Advanced Software Design
Final Report
Group 16

Alexander Sellström
alse7456@student.uu.se

Claudia Martinez Alquezar
claudia.martinezalquezar.5671@student.uu.se

Salum Nassor
salum.nassor.2008@student.uu.se

May 25, 2023

Contents

1 Introduction **4**

2 Definitions **4**
2.1 MATCH System . . . 4
2.2 Configuration . . . 4
2.3 Requester . . . 4
2.4 Responder . . . 4
2.5 Student . . . 4
2.6 Tutor . . . 4
2.7 Administrator . . . 4
2.8 Front-end . . . 5
2.9 Dashboard . . . 5
2.10 Front-end clients . . . 5
2.11 Weighted Round Robin . . . 5
2.12 Type Introspection . . . 5
2.13 Reflection . . . 5

3 Domain Model (B1 and B2) **6**

4 Software Architecture (C1 and C2) **7**
4.1 Class Diagram . . . 7
4.2 Component diagram . . . 8

5 Functional Requirements (C1) **9**
5.1 Users: Student and tutor . . . 9
5.2 Student Application . . . 10
5.3 Tutor Application . . . 12
5.4 Request Tracking . . . 13
5.5 Preferences . . . 13
5.6 Matcher . . . 14
5.7 Ethics and Feedback . . . 15
5.8 Learning . . . 15
5.9 User Management . . . 15

6 Non-Functional Requirements **16**
6.1 Usability requirements . . . 16
6.2 Language limitations . . . 17
6.3 Compatibility constraint . . . 17
6.4 Should be Scale-able . . . 17
6.5 Accessibility requirement . . . 17
6.6 Security requirement . . . 17
6.7 Responsiveness requirement . . . 17

7 Static Model (D1 and D2) **18**
7.1 GRASP and SOLID Principles (A1 and A2) . . . 19

8 Behavioral Model (D3 and D4) 21
8.1 User account registration . . . 21
8.2 Student making a request . . . 22
8.3 Tutor responding to a request . . . 23
8.4 Tutor generating Invoice for a request . . . 24

9 Implementation (E1 and E2) 24

10 Design Patterns (F1 and F2) 25
10.1 Observer . . . 25
10.2 Facade . . . 25

11 Refactoring (G1, G2 and G3) 26
11.1 New Use Case: Payment . . . 26
11.2 New requested changes, Design refactoring and Planning . . . 27

12 Reflection (I1 and F5) 28
12.1 Difficulties (I1) . . . 28
12.2 Ethical Considerations (F5) . . . 29

1 Introduction

This document outlines the functional and non-functional requirements for the MATCH software system, extended to a **Study and Tutor configuration**. This configuration is an extension of the MATCH system that matches students needing tutor services.

This document also provides brief explanations of the terms used throughout the document, and proposes a suitable underlying architecture for the Student and Tutor configuration, giving a complete overview of the system and its components.

2 Definitions

2.1 MATCH System

MATCH is a software infrastructure that provides mechanisms for matching users requiring information or services with potential suppliers of such information or services. The primary goal of the MATCH system is to connect people to other people in the most effective way.

2.2 Configuration

Purpose based software solution that is developed by extending the MATCH system features and capabilities and giving it purpose in a specific domain. In this document, Student and Tutors is the configuration in focus.

2.3 Requester

End user of the final software product that requests a service in the MATCH system.

2.4 Responder

End user of the final software product that provides a service in the MATCH system.

2.5 Student

Student is the user that will fulfill Requester responsibilities in the Student and Tutors configuration. The student uses the student application to create and manage requests of study-related services.

2.6 Tutor

Tutor is the user that will fulfill Responder responsabilities in the Student and Tutors configuration. A tutor uses a tutor application to create and manage their account, as well as to accept or decline service requests from students. A tutor can also ask question or request tutoring services.

2.7 Administrator

Personnel that manages the MATCH system daily administrative activities. They manage users in the MATCH system using a front-end dashboard. 2.9.

2.8 Front-end

Presentation layer that is concerned with views and user interaction. A mostly graphical user interface that is presented to the user and which the user can use to perform certain tasks within the system, or view certain information.

2.9 Dashboard

A front-end 2.8 component that provides administrative capabilities in the MATCH system. Administrative abilities such as managing MATCH system users, managing user preferences, managing feedback and reports and other mentioned in this document.

2.10 Front-end clients

These are front-end components 2.8 that serve a specific purpose, corresponding to the client they are intended for. In this configuration there exist 3 front-end clients:

- Student Application
- Tutor Application
- Dashboard

2.11 Weighted Round Robin

Load balancing method, each server is assigned a weight (or preference), usually commensurate with its capacity. The higher the weight, the more requests a server receives. For instance, a server assigned a weight value of two receives double the requests of a server assigned a value of one.

2.12 Type Introspection

This is the ability to inspect the code in the system and see object types

2.13 Reflection

This is the ability to make modifications at runtime by making use of introspection.

3 Domain Model (B1 and B2)

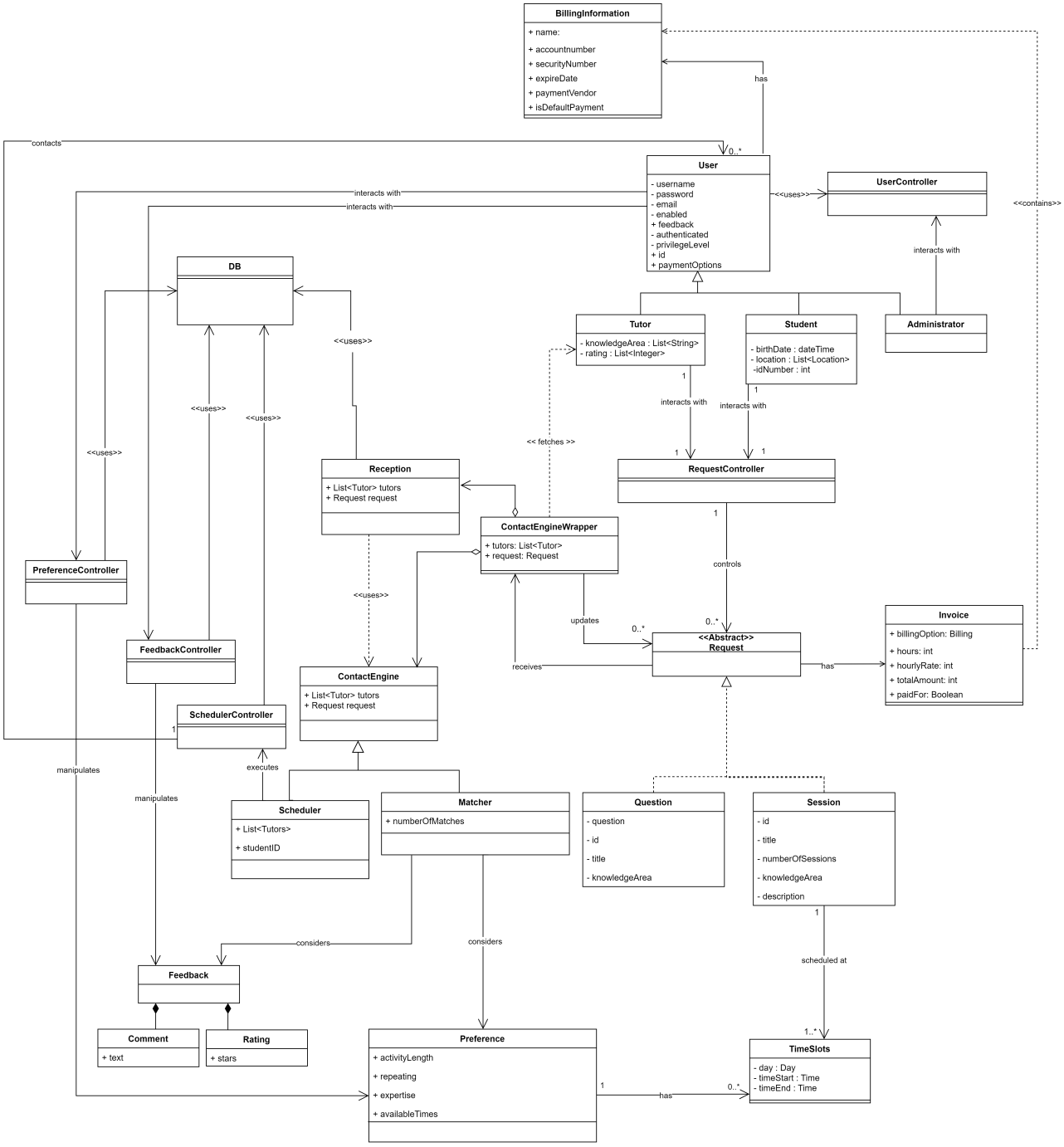


Fig. 1. Domain Model

4 Software Architecture (C1 and C2)

4.1 Class Diagram

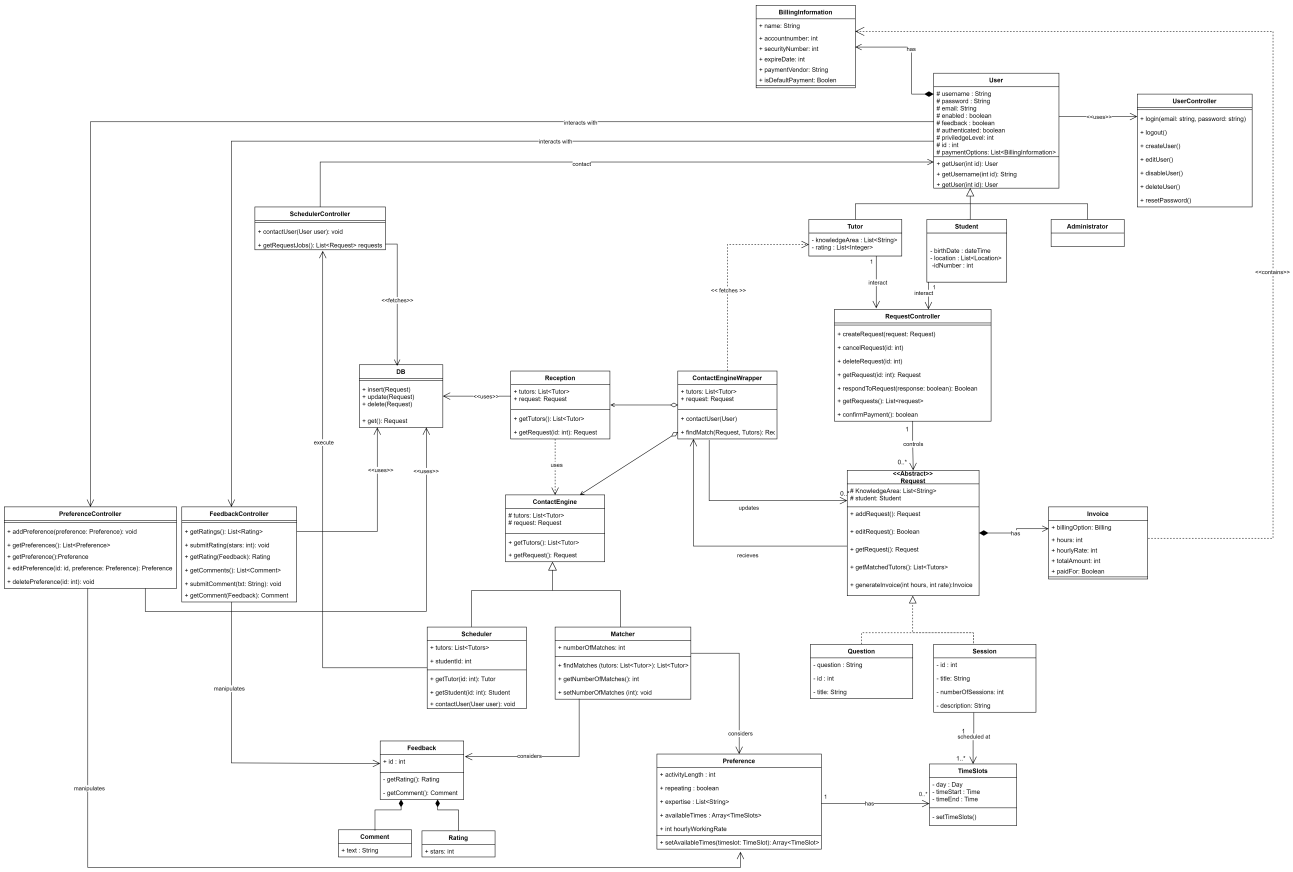


Fig. 2. Class Diagram

4.2 Component diagram

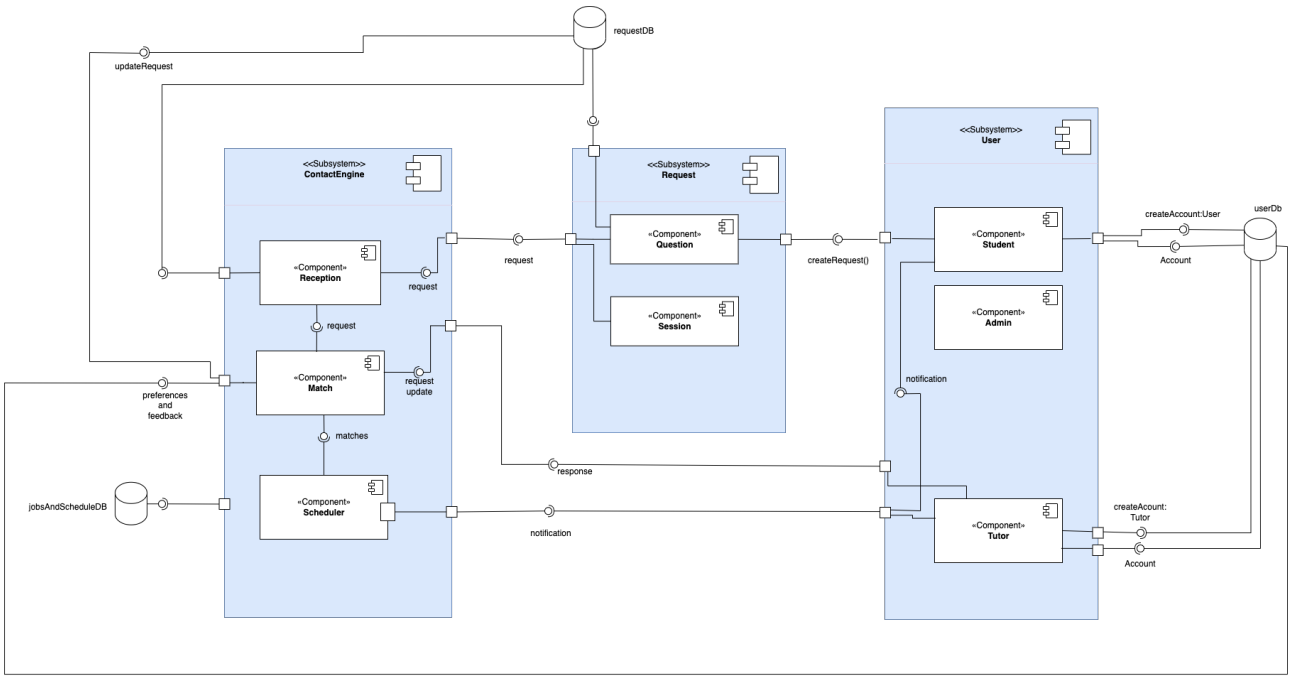


Fig. 3. Software architecture for the system under design

Referring to figure 3, It tries to visualize the organization of system components and the dependency relationships between them. It also provides a higher level view of the components within a system. Also tries to explain the functions of the system being built without showing the specifics. The system under design is built upon a communication of 3 sub-systems with each of these sub-systems containing more than one component. Sub-system in the diagram Contact engine, Request component and User sub-system.

4.2.1 Contact engine

It receives requests from Student and finds appropriate Tutor to address those questions or session. It as well manages all student and tutor information, feedback, preferences, etc.

The contact engine is the primary component of the system. It handles inbound and outbound communication with the system and provides the intelligent matching and scheduling functionality. The heart of the contact engine is the request loop. Requests loop through the system until they are fully completed, managed by the contact engine. Logically, the contact engine can be thought of as three (3) components:

The major role of the **Reception** component is to determine what action should be taken by the MATCH system based on a request.

The **Matcher** component tries to find matching users for a request.

The **Scheduler** component realises the execution of various types of scheduled jobs.

- Contacting potential tutors about availability

- Contacting student about connection made
- Contacting students and tutors to obtain feedback about earlier connections

4.2.2 Request subsystem

This sub-system is responsible for handling requests made by the Student in their student front-end client 2.8. The requests come in two forms handled by two components, question and sessions component. The **question** component is responsible of handling all request from students that comes in form of questions. Session component handles all session requests made by the Student.

4.2.3 User sub-system

The user sub-system responsibility is to issue request from the student with the students front-end client, and is also responsible for creating and managing the users (tutors and students).

5 Functional Requirements (C1)

This section will list and briefly guide the developer on functional requirements extracted from the MATCH system requirement document, and then list down functional requirements specific to the Student and Tutors configuration.

5.1 Users: Student and tutor

These are some of functional requirements that are focus on the users of the student and tutor configuration. In this case our users are Students and Tutors, and below are some of the functionalities that applies to both.

5.1.1 User should be able to create an account

Both student and tutor in the MATCH system should be able to create a user account in their respective end user client application. Both student and tutor are required to fill the information fields during account creation. A student should provide:

- Full name
- Email
- Date of birth
- Place of residence
- Identification number of the student. If the user is only student and not above 18, this field is not required.

5.1.2 Administrator, tutors and students should be able to login in to their respective applications

Administrator, tutors and students should provide email and password in their respective applications to be able to be authenticated by the server and start a session. All actions must be authorised, based on the user's identity and validity of a user's subscription.

5.1.3 A logged in user 5.1.2 should be able to logout in their respective applications

While logged in their respective applications, a user should be able to log out of their account by selecting "Log Out" in their account settings.

5.1.4 A user should be able to manage their account

In their respective applications, both student and tutors should be able to add, delete or edit the information they had previously input during account creation 5.1.1. They should be able to do this by navigating to user settings and add, delete or edit the information in the provided fields.

5.1.5 A user should be able to reset their password

In their respective applications, both student and tutor should have the option to reset their password. The options should be available and visible to a registered user of the system under one of the following conditions:

- Unauthenticated user (the user, student or tutor, has forgotten their password), preferably in the login page.
- Authenticated the user, through the account settings.

5.2 Student Application

A student application is a front-end client 2.10 that enables the student to ask questions or request tutoring services from a registered tutor. The application has the following functional requirements.

5.2.1 Student should be able to open a request for a service

The student should be able to create a request by filling different fields in the provided user interface.

- The student will select the type of service: a single question, a single tutoring session, or continued sessions. Different fields will be available depending on which service is selected.
- If the student has selected a single question, the student will then write their question in the provided "Question" field.
- If the student has selected a tutoring session, the student will write more information regarding their inquiry in the "Additional information" field.
- If the student has selected continued sessions, the student will write more information regarding their inquiry in the "Additional information" field and they shall also be able to select their preferred dates to schedule tutoring sessions, by selecting which days of the week (Monday-Sunday) and which time slot (morning or afternoon) they would prefer.
- The student must select if prefer a tutor from their past fulfilled requests, or a new tutor. If there are no past tutors, the option to choose past tutors will be unavailable.

- The student will proceed to click "Request Service".
- The system must send a confirmation email to the student whenever a service request is successfully placed.

5.2.2 A student should be able to view a list of all requests made

A student should be able to see a list of all the completed and pending requests they have created in the system so far, by navigating to the "My Requests" section.

5.2.3 A student should be able to see the details of a request they made

Details of all past and present requested services by the student, complete or incomplete, should be accessible for the student to view. The student should be able to view requests by:

- Navigating to "My Requests" 5.2.2
- Selecting a request
- Viewing the request

The Request view should show the student:

- Progress and Status of the request,
- Tutor assigned to the request,
- Date request made,
- Date request completed (if complete) or "To be completed" otherwise,
- Request type, request headline (Question or Inquiry of tutoring service),
- Additional information and description
- Option to cancel their request

5.2.4 The student should be able to cancel a request

A student can cancel their pending request at any state of the request, except 2 hours or less before the tutoring session is scheduled to start. The student should be able to do this by:

- Navigating to their request 5.2.3
- Selecting to cancel the request
- Confirming the request cancellation

5.2.5 A student must be able to view request status and progress

A student must be able to see the progress through various states of their request, by:

- Navigating to the specific request 5.2.3
- Opening the request to view progress and status

A student should be able to manage their requested service as it progresses through various states. The Student can create a request 5.2.1, cancel a requested service 5.2.4 and see the progress of their request 5.2.5.

5.3 Tutor Application

A Tutor application is a front-end client 2.10 that enables the Tutors to respond to pending requests. The tutors will interact with this application, which offers different features from the Student application 5.2.

5.3.1 A tutor should receive notifications for a request

A tutor shall receive a notification of a request when and only when their personal information and preferences satisfy the conditions and preferences set by the requester, a student.

5.3.2 A tutor should be able to view a request once they are matched

Once the request notification is received 5.3.1, the tutor must be able to view the service requested by the student. The tutor should be able to see the request details:

- Headline of the request
- Type of service (question, tutoring session or continued tutoring request)
- Request description and additional information
- Date of expected completion
- Username of the student. Username should be clickable, and when selected it should take the tutor to the student's profile.
- Current status of the request
- A prompt to respond to the request, with options to accept or decline 5.3.3

5.3.3 A tutor should be able to respond to a request

A tutor should be able to respond to a request once they receive a service request from a Student. Once the notification is received, the Tutor will select the notification and view the request 5.3.2. The tutor will proceed to respond to the prompt by choosing "Accept" or "Decline".

5.3.4 A Tutor should be able to cancel an accepted request

A tutor can cancel their incomplete accepted requests at any time, except 2 hours or less before a tutoring session is scheduled to start. The tutor should be able to do this by:

- Navigating to the request
- Selecting to cancel the request
- Confirming the request cancellation

5.3.5 A tutor should be able to give feedback after every completed service

The tutor shall provide feedback on a service offered to a student once the service request has been completed.

- After every completed service, the tutor will confirm that the service is completed and will be prompted to give a feedback review.
- They will select the category their feedback belongs to, and then type in additional information in the field provided.
- Some categories of feedback have scales. The Tutor will be able to rate these feedback categories on a scale of 0 to 5.

5.3.6 The tutor should be able to see information regarding their matched requests

From their account, the tutor should be able to access:

- Pending requests
- Request history
- Payments due

5.4 Request Tracking

5.4.1 All users in the match system and configuration should be able to track a live request

The administrators, as well as the student and tutor involved in a service request should be able to track a live request through all its states. The tutoring request must be tracked from initiation to completion, feedback and finally closure.

The student can close their request by either cancelling their request 5.2.4 or marking it as completed.

Progress of the request should be visible to all user associated with the request. On completion, the student, tutor and system are to be notified, after which the request status is flagged as completed. Feedback from the student and tutor should be visible to all of the associated parties.

5.5 Preferences

5.5.1 A student and tutor must have preferences that influence request connection

A student and tutor should have preferences that contribute in matching a candidate with the requested service. These preferences must be satisfied for a match to happen. The preferences are:

- Time of the activity

- One-off session or repeating sessions, in the case of request for tutoring sessions
- Area of expertise
- Availability, a tutor must be able to specify their availability by choosing their preferred days (Monday-Sunday) and time slots.

5.6 Matcher

The Matcher component should support matching the student request with a single tutor or a bounded list of tutors.

5.6.1 The system should employ different strategies to match student request to tutor candidates

The Matcher should be able to use more than one strategy to match a tutor with a student request. Strategies employed should be:

- Match and invite all the tutors that satisfy the preferences 5.5 set by the student, cancel those who don't win/accept.
- As above, but student has to accept the best possible match for the match to be complete.

5.6.2 MATCHER should return several prioritised matches for the student to select

After the system matches student request with several tutors, the student will be able to select between the prioritized tutor matches. The prioritization is based on a combination of factors and priorities 5.5:

- **Knowledge area and skill rating:** The matches should consider if the knowledge area requested by the student and the knowledge area of the tutor match, as well as the skill rating of the tutor in the given area. The higher the skill rating, the higher their position in the list.
- **Time schedule:** The student will first see tutors that are available during their specified preferred day and time for tutoring. After that, the student will see tutors that are close to their preferred times, but not perfectly matched (for example, tutors that are available on their preferred days, but not in their preferred time slots).
- **Location:** The student will first see tutors that are closer to their location.
- **Feedback:** The Student can see the feedback given to the tutor by other students, and select the tutor with the best rating.

5.6.3 The MATCHER shall employ a fall on strategy once no more matches are available

When no more matches are available, the MATCH system shall try again after some time. If the system could not match the request to a tutor after trying again, it will inform the student a few moments before the requested service is expected to be given. The student can then choose to try again or cancel the request.

5.7 Ethics and Feedback

5.7.1 Both tutor and student should be able to give feedback

After a request is completed, both the student and tutor involved can leave feedback about the provided service.

5.7.2 Feedback should be disabled if either the tutor or students decide to disable

Both the student and the tutor can disable feedback for their account, by selecting "Disable Feedback" in their account settings. Any user can see if a student or tutor has disabled feedback by checking their profile.

5.7.3 Both tutors and students should be able to report abuse or illicit conduct

Both parties should be able to report abuse, by flagging a comment made on a student request or flagging the student's request as a whole. When flagging, the party that is flagging the content should input details of the abuse. This will then be recorded in the database.

5.7.4 Abuse should be reported to the MATCH system admin

Any abuse reported should be saved in the database and then forwarded immediately to the MATCH system admin. This should be through email and MATCH system dashboard notifications.

5.8 Learning

5.8.1 The MATCH system should be able to learn

The matcher should be able to learn to determine good connections between the student and tutor and improve them based on feedback. Learning from past experiences of all kinds and forecasting based on these experiences plays a crucial role in MATCH.

To facilitate learning, all tutors and students and their preferences need to be registered with the learning feature.

In addition, all requests (active or otherwise) plus all feedback needs to pass to the learning feature.

The feature will facilitate:

- Matching student and tutors by using preferences 5.6.1
- Prioritizing the list, which tutor goes on top of the list.
- Who is excluded in the tutor matched list, even if they match/satisfy the students preferences. In cases where the student has a bad experience with a tutor or they explicitly mentioned them as a service provider they don't want to be matched with.

5.9 User Management

The MATCH system should offer facilities for managing user accounts, including creating accounts, resetting passwords, disabling accounts, etc. These facilities are to be used by the product owner administrative personnel, **Administrator**.

5.9.1 An admin should be able to manage user accounts in MATCH system

MATCH system admin should be able to manage users in the MATCH system through an application, MATCH system dashboard in this case.

- The dashboard 2.9 should present the Administrator with all users and their respective account status (Student or Tutor).
- The administrator will be able to view all users
- While view users, Admin has options to disable, reset password and edit account information

5.9.2 An admin should be able to create user accounts

A MATCH system should enable the administration to create user, tutors and students included. To create users for student and tutor configurations, they will input data similar to 5.1.1 for a student and tutor account.

5.9.3 An admin should be able to reset users' account password

A MATCH system shall let the admin reset a users' password whenever necessary. The user will then receive a link through an email or phone and they will be able to reset or change the password.

5.9.4 MATCH system dashboard should let the Administrator to be able to disable accounts

The Administrators should be able to disable any user's account, except those of other Administrators. They should do this by:

- Going to the Dashboard 2.9
- Selecting "All Users"
- The administrator will be able to view all users and search for a specific user by username
- While viewing users, Admin has options to disable, reset password and edit account information

6 Non-Functional Requirements

6.1 Usability requirements

The tutor and student should be able to intuitively use their respective applications. Each application should serve a single purpose. But a tutor can use their application to request for a service as well.

6.2 Language limitations

All front-end clients will only be available in English.

6.3 Compatibility constraint

The tutors and students should be able to access notifications on multiple devices. Example; smartphones and personal computers.

6.4 Should be Scale-able

The MATCH system should be able to handle multiple users concurrently, from 1000 users to more than 1 million users'.

6.5 Accessibility requirement

All front-end clients should ensure any person can use it comfortably and without major complications. In other words, it focuses on ALL users and it aims to provide the same user experience for all.

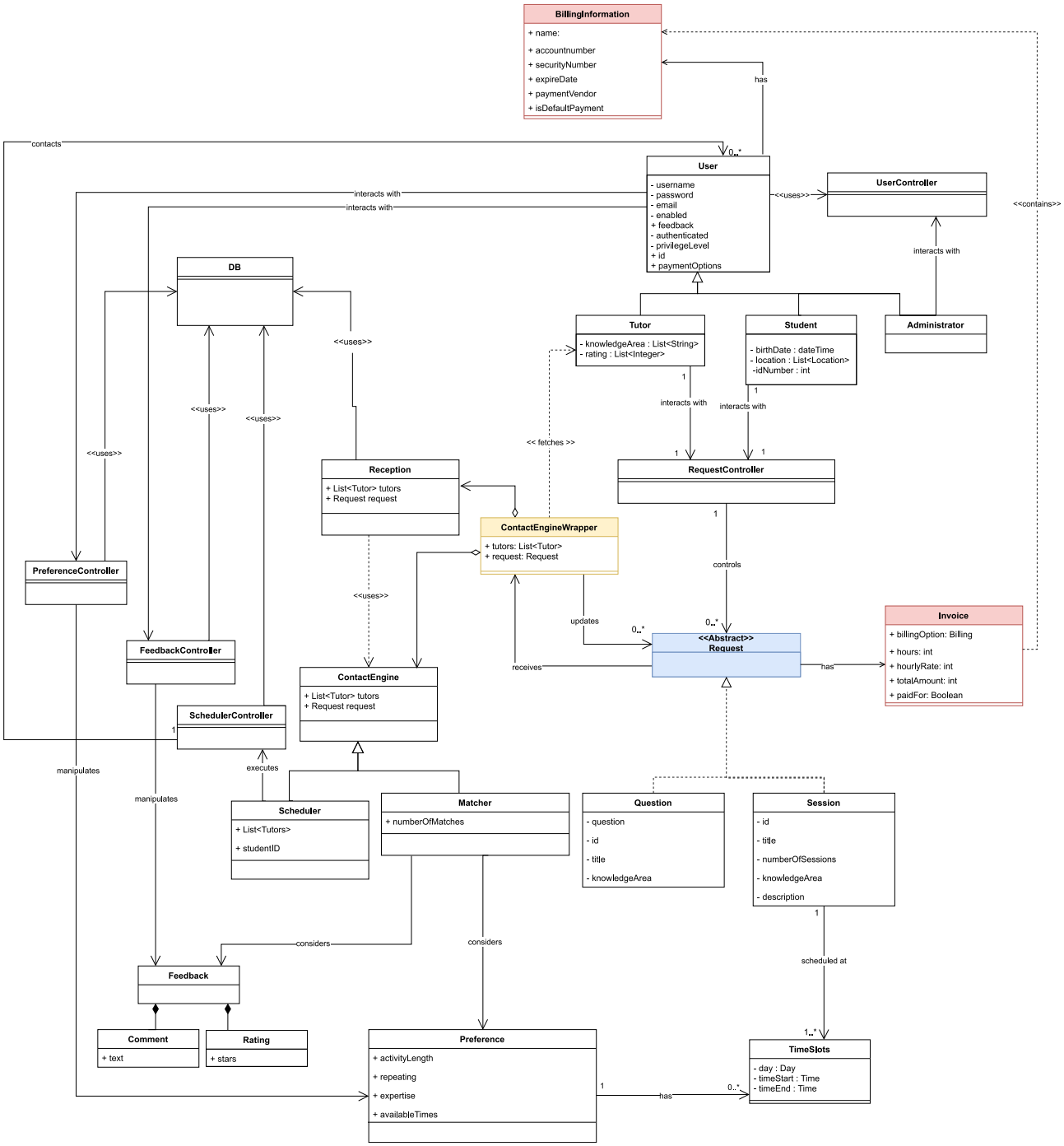
6.6 Security requirement

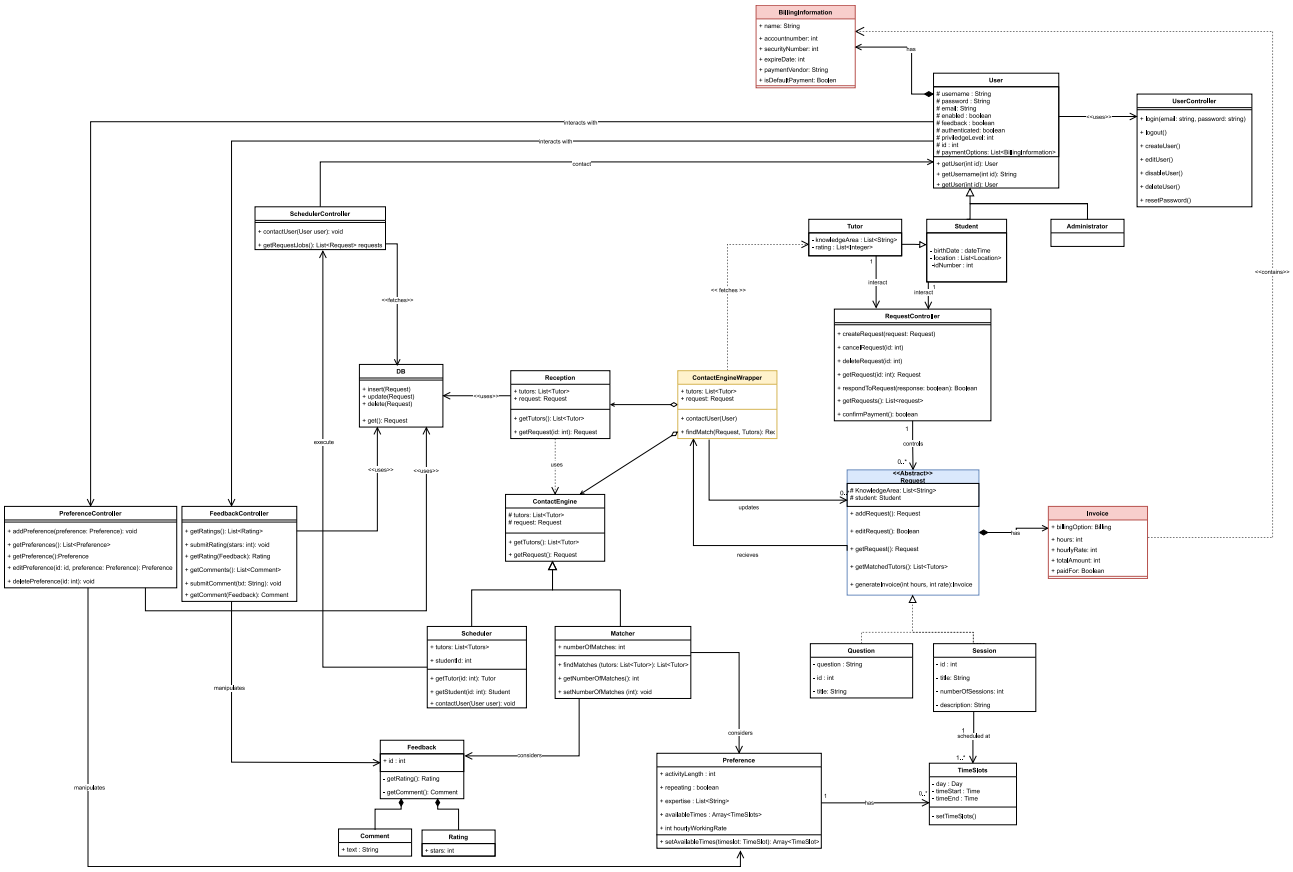
All data inside the system should be protected against bad actors trying to gain unauthorized access. The users' accounts should be protected with sufficiently secure passwords and password encryption. Implementing two-factor authentication is optional.

6.7 Responsiveness requirement

The system should respond to users' actions with a maximum response time of 5 seconds. The system should find a match for a specific request in less than 3 hours, or return a message showing that no matches could be found.

7 Static Model (D1 and D2)





The Requests are created by Students using the RequestController. The Reception module receives all the requests and stores them in the Database, to be able to edit and access them later.
The Reception module determines what action to take depending on the request it has received. It uses the ContactEngine to find tutors that match the request and to contact the student and tutors involved in the request.
The ContactEngine matching capabilities are implemented by the Matcher. When a request is received, the Matcher finds appropiate Tutor matches by considering the Preferences and the Feedback of the prospects.
The Scheduler implements the scheduling capabilities. It schedules the requests based on the job descriptions in the database and accesses the Database to access these descriptions. This module contacts the Users involved and handles communication between them during the request. The Receivers contacted can accept or reject the job by communicating with the Scheduler.

7.1 GRASP and SOLID Principles (A1 and A2)

Information expert: This principle is used to decide how to assign responsibilities. Responsibilities (such as methods) should be assigned to the class that has the information needed to fulfill them. Example in the class diagram: Reception should have the method to obtain the request, because it knows its ID and has access to the Database.

Pure Fabrication: Consists of a made-up class that is assigned a set of responsibilities to avoid decreasing cohesion or increasing coupling. An example in the class diagram: DB Class is used to store everything, and it is a general class that all the other classes that need to store "something" can use, instead of having multiple methods of saving data in every class.

High Cohesion: This principle aims to keep objects focused and manageable by making sure objects have few methods that are all related, and focus on doing only one thing, but doing it well. An example in our class diagram: PreferenceController manages preferences and nothing else, keeping it focused on this aspect of the system.

Low coupling: This identifies how independent the different modules are between each other, with more independence (and therefore lower coupling) being the aim. Example: Rating and Comment are only dependant on Feedback due to them being compositions of the Feedback class, instead of being separate modules associated with Feedback and with the Matcher separately. The Matcher goes through Feedback to obtain the Rating.

Controller: This principle assigns the responsibility of dealing with system events to a class that represents a use case scenario or an overall system. For example: RequestController is responsible for controlling Requests (creating, deleting, editing them, given an ID), and receives the commands from the Tutor and Student classes. It delegates the work to the Request class.

Creator: Aims to decide which class is responsible for creating objects, by following a set of principles in order to select the correct class. For example: Feedback creates instances of Comment and Rating, since it has the initializing information for them and they can only exist as long as Feedback exists, due to being compositions of the Feedback class.

8 Behavioral Model (D3 and D4)

8.1 User account registration

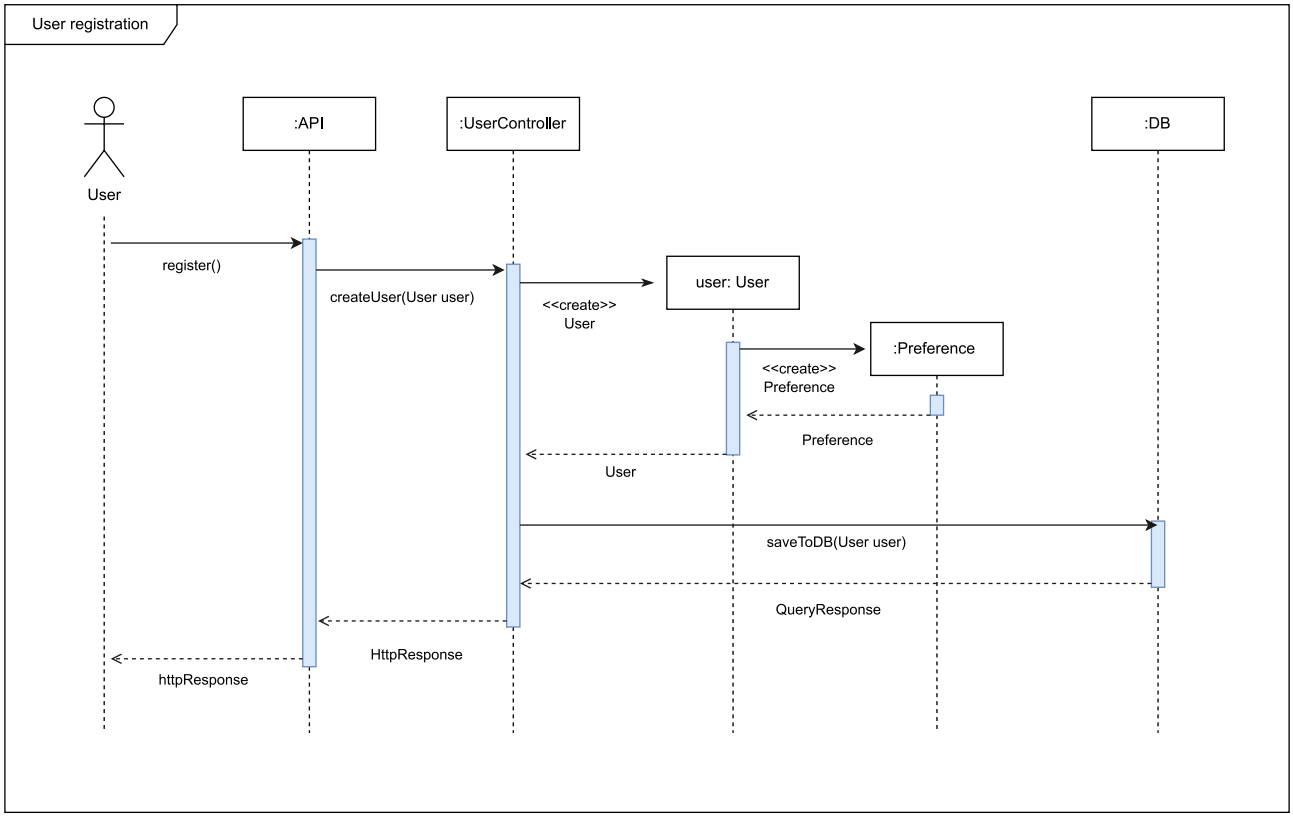


Fig. 4. User account registration sequence diagram

The user registration process begins with an API request. This request is handled by the UserController service, which utilizes the CreateUser function to instantiate a new User object. Preferences are subsequently established for the User object and subsequently attached. Our database service then persists the User object. The UserController service subsequently returns a response to the API and the user, indicating successful completion of the registration process.

8.2 Student making a request

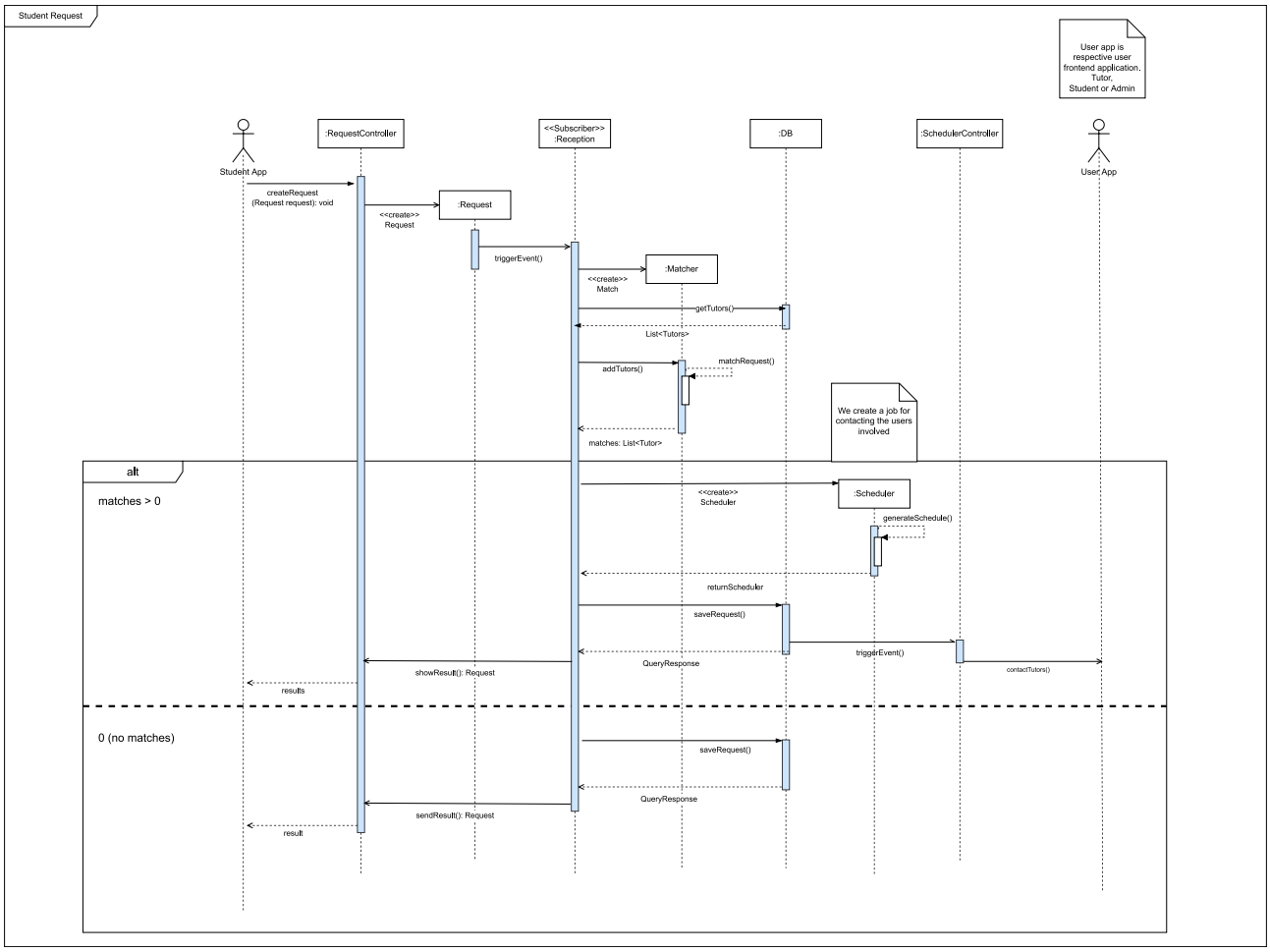


Fig. 5. Student making a request sequence diagram

The process of handling student requests through the RequestController involves several steps. When a student makes a request, the RequestController first creates a Request Object. This object contains information about the student's requested service and preferences.

Once the Request Object is created, it is passed to the reception, which acts as an observer. The reception then creates a new Matcher Object, which is responsible for matching tutors with the student's requested service and preferences. To do this, the reception fetches a list of tutors from the database and adds it to the Matcher Object. The Matcher Object then compares the qualifications and availability of each tutor with the student's requested service and preferences. If a match is found, the Matcher Object returns a list of matched tutors.

A Scheduler Object is then created to establish a schedule for contacting the matched tutors. The Scheduler Object contains information such as the date and time of the scheduled session, the contact details of the matched tutors, etc. This object is saved to the database for persistence and the matched tutors are notified about the schedule via the contact information provided.

If the Matcher Object is unable to find a match, the request is logged in the database and a response indicating the unavailability of a suitable tutor is sent to the RequestController.

In addition, the reception can also retrieve the requests from the database and check for the matching process, schedule and update the matched tutors accordingly.

Overall, the RequestController, Matcher Object, and Scheduler Object work together to efficiently handle student requests and match them with available and qualified tutors.

8.3 Tutor responding to a request

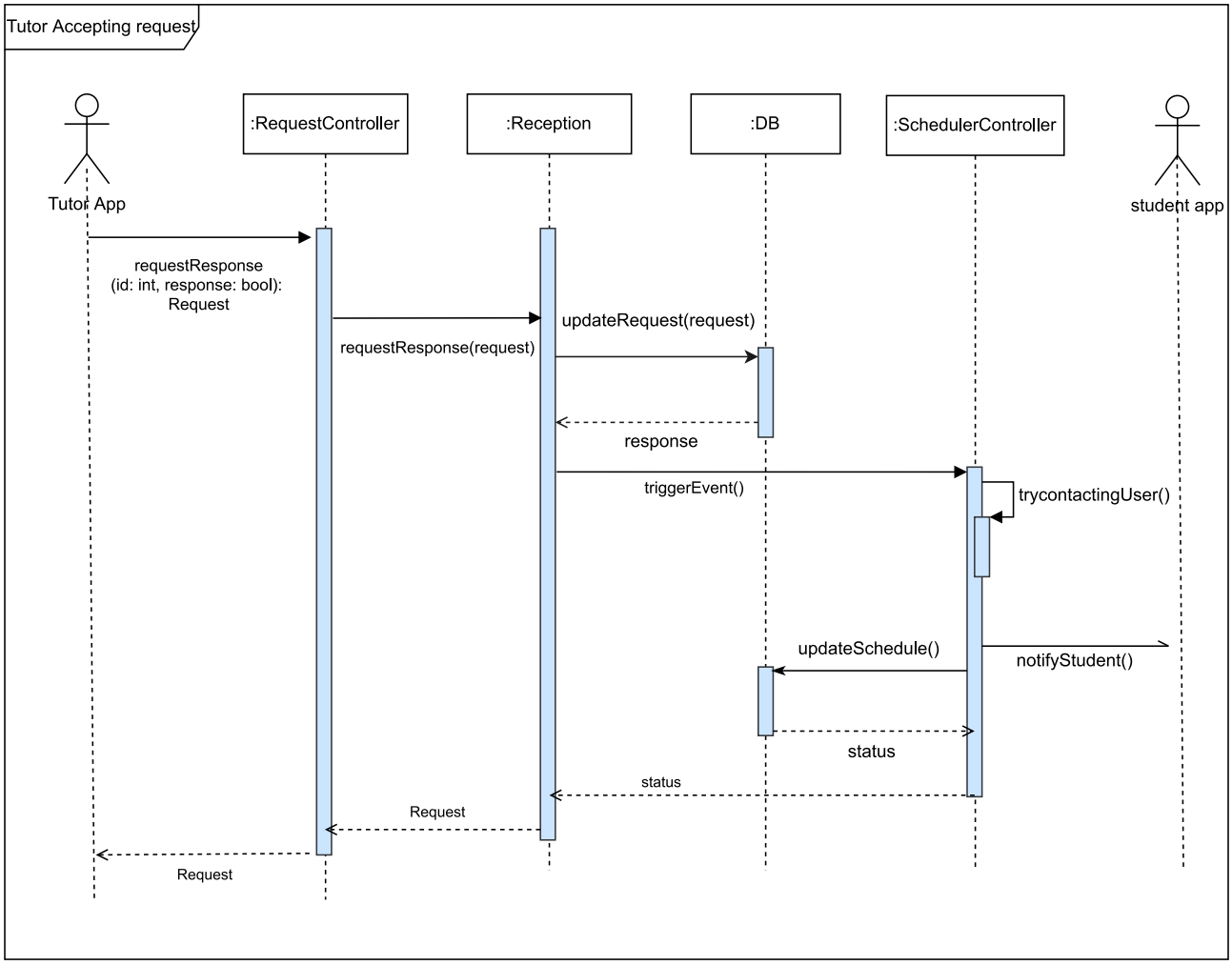


Fig. 6. Tutor responding to a request sequence diagram

The process of a tutor responding to a request for service involves several steps, beginning with the RequestController handling the initial request. Once the request is received, the RequestController dispatches a RequestResponse to the reception. The reception then updates the request in the database and retrieves the response.

Next, the reception triggers a contact through the Scheduler Controller to the student. The student is then notified, and the schedule is updated in the database. The status of the schedule is then returned to the Scheduler Controller.

The reception receives the status of the scheduler updates it accordingly. Once the notification is dispatched, the status is returned to the reception and the RequestController gets a httpRe-

sponse. The tutor is also informed of the status of the schedule.

Overall, the RequestController, Reception, Scheduler Controller, and the Database work together to efficiently handle the process of a tutor responding to a request for service, from receiving the initial request to updating the status of the schedule and informing the student involved.

8.4 Tutor generating Invoice for a request

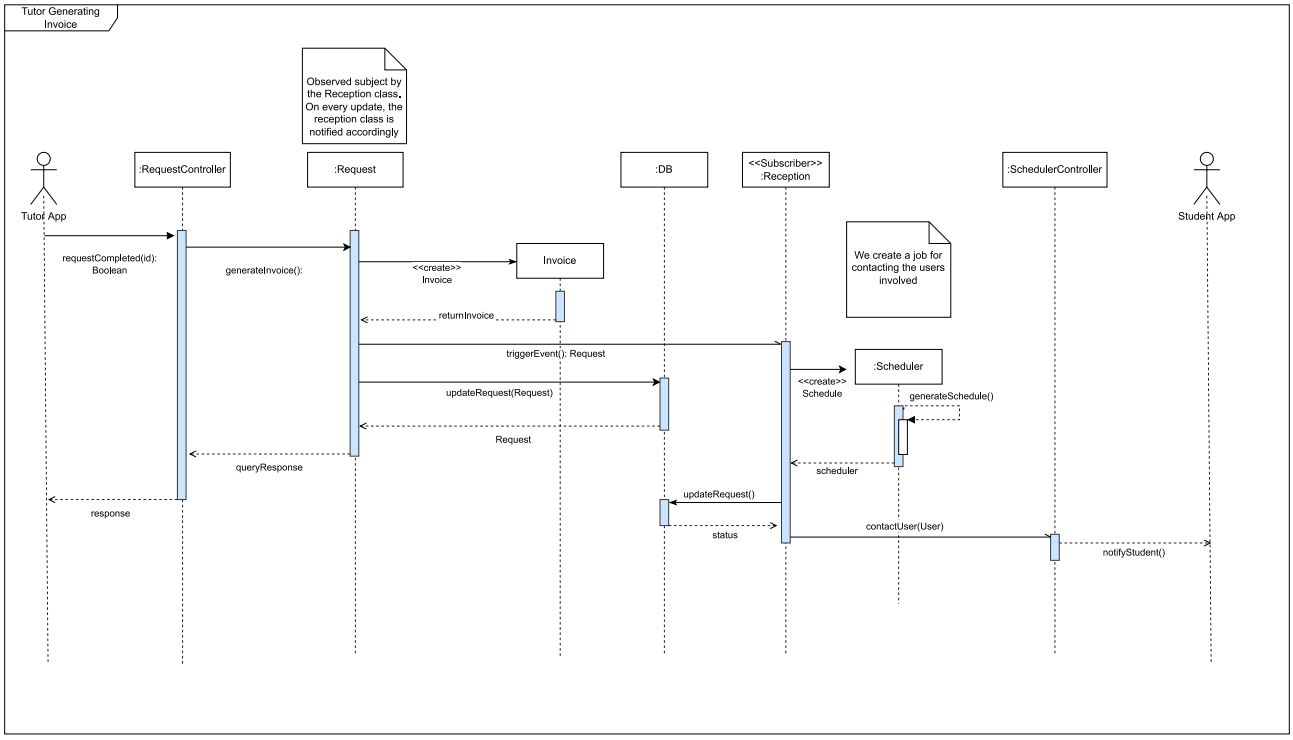


Fig. 7. Tutor responding to a request sequence diagram

The process of a tutor generating an invoice involves several steps, beginning with the tutor invoking the requestCompleted method on the RequestController. The RequestController then generates an invoice using the Request Object, and saves it to the database.

Once the invoice is saved, the reception is notified this is an asynchronous call, and it creates a Scheduler Object. The Scheduler Object is then returned to the reception, which uses it to contact the user. The request is then updated in the database, and a queryStatusResponse is returned to the RequestController.

The RequestController then returns a HTTP response to the tutor, indicating the status of the invoice generation and whether it was successful or not.

9 Implementation (E1 and E2)

The implementation follows behavioural pattern explained in 8. The implementation also considers using Model-View-Controller software architecture which could not be portrayed in 4 Software architecture.

Also utilized design patterns such as Observer for observer and observable and Singletons for most of the controllers to make sure only one instance of a controller is made during the software life time.

The connection between static and behavioural models were made possible by using design patterns explained above and as well as in the classes methods and operations/actions it does. Example: a class RequestController is responsible for creating an object of super class Request(Question or Session).

Link: https://git.its.uu.se/sana2008/ASD.git

Notes about differences between diagrams and code In figure 5, the RequestController sends a general message to create a Request. However, in the code it has been implemented separately, with the RequestController class implementing two methods: createQuestionRequest and createSessionRequest, which create a Request of the type Question or Session. Question and Session are classes that inherit from the Request class. In figure 6 and 5 the result is returned to the user that started the request. However, this return of information has not yet been implemented in the code.

10 Design Patterns (F1 and F2)

10.1 Observer

Problem: The design problem was that there was no way for the system to efficiently notify students and tutors of new requests or of changes in the status of their requests. For example, if a student made a request for tutoring and then changed their availability or added new time constraints, there was no way for the system to notify the tutor of these changes. This led to confusion and frustration for both students and tutors, as they were unable to effectively coordinate their schedules.

The Observer design pattern addresses this problem by providing a mechanism for the MATCH system to monitor and track the state of requests made by students and responses made by tutors.

Solution: The Observer design pattern is useful when you are interested in the state of an object and want to get notified whenever there is any change. In the Observer pattern, the object that watches the state of another object is called Observer and the object that is being watched is called Subject. In our case we have an Observer called Reception and the Subject or Observable is an abstract class called Request (with two concrete implementations in the sub-classes Question and Session). Whenever there is a state change in the Request the Observer will be notified in order to take action. This pattern gives us loose coupling and flexibility.

10.2 Facade

Problem: The contact engine is responsible for a variety of complex tasks, including receiving requests from students, matching those requests with potential tutors, and contacting users through the scheduler component. This complexity can make it difficult for other components of the system to effectively interact with the contact engine, as they may need to understand

and navigate its complex implementation in order to use its functionality.

Solution: The Facade design pattern fixes this problem by providing a simplified interface for interacting with the contact engine subsystem. From our design, it is well elaborated on how the these components of the subsystem contact engine work. The design needs a simple way of encapsulating this complex implementation by introducing a wrapper class between contact engine sub system and request sub system. This wrapper class will offer encapsulated methods to access and use complex interactions offered by the contact engine sub system without needing to understand its full implementation.

11 Refactoring (G1, G2 and G3)

11.1 New Use Case: Payment

11.1.1 Tutors should be able to generate invoice

This functional requirement outlines the steps necessary for a tutor to generate an invoice based on hours worked on an hourly rate.

- The tutor will mark the requested service as done after they have finished with the requested service.
- The tutor will need to provide how many hours they have worked if it deviated with the hours specified by the student.
- The total amount will be calculated from the hours worked with the hourly rate of the tutor.
- The system will then generate an invoice depending on the payment information the tutor has added and saved.
- Once saved the system will send the generated invoice to the student as a "Notification".

11.1.2 Students should be able to make payments for tutoring sessions

The student should be able to pay for completed tutoring sessions. The student should do so as follows:

- Student will receive a notification to confirm and pay the invoice
- Navigate to their user home page.
- Navigating to "Payments"
- Selecting "unpaid invoices"
- Choosing the session they want to pay for, or selecting "pay all outstanding balance"
- Introducing their payment details and selecting whether they want to save the information for future payments
- Clicking on "confirm payment"

11.2 New requested changes, Design refactoring and Planning

11.2.1 Request Class - interface to Abstract

To make the change, we did type introspection 2.12 and reflected 2.13 upon Request Interface class and concluded that the *interface* Request had to be changed to *Abstract* Request. We wanted to implement polymorphism but it was better to use an *Abstract* class than an *Interface*. If we actually did want to implement a common implementation for polymorphism it's better to use abstracts than interfaces because abstract classes allow for both implementation and declaration of methods while interface only allows for declaration of method. So this means with *Abstract* classes, some logic can be shared among sub-classes through the implemented methods while an interface can only provide a contract for methods that must be implemented by sub-classes.

11.2.2 All Controllers in system under design

It's trivial to initiate an object of a class — but how do we ensure that only one object ever gets created? By making the constructor private and be accessed only by members of its class, we make the class a Singleton. Why singleton?

1. It's beneficial when one and only one object is needed to coordinate actions and tasks within the system.

2. Consistency: Singletons ensure that there is only one instance of the middleware, which provides consistency in the behavior of the controllers.

3. Memory Management: Singletons can help with memory management by reducing the number of instances of the middleware that need to be created, which can be particularly important for large-scale applications.

4. Flexibility: Singletons can be easily modified or replaced, which can be useful for implementing changes to the middleware.

In the system mostly all controllers are Singletons, as we only need one object (and only one) to handle operations and actions in the system. A few exampls would be UserController handles actions by the user from a front-end client to the system and RequestController handled actions by the user in the system to create, handle and respond to requests.

11.2.3 Invoice and Billing Information

The invoices used to be a part of the session and the payment information used to be a part of the user. This wasn't very cohesive. "Invoice" and "BillingInformation" have been created as new classes in the refactoring process to solve this issue. We also achieve lower coupling through this change, since any future changes to how payments work don't have to affect classes related to Session and User. This also allows the user have more than one billing/payment information in their user account, since they can have multiple instances of BillingInformation as opposed to User.

Having the ability to add or change multiple payment or billing information for a user in the software-under-design can have a significant impact on its architecture and design.

Firstly, class diagram design will be affected by the new requirement. A new class, PaymentInformation or BillingInformation, will be added to the class diagram to represent the

new data model. This class will have attributes such as paymentMethod, billingAddress, and expirationDate. This class will be associated with the existing User class with a one-to-many relationship.

Secondly, behavioral diagrams such as state diagrams and activity diagrams will also need to be updated to reflect the new functionality. For example, a state diagram for the User class will need to be updated to include states for adding or editing payment information. An activity diagram for the payment process will also need to be created or updated to include the new functionality.

Thirdly, the domain model will be affected as well, it will need to be updated to include the new PaymentInformation or BillingInformation class. Additionally, the domain model will need to be updated to reflect the new business rules and constraints related to adding or editing payment information.

To implement these changes, following plan was proposed:

1. Then update the domain model to include the new PaymentInformation or BillingInformation class and the new business rules and constraints related to adding or editing payment information.

2. Start by updating the class diagram design to include the new PaymentInformation or BillingInformation class and its relationship with the User class.

3. Next, update the behavioral diagrams such as the sequence diagram to elaborate further on how the payment actually works and how the user can create multiple payment information.

12 Reflection (I1 and F5)

12.1 Difficulties (I1)

1. **Lack of time**: One of the difficulties the team encountered is poor time management, which lead to last-minute rushed work and missed deadlines. This may be due to a lack of experience in managing their time effectively, or due to having other priorities such as jobs and other courses. A solution to this problem could be to create a schedule at the beginning of the course and allocate specific blocks of time and deadlines for each task. The team could also use a project management tool to help track their progress and see which areas they may be falling behind.

2. **Insufficient collaboration and meetings**: Another difficulty the team encountered is a lack of sufficient collaboration and communication, leading to missed opportunities for working and problem-solving together. This also led to some members unknowingly working on the same problem separately, wasting time and effort. To address this problem, the team could schedule regular meetings to discuss the progress of their project and work together on submissions. By setting aside dedicated time the team can ensure that they are on the same page and working towards their goals.

3. **Lack of focus and direction**: The team has encountered difficulties in knowing what tasks to prioritize and what their main goals were for the project, leading to confusion and difficulty in organizing and progressing in their work. The main problem with this was

that it limited the team's ability to do as much as they wanted, since they were aiming for a high grade on the project. To address this problem, the team could have reached out to their TA for guidance on structuring their work and identifying their priorities. With additional support and clarification, the team could have better maintained focus and worked on their project more efficiently.

12.2 Ethical Considerations (F5)

12.2.1 Anonymity

At least two glaring ethical issues arise when users are always able to identify other users; especially if their full name in real life is on display.

Collusion: Users might collude to boost their ratings. For instance, each time someone sees an answer posted by a friend they might leave a an undeservedly high rating.

Bullying: Users can 'smite' people they don't like by leaving poor ratings on everything they post.

A simple solution to these problems is to not display names or usernames in threads. Each post will instead have a randomized identity, similar to the animals in google docs for users that aren't logged in. This randomized identity should only be persistent within a question thread. This would allow users to post anonymously without causing confusion. User ratings should still be visible, but with low enough granularity for it to be hard to identify a user by their rating alone.

12.2.2 Plagiarism

Some students will surely plagiarize answers to their question requests if they can't be detected by plagiarism detectors. The simplest solution is to make all answers publicly accessible on a website, indexed by popular search engines, to make it easy for plagiarism detectors to crawl the site like any other site. It's important that we inform the users that their questions and answers will be publicly accessible on the internet, but without their names attached as explained in 12.2.1.

12.2.3 Accessibility

Requiring a personal identification number completely excludes many people from being able to to use the platform. The simplest solution to this is to forego the personal identification number in favor of email. Email addresses are unique for every user and are trivial to verify, unlike personal numbers.

12.2.4 Identity Verification

If two users are to meet up in real life for a tutoring session, it's important that both users are who they say they are. Criminals might use the platform to get their victims to meet them at a certain location. A simple measure to discourage would-be criminals and aid in prosecuting them is to verify the full name and date of birth of users that wish to use the tutoring feature in-person. If this measure is required to use the platform at all trolls won't be able to circumvent bans, but this may also discourage desirable users out of privacy concerns or laziness.

12.2.5 Data Protection

Personal information is any information that can be used to identify an individual, with or without combining it with other publicly available information. Storing such information entails certain responsibilities. Users must be informed of and consent to how their data is stored and used. One approach is to not store personal identifiable information beyond the identification process. This can be achieved by creating a cryptographic proof, signed by the user and the platform. When a user wishes to identify themselves to another user, they provide a message containing their name that is digitally signed. A self-sovereign identity.