POLITECNICO
MILANO 1863

Travlendar+ project YOUR NAMES

Requirement Analysis and Specification Document

Deliverable:	RASD
Title:	Requirement Analysis and Verification Document
Authors:	YOUR NAMES
Version:	1.0
Date:	31-January-2016
Download page:	LINK TO YOUR REPOSITORY
Copyright:	Copyright © 2017, YOUR NAMES – All rights reserved

Contents

List of Figures

List of Tables

1 Introduction

1.1 Purpose

In today's digital age, education has evolved significantly, with online learning becoming more and more important, especially in subjects like computer science. However, only theoretical learning often can be insufficient in practical fields like coding. This is where the CodeKataBattle (CKB) project steps in, addressing the need for a hands-on, collaborative, and engaging approach to learning programming and software development.

Although theoretical knowledge is essential in computer science, it is the practical application and coding practice that strengthen students' understanding and skills. Hence, CKB is designed to support students with a platform where they can actively practice coding, experiment, learn from their mistakes, and improve their skills. Moreover, it encourages students to work together in teams, promoting teamwork and communication, essential qualities in the software development field. For educators, CKB offers a valuable teaching tool, enabling them to create coding challenges in accordance with their curricula, making learning more engaging, and helping assess students' progress.

In summary, the CodeKataBattle project serves as a bridge between theoretical learning and practical application in computer science education. It offers students a platform to practice coding, collaborate, and improve their skills, and provides educators with effective teaching tools and assessment methods. The aim of CKB is to make computer science education more engaging, practical, and rewarding for both students and educators.

1.1.1 Goals

1. Students are able to find, join, or create teams for specific code kata battles within a tournament.
2. Educators can create and manage code kata battles, including the provision of detailed descriptions, project templates with test cases, and build automation scripts.
3. Educators have the capability to set up and customize various battle parameters such as team size limits, registration deadlines, submission deadlines, and scoring configurations.
4. Students commit their code implementations to a specific GitHub repository for each battle, triggering automated testing and scoring through the platform.
5. The platform automatically calculates and updates battle scores based on predefined criteria including the number of passed tests, commit timeliness, and code quality.
6. Educators can manually score submissions, considering aspects outside of automated evaluations.
7. Students and educators can view the evolving ranks and scores of teams during a battle.
8. The final battle ranks are determined after considering both automated and manual evaluations, and made visible to all participants upon completion of each battle.
9. Cumulative personal tournament scores are updated and displayed for each student, contributing to an overall tournament ranking.
10. The platform notifies students of new tournaments, battle opportunities, and updates on their tournament standings.
11. Students can view their rankings on their profiles.

1.2 Scope

1.2.1 Product Identification

- The **target audience** of the platform is mainly **students and educators** in computer science and related fields.
- CKB is an **interactive web-based coding platform** specifically designed to improve the coding skills of students by participating in battles designed by educators. It is mainly an **educational tool** where practice can be applied.
- CKB is also a **competitive learning environment**. By organizing the coding exercises in battle format, the students are expected to be more motivated to perform their best in challenges.
- The platform **promotes teamwork and collaboration** by allowing students to form groups to tackle battles together. This reflects real-world software development scenarios where collaborative team dynamics are key.
- The platform has an **educator-centric control**. The educators play a crucial role in the platform because they are the ones creating, managing, and scoring the battles and tournaments.
- CKB enables students to **use professional tools and practices** such as version control, and fork-pull-push mechanisms so that they can enhance their real-world software development skills with the help of Github.
- The platform makes use of **automated testing** to assess student submissions, ensuring objective evaluation of functional correctness and code quality. Additionally, it allows educators to perform manual evaluations for aspects that require subjective judgment.
- CKB is designed to provide **instant feedback** on submissions and **real-time updates** of team scores and rankings.
- The platform offers **flexibility** to educators in terms of the choice of programming languages, difficulty levels of challenges, and the scope of coding tasks, making it adaptable to various learning curves and educational needs.

1.2.2 Domain Analysis

The CKB platform operates within the domain of educational technology, specifically tailored for coding and software development education. From this perspective, we have the following users:

Educators: Educators use CKB to **create and manage coding exercises**, known as code katas, in a battle format. They have the capability to set parameters for these exercises, including difficulty levels, deadlines, and specific technical requirements. Educators can **evaluate student submissions** both automatically (using the platform's tools) and manually, **providing feedback and scores**.

Students: Students exploit the platform to enhance their coding skills by **participating** in these code battles. They work individually or in teams to solve the challenges set by educators, fostering both individual and collaborative learning experiences. Students use the platform **to submit their code, receive real-time feedback, and track their progress in coding proficiency**.

In summary, the domain of the CKB platform is focused on interactive, competitive coding education, enabling students to have an engaging learning experience while offering educators powerful tools for managing and evaluating coding exercises.

1.2.3 World Phenomena

1. Educators prepare the necessary documents for code battles including descriptions, test cases and build automation scripts.
2. Students fork the GitHub repository.
3. Students set up an automated workflow through GitHub Actions.
4. Students write code on their devices.

1.2.4 Shared Phenomena

World Controlled

1. Educators upload coding challenges, including descriptions and test cases, to the platform.
2. Educators create tournaments and give permission to their colleagues to create battles for that tournament.
3. Educators set specific rules and criteria such as deadlines, number of team members, and additional configurations for scoring for code kata battles.
4. Educators decide on the inclusion of manual scoring components for battles.
5. Students invite peers to form teams within the platform or join individually.
6. Students submit their code solutions by pushing the code via GitHub.

Machine Controlled

7. The CKB platform sends notifications to students about new battles and tournaments, final battle ranks and final tournament ranks.
8. The platform generates and manages GitHub repositories for each battle and sends links to students that are participating.
9. The CKB platform automatically evaluates code submissions against test cases.
10. The platform updates battle scores and battle rankings in real-time.
11. The platform displays leaderboards (i.e. the rank of the sum of all battle scores received in that tournament).

1.3 Definitions, Acronyms, Abbreviations

1.3.1 Definitions

- **Student**: An individual enrolled in an educational program or course who uses the platform to participate in coding exercises and improve software development skills.
- **Educator**: A person, such as a teacher or an instructor, responsible for creating coding challenges and managing learning activities on the platform.
- **Automated Testing**: A process where the CKB platform automatically executes predefined tests on student code submissions to assess their functionality and correctness without manual intervention.

- **Manual Scoring**: The process where educators evaluate student code submissions subjectively, complementing the automated testing system.
- **Battle**: A competitive coding challenge on the platform where students or teams of students solve specific programming problems within set parameters and time frames.
- **Tournament**: A series of code kata battles organized and managed by educators on the CKB platform, that ranks students or teams based on cumulative scores from individual battles.
- **Ranking**: A system within the CKB platform that orders participating students or teams based on their performance in individual code kata battles, determined by scores from automated and manual evaluations.
- **Leaderboard**: A feature on CKB that displays the standings of students or teams based on their performance overall in a tournament.
- **Institution Information**: Institution Information is multiple choice of institutions for the educators, a single institution for the students.
- **Unregistered User**: Users that haven't registered to the platform yet.
- **Registered User**: User that have registered to the platform.
- **Authenticated User**: User that have logged in to the platform.
- **Availability**: Availability is the status of tournament in terms of Closed, Open, or Upcoming.
- **Own Tournaments**: Own Tournaments means the tournaments they created from Educator perspective, on the other hand, it means the tournaments they registered from Student perspective.
- **Own Battles**: Own Battles means the battles they created from Educator perspective, on the other hand, it means the battles they registered from Student perspective.
- **Scoring Criteria**: Scoring Criteria includes Test Cases, Timeliness and Quality.

1.3.2 Acronyms

- **CKB**: CodeKataBattle

1.3.3 Abbreviations

- G_x: x-th Goal
- WP_x: x-th World Phenomena
- SP_x: x-th Shared Phenomena

2 Overall Description

2.1 Product Perspective

2.1.1 Scenarios

1. **Educator creates a tournament**
   Professor Emanuelle, a computer science educator giving an "Introduction to Programming" course in Politecnico di Milano, decides to improve her student's understanding of programming through coding exercises. She is already registered in CodeKataBattle as an educator, so she logs in to the platform with his credentials, which are email and password. Then, she clicks the "Create Tournament" and a form screen is prompted. She fills out the form respectively:
   - entering tournament title, "Winter Semester Coding Exercise 1",
   - adding a description of the tournament,
   - choosing a deadline for the subscription. After clicking the "Next" button
   Professor Emanuelle is asked to select colleagues for permission to create battles in this tournament. She selects 2 of her colleagues because they all together gives the same lecture. Finally, she clicks on the "Create" button, and tournament is created. She is directed to the "My tournaments" section after creation.

2. **Student Registers for tournament**
   Emre is a Bachelor's student in Computer Science at Politecnico di Milano. At the start of the winter semester, he registered CodeKataBattle as a requirement of his "Introduction to Programming" course. In the middle of the semester, he gets an email notification about a tournament created by Professor Emanuelle on the platform. Then, He clicks the link in the email. Because he has already logged in to the platform with his email and password, he is redirected to the tournament's page, where he finds detailed information about Emanuelle's tournament. After reading the description, he thinks that this tournament will be very helpful for him to understand the topics covered in the lecture. So he clicks to the "Register" button to enroll tournament. Some registration details consisting of descriptions, educators, and tournament creator are shown on the screen. Also, he is informed that he will get email notifications about upcoming battles, battle rankings and tournament rankings. Finally, Emre clicks on the "Accept and Register" button and registers the tournament.

3. **Educator Sets Up a Battle**
   A week ago Professor Mottola, was invited to a tournament created by his colleague in the CodeKataBattle platform. He accepted the invitation and learned about the tournament from its description. He also gives the "Introduction to Programming" course and wants to create a new exercise about the topic he showed in the last lecture, which is "Recursion". After the subscription deadline passed, he logs back into CKB, he selects the "Create Battle" option within the "Winter Semester Coding Exercise 1" tournament. He is presented with a detailed form where he inputs the battle's title, "Check if String is Palindrome" and provides a thorough description that includes the battle's focus on recursion, expected coding languages (Java and Python, both accepted), and software project including necessary scripts for build automation (Gradle for Java, no need for Python) and test cases. To encourage collaboration, he sets the minimum and maximum group size to three and five students, respectively. he then specifies the registration deadline, two weeks from the current date, and a final submission deadline, giving students a month to work on their solutions. Finally, he configures additional scoring parameters as **efficiency**, choosing from a list of aspects including reliability, maintainability, etc. Then he clicks on the "Create" button and he is redirected to battle main page illustrating information about the battle. After a couple of minutes, he got an email saying that an email about this battle was sent to all students in the tournament.

4. **Student Joins for Battle**
   Samet got a notification from the tournament he enrolled in saying that Professor Mottola has created a battle with the name "Check if String is Palindrome". He clicks on the link and reads the description carefully. He clicks on the "Register" button then a screen is shown asking Samet whether he wants to invite other students to form a team or not. Samet selects "Yes", then, he names her team 'Code Warriors' as a first step. Then he chooses students from a list of students registered for the tournament, considering minimum and maximum group size. Samet knows Emre, Jack, and Luca also registered for the tournament, so he sends them invitations. After clicking "Complete", he is redirected to the battle information page. On this page, there is a section showing group status, which is pending until all invitations are answered. After a while, Samet sees that Emre and Luca accepted but Jack rejected it. Because the minimum group size is fulfilled, Samet clicks on the "Finalize" button indicating the final decision. Thanks to the help of instructions during the process, Samet understands that if he wants to decline registration, he should have clicked on the "Decline" button. After every member accepts the invitation to finalize registration, the process ends and the status becomes "Registered". They are given a competitor id.

5. **Students Sets Up Environment for CodeKataBattle**
   With the 'Code Warriors' team formed and the registration deadline for the "Check if String is Palindrome" battle passed, the CKB platform takes its next automated step. It creates a unique GitHub repository for the battle, containing the provided code kata with its test cases and build scripts. The system then sends an email to Emre, Samet, and Luca with the repository link and instructions. The team members collaboratively decide to schedule a virtual meeting to set up their working environment. During the meeting, they fork the repository to their group account and set up GitHub Actions in order to make proper API calls with the competitor id they have. This setup is crucial for automating their workflow and ensuring that every code push not only updates their repository but also notifies the CKB platform. They test the setup by pushing a minor change, and upon seeing that the CKB platform acknowledges their commit, they know their system is correctly configured. This marks the start of their coding journey in the battle.

6. **Students Solve Battle Challenge**
   After forking the project and setting environment, they started to think about the solution of the project. As they understand from the description they should return "True" or "False" regarding whether the given string is a palindrome. They implement the algorithm and commit/push the code to the repository. Each push prompts the platform to pull the latest code, run tests, and analyze the quality of their solutions using static analysis tools. After a while, they revisit the ranking on the battle's page to see their score of the last push. They see their scores in 3 different categories:
   - functional aspects: 32/80 (4/10 test cases passed)
   - timeliness: 5/5
   - quality level of the sources: 8/10 (Aspects: Efficiency)
   In the rankings, they are informed via tooltips about the scaling of their scores and the calculation methods of them. So they decided to focus more on code and try to find why some test cases are not passed. The team keeps an eye on the CKB dashboard, which updates their battle score after each commit. They note improvements in their score as they refine their solutions, ensuring more test cases pass and optimizing their code for better quality. This iterative process of coding, committing, and refining continues, with the team members frequently discussing strategies and sharing insights to improve their solutions. After looking at some exercises related to recursions they finally find the wrong part in the algorithm they implemented. After changes they commit and push the code. They see in the rankings that they got 93 points from battle. This iterative process of coding, committing, and refining continues, with the team members frequently discussing strategies and sharing insights to improve their solutions. They think that this is the most efficient

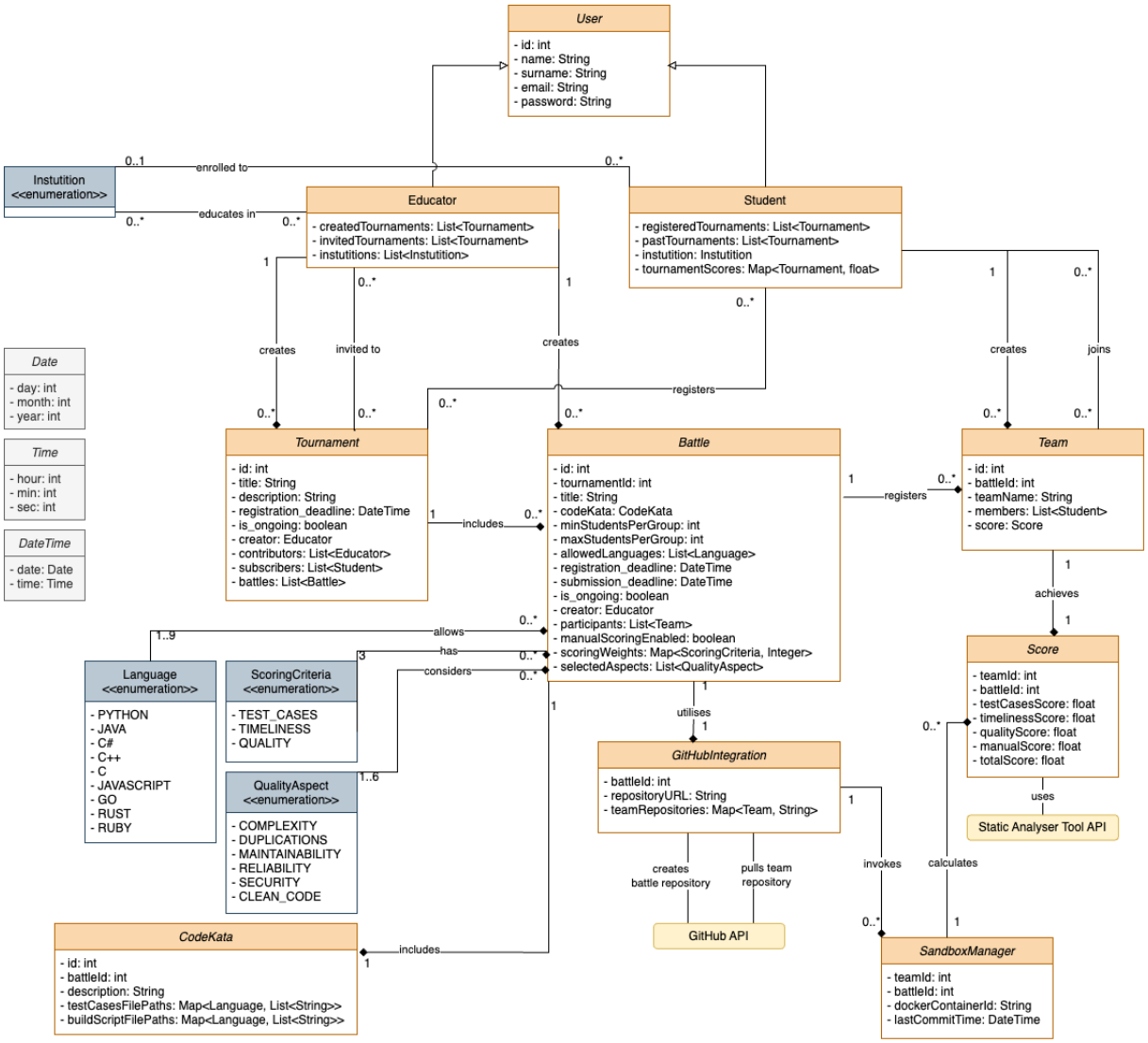
algorithm they can implement. So they decide not to do anything else until code kata battle deadline.

7. **Educator Evaluates Submission Manually**
   After the submission deadline expires, there is a consolidation stage enabling educators to assign optional points for teams. Professor Mottola begins her manual evaluation of the submissions for the "Check if String is Palindrome" battle. He logs into CKB and accesses the educator's dashboard from the battle's page, where he can review each team's submission. On this page code of the submission and other materials from the repository are shown to the educator. He starts from the latest submission and gives extra points if he finds the solution by analyzing 'Code Warriors', impressed by their timely submission and high score, viewable in the rankings table. Professor Mottola examines their code, focusing on the efficiency of their solution, which got 8/10 from the analysis tool. Considering these factors, she awards them a high personal score. This score reflects her assessment of their problem-solving skills and coding proficiency, adding a crucial human element to the automated evaluation.

8. **Educators Analyze Tournament Leaderboard**
   Professor Emanuelle have created a tournament called "Winter Semester Coding Exercise 1" at the beginning of the semester. Some other colleagues of hers registered for the tournament as educators and created koda battles during the semester. During the semester, she has viewed the leaderboard and at the end of every week, she has exported the result. Eventually, at the end of the semester, Professor Emanuelle closes the tournament. She then goes to the leaderboard screen and starts to see the general success of her students by comparing leaderboard exports from the very first battle. In this way, she notices that the leaderboard becomes more competitive as we get closer to the end of the year. She interprets this situation as students improving their programming skills battle by battle. After analyzing the leaderboard she searched other ongoing tournaments of other professors from the university to see the success of their students in different tournaments created by teachers of different courses.

2.1.2 Domain Class Diagram

In the figure below, the Domain Class Diagram for the CodeKataBattle (CKB) platform is illustrated. This diagram is a crucial element in understanding the structure of the classes and relationships within the CKB system. It visually represents the key classes, their attributes, and the interactions among them in the context of the platform. This diagram is designed to provide a clear and concise overview of the domain model, offering insights into how the different components of the CKB platform interact to facilitate a competitive and educational coding environment.



The descriptions below are the explanations of the important classes that take place in the diagram:

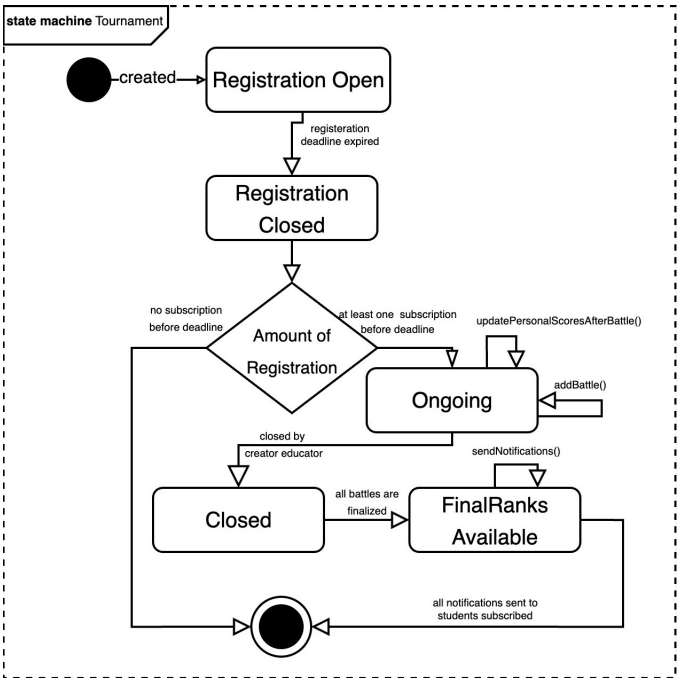
- **User:** Serves as a base for different types of users. It encapsulates common attributes that are shared across students and educators. This class is essential for authentication.
- **Educator:** The objects of this class have the ability to create tournaments and battles. They can also be invited to a tournament by the creator of that tournament.
- **Students:** The objects of this class have the ability to register for the tournaments, and they can create teams for battles or join teams that they are invited to.

- **Tournament:** Tournaments are a vital part of the project. They are created by educators and they include coding battles for teams of students to engage and solve.
- **Battle:** Battles are created by educators who take part in a tournament. During battle creation, educators set some of the fields of the battle object.
  - **Scoring Criteria:** There exists 3 scoring criteria whose percentages are decided by the educator during battle creation.
  - **Language:** There are 9 languages allowed in the system. The educator chooses the allowed languages during battle creation. At least one language must be chosen.
  - **Quality Aspect:** There are 6 quality aspects that can be selected by the educator during battle creation time. At least one quality aspect must be chosen.
  - **CodeKata:** It is the most important part of the battle. The test cases and the build scripts for each allowed language should be uploaded by the educator to the system. The paths to the files in the file system are stored in the fields.
- **Team:** Teams are created by students, and the teams are able to register for the battles. They have a score for the battle.
- **GitHub Integration:** This class has one object for each battle and it stores the link to the GitHub repository for the battle after creating it by interacting with the GitHub API. It also stores the links to the repositories of the participating teams after they push their code and trigger a function that belongs to this class.
- **Sandbox Manager:** There exists a Sandbox Manager for each team in the corresponding battle. If there exists a docker container (the team previously pushed a code and their GitHub actions already communicated with our RESTful APIs), the same container is called, and the code is run on that container. If there is not an existing container (the team pushes code for the first time), a new container is created and assigned to the team for the rest of the battle. The Sandbox Manager interacts with the Score class after running the code to calculate the score of the team after each push/commit.
- **Score:** The score of a team for each criterion is calculated for the battle. This class uses an external Static Analyser Tool to calculate the score of the quality aspect of the code. The timeliness and test case scores are handled internally.

2.1.3 Statecharts

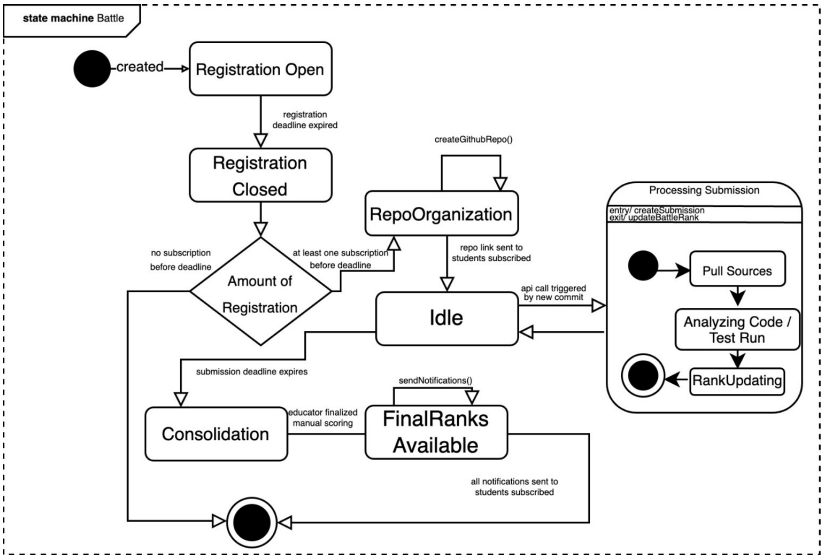
The state charts presented in this section elucidate the operation of the CodeKataBattle (CKB) project, illustrating the state transitions for a code kata battle and a tournament within the platform. Especially for tournaments, a state chart is very crucial to gain a better understanding of the tournament.

Tournament



Firstly, the tournament is in the very first state called "Registration Open", which indicates that Tournament is created and registration is open for students and invited educators. Until the registration deadline, they can register. When the deadline expires registration is closed, so the tournament goes into the "Registration Closed" state. In this state, the process controls the amount of registration for the tournament. If there is no registration it goes to "Final" state. Otherwise, the tournament starts with the "Ongoing" state. In this state, throughout the tournament, educators can create battles, and personal scores are updated after every battle. If the creator educator closes the tournament, it goes to the "Closed" state. However, there can be some battles in the consolidation stages and it is mandatory to finalize them to calculate final ranks. After every battle is finalized, the tournament transitions to the "FinalRankAvailable" state in which notifications about final rankings are sent to the students. After finishing this notification process tournament reaches the "Final" state. There won't be any rank changes or notifications hereafter.

Battle



Firstly tournament is created with the "Registration Open" state. In this state, students can register to battle with teams or individually. When the deadline expires, Battle goes into the "Registration Closed" state. At this time, it is calculated that if there is any registration for battle. If no, it goes to the "Final" state with no registration and any other action. If there are registrations, it goes to a new state called as RepoOrganization in which GitHub repository for the battle is created with proper instructions. Sending the link and instructions triggers the process to go into the "Idle" state. This means that Battle has started and waits for API calls from forked repositories, triggered by a new commit. An API call from repositories of competitor leads to go into "Processing Submission. Here with the "createSubmision" input 3 sequential states follow each other, respectively, "Pull Sources", "Analyzing Code / Test Run" and "RankUpdating". In this 3 state code is pulled and analyzed. Then, the score and rankings are updated. Eventually, it returns to the "Idle" state. Until the submission deadline, students' commit can change their scores. When the deadline expires, Battle goes into the "Consolidation Stage" state in which educators can change rankings via manual scorings. When they are finalized, Battle is closed, indicating as state "FinalRanksAvailable". Immediately, the notification sending process starts. After all notifications are sent, Battle goes into the Final state.

2.2 Product Functions

The major functions of the project will be organised according to the user types. We will have three sections, first for the common function both regarding the educators and the students. The second and the third will be specific to each user type respectively.

2.2.1 Common Functions

- **Account Registration:** Both students and educators can be registered to the platform. During registration, the user indicated which type of user they are along with their credentials.
- **Viewing Leaderboards and Rankings:** Both user types can view leaderboards that display rankings of participants based on performance in battles and overall tournament standings.

2.2.2 Educator Functions

- **Creating a Tournament:** Educators can create and configure tournaments, setting specific parameters such as time constraints. In these tournaments, multiple battles can take place.
- **Creating a Battle:** Educators can create and configure coding battles in the tournament that they have created or they have been permitted by their fellow educators. During battle creation, specific parameters such as coding languages, and team sizes are set by the educator. On top of that, the description of the battle and the test file including test cases, automated build scripts or manual evaluation choice should be provided.
- **Giving Permission to Other Educators:** This function can be utilised both in the creation of a tournament and during the management of a tournament. Educators are able to permit fellow educators to create battles in the tournaments that they have created.
- **Scoring:** This function is vital since it is used to compute the points of the students' submissions in the battles. The output of this method determines the ranking of the students. Both automated and manual scoring reply to this function call.

2.2.3 Student Functions

- **Join a Tournament:** Students can join tournaments that the educators are created.
- **Join a Battle:** Students can join battles which is a part of the tournament they already joined. When joining the battle, students can send requests to other students to form a team if they want or need to.
- **Send Request for Teaming:** This function can be called during students join the battle. Students send requests to other students using their usernames, and if they accept the invitation, they are now a part of the team for the battle they are joining.
- **Submit Solution:** Students submit their solutions for each code battle. After submission successfully completed, the scoring function is called.

2.3 User Characteristics

2.3.1 Educators

A. **Profile**

- Typically instructors, professors, or teachers in computer science or related fields, with varying levels of experience in coding and software development education.

B. **Needs**

- Tools for creating, managing, and monitoring coding challenges and tournaments.
- Flexibility in setting parameters for battles, including team sizes, deadlines, and testing criteria.
- Ability to assess student work both automatically or manually.
- Resources to track student progress and engagement in coding exercises.

C. **Interactions**

- Making students engage with battles for educational purposes.
- Collaboration with other educators within or across institutions.

2.3.2 Students

A. **Profile**

- Individuals enrolled in computer science and similar courses or those seeking to enhance their coding skills, ranging from beginners to advanced levels.

B. **Needs**

- Access to a variety of coding challenges that help to improve the student's skills.
- Tools for collaborative coding and team formation.
- Real-time feedback on code submissions for iterative learning.
- Opportunities to apply theoretical knowledge in practical scenarios.

C. **Interactions**

- Active participation in coding battles and tournaments.
- Collaboration with peers for team-based challenges.
- Utilisation of platform resources for self-directed learning and improvement.

2.4 Assumptions, Dependencies, and Constraints

2.4.1 Domain Assumptions

1. Educators and students have basic proficiency in using web-based platforms and are familiar with basic operations such as account creation, logging in, and navigating through a digital interface.
2. Educators have the necessary skills to create and manage coding challenges, including the ability to correctly write problem descriptions, test cases, and understand code quality metrics.
3. The coding problems and challenges provided by educators are free from errors and ambiguities.
4. Students have at least foundational knowledge in programming and can understand and respond to coding challenges.
5. Students' submissions to the platform are their original work.
6. The users have access to reliable internet connectivity and devices capable of supporting the web-based CKB platform.
7. Students have familiarity with GitHub operations such as forking a repository, setting up GitHub Actions, and committing and pushing their codes.
8. The automated testing and scoring systems within the CKB platform are trusted by users to fairly and accurately assess coding submissions. Similarly, educators are trusted in the case of manual evaluation of submissions.

2.4.2 Dependencies

- **GitHub Services:** The system's functionality for code submission and automated workflow relies on the availability and reliability of GitHub services.
- **Internet Connectivity:** Access to the internet is essential for both users.
- **Web Browser Compatibility:** The platform's user interface and features depend on their compatibility with various web browsers, ensuring all users can access and use the platform effectively.
- **Hosting Services:** Dependence on reliable hosting services (like cloud-based servers) for hosting the web application, database, and related services.
- **Notification Service:** The platform depends on an external notification service to send timely alerts and updates to students and educators.
- **Email Verification Service:** The platform relies on an external email service provider to send verification codes to users during the account registration process.

2.4.3 Constraints

- **Technological Constraints:** Compatibility requirements with various web browsers and devices may limit certain design or feature implementations.
- **Resource Constraints:** Hosting and operational costs may limit the extent of scalability and redundancy features of the platform.
- **Security and Privacy Constraints:** The system must adhere to strict data protection and privacy regulations, such as GDPR, which may limit certain data collection and processing activities.

- **Legal and Compliance Constraints:** Compliance with intellectual property laws, particularly in the use and distribution of coding challenges and educational content, i.e. the rights of the contents are reserved.

3 Specific Requirements

3.1 External Interface Requirements

3.1.1 User Interfaces

3.1.2 Hardware Interfaces

The CodeKataBattle platform operates entirely through web interfaces, eliminating the need for specialized hardware interfaces. Users can access the platform using standard web-enabled devices, such as computers, tablets, and smartphones. The platform is designed to function within a web browser, requiring no specific hardware beyond a device capable of running a modern browser and accessing the internet. This approach ensures broad accessibility without necessitating particular hardware configurations, making the platform versatile and user-friendly across various hardware setups. However, for optimal experience, at least 2GB of RAM and a minimum of 1GB of free space are recommended for browser caching and temporary files.

3.1.3 Software Interfaces

Our platform CKB interfaces with various software systems:

- **Web Browsers:** Platform operates on web browsers. For universal access and easy use, it is compatible with major browsers like Chrome, Edge, Safari, Opera, and Firefox.
- **GitHub API:** Integrated for creating coding repositories and managing code *submission* processes.
- **Sandboxing:** To create an isolated testing environment, our Sandbox Manager will utilise Docker.
- **Static Analysis Tool:** SonarQube API will be integrated for scoring the quality aspect of the code with static analysis.
- **Database Technology:** Platform uses databases such as PostgreSQL.
- **Hosting:** Our platform uses Amazon Web Services EC2 for web server and database hosting.
- **Cloud File Storage:** The files will be stored using Amazon S3 services.
- **Email Service:** Amazon Simple Email Service SES will be used to send automated email notifications.

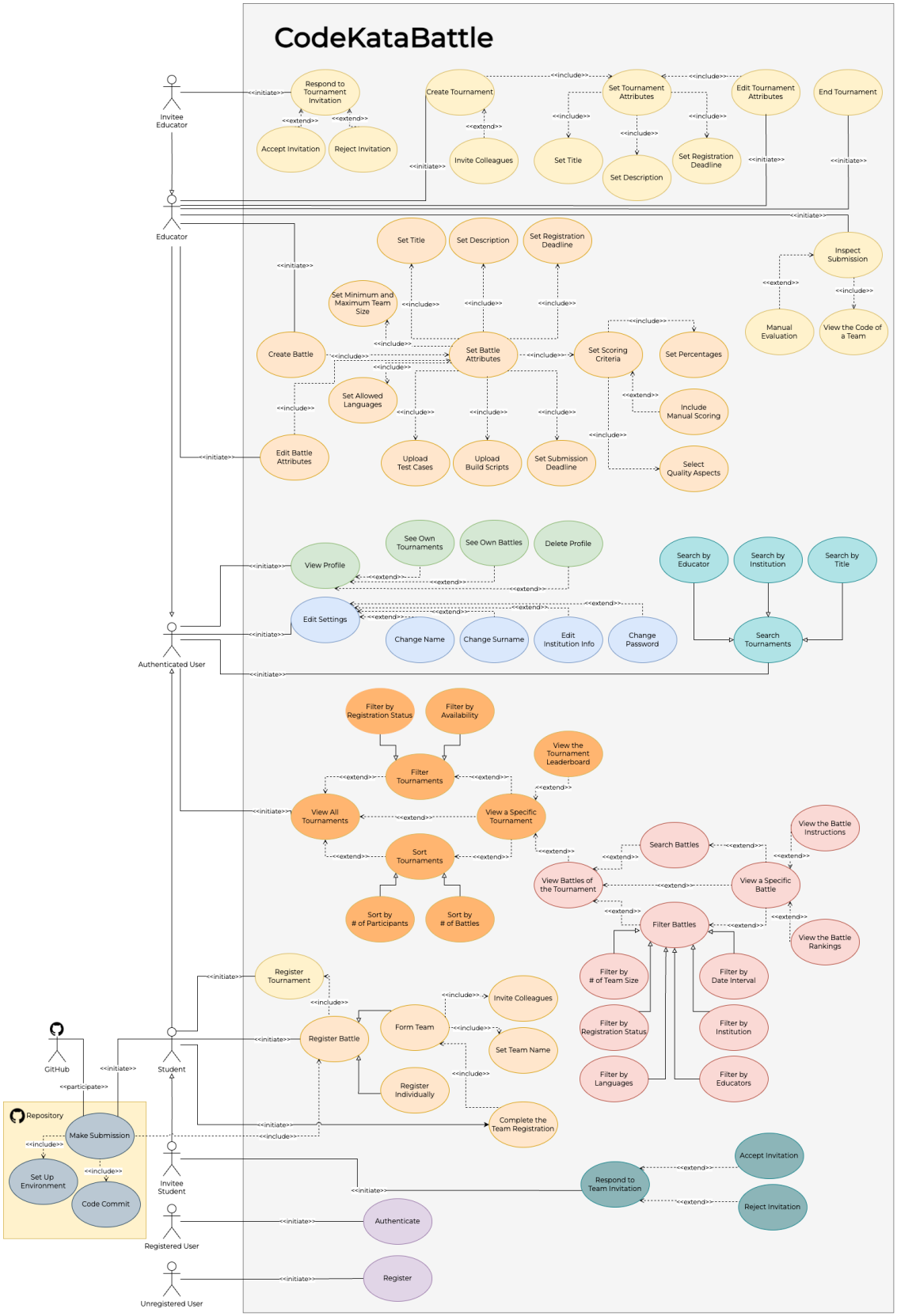
3.1.4 Communication Interfaces

Our platform utilises various communication systems:

- **HTTPS:** For secure connection over the internet.
- **RESTful APIs:** We have RESTful APIs to communicate with the requests coming from GitHub Actions and the web application.
- **SMTP:** It is used to manage to send automated emails.

3.2 Functional Requirements

3.2.1 Use Case Diagram



Unregistered User Actor has been used to illustrate the Registration Use Case. Similarly, the Registered User Actor has been used to show the Authentication Use Case. Other user actors are assumed to be registered and authenticated.

Our system is composed of two main user types: Student and Educator. In the use case diagram, we showed common use cases with the Authenticated User actor. Authenticated User, simply every user registered to the platform, can view Tournaments, Battles, and their rankings with different filters and search options. Moreover, Registered User has the capability of viewing their profile and editing settings.

Use case related Educator and Student Actors summarizes the goal of these users in the system.

Invitee Student and Invitee Educator Actors are used to demonstrate use cases for tournament (for educators) and team (for students) invitations.

GitHub System Boundary contains use cases about code submission with the participation of GitHub API Actor.

3.2.2 Use Cases

The Use Case Tables below, depict the possible use cases in detail. We tried to cover all the use cases that are included in the Use Case Diagram (see 3.2.1). However, to reduce the complexity and the number of tables, some use cases that are strongly related to each other have been introduced in the same use case table. To keep track of them, they are written in *italic* in the Event Flow if they occur. Additionally, the Related Use Case(s) row is added to the tables, to see explicitly which use cases from the Use Case Diagram are included in the table.

1. Register

Name	Register
Actor(s)	Unregistered User
Entry Condition	The user is unregistered to the platform.
Event Flow	(1) The actor enters the platform and clicks the Sign Up button. (2) The actor fills out the registration form by providing email, password, name, surname, and institution information. (3) The actor clicks the Sign Up button to *register*.
Exit Condition	The actor is registered and the Login page is displayed.
Related Use Case(s)	• *Register*

2. Login

Name	Login (i.e. Authenticate)
Actor(s)	Registered User
Entry Condition	The user is already registered to the platform.
Event Flow	(1) The actor enters the platform web page. (2) The actor fills out the form by providing an email address and a password. (3) The actor clicks the Login button to *authenticate*.
Exit Condition	The actor is logged in and the dashboard is displayed.
Exception(s)	• The email and password do not match.
Related Use Case(s)	• *Authenticate*

3. View a Tournament

Name	View a Tournament
Actor(s)	Authenticated User
Entry Condition	The actor is logged in.
Event Flow	(1) The actor clicks the Tournaments button from the sidebar. (2) In the tournaments page, the actor *views all tournaments* as cards. (3) The actor *filters tournaments* by registered filter. (4) The actor *sorts tournaments* by # of participants. (5) The actor clicks the first tournament card to *view a specific tournament*.
Exit Condition	The specific tournament that the actor wants to view is displayed.
Exception(s)	• There are no existing tournaments. • There are no tournaments with the filter set by the actor.

Related Use Case(s)	• *View All Tournaments*
	• *Filter Tournaments*
	  – *Filter by Registration Status*
	• *Sort Tournaments*
	  – *Sort by # of Participants*
	• *View a Specific Tournament*

4. View the Tournament Leaderboard

Name	View the Tournament Leaderboard
Actor(s)	Authenticated User
Entry Condition	The actor is viewing a specific tournament.
Event Flow	(1) The actor clicks the Leaderboard button to *view the tournament leaderboard*.
Exit Condition	The leaderboard of that tournament is displayed.
Related Use Case(s)	• *View the Tournament Leaderboard*

5. View a Battle

Name	View a Battle
Actor(s)	Authenticated User
Entry Condition	The actor is viewing a specific tournament.
Event Flow	(1) The actor *views the battles of the tournament* in that tournament's page. (2) The actor *searches battles* using the search bar at the bottom of the page. (3) The actor *filters battles* using the filter section on the right-hand side of the page. (4) The actor chooses one of the results to *view a specific battle*.
Exit Condition	The specific battle that the actor wants to view is displayed.
Exception(s)	• There are no existing battles in that tournament. • There are no battles with the filters set by the actor.
Related Use Case(s)	• *View Battles of the Tournament* • *Search Battles* • *Filter Battles* • *View a Specific Battle*

6. View the Battle Instructions

Name	View the Battle Instructions
Actor(s)	Authenticated User
Entry Condition	The actor is viewing a specific battle.
Event Flow	(1) The actor clicks the Instructions button to *view the battle instructions*.
Exit Condition	The instructions of that battle are displayed.
Related Use Case(s)	• *View the Battle Instructions*

7. View the Battle Rankings

Name	View the Battle Rankings
Actor(s)	Authenticated User
Entry Condition	The actor is viewing a specific battle.
Event Flow	(1) The actor clicks the Scores button to *view the battle rankings*.
Exit Condition	The rankings of that battle are displayed.
Related Use Case(s)	• *View the Battle Rankings*

8. Inspect a Submission

Name	Inspect a Submission
Actor(s)	Educator
Entry Condition	The actor is viewing the battle rankings of a specific battle.
Event Flow	(1) The actor clicks one of the teams from the rankings table to *view the team's submission*. (2) The actor looks at the code and decides to *manually evaluate*. (3) The actor enters the bonus points in the respective field and clicks the Add Scoring button.
Exit Condition	The total score after manual evaluation of that team are displayed.
Exception(s)	• The manual scoring is not enabled.
Related Use Case(s)	• *View the Code of a Team* • *Manual Evaluation*

9. Search Tournaments

Name	Search Tournament
Actor(s)	Authenticated User
Entry Condition	The user is logged in.
Event Flow	(1) The actor clicks the button in the search area to select the search criteria. (2) The actor sees three buttons: *Search by Educators*, *Search by Titles*, *Search by Institutions* and chooses to search by title. (3) The actor writes "Fall'23 Tournament" to the search bar and hits enter to *search tournament*.
Exit Condition	The tournaments with the corresponding title are displayed.
Exception(s)	• The tournament with the given title does not exist.
Related Use Case(s)	• *Search by Educator* • *Search by Title* • *Search by Institution* • *Search Tournament*

10. Register to a Tournament

Name	Register to a Tournament
Actor(s)	Student
Entry Condition	The actor is logged in and the tournaments page is on display.
Event Flow	(1) The actor clicks a card of a tournament in which s/he is not registered. (2) The actor clicks the Register button in the pop-up to *register the tournament*.
Exit Condition	The actor is registered to the tournament and the page of that tournament is displayed.
Exception(s)	• There are not any tournaments with not registered status.
Related Use Case(s)	• *Register Tournament*

11. Register to a Battle

Name	Register to a Battle
Actor(s)	Student
Entry Condition	The actor registered to the tournament in which the battle takes place and that tournament's page is on display.
Event Flow	(1) The actor clicks a register button next to the battle that s/he wants to participate in. (2) The actor clicks the Register button in the pop-up. (3) The actor chooses between to *form a team* and to *register individually*. (4) If there is going to be a team, the actor *sets a team name* and *invites colleagues*. (5) The actor clicks the Complete button to *register to the battle*.
Exit Condition	• If registered individually, the actor is registered to the battle. • If a team is formed, the invitation requests are sent to the colleagues.
Exception(s)	• The actor is already registered for all battles in that tournament.
Related Use Case(s)	• *Register Battle* • *Form Team* • *Register Individually* • *Invite Colleagues* • *Set Team Name*

12. Complete the Team Registration for the Battle

Name	Complete the Team Registration for the Battle
Actor(s)	Student
Entry Condition	The specific battle that the actor wants to complete the registration is on display.
Event Flow	(1) The actor can see which colleague accepted the invitation and which colleague rejected it. (2) The actor can finalize the team formation if there are enough colleagues on the team by using the Finalize button, or discard the team and battle registration entirely by using the Decline button to *complete the team registration*.
Exit Condition	• If finalized, the team is registered to the battle. • If declined, the team is not registered to the battle and the dashboard is displayed.
Related Use Case(s)	• *Complete the Team Registration*

13. Respond to the Team Invitation

Name	Respond to the Team Invitation
Actor(s)	Invitee Student
Entry Condition	The actor is logged in and invited to a team by another student.
Event Flow	(1) The actor opens up the notifications and sees the invitation to join a team formed by another student. (2) The actor clicks the notification and a pop-up shows up. (3) The actor can choose to *Accept Invitation* by clicking the Accept button or *Reject Invitation* by clicking the Reject button to *respond to the invitation*.
Exit Condition	• If accepted, the specific battle is displayed. • If rejected, the dashboard is displayed.
Related Use Case(s)	• *Accept Invitation* • *Reject Invitation* • *Respond to Team Invitation*

14. Make Submission

Name	Make Submission
Actor(s)	Student, GitHub API
Entry Condition	The student actor forked the battle repository, and initiated GitHub Actions on their repository.
Event Flow	(1) The student actor *sets up the necessary environment*. (2) The student actor works on the problem and writes code. (3) The student actor *commits code* to their repository to *make the submission*. (4) The GitHub API actor informs the CKB platform to pull the code.
Exit Condition	The CKB platform pulls the code from the team's repository that the student actor is in.
Exception(s)	• The GitHub API stops responding. • The student is not able to provide a solution.
Related Use Case(s)	• *Make Submission* • *Set Up Environment* • *Code Commit*

15. View Profile

Name	View Profile
Actor(s)	Authenticated User
Entry Condition	The actor is logged in.
Event Flow	(1) The actor clicks the Profile button from the sidebar to *view the profile*.
Exit Condition	The profile page is displayed.
Related Use Case(s)	• *View Profile*

16. See Own Tournaments

Name	See Own Tournaments
Actor(s)	Authenticated User
Entry Condition	The actor is on the profile page.
Event Flow	(1) The actor clicks the Tournaments icon to *to see own tournaments*.
Exit Condition	The tournaments that the actor engaged (either closed, ongoing, or upcoming) are displayed.
Related Use Case(s)	• *See Own Tournaments*
Note(s)	• Engage means registered for students, and created for educators.

17. See Own Battles

Name	See Own Battles
Actor(s)	Authenticated User
Entry Condition	The actor is on the profile page.
Event Flow	(1) The actor clicks the Battles icon to *to see own battles*.
Exit Condition	The battles that the actor engaged (either closed, ongoing, or upcoming) are displayed.
Related Use Case(s)	• *See Own Battles*
Note(s)	• Engage means registered for students, and created for educators.

18. Delete Profile

Name	Delete Profile
Actor(s)	Authenticated User
Entry Condition	The actor is on the profile page.
Event Flow	(1) The actor clicks the Delete Profile button.
Exit Condition	The actor is removed from the platform.
Exception(s)	• The actor is of type Educator and s/he has an ongoing tournament that s/he created.
Related Use Case(s)	• *Delete Profile*
Note(s)	• The educators who have an ongoing tournament created by themselves cannot delete their profile before they end the tournament.

19. Edit Settings

Name	Edit Settings
Actor(s)	Authenticated User
Entry Condition	The actor is logged in.
Event Flow	(1) The actor clicks the Settings button from the sidebar. (2) The actor changes the fields that s/he wants to change. The actor is able to *change name*, *change surname*, *edit institution info*, *change password*. (3) The actor clicks the Save button to *edit the settings*.
Exit Condition	The fields that the actor wanted to change are changed and the profile page is displayed.
Exception(s)	• The old password may be wrong during changing the password.
Related Use Case(s)	• *Edit Settings* • *Change Name* • *Change Surname* • *Change Password* • *Edit Institution Info*
Note(s)	• The actor provides the old password if they want to change their passwords for security measures.

20. Create Tournament

Name	Create Tournament
Actor(s)	Educator
Entry Condition	The actor is viewing the tournaments page.
Event Flow	(1) The actor clicks the Create Tournament button at the top of the page. (2) The actor sets the tournament attributes by *setting a title*, *setting a description*, and *setting a registration deadline* using the corresponding fields in the pop-up and then clicks the Next button. (3) The actor can *invite colleagues* to the tournament. (4) The actor finishes the *tournament creation* by clicking the Create button.
Exit Condition	The tournament is created and my tournaments page is displayed.
Related Use Case(s)	• *Create Tournament* • *Set Title* • *Set Description* • *Set Registration Deadline* • *Invite Colleagues*

21. Edit Tournament Attributes

Name	Edit Tournament Attributes
Actor(s)	Educator
Entry Condition	The actor is viewing a specific tournament's page that s/he has created.
Event Flow	(1) The actor clicks the Edit Tournament button. (2) The actor edits the tournament attributes by *setting a title*, *setting a description*, and *setting a registration deadline* using the corresponding fields in the pop-up. (3) The actor finishes *editing the tournament* by clicking the Done button.
Exit Condition	The tournament is edited and the page of that tournament is displayed.
Exception(s)	• The registration deadline may be already left behind. In this case, editing is not allowed. A tournament cannot be edited after it has started.
Related Use Case(s)	• *Edit Tournament Attributes* • *Set Title* • *Set Description* • *Set Registration Deadline*

22. End Tournament

Name	End Tournament
Actor(s)	Educator
Entry Condition	The actor is viewing a specific tournament's page that s/he has created.
Event Flow	(1) The actor clicks the End Tournament button to *end the tournament*.
Exit Condition	The tournament is ended and all battles inside that tournament have come to an end even if their submission deadline has not arrived yet.
Related Use Case(s)	• *End Tournament*

23. Respond to the Tournament Invitation

Name	Respond to the Tournament Invitation
Actor(s)	Invitee Educator
Entry Condition	The actor is logged in and invited to a tournament by another educator.
Event Flow	(1) The actor opens up the notifications and sees the invitation to join a tournament created by another educator. (2) The actor clicks the notification and a pop-up shows up. (3) The actor can choose to *Accept Invitation* by clicking the Accept button or *Reject Invitation* by clicking the Reject button to *respond to the tournament invitation*.
Exit Condition	• If accepted, the specific tournament is displayed. • If rejected, the dashboard is displayed.
Related Use Case(s)	• *Respond to Tournament Invitation* • *Accept Invitation* • *Reject Invitation*

24. Create Battle

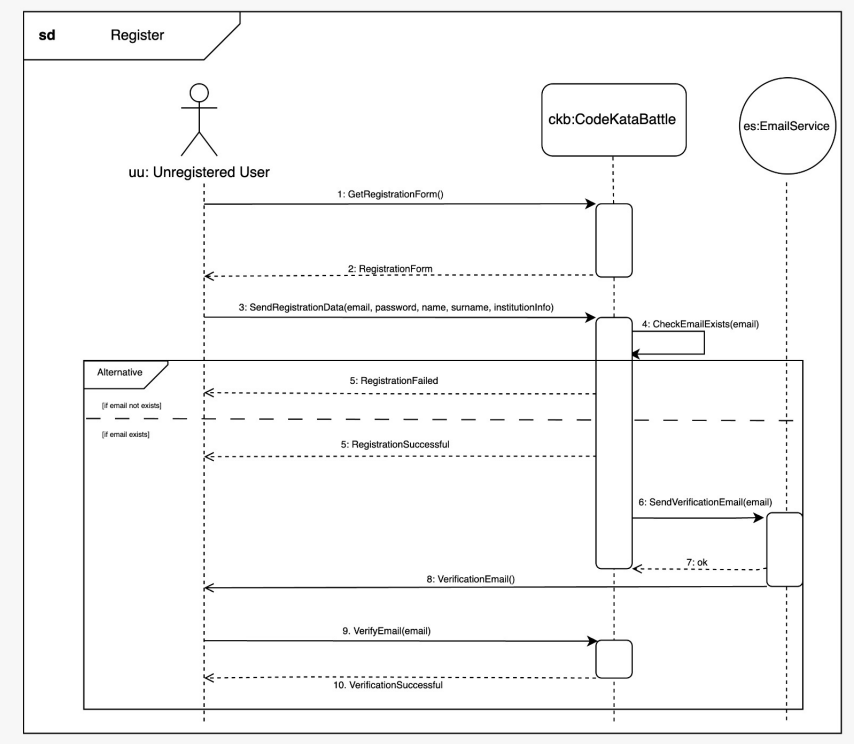
Name	Create Battle
Actor(s)	Educator
Entry Condition	The actor is viewing the page of a tournament that s/he has joined.
Event Flow	(1) The actor clicks the Create Battle button at the bottom right of the page. (2) The actor *sets the battle attributes* by *setting a title*, *setting a description*, *setting a registration deadline*, and *setting a submission deadline* using the corresponding fields in the pop-up and then clicks the Next button. (3) The actor continues to *set the battle attributes* by *setting the allowed languages*, and *uploading the test cases* using the corresponding fields in the pop-up and then clicks the Next button. (4) The actor continues to *set the battle attributes* by *uploading the build scripts*, and *setting the minimum and maximum group size* using the corresponding fields in the pop-up and then clicks the Next button. (5) The actor continues to *set the battle attributes* by *setting scoring criteria*. The actor *sets the percentages* of different scoring aspects, *enables or disables manual scoring*, and *selects the quality aspects* to be inspected by the Static Analysis Tool using the corresponding fields in the pop-up. (6) The actor finishes the *battle creation* by clicking the Create button.
Exit Condition	The battle is created and the page of that battle is displayed.
Exception(s)	• The actor may upload the wrong type of files during the test case or build script upload.
Related Use Case(s)	• *Create Battle* • *Set Battle Attributes* • *Set Title* • *Set Description* • *Set Registration Deadline - Set Submission Deadline* • *Set Minimum and Maximum Team Size* • *Set Allowed Languages* • *Upload Test Cases - Upload Build Scripts* • *Set Scoring Criteria*: *Set Percentages*, *Enable Manual Scoring*, *Select Quality Aspects*
Note(s)	• In the entry condition, joined means either created or invited & accepted by the actor.

25. Edit Battle Attributes

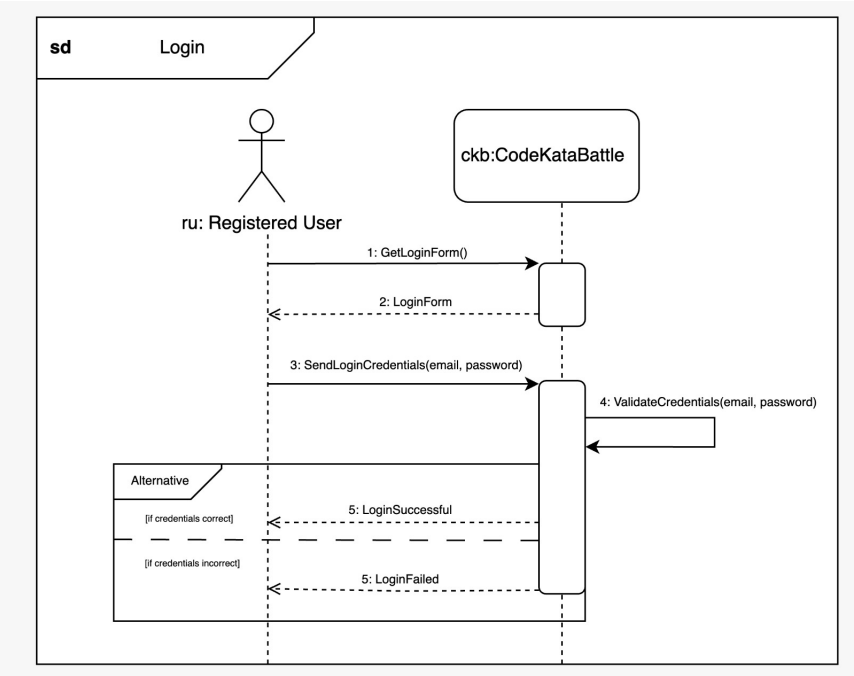
Name	Edit Battle Attributes
Actor(s)	Educator
Entry Condition	The actor is viewing a specific battle's page that s/he has created.
Event Flow	(1) The actor clicks the Edit Battle button. (2) The actor *sets the battle attributes* by *setting a title*, *setting a description*, and *setting a registration deadline* using the corresponding fields in the pop-up and then clicks the Next button. (3) The actor continues to *set the battle attributes* by *setting the allowed languages*, and *uploading the test cases* using the corresponding fields in the pop-up and then clicks the Next button. (4) The actor continues to *sets the battle attributes* by *uploading the build scripts*, and *setting the minimum and maximum group size* using the corresponding fields in the pop-up and then clicks the Next button. (5) The actor continues to *set the battle attributes* by *setting scoring criteria*. The actor *sets the percentages* of different scoring aspects, *enables or disables manual scoring*, and *selects the quality aspects* to be inspected by the Static Analysis Tool using the corresponding fields in the pop-up. (6) The actor finishes *editing the battle* by clicking the Done button.
Exit Condition	The battle is edited and the page of that battle is displayed.
Exception(s)	• The actor may upload the wrong type of files during the test case or build script upload.
Related Use Case(s)	• *Edit Battle Attributes* • *Set Battle Attributes* • *Set Title* • *Set Description* • *Set Registration Deadline - Set Submission Deadline* • *Set Minimum and Maximum Team Size* • *Set Allowed Languages* • *Upload Test Cases - Upload Build Scripts* • *Set Scoring Criteria*: *Set Percentages*, *Enable Manual Scoring*, *Select Quality Aspects*

3.2.3 Sequence Diagrams

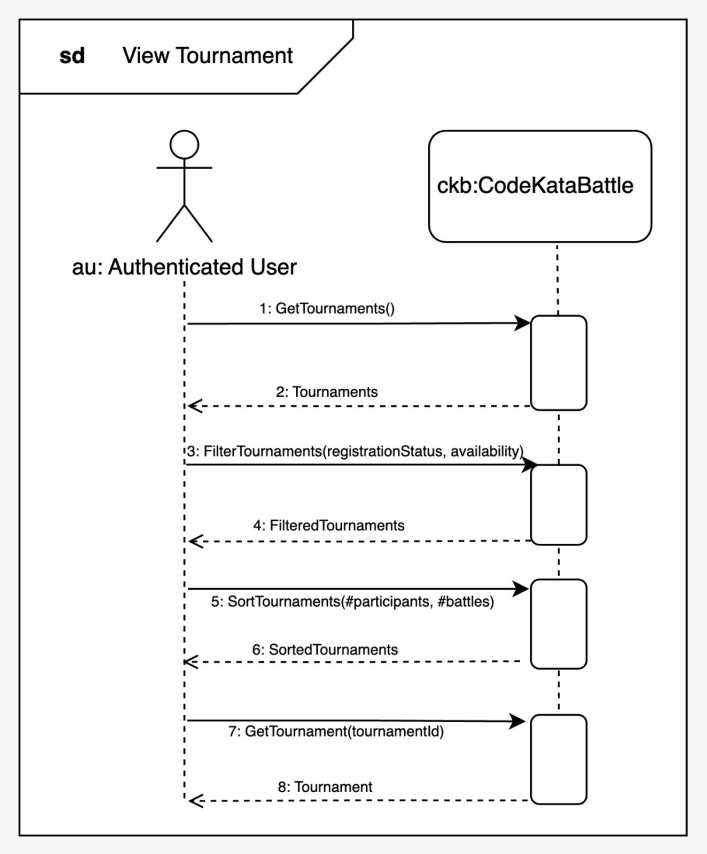
1. Register



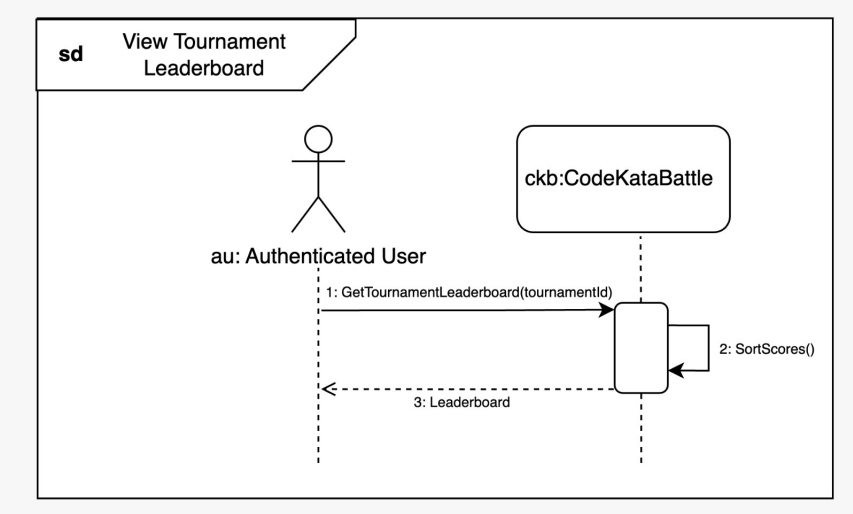
2. Login



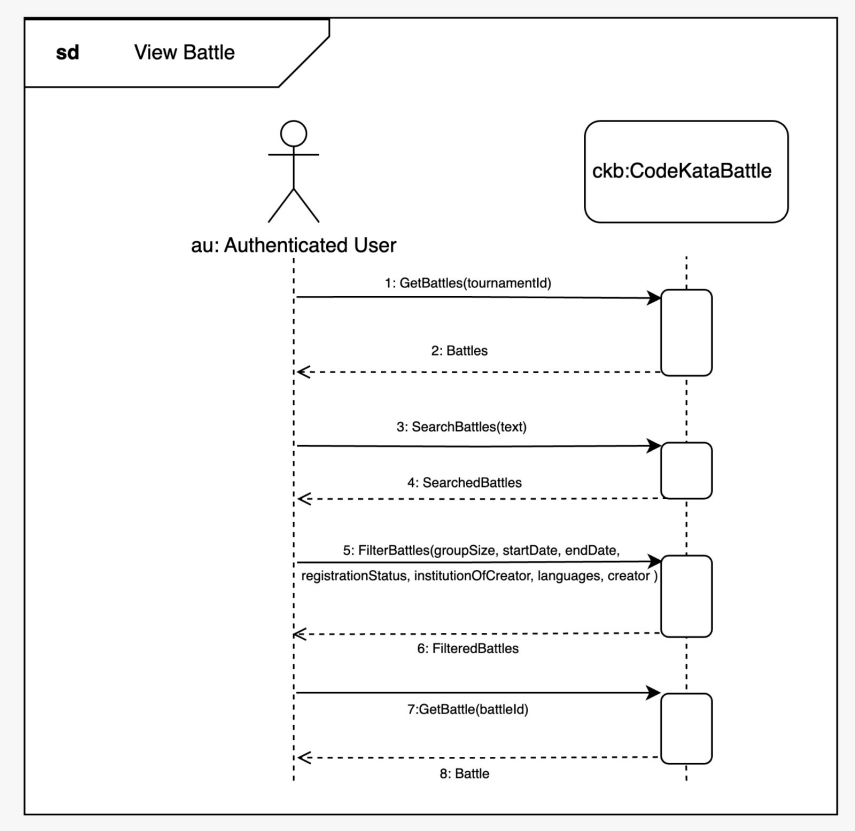
3. View a Tournament



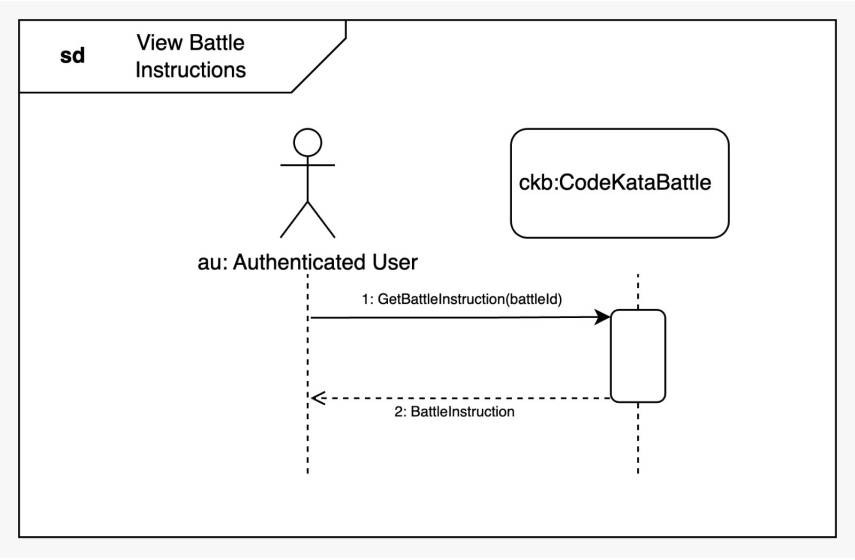
4. View the Tournament Leaderboard



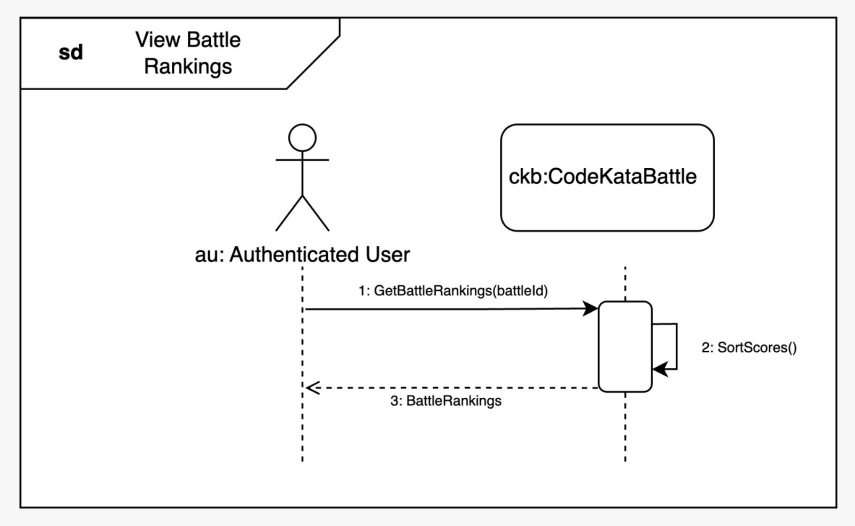
5. View a Battle



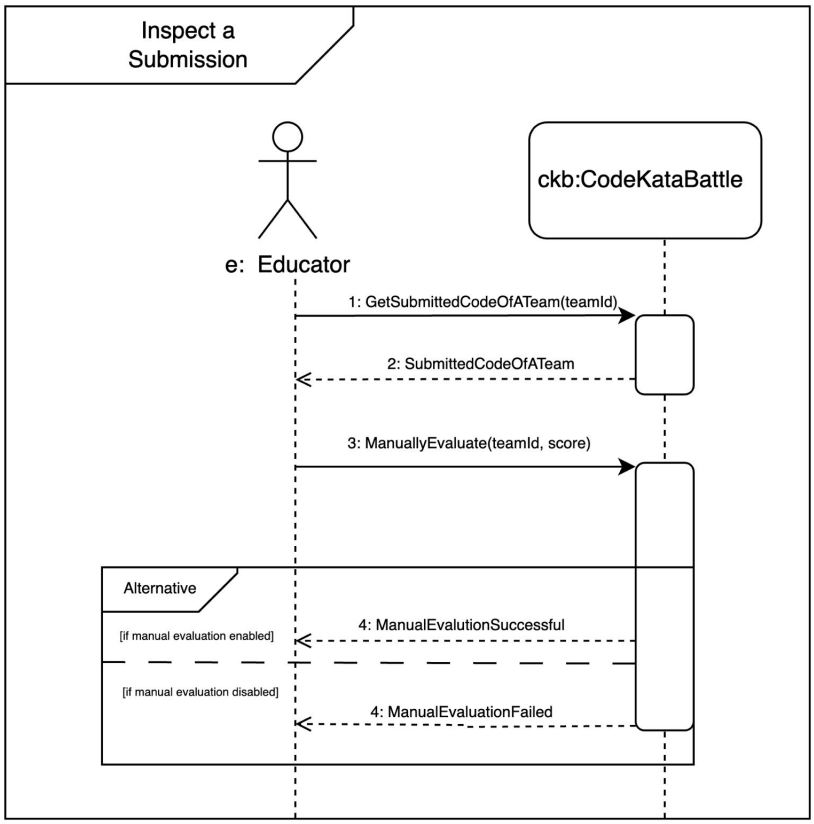
6. View the Battle Instructions



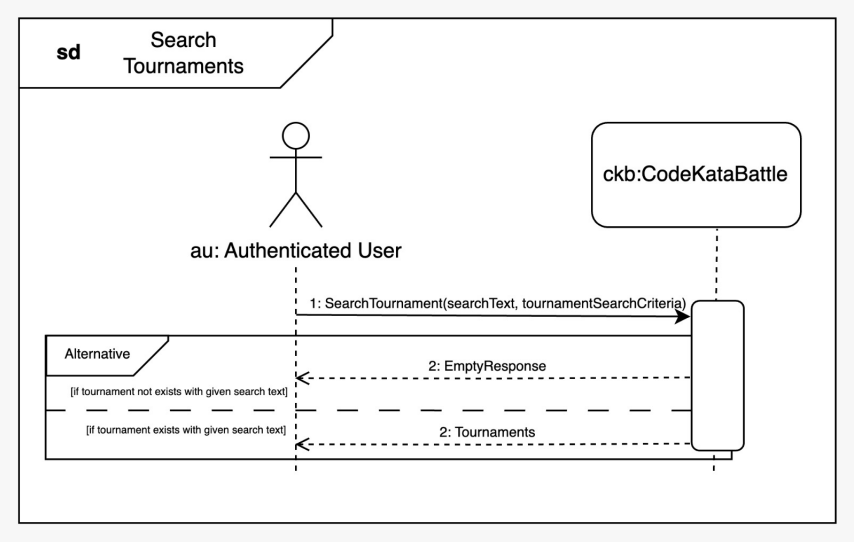
7. View the Battle Rankings



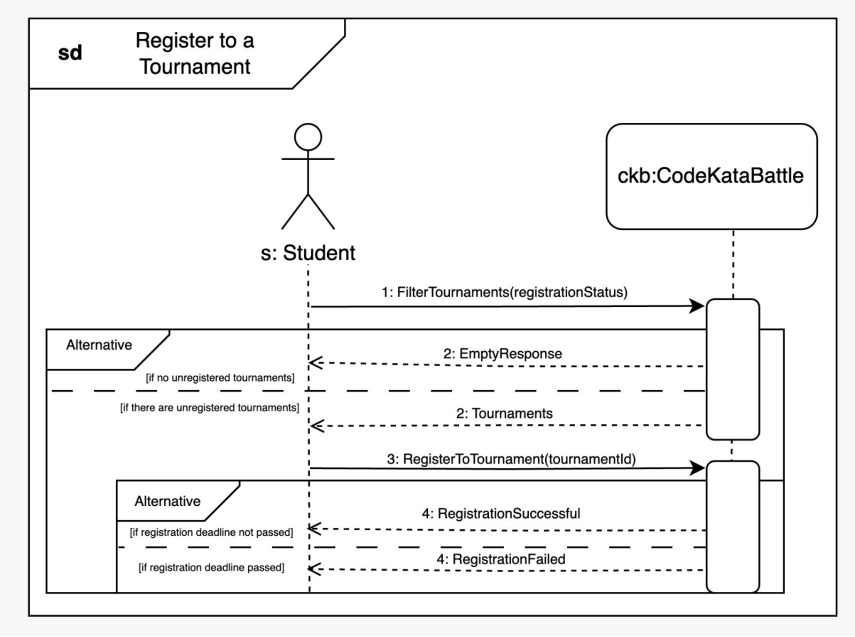
8. Inspect a Submission



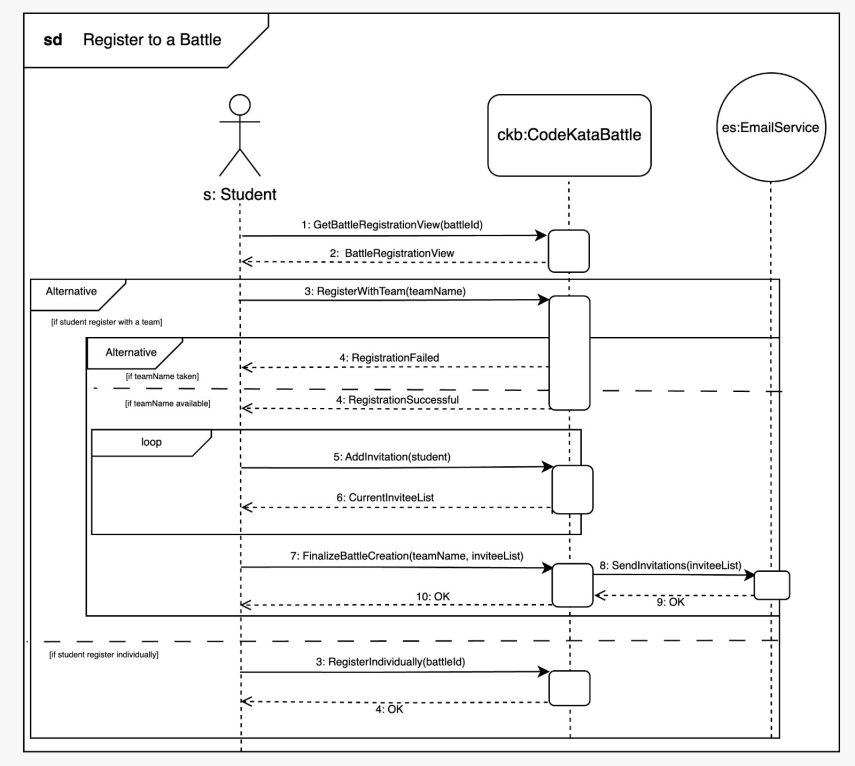
9. Search Tournaments



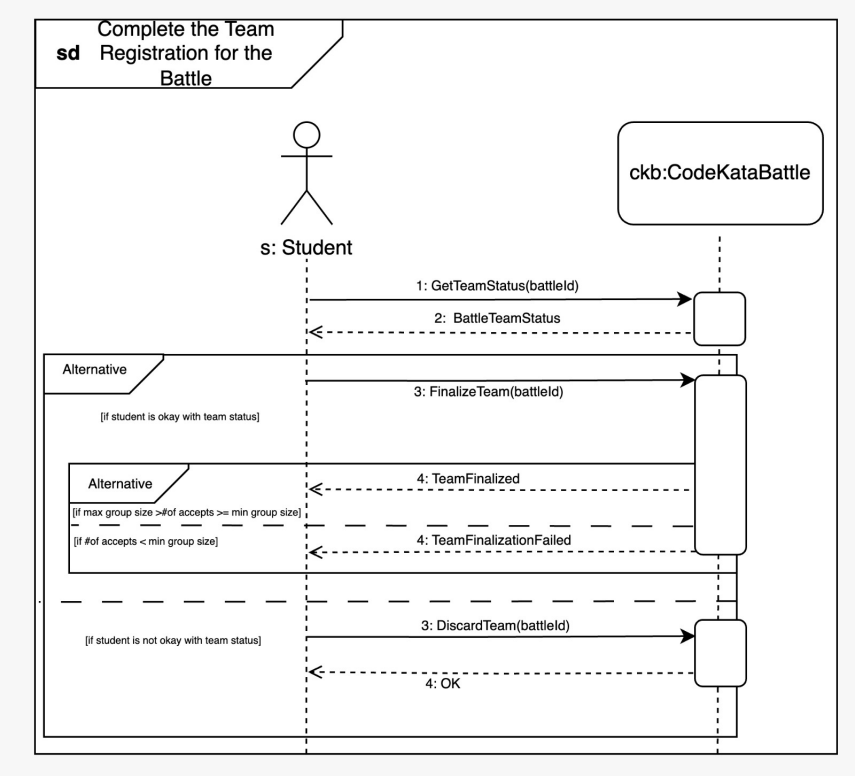
10. Register to a Tournament



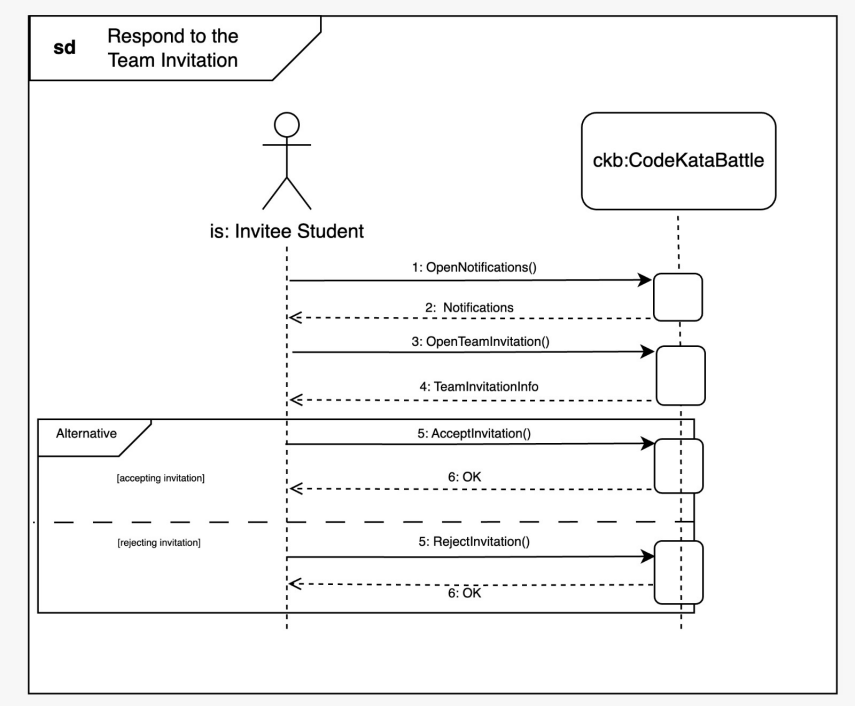
11. Register to a Battle



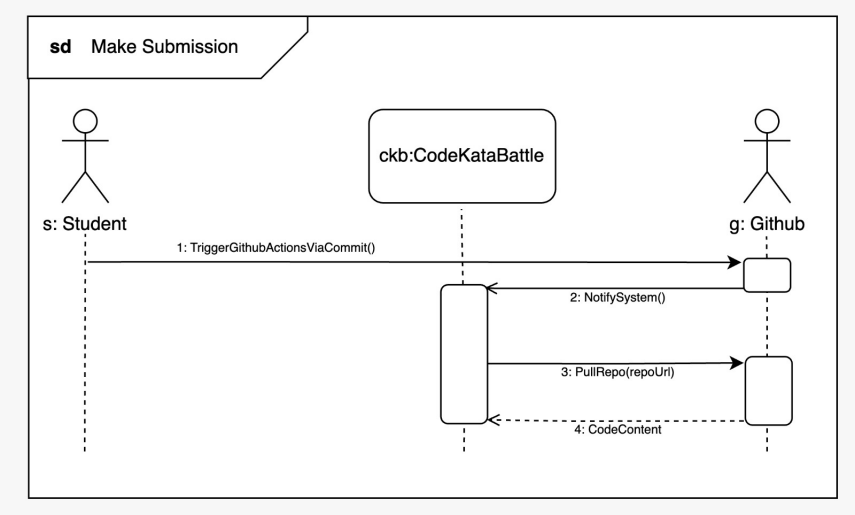
12. Complete the Team Registration for the Battle



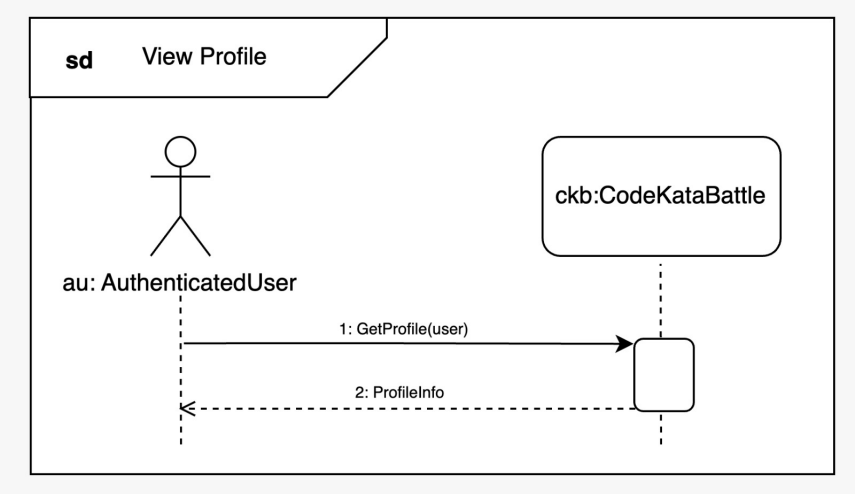
13. Respond to the Team Invitation



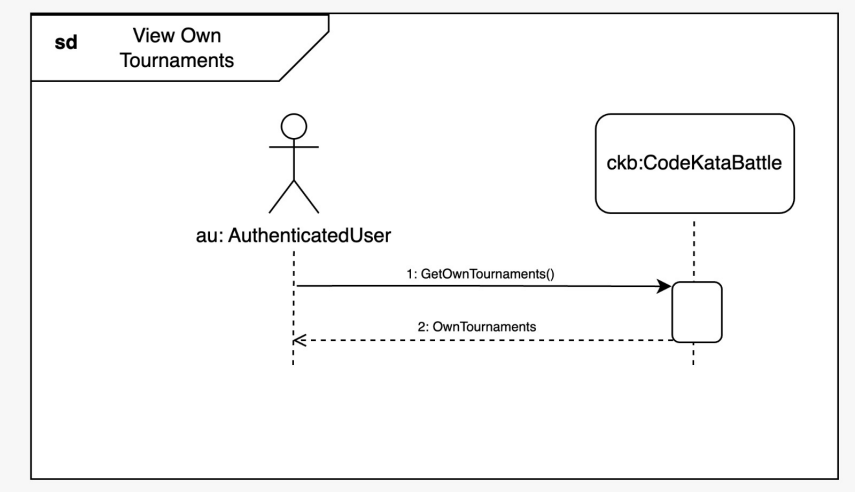
14. Make Submission



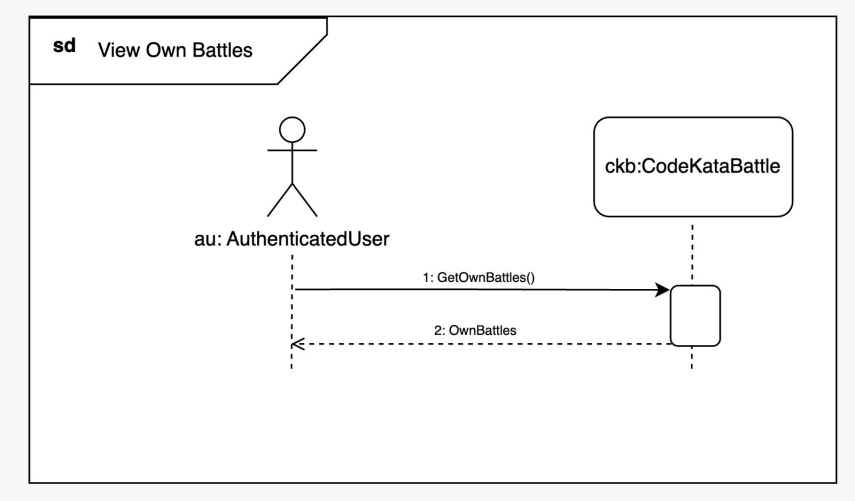
15. View Profile



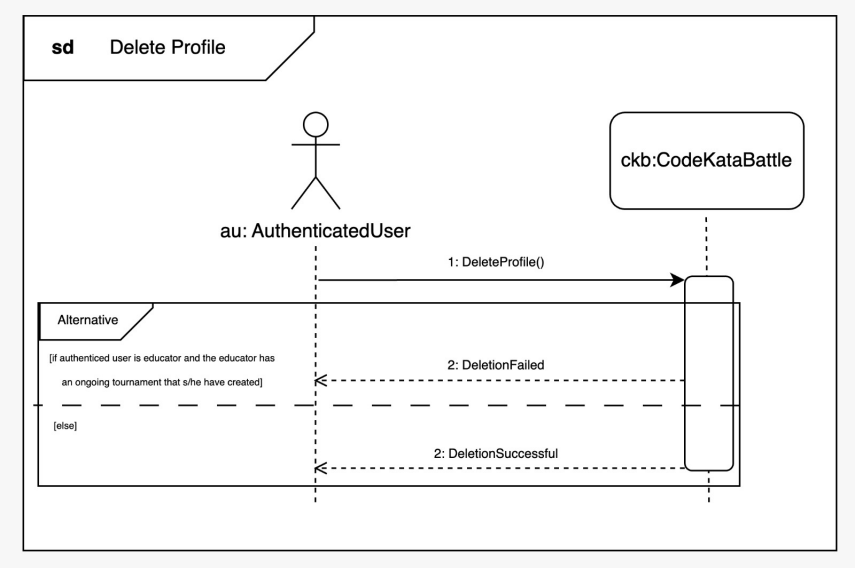
16. See Own Tournaments



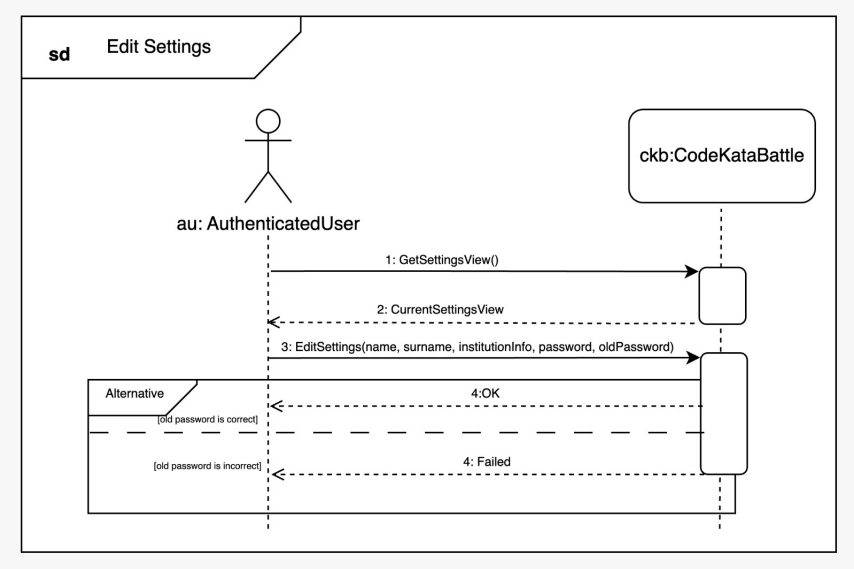
17. See Own Battles



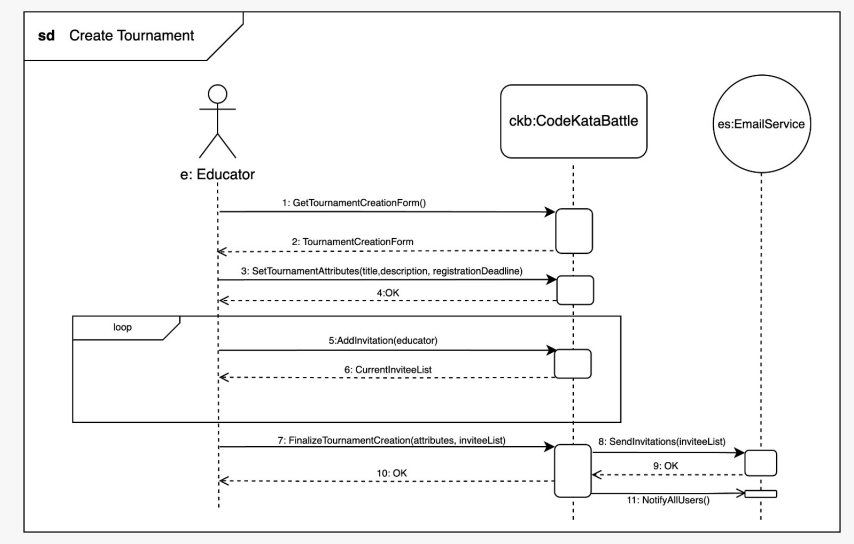
18. Delete Profile



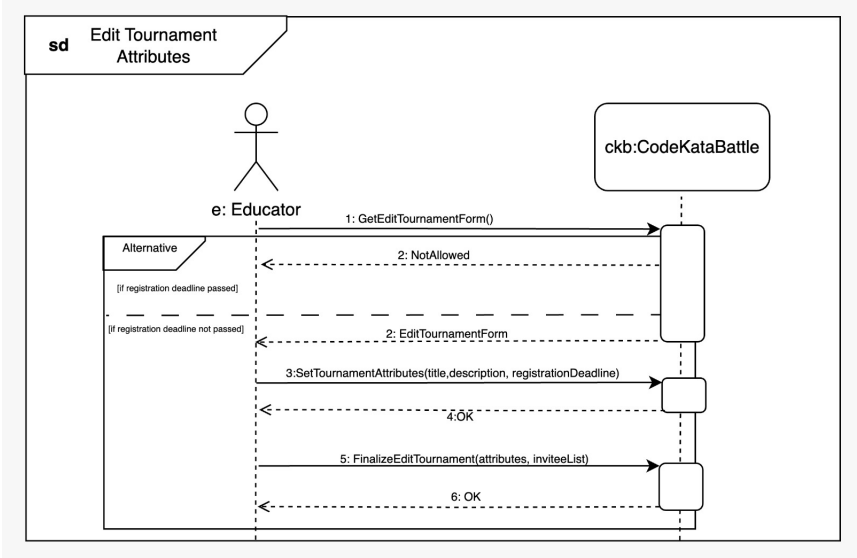
19. Edit Settings



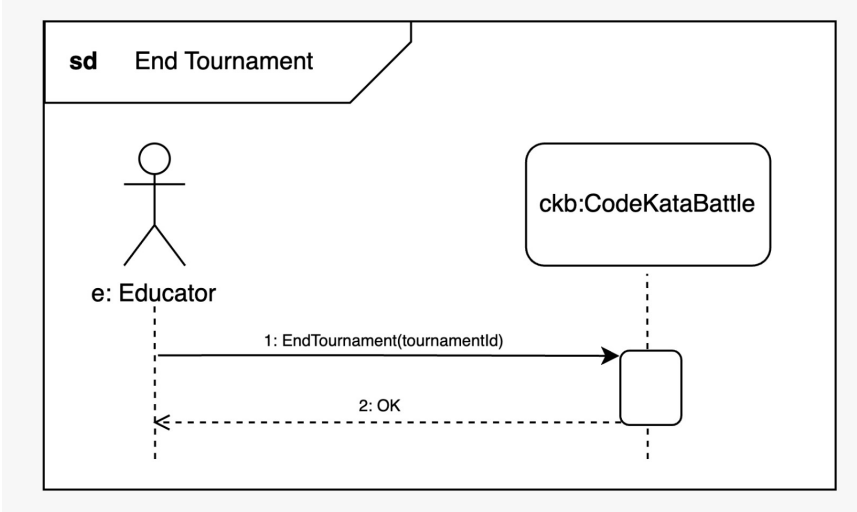
20. Create Tournament



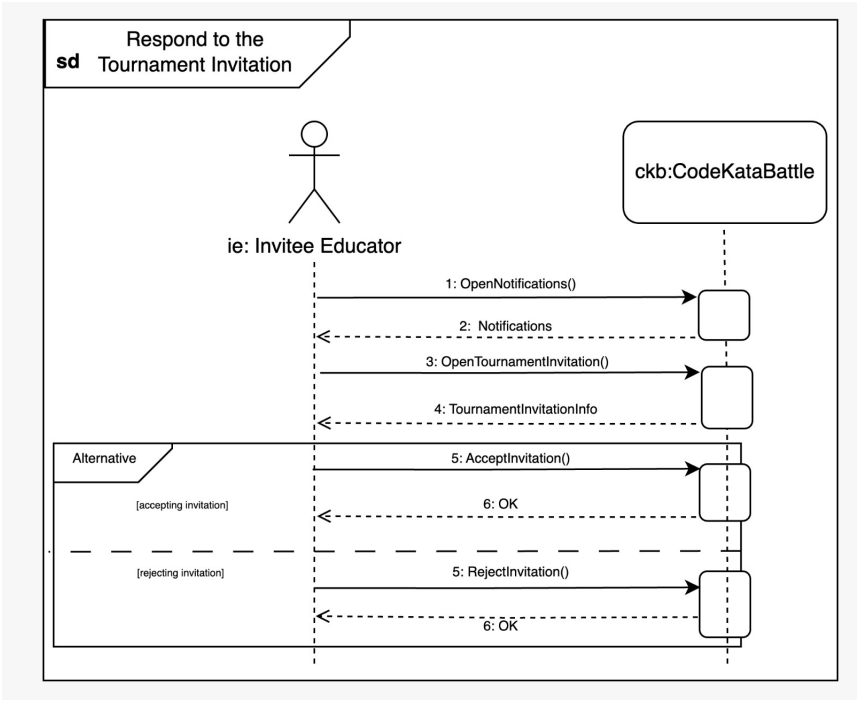
21. Edit Tournament Attributes



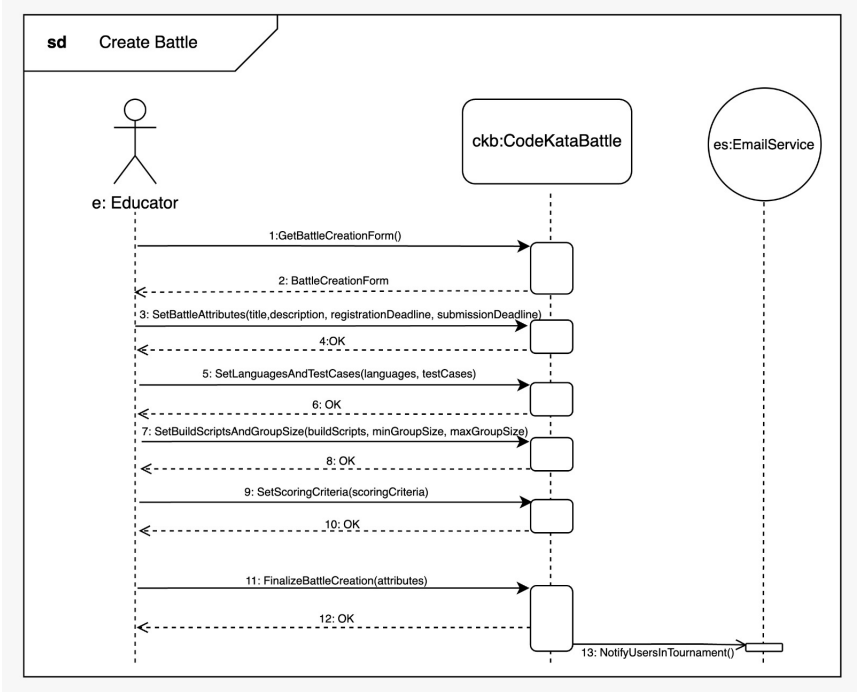
22. End Tournament



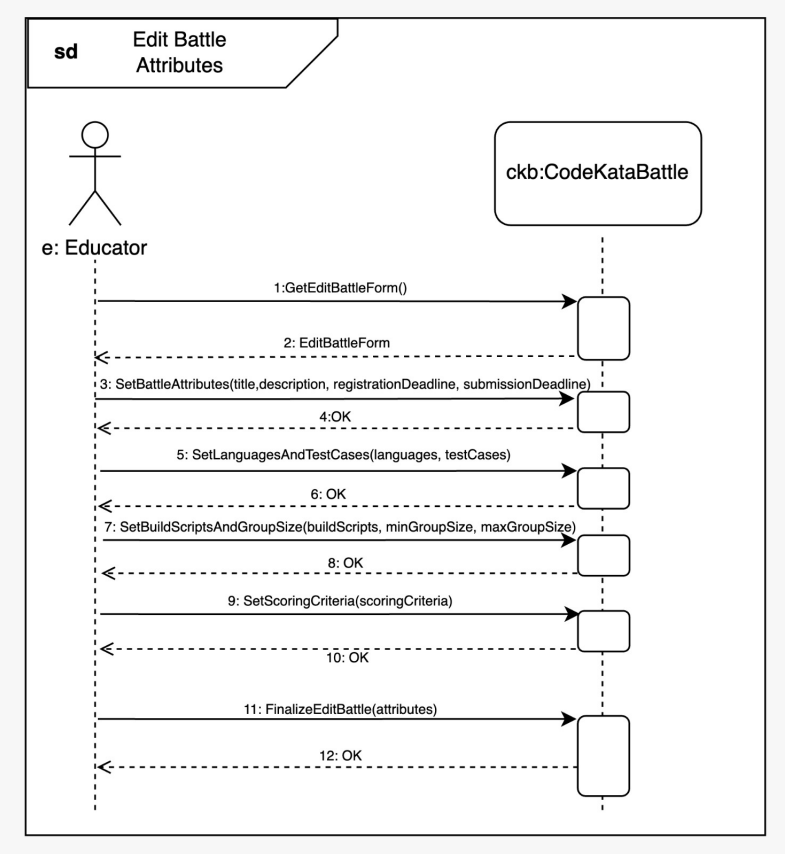
23. Respond to the Tournament Invitation



24. Create Battle



25. Edit Battle Attributes



3.2.4 Functional Requirements

In this section, we organized the Functional Requirements with respect to the user types. This section will consist of four parts. **Common** will include the requirements regarding both the educators and the students. **Educator** and **Student** will include the requirements related only to their respective parts. The last part, **The Platform** will include the requirements that are not directly related to any user type but still need to be executed by the system.

- Common

    1. The system shall allow unregistered users to register by providing an unique email, a password, a name, a surname, and institution information.
    2. The system shall allow unregistered users to select the user type during registration.
    3. The system shall allow users who received a verification email to verify their emails.
    4. The system shall allow registered users to log in by providing an email, and a password.
    5. The system shall allow authenticated users to view all tournaments.
    6. The system shall allow authenticated users to filter the tournaments by the registration status and availability.
    7. The system shall allow authenticated users to sort the tournaments by number of participants and number of battles in it.
    8. The system shall allow authenticated users to view a specific tournament.
    9. The system shall allow authenticated users to view the leaderboard of the tournament.
    10. The system shall allow authenticated users to view all battles in a tournament.
    11. The system shall allow authenticated users to filter the battles by group size, start date - end date, registration status, institution of the battle creator, allowed programming languages, and the battle creator.
    12. The system shall allow authenticated users to search battles by text search.
    13. The system shall allow authenticated users to view a specific battle and its instructions.
    14. The system shall allow authenticated users to view the rankings of the battle.
    15. The system shall allow authenticated users to search tournaments with text search by educators, by titles, or by institutions.
    16. The system shall allow authenticated users to view their profiles.
    17. The system shall allow authenticated users to delete their profiles unless they do not have an ongoing tournament created by themselves.
    18. The system shall allow authenticated users to view their own tournaments.
    19. The system shall allow authenticated users to view their own battles.
    20. The system shall allow authenticated users to view their settings.
    21. The system shall allow authenticated users to edit their settings by name, surname, password, and institution information.
    22. The system shall oblige authenticated users to enter their old password during settings editing.

- Educator

    23. The system shall allow educators to create tournaments by providing a title, a description, and a registration deadline.

24. The system shall allow educators to invite other educators to their tournaments during tournament creation.
25. The system shall allow educators to edit the tournaments that they have created by providing a title, a description, and a registration deadline unless the registration deadline has not passed.
26. The system shall allow educators to end the tournaments that they have created.
27. The system shall allow educators to accept or reject the tournament invitation coming from other educators for a tournament.
28. The system shall allow educators to create battles for tournaments, that they have either created or joined by invitation.
29. The system shall allow educators to create battles by providing a title, a description, a registration deadline, a submission deadline, the allowed languages, test cases, build scripts, minimum & maximum group size, and the scoring criteria.
30. The system shall oblige educators to upload test case file and build script for every allowed language in battle.
31. The system shall allow educators to manually evaluate the submissions giving extra point between 0 and 10 after the submission deadline has passed.
32. The system shall allow educators to edit the battles that they have created.

- Student

33. The system shall allow students to register for tournaments.
34. The system shall allow students to register for battles in which the tournaments that they have registered for.
35. The system shall allow students to register for battles individually.
36. The system shall allow students to register for battles by a team and set a team name.
37. The system shall allow students to invite other students to their team during battle registration.
38. The system shall allow students to accept or reject the team invitation coming from other students for a battle.
39. The system shall allow students to finalize their team registration or decline it.
40. The system shall allow students to

- GitHub

41. The system shall create a repository for a battle after the registration deadline for that battle has passed.
42. The system shall pull the repository of a team following a trigger from GitHub Actions.

- Email Service

43. The system shall trigger the email service to send a verification email to the recently registered users.
44. The system shall trigger the email service to send a notification email for a newly created tournament for all registered users.
45. The system shall trigger the email service to send an invitation email for the battle to the invitee students.
46. The system shall trigger the email service to send a notification email including the link to the battle repository to the students registered for it.

47. The system shall trigger the email service to send an invitation email for the tournament to the invitee educators.

- The Platform

48. The system shall automatically evaluate submissions by scoring criteria.
    (a) The system shall score the submission with respect to test cases, and test case weight.
    (b) The system shall score the submission with respect to timeliness, and timeliness weight.
    (c) The system shall score the submission with respect to quality aspects, and quality aspect weight.
49. The system shall utilise a Static Analysis Tool to calculate the score in terms of quality aspects.
50. The system shall create a sandbox environment for each team for the submissions in order to run the codes.
51. The system shall automatically update the battle score of a team after the evaluation of the submission.
52. The system shall automatically update the battle rankings when a score is updated.
53. The system shall automatically update the tournament leaderboard at the end of each battle.

The following subsections in Section 3: Specific Requirements will include different types of **Non-Functional Requirements**. Even though they are separated into different subsections, the enumeration of elements will resume between subsections as they all represent Non-Functional Requirements.

3.3 Performance Requirements

1. The system shall respond to each user within 3 seconds for at least 90% of the requests.
2. The system shall support up to 1000 users concurrently.
3. The system shall work 24/7 with no more than 2% downtime, excluding scheduled maintenance.
4. The system shall process the codes pushed to the GitHub repositories within 60 seconds unless the GitHub API is down.

3.4 Design Constraints

3.4.1 Standards Compliance

5. The system shall comply with the General Data Protection Regulation (GDPR).
6. The system shall comply with ISO/IEC 27001:2022, managing the information security.
7. The system shall use JSON for data interchange within the APIs.

3.4.2 Hardware Limitations

8. The system requires a web-supporting device.

3.4.3 Other Constraints

9. The system user interface language shall be English.
10. The system shall be implemented in Python & FastAPI at the backend, and Javascript & React at the frontend.

3.5 Software System Attributes

3.5.1 Reliability

11.

3.5.2 Availability

12.

3.5.3 Security

13.

3.5.4 Maintainability

14.

3.5.5 Portability

15.

4 Formal Analysis Using Alloy

```
module CodeKataBattle


//These can be Enum
abstract sig Language {}
one sig PYTHON extends Language {}
one sig JAVA extends Language {}
one sig C extends Language {}
one sig CPP extends Language {}
one sig JAVASCRIPT extends Language {}
one sig RUBY extends Language {}
one sig GO extends Language {}
one sig RUST extends Language {}
one sig CSHARP extends Language {}
abstract sig QualityAspect {}
one sig COMPLEXITY, DUPLICATIONS, MAINTAINABILITY, RELIABILITY, SECURITY, CLEAN_CODE
    ↪ extends QualityAspect {}

//These can be Enum
abstract sig  TournamentStatus {}
one sig REG_OPEN, ONGOING, CLOSED extends TournamentStatus{}
//These can be Enum
abstract sig BattleStatus {}
one sig REG_OPEN_BATTLE, ONGOING_BATTLE, CONSOLIDATION, CLOSED_BATTLE extends
    ↪ BattleStatus{}

// Basic types
abstract sig Bool {}
one sig TRUE extends Bool {}
one sig FALSE extends Bool {}

sig Time{
time: one Int
} {time>0}

//These fields are removed from User for the sake of simplicity because they are not
    ↪ related to anything but user specific attributes.  They can be unique or same, no
    ↪ check in the system related to this.
//sig Password{}
//sig Name{}
//sig Surname{}

abstract sig Notification{}
sig TournamentCreated extends Notification{
  tournamentCreated: one Tournament
}
sig TournamentEnded extends Notification{
tournamentEnded: one Tournament
}
sig TournamentInvited extends Notification{
tournamentInvited: one Tournament
}
sig BattleStarted extends Notification{
battleStarted: one Battle
}
sig BattleFinished extends Notification{
battleFinished: one Battle
}

sig Institution{}
sig TournamentTitle{}
sig TournamentDescription{}

sig TeamName{}

sig URL{}

sig Email{}
sig User {
```

```
    email: one Email,
    notifications: set Notification
}

sig Educator extends User {
    var creates: set Tournament,
    var contributesTo: set Tournament,
    var createsBattle: set Battle,
    institutions: set Institution,
    var givenManualScores: set ManualScore
}

sig Student extends User {
    var registers: set Tournament,
    var tournamentScores: Tournament -> lone Int,
    institution: lone Institution,
}

sig Tournament {
    title: one TournamentTitle,
    description: one TournamentDescription,
    registrationDeadline: one Time,
    closingTime: lone Time,
    var status: one TournamentStatus,
    var hasBattle: set Battle,
    var subscribers: set Student,
} {
     registrationDeadline.time < closingTime.time

}

sig ScoringWeights{
    TEST_CASES:  Int,
    TIMELINESS:  Int,
    QUALITY:  Int
    } {
     TEST_CASES > 0 ∧ TIMELINESS > 0 ∧ QUALITY>0 ∧ plus[plus[TEST_CASES,TIMELINESS],
         ↪ QUALITY]=6
    }

sig Team {
    teamName: one TeamName,
    hasMember: some Student,
    submits: lone Submission,
}

sig BattleTitle {}
sig Battle {
    title: one BattleTitle,
    codeKata: one CodeKata,
    minStudentsPerGroup: one Int,
    maxStudentsPerGroup: one Int,
    creationTime: one Time,
    registrationDeadline: one Time,
    submissionDeadline: one Time,
    var status: BattleStatus,
    allowedLanguages: some Language,
    chosenQualityAspects: some QualityAspect,
    var teams: set Team,
    scoringWeights: one ScoringWeights,
    manualScoringEnabled: one Bool,
    battleRepositoryUrl: lone URL,
    sandboxEnvironment: SandboxEnvironment,
} {
  registrationDeadline.time < submissionDeadline.time ∧
  minStudentsPerGroup≤maxStudentsPerGroup ∧
  minStudentsPerGroup≥0 ∧
  maxStudentsPerGroup≥0
 }

sig Code{}
sig Submission{
```



```
code: one Code,
repositoryUrl: one URL,
commitTime: one Time,
score: one BattleScore,
sandboxEnvironment: SandboxEnvironment
}
sig ManualScore{
    score: one Int
} {score≥0}
sig BattleScore {
    testCasesScore: one Int,
    timelinessScore: one Int,
    qualityScore: one Int,
    manualScore: lone ManualScore,
    totalScore: one Int
} {
   totalScore = calculateBattleScore[this]
   testCasesScore ≥0
   timelinessScore ≥0
   qualityScore ≥ 0
}

fun calculateBattleScore(bs:BattleScore): Int {
    bs.testCasesScore + bs.timelinessScore + bs.qualityScore
}

sig BattleDescription{}
sig CodeKata {
    description: one BattleDescription,
    testCases: some TestCase,
    buildScripts: some BuildScript
}

sig TestCase {
language: one Language
}

sig BuildScript {
language: one Language
}


//Each  battle has a sandbox environment that runs the code
sig SandboxEnvironment{}

// Relationships and constraints


//Every user has unique email
fact EmailsAreUnique{
no disjoint u1, u2: User | u1.email = u2.email
}

//Every user is eiher Educator or Student
fact AllUsersAreEitherEducatorOrStudent {
    all u: User |
        (u in Educator and u not in Student) or (u in Student and u not in Educator)
}

// Each team must be associated with a battle and have members within the specified group
    ↪  size constraints.
fact TeamsWithinGroupSizeConstraints {
    all t: Team |some b: Battle |       t in b.teams    and     #t.hasMember ≥ b.
        ↪ minStudentsPerGroup and #t.hasMember ≤ b.maxStudentsPerGroup
}

//In a battle there must be test cases and build scripts for each allowed language
fact ArtifactsProvided {
    all b: Battle | {
        all l: b.allowedLanguages | {
            one tc: TestCase | tc.language=l    and     tc in b.codeKata.testCases
            one bs: BuildScript| bs.language=l and bs in b.codeKata.buildScripts
```

```
        }
    }
}

//no other than allowed languages
fact TestCasesAndBuildScriptsForAllowedLanguagesOnly {
    all b: Battle |
            b.codeKata.testCases.language in b.allowedLanguages and b.codeKata.
                ↪ buildScripts.language in b.allowedLanguages
}



// Each Test Case and Build Script Belongs To One Code Kata
fact ArtifactsBelongsToOneCodeKata {
    // For every TestCase, there is exactly one CodeKata that it's related to
    all tc: TestCase | one ck: CodeKata | tc in ck.testCases

    // For every BuildScript, there is exactly one CodeKata that it's related to
    all bs: BuildScript | one ck: CodeKata | bs in ck.buildScripts
}

// Every tournament is created by exactly one educator
fact UniqueEducatorForTournament {
    all t: Tournament | one e: Educator | e.creates = t
}

fact UniqueBattleForCodeKata {
    // Every CodeKata is associated with exactly one Battle
    all ck: CodeKata | one b: Battle | ck = b.codeKata
}


// Every tournament is contributed to by at least one educator which is the creator of
    ↪ the tournament
fact CreatorContributesTournament {
    all t: Tournament | one e: Educator | t = e.creates and t in e.contributesTo
}

// Every battle is created by exactly one educator
fact UniqueCreatorForBattle {
    all b: Battle | one e: Educator | e.createsBattle = b
}

// Every battle is associated with exactly one tournament
fact UniqueTournamentForBattle {
    all b: Battle | one t: Tournament | b in t.hasBattle and
        all tOther: Tournament - t | b not in tOther.hasBattle
}

//for consitency
fact SubscribersAreRegistered {
    all t: Tournament, s: Student |  t in s.registers implies s in t.subscribers
}



// Every educator can create battles only for tournaments they contribute to
fact ContributerCreatesBattle{
    all e: Educator, b: Battle | b in e.createsBattle implies b in e.contributesTo.
        ↪ hasBattle
}

// Unique battle titles within a tournament
fact UniqueBattleTitle{
    all t: Tournament | no disjoint b1, b2: t.hasBattle | b1.title = b2.title
}

// Unique tournament titles
fact UniqueTournamentTitles {
    no disjoint t1, t2: Tournament | t1.title = t2.title
}
```

```
// Unique team names within a battle
fact UniqueTeamNamesWithinBattle {
    all b: Battle | no disjoint t1, t2: b.teams | t1.teamName = t2.teamName
   all tn:TeamName      |       one t: Team     |       t.teamName=tn
}

// Ensure submission constraints
fact EveryManualScoreHasAScoreAndSubmission {
    all ms: ManualScore |
        one bs: BattleScore | bs.manualScore = ms
}

fact EveryManualScoreHasAScoreAndSubmission {
    all bs: BattleScore |
        one s: Submission | bs in s.score
}

fact EveryManualScoreHasAScoreAndSubmission {
    all s: Submission |
        one t: Team | s in t.submits
}

fact EveryManualScoreHasAScoreAndSubmission {
    all t: Team |
        one b: Battle | t in b.teams
}


fact EveryCodeHasASubmission {
    // For every Code, there exists a Submission that has this score
    all cd: Code | one s: Submission | s.code = cd
}

// A student's submission must be their own work and associated with a team that is part
    ↪ of a battle. A domain assumption
fact UniqueSubmissionRepositoryUrlWithinBattle {
    all url: URL | one s: Submission | s.repositoryUrl = url
}

// A student can only be part of at most one team per battle
fact StudentInAtMostOneTeamPerBattle {
    all b: Battle | no disj t1, t2: b.teams | some s: Student | s in t1.hasMember and s
        ↪ in t2.hasMember
}


// Students can only be part of teams for battles in tournaments they have registered for
    ↪ .
fact StudentTeamRegistrations {
    all s: Student | all t: s.registers | let battles = t.hasBattle |
    all team: Team | s in team.hasMember implies team in battles.teams
}


// There is no created battle if tournament status is REG_OPEN
fact noBattlesInRegistration {
    all t: Tournament | t.status = REG_OPEN implies no t.hasBattle
}

// There is no submission  if battle status is REG_OPEN_BATTLE
fact noSubmissionsAndRepoForOpenBattles {
    all b: Battle | b.status = REG_OPEN_BATTLE implies {
        no t: b.teams | some t.submits and
        no b.battleRepositoryUrl
    }
}


fact AllBattlesClosedIfTournamentClosed {
    all t: Tournament | t.status = CLOSED implies all b: t.hasBattle | b.status =
        ↪ CLOSED_BATTLE
```



```
}


// Battle registration time is less than Tournament closing Time. It is check for closed
    ↪ tournaments. A closed tournament can not have a battle with creation time later
    ↪ than closing time
fact BattlesWithinTournamentClosingTime {
    all t: Tournament, b: t.hasBattle | one t.closingTime ∧ b.creationTime.time < t.
        ↪ closingTime.time
}

fact ManualScoreConstraint {
    all bs: BattleScore |
       one  bs.manualScore  implies bs.manualScore.score ≥ 0 ∧ bs.manualScore.score ≤ 2
}


//Scores should be in range 0 and their weights
fact ScoreWithinWeightRange {
    all t: Team | some t.submits implies {
        let s = t.submits.score |
            // Find the battle that includes this team
            some b: Battle | t in b.teams and {
                let weights = b.scoringWeights |
                s.testCasesScore ≥ 0 and s.testCasesScore ≤ weights.TEST_CASES and
                s.timelinessScore ≥ 0 and s.timelinessScore ≤ weights.TIMELINESS and
                s.qualityScore ≥ 0 and s.qualityScore ≤ weights.QUALITY
            }
    }
}

fact enforceManualScoringRules {
    all b: Battle | {
        // If the battle is in consolidation and manual scoring is enabled,
        // there must be a manual score for each battle score
        (b.status = CLOSED_BATTLE and b.manualScoringEnabled = TRUE) implies {
            all s: b.teams.submits.score | some ms: ManualScore | ms in s.manualScore
        }

        // If manual scoring is not enabled, the status cannot be CONSOLIDATION
        (b.manualScoringEnabled = FALSE) implies b.status ≠ CONSOLIDATION

        // If the battle is in before consolidation or manual scoring is not enabled,
        // there cannot be a manual score for any battle score
        (b.status = REG_OPEN_BATTLE  or b.status = ONGOING_BATTLE or b.
            ↪ manualScoringEnabled = FALSE) implies {
            all s: b.teams.submits.score | no s.manualScore
        }
    }
}


//This fact enforces the rule that if a ManualScore exists for a BattleScore in a Battle,
//it must have been given by the Educator who created that Battle.
//If there is no ManualScore, this fact does not impose any additional requirements.
// In other words, it only applies when a ManualScore is present.
fact manualScoresGivenByCreator {
    all b: Battle, s: b.teams.submits.score, ms: s.manualScore | {
        some e: Educator | e.createsBattle = b and ms in e.givenManualScores
    }
}

//if a tournament is created (exists), it should be in Student's notification with as
    ↪ TournamentCreated notification, for every tournament exist there must be a
    ↪ notification about it in their notifications field.
fact notificationForTournamentCreation {
    all t: Tournament, s: Student |
        one n: s.notifications | n in TournamentCreated and n.tournamentCreated = t
}

//if tournament is ended (status is CLOSED), it should be in Student's notification with
    ↪ as TournamentEnded notification, for every tournament ended there must be a
```



```
    ↪ notification about it in their notifications field.
fact notificationForTournamentEnding {
    all t: Tournament | t.status = CLOSED implies
        all s: Student | s in t.subscribers implies
            one n: s.notifications | n in TournamentEnded and n.tournamentEnded = t
}

fact TournamentEndedNotificationMeansTournamentEnded {
    all n: TournamentEnded | n.tournamentEnded.status = CLOSED
}


// if an educator is contributes to a tournament but not a creator, there must be a
    ↪ notification in its notifications with a TournamentInvited notification.
fact notificationForEducatorInvitation {
    all e: Educator, t: Tournament | t in e.contributesTo  and t not in  e.creates
        ↪ implies
        one n: e.notifications | n in TournamentInvited and n.tournamentInvited = t
}

//if a battle is not in REG_OPEN_STAGE, there must be a BattleStarted Notification of the
    ↪  notification field of students enrolled in that battle. For every battle started
    ↪  there must be one .
fact notificationForBattleStarting {
    all b: Battle, s:Student |   b.status ≠ REG_OPEN_BATTLE and s in b.teams.hasMember
        ↪ implies
            some n: s.notifications | n in BattleStarted and n.battleStarted = b
}


//if a battle is CLOSED_BATTLE stage, there should be a BattleFinished notification in
    ↪ that users notifications, if he or she enrolled in that battle.
fact notificationForBattleFinishing {
    all b: Battle,s: Student | b.status = CLOSED_BATTLE and s in b.teams.hasMember
        ↪ implies
            one n: s.notifications | n in BattleFinished and n.battleFinished = b
}

fact BattleFinishedNotificationMeansBattleIsClosed {
    all n: BattleFinished, s:Student | n.battleFinished.status = CLOSED_BATTLE and n in s
        ↪ .notifications
}


//forgetten fact above, noticed when writing tournament score facts
fact UserMemberOfTeamMustBeEnrolledInTournament {
    all s: Student, t: Team | s in t.hasMember implies
        some b: Battle | t in b.teams and b in s.registers.hasBattle
}


//TournamanetScores are summation of the scores of student's teams in battles
fact StudentTournamentScores {
    all s: Student |
                    s.tournamentScores = s.registers -> sumOfTeamTotalScores[{team: Team
                        ↪ | some b: {b: s.registers.hasBattle | b.status = CLOSED_BATTLE
                        ↪ }| team in b.teams and s in team.hasMember and team.submits ≠
                        ↪ none}]
}
fact StudentTournamentScoresMap {
    all s: Student |
        s.tournamentScores in s.registers -> Int
}

fun sumOfTeamTotalScores[teams: set Team]: Int {
    sum team: teams | some team.submits.score implies team.submits.score.totalScore else
        ↪ 0
}

fact AllNotificationsLinkedToUsers {
    all n: Notification | some u: User | n in u.notifications
}
```



```
fact AlwaysRegistered {
    all t: Tournament, s: Student | always ((s in t.subscribers implies t in s.registers)
        ↪  and ( t in s.registers implies s in t.subscribers))
}

fact EventuallyClosed {
    all t: Tournament, b: t.hasBattle | b.status = ONGOING_BATTLE implies eventually b.
        ↪ status = CLOSED_BATTLE
}

fact CorrectBattleStatusTransition {
    all b: Battle | historically (b.status = REG_OPEN_BATTLE implies (b.status' =
        ↪ ONGOING_BATTLE or b.status' = REG_OPEN_BATTLE)) and
                    (b.status = ONGOING_BATTLE implies (b.status' = CLOSED_BATTLE or b.
                        ↪ status' = CONSOLIDATION or b.status' = ONGOING_BATTLE) ) and
                        (b.status = CONSOLIDATION implies (b.status' = CONSOLIDATION or b
                            ↪ .status'=CLOSED_BATTLE))
}

fact NotificationAfterBattleClosure {
    all b: Battle, s: Student |b.status = CLOSED_BATTLE implies
                                always some n: s.notifications | n in BattleFinished
                                    ↪ and n.battleFinished = b
}

fact NotificationIfNotClosed {
    all b: Battle, s: Student |b.status ≠ CLOSED_BATTLE implies
                                always no n: s.notifications | n in BattleFinished and
                                    ↪ n.battleFinished = b
}


fact ScoreCalculationAfterClosure {
    all b: Battle | b.status = ONGOING_BATTLE and b.status' = CLOSED_BATTLE implies
                                after all s: b.teams.submits.score | s.totalScore =
                                    ↪ calculateBattleScore[s]
}


fact {
    always {
        all t: Tournament | t.status = CLOSED implies t.status' = CLOSED
        all b: Battle | b.status = CLOSED_BATTLE implies b.status' = CLOSED_BATTLE
    }
}



//total battle score is sum of partial scores
fact CalculateBattleScore {
    all bs: BattleScore |
        let result = 0 |
        addManualScoreIfExist[bs, result]
}

pred addManualScoreIfExist(bs: BattleScore, result: Int) {
    some bs.manualScore implies
        result = plus[bs.totalScore,bs.manualScore.score]
    else
        result = bs.totalScore
}


// a student's point from tournament is the sum of points from submissions of team he is
    ↪ part of submitted to battles finished in tournament
// The system shall update the tournament leaderboard at the end of each battle.


pred EducatorCreatesTournament[e: Educator, t: Tournament] {
```

```
    t not in Tournament
    // 'b' is a newly created battle
    t not in e.creates and
    // In the next state, 'b' is in the set of battles created by 'e'
    t' in e'.creates
}


pred studentRegistersForTournament[s: Student, t:Tournament] {
    s not in t.subscribers and
    s.registers' = s.registers + t and
    t.subscribers' = t.subscribers + s
}



pred EducatorContributesToTournament[e: Educator,t: Tournament ] {
    // Ensuring the new tournament is not already created by this educator
    t not in e.contributesTo and
    e'.contributesTo = e.creates + t
}

pred studentJoinsTeam[t: Team,s: Student,] {
    s not in t.hasMember and
    t'.hasMember = t.hasMember + s
}

pred createTeam[b: Battle, newTeam: Team] {
     newTeam not in b.teams
     b'.teams = b.teams + newTeam
}

pred educatorCreatesBattle[e: Educator, t: Tournament, b: Battle] {
    b not in Battle and
    e'.createsBattle = e.createsBattle + b and
    t'.hasBattle = t.hasBattle + b
}

pred setBattleParameters[e: Educator, b: Battle, minSize: Int, maxSize: Int, regDeadline:
    ↪  Time, subDeadline: Time, msEnabled:Bool] {
    b in e.createsBattle and
    b'.minStudentsPerGroup = minSize and
    b'.maxStudentsPerGroup = maxSize and
    b'.registrationDeadline = regDeadline and
    b'.submissionDeadline = subDeadline and
    b'.manualScoringEnabled = msEnabled
}

pred studentSubmitsCode[s: Student, t: Team, sub: Submission] {
    s in t.hasMember and
    t'.submits = sub'
}


pred educatorScoresSubmission[t: Team, manual: ManualScore] {
    t.submits.score.manualScore' = manual
}


pred educatorEditsTournament[e: Educator, t: Tournament, newTitle: TournamentTitle,
    ↪ newDesc: TournamentDescription, newRegDeadline: Time] {
    some ed: Educator|ed.creates = t and
    e in ed and
    t.title = newTitle and
    t.description = newDesc and
    t.registrationDeadline = newRegDeadline
}


// Assertions

// Predicate to encourage diversity in tournament attributes
```



```
pred diverseTournamentAttributes {
    all disj t1, t2: Tournament | {
        t1.description ≠ t2.description or
        t1.registrationDeadline ≠ t2.registrationDeadline or
        (some t1.closingTime and some t2.closingTime implies t1.closingTime ≠ t2.
            ↪ closingTime)
        // Add similar conditions for other attributes if necessary
    }
}

// Predicate to encourage diversity in battle attributes
pred diverseBattleAttributes {
    all disj b1, b2: Battle | {
        b1.title ≠ b2.title or
        b1.registrationDeadline ≠ b2.registrationDeadline or
        b1.submissionDeadline ≠ b2.submissionDeadline or
        b1.manualScoringEnabled ≠ b2.manualScoringEnabled or
        b1.battleRepositoryUrl ≠ b2.battleRepositoryUrl or
        b1.allowedLanguages ≠ b2.allowedLanguages or
        b1.chosenQualityAspects ≠ b2.chosenQualityAspects
        // Add similar conditions for other attributes if necessary
    }
}

// Assert that students can commit code and trigger automated testing
assert studentsCommitAndTriggerTesting {
    all s: Student, t: s.registers.hasBattle.teams, sub: t.submits |
        sub.repositoryUrl ≠ none and sub.commitTime ≠ none
    // This assertion checks that students in teams have submissions with repository URLs
        ↪  and commit times
}


check studentsCommitAndTriggerTesting for 5


//no battle created when tournament open, not ongoing
assert NoBattlesCreatedWhenTournamentOpen {
    all t: Tournament | t.status = REG_OPEN implies no t.hasBattle
}

check NoBattlesCreatedWhenTournamentOpen for 2



//every submission in  closed battle has manual score if manual scoring enabled
assert ManualScoreForClosedBattles {
    all b: Battle | b.status = CLOSED_BATTLE and b.manualScoringEnabled = TRUE  implies {
        all t: b.teams | some t.submits implies {
            all s: t.submits | some s.score.manualScore
        }
    }
}



check ManualScoreForClosedBattles for 5

assert EducatorCantCraeteBattleInUnrelatedTournament {
    all e3: Educator, t: Tournament, b: Battle | b in e3.createsBattle  and
        (t not in e3.contributesTo and t not in e3.creates) implies  b not in t.hasBattle
}


check EducatorCantCraeteBattleInUnrelatedTournament for 6



assert BattleEndedNotificationImpliesStudentEnrolled {
    all n: Notification, s: Student | n in s.notifications and n in BattleFinished
        ↪ implies
        all t: Team, b:Battle | t in b.teams and s in t.hasMember and b = n.
```



```
            ↪ battleFinished implies
            (some t.submits implies
                (some sc: t.submits.score | sc.totalScore ≠ none) and
                (n.battleFinished.manualScoringEnabled = TRUE implies some  t.submits.
                    ↪ score.manualScore))
}


check BattleEndedNotificationImpliesStudentEnrolled for 5



assert NoPositiveTournamentScoreForRegOpenTournaments {
    all t: Tournament | t.status = REG_OPEN implies
        all s: Student | s in t.subscribers implies
            s.tournamentScores[t] ≤ 0
}

check NoPositiveTournamentScoreForRegOpenTournaments for 5


assert CorrectBattleScoreCalculation {
    all b: Battle, t: b.teams, s: t.submits.score | {
        s.totalScore = calculateBattleScore[s] +
            (b.manualScoringEnabled = TRUE implies s.manualScore.score else 0)
    }
}


check CorrectBattleScoreCalculation for 5



//Total score for tournament
assert CorrectTotalScoreCalculation {
    all to: Tournament, s: Student, b:Battle, t:Team, n:Notification| s in to.subscribers
        ↪  and b in to.hasBattle and b.status=CLOSED_BATTLE  and  t = b.teams and s in t
        ↪ .hasMember and n in BattleFinished and n in s.notifications and n.
        ↪ battleFinished = b implies
        s.tournamentScores[to] = sum t.submits.score.totalScore
}

check CorrectTotalScoreCalculation for 5


//validity of programming languages
assert ConsistencyOfProgrammingLanguagesInBattles {
    all b: Battle, lang: Language | lang in b.allowedLanguages implies
        (lang in b.codeKata.testCases.language and lang in b.codeKata.buildScripts.
            ↪ language)
}

check ConsistencyOfProgrammingLanguagesInBattles for 5



//validity of battle teams and students
assert BattleParticipationConstraints {
    all b: Battle, s: Student,t:Tournament | s in b.teams.hasMember and  b = t.hasBattle
        ↪ implies s in t.subscribers
}

check BattleParticipationConstraints for 5

assert CorrectTournamentRegistration {
    all s: Student, t: Tournament |
        (studentRegistersForTournament[s, t] implies
         s' in t.subscribers')
}

check CorrectTournamentRegistration for 5
```

```
assert ValidBattleCreationByEducator {
    all e: Educator, b: Battle, t:Tournament |
        (educatorCreatesBattle[e, t, b] implies
         (b' in e.creates.hasBattle or b' in e.contributesTo.hasBattle))
}

check ValidBattleCreationByEducator for 5

assert TournamentCreatedEventuallyNotification {
    all e: Educator, t: Tournament |
        EducatorCreatesTournament[e, t] implies
        eventually (all s: Student | s in t.subscribers implies
                    some n: s.notifications | n.tournamentCreated = t' and n in
                        ↪ TournamentCreated)
}

check TournamentCreatedEventuallyNotification for 5

assert ConsistentTeamMembership {
    all s: Student, t: Team |
        (studentJoinsTeam[t, s] implies
         always s' in t.hasMember)
}

check ConsistentTeamMembership for 5


assert UserWithScoreJoinedTeamAndSubmitted {
    all s: Student, t: Tournament |
        (s.tournamentScores[t] > 0) implies
        some b: t.hasBattle, team: b.teams |
            (s in team.hasMember and some team.submits)
}

check UserWithScoreJoinedTeamAndSubmitted for 5


assert TournamentStatusValidity {
    all t: Tournament | {
        // If the tournament is REG_OPEN, it has always been REG_OPEN historically
        (t.status = REG_OPEN) implies historically (t.status = REG_OPEN) and

        // If the tournament is ONGOING, it was once REG_OPEN
        (t.status = ONGOING) implies once (t.status = REG_OPEN) and

        // If the tournament is CLOSED, its past states are REG_OPEN and ONGOING
            ↪ respectively
        (t.status = CLOSED) implies (once (t.status = REG_OPEN) and once (t.status =
            ↪ ONGOING))
    }
}

check TournamentStatusValidity for 5


assert BattleStatusValidity {
    all b: Battle | {
        // If the battle is REG_OPEN_BATTLE, it has always been REG_OPEN_BATTLE
            ↪ historically
        (b.status = REG_OPEN_BATTLE) implies historically (b.status = REG_OPEN_BATTLE)
            ↪ and

        // If the battle is ONGOING_BATTLE, it was once REG_OPEN_BATTLE
        (b.status = ONGOING_BATTLE) implies once (b.status = REG_OPEN_BATTLE) and

        // If the battle is in CONSOLIDATION, it was once ONGOING_BATTLE
        (b.status = CONSOLIDATION) implies (once (b.status = ONGOING_BATTLE) and (b.
            ↪ status' ≠ ONGOING_BATTLE or b.status'≠REG_OPEN_BATTLE)) and

        // If the battle is CLOSED_BATTLE, its past states are REG_OPEN_BATTLE,
```



```
            ↪ ONGOING_BATTLE, and possibly CONSOLIDATION
        (b.status = CLOSED_BATTLE) implies (once (b.status = REG_OPEN_BATTLE) and once (b
            ↪ .status = ONGOING_BATTLE) and once (b.status = CONSOLIDATION) and always (
            ↪ b.status' = CLOSED_BATTLE))
    }
}

check BattleStatusValidity for 5

pred Progress[e:Educator, b:Battle, sub:Submission, s:Student, t:Team, ms:ManualScore] {
        // Initial State: Battle is ongoing, manual scoring enabled, but no submission
            ↪ made by the team
        (b.status = ONGOING_BATTLE and b.manualScoringEnabled = TRUE and
            ↪ studentSubmitsCode[s, t, sub]) eventually {

                // After submission, the system eventually goes into the consolidation
                    ↪ stage
                 (b.status' = CONSOLIDATION) after {
                        (educatorScoresSubmission[t, ms]) eventually {

                        // Eventually, the battle goes into the finished stage
                        (b.status'' = CLOSED_BATTLE) after {

                            // Ensure there is a manual score for every team in the
                                ↪ battle
                            all team: b.teams | some team.submits.score.manualScore
                        }
                    }
                }
            }

}


run Progress for 5 but 1 Team


assert TournamentScoreUpdatesAfterBattleClosure {
    all e: Educator, b: Battle, s: Student, t: Team, sub: Submission, ms: ManualScore,
        ↪ tournament: Tournament |
        Progress[e, b, sub, s, t, ms] and b in tournament.hasBattle and t in b.teams and
            ↪ s in t.hasMember implies
        eventually (b.status' = CLOSED_BATTLE) implies
        always (s.tournamentScores'[tournament] = (s.tournamentScores[tournament] + sum t
            ↪ .submits'.score.totalScore))
}

check TournamentScoreUpdatesAfterBattleClosure for 5


assert ManualScoringDuringConsolidation {
    all e: Educator, b: Battle, s: Student, t: Team, sub: Submission, ms: ManualScore |
        Progress[e, b, sub, s, t, ms] implies eventually (b.status = CONSOLIDATION)
            ↪ implies always (some sub.score.manualScore)
}

check ManualScoringDuringConsolidation for 5


assert CorrectBattleStatusSequence {
    all e: Educator, b: Battle, s: Student, t: Team, sub: Submission, ms: ManualScore |
        Progress[e, b, sub, s, t, ms] implies
        (historically (b.status = ONGOING_BATTLE) and eventually (b.status =
            ↪ CONSOLIDATION) and eventually (b.status = CLOSED_BATTLE))
}

check CorrectBattleStatusSequence for 5



pred World {
all b:Battle    |       #b.allowedLanguages =2
```

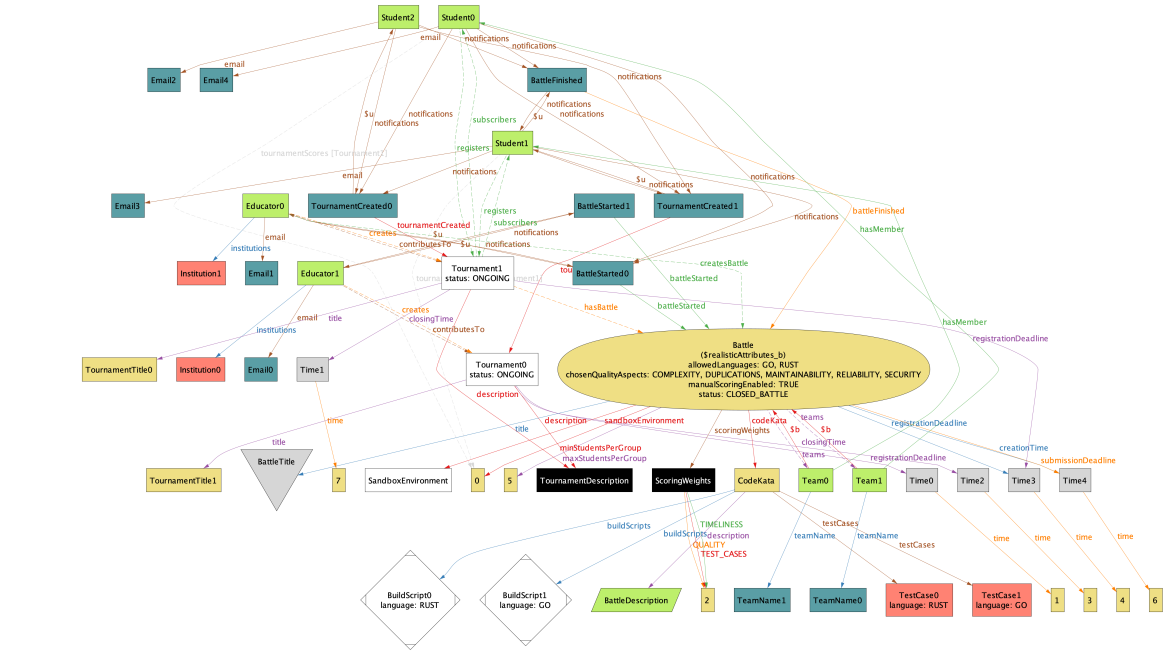


```
#Team=2
#Tournament=2
#Educator=2
#Student=3
some b: Battle  |       b.manualScoringEnabled=TRUE
}
run { World } for 5


pred World2 {
#Team>3
#Tournament=1
#Submission=2
some b: Battle  |       b.manualScoringEnabled=TRUE
}
run { World2 } for 5


pred show{ #Battle>0 #Team>0}
run show for 5
```

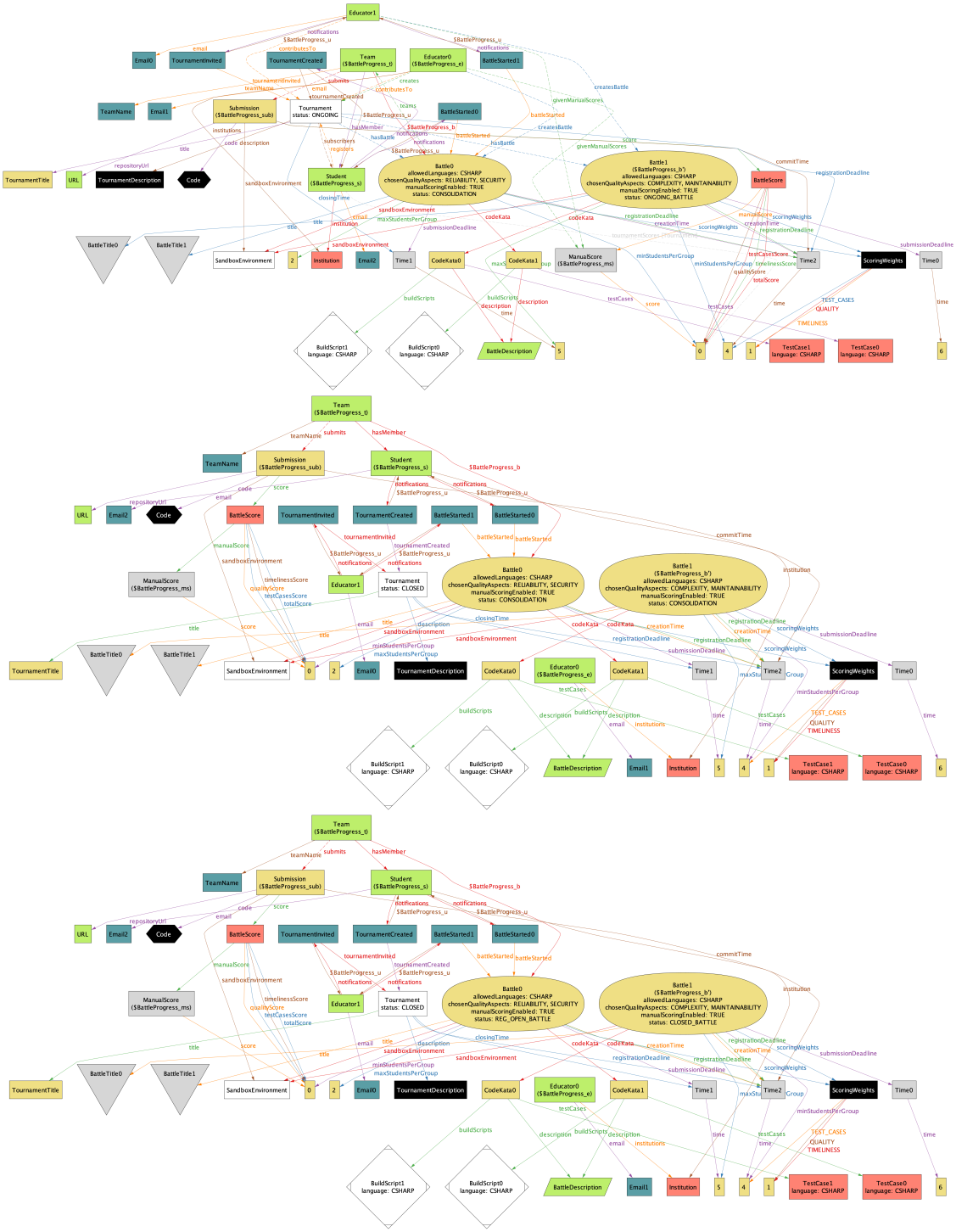
4.0.1 World Modelling

In the Alloy, in order to see the relation between signatures and general view of model, we generally use the see predicates with some constraints such as number of tournaments is greater than 0 etc. Also it really helps us to create lots of assertion because, initially, there are lots of mistakes in the world modelling and it help us to improve system with an iterative approach. Of course there can be more complex or more simple world models. The model attached below is one of the simple world modellings of our system. With the code above there can create more complex models. Also, below a battle progress is modeled with the help of the dynamic modelling in Alloy.


```
Executing "Run run$1 for 5"
  Solver=sat4j Steps=1..10 Bitwidth=4 MaxSeq=5 SkolemDepth=1 Symmetry=20 Mode=batch
  1..1 steps. 62072 vars. 2717 primary vars. 167116 clauses. 628ms.
  Instance found. Predicate is consistent. 179ms.
```



Dynamic Modelling



5 Effort Spent

Provide here information about how much effort each group member spent in working at this document. We would appreciate details here.

References