# Module Guide for The Crazy Four

Team #25, The Crazy Four

Ruida Chen
Ammar Sharbat
Alvin Qian
Jiaming Li

November 10, 2025

## 1 Revision History

| **Date** | **Version** | **Notes** |
|---|---|---|
| Nov 8 | Alvin | Module Hierarchy and decomposition |

# 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

This section lists possible changes to the system. According to the likeliness of the change, the possible changes are classified into two categories. Anticipated changes are listed in Section 4.1, and unlikely changes are listed in Section 4.2.

## 4.1 Anticipated Changes

Anticipated changes are the source of the information that is to be hidden inside the modules. Ideally, changing one of the anticipated changes will only require changing the one module that hides the associated decision. The approach adapted here is called design for change.

**AC1:** The specific hardware on which the software is running.

**AC2:** The format of the initial input data.

...

[Anticipated changes relate to changes that would be made in requirements, design or implementation choices. They are not related to changes that are made at run-time, like the values of parameters. —SS]

## 4.2 Unlikely Changes

The module design should be as general as possible. However, a general system is more complex. Sometimes this complexity is not necessary. Fixing some design decisions at the system architecture stage can simplify the software design. If these decision should later need to be changed, then many parts of the design will potentially need to be modified. Hence, it is not intended that these decisions will be changed.

**UC1:** Input/Output devices (Input: File and/or Keyboard, Output: File, Memory, and/or Screen).

...

# 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)

#### 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)

#### 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

## 7.2 Behaviour-Hiding Module

### 7.2.1 Game Engine Module (M15)

**Secrets:** The data structure representing the game state (players, deck, discard pile, turn). The state machine for turn management.

**Services:** Initializes a new game, applies a validated move to the game state, manages turn progression, detects end-of-game conditions.

**Implemented By:** The Crazy Four (TypeScript)

**State Variables:** ‘gameState‘: { ‘players‘: [ ], ‘deck‘: Card[ ], ‘discardPile‘: Card[ ], ‘currentTurn‘: PlayerID, ‘status‘: string }

**Access Programs:** ‘dealOpeningHands(players, options)‘ → ‘GameState‘; ‘playCard(gameState, playerID, cardID)‘ → ‘GameState‘; ‘drawCard(gameState, playerID)‘ → ‘GameState‘; ‘applyDeclaredSuit(gameState, playerID, suit)‘ → ‘GameState‘; ‘finalizeRound(gameState)‘ → ‘ScoreSummary‘

**Exceptions:** ‘InvalidGameStateError‘

### 7.2.2 Rules Module (M16)

**Secrets:** The specific logic for move validation: same suit, same rank, or sum equals 12 (base-12). The logic for special cards (e.g., ’10’ is wild).

**Services:** Validates if a move is legal given the current game state. Lists all legal moves for a player.

**Implemented By:** The Crazy Four (TypeScript)

**State Variables:** None (pure functions).

**Access Programs:** ‘isLegalMove(move, gameState)‘ → ‘bool‘; ‘getValidMoves(playerHand, gameState)‘ → ‘Card[]‘

**Exceptions:** ‘InvalidRuleConfig‘

### 7.2.3 Scoring Module (M17)

**Secrets:** The algorithm for calculating round scores based on opponents’ remaining cards.

**Services:** Calculates the score for a completed round.

**Implemented By:** The Crazy Four (TypeScript)

**State Variables:** None (pure functions).

**Access Programs:** ‘calculateRoundScore(winningPlayerID, gameState)‘ → ‘ScoreObject‘

**Exceptions:** None

### 7.2.4 Base Conversion Module (M18)

**Secrets:** The algorithm and symbols (e.g., ↊, ↋) for converting between base-10 (Decimal) and base-12 (Dozenal).

**Services:** Converts a decimal number to a Dozenal string. Converts a Dozenal string to a decimal number. Used for scoring and UI display.

**Implemented By:** The Crazy Four (TypeScript)

**State Variables:** None (pure functions).

**Access Programs:** ‘toDozenal(decimalValue)‘ → ‘string‘; ‘toDecimal(dozenalString)‘ → ‘number‘

**Exceptions:** ‘InvalidBaseString‘

### 7.2.5 Game Actions Module (M19)

**Secrets:** The data structure and serialization format for all player actions sent between client and server.

**Services:** Provides type-safe action constructors, validation, and JSON serialization/deserialization for network transmission.

**Implemented By:** The Crazy Four (TypeScript - shared code between frontend and backend)

**State Variables:** None (stateless utility module).

**Access Programs:**

- `createPlayCardAction(playerID, cardID)` → `PlayCardAction`
- `createDrawCardAction(playerID)` → `DrawCardAction`
- `createDeclareSuitAction(playerID, suit)` → `DeclareSuitAction`
- `createScoreTallyAction(playerID, remainingCards)` → `ScoreTallyAction`
- `validateAction(action)` → `boolean`
- `serializeAction(action)` → `JSON`
- `deserializeAction(json)` → `GameAction`

**Exceptions:** `InvalidCard`, `InvalidSuit`, `SerializationError`, `DeserializationError`

## 7.3 Software Decision Module - Backend

### 7.3.1 API Module (M1)

**Secrets:** REST API route definitions, request/response formats (JSON), HTTP status codes.

**Services:** Provides stateless HTTP endpoints for user authentication (FR-10..13), profile management (FR-15..17), and game creation.

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

**State Variables:** None (stateless).

**Access Programs:** ‘POST /api/auth/signup‘, ‘POST /api/auth/login‘, ‘GET /api/profile‘, ‘PUT /api/profile‘, ‘DELETE /api/profile‘

**Exceptions:** ‘AuthError‘ (401/403), ‘ValidationError‘ (400), ‘NotFound‘ (404)

### 7.3.2 Real-time Gateway Module (M2)

**Secrets:** WebSocket protocol, message serialization, room/session management. Server-authoritative state synchronization logic.

**Services:** Manages active game sessions, broadcasts game state updates to clients, receives and validates moves from clients in real-time.

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

**State Variables:** ‘activeGames‘: Map¡‘GameID‘, ‘GameSession‘¿

**Access Programs:** ‘on(’connection’, socket)‘; ‘on(’playCard’, action)‘; ‘on(’drawCard’, action)‘; ‘on(’declareSuit’, action)‘; ‘on(’submitScoreTally’, action)‘; ‘emit(’gameStateUpdate’, state)‘; ‘emit(’scoreTallyConfirmed’, summary)‘

**Exceptions:** ‘InvalidMove‘, ‘NotYourTurn‘, ‘SessionError‘

### 7.3.3 Matchmaking Module (M3)

**Secrets:** Logic for pairing players, managing game lobbies, and handling invites.

**Services:** Allows users to create, join, or be matched into a game session.

**Implemented By:** The Crazy Four (Node.js)

**State Variables:** ‘lobbies‘: Map¡‘LobbyID‘, ‘Lobby‘¿

**Access Programs:** ‘createLobby(userID)‘ → ‘LobbyID‘; ‘joinLobby(lobbyID, userID)‘ → ‘void‘

**Exceptions:** ‘LobbyFull‘, ‘LobbyNotFound‘

### 7.3.4 Authentication Module (M4)

**Secrets:** Password hashing algorithm (e.g., bcrypt), salt generation, JWT/session token structure and secret key.

**Services:** Handles user account creation, validates credentials, manages guest sessions, issues and verifies session tokens.

**Implemented By:** The Crazy Four (Node.js)

**State Variables:** None.

**Access Programs:** ‘registerUser(username, password)‘ → ‘User‘; ‘loginUser(username, password)‘ → ‘Token‘; ‘verifyToken(token)‘ → ‘UserID‘

**Exceptions:** ‘InvalidCredentials‘, ‘UserExists‘, ‘TokenExpired‘

### 7.3.5 Repository Module (M5)

**Secrets:** Database schema (tables, columns, relations), SQL queries, connection pooling.

**Services:** Provides an interface for all data persistence. Stores and retrieves user data (FR-14, FR-15), game history, and scores. Handles updates (FR-16) and deletions (FR-17).

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

**State Variables:** Database connection pool.

**Access Programs:** ‘findUserByUsername(username)‘, ‘createUser(data)‘, ‘saveGameResult(result)‘, ‘getUserProfile(userID)‘, ‘updateUserProfile(userID, data)‘, ‘deleteUser(userID)‘

**Exceptions:** ‘DatabaseConnectionError‘, ‘RecordNotFound‘, ‘UniqueConstraintViolation‘

### 7.3.6 Audit Module (M6)

**Secrets:** Log format, log storage location, retention policy.

**Services:** Logs important system events (e.g., login, game end, errors) for debugging and security auditing.

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

**State Variables:** Logger instance.

**Access Programs:** ‘log.info(message)‘, ‘log.warn(message)‘, ‘log.error(message)‘

**Exceptions:** ‘LogWriteError‘

## 7.4 Software Decision Module - Frontend

### 7.4.1 Real-time Client Module (M7)

**Secrets:** WebSocket connection state, reconnection logic.

**Services:** Connects to the M2, sends user moves, and receives game state updates, applying them to the client-side state.

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

**State Variables:** ‘socket‘: Socket, ‘isConnected‘: bool

**Access Programs:** ‘connect()‘; ‘disconnect()‘; ‘emit(’playCard’, action)‘; ‘emit(’drawCard’)‘; ‘emit(’declareSuit’, action)‘; ‘emit(’submitScoreTally’, action)‘; ‘on(’gameStateUpdate’, callback)‘; ‘on(’scoreTallyConfirmed’, callback)‘

**Exceptions:** ‘ConnectionFailed‘

#### 7.4.2 Application Shell Module (M8)

**Secrets:** Application routing table, global layout structure (header, footer), global state management (e.g., auth status).

**Services:** Renders the main application layout, controls page navigation, and manages global UI state.

**Implemented By:** The Crazy Four (React)

**State Variables:** ‘currentUser‘, ‘currentRoute‘

**Access Programs:** (Rendered React component)

**Exceptions:** ‘RouteNotFound‘

#### 7.4.3 Authentication Client Module (M9)

**Secrets:** How auth tokens are stored (e.g., localStorage, httpOnly cookie).

**Services:** Provides UI components (login/signup forms) and client-side logic for authentication. Communicates with M1.

**Implemented By:** The Crazy Four (React)

**State Variables:** ‘username‘, ‘password‘, ‘isLoading‘: bool, ‘error‘: string

**Access Programs:** ‘handleLogin()‘, ‘handleSignup()‘, ‘handleLogout()‘

**Exceptions:** ‘AuthUIError‘

#### 7.4.4 Lobby View Module (M10)

**Secrets:** UI for listing, creating, and joining games.

**Services:** Renders the game lobby, interacts with M1 and M7 to manage matchmaking.

**Implemented By:** The Crazy Four (React)

**State Variables:** ‘lobbiesList‘: [ ], ‘selectedLobby‘

**Access Programs:** ‘handleCreateGame()‘, ‘handleJoinGame(lobbyID)‘

**Exceptions:** None

#### 7.4.5 Game Board View Module (M11)

**Secrets:** The DOM/CSS for rendering cards, hands, and the board. Animation logic.

**Services:** Renders the main game interface (player hand, discard pile, deck). Visually highlights valid moves (FR-9).

**Implemented By:** The Crazy Four (React)

**State Variables:** ‘clientGameState‘, ‘validMoves‘: [ ]

**Access Programs:** (Rendered React component based on props)

**Exceptions:** None

#### 7.4.6 Move Controller Module (M12)

**Secrets:** Client-side input handling logic (click, drag-and-drop).

**Services:** Captures user input (e.g., clicking a card), performs client-side pre-validation, and submits the move to the M7.

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

**State Variables:** ‘selectedCard‘

**Access Programs:** ‘requestPlayCard(cardID)‘; ‘requestDrawCard()‘; ‘request-SuitChange(suit)‘; ‘requestScoreTally(remainingCards)‘

**Exceptions:** ‘InvalidMoveUI‘ (provides polite feedback)

#### 7.4.7 Scoreboard View Module (M13)

**Secrets:** UI layout for displaying end-of-game or end-of-round scores.

**Services:** Renders the score, clearly displaying both Decimal and Dozenal values (FR-7).

**Implemented By:** The Crazy Four (React)

**State Variables:** ‘scoreData‘

**Access Programs:** (Rendered React component based on props)

**Exceptions:** None

#### 7.4.8 Profile View Module (M14)

**Secrets:** UI layout for displaying game history and user statistics.

**Services:** Renders the user’s profile and game history (FR-15). Allows user to request data deletion (FR-17).

**Implemented By:** The Crazy Four (React)

**State Variables:** ‘profileData‘, ‘gameHistory‘: [ ]

**Access Programs:** ‘handleDeleteAccount()‘

**Exceptions:** None

## 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)

| AC | Modules |
|---|---|
| AC?? | M16, M15 |
| AC?? | M18, M17, M16, M13 |
| AC?? | M17, M13 |
| AC?? | M8, M11, M13, M10, M14 |
| AC?? | M5, M1, M14 |
| AC?? | M2, M7 |
| AC?? | M4, M9, M1 |

## 9 Use Hierarchy Between Modules

In this section, the uses hierarchy between modules is provided. Parnas (1978) said of two programs A and B that A *uses* B if correct execution of B may be necessary for A to complete the task described in its specification. That is, A *uses* B if there exist situations in which the correct functioning of A depends upon the availability of a correct implementation of B. Figure 1 illustrates the use relation between the modules. It can be seen that the graph is a directed acyclic graph (DAG). Each level of the hierarchy offers a testable and usable subset of the system, and modules in the higher level of the hierarchy are essentially simpler because they use modules from the lower levels.

[The uses relation is not a data flow diagram. In the code there will often be an import statement in module A when it directly uses module B. Module B provides the services that module A needs. The code for module A needs to be able to see these services (hence the import statement). Since the uses relation is transitive, there is a use relation without an import, but the arrows in the diagram typically correspond to the presence of import statement. —SS]

[If module A uses module B, the arrow is directed from A to B. —SS]

Figure 1: Use hierarchy among modules

## 10 User Interfaces

[Design of user interface for software and hardware. Attach an appendix if needed. Drawings, Sketches, Figma —SS]

## 11 Design of Communication Protocols

[If appropriate —SS]

## 12 Timeline

[Schedule of tasks and who is responsible —SS]

[You can point to GitHub if this information is included there —SS]

## References

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

David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press. ISBN none.

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.