SOEN390 - Software Engineering Team Design Project
Team 6 - Deliverable 1

Architecture Description of
Condo Management System

Winter 2024

Done by:
Hoang Minh Khoi Pham 40162551
Michaël Gugliandolo 40213419
Jessey Thach 40210440
Mahanaim Rubin Yo 40178119
Vanessa DiPietrantonio 40189938
Ahmad Elmahallawy 40193418
Clara Gagnon 40208598
Khanh Huy Nguyen 40125396
Jean-Nicolas Sabatini-Ouellet 40207926
Mohamad Mounir Yassin 40198854

Professor Junqiu Yang
Department of Computer Science and Software Engineering
Gina Cody School of Engineering and Computer Science

Concordia University

Contents

Contents **2**
Using the template **4**
License **5**
Version history **5**
Editions **5**
Comments or questions **5**
1 Introduction **6**
1.1 Identifying information **6**
1.2 Supplementary information **6**
1.3 Other information **7**
1.3.1 Overview (optional) **7**
1.3.2 Architecture evaluations **7**
1.3.3 Rationale for key decisions **7**
2 Stakeholders and concerns **8**
2.1 Stakeholders **8**
2.1.1 User 8
2.1.2 Project Owner 9
2.2 Concerns **10**
2.3 Concern–Stakeholder Traceability 10
3 Viewpoints+ **11**
3.1 Logical View **11**
3.1.1 Overview 11
3.1.2 Concerns and stakeholders 11
3.1.3 Model kinds 12
3.2 Process View **13**
3.2.1 Overview 13
3.2.2 Concerns and stakeholders 13
3.2.3 Model kinds 13
3.3 Physical View **14**
3.3.1 Overview 14
3.3.2 Concerns and stakeholders 14
3.3.3 Model kinds 15
3.4 Development View **16**
3.4.1 Overview 16
3.4.2 Concerns and stakeholders 16
3.4.3 Model kinds 17
3.5 Use Case View **17**
3.5.1 Overview 17
3.5.2 Concerns and stakeholders 17

3.5.3 Model kinds 18
3.6 Operations on views 19
3.7 Correspondence rules 19
3.8 Examples (optional) 19
3.9 Notes (optional) 19
3.10 Sources 19
4 Views 20
4.1 View: Logical 20
4.1.1 Models 20
4.1.2 Known Issues with View 21
4.2 View: Process 21
4.2.1 Models 21
4.2.2 Known Issues with View 28
4.3 View: Physical 29
4.3.1 Models 29
4.3.2 Known Issues with View 30
4.4 View: Development 30
4.4.1 Models 30
4.4.2 Known Issues with View 31
4.5 View: Use Case 31
4.5.1 Models 31
4.5.2 Known Issues with View 34
5 Consistency and correspondences 34
5.1 Known inconsistencies 34
5.2 Correspondences in the AD 34
5.3 Correspondence rules 34
Bibliography 35

Using the template

ISO/IEC/IEEE 42010, *Systems and software engineering — Architecture description*, defines the contents of an architecture description (AD) [4].

Figure 1 depicts that contents in terms of a UML class diagram. The AD template in this document defines places for all required information and offers the user some additional guidance on preparing an AD.

An AD may take many forms, not prescribed by the Standard: it could be presented as a document, a set of documents, a collection of models, a model repository, or in some other form – as long as the required content is accessible in some manner. In particular, organization and ordering of required information is not defined by the Standard. Thus, headings and subheadings in this template are merely suggestive – not required.

The template uses a few conventions, as follows.

* “Musts” are items which must be present to satisfy the Standard. Musts are marked like this.

Δ “Shoulds” are items recommended to be present, but not required by the Standard. Shoulds are marked like this.

Optional items are marked with this: (optional). Guidance that defines, explains or otherwise amplifies upon the required items, or terms used therein, looks like this.

<Items> like <this> signal names to be filled-in by a user of the template and used throughout the resulting AD.

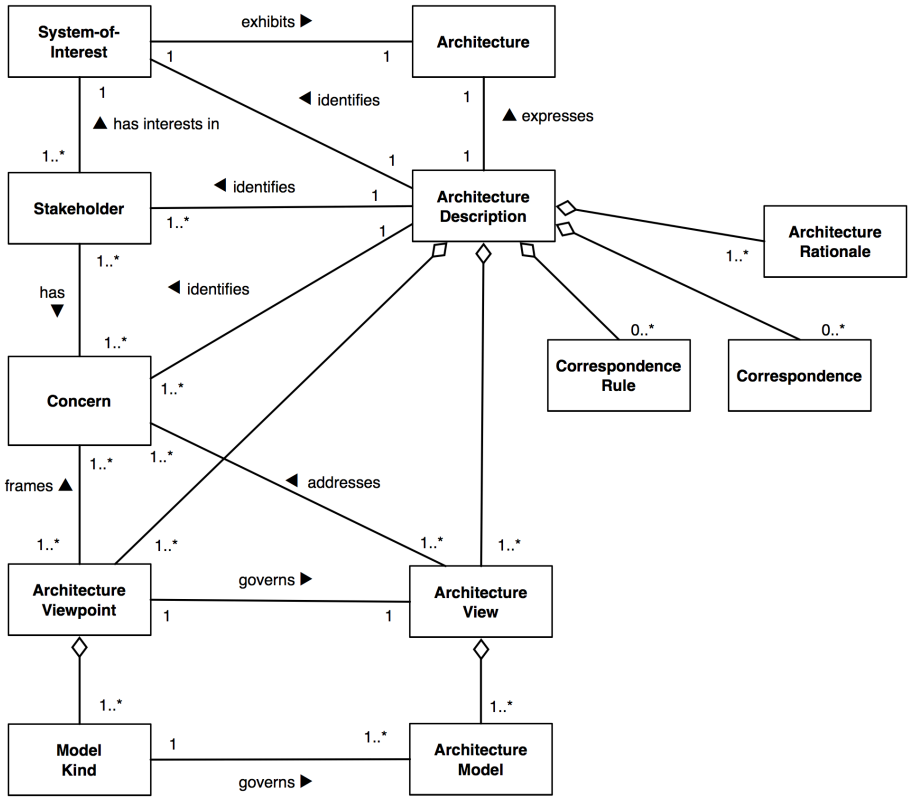

Figure 1

Figure 1: Content model of an architecture description

License

Version history

This template is based on one originally designed for use with IEEE std 1471:2000 [3] and now updated for ISO/IEC/IEEE 42010:2011. The present document is an enhanced version of the earlier template, with additional guidance, clarifications and examples for readers.

rev 2.2 7 October 2014, Moved bibliography from bibtex to biblatex. Released revision with minor formatting fixes.

rev 2.1a June 2012, Initial release on 42010 website.

Editions

This is the "bare bones" edition – it contains exactly only information items required by the Standard. Other editions meet the requirements of the Standard and contain additional information used in various documentation approaches (such as [6, 1]).

Comments or questions

Contact the author Rich Hilliard ⟨r.hilliard@computer.org⟩ with comments, suggestions, improvements or questions.

For more information on ISO/IEC/IEEE 42010, visit the website:
http://www.iso-architecture.org/42010/.

The template begins here (next page) . . .

1 Introduction

1.1 Identifying information

The Software Architecture Document details the software architecture of the Condo Management System, a web application designed to streamline the management of condominium properties, facilitating tasks such as facility reservations, financial management, resident registration, and communication amongst users and between users and management. The condo management system will follow the model-view-controller (MVC) architecture. This architecture pattern organizes the system's logic into three distinct layers, each carrying specific tasks. The model layer represents our database and how our data will be stored. The view layer is the UI of the application; either our webpage or the view of the mobile application. The controller layer is our backend, handling requests and triggered events from the view and modifying the model's data. Our controller layer will be separated into microservices for increased separation of concerns. Microservices are small loosely coupled independent services that communicate with APIs.

1.2 Supplementary information

Date of Issue and Status: Wednesday, February 7th Draft (This is a document that will be worked on continuously until the end of sprint 5)

Authors, Reviewers, Approving Authority, Issuing Organization: The authors, reviewers, and approving authority consists of the members of team 6. Every member has responsibilities as an author, reviewer, and approving authority.

Change History: The first issue of this document contains the following sections, which were:

1. Introduction
2. Stakeholders and Concerns
3. Viewpoints
4. Views
5. Consistency and Correspondence

Within the 5 sections, the following requirements were taken by the members of team 6:

Identifying Information: Vanessa & Jessey & Michaël

Supplementary Information: Vanessa & Jessey & Michaël

Architecture Evaluations and Rationale for Key Decisions: Jessey & Michaël

Stakeholders: Jessey & Michaël

Concerns: Jessey & Michaël

Concern-Stakeholder Traceability: Jessey & Michaël

Logical View Description: Jessey & Michaël

Physical View Description: Jessey & Michaël

Process View Description: Jessey & Michaël

Use Case View Description: Jessey & Michaël

Domain Model: Vanessa & Ahmad

Class Diagram: Vanessa

Activity Diagram: Vanessa

Sequence Diagram: Mahan & Jessey

Use Case Diagram: Jessey & Michaël & Khoi

Component Diagram: Jessey & Michaël & Khoi

Deployment Diagram: Jessey & Michaël & Khoi

Version Control: Version number v.1.0 (This is the first version for the SAD documentation, composing of the initial and preliminary parts for sprint 1)

References:

- Requirements and User Story Backlog Sprint 1

1.3 Other information

1.3.1 Overview (optional)

1.3.2 Architecture evaluations

1.3.3 Rationale for key decisions

The model-view-controller (MVC) architecture is the most appropriate architecture design for our system. Since we must develop the system on a website and a mobile app version, we can easily swap the view component. The controller and the model components will remain the same for both the website and the mobile app, only the view component will differ between the two.

We chose this model over the layered architecture model because the latter is too complex for the project's scope. Moreover, since the layers are dependent from one to an adjacent one, a change in a single layer can affect the entire system because it operates as a single unit. Therefore, scalability and maintenance is difficult.

The Condo Management System’s controller layer will be separated into independent microservices. Separating these concerns allows for improved scalability, better fault isolation and increased transfer speed. Moreover, microservices allow for better data security, because each service is responsible for a specific task, thus it makes it easier to implement security.

2 Stakeholders and concerns

2.1 Stakeholders

2.1.1 User

1. Concerns common to all users

Concern	Quality Attribute
Will my private information be leaked?	Security
Will my email be spammed with junk mail?	Security
Does the system protect from all unauthorized access?	Security
Is it easy to operate the system?	Usability

2. Public User: A new user who isn't registered and doesn't rent any condo unit.

Concern	Quality Attribute
Is this the right product for me?	Relevance

3. Tenant: A registered user who is renting a condo unit.

Concern	Quality Attribute
Is my information correct?	Correctness
Does the system give a quick response?	Usability

4. Condo Owner: A registered user who owns a condo.

Concern	Quality Attribute
Is my information correct?	Correctness
Does the system give a quick response?	Usability

5. Property Owner: A registered user who owns properties.

Concern	Quality Attribute
Is the system available all the time?	Availability
Will my condo and property lists load quickly?	Accessibility
Will my dashboard be often updated?	Accessibility

6. Condo Management Staff: A registered condo management staff

Concern	Quality Attribute
Is the system available all the time?	Availability
Does the system give a quick response?	Usability

2.1.2 Project Owner

1. Project Manager: Software project manager who plans, organizes and directs the project

Concern	Quality Attribute
Will we have enough time to finish?	Timeliness
Will we follow the budget?	Affordability
Have we identified all risks?	Risk identifiability
Will my team stay organized and motivated?	Team Motivation and Accountability
Which development process will we use?	Process Adaptability

2. Software Architect: Engineer who designs and ensures system requirements are met

Concern	Quality Attribute
Will our architecture be able to handle many users?	Scalability
Is our architecture secure?	Security
Will our architecture be easy to maintain?	Maintainability

3. Developer: Follow the development process to build the system

Concern	Quality Attribute
Will we collaborate well as a team?	Collaboration
Are the requirements precise and realistic?	Requirement Accuracy
Will the requirements change a lot during development?	Adaptability

4. Tester: Test the system through various checks

Concern	Quality Attribute
Will we collaborate well as a team?	Collaboration
Is testing planned from the beginning?	Testability
Will we automate tests?	Testability
Can we accept some small bugs?	Fault Tolerance

2.2 Concerns

#1 Concern: How feasible is it to construct and deploy the Condo Management System?

#2 Concern: What are the potential risks and impacts of the Condo Management System to its stakeholders throughout its life cycle?

#3 Concern: How is the Condo Management System to be maintained and evolved?

#4 Concern: How will the database be organized? How will the tables be related?

#5 Concern: How does the system deal with security (authentication)?

#6 Concern: How will the system perform under extreme conditions/system congestion (e.g. many requests are sent at the same time)?

#7 Concern: Will our controller layer adapt easily to multiple views?

2.3 Concern–Stakeholder Traceability

This association can be recorded via a simple table or other depiction.

Table 2.1: Association of stakeholders to concerns in an AD

	Concern 1	Concern 2	Concern 3	Concern 4	Concern 5	Concern 6	Concern 7
Public user	-	-	-	-	X	X	-
Tenant	-	X	X	-	X	X	-
Condo owner	-	X	X	-	X	X	-
Property owner	-	X	X	-	X	X	-
Condo management staff	-	X	X	-	X	X	-
Project manager	X	X	X	X	X	X	X
Software architect	X	X	X	X	X	X	X
Developer	X	X	X	X	X	X	X
Tester	X	X	-	-	X	X	X

3 Viewpoints+

3.1 Logical View

Also known as the structural view.

3.1.1 Overview

This view provides a structured representation of the key concepts, entities and relationships within a problem domain. The logical view concerns the system's overall functionality provided to the end-users. It focuses on the static structure of the system, therefore only dealing with essential components such as classes, objects, attributes and relationships. Some examples of this view are the domain model and the class diagram. The class diagram visually describes the dependencies and how different components of the system interact with each other through classes and associations. The logical view provides a perception of the overall design and organizational structure of the system. [1]

3.1.2 Concerns and stakeholders

3.1.2.1 Concerns

#4 Concern: How will the database be organized? How will the tables be related?

The domain model, which falls under the logical view, loosely represents how the data will be organized in the database. Some classes are equivalent to tables, such as the User class where its attributes are going to be mapped to columns in the database.

3.1.2.2 Typical stakeholders

- Project manager
- Software architect
- Developer

This view sets up the layout of the system, which concerns people directly working on the construction of the software, except testers. Testers aren't included because they check that everything is working fine but the minute details of the system's structure aren't relevant to them. General users, such as tenants and staff, aren't included because they only care about the end product.

3.1.2.3 "Anti-concerns" (optional)

3.1.3 Model kinds

3.1.3.1 Domain model

The domain model is a visual representation of meaningful conceptual classes from the real-world objects in the domain of interest. The domain model illustrates the domain objects, the associations between the objects and their attributes. At the level of the domain model, it is still not possible to develop or write code based on this representation. [2] [3]

3.1.3.1.1 Domain model conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.1.3.1.2 Domain model operations (optional)

3.1.3.1.3 Domain model correspondence rules

3.1.3.2 Class diagram

As explained in the introduction, our system isn't object-oriented, so we will do a sequence diagram instead of a class diagram. However, here's the description of a class diagram.

A class diagram is similar to the domain model, but the class diagram also shows the attributes, operations and relationships among objects from the system's conceptual classes. In other words, this UML (Unified Modeling Language) diagram is a type of static structure diagram which describes the structure of a system. It is the most important part of designing for an object-oriented system because the class diagram illustrates the classes like objects to be programmed. Therefore, translating the model into executable code is not too complex to do. [4] [5]

3.1.3.2.1 Class diagram conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.1.3.2.2 Class diagram operations (optional)

3.1.3.2.3 Class diagram correspondence rules

3.2 Process View

Also known as the behaviour view.

3.2.1 Overview

Contrary to the logical view which focuses on a static view, the process view focuses on the dynamic view of the system. This means that it details the system's behaviour by dissecting the flow of control between components during a process. This is a detailed breakdown of the steps needed to accomplish a use case inside the system. Any sequential flow of activities, interactions and dependencies are taken into account in this view. In this view, stakeholders gain a better insight into resource utilization, task sequencing and latent bottleneck, resulting in potential efficiency and optimized performance.An example of this view are activity diagram. [1]

3.2.2 Concerns and stakeholders

3.2.2.1 Concerns

#1 Concern: How feasible is it to construct and deploy the Condo Management System?

Activity diagrams illustrate system behaviour and how different components collaborate. They are built by system designers, and they can be used to demonstrate the feasibility of the Condo Management System.

3.2.2.2 Typical stakeholders

- Project manager
- Software architect
- Developer
- Tester

This view describes the behaviours of the system, which concerns people directly working on the development of the software. The project manager and software architect design the various processes, and developers and testers develop them and ensure full functionality in all possible scenarios. General users aren't included because they only care about the end product.

3.2.2.3 "Anti-concerns" (optional)

3.2.3 Model kinds

3.2.3.1 Activity diagram

In our condominium management system, we have adopted a microservices architecture, wherein each backend system corresponds with a distinct process. This design philosophy allows for granular control and scalability, ensuring efficient handling of various user interactions. Through meticulous activity diagram representations, we elucidate the intricacies of four pivotal user interactions: user registration, user login, reservation creation, and annual report generation.

3.2.3.1.1 Activity diagram conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.2.3.1.2 Activity diagram operations (optional)

3.2.3.1.3 Activity diagram correspondence rules

3.2.3.2 Sequence diagram

A sequence diagram is an interaction diagram that illustrates how the operations unfold. These diagrams depict the interaction between objects in collaboration. Sequence diagrams are arranged in a time sequence, visually portraying the order of interactions. [13]

3.2.3.2.1 Sequence diagram conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.2.3.2.2 Sequence diagram operations (optional)

3.2.3.2.3 Sequence diagram correspondence rules

3.3 Physical View

Also known as the deployment view.

3.3.1 Overview

The physical view is related to the hardware infrastructure needed for our system. For example, since we will use a server to host our website and database, the physical view will illustrate connections between each layer, which are nodes for this view. Furthermore, this view tackles scalability and resource allocation, since they depend on the server specifications, availability and plans for its future. An example of the physical view is the deployment diagram. [1]

3.3.2 Concerns and stakeholders

3.3.2.1 Concerns

#1 Concern: How feasible is it to construct and deploy the Condo Management System?

Deployment diagrams illustrate where and how the components are used in the system's deployment. They are built by system designers, and they demonstrate the physical infrastructure needed to deploy the system.

#3 Concern: How is the Condo Management System to be maintained and evolved?

Once again, deployment diagrams illustrate the detailed configuration of the run time system from the components. Therefore, this view exemplifies how the components interact through the APIs and servers, thus making it possible to plan the supervision, maintenance and evolution of the Condo Management System.

#6 Concern: How will the system perform under extreme conditions/system congestion (e.g. many requests are sent at the same time)?

The physical view contains specifications about servers and how they are connected together. To answer questions related to high traffic and resource allocation, developers use the physical view, since it's the only view containing the concrete information needed to answer those questions.

3.3.2.2 Typical stakeholders

- Public user
- Tenant
- Condo owner
- Property owner
- Condo management staff
- Project manager
- Software architect
- Developer
- Tester

This view shows the physical infrastructure of the system, which concerns all 9 stakeholders related to this project. The development team must know what they are working with and how their components connect. General users are concerned with the system's performance and maintainability for the future.

3.3.2.3 "Anti-concerns" (optional)

3.3.3 Model kinds

3.3.3.1 Deployment diagram

Deployment diagrams portray the physical architecture of a system, like how nodes in a distributed system are arranged. They reveal how software and hardware elements interact, showcasing the physical distribution of processing. They also display the artifacts stored on each node and the components they implement. Nodes represent hardware devices; some examples are computers, sensors, printers and any other device that supports the system's runtime environment. Communication paths and deployment relationships represent the connections in the system. [8]

3.3.3.1.1 Deployment diagram conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.3.3.1.2 Deployment diagram operations (optional)

3.3.3.1.3 Deployment diagram correspondence rules

3.4 Development View

Also known as the implementation view.

3.4.1 Overview

The development view shows how each component is interconnected when developing our system. It will demonstrate our environments and development classes, not present in the logical view. In addition, it will specify the interfaces connecting the different modules to ensure that the whole development team is on the same page. An example of the development view is the component diagram. [1]

3.4.2 Concerns and stakeholders

3.4.2.1 Concerns

#5 Concern: How does the system deal with security (authentication)?

The development view identifies all the software components in the system and their distribution. It identifies layers, what goes in each one, and how the layers are intertwined. This is crucial to plan precisely and in advance since architectures with components separated into independent layers or modules have increased security. For example, in our MVC architecture, the view doesn't have access to the model (database), it has to go through the controller and its user authentication.

#7 Concern: Will our controller layer adapt easily to multiple views?

To address this concern, it is key to identify all links between the controller and the view. These are represented best in the development view.

3.4.2.2 Typical stakeholders

- Public user
- Tenant
- Condo owner
- Property owner
- Condo management staff
- Project manager
- Software architect
- Developer
- Tester

This view is crucial for the development team, as it shows what they have to do during implementation rather than a more general design. It also lays out how to achieve the needed security, which concerns the general public and all users of the system.

3.4.2.3 “Anti-concerns” (optional)

3.4.3 Model kinds

3.4.3.1 Component diagram

A component diagram is a UML diagram which outlines how the physical components in a system are organized and interconnected. These diagrams are created to model implementation details and to ensure that the planned development adequately covers all the requirements of the system. Some examples of components that can be found in a component diagram are documents, database table, files, executables and any physical elements with a location in a storage. Moreover, these diagrams are almost identical to class diagrams but they focus on a system’s components, providing a static implementation view of the system. [9] [10]

3.4.3.1.1 Component diagram conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.4.3.1.2 Component diagram operations (optional)

3.4.3.1.3 Component diagram correspondence rules

3.5 Use Case View

Also known as the scenario view.

3.5.1 Overview

The use case view is the only view focusing on an external view of the system. It elaborates on all the actions a user can perform from their perspective. It models their interactions and how the system responds under all scenarios; this process allows us to verify the requirements and to identify possible shortcomings because of its different perspective from the other views. An example of the use case view is the use case diagram. [1]

3.5.2 Concerns and stakeholders

3.5.2.1 Concerns

#2 Concern: What are the potential risks and impacts of the Condo Management System to its stakeholders throughout its life cycle?

To view our system from the stakeholder’s perspective, we utilize the use case view. Since it explains interactions between users and the system, it highlights the validity of the requirements and any risks

related. As explained above, use cases cover how the system should respond under any scenario, so they can be used to identify limitations.

3.5.2.2 Typical stakeholders

- Tenant
- Condo owner
- Property owner
- Condo management staff
- Project manager
- Software architect
- Developer
- Tester

This view describes the behaviours of the system and how to interact with it. This concerns the development team since they have to consider every possible action that users can take to ensure full functionality in all scenarios. General users are concerned because they want the least amount of issues and risks. This view is also ideal for all stakeholders because it's the easiest to understand due to its high-level nature.

3.5.2.3 "Anti-concerns" (optional)

3.5.3 Model kinds

3.5.3.1 Use case diagram

A use case diagram falls under the category of dynamic or behaviour diagrams in UML. These diagrams capture a high-level overview functionality, scope and requirements of a system through interactions between the system and its actors. Use cases encompass a series of actions, services, and functions that the system must execute. The actors are individuals or entities fulfilling defined roles within the system. While use case diagrams describe what the system does and how actors engage with it, they do not delve into the internal workings of the system. [11] [12]

3.5.3.1.1 Use case diagram conventions

I) Model kind languages or notations (optional)

II) Model kind metamodel (optional)

III) Model kind templates (optional)

3.5.3.1.2 Use case diagram operations (optional)

3.5.3.1.3 Use case diagram correspondence rules

3.6 Operations on views

3.7 Correspondence rules

3.8 Examples (optional)

3.9 Notes (optional)

3.10 Sources

[1] vpadmin, "4 + 1 Views in Modeling System Architecture with UML," *Visual Paradigm Guides*, Sep. 12, 2023. https://guides.visual-paradigm.com/4-1-views-in-modeling-system-architecture-with-uml/ (accessed Feb. 01, 2024).
[2] "Domain model," *Wikipedia*, Jan. 05, 2024. https://en.wikipedia.org/wiki/Domain_model (accessed Feb. 01, 2024).
[3] O. Chursin, "A Brief Introduction to Domain Modeling," *Medium*, Dec. 27, 2017. https://olegchursin.medium.com/a-brief-introduction-to-domain-modeling-862a30b38353 (accessed Feb. 01, 2024).
[4] "Class diagram," *Wikipedia*, Jul. 28, 2022. https://en.wikipedia.org/wiki/Class_diagram (accessed Feb. 01, 2024).
[5] Visual Paradigm, "What is Class Diagram?," *Visual-paradigm.com*, 2019. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/ (accessed Feb. 01, 2024).
[6] "Activity diagram," *Wikipedia*, May 09, 2020. https://en.wikipedia.org/wiki/Activity_diagram (accessed Feb. 01, 2024).
[7] Visual Paradigm, "What is Activity Diagram?," *Visual-paradigm.com*, 2019. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-activity-diagram/ (accessed Feb. 01, 2024).
[8] "Deployment diagrams," *www.ibm.com*, Sep. 25, 2020. https://www.ibm.com/docs/en/rational-soft-arch/9.7.0?topic=diagrams-deployment (accessed Feb. 01, 2024).
[9] "What is Component Diagram?," *Visual-paradigm.com*, 2019. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-component-diagram/ (accessed Feb. 01, 2024).
[10] "Component Diagrams - See Examples, Learn What They Are," *www.smartdraw.com*. https://www.smartdraw.com/component-diagram/ (accessed Feb. 01, 2024).
[11] smartdraw, "Use Case Diagrams - Use Case Diagrams Online, Examples, and Tools," *Smartdraw.com*, 2019. https://www.smartdraw.com/use-case-diagram/ (accessed Feb. 01, 2024).
[12] IBM, "Use-case diagrams," *www.ibm.com*, Mar. 04, 2021. https://www.ibm.com/docs/en/rational-soft-arch/9.6.1?topic=diagrams-use-case (accessed Feb. 01, 2024).
[13] Visual Paradigm, "What is Sequence Diagram?," *Visual-paradigm.com*, 2019. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-sequence-diagram/ (accessed Feb. 01, 2024).

4 Views

4.1 View: Logical

4.1.1 Models

4.1.1.1 Domain Model

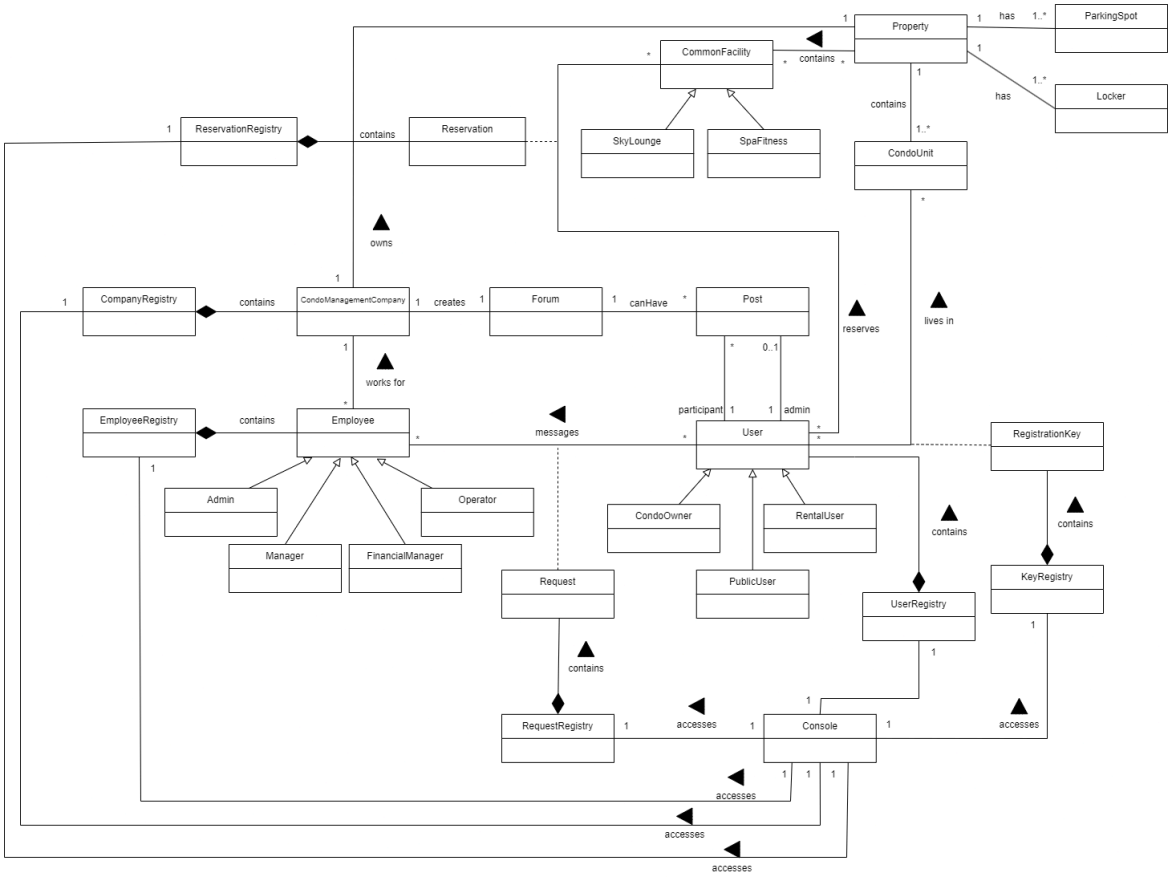

Figure 1: Domain model of the Condo Management System

The domain model organizes the user types, employee types and common facility types. Moreover, the console acts as the controller and is associated with 6 registries: key registry, employee registry, user registry, reservation registry, company registry, and request registry. The relationships between property, company, condo, users, and many more are modelled in this diagram.

4.1.1.2 Class Diagram

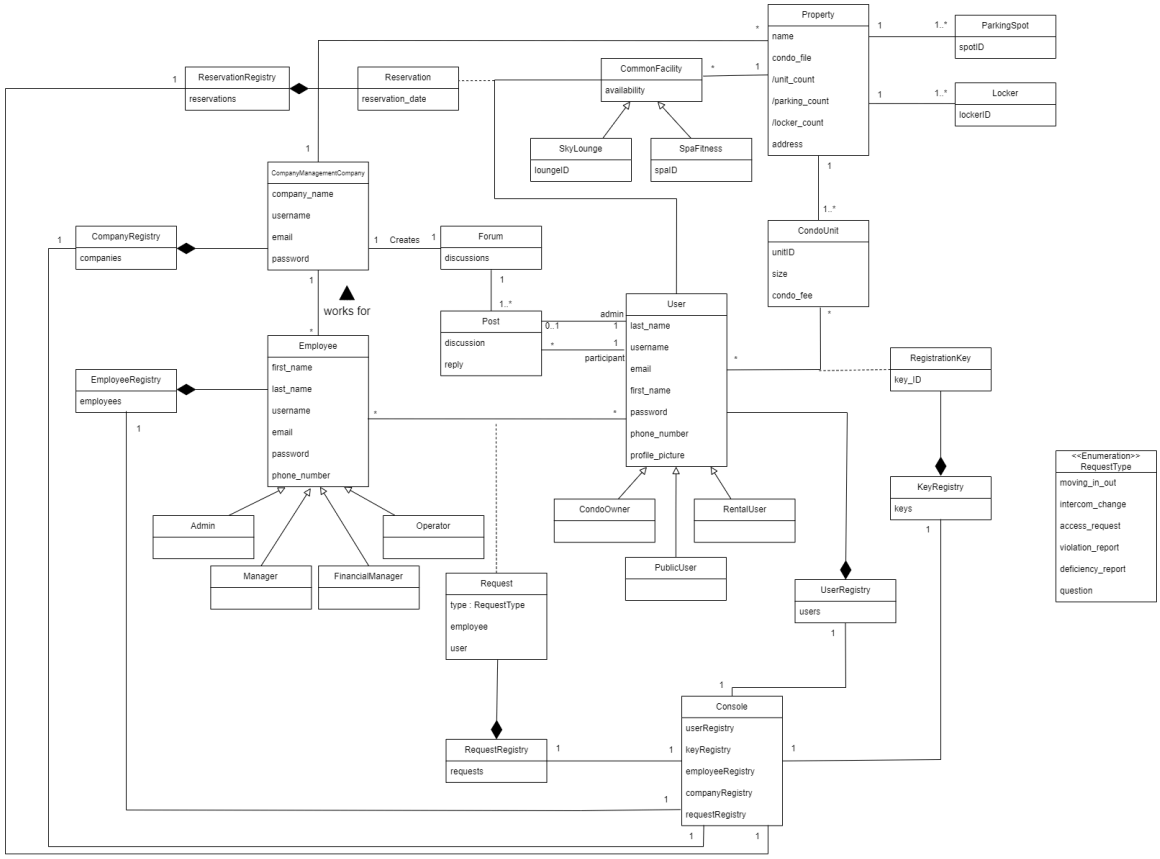

Figure 2: Class Diagram of the Condo Management System

The class diagram encapsulates the structure of the Condo Management system, represented by classes. It builds off the domain model, providing the static organization of user, employee, and common facility types as well as the relationships among all the entities within the system. Additionally, their attributes are listed in this diagram.

4.1.2 Known Issues with View

It's difficult to represent data flow in this view because it's a static view of the system and there is little definition of attribute dependencies between classes. Also, naming inconsistencies create confusion.

4.2 View: Process

4.2.1 Models

4.2.1.1 Activity Diagram

Each activity diagram addressed a group of related use cases or a simplified system within the Condo Management System. This serves to provide a clear visual representation of the steps involved in fulfilling a specific requirement. Each diagram is divided into swimlanes to represent the different

actors/entities involved in the process, and the series of actions and interactions within that particular entity.

Note that company registration and login follow the same flow of activities as for the public users, as illustrated in Figure 3 and Figure 4 below.

4.2.1.1.1 Public User Registration

This diagram depicts the process of actions which is taken by a public user when they would like to register for a new account.

Precondition: User is not logged in.

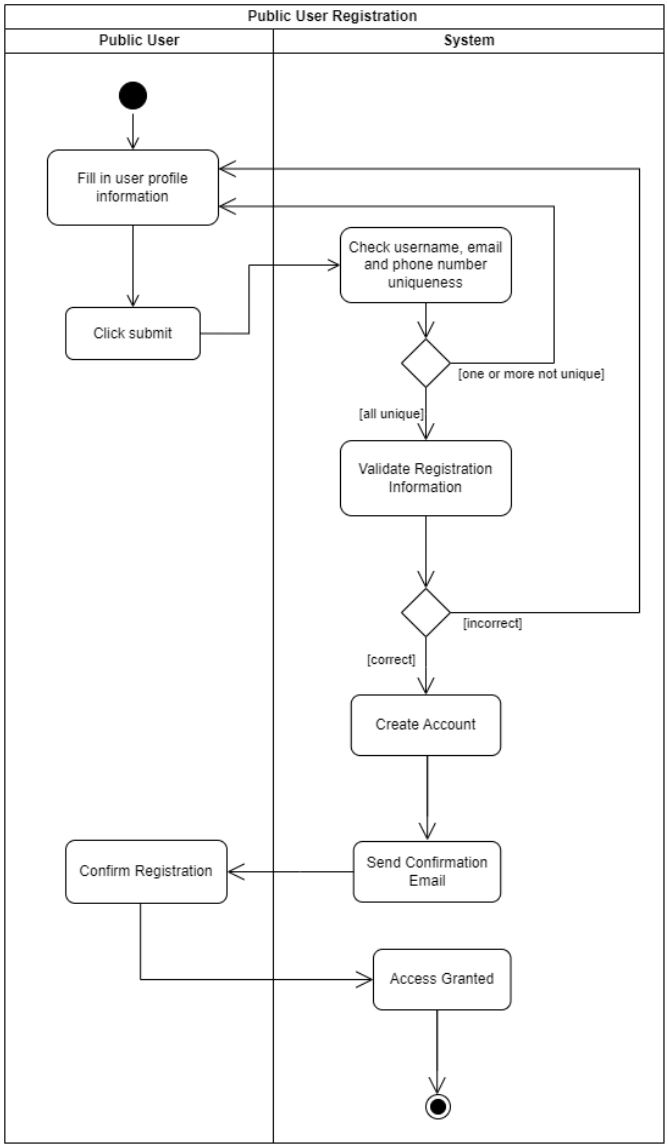

Figure 3: Activity Diagram for Public User Registration

4.2.1.1.2 Public User Login

This diagram depicts the process of actions which is taken by a public user when they would like to log in to their account.

Precondition: User is not logged in.

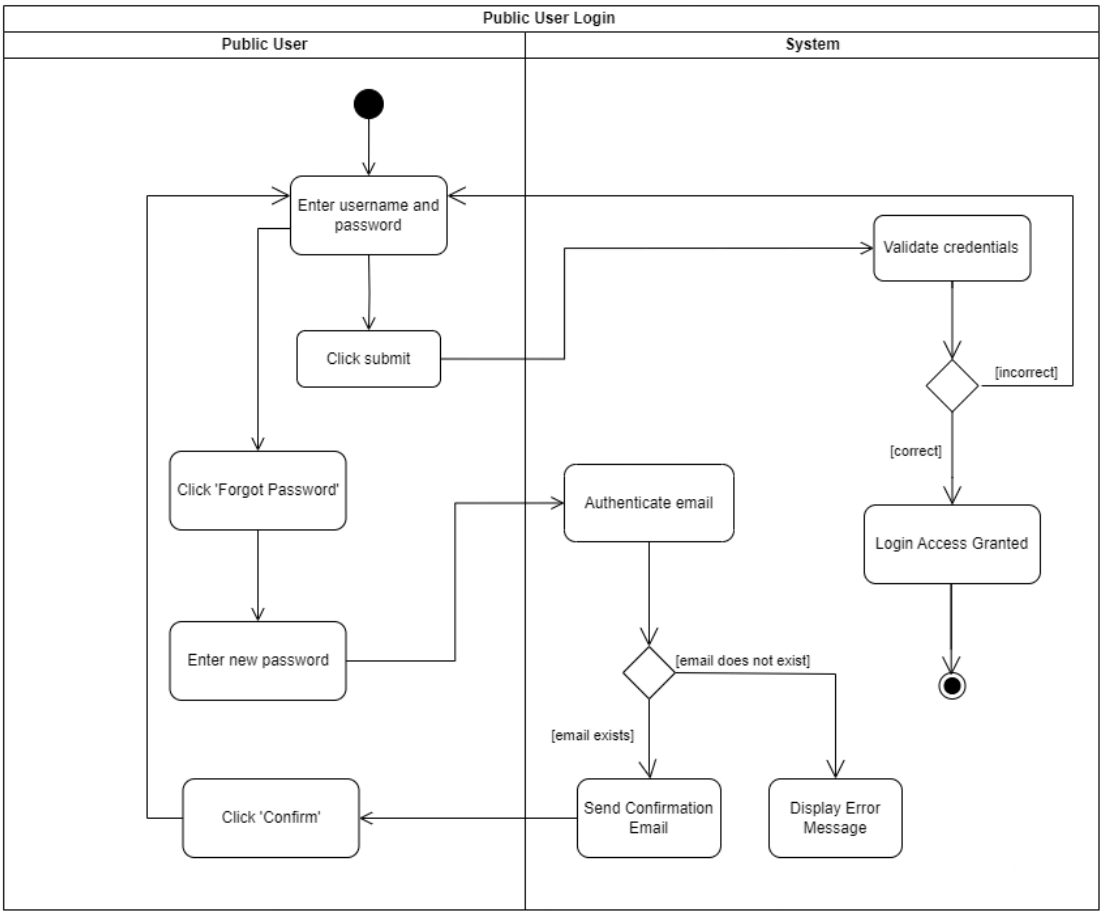

Figure 4: Activity Diagram for Public User Login

4.2.1.1.3 Simplified Reservation System

This diagram depicts the process of actions which is taken by a registered user (rental user or condo owner) when they would like to make a reservation for one of the common facilities of the property. The user can view and manage the common facilities on a calendar-like interface. The user is assumed to be logged in.

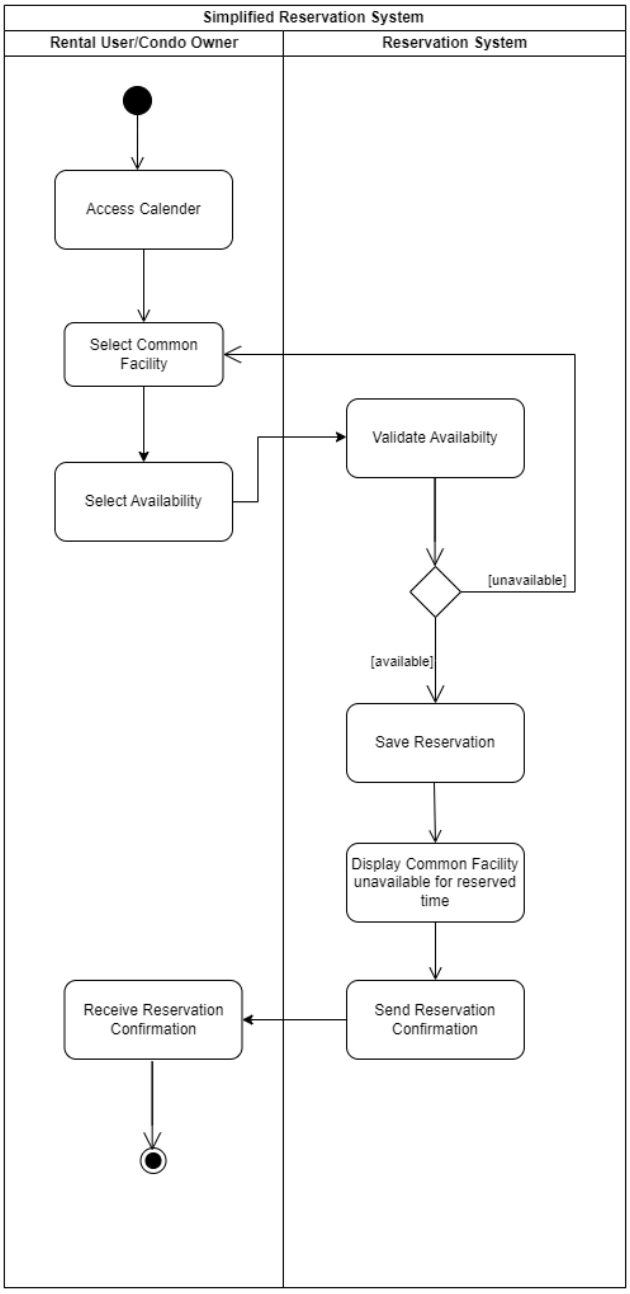

Figure 5: Activity Diagram for Simplified Reservation System

4.2.1.1.4 Simplified Financial System

This diagram depicts the process of actions which is taken by the company admin using the financial system within the Condo Management System. A user can enter a condo fee per square foot and parking spot, and enter operational cost. The financial system will record the operational budget, and generate an annual report. The condo fee of each unit will be calculated and presented to the unit owner.

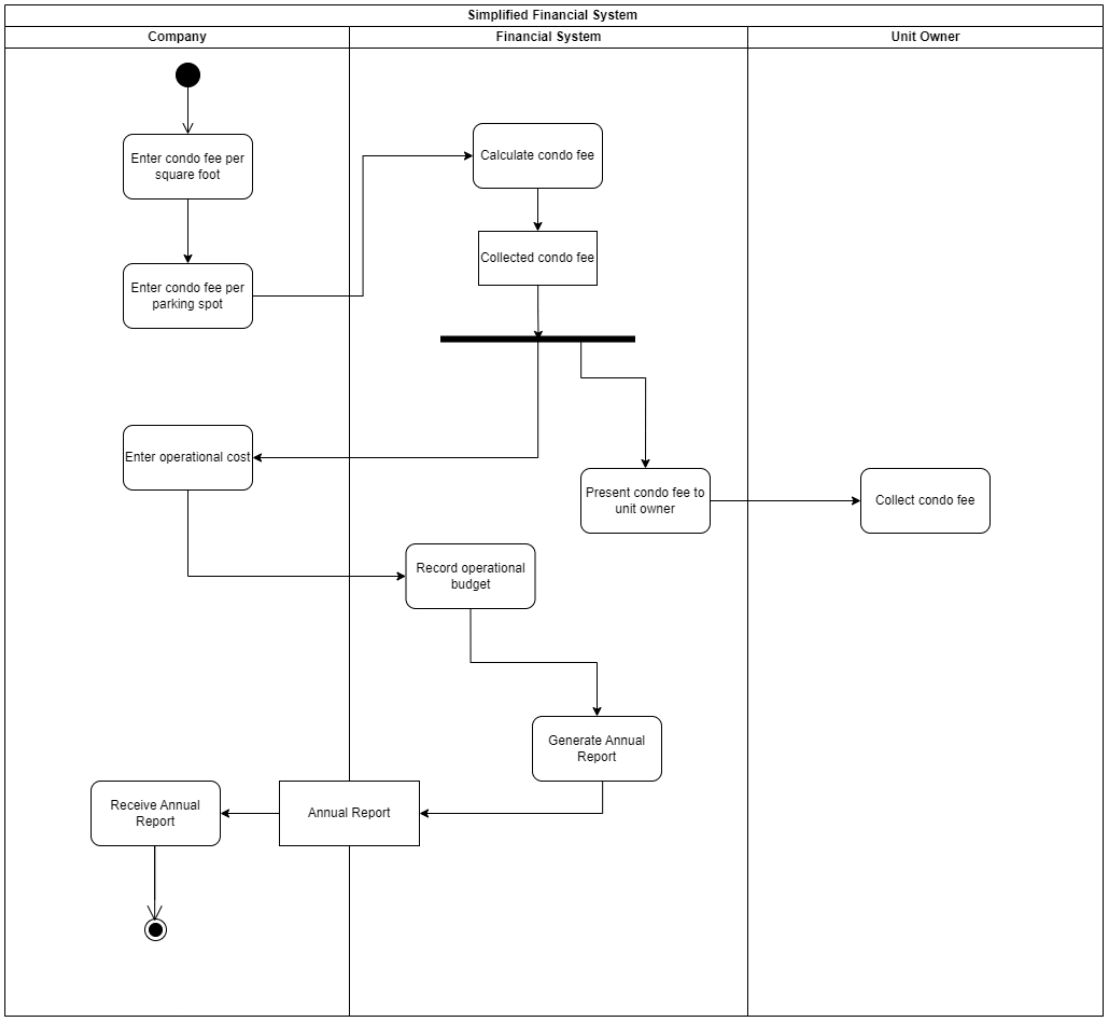

Figure 6: Activity Diagram for Simplified Financial System

4.2.1.1.5 Company Employee Management

This diagram depicts the process of actions which is taken by the company admin to create employee profiles and assign their roles in the system. The company admin can view and manage the employee profiles.

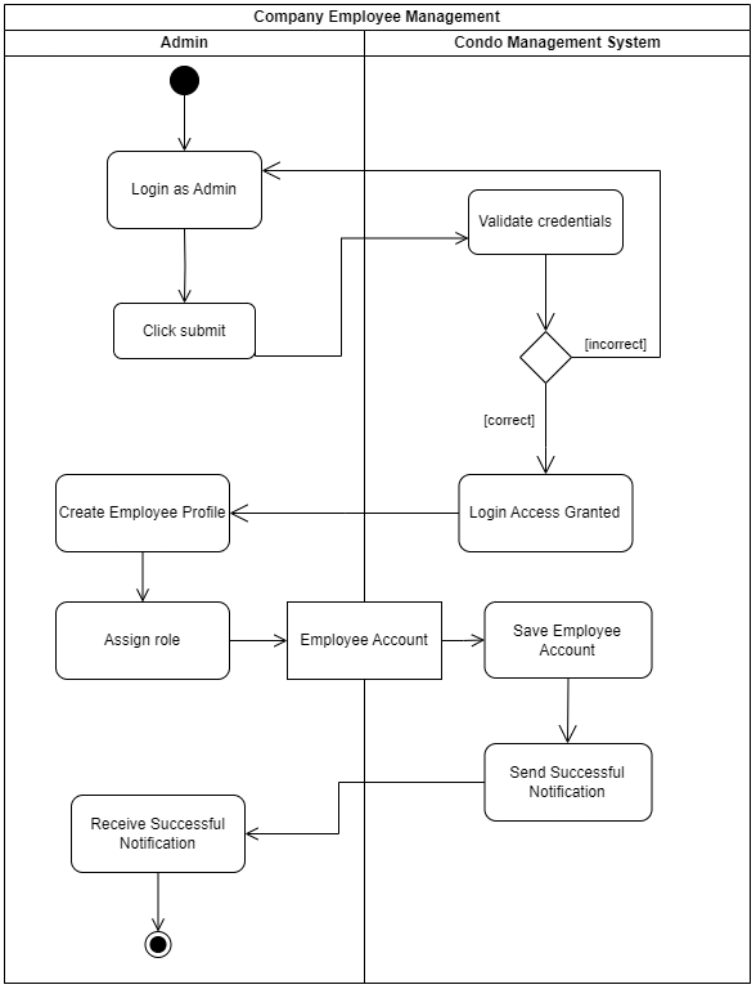

Figure 7: Activity Diagram for Company Employee Management

4.2.1.2 Sequence Diagram

4.2.1.2.1 Update user profile

This diagram sequentially represents the calls to modify the user's info on the profile page. The calls are made from the User to the UserController and Database.

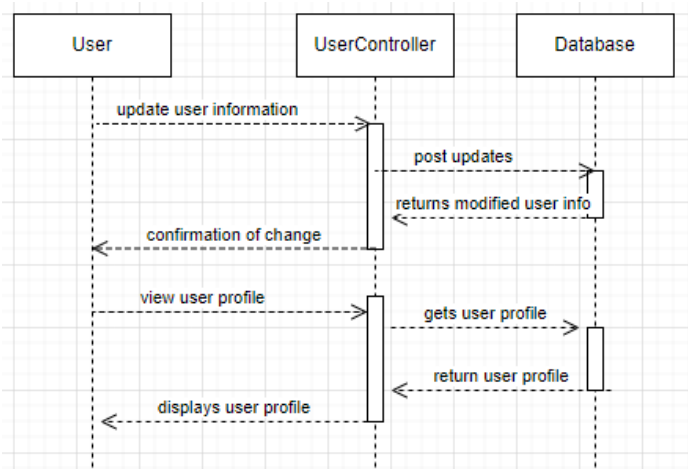

Figure 8: Updating existing user profile

• Step 1: The client sends a request to UserController to modify the profile.

• Step 2: ProfileController updates user details in the database. If successful, the database returns the modified data to the ProfileController

• Step 3: ProfileController sends confirmation of update success back to the user.

4.2.1.2.2 Simplified reservation system

This diagram sequentially illustrates the calls to reserve a service or an amenity based on the available slots. The calls are made from the User to the AppointmentController and Database.

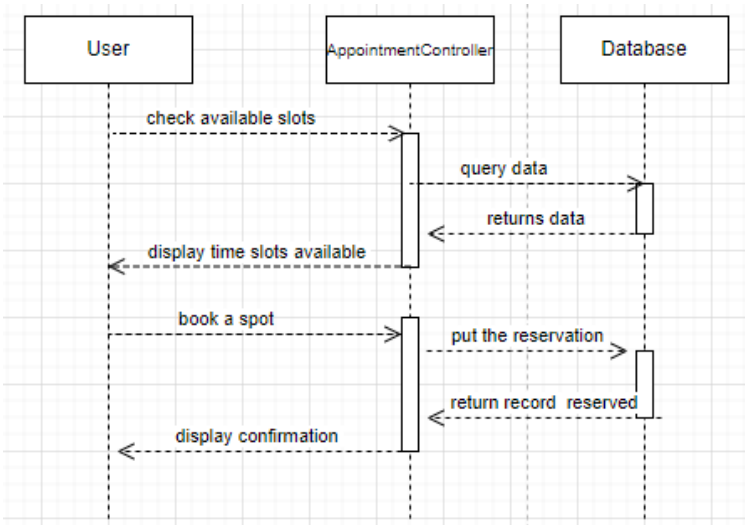

Figure 9: Simplified reservation system

Preconditions: User must be logged in to his/her profile.

• Step 1: The user sends a request to AppointmentController to check for available slots for the service for that given week(ex: spa, fitness lounge)

• Step 2: AppointmentController sends a query to the database and fetches all the available time slots for that service for the selected week. After retrieving the available time slots, the user will be shown the different time slots available.

• Step 3: The user will then select one of the timeslots displayed, the appointment controller will then post an update to the database to set the time slot as taken with the user ID associated with the timeslot.

• Step 4: The appointment controller waits for the database return and sends a confirmation to the client of the booking time and service chosen.

4.2.1.2.3 Simplified financial system

This diagram sequentially depicts the calls to calculate the predicted revenue of a property to generate an annual report. The calls are made from the CompanyAdmin to the FinanceController and to the Database.

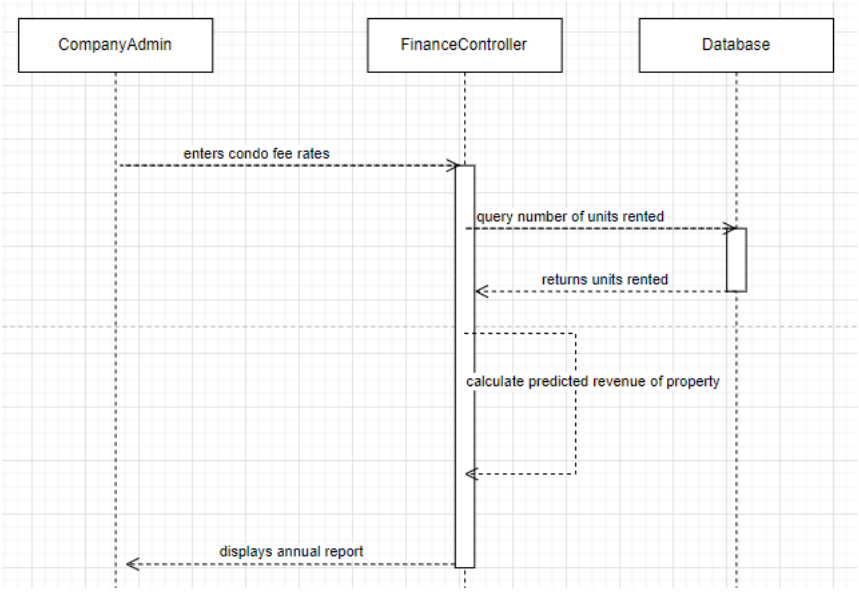

Figure 10: Simplified financial system

Preconditions: Company admin must be logged in to his/her profile.

• Step 1: The user with authorized access enters the rate of their condo fee based on the square footage of the property, with extra charges for amenities(ex: parking spot)

• Step 2: FinanceController sends a query to the database and fetches all units and amenities rented at that time along with their square footage.

• Step 3: The FinanceController will then calculate the predicted revenue based on the rate entered by the admin and the state of the rentals at that moment. The FinanceController then displays the estimated revenue to the company admin.

4.2.2 Known Issues with View

Some known issues that may arise in this view in the context of the Condo Management System, include incomplete activities, ambiguous flow and overly complex diagrams. These diagrams may lack items from the stakeholder requirements, leading to inaccuracies in depicting the system's behaviour.

4.3 View: Physical

4.3.1 Models

4.3.1.1 Deployment Diagram

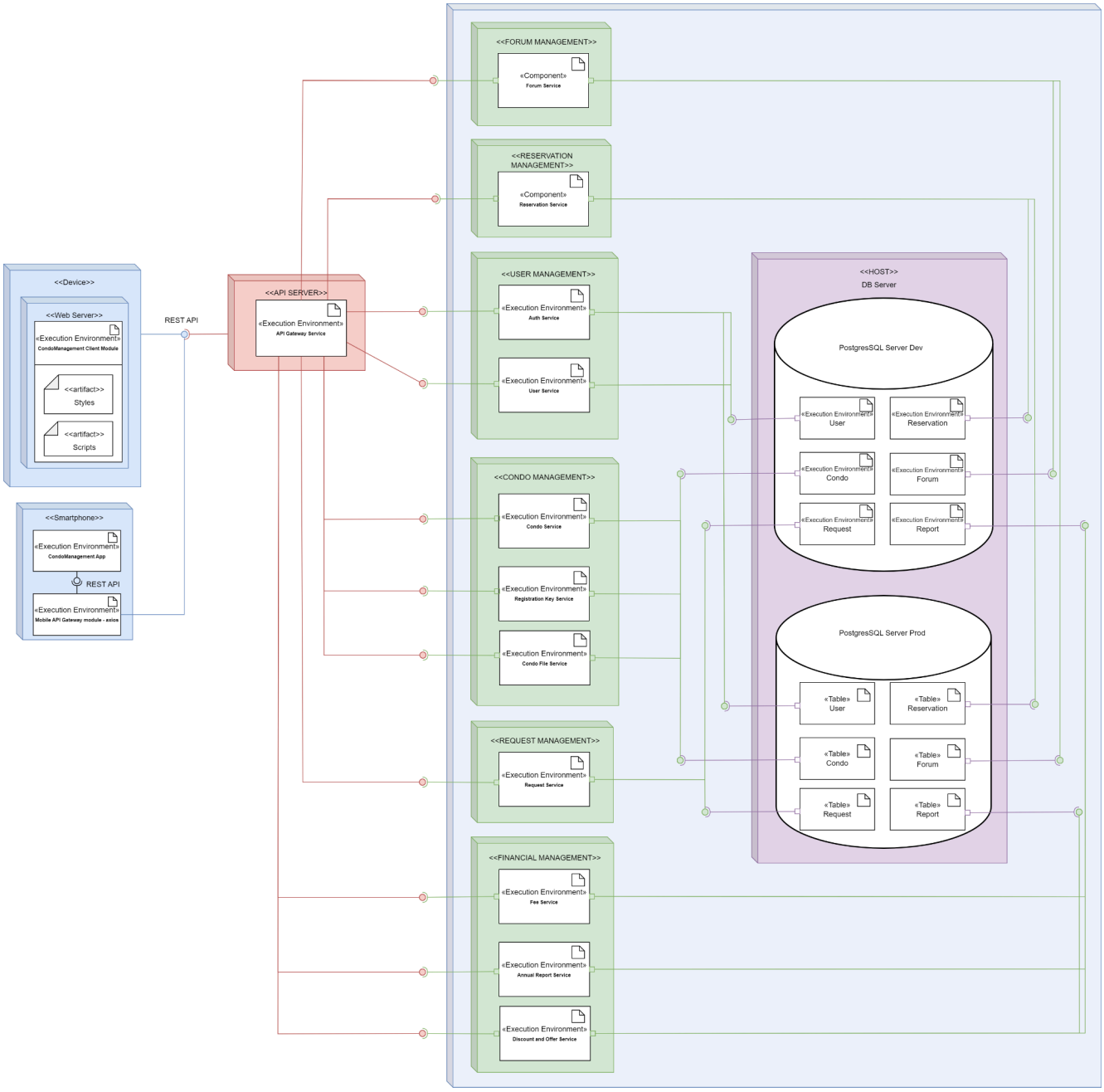

Figure 11: Deployment Diagram of the Condo Management System

The deployment diagram defines all our microservices and all the containers and tables in our server. It represents how our view makes API calls to the controller layer, which is hosted on a physical server. The server contains two independent environments: dev and prod.

4.3.2 Known Issues with View

Even though this view addresses scalability and resource allocation, it's challenging to well align diagrams with real-world infrastructure. Hence, it's complicated to plan solutions and convey those ideas in the diagrams. As the system scales, the diagrams change because of evolving connections between system layers. Similarly, it's hard to accurately represent network configurations and communication protocols.

4.4 View: Development

4.4.1 Models

4.4.1.1 Component Diagram

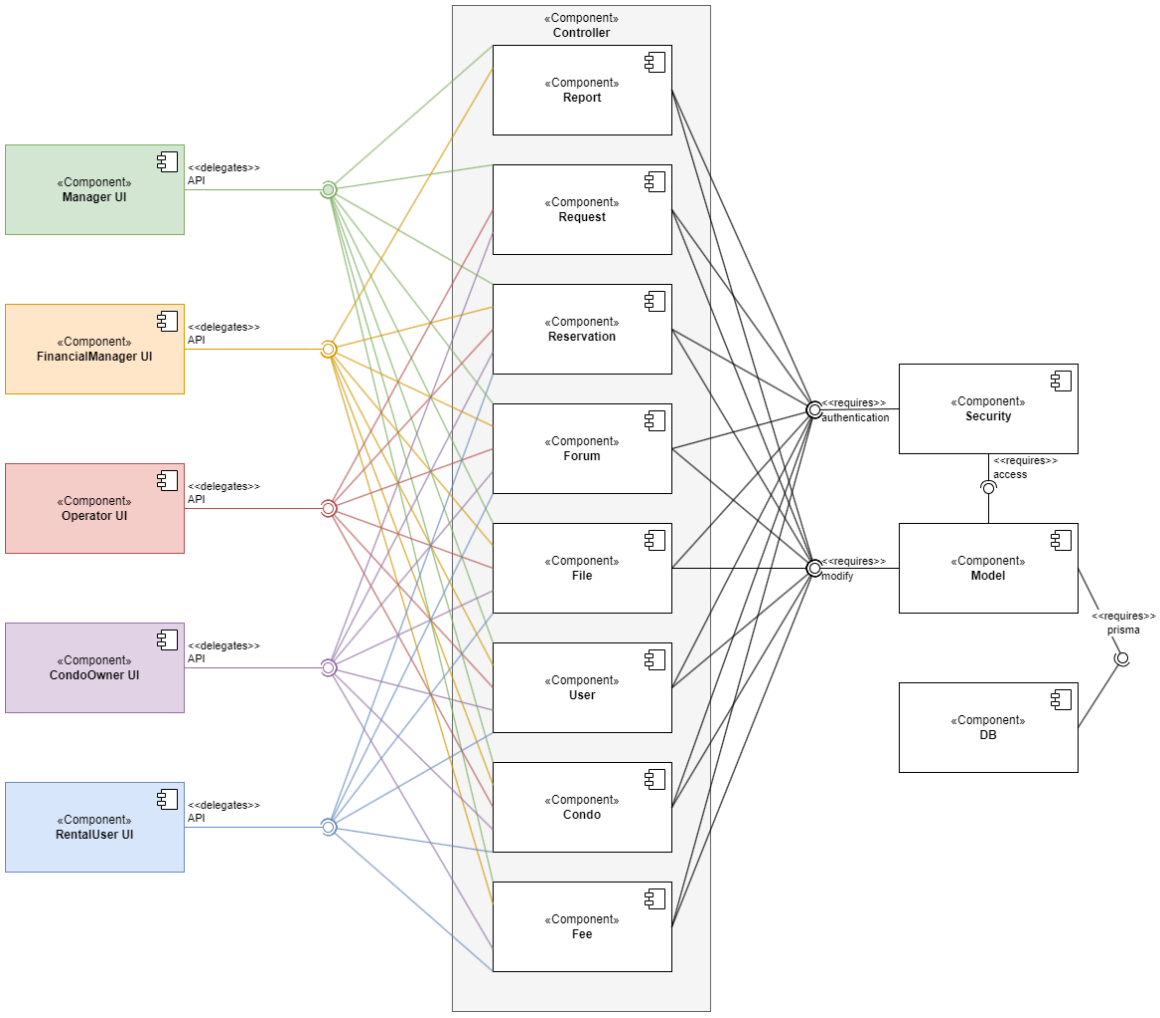

Figure 12: Component Diagram of the Condo Management System

Each UI component is connected to all the controller components it has access to, which varies depending on the user's access rights. Each controller components require authentication from the security component. They also require modification from the model component, which requires access to the security component. Moreover, the model component is required to access the database (DB) component with Prisma.

4.4.2 Known Issues with View

Lack of documentation and varying representations of component dependencies result in vague conclusions. Diagrams might show components interacting in a certain way, but they may not work well in real life, possibly due to problems with how information is passed through different parts.

4.5 View: Use Case

4.5.1 Models

4.5.1.1 Use Case Diagram

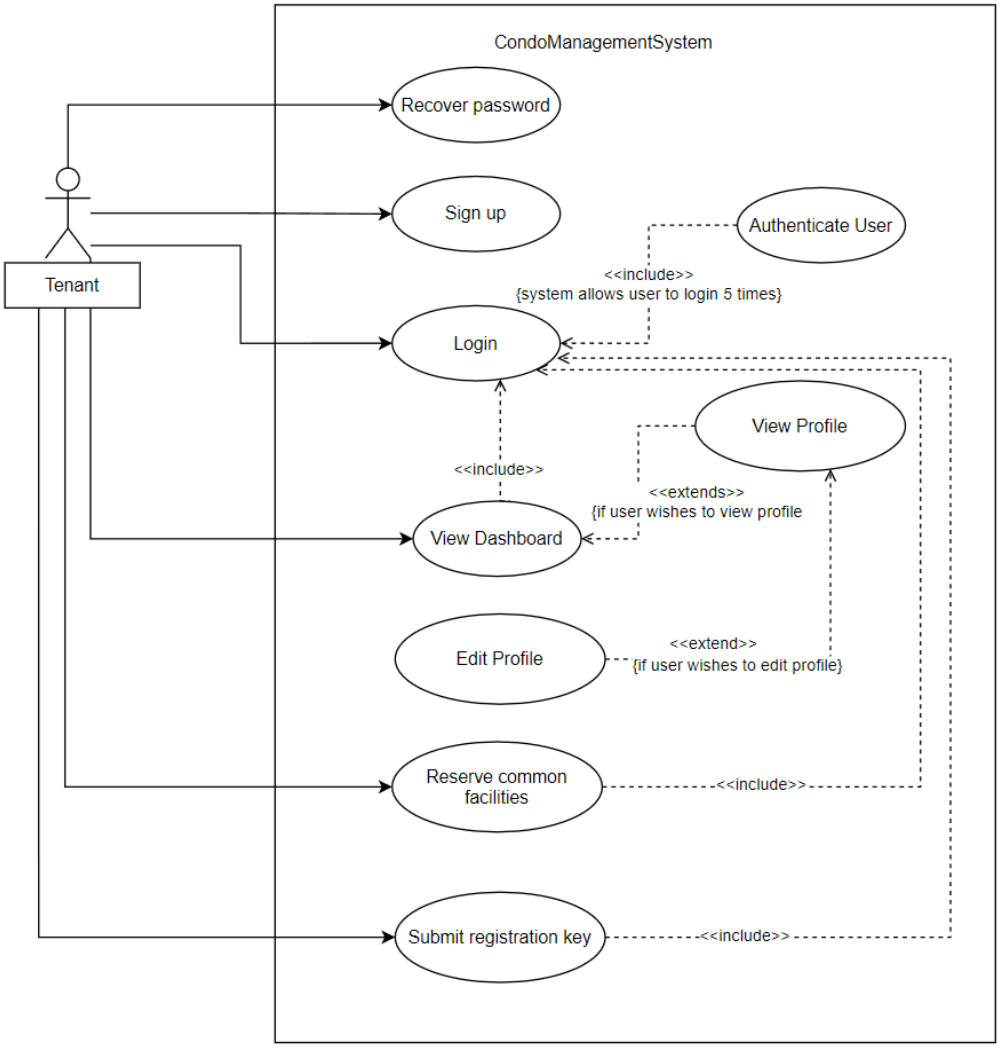

Figure 13: Use case diagram of the Condo Management System from a tenant's point of view

Tenants can create an account, then log in or recover their password if they forget. Login is associated with authentication, which is needed for the other functionalities: accessing their profile and submitting a registration key to link their account to their condo. Once registered, they can view their dashboard and reserve common facilities. Tenants have the same functionality as condo owners, except that they can't submit requests.

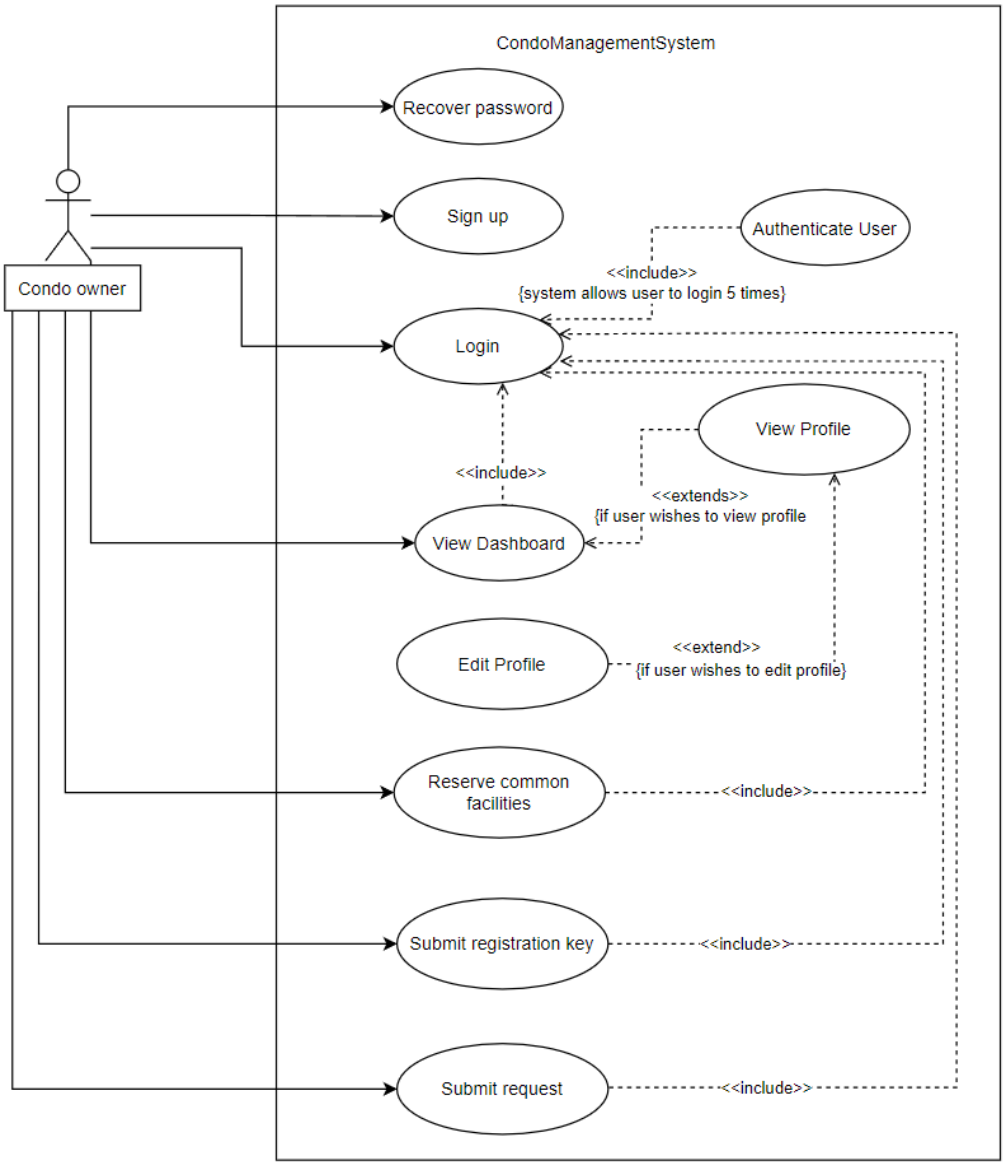

Figure 14: Use case diagram of the Condo Management System from a condo owner's point of view

Condo owners, unlike tenants, can submit requests. They can authenticate themselves and use a registration key to link their account with their condo. They have access to the facilities and a dashboard and can submit requests if they want to.

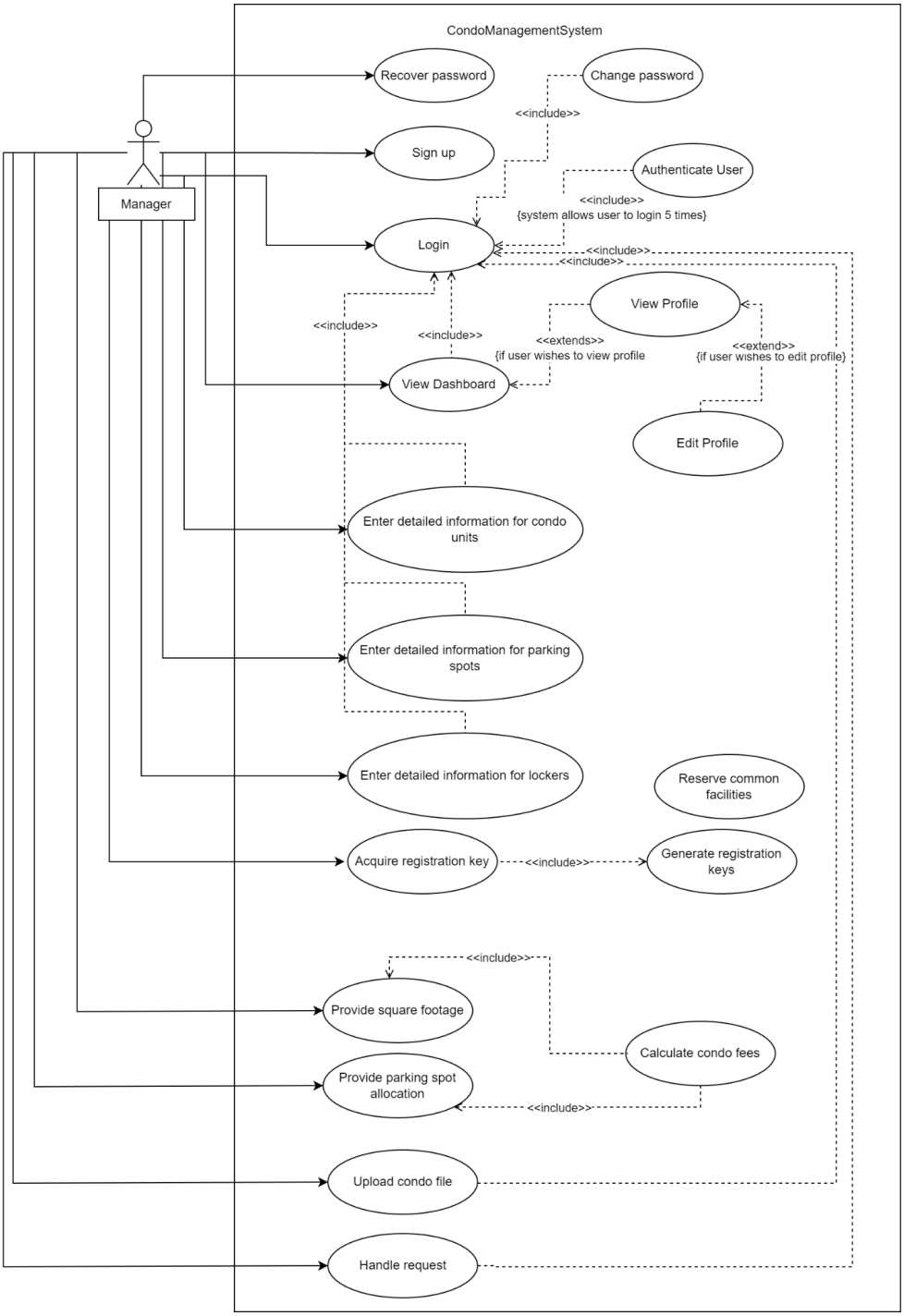

Figure 15: Use case diagram of the Condo Management System from a manager's point of view

Managers can view their profile through their dashboard after successfully logging in. Their dashboard is a hub to access all their information: edit their profile, access their property list, view active requests, upload condo files, and view financial reports. When selecting a single property, they can enter the information for condo units, parking spots and lockers.

4.5.2 Known Issues with View

Use cases can overlook edge cases and exceptions. Due to incomplete coverage of user interactions, the system can produce errors or inconsistent and vague responses. Furthermore, if there is a large number of use cases, it's challenging to have the same style of response in every scenario. When responses vary, it leads to a worse user experience. Lastly, the use case view fails to address potential conflicts between different user roles and their respective interactions with the system. For example, a tenant may want to pay the fee shown on their screen while the manager increases the fee on their side at the same time.

5 Consistency and correspondences

5.1 Known inconsistencies

Inconsistencies between the use case diagram and the activity diagram:

> Inconsistencies may arise between the interactions in the use case diagram and the detailed activities in the activity diagrams. This discrepancy results from the different levels of abstraction between the diagrams. Focusing on the external view, use case diagrams depict key high-level interactions between actors and the system. On the other hand, activity diagrams show the precise flow of activities within a particular use case. Stakeholders have varying priorities and technical expertise; hence they may understand requirements or design decisions differently. Resolving inconsistencies requires more negotiation and collaboration between stakeholders, which isn't always feasible.

Inconsistencies between the development view and the physical view:

> The development view describes the organization of software components such as modules, subsystems, and their dependencies. This view captures concerns related to software development and maintenance. On the opposite side, the physical view mainly focuses on the connections between software components and physical hardware, addressing concerns such as deployment, scalability, and performance. Inconsistencies can arise when the real-life deployment differs from the planned development view, increasing the risk that the software doesn't operate as expected.

Inconsistencies between the physical view and the process view:

> Inconsistencies may occur between the physical view and the process view when the runtime system is limited by physical constraints such as network latency or resource limitations. To realistically achieve the expected performance, we might have to adjust the process view, such as optimizing communication protocols, load balancing, or fault tolerance mechanisms.

Inconsistencies between the logical view and the process view:

> The logical view emphasizes the system's static structure (components, modules and relationships). In contrast, the process view highlights its dynamic behaviour, illustrating the runtime interactions among components. Inconsistencies may arise on how the logical structure aligns with the runtime behaviour, especially when changes happen in one view but not in the other.

Inconsistencies between the logical view and the physical view:

> A common inconsistency between these views is translating the logical designs into physical infrastructure. For instance, a logical view can portray a distributed system architecture, while the physical view can reveal limitations in the deployment environment. In complex

systems, it's challenging to ensure consistency across all views and diagrams, especially with frequent changes. As the system progresses, new requirements and constraints may come up, making it harder to maintain consistency.

5.2 Correspondences in the AD

Correspondences between the logical view and the development view:

The logical view and the development view associate the system's logical architecture and its organization into development modules, subsystems, or components. Changes in one view should be reflected in the other, thus ensuring that the system's structure stays aligned with its planned architecture during development.

Correspondences between the logical view and the physical view:

The logical view and the use-case view are related to the system's functionality and its logical components or modules. This ensures that the logical architecture supports the functionality required by the identified use cases and that changes in use cases are reflected appropriately in the logical structure of the system.

Correspondences between the process view and the physical view:

The runtime behaviour of the system must align with its physical configuration and constraints. This is ensured by the correlation between the process and physical views, where the processes and communication among components map to their physical deployment.

5.3 Correspondence rules

For the separation of concerns, our diagrams are well-defined and don't take elements from other views. For example, our class diagram focuses on our backend structure, which isn't present at all in the process view. Similarly, the separation of concerns of our controller is identified in the physical view, under the context of API calls, and the development view, under the context of their connections to the different views important to consider during development.

For completeness, all relevant architectural elements and relationships are represented in their corresponding diagrams, allowing the understanding of the system from an architectural point of view.

For contextual integration, as we are operating in an agile methodology, iterative cycles of development allow us to make incremental improvements to our web application, while making time for frequent feedback. This has allowed us to continuously adapt our design to the initial requirements of our stakeholders.

Lastly, using the principles of modularity and reusability, the website is designed as a collection of reusable modules used in all the pages of our website; this facilitates the scalability of the project, as well as regulating the format of our code.

The template ends here!

Bibliography

[1] Paul C. Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. Documenting Software Architectures: views and beyond. Addison Wesley, 2nd edition, 2010.

[2] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: a framework for integrating multiple perspectives in system development. International Journal of Software Engineering and Knowledge Engineering, 2(1):31–57, March 1992.

[3] IEEE Std 1471, IEEE Recommended Practice for Architectural Description of Software-Intensive Systems, October 2000.

[4] ISO/IEC/IEEE 42010, Systems and software engineering — Architecture description, December 2011.

[5] Alexander Ran. Ares conceptual framework for software architecture. In M. Jazayeri, A. Ran, and F. van der Linden, editors, Software Architecture for Product Families Principles and Practice, pages 1–29. Addison-Wesley, 2000.

[6] Nick Rozanski and Eo´in Woods. Software Systems Architecture: Working With Stakeholders Using Viewpoints and Perspectives. Addison Wesley, 2nd edition, 2011.

[7] Uwe van Heesch, Paris Avgeriou, and Rich Hilliard. A documentation framework for architecture decisions. The Journal of Systems & Software, 85(4):795–820, April 2012.