Introduction

For this coursework, my primary responsibility was the development of the Society Leader **subsystem**, and I contributed to the project by producing all UML models and architectural artefacts related to this part of the system. At the beginning of the project, our group held several meetings to establish responsibilities, define internal deadlines, and make sure that the design of each subsystem aligned with the overall architecture of the United Student Unions application. During these discussions, I clarified my focus areas, which included analysing the functional requirements associated with the Society Leader role and identifying the key features that required modelling—such as event creation and management, updating society information, handling membership options, and communicating with members.

Throughout the project, I created the Use Case Diagram, Activity Diagram, and Component Diagram for my subsystem. I actively collaborated with my team to maintain consistency in naming conventions, subsystem boundaries, and architectural assumptions. This coordination was particularly important when defining shared interfaces and interactions across subsystems, ensuring that my models integrated correctly into the team's overall system design. These collaborative efforts helped us maintain a coherent architecture and avoid conflicts between individually developed models.

One of the main challenges I encountered early in the coursework was understanding how the different tasks connected, as the overall assignment initially felt broad and overwhelming. To overcome this difficulty, I revisited the lecture materials from the beginning of the module, which significantly improved my understanding of UML modelling principles and the expectations for each task. Another challenge involved working with Papyrus, as I experienced several technical issues, such as layout misalignment, difficulties with connectors, and formatting inconsistencies. Through trial, practice, and repeated verification of UML rules, I was able to produce diagrams that were clear, structured, and compliant with the required notation.

Overall, this coursework substantially improved my understanding of model-driven software engineering, particularly in relation to UML diagrams, microservice-oriented architectural design, and the integration of subsystem-level models into a unified system. It also strengthened my ability to read specifications carefully, solve modelling problems independently, and collaborate effectively within a software engineering team.

Task 1 -- Quality Requirements for the Society Leader App (1. Specification of Quality Requirement)

-Selected Functional Requirement

For this task, I selected FR-SL-2: Management of Events as the functional requirement for my subsystem. This requirement explains what the Society Leader App must allow society leaders to do when organising events. Based on the case study, society leaders should be able to create new events, update existing ones, and provide important details such as the event title, theme, date and time, venue, participation rules, and any promotional materials like photos, videos, or posters. When an event is updated, the new information should appear immediately across the USU system so that students who follow the event receive the latest updates without delay.

Since event management is one of the core features of the Society Leader App, the subsystem needs to satisfy several quality requirements. These include aspects of **s**ecurity, performance, reliability, and scalability. Meeting these quality attributes is important because events involve many users across different universities, and the system must operate safely, respond quickly, remain stable, and be able to support increasing numbers of events and participants.

-Security and Privacy Requirements

Security Requirement 1 – Authentication
This ensures that only authorised society leaders can manage events, reducing the risk of data manipulation. Only verified society leaders such as the Brookes Unıon Leader, should be able to access event management features. Users must log in with their official USU student account, and the login process should use encrypted communication to prevent credential theft. This ensures that unauthorised individuals cannot create or modify events on behalf of a society.

Security Requirement 2 Role-based Permissions
The system should enforce strict role checks before allowing any event-related action. Only the leader of a specific society, for example vice president of Turkish Society, should have permission to create, edit, or delete events for that society. Normal members, students from other societies, or general users must not be able to access these functions. This helps protect events from accidental or intentional misuse.

Security Requirement 3: Protection of Personal Data

Event pages must not display personal information such as phone numbers or email addresses of society leaders or members. Any communication should take place through the internal messaging system so that private details remain hidden. This reduces the risk of unwanted contact, identity exposure, or data leaks both inside and outside the university environment. This is very important for security reasons, especially nowadays

Security Requirement 4: Safe Handling of Uploaded Files
Since leaders can upload images, videos, or posters for promotion, the system should validate all files before saving them. This includes checking file type, size limits, and running a basic malware scan. Uploaded files should be stored in a secure cloud location where access is controlled, rather than in publicly accessible folders. This prevents harmful files from entering the system and protects media content from being shared without permission.

Security Requirement 5: Audit Logging
Every important action, such as creating an event, updating details, or changing promotional materials, should be recorded in an audit log. These records help administrators track suspicious activity, understand how errors occurred, and maintain the reliability of the system. Audit logs also support accountability if misuse or security incidents arise.

- Performance Requirements

Performance Requirement 1 Fast Event Creation
When a society leader submits a new event, the system should process the request and save the event data within roughly two seconds under normal conditions. This includes writing the event information to the database and returning a confirmation message or updated interface view. Quick event creation is important because society leaders often need to make several edits or adjustments, and long delays can interrupt their workflow or cause confusion about whether the event was saved correctly.

Performance Requirement 2: Efficient Loading of Events
The list of events for a society should load within about three seconds, even if the society has created many events over time. To support this, the system should optimise how it retrieves data by loading essential details first, such as the event title, date, and status, while additional information like long descriptions or images can load afterwards. This helps society leaders quickly move between different events and manage their activities without waiting for unnecessary data to load.

Performance Requirement 3: Smooth Media Uploads
The system should support uploading promotional images and videos up to approximately 10MB without causing the interface to freeze or slow down. While a file is being uploaded, the user interface should remain responsive so that the society leader can continue editing other event details. Once the upload is complete, the app should provide a clear confirmation message within a few seconds. This ensures that leaders can attach media quickly and avoid uncertainty about whether the file was successfully uploaded.

Performance Requirement 4: Low Notification Delay
Whenever an event is created or updated, the system should trigger notifications for all subscribed students with minimal delay. Ideally, these notifications should be sent out within about ten seconds so that students receive the updated information in near real time. This is important because event changes often affect schedules, participation decisions, and planning, so students need timely and accurate updates to stay informed.

Performance Requirement 5: Offline Drafting
If the society leader temporarily loses internet connection, such as when moving between locations on campus, the app should still allow them to enter or edit event details. The information should be saved locally on the device and uploaded to the server automatically once the connection is restored. This ensures that event creation is not interrupted and prevents leaders from losing their work due to short network outages.

Reliability Requirements

Reliability Requirement 1: High Availability
The event management service should be available almost all the time, with an uptime of around 99.5%. Since society leaders may need to create or edit events at any hour, even outside regular class times, the system must stay accessible whenever they log in. Occasional maintenance is acceptable, but it should not interrupt normal usage because event deadlines and announcements can be time-sensitive.

Reliability Requirement 2: Consistent Data Updates
Whenever an event is updated, the system should save all changes together as one complete operation. This avoids situations where only part of the event is updated, such as the title changing but the description staying old. Inconsistent updates can easily confuse students or lead to misunderstandings about the event details. Using atomic updates helps ensure that everyone always sees the correct and complete information.

Reliability Requirement 3: Recovery from Errors
If something goes wrong while a leader is creating or editing an event, for example, if the Wi-Fi drops suddenly, the information they have already typed should not disappear. The app should temporarily store the entered details and let the user retry once the connection becomes stable again. This is especially useful on campus, where signals can fluctuate in busy areas, and it prevents frustration from losing work.

Reliability Requirement 4: Backup of Event Data
All event-related data should be included in the regular system backups, which are usually done on the server side. This means that if the system experiences an unexpected failure or if data is accidentally removed, the event can still be restored from the backup. Even though individual leaders might not notice this happening in the background, it is important for keeping the system dependable in the long run.

Reliability Requirement 5: Consistent Synchronisation Across Devices
If a society leader updates an event using one device, such as their phone, the updated information should appear on their other devices like a laptop or tablet within a short time. This prevents situations where a leader sees old information on one device and newer information on another. Consistent synchronisation helps maintain trust in the system and makes it easier for leaders who switch between devices during the day.

- Scalability Requirements

Scalability Requirement 1: Growth in Number of Societies
The system should be able to support more and more societies as the USU expands to different universities. Even if the total number of societies becomes very large, the event management subsystem should still work smoothly and not slow down. This is important because new societies are constantly being created, and the system must keep pace without requiring a redesign every year.

Scalability Requirement 2: High Numbers of Concurrent Users
During busy times, many society leaders across the country might be creating or editing events at the same moment, for example before the start of a new term. The subsystem should be able to handle thousands of these actions happening at once without becoming too slow or unresponsive. Because they might be editing videos or contents for students. Leaders should not notice any difference in performance even when the system is under heavy use.

Scalability Requirement 3: Large Amounts of Event Data
As the USU system grows older, it may end up storing tens of thousands of events from past and future societies. The event management microservice should be designed so that searching for events, opening an event page, or updating event details remains fast, even when the database becomes very large. This helps the system stay usable in the long term, rather than becoming slower every year. With this approach, students can benefit from the system securely and quickly.

Scalability Requirement 4 Notification System Capacity
The system should be able to send a large number of notifications, especially during busy times like sports seasons or university festivals when many societies are active. The notification service should automatically scale so that messages are delivered quickly, and students do not experience delays. Even if thousands of notifications are sent at once, the system should still keep up.

Scalability Requirement 5 Elastic Storage for Media Files
Because society leaders can upload images, videos and posters, the amount of stored media will increase over time. The cloud storage used by the subsystem should grow automatically when more files are uploaded, without requiring manual changes. This makes sure that leaders can always upload their media files without running into storage problems.

2.1 & 2.2-- UML Modelling for the Society Leader Subsystem

Introduction

This section presents the UML models developed for the Society Leader subsystem, focusing on both the Use Case Diagram and the Activity Diagram. Together, these models illustrate the key functionalities offered by the subsystem and clarify how they support the responsibilities of a Society Leader within the wider USU system. By modelling the interactions and internal processes, these diagrams help break down the subsystem into understandable components and show how users engage with its features in practice.

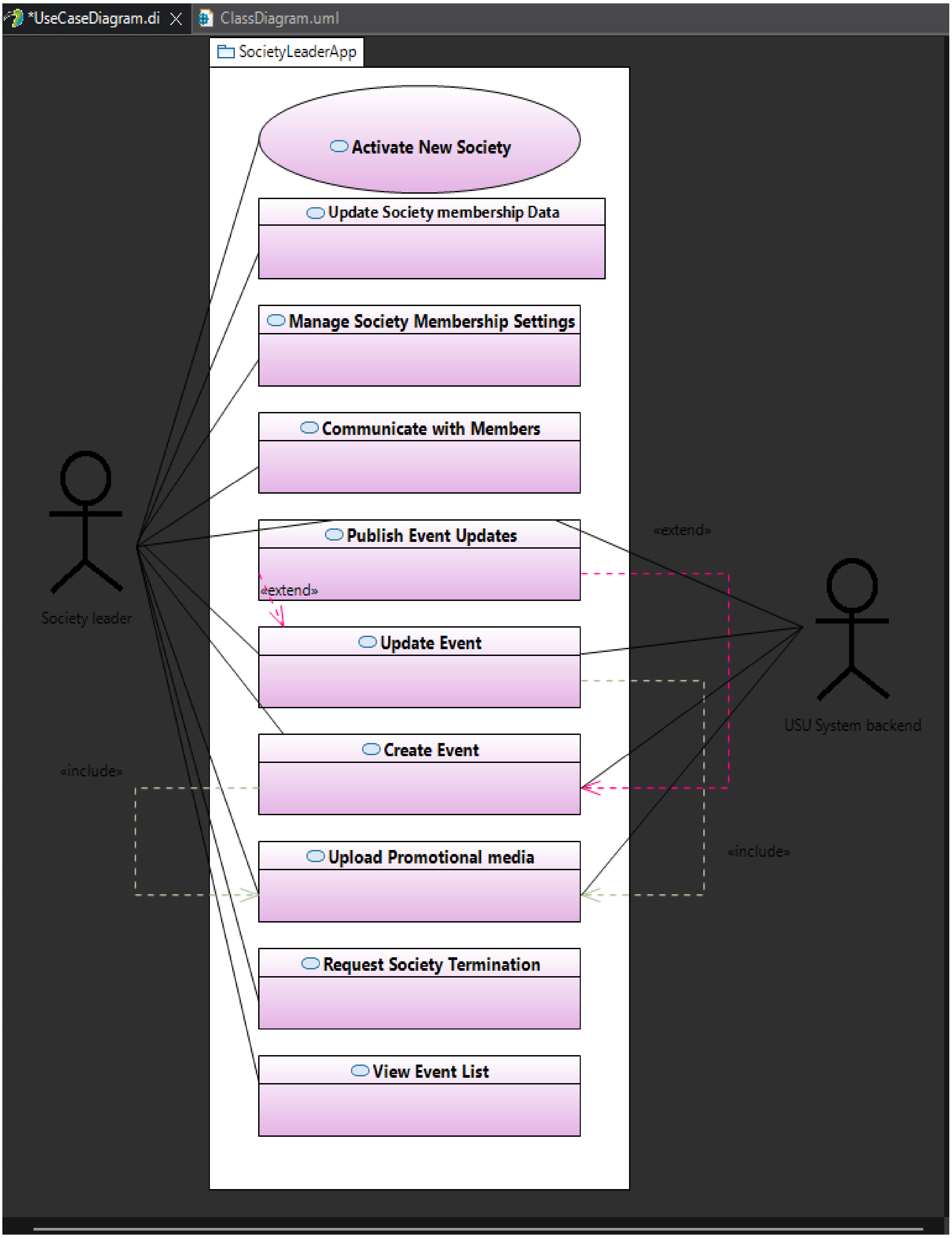
The Use Case Diagram provides a high-level overview of all interactions between the Society Leader and the system. It identifies the main capabilities available to the leader, such as creating and managing events, updating society details and communicating with members. In addition, the diagram highlights which areas rely on backend services or connections to other subsystems. This helps demonstrate the boundaries of the Society Leader subsystem and shows how it fits into the overall system architecture.

The Activity Diagram then focuses on one specific capability in more depth: the event creation process. It models how the task progresses step by step, starting from when the leader begins entering event details and continuing through validation, media uploads and the final confirmation. The diagram also shows the transitions between user actions and automatic system responses, making the workflow clearer and more predictable. This is useful for understanding how errors are handled, how data flows through the system and what checks occur behind the scenes.

Overall, these UML models provide a structured and coherent understanding of how the Society Leader subsystem operates. They help explain not only what the subsystem can do, but also how and when each function is carried out. By visualising the processes in this way, the models make the design easier to evaluate, refine and integrate with the rest of the USU system. If improvements or additional features are needed later on, these diagrams offer a solid foundation to build upon.

Use Case Diagram Explanation (2.1 Use case model)

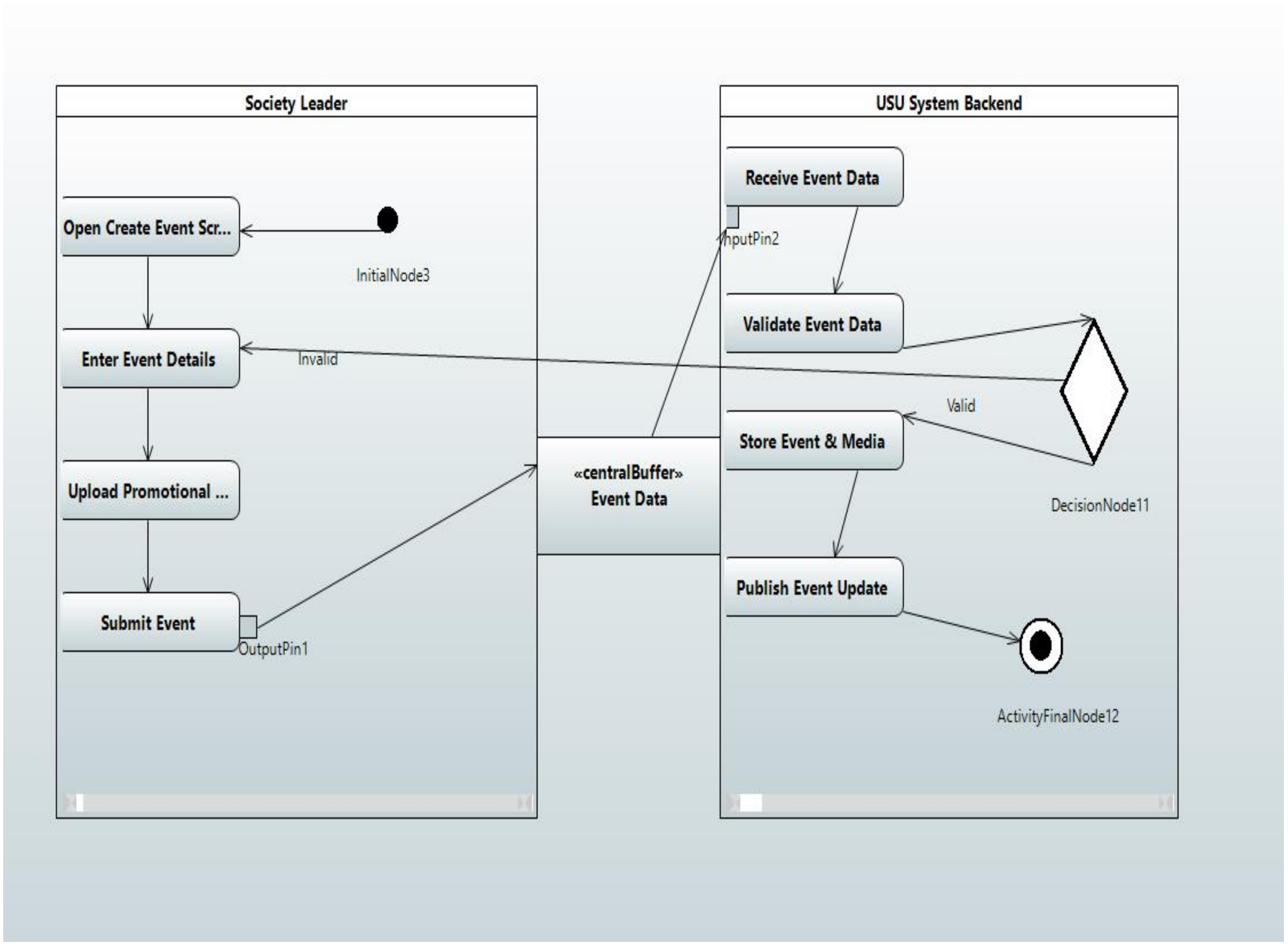
The Use Case Diagram outlines the main functionalities available to a Society Leader, including creating and updating events, managing society information, communicating with members, and submitting a termination request. It also identifies the USU System Backend as a supporting actor for tasks that require system-level processing, such as validating event information, storing updates in the central database, and sending notifications to students. The diagram uses «include» relationships to show actions that must always occur as part of a larger use case, for example uploading promotional media during event creation, which is an essential step in completing the process. In contrast, «extend» relationships represent behaviours that only happen under certain conditions, such as publishing updates once changes are finalised or when the leader chooses to notify members. By presenting these interactions in a structured way, the diagram offers a clear view of how the Society Leader communicates with the subsystem and how backend services support each action. This helps define the boundaries of the subsystem, shows how related tasks connect to each other, and clarifies the overall role of the Society Leader within the wider USU system.


*UseCaseDiagram.di
ClassDiagram.uml
SocietyLeaderApp
Activate New Society
Update Society membership Data
Manage Society Membership Settings
Communicate with Members
Publish Event Updates
«extend»
Update Event
Create Event
Upload Promotional media
Request Society Termination
View Event List
Society leader
«include»
«extend»
USU System backend
«include»

Activity Diagram Explanation (2.2)

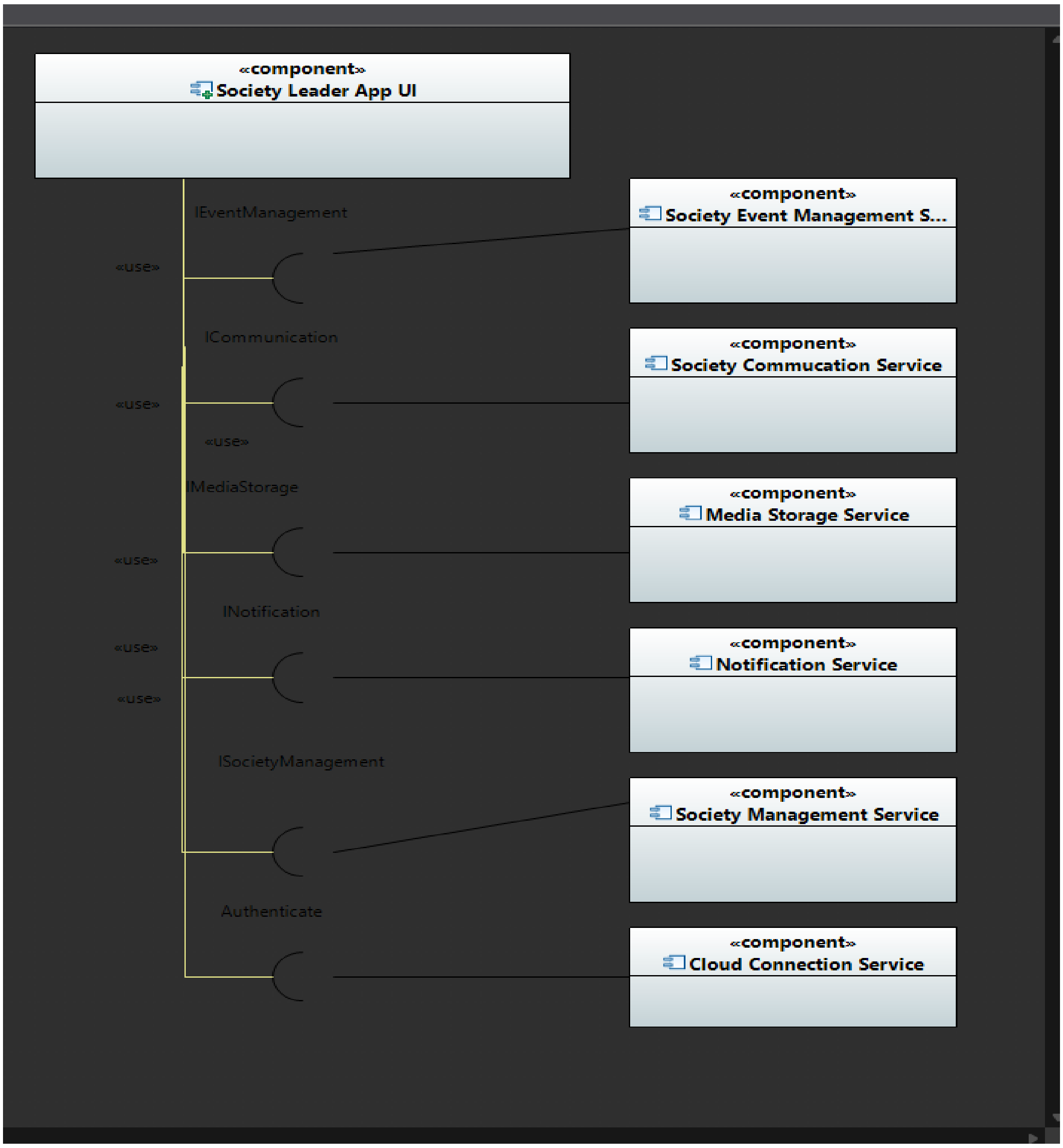
The activity diagram models the workflow for creating or updating an event in the Society Leader subsystem. The process begins with the Society Leader opening the event creation screen, entering the required details, optionally uploading promotional media, and then submitting the event. Once submitted, the event information is passed to the backend as an EventData object, representing the data flow from the user to the system. The backend first receives the data and then performs validation. A decision node determines whether the submission is valid or invalid. Invalid submissions return the flow to the Society Leader for correction, while valid submissions continue to Store Event & Media.

After storing the event, the backend publishes the update, completing the workflow. Overall, the diagram clearly shows how control and data flow between the user and backend, and how validation governs the progression of the process.



Task 4(a) — Subsystem Architecture Description

The architecture of the Society Leader subsystem is based on a microservices-oriented component design, which helps keep responsibilities clearly separated while supporting scalability and loose coupling. The subsystem includes a single frontend component, the Society Leader App UI, and five backend microservices that each provide their own interface to define the operations available to other components. The App UI communicates with the backend only through these interfaces, which keep the presentation layer independent from the underlying system logic. Through interfaces such as ICommunication, ISocietyManagement, IEventManagement, IMediaStorage, and INotification, the UI can perform the key functions required by Society Leaders, including creating and managing events, updating society information, uploading promotional media, and sending. The Society Management Service, which implements the ISocietyManagement interface,



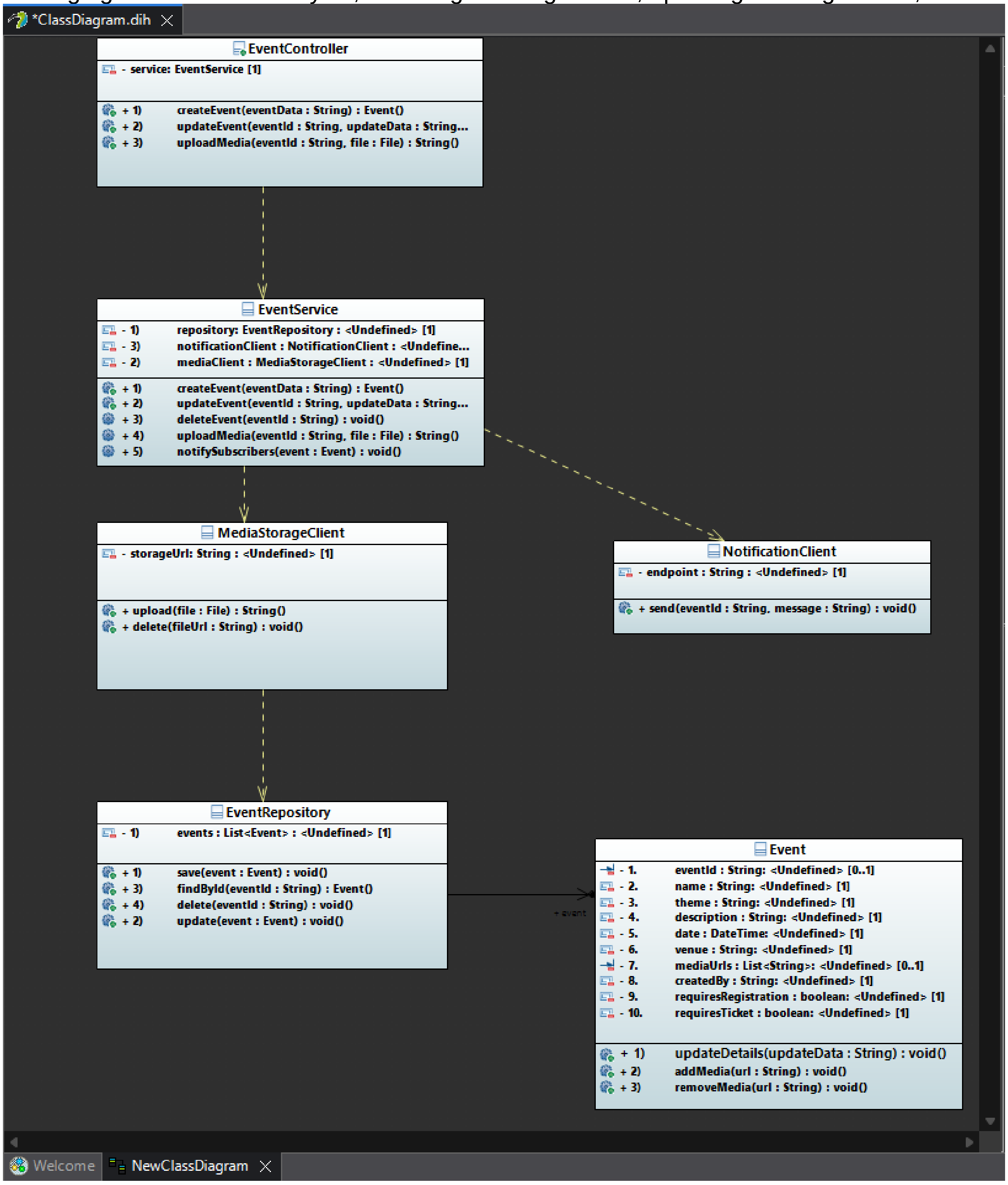
provides the key operations for updating society information, adjusting membership settings, and handling leader handovers.

The Society Event Management Service, which implements *IEventManagement*, oversees the entire lifecycle of events, including creating new events and updating existing ones. As part of this workflow, it interacts with the *INotification* and *IMediaStorage* interfaces to publish event updates and handle the upload of promotional media when needed. Alongside this service, the Society Communication Service implements *ICommunication* and provides messaging capabilities for both direct communication and broadcast announcements to society members. The Media Storage Service, which exposes the *IMediaStorage* interface, is responsible for storing and retrieving all promotional files used across the subsystem, while the Notification Service, implementing *INotification*, ensures that users receive real-time alerts whenever an event changes. Together, these backend components form a modular, loosely coupled microservices architecture in which each service handles a clearly defined functional responsibility and communicates only through well-specified interfaces.

To remain consistent with the team's integrated architectural model, the subsystem also incorporates an Authenticate interface and its implementing component, the Cloud Connection Service. This dedicated service performs essential security tasks such as verifying user identities and checking access permissions before any subsystem operation is carried out. Acting as the secure gateway to the shared cloud environment used across all USU subsystems, the Cloud Connection Service centralises authentication logic, reduces duplication, and strengthens overall system security. Including this component ensures that the Society Leader subsystem remains aligned with the wider system architecture while maintaining microservices principles such as loose coupling, clear separation of responsibilities, and architectural consistency across the platform.

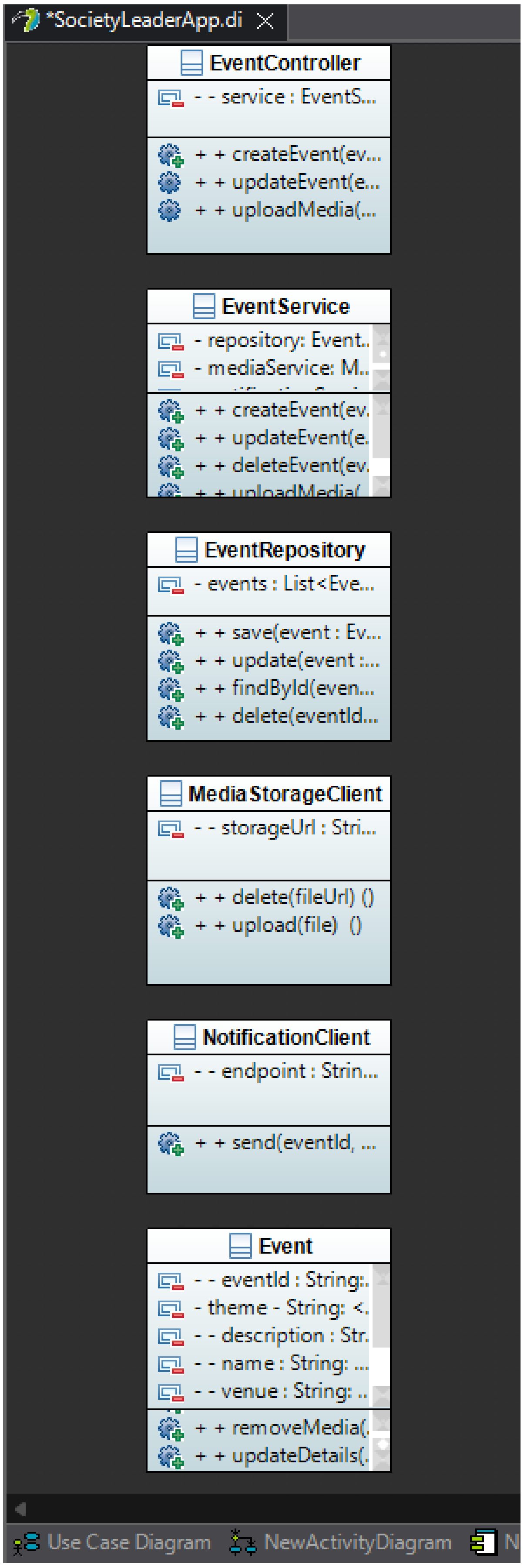
TASK 5 – Class Diagram for the Event Management Component (4.1)

For the Detailed Design stage, the component selected from the Society Leader subsystem architecture is the Society Event Management Service. This microservice is responsible for managing the full event lifecycle, including creating events, updating existing events,


*ClassDiagram.dih
EventController
- service: EventService [1]
+ 1) createEvent(eventData : String) : Event()
+ 2) updateEvent(eventId : String, updateData : String...
+ 3) uploadMedia(eventId : String, file : File) : String()
EventService
- 1) repository: EventRepository : <Undefined> [1]
- 3) notificationClient : NotificationClient : <Undefine...
- 2) mediaClient : MediaStorageClient : <Undefined> [1]
+ 1) createEvent(eventData : String) : Event()
+ 2) updateEvent(eventId : String, updateData : String...
+ 3) deleteEvent(eventId : String) : void()
+ 4) uploadMedia(eventId : String, file : File) : String()
+ 5) notifySubscribers(event : Event) : void()
MediaStorageClient
- storageUrl: String : <Undefined> [1]
+ upload(file : File) : String()
+ delete(fileUrl : String) : void()
NotificationClient
- endpoint : String : <Undefined> [1]
+ send(eventId : String, message : String) : void()
EventRepository
- 1) events : List<Event> : <Undefined> [1]
+ 1) save(event : Event) : void()
+ 3) findById(eventId : String) : Event()
+ 4) delete(eventId : String) : void()
+ 2) update(event : Event) : void()
+ event
Event
- 1. eventId : String: <Undefined> [0..1]
- 2. name : String: <Undefined> [1]
- 3. theme : String: <Undefined> [1]
- 4. description : String: <Undefined> [1]
- 5. date : DateTime: <Undefined> [1]
- 6. venue : String: <Undefined> [1]
- 7. mediaUrls : List<String>: <Undefined> [0..1]
- 8. createdBy : String: <Undefined> [1]
- 9. requiresRegistration : boolean: <Undefined> [1]
- 10. requiresTicket : boolean: <Undefined> [1]
+ 1) updateDetails(updateData : String) : void()
+ 2) addMedia(url : String) : void()
+ 3) removeMedia(url : String) : void()
Welcome
NewClassDiagram

uploading promotional media, and triggering notifications to students. The structural model of this component has been developed using a UML Class Diagram, which defines the internal classes, their attributes, methods, and the relationships that enable the behaviour of the microservice. The design ensures that the component is consistent with the microservice-based architectural model produced in Task 4.The component follows a layered internal structure composed of a controller, service, repository, domain entity, and two external clients. The EventController handles incoming requests from the user interface and acts as the entry point to the component. It delegates all logic to the EventService, which encapsulates the event-related business rules. The service interacts with two external microservices: the MediaStorageClient for handling the upload of promotional images or videos, and the NotificationClient for sending alerts when an event is created or updated. Event data is stored persistently through the EventRepository, which manages the lifecycle of Event entities. This design ensures a clear separation of concerns, supports testability, and closely reflects the microservice dependencies modelled in the subsystem architecture.

In addition to defining the internal structure of the component, the design also follows established object-oriented principles. The separation between the controller, service, repository, external clients, and the Event entity reflects the Single Responsibility Principle, ensuring that each class focuses on a single, well-defined task. This reduces coupling between classes, makes the component easier to test, and supports future extension without major redesign. Overall, the structural model not only represents the behaviour shown in the sequence diagram but also provides a clean and maintainable foundation for real implementation.



Class Details (Attributes and Methods)

EventController

Attributes

1. service : EventService (private)

Methods

1) createEvent(eventData : String) : Event
2) updateEvent(eventId : String, updateData : String) : Event
3) uploadMedia(eventId : String, file : File) : String

EventService

Attributes

1) repository: EventRepository (private)
2) mediaClient : MediaStorageClient (private)
3) notificationClient : NotificationClient (private)

Methods

1) createEvent(eventData : String) : Event
2) updateEvent(eventId : String, updateData : String) : Event
3) deleteEvent(eventId : String) : void
4) uploadMedia(eventId : String, file : File) : String
5) notifySubscribers(event : Event) : void

EventRepository

Attributes

1) events : List<Event> (private)

Methods

1) save(event : Event) : void
2) update(event : Event) : void
3) findById(eventId : String) : Event
4) delete(eventId : String) : void

MediaStorageClient

Attributes

1) -storageUrl: String (private)

Methods

1) -upload(file : File) : String
2) -delete(fileUrl : String) : void

NotificationClient

Attributes

1) -endpoint : String (private)

Methods

1) -send(eventId : String, message : String) : void

Event

Attributes

1. eventId : String
2. name : String
3. theme : String
4. description : String
5. date : DateTime
6. venue : String
7. mediaUrls : List<String>
8. createdBy : String
9. requiresRegistration : boolean
10. requiresTicket : boolean

Methods

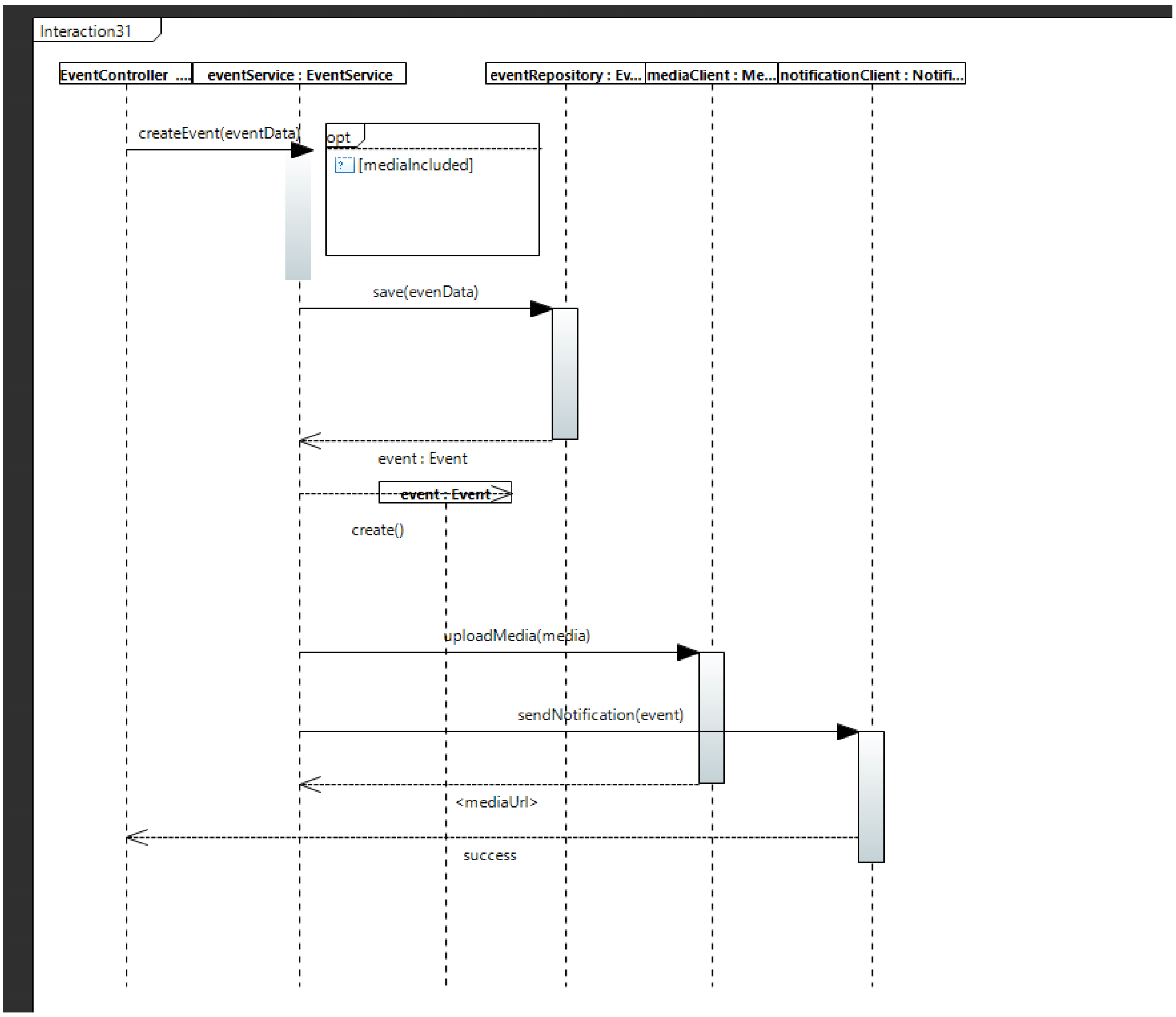
1) updateDetails(updateData : String) : void
2) addMedia(url : String) : void
3) removeMedia(url : String) : void

Overall, the detailed design of the Society Event Management Service provides a clear and coherent structural specification that aligns closely with both the subsystem architecture and the functional workflow modelled earlier. By defining explicit relationships between the controller, service, repository, domain entity, and external clients, the class diagram demonstrates how responsibilities are distributed across the component and how each part collaborates to support event creation, updates, media uploads, and subscriber notifications. The design remains consistent with the microservice-based architectural style by ensuring that all external interactions are handled through dedicated client classes rather than direct dependencies. This separation of concerns not only improves maintainability but also makes the component easier to test and extend in the future. In addition, the encapsulated domain logic within the Event class provides a solid foundation upon which the behavioural model in the sequence diagram can operate. Together, these models form a complete and well-structured representation of the internal workings of the Event Management component, demonstrating a coherent transition from requirements to architecture and finally to detailed object-oriented design.

5(b) Behaviour Model – Sequence Diagram for *Create Event (4.2)*

Overview

This sequence diagram illustrates the internal behaviour of the Society Event Management subsystem during the Create Event use case. It shows how the request from the Society Leader is processed through the controller and service layers, and how the system coordinates data storage, optional media upload, and user notifications. The diagram provides a clear representation of how the subsystem's components collaborate to complete the event creation workflow.



Lifelines (Objects in the Interactional)

Papyrus repeatedly crashed both on the university machines and on my own laptop, sometimes after I had already completed the diagrams, so I am listing the element names here again to ensure they are clear.

The following lifelines appear in the sequence diagram:

- eventController : EventController – receives the request from the Society Leader App UI and forwards it to the backend service.
- eventService : EventService – main microservice that contains the business logic for creating an event.
- eventRepository : EventRepository – data access component that stores and retrieves event data.
- mediaClient : MediaStorageClient – external media storage client used to upload promotional images or videos.
- notificationClient : NotificationClient client used to send notifications about new events to society members.
- Event : Event domain object representing the created event.

These lifelines correspond directly to the classes and components defined in my class diagram and component diagram.

Lifelines in the sequence diagram

The sequence diagram contains six lifelines that represent the key components involved in the event creation process. The eventController : EventController receives the request from the Society Leader App UI and forwards it to the backend. The eventService : EventService contains the core business logic and coordinates all actions required to create an event. The eventRepository : EventRepository handles data persistence by saving and retrieving event records. The mediaClient : MediaStorageClient manages the upload of optional promotional media, while the notificationClient : NotificationClient is responsible for sending update notifications to society members. Finally, the event : Event lifeline represents the domain entity that holds all event details. These lifelines align directly with the classes in the class diagram and the components in the subsystem architecture, ensuring consistency across all design models.

Main Scenario

1) The sequence begins when the Society Leader submits the event form, prompting the eventController to call createEvent(eventData) on the eventService.
2) The eventService passes the event data to the eventRepository through a synchronous save(eventData) call, where the event information is persisted.
3) The eventRepository returns the newly created event : Event object to the eventService, confirming successful storage.
4) The eventService then initialises the domain-level event : Event object (shown through the create() message), preparing it for any additional processing such as attaching optional media.

Optional Media Upload – *opt* Fragment

The diagram uses an opt combined fragment to show the optional upload and processing of promotional media. This fragment is controlled by the guard **[mediaIncluded]**, which means the interaction inside it only happens when the Society Leader actually attaches an image or video while creating the event. If the condition is true, the eventService sends an uploadMedia(media) request to the mediaClient, which handles external media storage. The mediaClient uploads the file, carries out any necessary checks or preprocessing steps, and then generates a link to the stored file. In the sequence diagram, this link appears as **<mediaUrl>**, and it is returned back to the eventService as confirmation that the upload was successful.

Once the eventService receives the media URL, it attaches it to the event : Event object so that the promotional content becomes part of the final event record. This allows the uploaded media to appear later in the app wherever the event is displayed. This workflow reflects how optional features are usually handled in real systems: the service interacts with an external storage client only when needed and then updates the event with whatever extra information is returned. By modelling this step inside an opt fragment, the diagram clearly separates the optional behaviour from the main event-creation flow, showing that the system adapts based on whether the user provided a file while still keeping the rest of the process consistent.

Notification and Response

After the event information and any optional media files have been successfully stored, the EventService moves on to the final stage of the workflow, which is notifying the relevant users. To do this, it calls sendNotification(event) on the NotificationClient. This step is important because it ensures that students who follow the society receive immediate updates about new or modified events. It also matches the behaviour shown in the “Publish Event Updates” use case and the related action in the activity diagram, where the system informs subscribers once the event is ready.

When the notification has been sent, the EventService prepares a success message and returns it to the EventController. The controller does not perform any extra processing; instead, it simply forwards this result back to the Society Leader App UI. This allows the interface to show a confirmation to the user, letting them know that the event has been created or updated without any issues. From a user perspective, this final response helps close the loop by reassuring them that all backend operations validation, storage, media upload, and notification have been completed successfully. The sequence therefore shows clearly how the system transitions from backend processing back to the user interface.

Consistency with Other Models

The sequence diagram reflects the Create Event and Upload Promotional Media use cases from the use case diagram by showing how the system reacts to the Society Leader’s actions step by step. The main stages from the activity diagram, such as validating the input, storing the event and any optional media, and publishing the final update, are clearly represented through calls like save(eventData), the opt [mediaIncluded] fragment, and sendNotification(event). Each lifeline in the sequence diagram matches one of the classes in the class diagram, including EventController, EventService, EventRepository, MediaStorageClient, NotificationClient and the Event entity itself. These lifelines also map directly to the microservices shown in the component diagram. Because of this alignment across the different models, the behaviour shown in the sequence diagram stays consistent with both the structural design and the architectural layout that were developed in the earlier stages of the coursework.

Reflection on Team Work

5.1 Contribution to Team Organisation

At the beginning of the semester, I could not attend the first two weeks of meetings because I was still in Turkey. However, once I arrived in the UK, I became an active participant in the organisation of our teamwork. In Week 3, I chaired the meeting by preparing the agenda, structuring the discussion, and helping the group clarify the upcoming tasks for each subsystem, including my own responsibility for the Society Leader App subsystem. I also helped establish our communication structure by contributing actively to our WhatsApp group and ensuring that every member stayed aligned with the weekly goals. Although I missed a few sessions later due to health-related issues, I made sure to stay updated on team decisions and maintained communication so that my absence did not disrupt the workflow. Additionally, I regularly wrote and uploaded the weekly meeting minutes and assisted in organising our GitHub repository structure, ensuring that documentation, diagrams and Papyrus files were consistently maintained.

5.2 Technical Contributions

For the technical aspects of the coursework, I completed all tasks related to the Society Leader App subsystem on time, including the use case diagram, activity diagram, subsystem architecture, class diagram, and sequence diagram. I committed my work to GitHub frequently and in small, traceable increments, ensuring that every stage from initial drafts to final Papyrus models was clearly documented in the repository. In addition to producing my own subsystem artefacts, I actively engaged in cross-checking the work of other team members. I reviewed their UML diagrams for syntax accuracy, consistency in naming conventions, and compatibility of interfaces across subsystems, and I provided comments whenever I identified inconsistencies or areas needing clarification. To improve the quality of my modelling decisions, I also made use of resources from the recommended reading list, which helped me refine my understanding of microservice-based design and apply UML rules more systematically. These activities collectively ensured that my subsystem integrated smoothly with the overall architecture and that the team maintained a coherent and high-quality set of technical artefacts.

Conclusion

This coursework helped me understand how different UML models fit together when designing a microservice-based subsystem. Working on the Society Leader subsystem showed me how requirements are translated into use cases, workflows, architectural components, and finally detailed class and sequence diagrams. As I progressed through each task, I realised how important consistency is, because a modelling choice in one diagram often affects the others.

I faced several challenges, especially at the beginning when the specification felt complex, and later when using Papyrus, which caused a lot of small technical issues. Even though it was frustrating at times, once I managed to solve the problems and get the diagrams working properly, it actually became enjoyable to see everything come together. Reworking my models and revisiting the lecture material helped me apply UML notation more confidently and understand the reasoning behind each design choice. Overall, this assignment strengthened my understanding of UML and gave me a clearer idea of how software systems are documented and designed in practice.

References (Alphabetical Order & Harvard Format)

Baker, D. (no date) *Papyrus Tutorial*. Available at: https://www.cs.fsu.edu/~baker/swe1/restricted/notes/tutorial_papyrus/tutorial_papyrus.html (Accessed: 10 November 2025).

Bass, L., Clements, P. & Kazman, R. (2024) *Software Architecture in Practice*. 4th edn. Boston, MA: Addison-Wesley. Accessed: 15 November 2024.

Booch, G., Rumbaugh, J. & Jacobson, I. (2005) *Systems Analysis & Design: UML Version 2.0 – An Object-Oriented Approach*. 2nd edn. Boston, MA: Addison-Wesley. Accessed: 23 November 2024.

Dennis, A., Wixom, B. H. & Roth, R. M. (2012) *Systems Analysis and Design: An Object-Oriented Approach with UML*. 5th edn. Hoboken, NJ: Wiley. Accessed: 23 November 2024.

Microsoft (no date) *Software Development Fundamentals*. Available at: https://learn.microsoft.com/en-us/shows/software-development-fundamentals/01 (Accessed: 10 November 2024).