

ECSE 321 - Intro to Software Engineering Design Specification Document - Deliverable 2

Harley Wiltzer
Camilo Garcia La Rotta
Jake Shnaidman
Robert Attard
Matthew Lesko

February 27, 2017

Contents

I	Architecture of the Proposed Solution	2
0.1	Description	3
0.2	Rationale	3
0.3	Block Diagram	3
II	Description of the Detailed Design	5
0.4	Description	6
0.4.1	Detailed Domain Model	6
0.4.2	Controller Classes	7
0.4.3	View Classes	7
0.5	Rationale	8
0.6	Detailed Design Diagram	9
0.7	Java and Android Controller Class Diagram (Desktop and Mobile)	10
0.8	PHP Controller Class Diagram (Web)	10
0.9	Java View Class Diagram (Desktop)	12
0.10	PHP View Class Diagram (Web)	13
0.11	Android View Class Diagram (Android)	13
III	Implementation of Prototypes	14
0.12	Desktop Application	15
0.12.1	Desktop Publish Job Posting Use Case	15
0.12.2	Desktop Apply to Job Use Case	16
0.13	Web Application	16
0.13.1	Web Publish Job Posting Use Case	17
0.14	Mobile Application	18
0.14.1	Mobile Apply to Job Posting Sequence Diagram	19
IV	Work Plan From Past and Future Iterations	20
0.14.2	Iteration 1	21
0.14.3	Iteration 2	21

Part I

Architecture of the Proposed Solution

0.1 Description

The software architecture comprises of two different patterns: a Model/View/Controller pattern and a Layered Architecture pattern. An "Authentication and Authorization" layer is on top of the MVC layer. Once the user is authenticated and authorized, they have access to the MVC layer. The MVC system contains three components which interact with each other:

- Controller
- View
- Model

The Model component manages the system data and associated operations on that data; it encapsulates all the entities that are part of the model (can be seen in the model class diagram). The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (key presses, mouse clicks, etc.) and passes these interactions to the View and the Model.

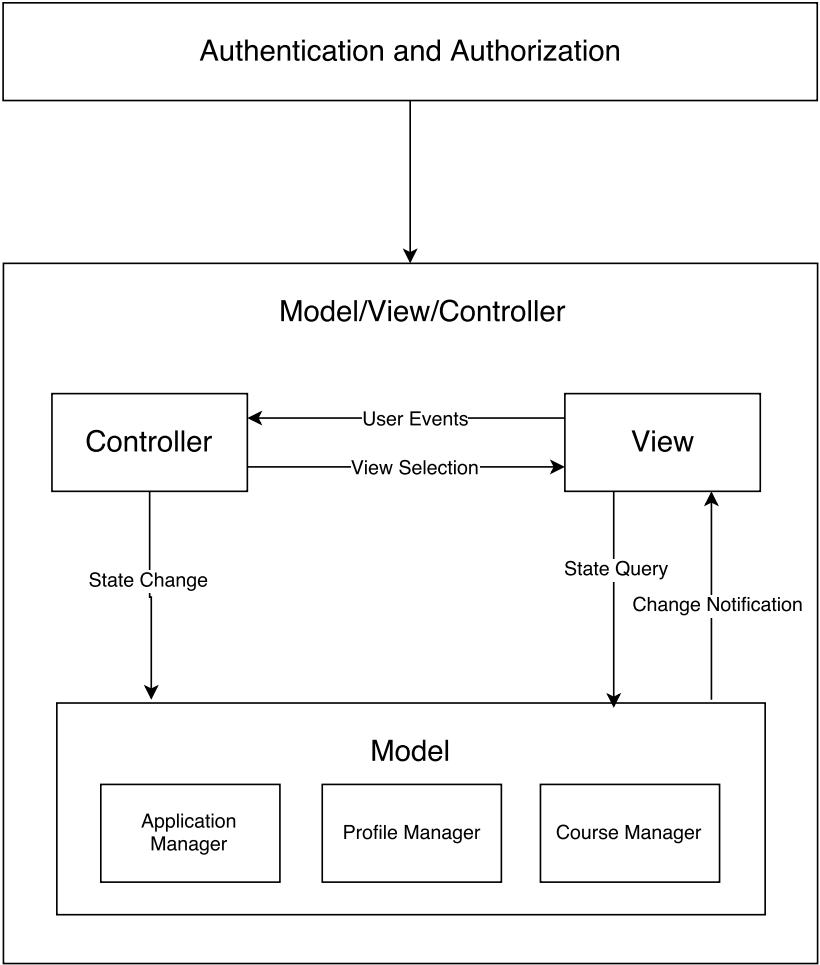
0.2 Rationale

The MVC pattern was chosen because this allows the components to be changed independently. For example, adding a new view or changing an existing view can be done without any changes to the underlying data in the model. It allows the data to change independently of its representation and vice versa. Moreover, it supports presentation of the same data in different ways with changes made in one representation shown in all of them.

Furthermore, the Model-View-Controller pattern makes multiplatform development very convenient. Using this paradigm, equivalent code may be generated for the model component on all platforms, as the model is isolated from the other components. Then, the views are isolated from the controllers so that modifications to the view classes do not affect code for other platforms that utilize the controller classes.

The Layered Architecture pattern was used because the user would need to first authenticate him/herself and then receive authorization in order to interact with the sublayer.

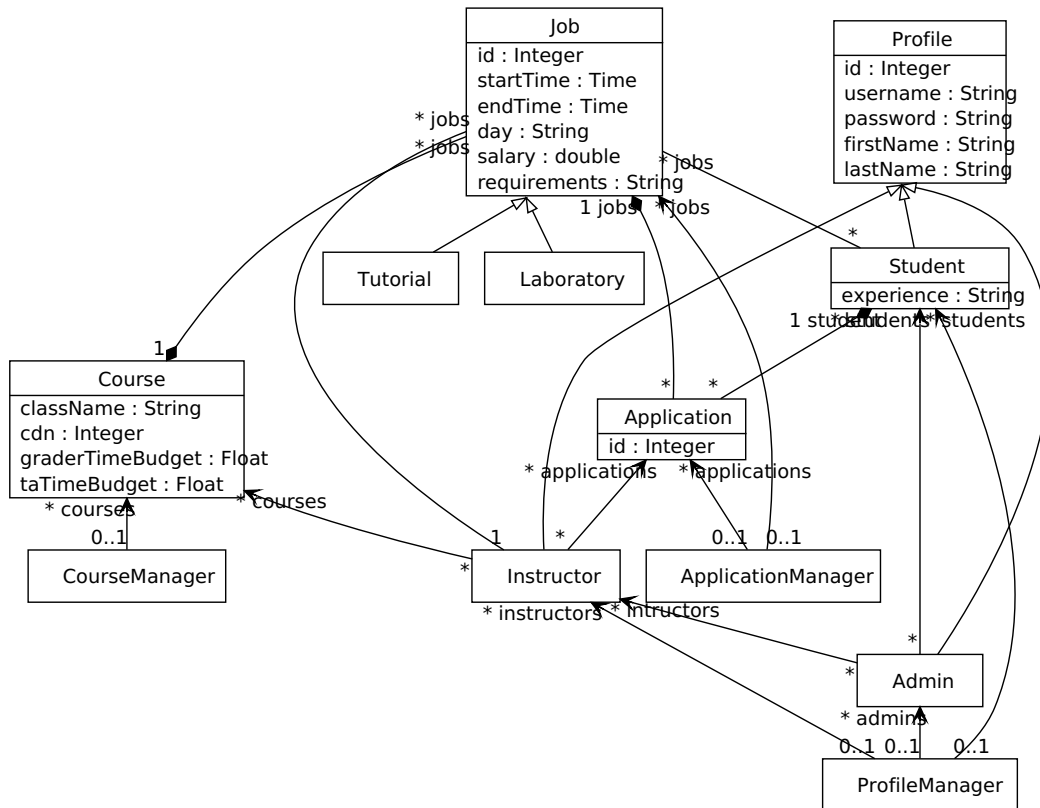
0.3 Block Diagram



Part II

Description of the Detailed Design

The Domain Model changed significantly since the last Deliverable, and the new domain model may be found below.



0.4 Description

0.4.1 Detailed Domain Model

The Detail Design Diagram consists of the following entities: ApplicationManager, ProfileManager, CourseManager, Application, Profile, Course, Job, Instructor, Admin, Student, Laboratory, and Tutorial. It consists of a Controller, called Controller, a Boundary, called View, and a Persistence, called Persistence XStream. The Controller uses the entities ApplicationManager, ProfileManager, and CourseManager to save, edit, and modify data within the model, which are then saved within a persistence layer. The functionalities of the three "Manager" classes are listed below.

- The `ApplicationManager` is in charge of `Applications`, which represent the job application created and submitted by the student for a job. Furthermore, the `ApplicationManager` is also in charge of managing `Job` data. It is associated with `Application`, `Job`, and `ProfileManager`.
- `ProfileManager` creates and manages `Admin`, `Instructor`, and `Student` entities, all of which inherit from the `Profile` class.
- `CourseManager` creates and manages `Course` entities.

0.4.2 Controller Classes

In total there will be three controller classes in the Controller Packages with an additional class for input exceptions or input validation. Each controller class has an associated Manager class, and is in charge of utilizing the manager class safely so that appropriate data is guaranteed to be entered into the persistence layer.

The Controller classes serve the purpose of isolating the model from the input, keeping in mind the philosophies of the Model-View-Controller paradigm.

0.4.3 View Classes

Finally, there are multiple View classes, dependent on the application platform, that act as boundary classes. These classes are in charge of gathering user input in a user-friendly manner. The Web and Mobile applications have four and two Views, respectively, and the Desktop application has five Views.

The View classes are listed by platform below:

Current Desktop Views

1. **MenuView**: Allows user to select which functionality to perform
2. **RegisterView**: Allows user to enter profile information to create a **Student**, **Instructor**, or **Administrator**
3. **CreateCourseView**: Allows user to enter data to create a **Course** given a specific **Instructor**
4. **PublishJobView**: Allows user to enter information required for creating a **Job**
5. **ApplicationView**: Allows user to create an **Application**

Current Mobile Views

1. **MainActivity**: Allows user to select which functionality to perform (analogous to **MenuView**)
2. **ApplyToJob**: Allows user to create an **Application** (analogous to **ApplicationView**)

Current Web Views

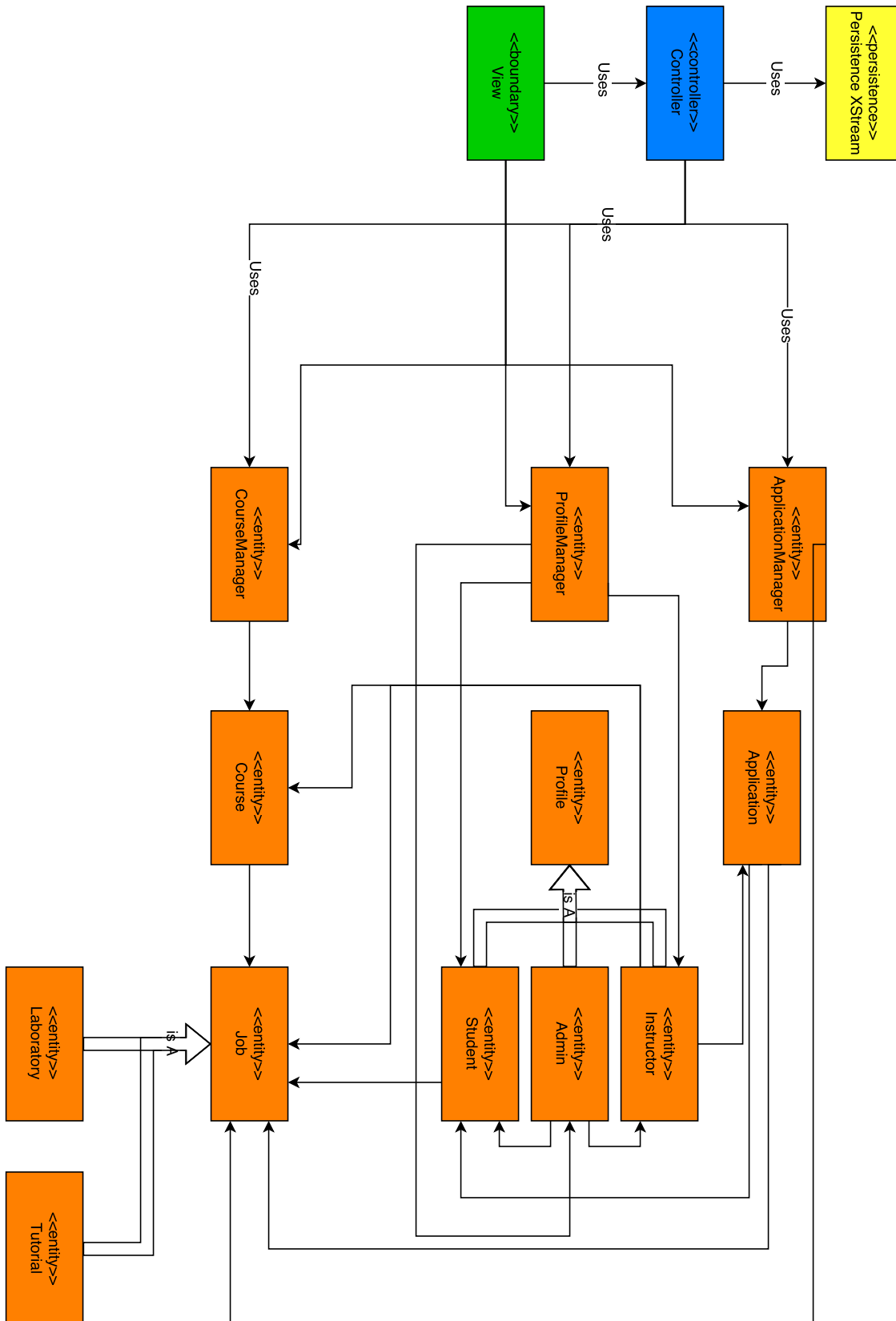
1. **index**: Allows user to select which functionality to perform (analogous to **MenuView**)
2. **createCourse**: Allows user to enter data to create a **Course** (analogous to **CreateCourseView**)
3. **createInstructor**: Allows user to enter data to create an **Instructor** (analogous to **RegisterView**). This View is temporary and will only be used for this Deliverable, as it does not satisfy the requirements of the project.
4. **job**: Allows user to submit data to publish a **Job** (analogous to **PublishJobView**)

0.5 Rationale

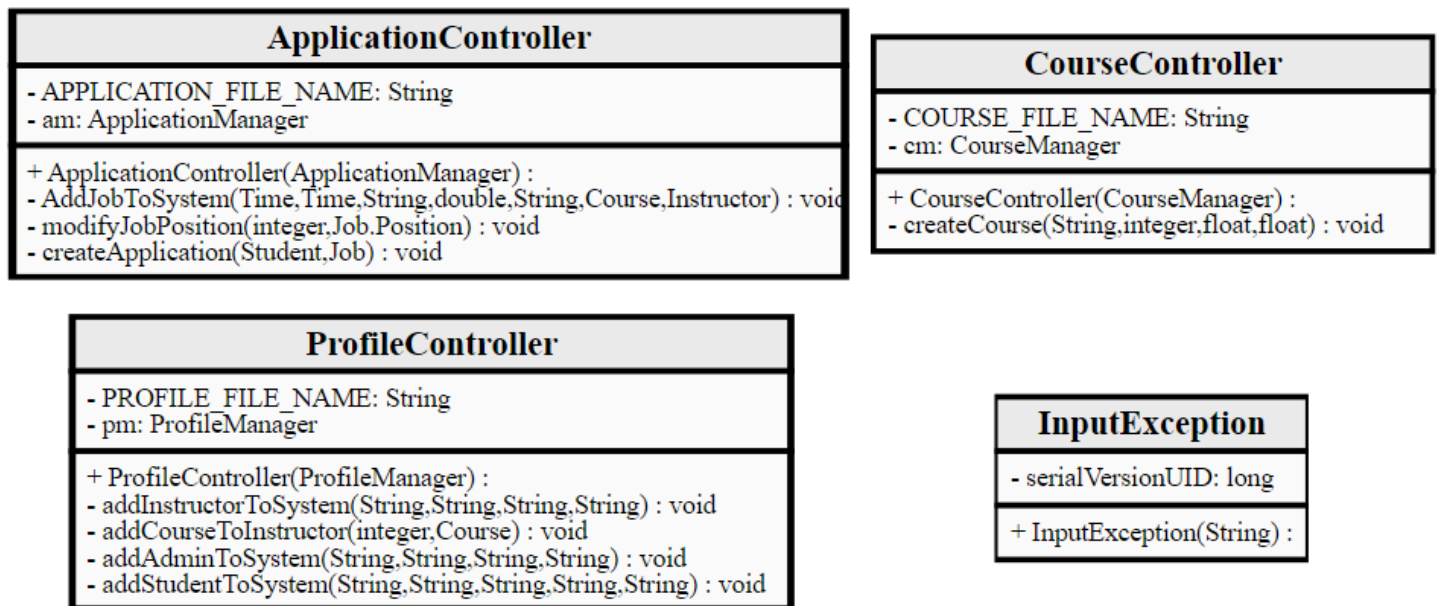
The program was designed this way so as to follow the principles of the Model-View-Controller design pattern. As discussed previously, this allowed for the effective isolation of the user input and the domain model. The Controller classes, along with the `PersistenceXStream` class, allow for persistence to be carried out in a safe manner. The Controller classes are designed in such a way that by using them to create their associated objects, the created objects will automatically be stored within the persistence layer assuming they were created successfully. To avoid storing faulty objects, the `InputException`, as well as several validation classes in PHP, were created. As such, the Controller classes throw exceptions when unworthy data is passed. The ultimate benefit to using this design is that the persistence is taken care of at the level of the controller classes, so the programmer needn't worry about it when designing the Views. With this foundation, the model may be altered reliably.

By perusing the list of Views, it is clear that the mobile and web Views are very similar to the desktop Views in terms of their purposes. This was done to create consistent interfaces across the platforms. The desktop has many views because it is meant to be used by Administrators, who can implement the functionalities possible on all other platforms.

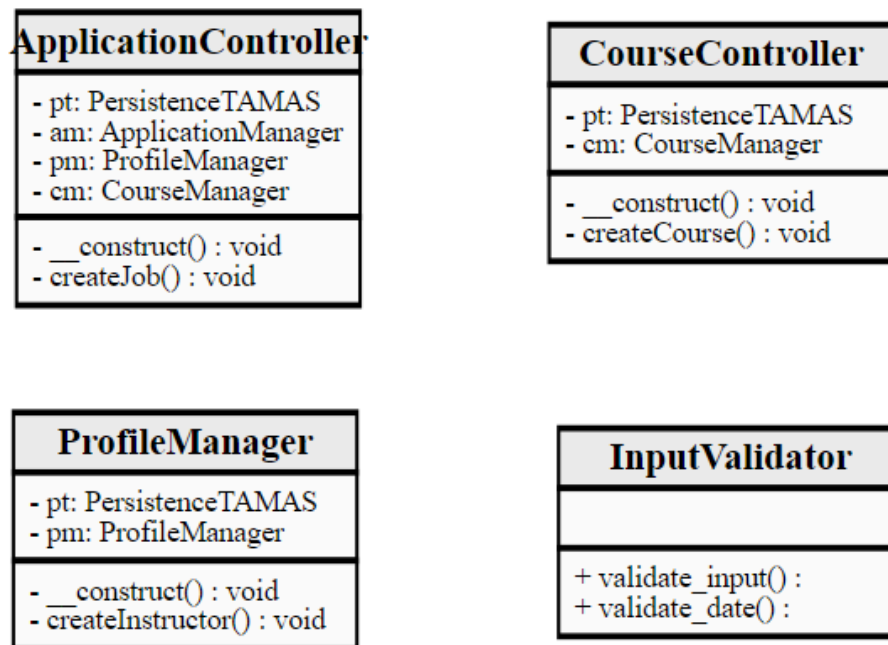
0.6 Detailed Design Diagram



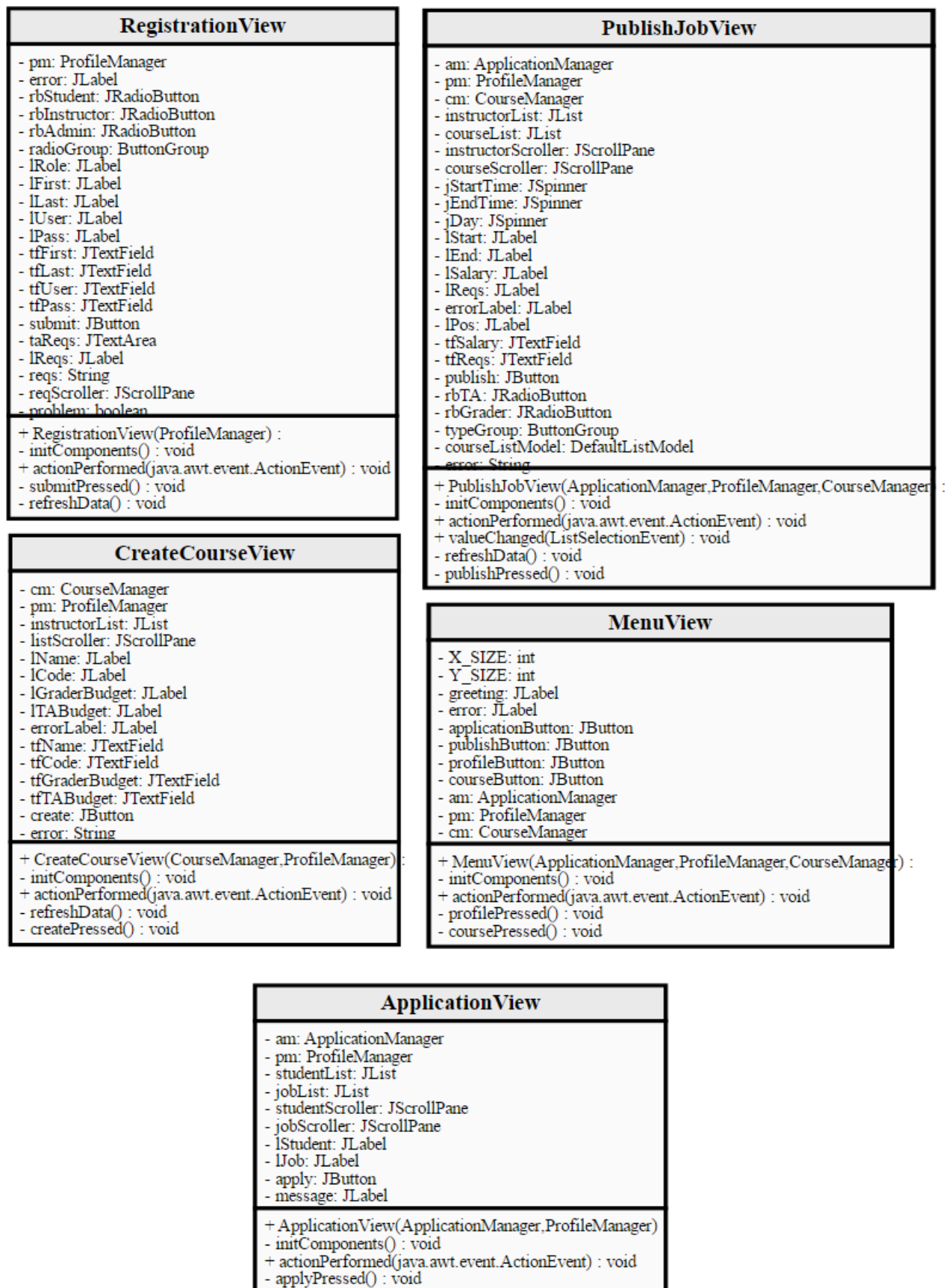
0.7 Java and Android Controller Class Diagram (Desktop and Mobile)



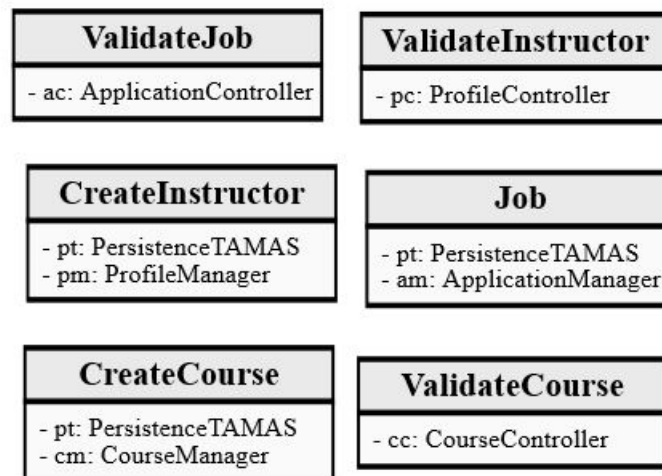
0.8 PHP Controller Class Diagram (Web)



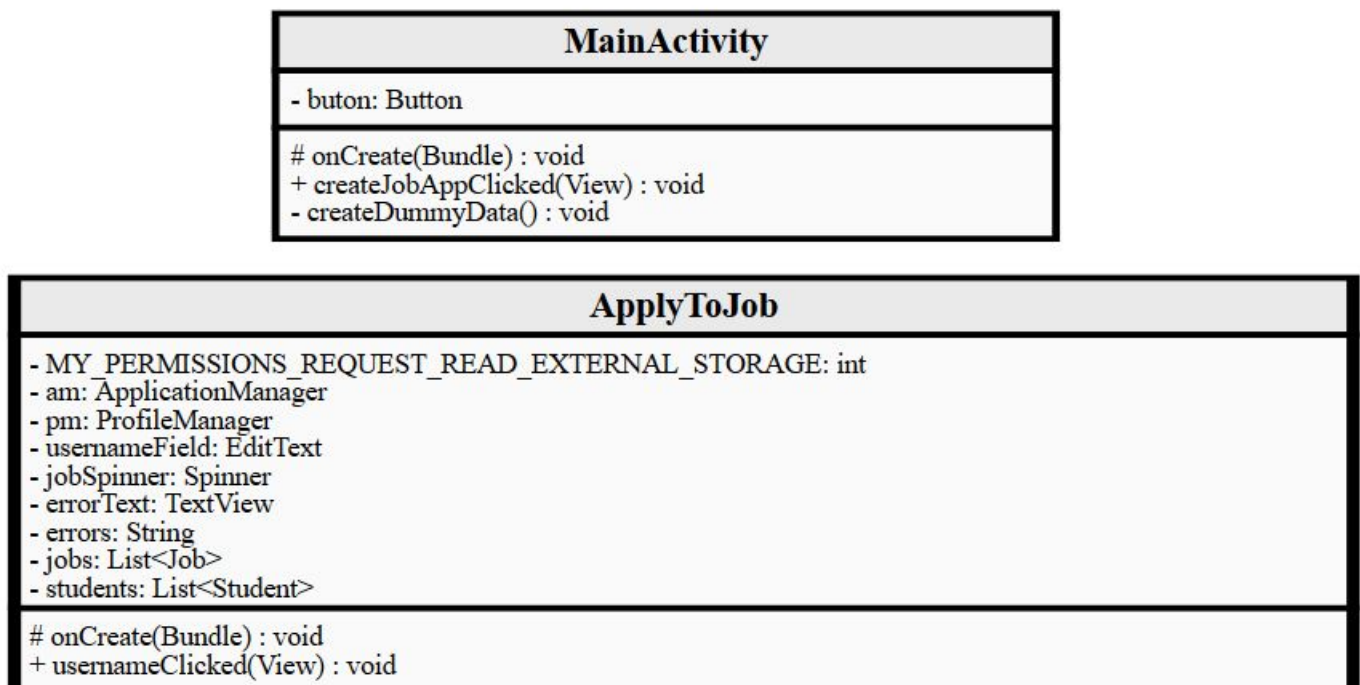
0.9 Java View Class Diagram (Desktop)



0.10 PHP View Class Diagram (Web)



0.11 Android View Class Diagram (Android)



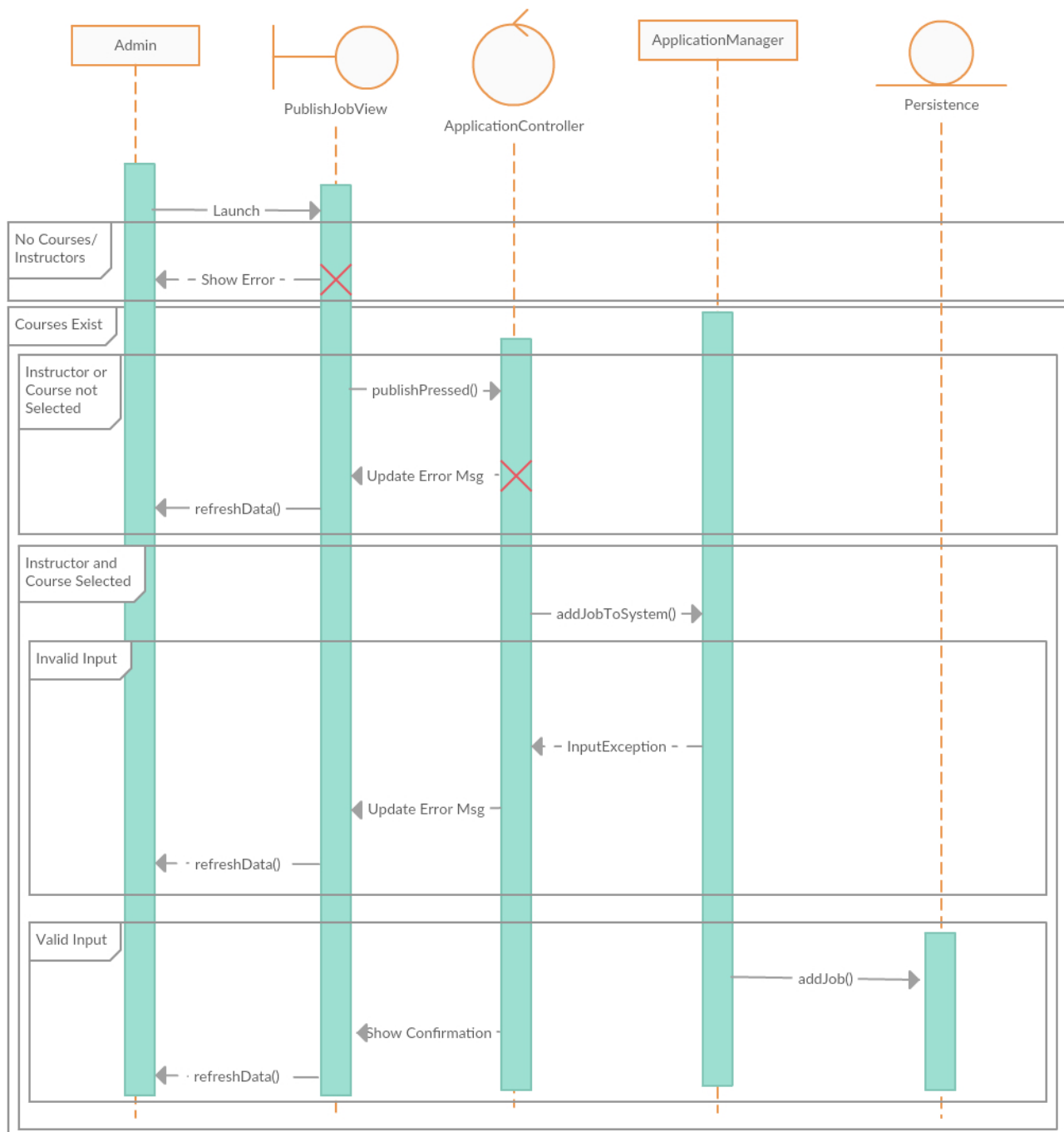
Part III

Implementation of Prototypes

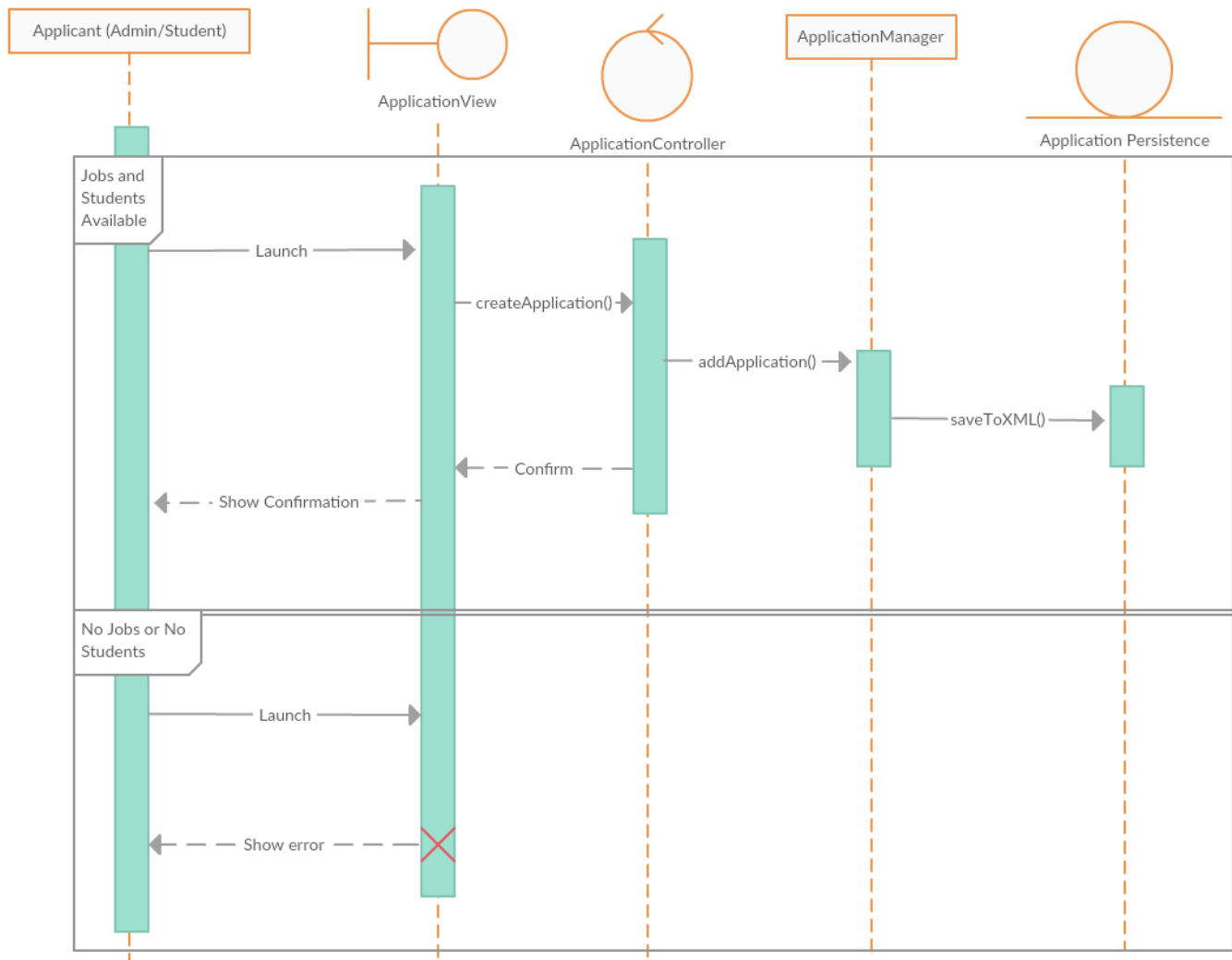
0.12 Desktop Application

The desktop application submitted in this Deliverable contains all functionalities necessary to generate the sufficient entities to complete the following required use cases: Apply to Job (Desktop), Apply to Job (Mobile), Publish Job Posting (Desktop). This includes the abilities to create profiles, create courses, create job postings, and submit job applications. All of the mentioned functionalities are persisted, so the XML files that are output may be used as data for the other platforms. Below are the sequence diagrams describing the desktop use cases.

0.12.1 Desktop Publish Job Posting Use Case



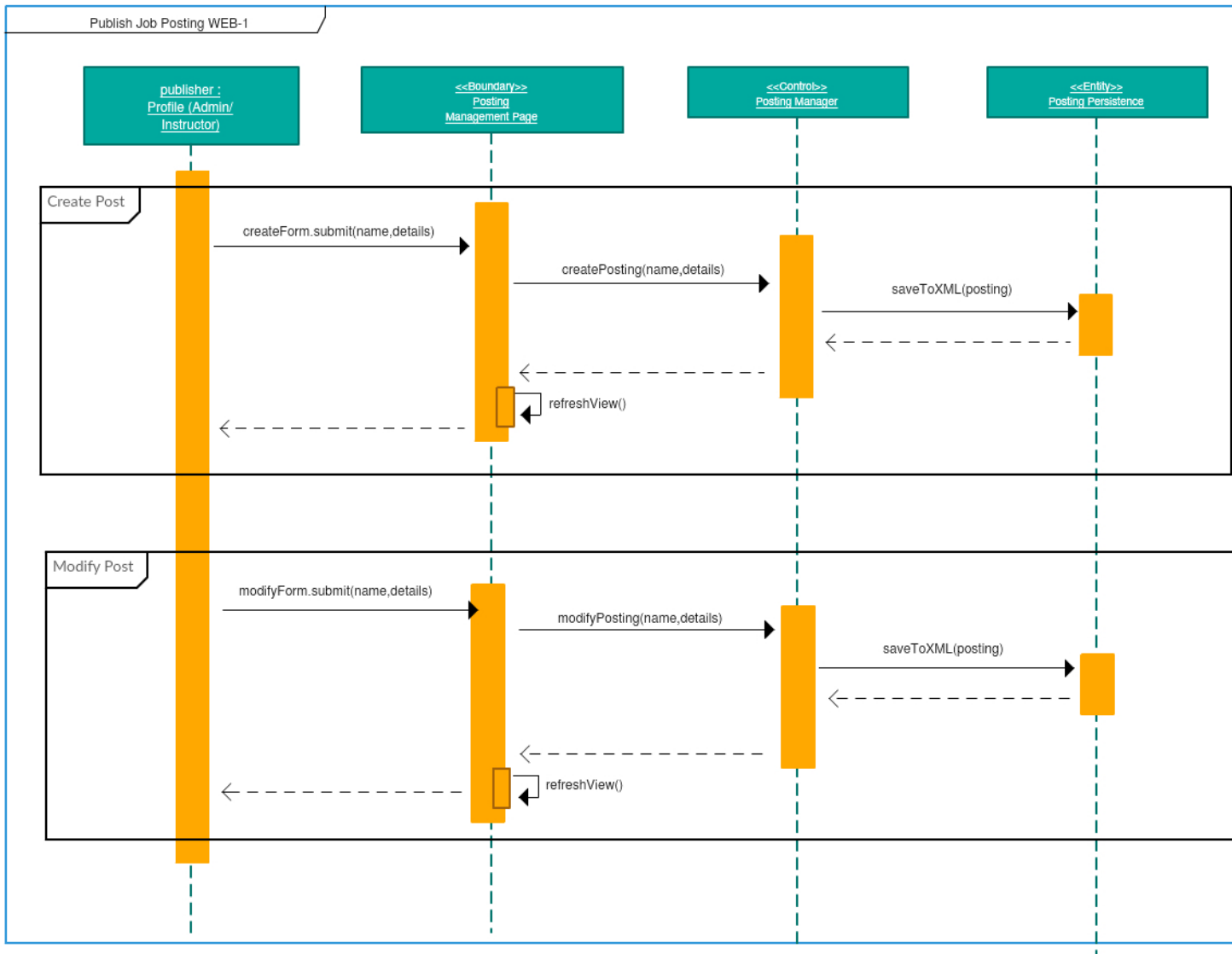
0.12.2 Desktop Apply to Job Use Case



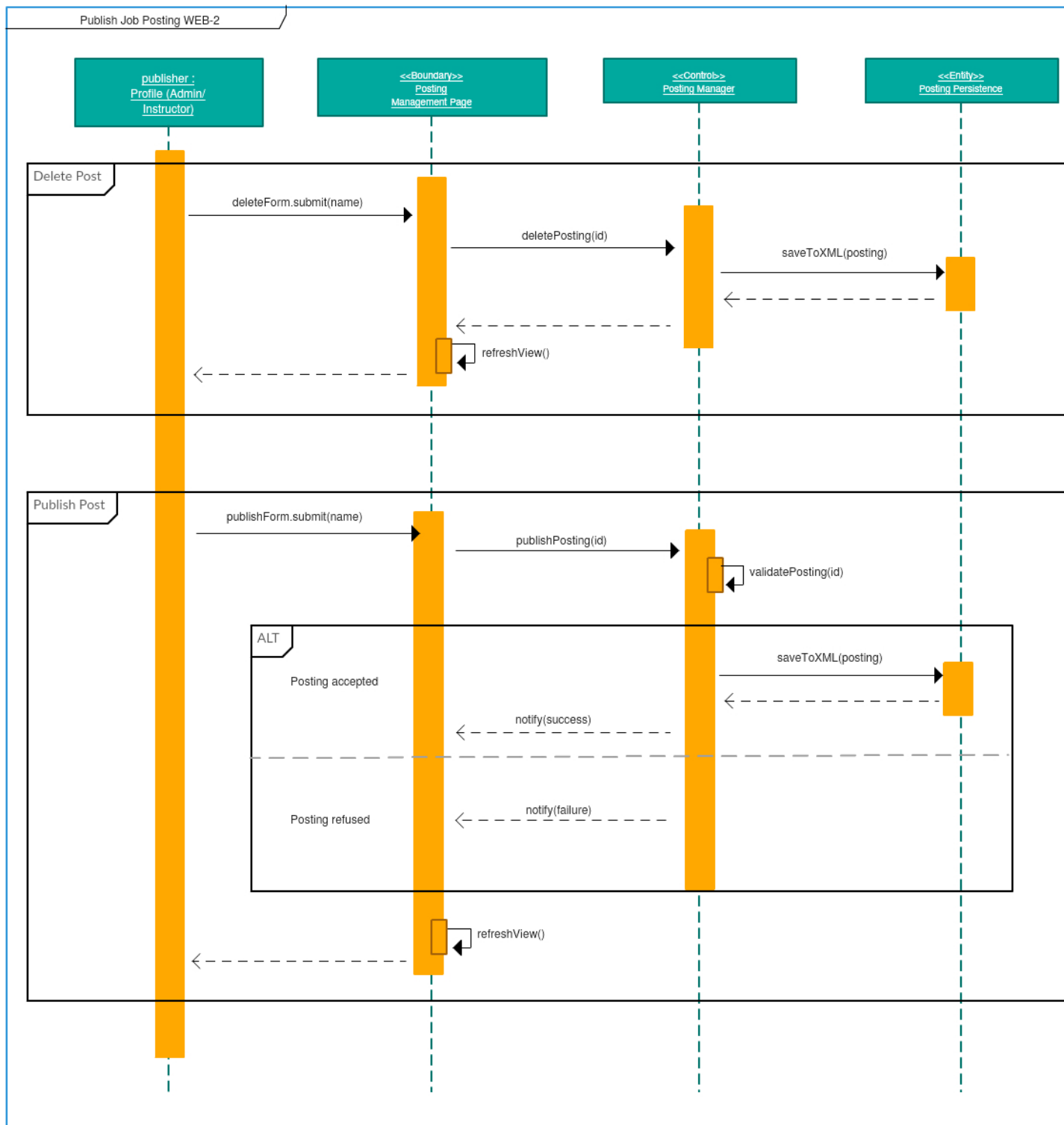
0.13 Web Application

The web application submitted in this Deliverable contains all functionalities required to complete the required Publish Job Posting (Web) use case. This means that the web application must be capable of creating instructors and creating courses, otherwise the Publish Job use case could not function without prior XML data. Although the desktop application can provide this XML data, it was easier in the development phase if the web application could do this independently. Therefore, the ability to create instructors and create courses is only temporary. Importantly, in publishing a job posting, the job data is saved in the persistence layer and may be copied to the appropriate directories on the other platforms for use. Following is the sequence diagram describing the required web use case.

0.13.1 Web Publish Job Posting Use Case



Continued on next page...

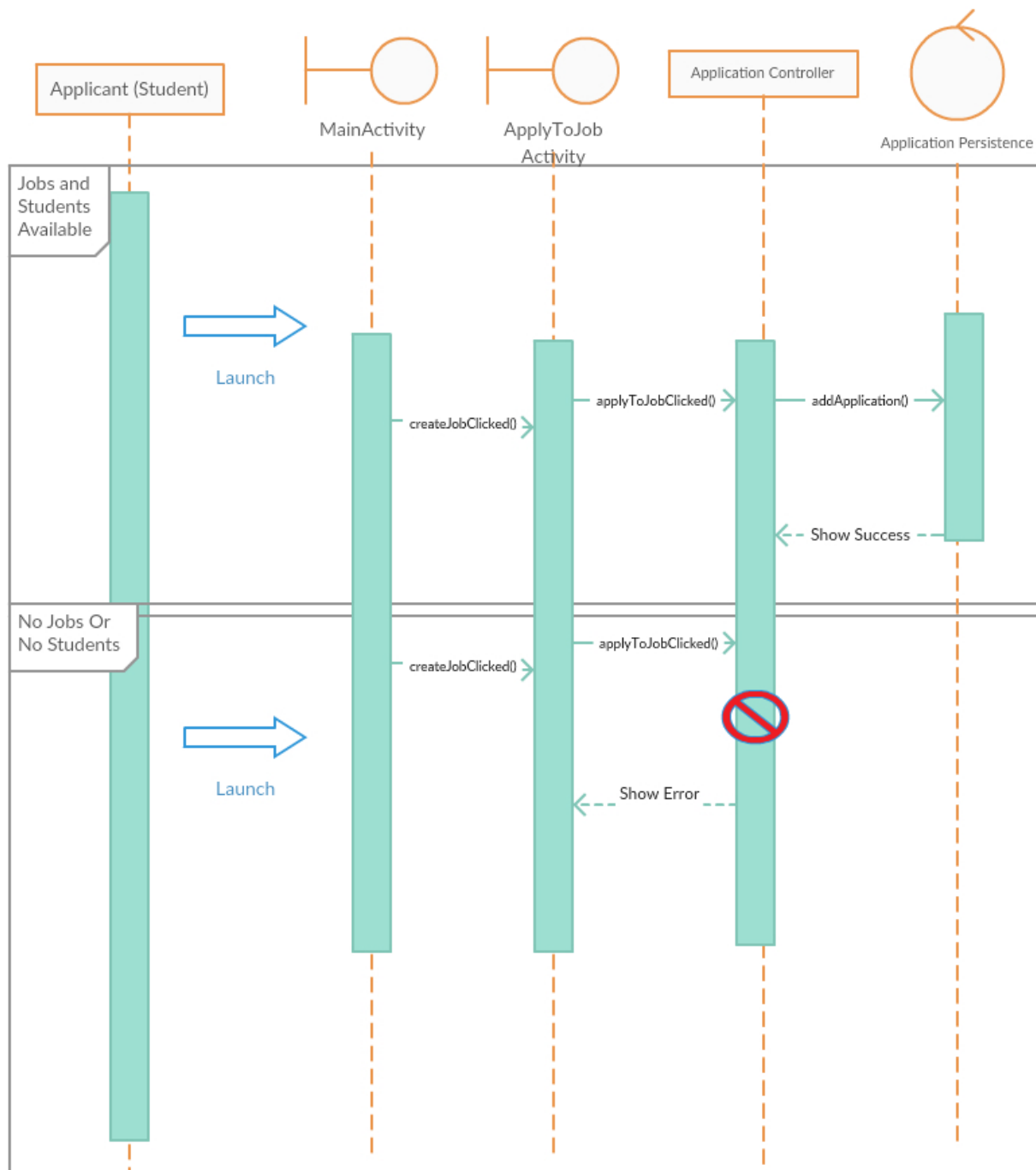


0.14 Mobile Application

The mobile application submitted in this Deliverable has the ability to complete the Apply to Job Posting (Mobile) use case, given prior XML data. For convenience, the mobile application generates sample entities if it is found that there is no data, to make testing and validation easier.

However, should someone wish to use their own XML data to test the application, this is possible by placing the XML files (say, that were generated from the desktop application) in the following directory: `/storage/emulated/0/Android/obb/ca.mcgill.ecse321.group10.tamas/`. This, of course, is also the directory of the output persistence of the mobile app. In applying to a job, the mobile application saves the `Application` data to a persistence XML file which can then be used on the other platforms. Below is the sequence diagram representing the required mobile use case.

0.14.1 Mobile Apply to Job Posting Sequence Diagram



Part IV

Work Plan From Past and Future Iterations

In this section, the updated workplan may be pondered. The full detailed backlog including work hours per team member, key decisions, and work history may be seen on the remote git repository's root directory in the file called `backlog.pdf`.

0.14.2 Iteration 1

In the following week, we will create more detailed sequence diagrams going into the details of the implementations on individual platforms. We will then develop prototype source code for the Publish Job Posting and Apply for Job use cases for the desktop and mobile applications respectively.

0.14.3 Iteration 2

In the following Deliverable, thorough testing needs to be completed. The following are deadlines and the estimated effort it will take to complete these tasks on a scale of 1 to 10.

- March 10 - First iteration of unit tests for Apply to Job and Publish Job posting on web, desktop and mobile. Estimated Effort: 8/10
- March 13 - Second iteration of unit tests for Apply to Job and Publish Job posting on web, desktop and mobile. Estimated Effort: 6/10
- March 13 - First iteration of System and Component tests. Estimated Effort: 6/10
- March 15 - First iteration of Stress Testing and Final iteration of unit tests. Estimated Effort: 6/10
- March 16 - First iteration of Descriptions of all tests done. Estimated Effort: 5/10
- March 10 - Deadline for Deliverable 3, including editing of backlog and update of workplan. Estimated Effort: 4/10