Project: The Calculator Mod

Project Report

Professor: Dr. Mike Mireku Kwakye

CSCI441: Software Engineering

Sept 13. 2025

Project website: https://spaces.w3schools.com/space/the-calculator-mod/editor

Team D Group Members:

Alexander Redinger

Thol Ucca Kool

Oziel Martinez

Uong SovanDara

Individual Contributions Breakdown

Every member in this group has contributed equally to the project. This is a breakdown of contributions in a table:

Sections	Uong Sovandara	Oziel Martinez	Thol Ucca Kool	Alexander Redinger
Cover Page				
Individual Contribution Breakdown				
Table of Content				
Work Assignment				
1a. Problem Statement				
1b. Decomposition into Sub-Problems				
1c. Glossary of Terms				
2a. Business Goals				
2b. Enumerated Functional Requirements				
2c. Enumerated Nonfunctional Requirements				
2d. User interface Requirements				
3a. Stakeholders				
3b. Actors and Goals				
3c. Use Cases				
3d. System Sequence Diagrams				
4a. Preliminary Design				
4b. User Effort Estimation				
5a. Identifying Subsystems				
5b. Architecture Styles				
5c. Mapping Subsystems to Hardware				

5d. Connectors and Network Protocols				
5e. Global Control Flow				
5f. Hardware Requirements				
6a. Conceptual Model				
6b. System Operation Contracts				
6c. Data Model and Persistent Data Storage				
6d. Mathematical Model				
7. Interaction Diagram				
8a. Class Diagram				
8b. Data Type and Operation Signature				
8c. Traceability Matrix				
9. Algorithms and Data Structures				
10. User Interface Design and Implementation				
11. Test Designs				
Project Management				
Reference				

Table of Contents

Section **Page #**

Work Assignment

Uong SovanDara and Thol Ucca Kool have been assigned to mainly do most of the frontend on the website in W3School while Oziel Martinez is designing the UI and UX of the website. Alexander Redinger is linking the excel sheet calculation with the website while also refining the modding process of the master mod itself.

Individual Student Competence

Thol Ucca Kool: Strong team player and problem solver with practical software development and system analysis skills. adept at generating ideas, debugging, and confirming functionality in practical tasks. The main programming languages include HTML/CSS, Java, JavaScript, Python, C++, and Java. knowledgeable about database administration, Prolog, Oracle APEX, and assembly (MASM). knowledgeable about software engineering projects, web and app development, and internship verification procedures.

Oziel Martinez: Skill set includes Python for scripting and automation, SQL with hands-on experience in CRUD operations, and Java for object-oriented programming projects. Proficient in Java, C++, Python, JavaScript, HTML, and Node.js for backend development, including building and consuming APIs. Skilled in database integration and backend system design, with a foundation in web technologies and software development principles.

Uong SovanDara: dependable team player, resourceful problem solver, and skilled communicator with strong organizational awareness and developing leadership skills. knowledgeable about HTML, CSS, SQL, Python, Java, C++, and JavaScript. Proficiency in Oracle APEX and Assembly Language, along with an awareness of web development and server upkeep.

Alexander Redinger: Currently, I can program in Python, C++, C#, Java, Javascript, HTML, CSS. I am extremely familiar with Excel and Google Spreadsheets. I am experienced in data analysis, and modding video games.

1. Customer Problem Statement of Requirement

1a. Problem Statement.

The Challenge of Balance and Customization in Total War: Rome II

The Total War: Rome II multiplayer community is experiencing a resurgence, largely thanks to a renewed interest in game modding. As more players venture into creating their own content, a significant problem has emerged: a lack of balance in the new mods. These imbalances often arise from a fundamental misunderstanding of unit value, a complex issue in a game where a unit's effectiveness is highly dependent on the specific in-game situation. A modder might see a unit perform exceptionally well in one engagement and, as a result, mistakenly increase its cost or downgrade its abilities, leading to a mod where units are either overpriced for their utility or so powerful for their cost that they become "overpowered" and ruin the competitive integrity of the game.

This challenge extends beyond simple balance. Many players find it difficult to find a faction in the vanilla game that perfectly aligns with their personal playstyle. While Rome II offers a diverse range of factions, each with its own set of strengths and weaknesses, players often feel they must compromise. For instance, a player might admire the defensive strength and discipline of Roman infantry but wish they could combine it with the rapid, shock-and-awe cavalry tactics of the Seleucids. However, without modding, this kind of hybrid faction is simply not possible. This forces players to choose between their preferred unit types, preventing them from experiencing the game with a truly tailored army.

The process of creating a custom, balanced faction is a formidable obstacle for most players. Total War: Rome II is an older game, and the resources available for new modders are few and far between. Even for those who manage to navigate the steep learning curve, the work itself is tedious and incredibly time-consuming. The bulk of the process involves manually editing large spreadsheets, a task that can take hours just to create a single faction. This combination of a lack of support and a high time commitment discourages many players from even attempting to create the custom content they so desperately want. The current system is a bottleneck, where a great idea for a custom faction often dies before it can even begin to be built.

Customer Stories

Negan's Quest for a Balanced Fantasy

Negan is a dedicated and highly skilled Total War: Rome II player, deeply embedded in the game's multiplayer community. While he is proficient with all of the base game's factions, his true passion is to merge his love for Rome II with his deep appreciation for the world of Lord of the Rings. He's scoured the internet for mods that do this, but has been consistently disappointed. The existing mods are often incomplete, and many suffer from poor balance, where certain units are laughably overpowered for their cost, which completely undermines the fun and competitive spirit of the game.

Negan's ultimate fantasy is to command an army of Elves in an epic confrontation against the Romans. He doesn't just want to participate in these battles; he wants to create the Elven faction himself, with units that accurately reflect his extensive knowledge of Lord of the Rings lore. He is confident that his vision is the most faithful and discerning. However, Negan is a busy individual with limited free time. He simply lacks the time and the desire to learn the complex and painstaking process of modding Rome II. As a player, not a developer, he wants a solution that allows him to bring his custom faction to life without spending hours on manual spreadsheet editing and complex development work.

Graikos' Struggle with Subjective Modding

Graikos is a talented mod developer for Total War: Rome II who has created a mod that allows players to build their own custom factions. His process involves having players select 15 of their favorite units from the game, which he then compiles into a unique faction. To ensure some level of balance, he asks players to choose a specific culture, which provides a framework of strengths and weaknesses. However, the pricing of each unit is determined entirely by Graikos' subjective "gut feeling." This highly personal approach to balancing means that his personal biases can affect the mod; he may unintentionally lower the prices for a player he likes or unfairly inflate them for someone he doesn't.

This subjectivity has become a major source of stress for Graikos. The players who use his mod frequently complain about the lack of balance and the inconsistent pricing, which puts a

significant strain on his already busy schedule. The manual nature of his work only adds to the pressure. He must change every single unit stat by hand, one at a time. A single mistake requires a long, frustrating debugging process to pinpoint the error. To attract more players, he has even started taking on requests for custom visual designs, which is an even more labor-intensive and time-consuming process than the stat changes. To make matters worse, there are even fewer resources available to help with the visual modding of Rome II, leaving Graikos to navigate this complex process almost entirely on his own. The joy he once found in modding has been replaced by the stress of managing player complaints and the tedious, manual work of development.

Can you write a conclusion that sums up the two user stories?

Conclusion: The Pain Points of Rome II Modding

Both Negan and Graikos represent the dual challenges facing the Total War: Rome II modding community. On one hand, you have the player who wants custom content but lacks the time and technical expertise to create it. Negan's story highlights the frustration of a passionate gamer who is forced to settle for poorly balanced, generic mods because the tools to build his perfect, lore-accurate faction simply aren't accessible to him. He is a creative user with a clear vision, but the complex and manual nature of the modding process is a massive barrier.

On the other hand, you have the mod developer who is overwhelmed by the manual, subjective, and time-consuming nature of the work. Graikos's story shows that even someone with the skills to create mods struggles under the weight of manual data entry, subjective balancing decisions, and the stress of dealing with a demanding community. The current process forces him to make every change by hand, leaving no room for a systematic, objective approach to balance. He is a bottleneck, and his personal feelings and limited time directly impact the quality and consistency of his work.

In essence, the core problem is a broken ecosystem: modding Rome II is a difficult and laborious process that is neither user-friendly for players nor efficient for developers. This leads to a vicious cycle where players can't get the custom, balanced content they crave, and the developers who try to provide it are pushed to their limits by the manual, tedious nature of the work.

1 b.) Decomposition into Sub-problems

1. Unit data management
    a. How do we gather, store and present unit stats in a structured way?
    b. Solution: build a backend/database that holds unit stats in accessible format
2. Custom Faction Builder
    a. How do we allow users to select units and form custom factions without the need for complex spreadsheets?
    b. Solution: Create a User Interface for players to create custom factions and and automatically generate faction data
3. Balance Calculator
    a. How do we prevent subjective balancing?
    b. Solution: Develop functions that will assign costs to the units based on weighted stats and balanced formulas
4. Lore and Customization Layer
    a. How do we let users name their custom lore accurate factions into them game while still maintaining balance?
    b. Solution: Provide options for faction themes and allow users to visually customize their faction's identity
5. Data editing
    a. How to remove manual data entry into a spreadsheet?
    b. Solution: Export mod-ready files directly from the calculator with minimal human editing
6. Error Prevention and Debugging
    a. How do we prevent hours of debugging to find errors?
    b. Solution: Implement validation(checking for missing stats, invalid costs), before export
7. User Accessibility and Time Efficiency
    a. How do we make this usable for casual players with little technical knowledge?
    b. Solution: Create a user interface(web app) that allows user to select units and click "Generate Mod"

1c. Glossary of Terms

Administrator: An individual with the authority and responsibility to configure and manage a system.

API (Application Programming Interface): An API is a set of rules that enable different applications to communicate with each other. In this system's context, an API is used for data transfer and to utilize the functionality of external third-party systems.

Application: A software package that performs a specific function for an end user or another Application.

Developer: An individual that creates software using code.

User: An individual that interacts with or utilizes a systems resources and services

Graphical User Interface (GUI): An interface that allows users to interact with a computer or electronic device.

Faction: A collection of units a player picks from to create an army

Unit: A group of soldiers the player commands in-game.

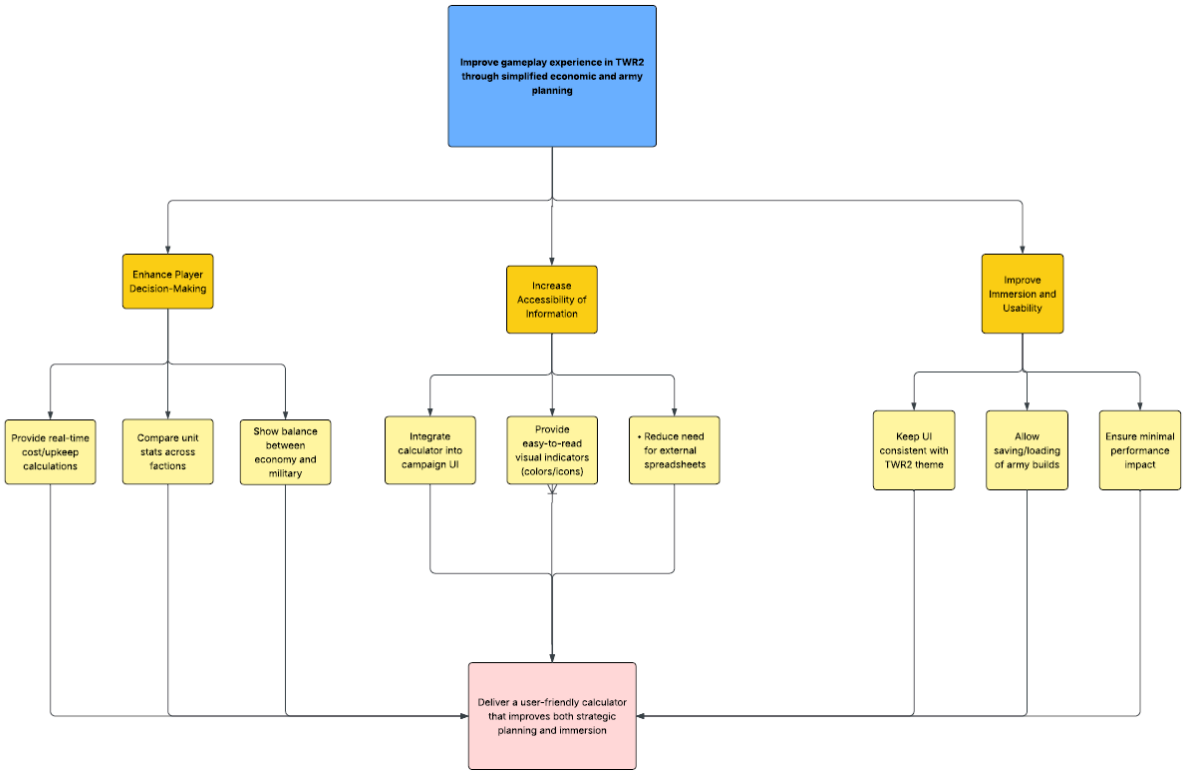
Balancing (of units): Correctly determining the price a unit is worth.

Vanilla: Playing the game without any mods.

Mod: A 3rd party modification of the game.

2. Goals, Requirements, and Analysis

2a. Business Goals



2b. Enumerated Functional Requirements

Identifier	PW	Requirement
REQ1	5	The system shall calculate and display the total cost of a unit given certain properties of the unit.
REQ2	5	The system shall provide the user all the tools necessary to create their own faction.
REQ3	5	The system shall format the information given by the user into a way identical to the game files for easy integration into the game.
REQ4	4	The system shall provide the user with Error Codes, and methods to fix them to ensure certain rules are unbroken.
REQ5	3	The system shall provide a method to keep every users' calculator up to date.
REQ6	3	The system shall keep track of a users' faction across sessions, and allow the admin to access their faction.
REQ7	2	The system should allow user to customize the graphical aspects of their units.
REQ8	4	The system shall provide the user with all necessary information in order to create their units.

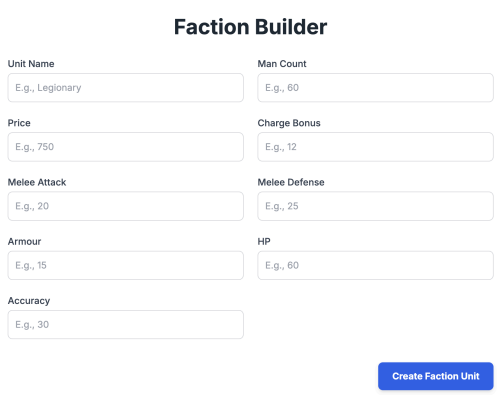
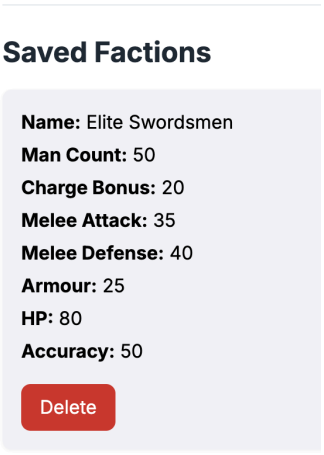
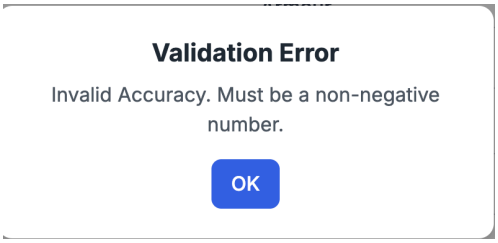
2c. Enumerated Nonfunctional Requirements

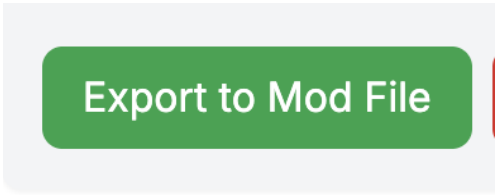
Identifier	PW	Requirement
REQ9	5	The faction builder shall provide a simple and intuitive user interface with clearly labeled input fields.
REQ10	3	The system shall comply with WCAG 2.1 accessibility standards for usability and inclusivity.
REQ11	5	All submitted faction data shall be reliably stored without corruption or loss
REQ12	3	The system shall display clear error messages and prevent crashes when invalid data is entered.
REQ13	5	The system shall process player submissions and update Sheets within 2 seconds
REQ14	5	The system shall allow easy integration of a new faction into the master mod.
REQ15	3	The system shall scale to support at least 100 players created factions without significant performance issues.
REQ16	5	The project shall be maintained in GitHub with documentation and remain compatible with the latest version of Total War:Rome II
REQ17	3	The web builder shall run reliably on modern browsers (Chrome, Firefox, Edge etc.)
REQ18	2	The system shall require internet connectivity for faction creation and integration.

FURPS Table

Category	Description
Functionality	● Provides a web based faction builder with input fields for unit composition, resources, and attributes. ● Automates calculations in and integrates results into the master mod. ● Enforces balancing rules such as unit caps and upkeep thresholds.
Usability	● The interface will be clear, simple, and accessible to new players. ● Input fields will be labeled, with tooltips for guidance. ● WCAG 2.1 accessibility standards will be followed.
Reliability	● Data will be stored securely and without corruption. ● Invalid inputs will generate error messages rather than crashes. ● Versioning ensures backup and recovery.
Performance	● Factions integrate into the master mod with minimal effort ● The system scales to 100+ factions without major slowdowns.
Supportability	● Source code will be documented and versioned in GitHub. ● Compatible with the latest Total War: Rome II Assembly Kit. ● Supported on modern browsers across platforms.

2 d) **User Interface Requirements**

Identifier	Priority	Requirement
REQ9	5	 A user should be able to input custom faction
REQ11	5	 Factions should be reliably stored without corruption or loss
REQ12	3	 A user should be able to see clear error messages

REQ13	5	 System should process player submissions
REQ14	5	 System should allow for easy integration to export to master mod

3. Functional Requirements Specification and Use Cases

3a. Stakeholders

The Calculator Mod provides value not only to players but also to the wider Total War modding community and supporting platforms. Having a clear picture of stakeholders ensures the system meets the needs of all groups who have a vested interest in its success.

Primary Stakeholders

- **Players (Casual and Competitive):** End-users who want to customize their Rome II experience by creating balanced factions. They gain an easy-to-use builder that removes technical barriers.
- **Community Modders:** Contributors who add content, test balance, and expand the master mod with creative new factions. Their input ensures replayability and long-term community engagement.
- **Developers (Team D):** The project team members who design, code, test, and integrate the system. Their main interest is delivering a functional and innovative modding tool that works reliably.

Secondary Stakeholders

- **Administrators:** Individuals responsible for maintaining the shared mod, ensuring updates, managing version control, and troubleshooting errors.
- **Total War Community (Broader Audience):** Players and online communities who benefit indirectly from a growing pool of balanced factions available for download.
- **Creative Assembly (Game Developer):** While not directly involved, their Assembly Kit and modding framework define the technical boundaries within which this project operates.

Supporting Platforms

- **Google Sheets API:** Provides backend computations for unit balance and upkeep.
- **W3Schools Spaces:** Hosts the faction builder web app, ensuring reliable online access.

- **GitHub Repository:** Maintains version control, collaboration, and distribution of the master mod.

3b. Actors and Goals

Actors	Participating/ Initiating	Role	Goal
Player	Initiating	End-user who wants to build a custom faction.	To create a personalized, balanced faction quickly without technical modding knowledge.
Community Modder	Initiating/Participating	Experienced modder who may both use and test the system.	To add creative factions, test balance, and expand the master mod for replayability.
Administrator	Participating	Maintains the system, reviews submissions, and manages the shared mod/master mod	To ensure the system runs smoothly, manage errors, and coordinate integration of factions.
Rome II Assembly Kit	Participating	Official creative assembly modding tool	To compile and integrate newly created factions into the game environment.
GitHub Repository	Participating	Source control and distribution platform	To store master mod files, provide version control, and make updates accessible to the community.
W3School Spaces	Participating	Hosting platform for the web application	To provide stable, accessible online hosting so players can use the faction builder anywhere.

3c. Use Cases

i. Casual Description

UC1: User Creates Faction

Description: User fills in every customization option and stat slot for their units.

Responds to Requirements: REQ1, REQ2, REQ4, REQ5, REQ6, REQ7, REQ8, REQ9, REQ10,REQ11, REQ12, REQ17, REQ18

UC2: User Updates their Faction

Description: User makes changes to their already-existing faction.

Responds to Requirements: REQ5, REQ6, REQ11, REQ12, REQ17, REQ18

UC3: Modder Integrates Units into Game

Description: Modder puts the units from the calculator into the game.

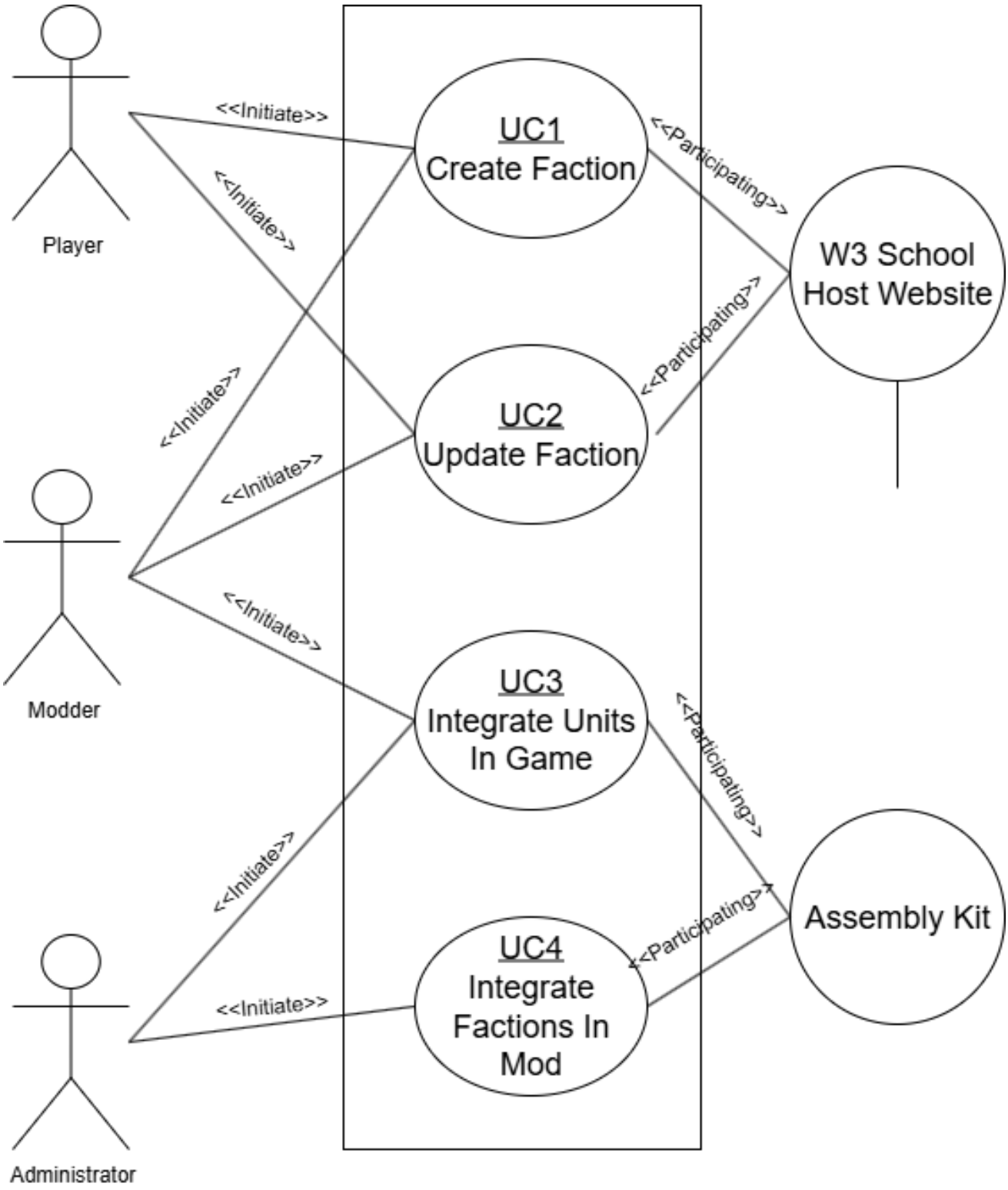
Responds to Requirements: REQ3, REQ14, REQ17, REQ18

UC4: Developer Integrating Others' Units into Mod

Description: Developer accessing others' factions to put into master mod.

Responds to Requirements: REQ3, REQ5, REQ6, REQ11, REQ13, REQ14, REQ15, REQ16, REQ17, REQ18

ii. Use Case Diagram



iii. Traceability Matrix

Requirements	PW	UC1	UCC2	UC3	UC4
REQ1	5	X			
REQ2	5	X			
REQ3	5			X	X
REQ4	4	X			
REQ5	3	X	X		X
REQ6	3	X	X		X
REQ7	2	X			
REQ8	4	X			
REQ9	5	X			
REQ10	3	X			
REQ11	5	X	X		X
REQ12	3	X	X		
REQ13	5				X
REQ14	5			X	X
REQ15	3				X
REQ16	5				X
REQ17	3	X	X	X	X
REQ18	2	X	X	X	X
Total PW	70	47	19	15	39

iv. Fully-Dressed Descripti on

UC1: User Creates Faction

Related Requirements: REQ1, REQ2, REQ4, REQ5, REQ6, REQ7, REQ8, REQ9, REQ10,REQ11, REQ12, REQ17, REQ18

Initiating Actor: User, or Modder

Participating Actor: None

Preconditions: The user interface allows user to enter information and select options

Postconditions: The data is saved and accessible by developer

Flow of Events for Main Success Scenario:

1. User enters Faction Customization Options
2. User enters unit stats, the calculator shows the appropriate outputs in response

UC2: User Updates Faction

Related Requirements: REQ5, REQ6, REQ11, REQ12, REQ17, REQ18

Initiating Actor: User, or Modder

Participating Actor: None

Preconditions: The user's data is accessible by the user

Postconditions: The data is saved and accessible by developer

Flow of Events for Main Success Scenario:

1. User opens up their faction
2. User modifies unit stats, the calculator shows the appropriate outputs in response

UC3: Modder Integrates Units into Game

Related Requirements: REQ3, REQ14, REQ17, REQ18

Initiating Actor: Modder

Participating Actor: None

Preconditions: The modder has a faction already filled out

Postconditions: The units are usable in-game.

Flow of Events for Main Success Scenario:

1. Modder opens up their faction
2. Modder accesses Backend, and copy-pastes into game files
3. Units are available without errors.

UC4: Developer Integrating Others' Units into Mod

Related Requirements: REQ3, REQ5, REQ6, REQ11, REQ13, REQ14, REQ15, REQ16, REQ17, REQ18

Initiating Actor: Developer

Participating Actor: None

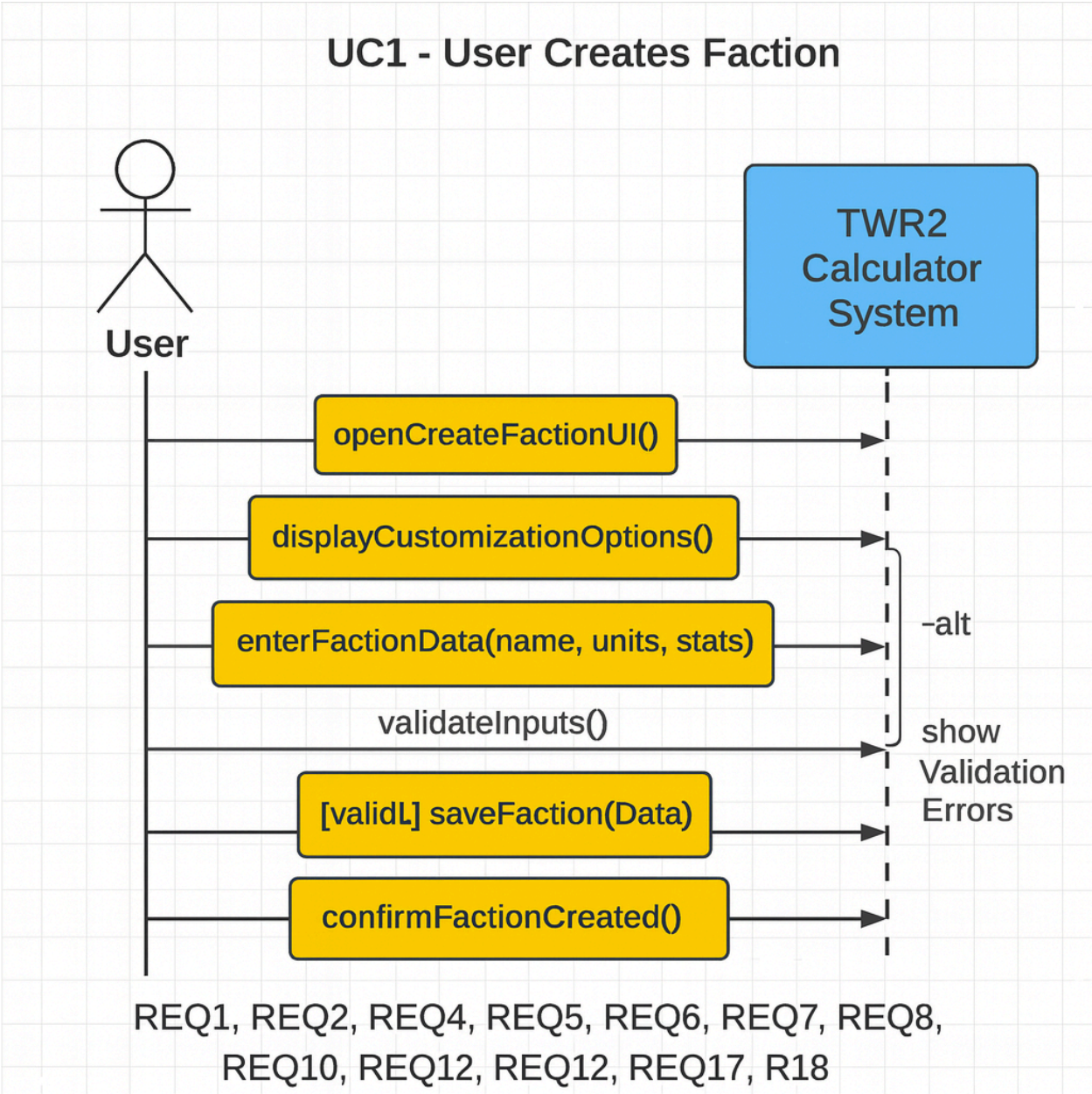
Preconditions: The developer has access to others' factions

Postconditions: The factions are usable in the mod

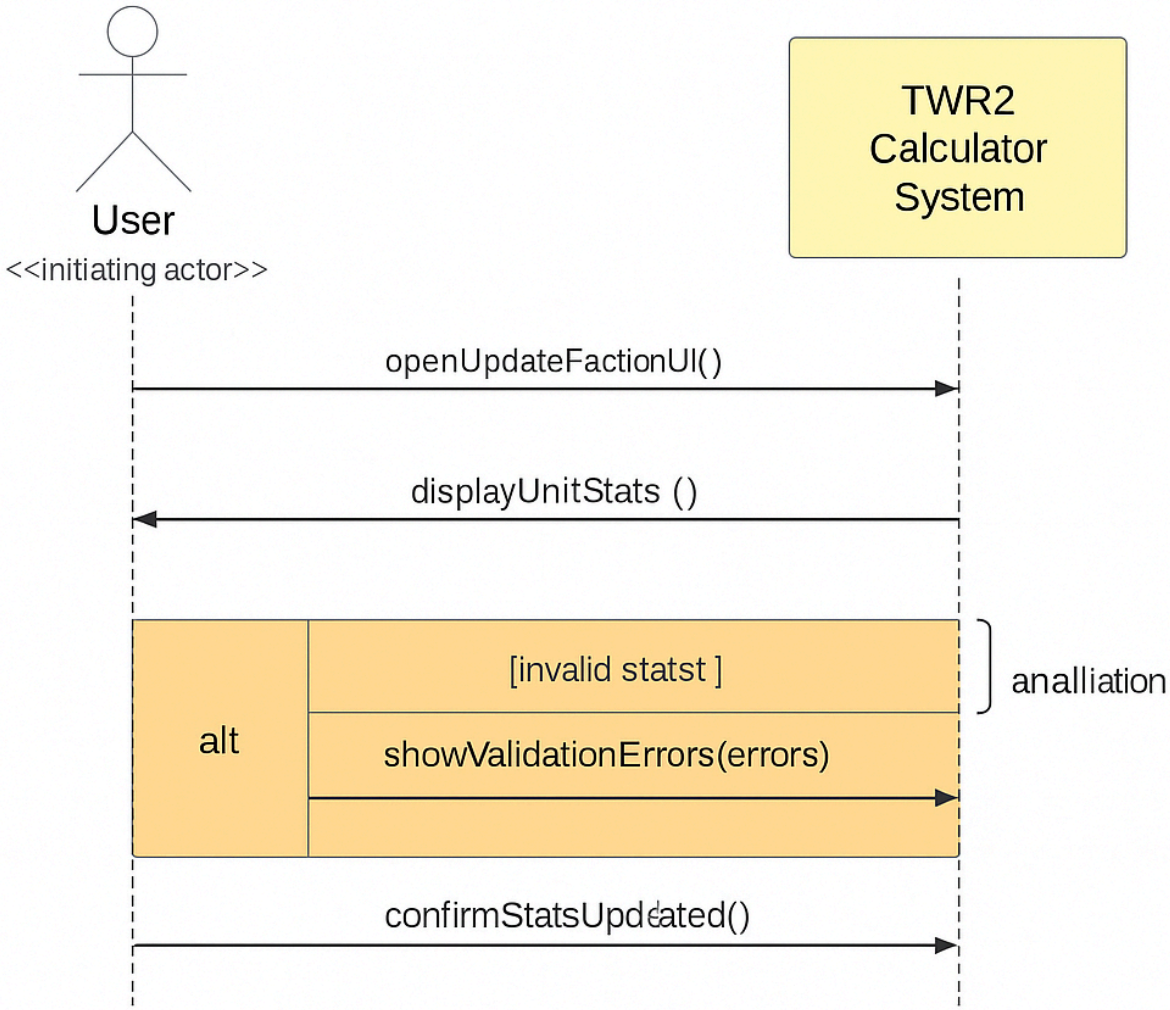
Flow of Events for Main Success Scenario:

4. Developer opens others' faction
5. Developer accesses Backend, and copy-pastes into game files
6. Factions are available without errors.

3d. System Sequence Diagrams

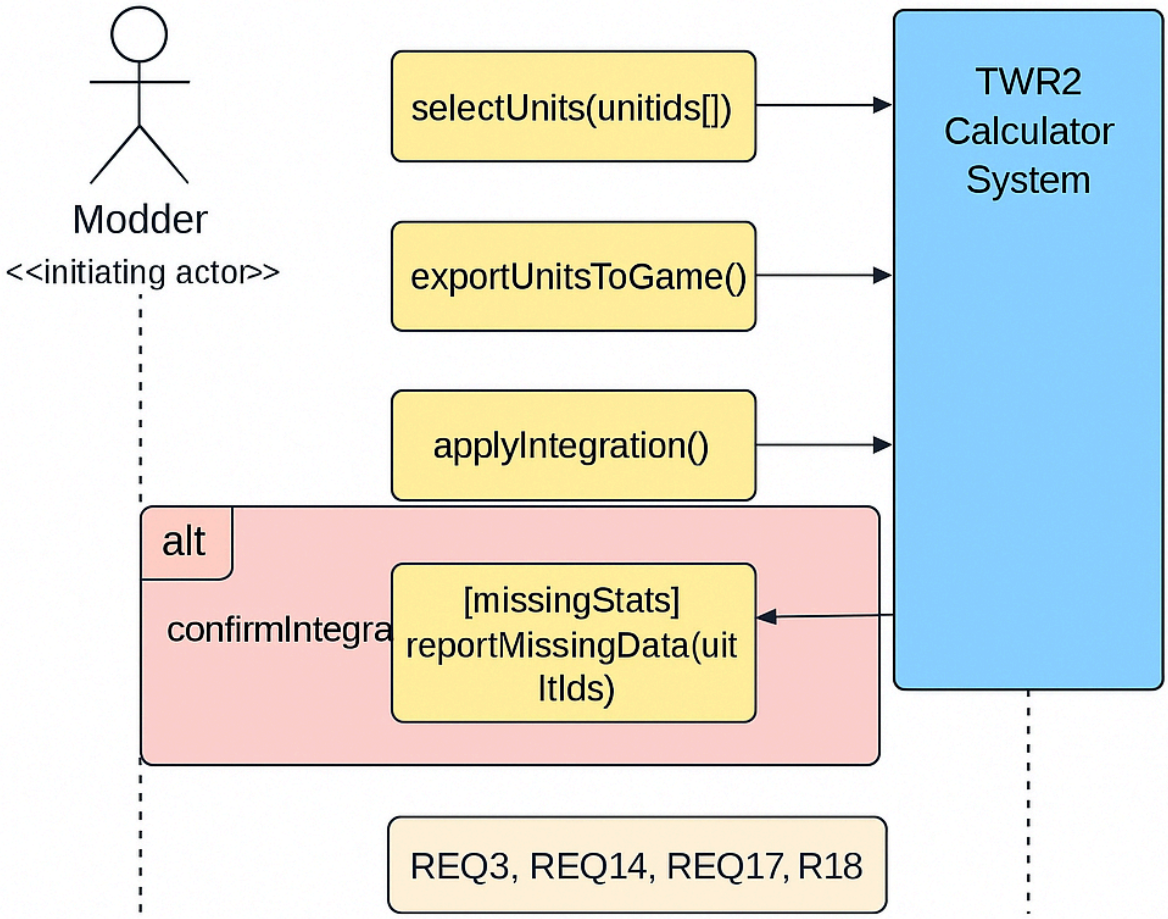


UC2: User Updates Faction

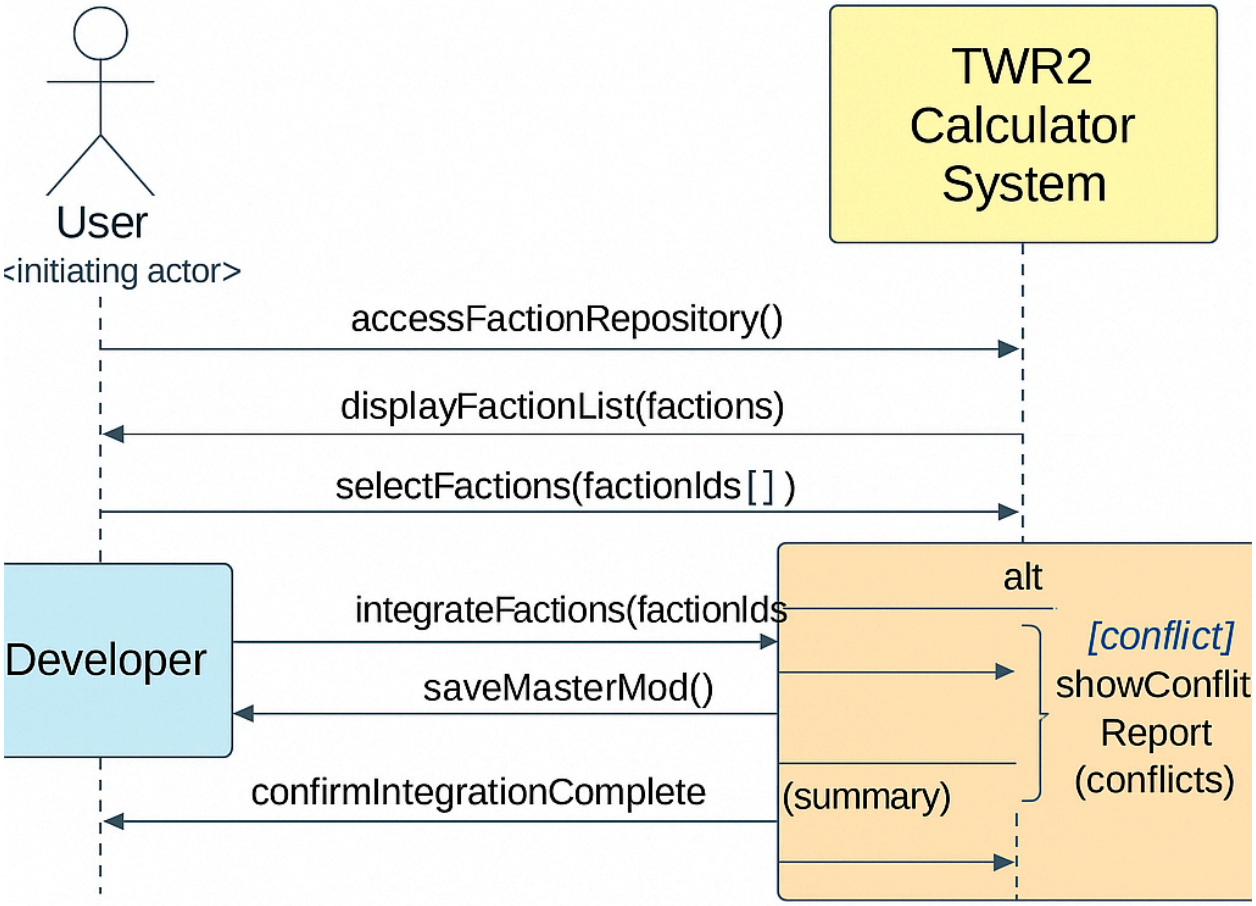


Related Req: REQ6, REQ11, REQ12, REQ17,

UC3: Modder Integrates Units in Game



UC4 - Developer Integrates Factions in Mod



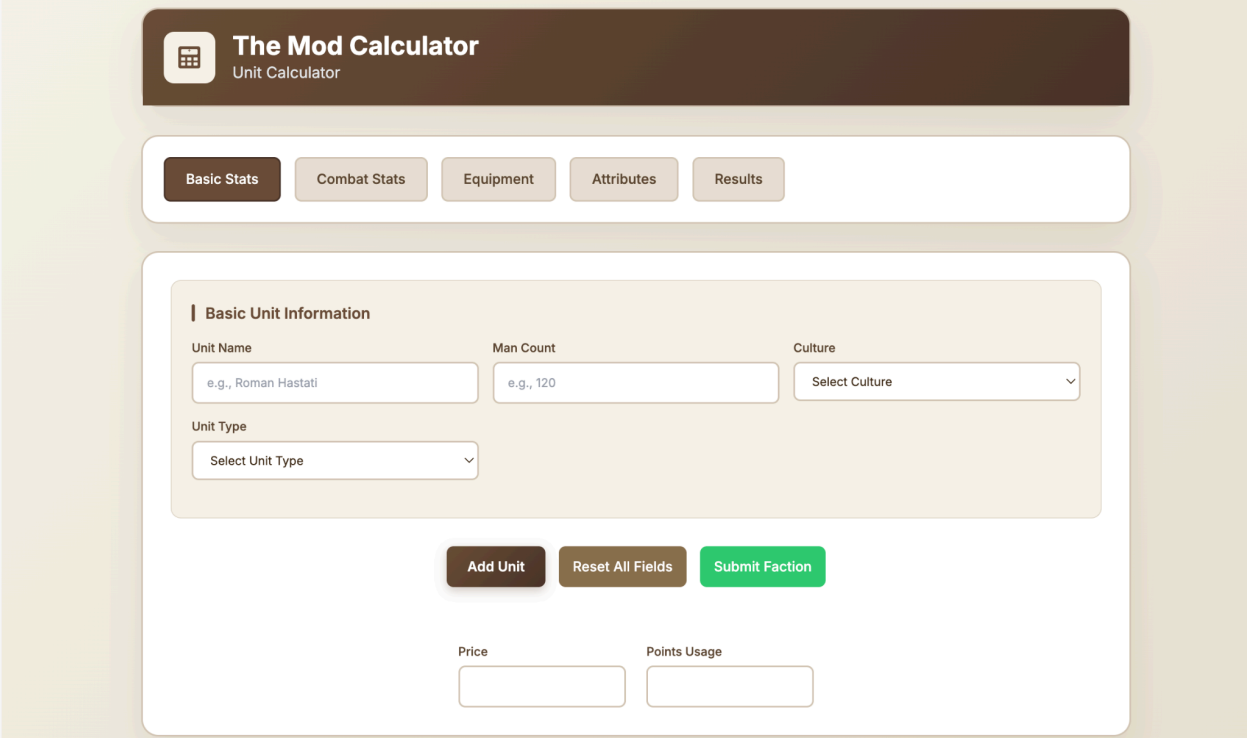
REQ3, REQ5, REG6, REQ11, REQ11, REQ14, REQ14, REQ14, REQ17
REQ17, REQ18, REQ17, REQ18

4. User Interface Specification

4a) Preliminary Design

1. Open the Faction Builder
    a. User selects “Create New Faction” from the navigation bar
    b. System displays the Faction Builder form, including input fields for unit attributes and dropdown menus for culture and unit type
2. Enter Unit Information
    a. The user then fills in the chosen fields Name, Man Count, Charge Bonus, Morale, Melee Attack, Melee Defense, Armour, HP, Accuracy, Ammo etc.
    b. Drop down menus are displayed for Melee Weapon, Missile Weapon, Shield, Mount, attributes, and abilities.
    c. The system validates input in real-time and displays price and points usage.
3. Add Unit to Faction
    a. The user presses the “Add Unit” Button
    b. The system displays the unit in a table below the form, showing a summary of entered values
4. Submit Faction
    a. Once the user has added all desired units, they press the “Submit Faction” button.
    b. System saves the faction data, validates all fields, and exports the custom faction
    c. A confirmation message is displayed: “Faction submitted successfully.”

Figure: The Mod Calculator (Desktop)



4b) User Effort Estimation

UC1: User Creates a faction with all the customization options and stats for their units

1. Select “Basic Stats” → 1 click (navigation)
2. Enter Unit Name (x15 units) → ~ 10 keystrokes(clerical)
3. Enter Man Count (x15 units) → ~3 keystrokes (clerical)
4. Select “Combat Stats” (x15 units) → 1 click(navigation)
5. Enter Melee Attack, Melee Defense, Charge Bonus, Morale, Armour, HP, Accuracy(x15) → 3 clicks(clerical) + 6 keystrokes(clerical)
6. Select Equipment (x15)→ 1 click (navigation)
7. Select Primary Weapon, Shield Type, Missile Weapon, Ammunation, Accuracy, Range(x15) → 3 clicks (clerical) + 6 keystrokes (clerical)
8. Select Attributes→ 1 click (navigation)
9. Select Mount, Artillery/Animal/Chariot(x15) → 2 clicks (navigation) + 2 clicks (clerical)
10. Enter Num of Artillery/Animal/Chariot(x15) → 2 keystrokes (clerical)
11. Select “Add unit” button to add the unit(x15) → 1 click (navigational)
12. Select “Submit Faction” → 1 click navigational

Totals:

Keystrokes and Clicks = approx. 856

Navigation vs Clerical : ~ 9 % navigational vs 91.1% Clerical

UC2: User Updates Faction

1. Select Unit to Update → 1 click(navigation)
2. Change Unit Name → 10 keystrokes (clerical)
3. Adjust 3 stat fields (e.g, melee attack, Charge Bonus, HP) → 6 keystrokes (clerical)
4. Click “Save Changes” -> 1 click (navigation)

Totals:

16 keystrokes, 2 clicks

11% navigation vs. 88.85% clerical

UC3: Modder Integrates Units into Game

1. Click “Export Faction to Sheet” → 1 click (navigation)
2. Choose export location → 2 clicks (navigation) + 10 keystrokes (filename, clerical)
3. Open PACS file reader → 2 clicks (navigation)
4. Navigate through folders (~3–4 levels deep) → 4 clicks (navigation)
5. Open the target mod file → 1 click (navigation)
6. Switch to exported sheet → 1 click (navigation)
7. Copy values-> 2 keystrokes, keyboard shortcuts(clerical)
8. Switch back to PACS file → 1 click (navigation)
9. Paste values → ~2 keyboard shortcuts (clerical)
10. Save changes → 1 click (navigation)

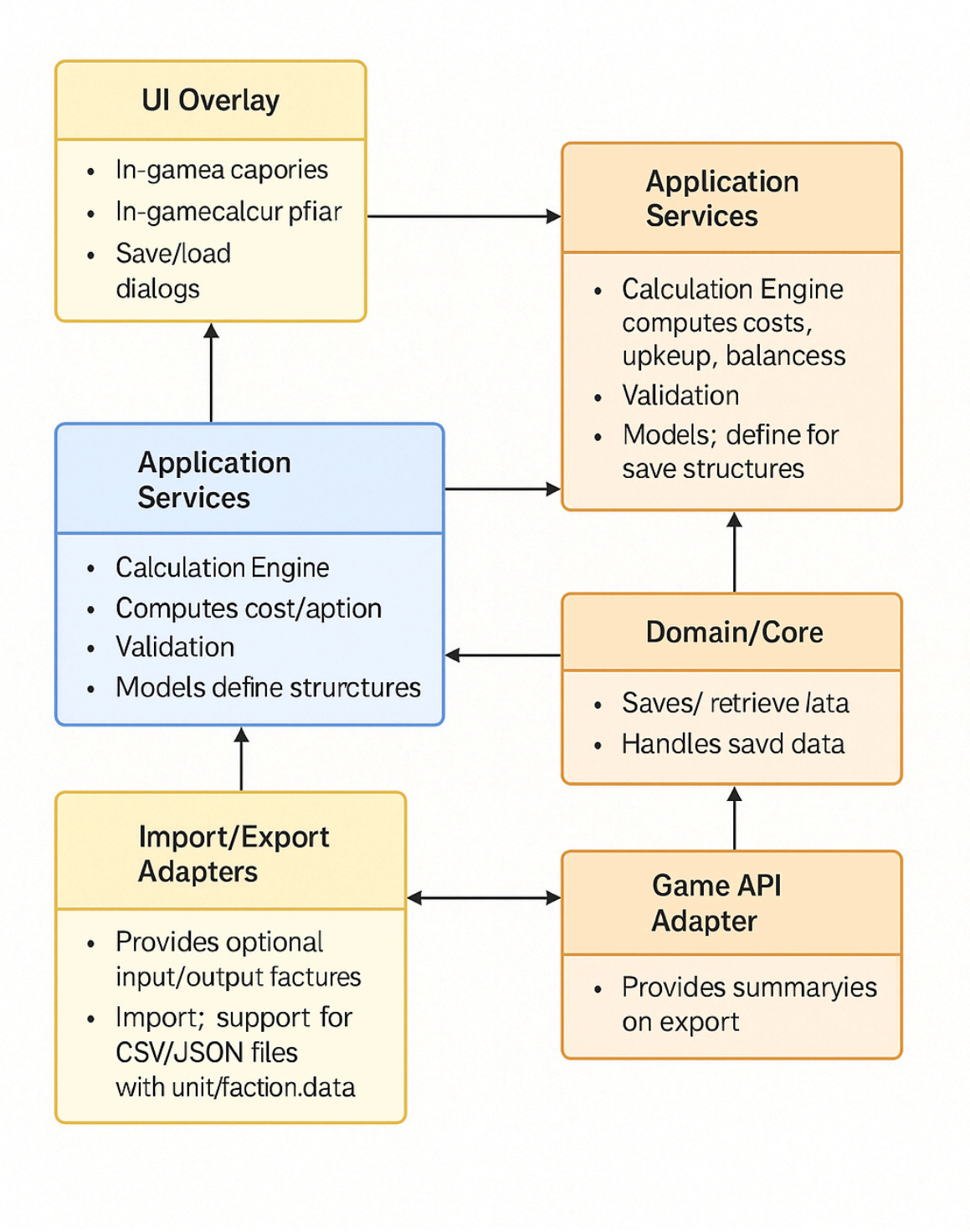
Totals 27 keyboards + clicks

48% Navigational

52% Clerical

5. System Architecture and System Design

5a. Identifying Subsystems



5b. Architecture Styles

We will develop The Calculator Mod as a monolithic web application. This design choice simplifies development and deployment by housing the user-facing website and all backend logic in a single, cohesive codebase. Users will interact directly with this website, which will be the central hub for creating and modifying custom factions. The application will leverage an event-driven architecture to dynamically update unit prices based on user actions.

The Calculator Mod will exist as a single, unified website where users can create their own custom factions. All components of the application, including the user interface (frontend), the core business logic (backend), and the database, will be tightly integrated into this single deployment. This architecture provides a straightforward user experience; a user simply navigates to the website and starts building their faction. There is no need for separate applications or complex setup processes. The application will serve as the sole platform for users to define their desired faction, including selecting units and customizing their stats.

The core functionality of The Calculator Mod will rely on an event-driven system. When a user makes a significant change, such as selecting a new faction or adding/removing a unit, an event will be triggered. This event will initiate a process that automatically calculates and updates the unit prices in real-time. For example, if a user changes their units to be more like Roman units, rather than Greek-style units, the system will adjust the price of all units to reflect the new cultural and gameplay attributes. The developer's workflow will be just as streamlined. After a user finishes creating their custom faction, the developer can access their data within the same monolithic application. This allows them to easily retrieve the completed faction data and integrate it directly into the game. This approach eliminates the need for manual communication or data transfer between the user and the developer, as all relevant information is stored and accessible in one place.

5c. Mapping Subsystems

- **Faction Buldier Subsytem:** This subsystem runs inside the user's browser. It provides the interface for entering unit stats (man count, attack, defence, etc), uploading banners. It also sends the user input to the server for validation, balance calculations, and storage

- **Database Subsystem:** Manages storage of user-created factions, unit stats, and customization options. Ensures persistence and supports scaling to 100+ factions. Hosted on the server, accessed via API calls from the Faction Builder

- **Balance Calculator Subsystem:** Processes unit data to ensure fair gameplay. Runs on the server for consistency and speed. Returns results to the browser in under 2 seconds.

- **Export and Integration System:** Generates output in Sheets/CSV formate for modders. Files are downloaded by the user’s browser. The exported file is then manually copied into the Total War: Rome II PACS mod structure by the modder (outside the sytem).

- **Communication Infrastructure:** Provides secure, reliable exchange between browser(client) and web server. Handles API calls, data submission, and error reporting

5d. Connectors and Network Protocols

The TWR2 Calculator Mod is designed to run locally on a single machine alongside Total War: Rome II. Because of this, the system does not depend on external networks or internet-based services. Instead, it relies on internal connectors and file-based protocols to manage communication between subsystems.

In-process calls.
The UI Overlay interacts with the Application Services through direct function calls. These calls then flow into the Domain/Core logic, ensuring that calculations for costs, upkeep, and comparisons are performed quickly and without noticeable delay for the player.

Event notifications.
To keep the interface responsive, certain actions within the Application Services trigger lightweight events, such as when totals are computed or a faction build is saved. These event signals allow the UI to update dynamically without requiring the player to restart processes.

File connectors.
Persistence is managed using JSON files for storing factions, units, and builds. When integrating these into the game, the Game API Adapter writes the data into pack or database files, which are the standard formats recognized by Rome II. This file-based connector approach allows the mod to maintain compatibility with existing game infrastructure.

Future extension.
Although not required for core functionality, the design could support online sharing of saved builds through HTTPS endpoints. This would provide players the option to upload and download builds while still allowing the mod to function entirely offline if network access is unavailable.

5e. Global Control Flow

- **Execution Orderness**:The system operates using an **event-driven architecture**. This means the program waits for specific events, such as user actions or system triggers, to occur. When an event happens, it activates a specialized handler to process the request. These handlers are designed to perform a specific task and return the results, allowing events to be processed in any order and at any time they happen.

- **Time Dependency:**The Calculator Mod is not a real-time system, meaning it doesn't have strict timing requirements for its core operations. Instead, it processes user events as they happen, and those events can occur at any time. The system's logic isn't dependent on a clock or a scheduler. It simply responds to user actions in an **on-demand fashion**. For example, when a user changes their chosen faction or modifies a unit's stats, the system immediately processes that event and recalculates the unit's cost. This ensures that the user always sees updated information as they build their custom faction, without any need to wait for a specific time or trigger.

5f. Hardware Requirements

- **Screen Display:** The Faction Builder subsystem will run on multiple devices such as laptops, desktops, and tablets. It must support modern web browsers at varying resolutions. Minimum supported resolution is 1280 x 720 pixels, but the system is optimized for 1920 x 1080 pixels or higher. A color display is required to view banners, icons, and faction customization options properly.

- **Communication Network:** The system requires a stable internet connection for real-time communication between the client (browser) and the server. A minimum 5 Mbps connection is recommended to ensure smooth interaction with the Balance Calculator, database access, and file export. Intermittent connectivity may result in delays or failed submissions.

- **Database Server:** The database subsystem will run on a server hosting PostgreSQL(or equivalent relational database). Minimum requirements include 2 CPU cores, 4 GB RAM, and 20 GB disk storage to support up to 100 user-created factions. Scaling beyond this capacity may require additional memory and disk space.

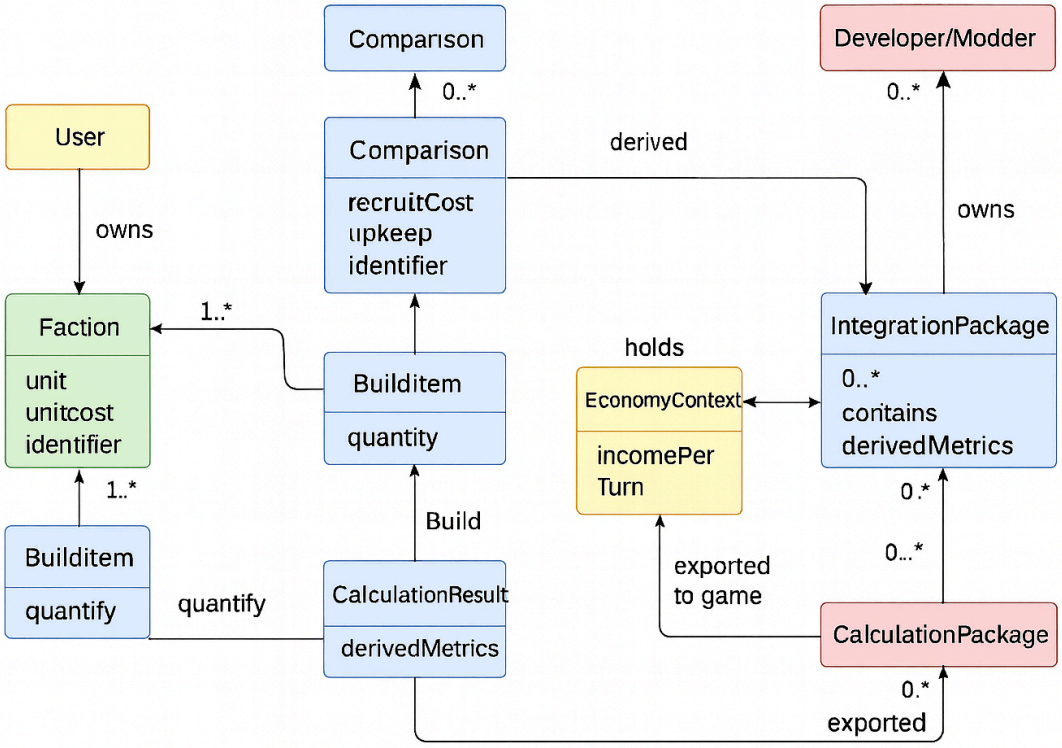
- **Application Server:** The server hosting the Balance Calculator and Export Subsystem must support Node.js(or equivalent backend runtime). Minimum requirements include 2 CPU cores and 4 GB RAM to ensure calculations are processed within 2 seconds.

- **User Machine:** Users must have a device capable of running modern web browsers. Minimum requirements are a dual-core processor, 4 GB RAM, and 500 MB of available disk space for the downloaded export file.

- **External Modding Environment:** to integrate exported factions into the ROME II game, modders will need a PACS file reader/editor installed on their local machine, along with sufficient disk space for Rome II game files.

6. Analysis and Domain Modeling

6a. Conceptual Model



i. Concept Definitions

Responsibility Description	Type	Concept Name
Store playable groups, their units, and associated bonuses.	D	Faction
Represent recruitable elements with costs, upkeep, and unique identifiers.	D	Unit
Hold detailed combat and economic stats for each unit (attack, armor, morale, etc.).	K	StatProfile
Save and load player-created army compositions for later use.	D	ArmyBuild
Represent each entry in an army build (unit + quantity).	K	Builditem
Provide computed totals such as cost, upkeep, balance, and efficiency indicators.	K	CalculationResult
Maintain income/budget context used to evaluate balance and affordability.	D	EconomyContext
Allow side-by-side comparison of units across factions with derived metrics.	D	ComparisonSet
Bundle selected units and factions for export into Total War: Rome II mod files.	D	IntegrationPackage
Represent the player who creates factions and builds.	D	User
Represent the developer or modder responsible for integrating content into the master mod.	D	Developer/Modder

ii. Association Definitions

Concept Pair	Association Description	Association Name
Faction ↔ Unit	A Faction contains one or more Units; each Unit belongs to exactly one Faction.	stores units
Faction ↔ ArmyBuild	A Faction may have multiple saved ArmyBuilds; each build belongs to a single Faction.	organizes builds
ArmyBuild ↔ BuildItem	An ArmyBuild is composed of BuildItems; each BuildItem references a specific Unit and its quantity.	composed of items
BuildItem ↔ Unit	Each BuildItem points to one Unit and specifies its quantity.	references unit
Unit ↔ StatProfile	Each unit has one stat profile with detailed stats.	provides stats
ArmyBuild ↔ CalculationResult	An army build produces a calculation result (totals, balance).	produces data
EconomyContext ↔ CalculationResult	A calculation result is computed within an economy context.	evaluates data
ComparisonSet ↔ Unit	A comparison set groups two or more units for evaluation.	compares data
IntegrationPackage ↔ Unit	An integration package includes units to be exported into the mod.	bundles units
IntegrationPackage ↔ Faction	An integration package can also include factions.	bundles factions
User ↔ Faction	A user creates and owns factions.	manages factions

User ↔ ArmyBuild	A user creates and owns army builds.	manages builds
Developer/Modder ↔ IntegrationPackage	A developer/modder generates integration packages for integration.	generates package

iii. Attribute Definitions

Concept	Attributes	Attribute Description
Faction	FactionId	Unique identifier for the faction.
	Name	The name of the faction.
	Description	Optional description of the faction's theme or purpose.
	Bonuses	Stores any economic or military modifiers applied to the faction.
Unit	UnitID	Unique identifier for the unit.
	Name	The name of the unit
	RecruitCost	One-time cost to recruit the unit.
	Upkeep	Recurring cost per turn to maintain the unit.
	Tags	Classifications such as "infantry" or "ranged."
StatProfile	ProfileID	Unique identifier for the stat profile.
	Attack	Attack strength of the unit
	Armor	Defensive armor value of the unit.
	Morale	Morale rating of the unit.
	Speed	Movement speed of the unit.
	Version	Version number of the stats (e.g., "1.0.0").
ArmyBuild	BuildID	Unique identifier for the army build.

	Name	The name assigned to the build.
	FactionID	Reference to the faction that owns this build.
	CreatedAt	Timestamp when the build was created.
	UpdatedAt	Timestamp when the build was last modified.
BuildItem	BuildItemID	Unique identifier for the build item.
	UnitID	The unit being referenced in the build.
	Quantity	The number of units of this type in the build.
CalculationResult	comparisonId	Unique identifier for the comparison.
	unitIds	List of unit IDs included in the comparison
	derivedMetrics	Calculated values like “cost per attack.”
IntegrationPackage	packageId	Unique identifier for the integration package.
	factionIds	List of factions being integrated.
	unitIds	List of units being integrated.
	exportPath	Location where the package will be saved.
	status	Current state (ready, applied, failed).
User	userId	Unique identifier for the user.
	displayName	The name or alias of the user.
Developer/Modder	devId	Unique identifier for the

		developer or modder.
	role	Defines whether the actor is a developer or modder.

IV. Traceability Matrix

Domain Concepts	UC1: User Creates Faction	UC2: User Updates Faction	UC3: Modder Integrates Units in Game	UC4: Developer Integrates Factions in Mod
Faction	x	x		x
Unit	x	x	x	x
StatProfile	x	x	x	x
ArmyBuild		x		x
BuildItem		x		x
EconomyContext	x	x		
CalculationResult	x	x		
ComparisonSet	x		x	x
IntegrationPackage			x	x
User	x	x		
Developer/Modder			x	x
Use Case	26	25	16	19

Justification

Why ComparisonSet appears in UC1, UC2, UC4: During Create Faction (UC1) and Update Faction (UC2), the user compares candidate units/stats to decide what to include—this uses a temporary ComparisonSet to show side-by-side metrics (REQ2). In Developer Integrates Factions (UC4), the developer may compare included units to verify balance before packaging, again using ComparisonSet as a transient analysis object.
Text Description of Traceability Matrix

Why not UC3: Modder Integrates Units (UC3) focuses on export/integration, not on analysis; it relies on Unit/StatProfile and IntegrationPackage, not the comparison helper.

UC1 — User Creates Faction

- **User:** starts the flow and submits inputs.
- **Faction:** new container the user is creating.
- **Unit:** items the user adds to the faction's catalog.
- **StatProfile:** provides the stats attached to each unit.
- **EconomyContext:** the income/budget the user selects for calculations.
- **CalculationResult:** totals (cost, upkeep, balance, indicator) produced from the user's selections.

UC2 — User Updates Faction

- **User:** selects an existing faction to modify.
- **Faction:** loaded record that is being changed.
- **Unit / StatProfile:** units or stats the user edits or replaces.
- **ArmyBuild:** saved compositions the user may update.
- **BuildItem:** individual lines (unit + quantity) inside a build that can change.
- **EconomyContext:** updated income/budget used to recompute values.
- **CalculationResult:** refreshed totals after edits.

UC3 — Modder Integrates Units in Game

- **Developer/Modder:** initiates the integration step.
- **Unit / StatProfile:** source data that defines the units to export.
- **IntegrationPackage:** bundle created for export into game files (pack/DB); represents the units being integrated.

UC4 — Developer Integrates Factions in Mod

- **Developer/Modder:** selects which content to merge into the master mod.
- **Faction:** high-level content chosen for integration.
- **Unit / StatProfile:** unit definitions that travel with the faction.
- **ArmyBuild / BuildItem:** optional saved builds included with the faction.
- **IntegrationPackage:** final package written to the mod, containing selected factions (and their units/builds).

6b. System Operation Contracts

Operation	**User creates faction**
Use Case	UC1
Preconditions	● User is authenticated in the system ● Faction Builder form is open and ready for input
Postconditions	● Faction data is validated and stored in the database ● Balance rules are applied automatically ● A confirmation message is displayed: "Faction submitted successfully."
Exceptions	● Invalid or missing input fields → system rejects submission and displays error message.

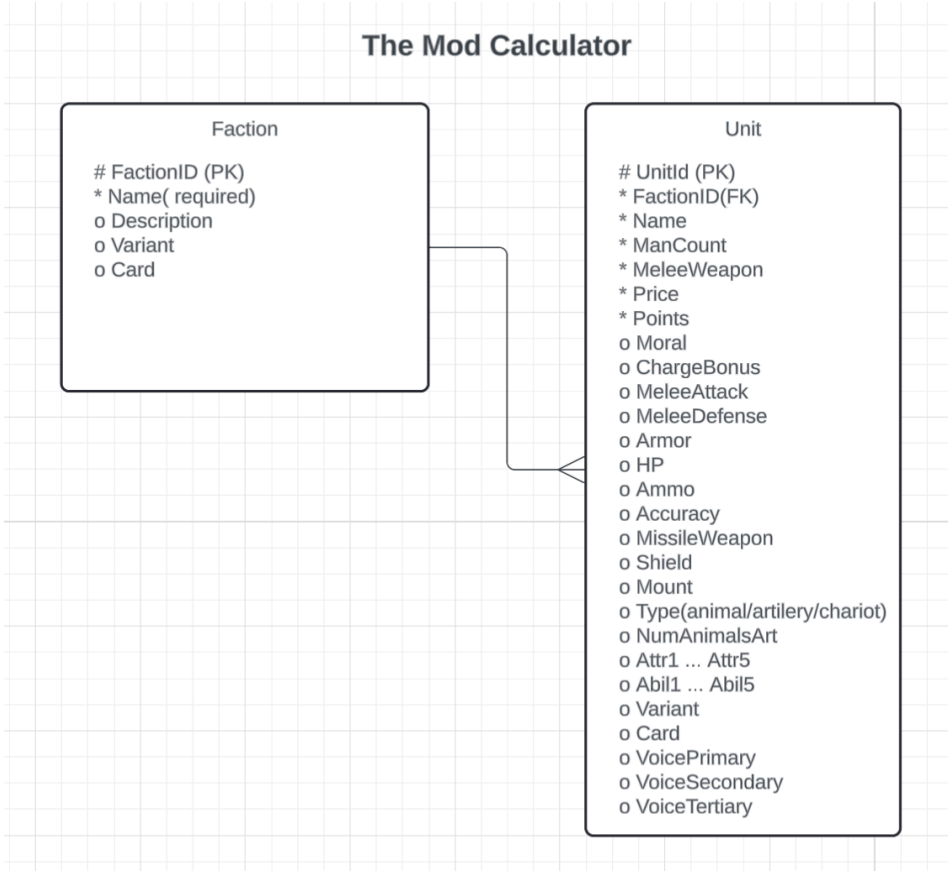
Operation	**User Updates Factions**
Use Case	UC2
Preconditions	● The faction already exists in the database. ● User is authenticated and authorized to modify the faction.
Postconditions	● Database records for the faction are updated with new values. ● Validation rules are reapplied to ensure balance compliance. ● Confirmation message is displayed: "Faction updated successfully."
Exceptions	● Faction ID not found → error messages shown. ● Invalid data → update is rejected.

Operation	**Modder integrates units into game**
Use Case	UC3
Preconditions	● Faction exists in the system. ● Export subsystem is online and functioning.
Postconditions	● Export file (CSV/Sheets format) is generated successfully. ● File is available for download. ● System displays status message confirming export.
Exceptions	● Invalid faction ID → export canceled. ● Export service unavailable → error logged and displayed.

Operation	**Developer integrates others units**
Use Case	UC4
Preconditions	● Developer has access to valid exported faction files. ● Rome II Assembly Kit is installed and operational.
Postconditions	● Faction is successfully compiled into the shared master mod. ● Updated mod is versioned and stored in GitHub. ● Confirmation message is displayed: “faction integrated successfully.”
Exceptions	● Invalid or corrupted file → integration aborted. ● Compatibility issue with Assembly Kit → error logged.

6c. Data Model and Persistent Data Storage (Updated)

Our system will allow users to save their created factions and return later to view, edit, or delete them. The system uses a flat file strategy with JSON as the storage format. Factions are stored either in Local browser storage via IndexedDB, ensuring persistence between sessions on the same device, or they can be downloadable JSON files, which will allow users to back up and reload their factions manually. The database system we will use is IndexedDB, which supports storing objects directly without requiring stringification. It is scalable, capable of holding hundreds of megabytes, and supports multiple object stores (similar to tables). In our design, one store is dedicated to Factions, and another to Units, linked by a foreign key relationship (FactionID).



.

6d. Mathematical Model

The bulk of the Calculator Mod's usefulness lies in its "Mathematical Model." The calculation of unit prices is a complex mathematical model. Essentially it works in a few stages:

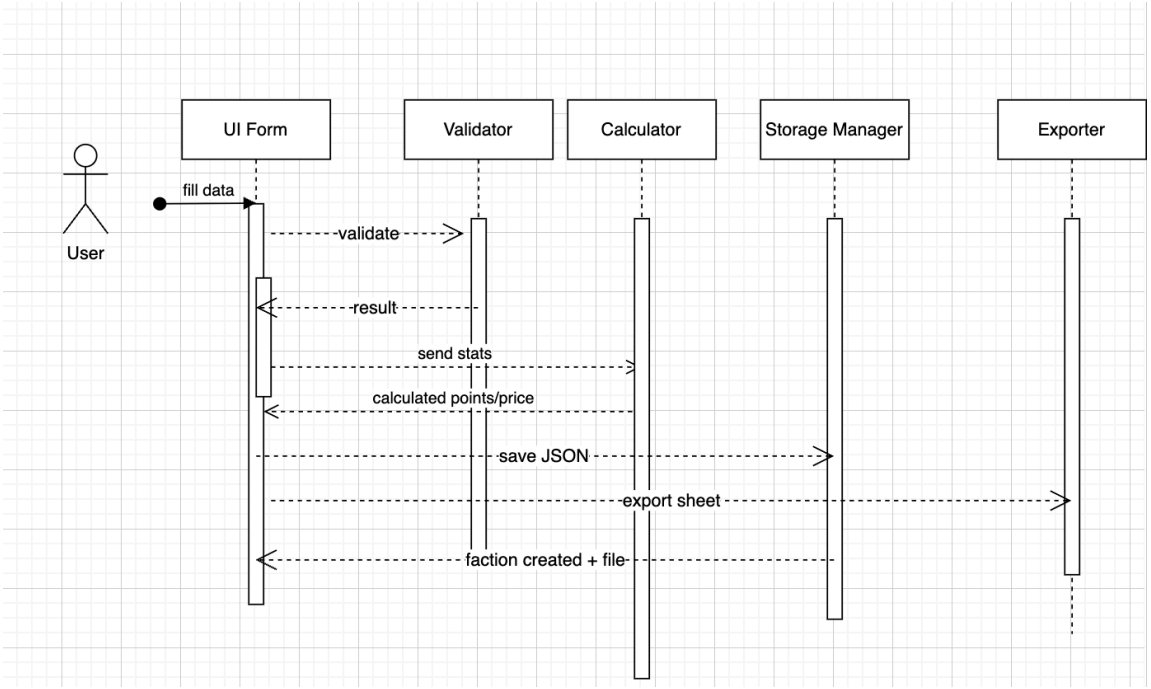
1. Category Calculation
    a. The system determines what kind of unit the user has input
    b. The prices for many stats are based on the category of the unit
2. Stat Assessment
    a. The system calculates the real value of the stat, which could different than what the user put in based on the weapon of the unit
    b. The system checks whether, based on the category of the unit, it breaks any rules
    c. If it breaks a rule, it will present an error code to the user, so they can fix it
3. Stat Calculation
    a. Every stats' price is priced in tiers, the code looks like this:

```
chgCost=(
            Math.min(chgBonus, 20)*1+
            Math.max(Math.min(chgBonus-20,5),0)*6+
            Math.max(Math.min(chgBonus-25,5),0)*8+
            Math.max(Math.min(chgBonus-30,5),0)*9+
            Math.max(Math.min(chgBonus-35,5),0)*11+
            Math.max(Math.min(chgBonus-40,5),0)*13+
            Math.max(Math.min(chgBonus-45,5),0)*15+
            Math.max(chgBonus-50,0)*18
        )
```

    b. This means that the price of a stat is a pile, where each increment of five adds a different amount to the price.
    c. The price for each stat increment is based entirely on in-game experience and testing.
4. Ability and Attributes
    a. These are flat price increases to the unit, based on a few factors.
    b. Certain kinds of units get free ability slots, otherwise abilities are a flat price increase
    c. Some attributes are inherent to certain categories, and are thus free, otherwise they're a flat price increase.

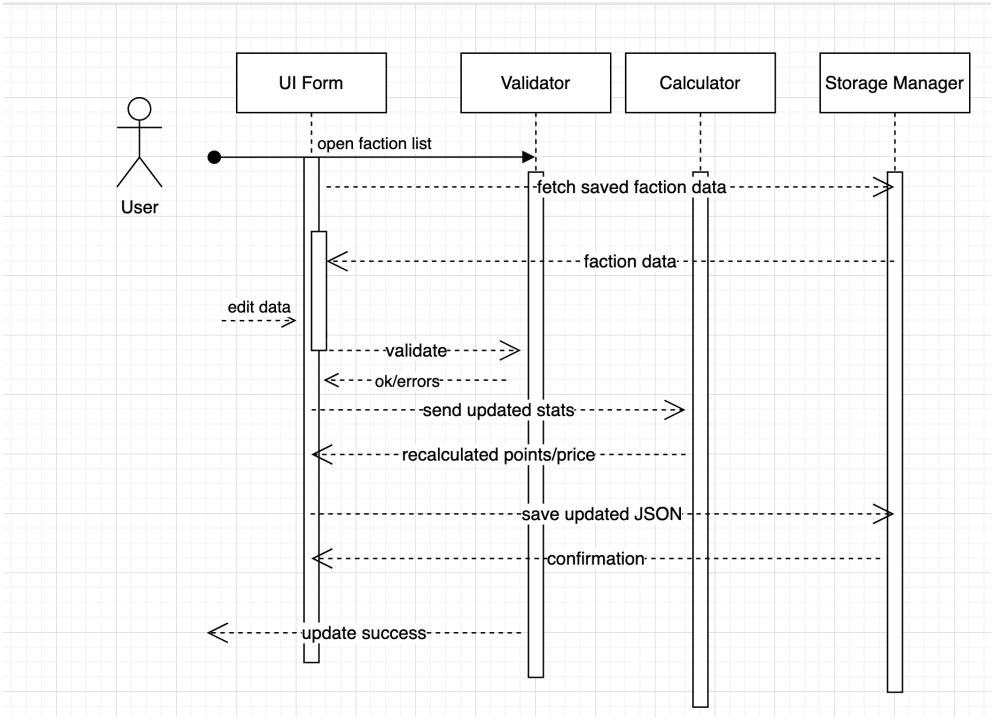
7. Interaction Diagrams

UC1:User Creates a Faction



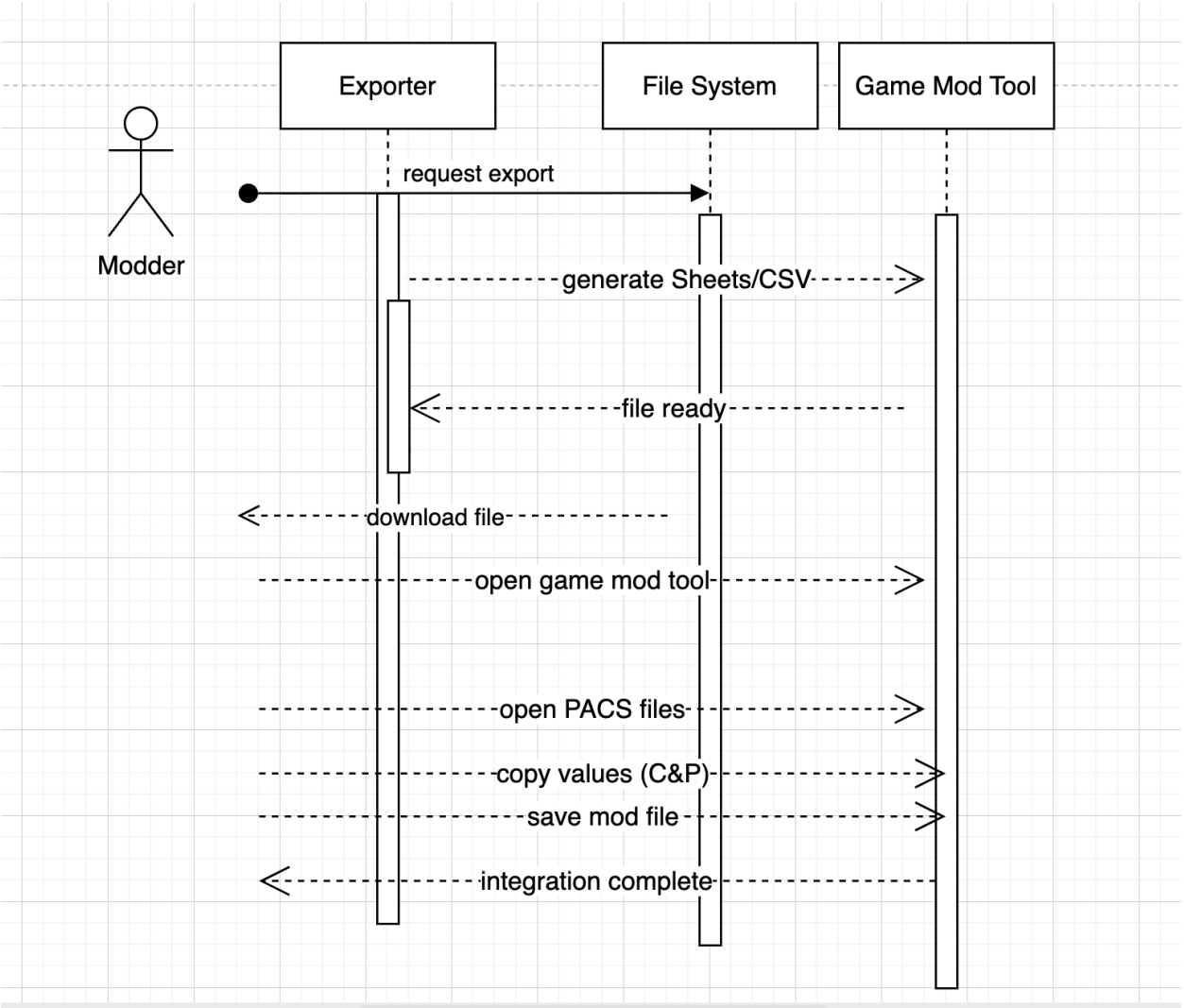
For this sequence, we assigned the responsibility of calculating derived values like points and price to the Calculator, ensuring logic stays centralized and reusable. The faction model is responsible for holding the unit's attributes, such as man count, melee weapon, morale, and other stats. Persistence of the newly created faction is handled by the Storage Manager, which manages saving to the browser in JSON format. This assignment of responsibilities increases cohesion by separating state management from calculation and persistence.

UC2 : User Updates Their Faction



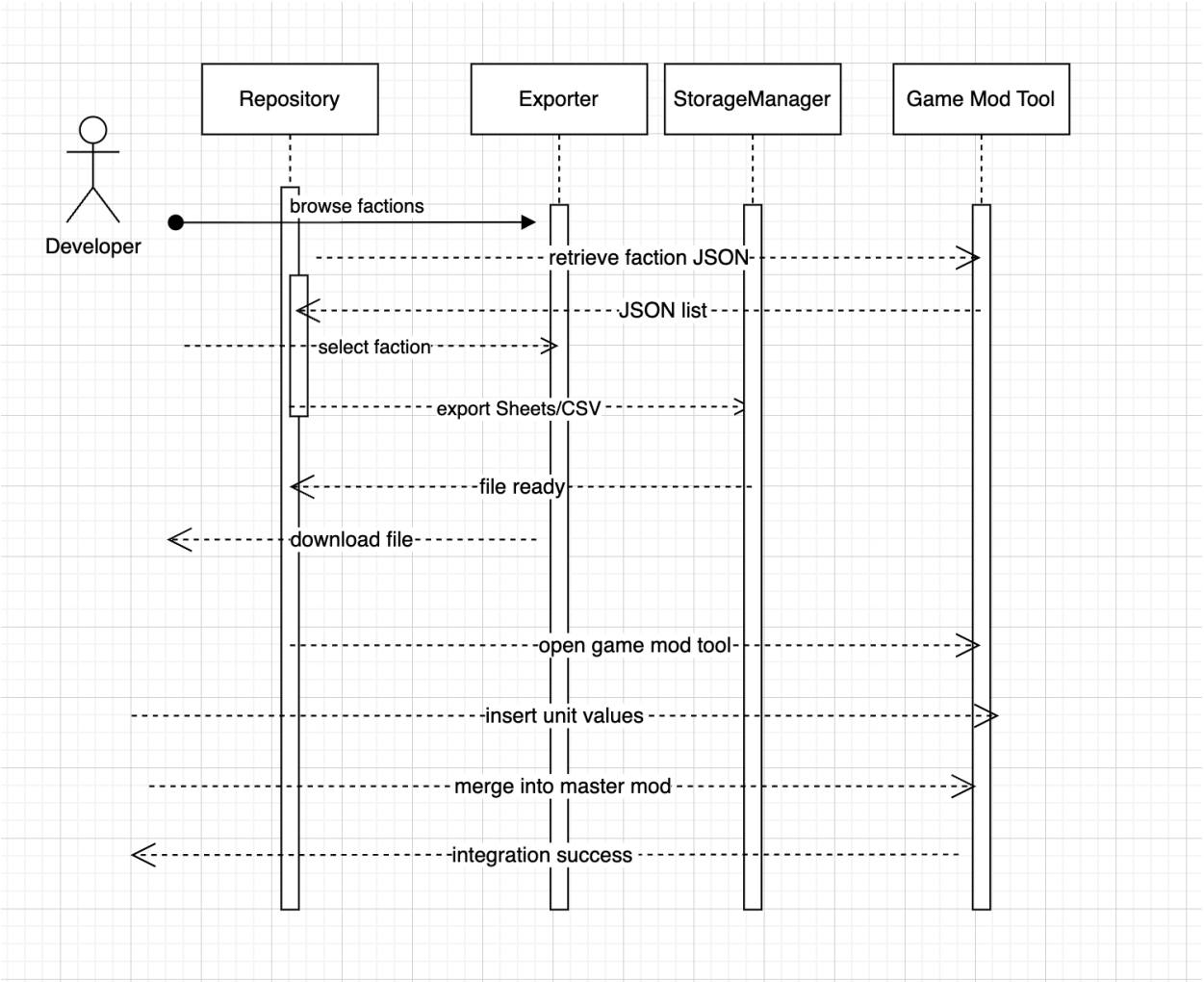
The calculator again takes responsibility for recalculating dependent values when attributes are changed. The faction model ensures the updated state is properly reflected. The StorageManager is responsible for saving the updated faction back to persistent storage, ensuring changes outlive a single session. By keeping recalculation, state, and persistence separated, we maintain high cohesion and low coupling across the system.

UC3: Modder Integrates Units into Game



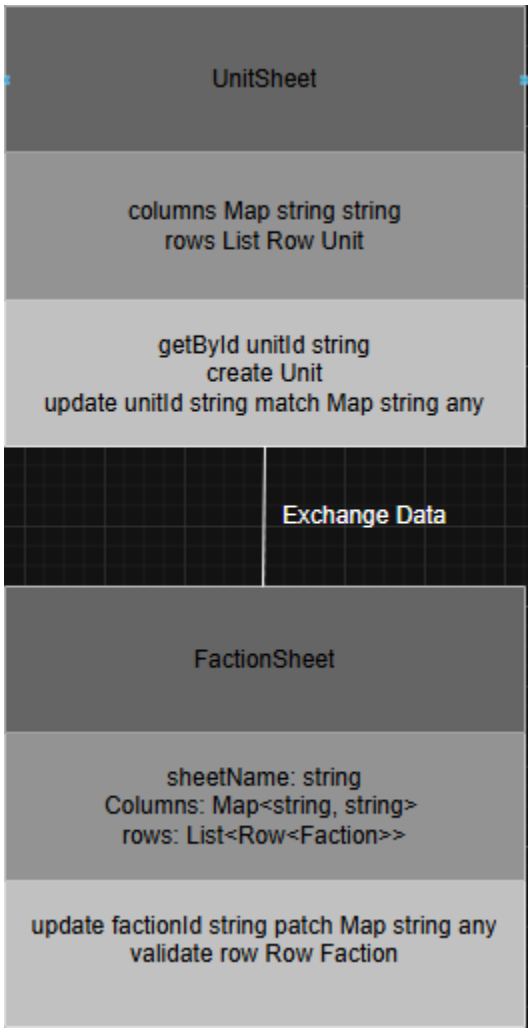
We assigned the responsibility of converting the stored faction data into game-ready files to the Exporter. It retrieves faction data from the StorageManager, formats it into the expected PACS-compatible JSON, and prepares it for integration. The Faction model provides a clean data representation to the Exporter, avoiding duplication of logic. This design choice keeps export-specific logic isolated in the Exporter while still relying on the StorageManager for access to persistent objects.

UC4: Developer Integrating Others' Units into Mod



For this sequence, the StorageManager is responsible for retrieving faction data created by other users. The Faction model ensures that imported units maintain a consistent structure with local ones. The Exporter again handles the conversion to game-compatible files, ensuring cohesion between integration processes. This separation of concerns allows the StorageManager to focus on persistence and retrieval, while the Exporter focuses solely on formatting for the master mod. The design promotes reuse by keeping integration workflows consistent across both user-created and externally sourced factions.

8a. Class Diagram



Class Diagram Explanation:
Our class diagram is quite simple, because our system is event driven, most of our system's utility lies in easing access for users, and this system is as simple as we can get for performance and simplicity. The user interacts with their sheets, and the data gets stored. Our system doesn't have many moving parts, all its complexity lies inside of the class, not the interactions between classes.

8b.Data Types and Operation Signatures

Our system is implemented using javascript, where each spreadsheet tab acts as a persistent "class" storing domain objects. Below are the defined data types, operations, and their plain-language meanings.

Class: FactionSheet
Attributes:

- **sheetName: string -** name of the Google Sheet tab that stores factions.
- **Columns: Map<string, string> -** defines the column structure for the sheet.
- **rows: List<Row<Faction>>** contains all faction entries.

Operations:

- **update factionId string patch Map string any** – updates faction information
- **validate row Row Faction** – checks if faction data is valid

Meaning
Manages all faction data including name, banner, and colors

UnitSheet

Attributes
columns Map string string – defines unit data structure
rows List Row Unit – stores unit entries

Operations
getById unitId string – retrieves a unit by row
create u Unit – modify row of unit data
update unitId string patch Map string any – updates existing unit information

Meaning
Stores data for each unit, including its stats, costs, and faction link.

7c. Traceability Matrix - Domain Concept Objects to Class Objects

Domain Concept	FactionSheet	UnitSheet	StatProfileSheet	BuildSheet	BuildItemSheet	EconomyContextSheet	CalculationService	IntegrationPackageSheet	User
Faction	X								
Unit		X	X						
ArmyBuild				X					
BuildItem					X				
EconomyContext						X			
CalculationResult							X		
ComparisonSet							X		
IntegrationPachage								X	
User									X
Developer/Modder								X	

Explanations of Mappings

- **Faction → FactionSheet:** The faction concept is stored persistently in Google Sheets, where each row represents a faction with its defining metadata (name, description, culture).

- **Unit → UnitSheet, StatProfileSheet:** The *Unit* concept was split into two persistent classes—one for unit identity and one for its updatable combat stats—to support normalization and faster recalculation.

- **ArmyBuild → BuildSheet:** Represents an army composition header for players. This class holds metadata about a player's saved build.

- **BuildItem → BuildItemSheet:** Represents a single entry (unit + quantity) under a BuildSheet, enabling one-to-many relationships for each build.

- **EconomyContext → EconomyContextSheet:** Stores per-player financial parameters (income, upkeep multipliers) that influence balancing formulas.

- **CalculationResult → CalculationService / CalculationResult:** The *CalculationResult* is a transient concept; it is implemented as a *data transfer object (DTO)* returned by *CalculationService* methods.

- **ComparisonSet → CalculationService:** A behavioral construct for ad-hoc comparison of unit stats or costs; handled by *CalculationService* methods rather than a persistent class.

- **IntegrationPackage → IntegrationPackageSheet / IntegrationService:** The metadata is stored in a Sheet class, while the export and validation logic are handled in *IntegrationService*. This split ensures reliability and recoverable integration attempts.

- **User → (implicit):** Users are represented indirectly through their faction and economy records. A dedicated *UserSheet* can be introduced later if authentication is required.

- **Developer/Modder → IntegrationService consumer:** These roles interact with the integration layer to export factions into the Rome II Assembly Kit environment.

9. Algorithms and Data Structures

9a. Algorithms

Our project primarily uses spreadsheet based calculations rather than traditional programming algorithms. However, several internal scripts within Google Apps Script perform automated operations:

- Total Cost and Upkeep Calculation:
  It iterates through each unit selected by the user and multiplies the unit's cost and upkeep by the selected quantity, then sums the totals for the faction build.

- Comparison Metric Calculation:
  Compares units' cost-to-performance ratios (such as cost per attack or cost per armor) to highlight efficient options for the player.

- Data Validation Algorithm:
  Ensures valid numeric entries (e.g., no negative costs or upkeep values) and prevents duplicate unit entries within a faction.

All these algorithms are simple iterative loops running in On time, since they process each selected unit or entry once.

9b. Data Structures

The system uses the following data structures:

- **Sheets as Tables:** Each spreadsheet tab (e.g., Factions, Units, Builds) functions as a structured table, storing persistent data in rows and columns.

- **Arrays:** Used internally by Apps Script when reading or writing data from Sheets (getValue()/setValue()).

- **Objects and Maps:** Used for quick access to units and stats by their ID (e.g., unitMap[id] = {cost, upkeep, stats}) providing near O(1) lookup performance.

- **Sets:** Used to ensure uniqueness of unit identifiers when creating faction builds.

Choice Rationale:
These data structures were chosen for simplicity and compatibility with Google Sheets' ecosystem. Arrays provide batch operations for speed, while Maps and Sets improve lookup efficiency and data validation.

9c. Concurrency

Google Apps Script executes in a **single-threaded environment** per user request, so the system does not employ concurrency in the traditional sense.

However, synchronization is still important for simultaneous user actions. To prevent conflicts:

- **LockService** is used when writing or updating shared data (e.g., saving builds or exporting stats) to ensure only one operation modifies the sheet at a time.

- Each operation runs atomically, meaning once a function starts, it completes before another function begins execution.

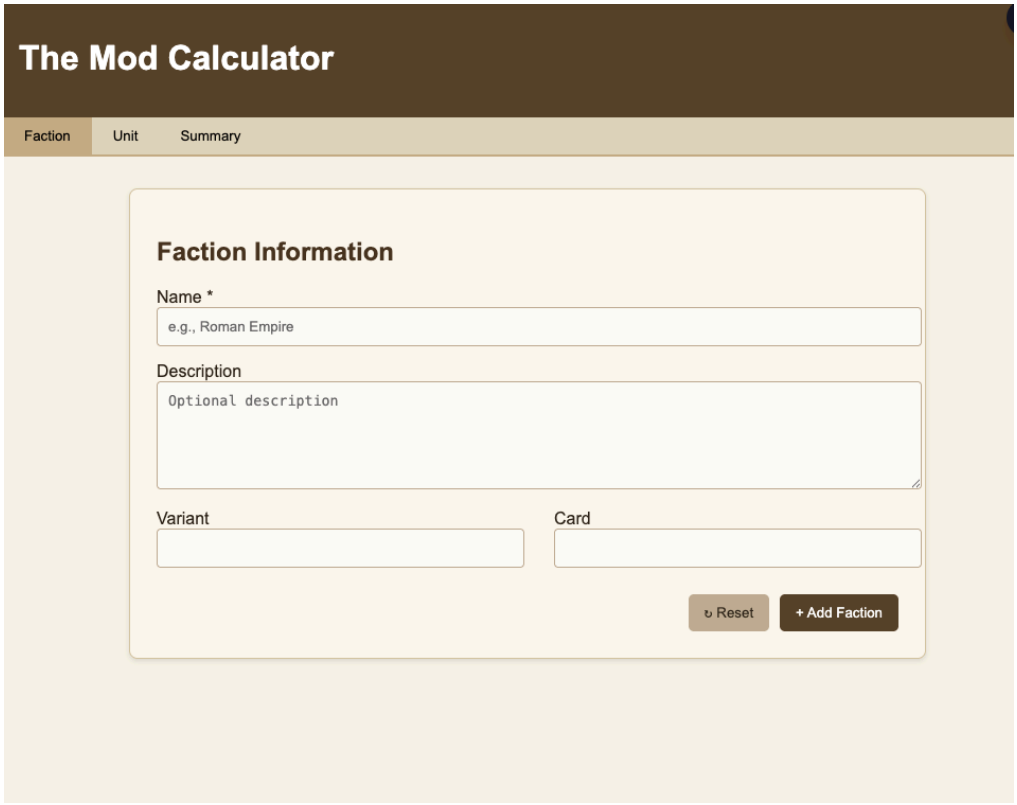
This design guarantees consistent results without introducing race conditions or the need for multi-threaded synchronization logic.

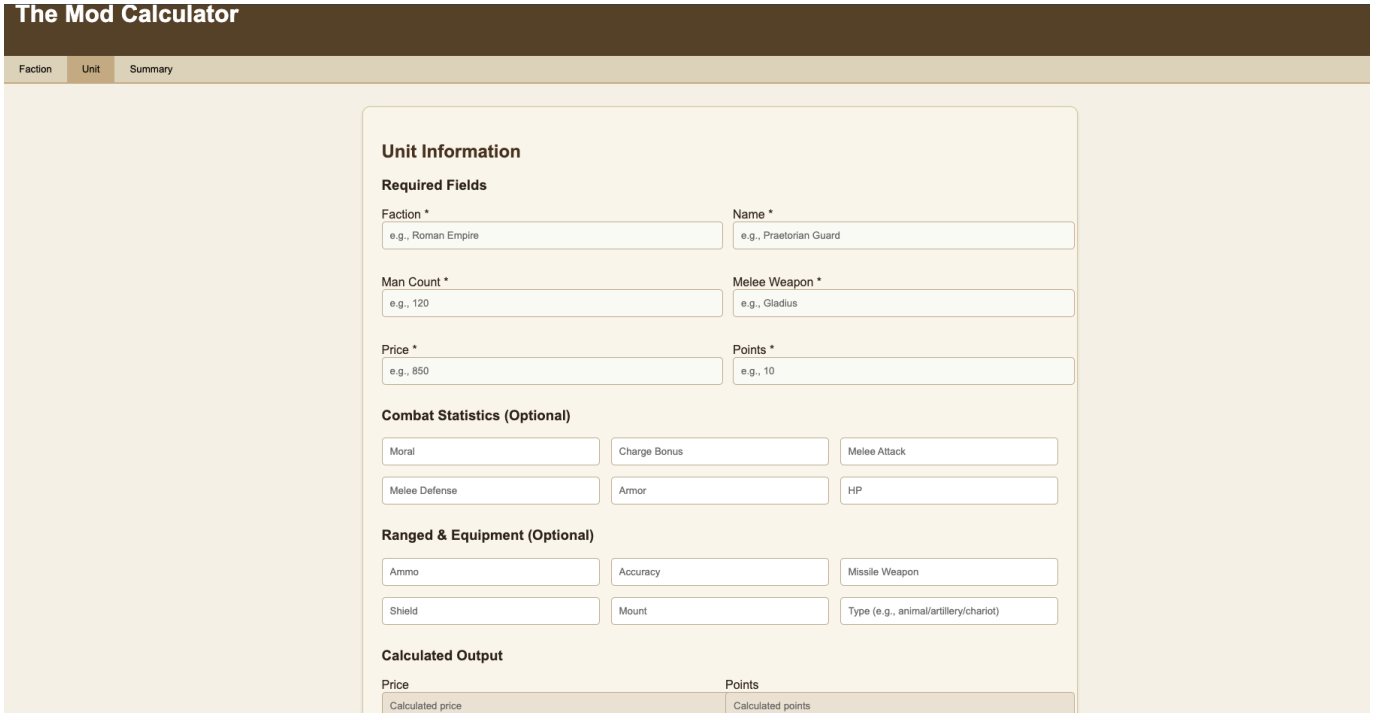
10.) User Interface Design and Implementation(Updated)

We modified our initial Mod Calculator interface after evaluating user interaction complexity and reviewing feedback from our initial testing. The original layout divided the calculator into multiple content-specific tabs like Basic Unit, Combat Stats, Equipment, Attributes, and Results. While this offered detailed segmentation, it was confusing about where certain data should be entered and did not accurately reflect how information was stored in the database.

We merged these sections and created three more meaningful tabs: Faction, Unit, and Summary. This change creates a clearer hierarchy of and mirror hwo data is grouped and related in the backend. Users will now be able to understand that each faction must contain multiple units. The merged sections allow for a more intuitive workflow. Previously, tasks may have required switching between multiple tabs, and now they can be done in a single logical section.

The streamlined layout better communicates the structure of the system while maintaining all necessary input functionality. The updated interface and tab structure can be seen in the redesigned Faction and Unit screens shown below.




The Mod Calculator
Faction
Unit
Summary
Unit Information
Required Fields
Faction *
e.g., Roman Empire
Name *
e.g., Praetorian Guard
Man Count *
e.g., 120
Melee Weapon *
e.g., Gladius
Price *
e.g., 850
Points *
e.g., 10
Combat Statistics (Optional)
Moral
Charge Bonus
Melee Attack
Melee Defense
Armor
HP
Ranged & Equipment (Optional)
Ammo
Accuracy
Missile Weapon
Shield
Mount
Type (e.g., animal/artillery/chariot)
Calculated Output
Price
Points
Calculated price
Calculated points

11. Test Designs
11a. Test Cases

Use Case Tested: UC-1 **Pass/Fail Criteria:** The test will succeed if the tester is able to create a complete faction **Input Data:** A set of data for a faction	
Test Procedure	**Expected Results**
A set of data will be input into the calculator	The calculator outputs the correct results
A json file will be given to the calculator	The calculator fills in the table and outputs the correct results

Use Case Tested: UC-2 **Pass/Fail Criteria:** The test will succeed if the tester is able to update a completed faction **Input Data:** An already created faction	
Test Procedure	**Expected Results**
The tester will attempt to modify the existing faction	The calculator change the given results correctly

Use Case Tested: UC-3 **Pass/Fail Criteria:** The test will succeed if the program puts the given data into the correct format to be copy pasted into the game files. **Input Data:** A set of data for units	
Test Procedure	**Expected Results**
A modder will put the data from the program into the game files	The units show up in the game without any errors

Use Case Tested: UC-4

Pass/Fail Criteria: The test will succeed if the program puts the given data into the correct format to be copy pasted into the game files.

Input Data: A set of data for a faction

Test Procedure	**Expected Results**
A modder will put the data from the program into the game files	The faction shows up in the game without any errors

12b. Test Coverage

We have created a robust set of test cases to confirm the operational integrity of all critical classes within the system. Our validation strategy is twofold: ensuring source code coverage and verifying correct interaction between classes. By prioritizing the expected response of each individual unit, we can more easily pinpoint and debug issues, preventing the accumulation of errors. To facilitate this process, we will utilize a bottom-up integration testing methodology. While our current tests align with the system's core goals, they are designed to be extensible, allowing for future revisions should we proceed with developing stretch features. This approach is foundational to delivering a resilient and trouble-free service.

12c. Integration Testing

Our group has successfully developed a robust and extensive set of test cases designed to comprehensively verify the operational integrity and reliability of all core classes within our system architecture. Our overarching quality assurance strategy places a strong emphasis on achieving high source code coverage, a measure critical for ensuring that every line of essential logic is thoroughly exercised. Beyond individual unit testing, we prioritize the meticulous validation of inter-class communication, confirming that data and control flow correctly between dependent components. Given the inherent dependency structure of our system, ensuring the precise and error-free response of each individual unit before integration is not just beneficial, but absolutely critical to the system's overall stability. This unit-level verification makes it significantly easier to detect the source of problems and quickly isolate any potential issues that require debugging. Furthermore, to strategically mitigate the risk of developing cumulative, difficult-to-track errors that often plague complex software, we are implementing a phased bottom-up integration testing methodology. The design of these individual tests is structured for clarity, ensuring they are easily managed for straightforward pass/fail confirmation. While our current suite fully covers all essential functional features aligned with our initial project goals, the test plans are architected to be readily revisable and extensible. This flexibility allows us to seamlessly support any future stretch goals or enhanced features we may decide to implement, thereby guaranteeing the delivery of a highly resilient, maintainable, and ultimately trouble-free system for end users.

12d. System Testing

We plan to implement a comprehensive System Testing strategy to ensure our final product is robust, reliable, and meets all specified criteria.

Our plan goes beyond basic unit checks to validate the system as a whole across two key areas:

1. Functional Testing: We will test all core algorithms and the User Interface (UI) to ensure they meet their specified requirements. This includes end-to-end scenario testing where we verify the entire workflow, from user input in the UI to the processing of data by the backend algorithms, ensuring correct output is displayed.
2. Non-Functional Testing: We will rigorously test critical non-functional requirements such as performance, security, and stability. This involves stress testing our system's algorithms with boundary conditions and high data loads, as well as checking the system's response to unexpected inputs and resource limitations.

This multi-faceted approach ensures that all components, including the user experience, core logic, and external dependencies, function correctly when integrated, allowing us to deliver a high-quality, dependable product.

Project Management

12a. Merging the Contributions from Individual Team Members

All members contributed substantial portions across documentation, diagrams, and prototype implementation.
The merging process required consolidating multiple document formats (Word, PDF, Google Docs) into a single consistent report. The main challenge was aligning terminology across sections (e.g., “Faction Builder,” “Calculator,” “Integration System”) to maintain conceptual consistency. Formatting differences between exported diagrams and text sections were resolved through unified font and style templates.

Issues Encountered and Resolutions

- **Inconsistent Terminology:** Some early drafts referred to “Builder” vs. “Calculator.” This was standardized to *The Calculator Mod* throughout.
- **Integration of UML Diagrams:** Diagrams were contributed using various tools (Lucidchart, Draw.io). Standardized exports (PNG, same aspect ratio) were used for alignment.
- **Version Control Conflicts:** Edits made offline by multiple members were merged using GitHub’s comparison feature, avoiding overwriting progress.
- **Formatting Loss:** When converting Google Docs to PDF, page breaks shifted; resolved by exporting as PDF directly from Docs with fixed section numbering.

The final merging process was coordinated by **Uong SovanDara**, ensuring all sections followed the report structure, numbering, and style guide set in previous reports.

12b. Project Coordination and Progress Report

Implemented Use Cases:

- **UC1: User Creates Faction** – fully implemented and functional in the W3Schools-hosted prototype. Users can enter unit stats, view real-time balance outputs, and export JSON data to Sheets.

- **UC2: User Updates Faction** – partially functional; update operations work in Google Sheets, but UI confirmation prompts and validation logic are being finalized.

- **UC3: Modder Integrates Units into Game** – backend pipeline in progress. The Apps Script exporter successfully produces PACS-compatible CSV files, awaiting integration tests with Rome II Assembly Kit.

- **UC4: Developer Integrates Factions into Mod** – under development; the GitHub release automation and master mod synchronization pipeline are being designed by Alexander.

Current Work:

- UI improvements and accessibility validation (SovanDara).

- Backend data testing, API synchronization, and formula validation (Oziel).

- System debugging, performance profiling, and flow refinement (Thol).

- Export logic expansion and in-game integration testing (Alexander).

Other Management Activities:

- Weekly coordination via Google Meet and Discord for task updates.

- GitHub issues a tracker for versioning and feedback.

- Integration testing planned on local builds before public demonstration.

12C. Plan of Work:

Through our Calculator Mod, we will give gamers access to a faction builder integrated into Total War: Rome II using widely accessible online technologies and hosting from W3Schools spaces and GitHub. A web application is created, linked to Google Sheets for computations, and scripts are written to include each faction into a common master mod.

Languages: HTML, CSS, JavaScript(frontend), Python(Backend)
Platforms/Tools: GitHub, W3Schools Spaces(hosting), Google Apps Script
Integrations: Google Sheets API, Total War: Rome II Assembly Kit, GitHub repository for version tracking.

Success Criteria:

- The online builder, which is hosted on W3Schools Spaces, allows players to establish factions, and the results are saved in Google Sheets.
- When creating a faction, the balance rules are automatically applied.
- The master mod seamlessly incorporates data from Sheets.
- All community factions are constantly updated in the common master mod.

Road Map:

Phase	Week 1	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7	Week 8	Week 9	Week 10	Week 11	Week 12	Week 13	Week 14
Research														
Design														
Implementation														
Integration														
Testing														
Finalization														

12d. Breakdown of Responsibilities

Team Member	Modules/Classes Responsible	Primary Tasks	Testing Responsibility
Uong SovanDara (Coordinator & Web Lead)	FactionBuilder UI, User Effort Estimation, Accessibility Components	Coordinates team efforts and develops the main website interface on W3Schools Spaces. Ensures consistency and accessibility.	UI testing and front-end to backend integration checks.
Oziel Martinez (Data Integration & Testing Lead)	FactionBuilder UI, Interface Mockups, UI Scripts	Develops UI/UX, manages frontend logic, ensures WCAG compliance, integrates with Sheets API.	UI functionality and user experience testing.
Thol Ucca Kool (Systems Analyst & Debugging Lead)	SystemSequenceManager, ConnectorProtocols, DebugHandler	Oversees system logic flow, ensures inter-module communication stability, and manages debugging logs.	Integration testing, backend performance analysis.
Alexander Redinger (Backend & Mod Integration Lead)	BalanceCalculator, EconomyContextSheet, IntegrationService, Exporter, GitHubSync	Develops export scripts, integrates mod files into PACS, manages version control for releases.	In-game testing with Assembly Kit, final release validation, backend functionality.

Integration Coordination:

- Integration coordination is handled by **Thol Ucca Kool**, who ensures that all front-end, backend, and export modules align during combined testing.

Integration Testing

Integration testing is performed collaboratively by all members:

- **Alexander Redinger** – Backend and in-game validation.
- **Uong SovanDara** – UI-to-backend data checks.
- **Oziel Martinez** – UI usability and responsiveness tests.
- **Thol Ucca Kool** – Oversees coordination and debugging.

References:

1. GitHub, version control and project planning https://docs.github.com/en
2. Creative Assembly. *Total War: Rome II – Assembly Kit (TWR2)*. Available via Total War Wiki. https://wiki.totalwar.com/w/Assembly_Kit_%28TWR2%29.html
3. W3Schools. *HTML, CSS, and JavaScript Tutorials*. https://www.w3schools.com
4. Google. *Google Docs Editors (Docs, Sheets, Slides)*. https://www.google.com/docs/about