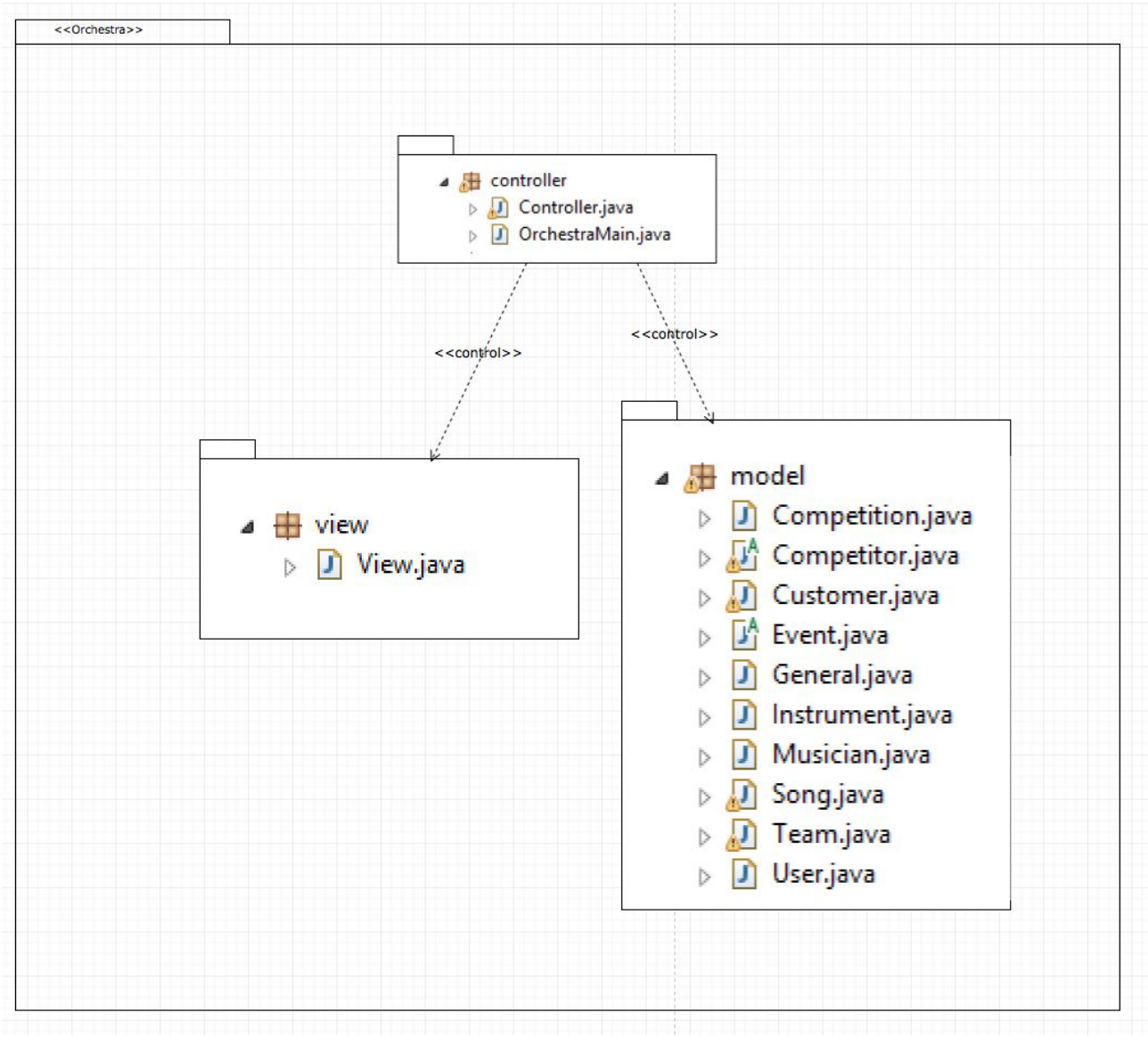
6. Add Musician To Event



7. *Delete Musician From Event*



❖ *Package diagram :*



Appendix B: Issues List

1. the login case was not implemented yet so the customer and admin now have access to all of the functionality.

Appendix C: Software Metrics

The software metrics is a standard of measure of a degree to which a software system or process possesses some property, it could be interpreted as the measurements to display

(represent) performance of software. In this appendix, we cover the CBO (Coupling between classes), the RFC(response for class), DIT(Depth of Inheritance Tree) and NOC(Number of Children).

Class	CBO	RFC	DIT	NOC
Musician	1	14	0	0
Instrument	0	8	0	0
Event	2	15	0	2
Song	0	5	0	0
Team	1	6	0	0
Competition	3	20	1	0
General	4	24	1	0
Customer	1	12	1	0
User	0	7	0	2
Admin	0	8	1	0

As we can see here, we succeeded to avoid the content coupling, common coupling, external coupling and control coupling, there are only stamp coupling and data coupling in our project. That's why we can classify our software as low-coupling.

Meanwhile, as each class has very distinct and isolated responsibility, using single inheritance (without unnecessarily deep class hierarchy), the system can be classified as “high-coercive”.