Module Guide for The Crazy Four

Team #25, The Crazy Four

Ruida Chen
Ammar Sharbat
Alvin Qian
Jiaming Li

November 17, 2025

1 Revision History

Date	Version	Notes
Nov 6	Alvin, Ammar, Ruida, Jim	Initial draft of basic MG (not in report)
Nov 7	Above & Chris Schankula	Review and feedback of basic MG
Nov 8	Alvin	Module Hierarchy and decomposition
Nov 9	Alvin	Updated API's for game action modules
Nov 13	Alvin	modified module decomposition to remove incorrect content
Nov 13	Ammar	added ACs, AC Traceability, Use Hierarchy, UI, Design of CPs; WebSocket, Timeline

2 Reference Material

This section records information for easy reference.

2.1 Abbreviations and Acronyms

symbol	description
AC	Anticipated Change
ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
CSS	Cascading Style Sheets
DAG	Directed Acyclic Graph
DOM	Document Object Model
FR	Functional Requirement
HTTP	Hypertext Transfer Protocol
JSON	JavaScript Object Notation
JWT	JSON Web Token
M	Module
MG	Module Guide
NFR	Non-Functional Requirement
OS	Operating System
R	Requirement
REST	Representational State Transfer
SC	Scientific Computing
SR	Safety Requirement
SRS	Software Requirements Specification
SQL	Structured Query Language
UC	Unlikely Change
UI	User Interface

Contents

List of Tables

List of Figures

3 Introduction

Decomposing a system into modules is a commonly accepted approach to developing software. A module is a work assignment for a programmer or programming team (Parnas et al., 1984). We advocate a decomposition based on the principle of information hiding (Parnas, 1972). This principle supports design for change, because the "secrets" that each module hides represent likely future changes. Design for change is valuable in SC, where modifications are frequent, especially during initial development as the solution space is explored.

Our design follows the rules layed out by Parnas et al. (1984), as follows:

- System details that are likely to change independently should be the secrets of separate modules.
- Each data structure is implemented in only one module.
- Any other program that requires information stored in a module's data structures must obtain it by calling access programs belonging to that module.

After completing the first stage of the design, the Software Requirements Specification (SRS), the Module Guide (MG) is developed (Parnas et al., 1984). The MG specifies the modular structure of the system and is intended to allow both designers and maintainers to easily identify the parts of the software. The potential readers of this document are as follows:

- New project members: This document can be a guide for a new project member to easily understand the overall structure and quickly find the relevant modules they are searching for.
- Maintainers: The hierarchical structure of the module guide improves the maintainers' understanding when they need to make changes to the system. It is important for a maintainer to update the relevant sections of the document after changes have been made.
- Designers: Once the module guide has been written, it can be used to check for consistency, feasibility, and flexibility. Designers can verify the system in various ways, such as consistency among modules, feasibility of the decomposition, and flexibility of the design.

The rest of the document is organized as follows. Section 4 lists the anticipated and unlikely changes of the software requirements. Section 5 summarizes the module decomposition that was constructed according to the likely changes. Section 6 specifies the connections between the software requirements and the modules. Section 7 gives a detailed description of the modules. Section 8 includes two traceability matrices. One checks the completeness of the design against the requirements provided in the SRS. The other shows the relation between anticipated changes and the modules. Section 9 describes the use relation between modules.

4 Anticipated and Unlikely Changes (completed)

This section lists the project changes that we expect ("Likely Changes") and those we consider unlikely within the capstone timeframe ("Unlikely Changes"). Where applicable each anticipated change (AC) has been given an identifier to support traceability and to guide module secrets.

4.1 Likely Changes (each becomes an AC / module secret)

AC1 — Rule Engine Changes: The validation and behavioural details of the game's rule engine (e.g., how special cards behave, new house rules or alternate base arithmetic rules).
Rationale: Rules are likely to be refined during playtesting and instructor feedback, so they should be isolated in the M16 module.

AC2 — Numeric Base Support (Dozenal and beyond): Support for additional numeral systems (Octal, Hex, etc.), and how scores and hints convert and present values.
Rationale: We intend the product to be extensible to multiple bases; the conversion logic and scoring changes must be encapsulated in M18, M17, and M13.

AC3 — Scoring Formula / Presentation: Changes to how round or match scoring is calculated, aggregated or displayed (e.g., bonus rules, alternate tallying).
Rationale: Scoring may evolve after playtests; keep scoring logic isolated in M17.

AC4 — UI and Interaction Patterns: Layout, hint presentation, animations, accessibility adaptations, and onboarding flow.
Rationale: UI polish and accessibility tweaks are frequent after usability sessions; the M8, M11, and M12 should hide these secrets.

AC5 — Database Schema and Persistence Details: Changes to the persistent schema (profiles, history, telemetry format) and indexes.
Rationale: Data collection needs evolve; the M5 module hides these details.

AC6 — Real-time Messaging Strategy: Room lifecycle, message batching, reconnection rules, and state-resync semantics.
Rationale: Real-time policies affect multiplayer stability; isolate them in M2 and M7.

AC7 — Authentication and Session Policies: Token lifetime, guest-session semantics, password storage scheme, and auth flows.
Rationale: Security/configuration choices can change; these are hidden by M4 and the M1 interface.

4.2 Unlikely Changes (explicitly documented)

These items are not expected to change during the capstone, so they are *not* treated as module secrets and are documented rather than encapsulated in a separate module secret:

- **Target Runtime Platforms:** Modern desktop browsers (Chrome/Firefox/Edge) remain the deployment target for the deliverable.
- **Primary Tech Stack:** React frontend, Node.js backend, PostgreSQL for persistence.
- **Core Gameplay Pattern:** The turn-based matching mechanics (match by suit / match by value / sum-to-12 in Dozenal) are a stable product decision.
- **Non-functional goals baseline:** Performance targets (sub-300 ms updates), basic security expectations (TLS usage for network), and the expectation of GitHub-based CI.

5 Module Hierarchy

This section provides an overview of the module design. Modules are summarized in a hierarchy decomposed by secrets in Table 1. The modules listed below, which are leaves in the hierarchy tree, are the modules that will actually be implemented.

M1: API Module

- Provides stateless HTTP (REST) endpoints for auth and profile management.

M2: Real-time Gateway Module

- Manages stateful WebSocket connections for live gameplay and state syncing.

M3: Matchmaking Module

- Handles game lobby creation, joining, and starting a match.

M4: Authentication Module

- Manages user identity, password hashing, and session token generation.

M5: Repository Module

- Abstracts all database queries (SQL) for creating, reading, updating, and deleting data.

M6: Audit Module

- Logs important server-side events for debugging and security.

M7: Real-time Client Module

- Establishes and maintains the client-side WebSocket connection; sends/receives game events.

M8: Application Shell Module

- The main React component providing global layout, navigation, and state.

M9: Authentication Client Module

- Provides the UI and logic for login/signup forms.

M10: Lobby View Module

- UI component for displaying, creating, and joining game lobbies.

M11: Game Board View Module

- UI component that renders the main game interface (hands, deck, discard pile).

M12: Move Controller Module

- Manages user input (e.g., card clicks) and highlights valid moves.

M13: Scoreboard View Module

- UI component for displaying end-of-round scores in decimal and Dozenal.

M14: Profile View Module

- UI component for displaying user statistics and game history.

M15: Game Engine Module

- Manages the core game state (deck, hands) and turn progression.

M16: Rules Module

- Stateless logic to validate moves (e.g., match suit, rank, or Dozenal sum).

M17: Scoring Module

- Calculates scores at the end of a round.

M18: Base Conversion Module

- Utility to convert numbers between decimal and Dozenal.

M19: Game Actions Module

- Defines types and structure for player actions (play card, draw, declare suit, submit score tally).

M20: Operating System Module

- Represents the server's OS, providing the Node.js runtime environment.

M21: Browser Runtime Module

- Represents the client's web browser, providing the React runtime environment.

M22: Database Module

- Represents the PostgreSQL software that handles physical data storage.

Level 1	Level 2	Level 3 (Leaf Modules)
Hardware-Hiding Module		M20 (Server OS)
		M21 (Client Runtime)
		M22 (PostgreSQL)
Behaviour-Hiding Module	(Core Domain Logic)	M15
		M16
		M17
		M18
		M19
Software Decision Module	Backend (Server)	M1
		M2
		M3
		M4
		M5
		M6
	Frontend (Client)	M7
		M8
		M9
		M10
		M11
		M12
		M13
		M14

Table 1: Module Hierarchy

6 Connection Between Requirements and Design

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in Table 2.

The design of the system is intended to satisfy the requirements developed in the SRS. In this stage, the system is decomposed into modules. The connection between requirements and modules is listed in the Traceability Matrix in Section 8 (Table 2). This decomposition ensures that each Functional Requirement (FR), Non-functional Requirement (NFR), and

Safety Requirement (SR) has a clear owner in the design, facilitating implementation and verification.

For example, core gameplay logic (FR-1 to FR-5) is satisfied by the M15 and M16 modules, while the user-facing presentation (FR-7, FR-9) is handled by frontend modules like M13 and M11. Security and data persistence requirements (FR-10 to FR-17, SR-3, SR-8) are satisfied by the backend's M4 and M5 modules.

7 Module Decomposition

Modules are decomposed according to the principle of "information hiding" proposed by Parnas et al. (1984). The *Secrets* field in a module decomposition is a brief statement of the design decision hidden by the module. The *Services* field specifies *what* the module will do without documenting *how* to do it. For each module, a suggestion for the implementing software is given under the *Implemented By* title.

Only the leaf modules in the hierarchy have to be implemented.

7.1 Hardware Hiding Modules

7.1.1 Operating System Module (M20)

Secrets: Process scheduling, filesystem, Node.js runtime environment, network stack.

Services: Provides the execution environment for the backend server.

Implemented By: OS (such as Linux)

Type of Module: Hardware

7.1.2 Browser Runtime Module (M21)

Secrets: DOM rendering, event loop, TypeScript (React) execution, WebSocket/HTTP client implementation.

Services: Provides the execution environment for the frontend client.

Implemented By: Browser (Chrome, Firefox, Edge)

Type of Module: Hardware

7.1.3 Database Module (M22)

Secrets: Data storage on disk, indexing, transaction (ACID) implementation, SQL query optimization.

Services: Provides persistent storage for user and game data.

Implemented By: PostgreSQL

Type of Module: Hardware

7.2 Behaviour-Hiding Module

7.2.1 Game Engine Module (M15)

Secrets: Internal representations of the cards, deck, discard pile, players, and the turn-management state machine.

Services: Creates new matches, enforces turn order, applies validated moves, and determines when a round or match is finished.

Implemented By: The Crazy Four (TypeScript)

Type of Module: Abstract Object

7.2.2 Rules Module (M16)

Secrets: Exact move-validation criteria, including how matching ranks, suits, dozenal sums, and special cards are handled.

Services: Confirms whether a proposed move is legal and enumerates valid moves for a player based on the current game situation.

Implemented By: The Crazy Four (TypeScript)

Type of Module: Abstract Object

7.2.3 Scoring Module (M17)

Secrets: The scoring equation that converts remaining cards into round points and aggregates them over a match.

Services: Produces the score summary for each player whenever a round ends.

Implemented By: The Crazy Four (TypeScript)

Type of Module: Abstract Object

7.2.4 Base Conversion Module (M18)

Secrets: The mapping of digits and symbols used to move between decimal and dozenal numbers.

Services: Translates numeric values to and from dozenal form for scoring logic and UI presentation.

Implemented By: The Crazy Four (TypeScript)

Type of Module: Abstract Data Type

7.2.5 Game Actions Module (M19)

Secrets: The canonical data structures and serialization format that represent every move a player can make.

Services: Defines and validates the action payloads that flow between client and server, keeping both sides in sync on network contracts.

Implemented By: The Crazy Four (TypeScript shared library)

Type of Module: Abstract Data Type

7.3 Software Decision Module - Backend

7.3.1 API Module (M1)

Secrets: REST endpoint structure, payload schemas, and HTTP conventions for every backend capability.

Services: Exposes stateless HTTP routes for authentication, profile management, and bootstrapping new games as defined in the SRS.

Implemented By: The Crazy Four (Node.js, Express)

Type of Module: Abstract Object

7.3.2 Real-time Gateway Module (M2)

Secrets: Real-time messaging strategy, room management, and conflict-resolution logic for server-authoritative play.

Services: Hosts WebSocket connections, validates incoming moves, and pushes synchronized game state updates to every participant.

Implemented By: The Crazy Four (Node.js, Socket.io)

Type of Module: Abstract Object

7.3.3 Matchmaking Module (M3)

Secrets: The pairing heuristics, lobby data structures, and invitation policies for assembling tables.

Services: Creates, lists, and manages lobbies so players can host private games or enter matchmaking queues.

Implemented By: The Crazy Four (Node.js)

Type of Module: Abstract Object

7.3.4 Authentication Module (M4)

Secrets: Password hashing configuration, credential storage details, and token-signing keys.

Services: Creates accounts, validates logins, manages guest sessions, and issues/verifies tokens used by the rest of the backend.

Implemented By: The Crazy Four (Node.js)

Type of Module: Abstract Object

7.3.5 Repository Module (M5)

Secrets: Schema design, optimized SQL queries, and database access strategies.

Services: Offers a clean persistence interface for storing players, credentials, match history, and statistics while shielding callers from database details.

Implemented By: The Crazy Four (Node.js, node-postgres)

Type of Module: Abstract Data Type

7.3.6 Audit Module (M6)

Secrets: The exact event schema, retention policy, and storage targets for operational logs.

Services: Captures authentication, gameplay, and system events to support debugging, compliance, and user inquiries.

Implemented By: The Crazy Four (Node.js, Winston)

Type of Module: Abstract Object

7.4 Software Decision Module - Frontend

7.4.1 Real-time Client Module (M7)

Secrets: Connection lifecycle logic, buffering strategy, and reconnection heuristics for the browser client.

Services: Establishes WebSocket links to the M2, relays user actions, and applies server updates to the local UI state.

Implemented By: The Crazy Four (TypeScript, Socket.io-client)

Type of Module: Abstract Object

7.4.2 Application Shell Module (M8)

Secrets: App-wide navigation plan, shared layout primitives, and global state wiring (theme, auth awareness).

Services: Hosts the consistent chrome of the site and orchestrates routing between major views.

Implemented By: The Crazy Four (React)

Type of Module: Abstract Object

7.4.3 Authentication Client Module (M9)

Secrets: Decisions about secure token storage and the flows for refreshing or clearing credentials in the browser.

Services: Presents login, signup, and logout experiences while coordinating with the M1 for authentication calls.

Implemented By: The Crazy Four (React)

Type of Module: Abstract Object

7.4.4 Lobby View Module (M10)

Secrets: Layout, styling, and interaction patterns for discovering or hosting lobbies.

Services: Shows available rooms, lets players create or join sessions, and triggers matchmaking calls via M1 and M7.

Implemented By: The Crazy Four (React)

Type of Module: Abstract Object

7.4.5 Game Board View Module (M11)

Secrets: Visual composition of the board, card animations, and responsive behavior across devices.

Services: Draws the playable surface, displays player hands and discard piles, and highlights valid actions per the rule engine.

Implemented By: The Crazy Four (React)

Type of Module: Abstract Object

7.4.6 Move Controller Module (M12)

Secrets: Gesture-handling patterns and UX rules for how players select cards or declare suits.

Services: Interprets user intent on the board, performs light validation, and forwards structured actions through the M7.

Implemented By: The Crazy Four (React hooks and event handlers)

Type of Module: Abstract Object

7.4.7 Scoreboard View Module (M13)

Secrets: Presentation choices for multi-base score displays and animations for round summaries.

Services: Shows standings after each round, presenting both decimal and dozenal scores in a clear, accessible format.

Implemented By: The Crazy Four (React)

Type of Module: Abstract Object

7.4.8 Profile View Module (M14)

Secrets: Layout decisions for profile cards, statistics summaries, and sensitive account actions.

Services: Displays player history, stats, and account-management controls, including export or deletion requests tied to FR-15..17.

Implemented By: The Crazy Four (React)

Type of Module: Abstract Object

8 Traceability Matrix

This section shows two traceability matrices: between the modules and the requirements and between the modules and the anticipated changes.

Table 2: Trace Between Requirements and Modules (TblRT)

Requirement (FR/NFR/SR)	**Primary Modules**
FR-1 Start new game	M1, M3, M15, M5
FR-2 Turn management	M15, M16, M19, M2, M7, M12, M11
FR-3 Rule validation	M16, M15, M19, M12, M11
FR-4 Special cards	M16, M15, M19, M12, M11
FR-5 End of game	M15, M17, M13, M5
FR-6 Calculate score	M17, M18, M19, M13
FR-7 Display score	M13, M18
FR-9 Highlight valid moves	M12, M11, M16
FR-10 Account creation	M1, M4, M5, M9
FR-11 Login or Logout	M1, M4, M5, M9
FR-12 Guest mode	M1, M4, M9
FR-13 Credential validation	M4, M1, M5
FR-14 Data storage	M5, M1, M6
FR-15 Data retrieval	M5, M1, M14
FR-16 Data update	M5, M1, M14
FR-17 Data deletion	M5, M1, M14
NFR (Performance)	M2, M15, M7, M11
NFR (Usability)	M11, M8
NFR (Robustness)	M7, M2, M5
NFR (Maintainability)	M16, M17, M1, M6
SR-1 (Dozenal validation)	M16, M18, M15
SR-2 (UI feedback)	M11, M8
SR-3 (Data persistence)	M5, M1
SR-4 (Accurate scoring)	M17, M18
SR-5 (Session recovery)	M7, M2, M4
SR-7 (Encrypted transmit)	(Hardware-Hiding: TLS Layer), M1, M2
SR-8 (Secure storage)	M5, M4
SR-10 (Input validation)	M1, M2, M12

Table 3: Trace Between Anticipated Changes and Modules (TblACT)

Anticipate Change	**Modules**
AC1 Rule Engine Changes	M16, M15
AC2 Numeric Base Support	M18, M17, M16, M13
AC3 Scoring Formula / Presentation	M17, M13
AC4 UI / Interaction Patterns	M8, M11, M12, M13
AC5 DB Schema	M5, M1, M14
AC6 Real-time Messaging	M2, M7, M15
AC7 Authentication Policies	M4, M9, M1

Notes: Each AC above corresponds to a “secret” that should be implemented and documented within the named module(s). When an AC is realized, the owning module’s MIS (or submodule docs) should be updated so that the secret is well described and the code-to-docs trace remains intact.

9 Use Hierarchy Between Modules (Uses Relation)

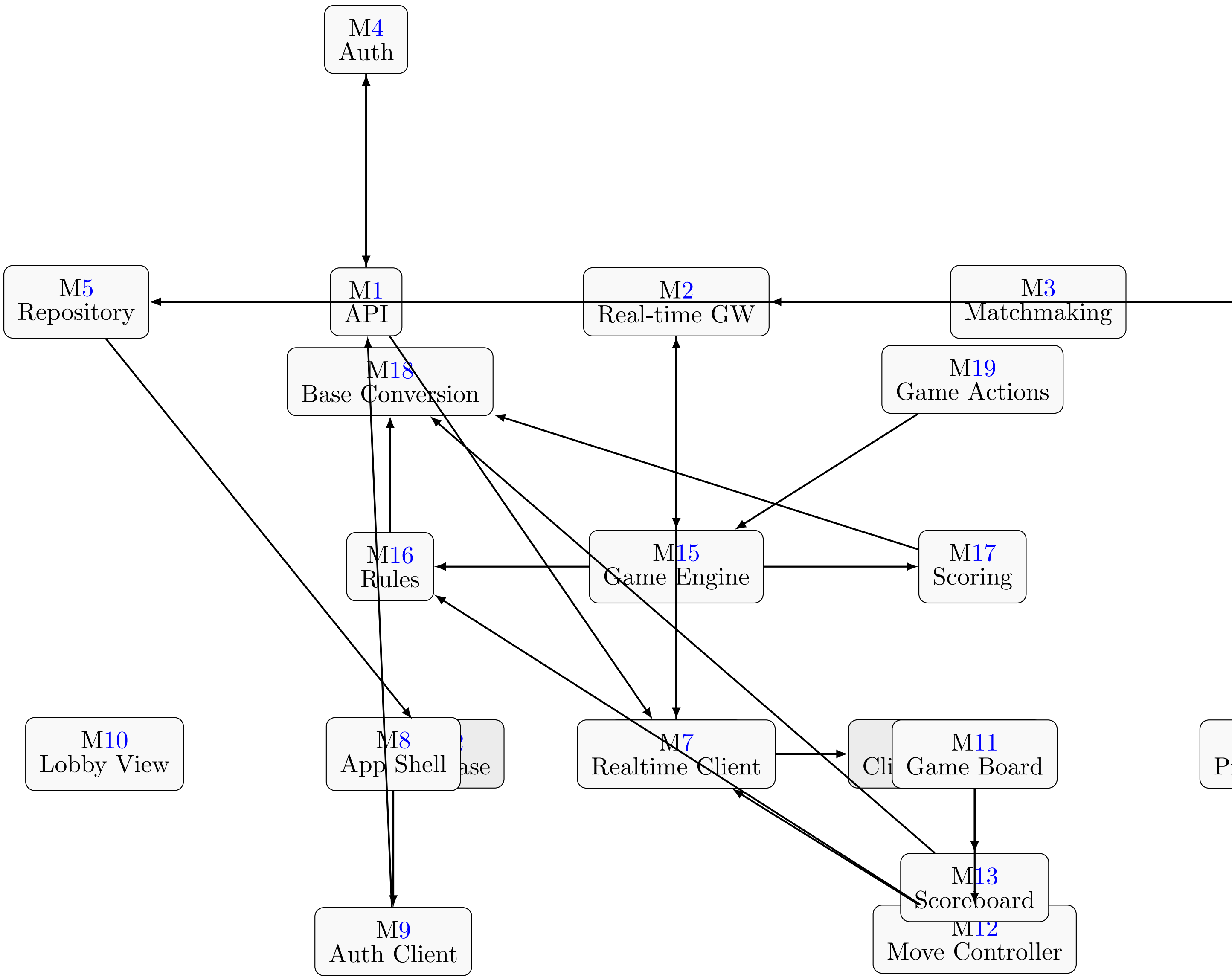
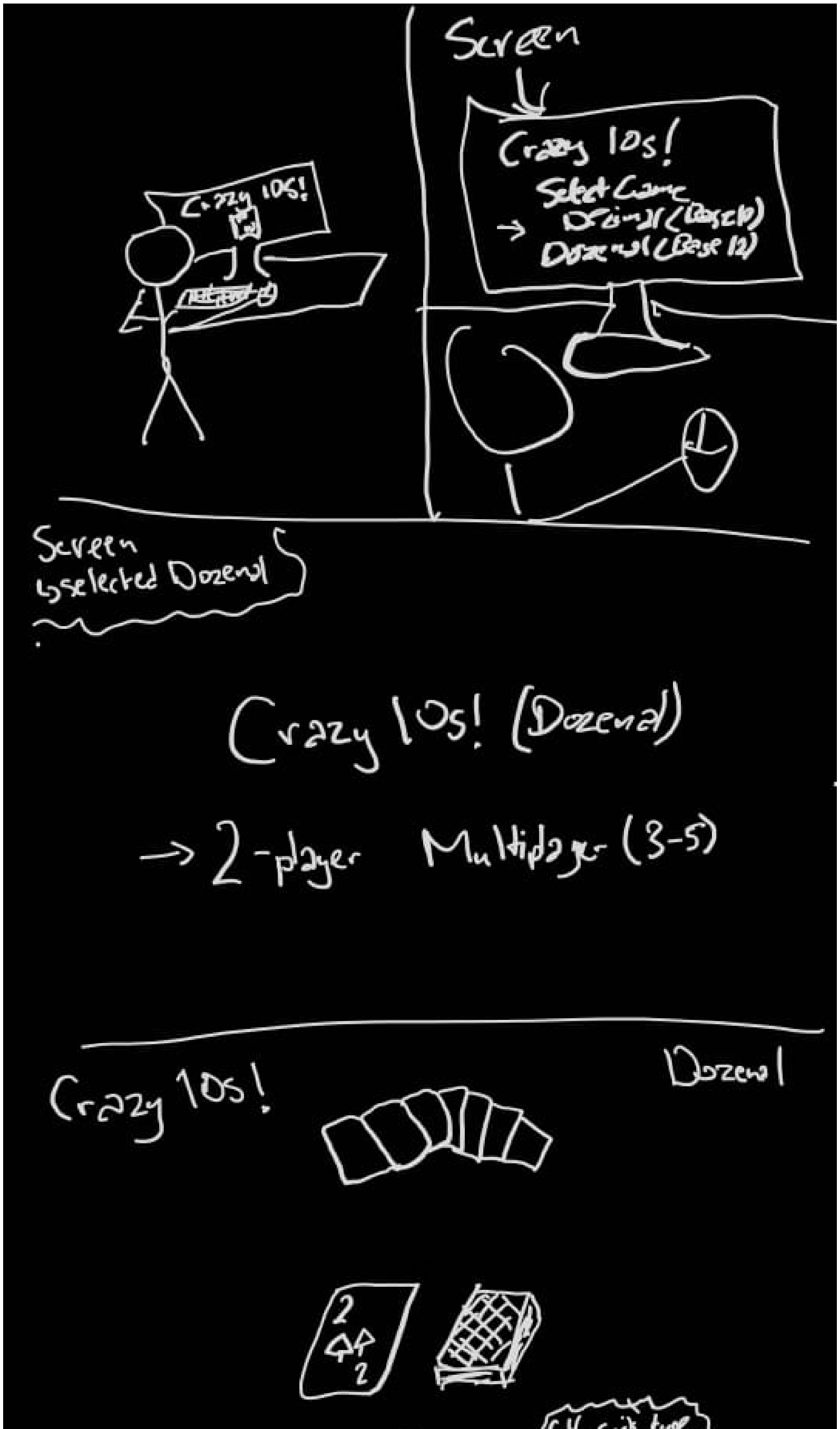


Figure 1: Use hierarchy (DAG) showing primary "uses" relations between modules.

Design rationale: The DAG above shows the server-authoritative pattern: Realtime gateway owns live interactions and forwards validated moves to the Game Engine; the Game Engine uses the Rules and Scoring modules, and persistence is abstracted via the Repository to the Database. Frontend views rely on the Realtime client for live updates and the API for non-realtime operations (profile, auth, lobby listing).

10 User Interfaces (UI) — sketches and wireframes



11 Design of Communication Protocols

This section captures the protocol-level design for server-client communication, with a focus on login/auth flows (REST) and gameplay flows (WebSocket). The approach favours a server-authoritative model: the server is the source of truth for game state.

11.1 Authentication and Profile (REST)

Endpoints (HTTP/JSON)

- `POST /api/auth/signup` { username, password } → 201 / error
- `POST /api/auth/login` { username, password } → { token, user }
- `POST /api/auth/guest` → { token, guestId }
- `GET /api/profile/:userId` Authorization: Bearer token → profile
- `PUT /api/profile/:userId` Authorization: Bearer token → updated profile

Auth notes: JWT-like Bearer tokens are issued on login and short-lived (configurable). Refresh tokens are out-of-scope for MVP but supported in design. Tokens are stored in secure HTTP-only cookies or in-memory (client decision in M9).

11.2 Real-time Gameplay (WebSocket / Socket.IO style)

Connection lifecycle

1. Client connects to the Realtime Gateway with an Authorization header (Bearer token).
2. Server validates token, optionally checks profile/permissions, and assigns the socket to the user's session.
3. Client requests to join or create a lobby room via `joinRoom / createRoom`.

Message schema (JSON) — examples

- **Client → Server:** `{ type: "playCard", payload: { sessionId, action: { kind:"play", card:{suit,rank}, meta:{}} } }`
- **Client → Server:** `{ type: "drawCard", payload: { sessionId } }`
- **Server → Client:** `{ type: "stateUpdate", payload: { sessionId, state: { hands, discardTop, turn, scores} } }`

Protocol properties and constraints

- **Authoritative server:** All move validation occurs on the server (M16/M15). Clients may do local pre-checks for responsiveness, but final acceptance comes from the server.
- **Idempotency:** Actions include client-generated request IDs so duplicate deliveries during reconnection are ignored.
- **Delta updates:** For performance, `stateUpdate` messages typically contain deltas; full-state snapshots are reserved for joins/reconnects.
- **Resync strategy:** On reconnect, the client requests a `reconnectSnapshot`. The server supplies a full state and the client applies it atomically to avoid inconsistent UI states.
- **Security:** All WebSocket connections are over WSS (TLS). Authorization is checked per message where required.

11.3 Sequence Example: Play Card flow

1. Client sends `playCard` with action payload.
2. Realtime Gateway authenticates and forwards the action to Game Engine worker.
3. Game Engine validates via Rules module:
    - If accepted: update state, persist via Repository, emit `actionResult(ok:true)`, then emit `stateUpdate` to all room clients.

11.4 Failure modes and mitigation

- **Dropped messages / short disconnects:** idempotent request IDs plus reconnection snapshot mitigate lost messages.
- **Lag / delayed updates:** server sends small heartbeats and uses sequence numbers so clients can detect missing updates.
- **Untrusted clients:** server always re-checks any client-supplied action against the Rules module — never trust client state for authoritative decisions.

12 Timeline

References

David L. Parnas. On the criteria to be used in decomposing systems into modules. *Comm. ACM*, 15(2):1053–1058, December 1972.

D.L. Parnas, P.C. Clement, and D. M. Weiss. The modular structure of complex systems. In *International Conference on Software Engineering*, pages 408–419, 1984.