System Verification and Validation Plan for ProgName

Team #25, The Crazy Four

Ruida Chen
Ammar Sharbat
Alvin Qian
Jiaming Li

October 27, 2025

Revision History

Date	Version	Notes
Oct 24	Alvin	System test functional requirements
Oct 26	Ruida	Plan
Oct 27	Jiaming	System test Non-functional requirements
Oct 27	Alvin	Fixed errors in FR tests

[The intention of the VnV plan is to increase confidence in the software. However, this does not mean listing every verification and validation technique that has ever been devised. The VnV plan should also be a **feasible** plan. Execution of the plan should be possible with the time and team available. If the full plan cannot be completed during the time available, it can either be modified to "fake it", or a better solution is to add a section describing what work has been completed and what work is still planned for the future. —SS]

[The VnV plan is typically started after the requirements stage, but before the design stage. This means that the sections related to unit testing cannot initially be completed. The sections will be filled in after the design stage is complete. the final version of the VnV plan should have all sections filled in. —SS]

Contents

List of Tables

[Remove this section if it isn't needed —SS]

List of Figures

[Remove this section if it isn't needed —SS]

1 Symbols, Abbreviations, and Acronyms

symbol	**description**
T	Test

[symbols, abbreviations, or acronyms — you can simply reference the SRS (Author, 2019) tables, if appropriate —SS]

[Remove this section if it isn't needed —SS]

This document ... [provide an introductory blurb and roadmap of the Verification and Validation plan —SS]

2 General Information

2.1 Summary

[Say what software is being tested. Give its name and a brief overview of its general functions. —SS]

2.2 Objectives

[State what is intended to be accomplished. The objective will be around the qualities that are most important for your project. You might have something like: “build confidence in the software correctness,” “demonstrate adequate usability.” etc. You won’t list all of the qualities, just those that are most important. —SS]

[You should also list the objectives that are out of scope. You don’t have the resources to do everything, so what will you be leaving out. For instance, if you are not going to verify the quality of usability, state this. It is also worthwhile to justify why the objectives are left out. —SS]

[The objectives are important because they highlight that you are aware of limitations in your resources for verification and validation. You can’t do everything, so what are you going to prioritize? As an example, if your system depends on an external library, you can explicitly state that you will assume that external library has already been verified by its implementation team. —SS]

2.3 Challenge Level and Extras

[State the challenge level (advanced, general, basic) for your project. Your challenge level should exactly match what is included in your problem statement. This should be the challenge level agreed on between you and the course instructor. You can use a pull request to update your challenge level (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

[Summarize the extras (if any) that were tackled by this project. Extras can include usability testing, code walkthroughs, user documentation, formal proof, GenderMag personas, Design Thinking, etc. Extras should have already been approved by the course instructor as included in your problem statement. You can use a pull request to update your extras (in TeamComposition.csv or Repos.csv) if your plan changes as a result of the VnV planning exercise. —SS]

2.4 Relevant Documentation

[Reference relevant documentation. This will definitely include your SRS and your other project documents (design documents, like MG, MIS, etc). You can include these even before they are written, since by the time the project is done, they will be written. You can create BibTeX entries for your documents and within those entries include a hyperlink to the documents. —SS]

Author (2019)

[Don't just list the other documents. You should explain why they are relevant and how they relate to your VnV efforts. —SS]

3 Plan

This section details the comprehensive, multi-phase plan for the verification and validation (V&V) of the " The Crazy Eights" educational card game, an application designed to teach the Dozenal (base-12) number system through a variant of the Crazy Eights card game. It establishes the V&V-related responsibilities of the development team, the processes for verifying all key project artifacts (Software Requirements Specification, System Design, Implementation), the specific tools to be leveraged for automation, and the validation strategy to ensure the final product meets its core educational and functional objectives.

3.1 Verification and Validation Team

Table 1: Verification and Validation Team Roles and Responsibilities

Name	Responsibilities
Ruida Chen	Owns SRS checklist and requirement traceability; leads ambiguity inspections; opens and triages SRS issues on GitHub.
Ammar Sharbat	Conducts MG/MIS walkthroughs; checks FR/NFR coverage; verifies pre/post-conditions in design.
Alvin Qian	Develops unit and integration tests; maintains CI coverage metrics; triages static analysis warnings.
Jiaming Li	Organizes stakeholder playtests and usability sessions; aligns validation with personas and client expectations.

3.2 SRS Verification

Objective: To ensure the Software Requirements Specification (SRS) is complete, consistent, verifiable, and aligns with the project goals and constraints defined by the client and supervisor.
Scope: This verification applies to all functional and non-functional requirements described in the SRS. It covers requirement clarity, consistency, completeness, and testability.
Supervisor Involvement: The SRS verification process will be conducted under the supervision of **Dr. Paul Rapoport**. Dr. Rapoport will act as the primary reviewer and domain expert, ensuring that the documented requirements correctly reflect the intended educational goals and gameplay design. He will provide written and verbal feedback during the SRS review meeting and confirm whether the requirements meet the course expectations. Any clarifications or corrections raised by Dr. Rapoport will be recorded in the SRS issue tracker and addressed.
Verification Techniques:

- **Checklist-based inspection:** Each requirement will be reviewed using a formal checklist to ensure it is unambiguous, measurable, and testable.

- **Cross-traceability review:** Every functional requirement (FR) will be linked to at least one design component and one planned test case.
- **Peer review:** Team members who did not author the section will perform independent reviews to prevent bias.
- **Supervisor review:** Dr. Rapoport will conduct a targeted review session to evaluate correctness, completeness, and educational intent.

Supervisor Involvement: The SRS verification process will be formally reviewed under the supervision of **Dr. Paul Rapoport**. The review will take the form of a scheduled meeting lasting approximately 30–45 minutes during the week following the internal peer review.

- **Meeting preparation:** Before the meeting, the team will prepare and send Dr. Rapoport a concise review package containing:
  1. The finalized SRS document (PDF);
  2. A one-page summary of critical or ambiguous requirements;
  3. The SRS verification checklist used in internal inspection;
  4. The traceability matrix linking requirements to design/test artifacts.
- **Structured discussion topics:** During the meeting, the team will:
  1. Present how each major functional area (Game, Scoring, Educational Support, User Management) meets the educational and usability goals;
  2. Ask Dr. Rapoport to identify unclear, incomplete, or pedagogically inconsistent requirements;
  3. Confirm mutual understanding of success criteria and measurable outcomes (e.g., gameplay flow, scoring accuracy, base conversion learning intent).
- **Feedback capture and follow-up:** Comments and action items from the meeting will be recorded in the GitHub Issue Tracker under the tag `SRS-Review`, each with an assigned owner and deadline. The SRS Lead (Ruida Chen) will ensure all feedback is incorporated into the updated SRS before sign-off.

- **Supervisor:** After corrections, Dr. Rapoport will be provided with a change summary.

Checklist

Table 2: SRS Verification Checklist

Checklist Item ID	Category	Verification Question
SRS-C-01	Completeness	Have all Open Issues been resolved and their resolutions integrated?
SRS-C-02	Completeness	Has all placeholder content been replaced with substantive text?
SRS-C-03	Testability	Does every functional requirement have a unique identifier?
SRS-C-04	Testability	Does every requirement now have a specific, measurable, and unambiguous Fit Criterion?
SRS-C-05	Consistency	Are the game rules consistent? (e.g., Is FR-3 consistent with PUC-5?)
SRS-C-06	Consistency	Is the technology stack (React, Node.js, Postgres) consistent across Sec 3.1, 13.2, and 26.1?
SRS-C-07	Correctness	Do all requirements trace back to a stated Project Goal or Stakeholder?
SRS-C-08	Traceability	Is the scope unambiguous?
SRS-C-09	Feasibility	Can all requirements be realistically implemented within the project constraints?

3.3 Design Verification

Objective: To verify that the system design is a correct, complete, and feasible implementation of the verified SRS
Process: Design Walkthroughs & Peer Review

- The team will create design artifacts such as UML diagram, database Enitity-Relationship-Diagram (ERD), etc.
- Schedule formal review meetings
- In these meetings, the team will "walk through" the design, explaining how it fullfills specific requirements from the SRS.
- The team will use the checklist below to challenge the design and ensure full traceability from requirements to design.
- All identified defects will be logged as GitHub Issues.

Checklist

Table 3: Design Verification Checklist

Checklist Item ID	**Category**	**Verification Question**
DS-C-01	Traceability	Does every component in the architecture map to one or more Functional Requirements?
DS-C-02	Completeness	Does the design for the ”Game Manager” (Sec 9.1) component explicitly implement all rules)?
DS-C-03	NFR-Performance	Does the architecture (Sec 26.1) support the 200 concurrent user requirement?
DS-C-04	NFR-Security	Does the design explicitly address all security requirements, including ”server-authoritative game state” ?
DS-C-05	NFR-Maintain.	Does the design adhere to the mandated stack (React, Node.js)?[1] Is it modular and extensible ?
DS-C-06	NFR-Usability	Does the UI/UX design explicitly address Accessibility and Learning?
DS-C-07	Interface	Are all API endpoints clearly defined?
DS-C-08	Data	Does the database schema (ERD) correctly model the Business Data Model (Sec 7.1) and Data Dictionary (Sec 7.2)?

3.4 Verification and Validation Plan Verification

Objective: To verify this VnVPlan for completeness, feasibility, internal consistency, and adherence to all course rubrics.
Process: Formal Peer Review

- The entire teamshall conduct one full read-through of this document before submitting it.
- The team will use the checklist below, which is directly derived from the course grading rubric, to ensure all requirements have been met.
- After the team review, the plan shall be informally presented to the course supervisor to get a ”feasibility check” before implementation begins, mitigating project risk.

:Checklist

Table 4: V&V Plan Verification Checklist

Checklist Item ID	**Verification Question**
VVP-C-01	Are team roles for V&V clear, specific, and feasible?
VVP-C-02	Is the SRS verification plan clear, specific, and feasible?
VVP-C-03	Is the Design verification plan clear, specific, and feasible?
VVP-C-04	Is the Implementation verification plan clear, specific, and feasible?
VVP-C-05	Are the automated testing and verification tools specific?
VVP-C-06	Is the software validation plan clear, specific, and feasible?
VVP-C-08	Is the plan feasible given the team size and academic term timeline?

3.5 Implementation Verification

Objective: To verify that the source code artifacts correctly implement the verified design, adhere to all coding standards, are free of common defects, and meet all non-functional requirements.
Unit testing

- **Frontend**:All UI components, verifying correct rendering and user-interaction behavior.

- **Backend**: All API endpoints, authentication logic, and data models.
- **Game Logic**: This is the most critical module to unit test. Tests will cover every game rule, include happy path + all edge cases.

Integration testing

- **Frontend**:API + DB (login/session/history), engine + UI interaction (play card → rule check → UI update).

Static Verification: In addition to dynamic testing, static verification activities will include:

- **Code walk-throughs:** Weekly peer reviews where presents newly developed code to the team, explaining logic and verifying alignment with design contracts.
- **Code inspection:** Checklist-based inspections for coding-standard compliance (naming, commenting, error handling).
- **Static analyzers:** Use of TypeScript's –strict mode, ESLint rules, and CodeQL analysis for detecting unused variables, type mismatches, and security issues.
- **Continuous Integration enforcement:** The CI pipeline will fail automatically if linting, type checking, or static analysis errors occur.

3.6 Automated Testing and Verification Tools

Objective: To ensure that software verification activities are repeatable, efficient, and integrated with the development workflow.
Tools and Automation:

- **Jest** — used for unit and integration testing of both frontend and backend components.
- **Playwright** — performs automated end-to-end testing of the user interface.
- **ESLint & Prettier** — automatically enforce coding style and detect syntax or logical issues.

- **GitHub Actions CI/CD** — runs all tests automatically on each push or pull request; generates coverage and linting reports.
- **Codecov** — collects and visualizes code coverage results from CI.

Automation Outcome: Automated testing ensures that every commit is verified for correctness, code quality, and integration stability before merging into the main branch.

3.7 Software Validation

Objective: To confirm that the final system meets user needs, project goals, and the expectations of the client and supervisor.
Scope: Validation focuses on ensuring that the software behaves as intended by users and stakeholders, not only that it was built correctly.
Validation Methods:

- **Stakeholder Review:** Conduct a demonstration of the working prototype for Dr. Paul Rapoport. Feedback will be collected and documented for any changes before final submission.
- **Task-based User Testing:** Team members and sample users will perform core gameplay tasks (e.g., joining a lobby, playing a round, viewing results). Observations will be recorded to identify usability or clarity issues.
- **Survey and Feedback:** A short questionnaire will be used to gather feedback on ease of use, performance, and enjoyment.

Success Criteria:

- All core gameplay and scoring functions work as described in the SRS.
- Supervisor and users confirm that the software meets educational and functional goals.
- No critical usability or stability issues remain unresolved.

Deliverables: Validation summary report including stakeholder feedback, test logs, and any final fixes implemented before release.

4 System Tests

This section defines end to end system tests that validate the software against the SRS Functional Requirements (FR-1..FR-17) for the MVP implementation state. Tests are grouped by related functionality to improve traceability and reuse of fixtures. Unless stated otherwise, tests assume a deterministic deck seed, two authenticated test users, and a clean database. All expected results are observable via the UI and server state exposed to the test harness.

4.1 Tests for Functional Requirements

The subsections below cover: (1) game flow, rule validation, special cards, and end conditions (SRS Game Manager: FR-1..FR-5), (2) scoring and valid move highlighting (SRS Score Manager and Highlighting: FR-6..FR-9), (3) account access flows (Login Manager: FR-10..FR-13), and (4) persistence operations (Data Manager: FR-14..FR-17). Where tests rely on input constraints, they use the card value and move legality definitions in the SRS.

4.1.1 Game Manager: Start, Turn, Rules, Specials, End (FR-1..FR-5)

This area verifies new game initialization, turn sequencing, rule validation for same suit or same value or sum equals 12 (base-12), handling of wild eights, and detecting the end of a game. It directly traces to FR-1 Start new game, FR-2 Turn management, FR-3 Rule validation, FR-4 Special cards, FR-5 End of game.

Start new game initializes state (ST-GM-01)

1. ST-GM-01
   Control: Automatic

   Initial State: No active sessions. Two users are authenticated and in the lobby. Deterministic deck seed S is configured.

   Input: User A clicks New Game and invites User B. User B accepts.

   Output: A new session is created with a shuffled deck using seed S, a discard starter card is placed, each player receives the configured hand size H, and the UI shows turn = User A.

Test Case Derivation: From FR-1 and SRS deck initialization semantics. With a fixed seed S, the initial state is uniquely determined.

How test will be performed: Run a Node console harness that calls `initGame('S')`, then logs and checks via simple equality that `hands` have size H, `discard.length == 1`, and `turn == UserA`. No browser, no sockets.

Legal move: same suit (ST-GM-02)

1. ST-GM-02
   Control: Automatic

   Initial State: Active session. Discard top = 5♡. User A hand contains K♡ and others. Turn = User A.

   Input: User A plays K♡.

   Output: Move is accepted. Discard top becomes K♡. Turn advances to User B.

   Test Case Derivation: FR-3 allows a move when suits match.

   How test will be performed: In the console harness set `discard = [suit:'H',v:5]` and handA contains K♡. Call `canPlay()` then `playCard()`. Log PASS if top becomes K♡ and turn flips to UserB.

Legal move: sum equals 12 (base-12) (ST-GM-03)

1. ST-GM-03
   Control: Automatic

   Initial State: Discard top = 5♢ (value 5). User A hand contains 7♠ (value 7). Turn = User A.

   Input: User A plays 7♠.

   Output: Move is accepted. Discard top becomes 7♠. Turn advances to User B.

   Test Case Derivation: In base-12, $5 + 7 = 10_{12}$ which equals 12 in decimal. FR-3 permits play when values sum to 12 base-12.

   How test will be performed: Set top to value 5 then attempt to play value 7. Use `canPlay()` to assert true and log PASS.

Special card: wild eight suit selection (ST-GM-04)

1. ST-GM-04
   Control: Automatic

   Initial State: Discard top = K♣. User A hand contains 8♡. Turn = User A.

   Input: User A plays 8♡ and selects Spades as the new suit.

   Output: Move is accepted. Discard shows 8 with chosen suit indicator = Spades. Only Spades or otherwise legal responses are allowed for the next player.

   Test Case Derivation: FR-4 specifies 8s are wild and allow the player to declare a suit.

   How test will be performed: Give A an 8, play with `playCard(state,'A',eight,{choose` then check `chosenSuit == 'S'` and `turn == 'B'` via console logs.

Illegal move is rejected with guidance (ST-GM-05)

1. ST-GM-05
   Control: Automatic

   Initial State: Discard top = 9◇. User A hand contains Q♣ and no other legal card.

   Input: User A attempts to play Q♣.

   Output: Move is rejected. UI shows a polite message explaining why the move is invalid and highlights available actions: draw or play a legal card if any.

   Test Case Derivation: FR-3 rejects moves that are not same suit, not same value, and do not sum to 12 base-12.

   How test will be performed: Call `canPlay()` on an invalid card and expect `false`. For now we just assert the engine rejects the move; UI message will be verified later in UI tests.

End of game detection (ST-GM-06)

1. ST-GM-06
   Control: Automatic

Initial State: User A has 1 card, which is legal to play. User B has multiple cards.

Input: User A plays the last card.

Output: Game ends. Winner = User A. System transitions to score calculation and displays results.

Test Case Derivation: FR-5 requires ending the game when a player has no cards left.

How test will be performed: Have a scripted state where A has one legal card, call `playCard()`, then log PASS if `state.status == 'Completed'` and `winner == 'A'`.

4.1.2 Score Features: Dual-base, Highlights (FR-6..FR-9)

This area validates round scoring in decimal and Dozenal, correct conversion and display, and visual highlighting of valid moves. Education Support and hints are out of scope.

Score calculated and shown in decimal and Dozenal (ST-SC-01)

1. ST-SC-01
   Control: Automatic

   Initial State: Game just ended. Opponent remaining cards have numeric values 2, 4, 6 (per SRS value function).

   Input: Trigger score calculation screen.

   Output: Decimal total = 12. Dozenal total = 10_{12}. Both are displayed side by side.

   Test Case Derivation: 2 + 4 + 6 = 12 decimal which equals 10 in base-12.

   How test will be performed: Call `calculateScore({B:[2,4,6]})` and expect { `decimal:12, dozenal:'10'` }. Log PASS or FAIL.

Valid moves highlighted exactly (ST-SC-03)

1. ST-SC-03
   Control: Automatic

Initial State: Discard top = 4♠. Hand contains 4♣, 8♢, 7♡, Q♠, 6♣.

Input: None beyond rendering the hand.

Output: Highlighted cards are exactly {4♣, Q♠, 8♢}. 7♡ and 6♣ are not highlighted.

Test Case Derivation: Same value (4), same suit (Spades), and wild eight are valid. Others are invalid because $7 + 4 = 11$ decimal which is not 10_{12} and $6 + 4 = 10$ decimal which is not 10_{12}.

How test will be performed: Call exported `listValidMoves(state, hand)` and assert it returns exactly the IDs for same value, same suit, and wild eight. Log PASS or FAIL.

Primary base toggle affects labeling only (ST-SC-04)

1. ST-SC-04
   Control: Automatic

   Initial State: Score view shows decimal on the left and Dozenal on the right with decimal as primary.

   Input: Toggle primary base to Dozenal.

   Output: Labels and prominence swap, but numeric values remain identical pair {12, 10_{12}}. No recomputation changes total values.

   Test Case Derivation: FR-7 specifies dual display; the toggle is a presentation choice.

   How test will be performed: For MVP engine tests, verify the conversion outputs are stable. The UI toggle is deferred; assert that conversion of 12 dec remains 10_{12} before and after calling a `setPrimaryBase()` no-op stub.

4.1.3 Login Manager: Account, Sessions, Guest, Validation (FR-10..FR-13)

This area verifies account creation, login and logout, guest sessions, and credential checks.

Account creation creates unique user and session (ST-LM-01)

1. ST-LM-01
   Control: Automatic

   Initial State: No account exists for username `U`. Clean auth store.

   Input: Submit sign up with `username=U`, `password=P` that meets policy.

   Output: Account is created with unique `userId`. Password is stored as a salted hash (not plaintext). A logged-in session for `U` is active.

   Test Case Derivation: FR-10 requires creation with unique username and password.

   How test will be performed: In Node harness call `createAccount(U,P)`. Assert `store.users[U].passwordHash` exists and `store.sessions.has(userId)`. Then attempt a second `createAccount(U,P2)` and assert rejection.

Login and logout preserve progress (ST-LM-02)

1. ST-LM-02
   Control: Automatic

   Initial State: Account `U` exists with prior game history `H0`. No active session.

   Input: `login(U,P)` then start a game, finish one hand, `logout()`, then `login(U,P)`.

   Output: Session is established on both logins. After re-login, history includes prior `H0` plus the new hand `H1`. No progress is lost at logout.

   Test Case Derivation: FR-11 requires login and logout at any time without losing progress.

   How test will be performed: Call `login`, run minimal game through engine hooks, call `logout`, then `login` again. Compare history count before and after.

Guest mode creates temporary session (ST-LM-03)

1. ST-LM-03
   Control: Automatic

Initial State: No active session.

Input: `createGuestSession()` then play a short hand.

Output: Session has `guest:true`. No persistent user record is created. On process restart or `endGuestSession()`, the profile is not retrievable.

Test Case Derivation: FR-12 allows a temporary session without login.

How test will be performed: Call `createGuestSession()`, verify `session.guest == true`. Write a hand, restart in-memory store, assert no user profile exists for the guest.

Credential validation gates access (ST-LM-04)

1. ST-LM-04
   Control: Automatic

   Initial State: Account `U` exists with password `P`. No active session.

   Input: Attempt `login(U,'wrong')`, then `login(U,P)`.

   Output: First attempt is rejected with an auth error. No session is created. Second attempt succeeds and creates a session.

   Test Case Derivation: FR-13 requires validating credentials against stored records before granting access.

   How test will be performed: Call `login` with wrong password and assert failure and `sessions.size` unchanged, then with correct password and assert success.

4.1.4 Data Manager: Storage, Retrieval, Update, Deletion (FR-14..FR-17)

This area verifies secure storage of usernames, game history, and Dozenal scores, along with retrieval, update, and deletion.

User data is stored on events that require persistence (ST-DM-01)

1. ST-DM-01
   Control: Automatic

   Initial State: Authenticated user `U`. Clean data store.

Input: Finish a game where opponent holds cards with values {2,4,6}. Trigger save.

Output: A record exists for `U` containing username, a new game history entry with decimal total 12 and Dozenal 10_{12}.

Test Case Derivation: FR-14 requires secure storage of user data including game history and Dozenal scores.

How test will be performed: Call `saveGameResult(U, result)`. Assert schema fields exist and values match expected totals. For MVP we verify presence and correctness; deeper security assessments are separate.

Stored data can be retrieved on login or profile view (ST-DM-02)

1. ST-DM-02
   Control: Automatic

   Initial State: Data store contains prior records for `U`.

   Input: `login(U,P)` then `getProfile(U)`.

   Output: Retrieved profile includes username, cumulative game history, and Dozenal scores as saved.

   Test Case Derivation: FR-15 requires retrieval of stored user data.

   How test will be performed: Assert `getProfile(U)` returns non-empty history and last score equals most recent saved value.

Data updates after games and profile changes (ST-DM-03)

1. ST-DM-03
   Control: Automatic

   Initial State: Profile for `U` exists with history length `n`.

   Input: Complete another game and call `saveGameResult`. Then change profile display name via `updateProfile(U,{displayName:'A'})`.

   Output: History length becomes `n+1`. Profile `displayName` is updated. Timestamps reflect the latest update.

   Test Case Derivation: FR-16 requires updating stored data after each game or profile change.

How test will be performed: Compare history length and profile fields before and after the operations.

User can permanently delete data (ST-DM-04)

1. ST-DM-04
   Control: Automatic

   Initial State: Profile and history exist for `U`.

   Input: `deleteUserData(U)` then attempt `getProfile(U)` and `login(U,P)`.

   Output: User records, history, and scores are removed. `getProfile(U)` returns not found. Login is rejected since the account data has been deleted or deactivated per policy.

   Test Case Derivation: FR-17 requires permanent deletion upon request.

   How test will be performed: Call `deleteUserData` and assert data store no longer contains keys for `U`. Verify subsequent access fails.

5 System Tests

This section defines end to end system tests that validate the software against the SRS Functional Requirements (FR-1..FR-17) for the MVP implementation state, with SRS FR-9.3 (Education Support and hints) excluded from scope. Tests are grouped by related functionality to improve traceability and reuse of fixtures. Unless stated otherwise, tests assume a deterministic deck seed, two authenticated test users, and a clean database. All expected results are observable via the UI and server state exposed to the test harness.

5.1 Tests for Functional Requirements

The subsections below cover: (1) game flow, rule validation, special cards, and end conditions (SRS Game Manager: FR-1..FR-5), (2) scoring and valid move highlighting (SRS Score Manager and Highlighting: FR-6..FR-9 except FR-9.3), (3) account access flows (Login Manager: FR-10..FR-13), and (4) persistence operations (Data Manager: FR-14..FR-17). Where tests rely on input constraints, they use the card value and move legality definitions in the SRS.

5.1.1 Game Manager: Start, Turn, Rules, Specials, End (FR-1..FR-5)

This area verifies new game initialization, turn sequencing, rule validation for same suit or same value or sum reaches 11 in the active base, handling of wild tens, and detecting the end of a game. It directly traces to FR-1 Start new game, FR-2 Turn management, FR-3 Rule validation, FR-4 Special cards, FR-5 End of game.

Start new game initializes state (ST-GM-01)

1. ST-GM-01
   Control: Automatic

   Initial State: No active sessions. Two users are authenticated and in the lobby. Deterministic deck seed S is configured. Base mode is set by configuration.

   Input: User A clicks New Game and invites User B. User B accepts.

   Output: A new session is created with a shuffled deck using seed S, a discard starter card is placed, each player receives the configured hand size H, and the UI shows turn = User A. If Base = Decimal then deck size = 52. If Base = Dozenal then deck size = 64.

   Test Case Derivation: From FR-1 and SRS deck initialization semantics. With a fixed seed S, the initial state is uniquely determined.

   How test will be performed: Run a Node console harness that calls `initGame('S')`, then logs and checks via simple equality that `hands` have size H, `discard.length == 1`, `turn == UserA`, and deck size matches base mode. No browser, no sockets.

Legal move: same suit (ST-GM-02)

1. ST-GM-02
   Control: Automatic

   Initial State: Active session. Discard top = 5♡. User A hand contains K♡ and others. Turn = User A.

   Input: User A plays K♡.

Output: Move is accepted. Discard top becomes K♡. Turn advances to User B.

Test Case Derivation: FR-3 allows a move when suits match.

How test will be performed: In the console harness set `discard = [suit:'H',v:5]` and handA contains K♡. Call `canPlay()` then `playCard()`. Log PASS if top becomes K♡ and turn flips to UserB.

Legal move: sum reaches target 11 in active base (ST-GM-03)

1. ST-GM-03
   Control: Automatic

   Initial State: Base mode = Dozenal. Discard top has value 6. User A hand contains a card with value 7. Turn = User A.

   Input: User A plays the 7.

   Output: Move is accepted. Discard top becomes the 7. Turn advances to User B.

   Test Case Derivation: In dozenal, $6 + 7 = 11_z$. In decimal mode, valid examples include $5 + 6 = 11_d$. Rule is parameterized by base.

   How test will be performed: Set top value and hand to match the base mode, call `canPlay()`, expect `true`, log PASS.

Special card: wild ten suit selection (ST-GM-04)

1. ST-GM-04
   Control: Automatic

   Initial State: Discard top = K♣. User A hand contains 10♡. Turn = User A.

   Input: User A plays 10♡ and selects Spades as the new suit.

   Output: Move is accepted. Discard shows 10 with chosen suit indicator = Spades. Only Spades or otherwise legal responses are allowed for the next player.

   Test Case Derivation: FR-4 specifies the wild ten allows the player to declare a suit.

   How test will be performed: Give A a ten, call `playCard(state,'A',ten,{chooseSuit:'` then check `chosenSuit == 'S'` and `turn == 'B'` via console logs.

Illegal move is rejected with guidance (ST-GM-05)

1. ST-GM-05
   Control: Automatic

   Initial State: Discard top = 9♢. User A hand contains Q♣ and no other legal card.

   Input: User A attempts to play Q♣.

   Output: Move is rejected. UI shows a polite message explaining why the move is invalid and highlights available actions: draw or play a legal card if any.

   Test Case Derivation: FR-3 rejects moves that are not same suit, not same value, and cannot reach the target 11 in the active base.

   How test will be performed: Call `canPlay()` on an invalid card and expect `false`. For now we just assert the engine rejects the move; UI message will be verified later in UI tests.

End of game detection (ST-GM-06)

1. ST-GM-06
   Control: Automatic

   Initial State: User A has 1 card, which is legal to play. User B has multiple cards.

   Input: User A plays the last card.

   Output: Game ends. Winner = User A. System transitions to score calculation and displays results.

   Test Case Derivation: FR-5 requires ending the game when a player has no cards left.

   How test will be performed: Have a scripted state where A has one legal card, call `playCard()`, then log PASS if `state.status == 'Completed'` and `winner == 'A'`.

Player may draw repeatedly until a legal move exists (ST-GM-07)

1. ST-GM-07
   Control: Automatic

Initial State: No legal moves in hand for the current discard top. Draw pile has at least 3 cards.

Input: Player chooses Draw, Draw, Draw, then attempts a play when a legal card is obtained.

Output: Each draw adds a card to hand. No penalty. As soon as a legal card exists, a play is permitted.

Test Case Derivation: UC5 allows repeated draws.

How test will be performed: Force a state with zero legal moves, call `drawCard()` multiple times, assert no error, then `canPlay()` becomes `true`.

Optional Queen rule only applies when enabled (ST-GM-08)

1. ST-GM-08
   Control: Automatic

   Initial State: Config A: `queenRule=false`. Config B: `queenRule=true`. Discard top as required by the rule.

   Input: Attempt a Queen move under both configs.

   Output: Under Config A the move is rejected. Under Config B the move is accepted with the documented effect.

   Test Case Derivation: Optional Queen rule is a configurable variant in the SRS.

   How test will be performed: Toggle config flag, call `canPlay()` and `playCard()`, assert behavior matches config.

5.1.2 Score Features: Dual-base, Highlights (FR-6..FR-9 except FR-9.3)

This area validates round scoring in decimal and dozenal, correct conversion and display where enabled, and visual highlighting of valid moves. Education Support and hints are out of scope.

Score calculated and shown in decimal and dozenal (ST-SC-01)

1. ST-SC-01
   Control: Automatic

   Initial State: Game just ended. Opponent remaining cards have numeric values 2, 4, 6 (per SRS value function).

   Input: Trigger score calculation screen.

   Output: Decimal total = 12. Dozenal total = 10_{12}. If dual display is enabled both are shown side by side.

   Test Case Derivation: 2 + 4 + 6 = 12 decimal which equals 10_{12}.

   How test will be performed: Call `calculateScore({B:[2,4,6]})` and expect { `decimal:12, dozenal:'10'` }. Log PASS or FAIL.

Valid moves highlighted exactly (ST-SC-03)

1. ST-SC-03
   Control: Automatic

   Initial State: Base mode = Decimal. Discard top = 4♠. Hand contains 4♣, 10♢, 5♡, Q♠, 6♣.

   Input: None beyond rendering the hand.

   Output: Highlighted cards are exactly {4♣, Q♠, 10♢}. 5♡ and 6♣ are not highlighted.

   Test Case Derivation: Same value (4), same suit (Spades), and wild ten are valid. Others are invalid because $4 + 5 = 9_d$ and $4 + 6 = 10_d$, which do not reach the target 11_d.

   How test will be performed: Call exported `listValidMoves(state, hand)` and assert it returns exactly the IDs for same value, same suit, and wild ten. Log PASS or FAIL.

Primary base toggle affects labeling only (optional) (ST-SC-04)

1. ST-SC-04
   Control: Automatic

   Initial State: Dual display is enabled by config. Score view shows decimal on the left and dozenal on the right with decimal as primary.

   Input: Toggle primary base to dozenal.

Output: Labels and prominence swap. Numeric pair stays $\{12, 10_{12}\}$. No recomputation changes totals.

Test Case Derivation: Presentation feature that is optional per SRS.

How test will be performed: Verify conversion outputs are stable across a no-op toggle in the engine.

Match ends when a player reaches 100 in the active base (ST-SC-05)

1. ST-SC-05
   Control: Automatic

   Initial State: Base mode = Decimal. Cumulative scores: A = 96_d, B = 80_d. Round just ended and A earned 4_d.

   Input: Trigger match score update.

   Output: A reaches or exceeds 100 in the active base, match status = Completed, winner = A.

   Test Case Derivation: SRS requires multiple rounds until a player hits 100 in the active base. A dozenal variant would check 100_z with appropriate cumulative values.

   How test will be performed: Call `applyRoundScore()` with base mode set, assert match completion when the threshold is met.

5.1.3 Login Manager: Account, Sessions, Guest, Validation (FR-10..FR-13)

This area verifies account creation, login and logout, guest sessions, and credential checks.

Account creation creates unique user and session (ST-LM-01)

1. ST-LM-01
   Control: Automatic

   Initial State: No account exists for username `U`. Clean auth store.

   Input: Submit sign up with `username=U`, `password=P` that meets policy.

Output: Account is created with unique `userId`. Password is stored as a salted hash (not plaintext). A logged-in session for `U` is active.

Test Case Derivation: FR-10 requires creation with unique username and password.

How test will be performed: In Node harness call `createAccount(U,P)`. Assert `store.users[U].passwordHash` exists and `store.sessions.has(userId)`. Then attempt a second `createAccount(U,P2)` and assert rejection.

Login and logout preserve progress (ST-LM-02)

1. ST-LM-02
   Control: Automatic

   Initial State: Account `U` exists with prior game history `H0`. No active session.

   Input: `login(U,P)` then start a game, finish one hand, `logout()`, then `login(U,P)`.

   Output: Session is established on both logins. After re-login, history includes prior `H0` plus the new hand `H1`. No progress is lost at logout.

   Test Case Derivation: FR-11 requires login and logout at any time without losing progress.

   How test will be performed: Call `login`, run minimal game through engine hooks, call `logout`, then `login` again. Compare history count before and after.

Guest mode creates temporary session (ST-LM-03)

1. ST-LM-03
   Control: Automatic

   Initial State: No active session.

   Input: `createGuestSession()` then play a short hand.

   Output: Session has `guest:true`. No persistent user record is created. On process restart or `endGuestSession()`, the profile is not retrievable.

   Test Case Derivation: FR-12 allows a temporary session without login.

How test will be performed: Call `createGuestSession()`, verify `session.guest == true`. Write a hand, restart in-memory store, assert no user profile exists for the guest.

Credential validation gates access (ST-LM-04)

1. ST-LM-04
   Control: Automatic

   Initial State: Account `U` exists with password `P`. No active session.

   Input: Attempt `login(U,'wrong')`, then `login(U,P)`.

   Output: First attempt is rejected with an auth error. No session is created. Second attempt succeeds and creates a session.

   Test Case Derivation: FR-13 requires validating credentials against stored records before granting access.

   How test will be performed: Call `login` with wrong password and assert failure and `sessions.size` unchanged, then with correct password and assert success.

5.1.4 Data Manager: Storage, Retrieval, Update, Deletion (FR-14..FR-17)

This area verifies secure storage of usernames, game history, and dozenal scores, along with retrieval, update, and deletion.

User data is stored on events that require persistence (ST-DM-01)

1. ST-DM-01
   Control: Automatic

   Initial State: Authenticated user `U`. Clean data store.

   Input: Finish a game where opponent holds cards with values {2,4,6}. Trigger save.

   Output: A record exists for `U` containing username, a new game history entry with decimal total 12 and dozenal 10_{12}.

   Test Case Derivation: FR-14 requires secure storage of user data including game history and dozenal scores.

How test will be performed: Call `saveGameResult(U, result)`. Assert schema fields exist and values match expected totals. For MVP we verify presence and correctness; deeper security assessments are separate.

Stored data can be retrieved on login or profile view (ST-DM-02)

1. ST-DM-02
   Control: Automatic

   Initial State: Data store contains prior records for `U`.

   Input: `login(U,P)` then `getProfile(U)`.

   Output: Retrieved profile includes username, cumulative game history, and dozenal scores as saved.

   Test Case Derivation: FR-15 requires retrieval of stored user data.

   How test will be performed: Assert `getProfile(U)` returns non-empty history and last score equals most recent saved value.

Data updates after games and profile changes (ST-DM-03)

1. ST-DM-03
   Control: Automatic

   Initial State: Profile for `U` exists with history length `n`.

   Input: Complete another game and call `saveGameResult`. Then change profile display name via `updateProfile(U,{displayName:'A'})`.

   Output: History length becomes `n+1`. Profile `displayName` is updated. Timestamps reflect the latest update.

   Test Case Derivation: FR-16 requires updating stored data after each game or profile change.

   How test will be performed: Compare history length and profile fields before and after the operations.

User can permanently delete data (ST-DM-04)

1. ST-DM-04
   Control: Automatic

   Initial State: Profile and history exist for `U`.

   Input: `deleteUserData(U)` then attempt `getProfile(U)` and `login(U,P)`.

   Output: User records, history, and scores are removed. `getProfile(U)` returns not found. Login is rejected since the account data has been deleted or deactivated per policy.

   Test Case Derivation: FR-17 requires permanent deletion upon request.

   How test will be performed: Call `deleteUserData` and assert data store no longer contains keys for `U`. Verify subsequent access fails.

...

5.2 Tests for Nonfunctional Requirements

5.2.1 Performance and Responsiveness (NFR-1)

Objective: Verify that all user interactions, such as playing a card or updating the score, occur with a latency below 200 ms under normal network conditions.

Type: Dynamic, Automatic

Initial State: Running MVP build on localhost with two authenticated test users and a seeded deck.

Input / Condition: Simulated play sessions of 100 rounds using Playwright scripts.

Expected Output / Metric: Average response time $\leq$ 200 ms, 95th-percentile $\leq$ 350 ms; no dropped WebSocket messages.

How test will be performed: Use Playwright performance APIs to measure latency between client events and UI updates. Aggregate results into a summary table through CI (GitHub Actions). If thresholds are exceeded, flag the run as a warning and create an issue.

5.2.2 Usability and User Satisfaction (NFR-2)

Objective: Assess how easily new users can understand and play the Dozenal Crazy Eights game.

Type: Dynamic, Manual

Initial State: Usable prototype deployed on a test server. Five participants who have never played the Dozenal version will take part in a guided session.

Input / Condition: Each participant completes a 15-minute task sequence (start a game, play 3 rounds, view score explanation).

Expected Output / Metric: At least 80% of participants successfully finish the task without assistance; post-test survey average rating ≥ 4 (out of 5) on clarity and enjoyment.

How test will be performed: Observation and screen recording during gameplay, followed by a 5-question Likert survey (see Appendix 6.2). Feedback will be compiled into a usability report.

5.2.3 Stability and Robustness (NFR-3)

Objective: Confirm that the application handles unexpected network events and invalid inputs gracefully.

Type: Dynamic, Automatic + Manual Fault Injection

Initial State: Running multi-client session connected via WebSocket.

Input / Condition: Simulate network interruptions and send malformed game actions through a test harness.

Expected Output / Metric: Application recovers without crash or data loss; users receive an error message "Connection lost – reconnecting. . . " and game state resyncs within 5 seconds.

How test will be performed: Inject disconnects using Playwright's network emulation tools. Review browser console and server logs for exceptions. Report mean recovery time and failure rate.

5.2.4 Maintainability and Code Quality (NFR-4)

Objective: Evaluate source-code consistency and test coverage as indicators of maintainability.

Type: Static Analysis / Automated

Initial State: Repository after successful build on GitHub Actions.

Input / Condition: Run ESLint, Prettier, and CodeQL scans; collect coverage data via Codecov.

Expected Output / Metric: No ESLint errors or critical CodeQL alerts; line coverage $\geq 80\%$.

How test will be performed: Continuous Integration pipeline executes static analysis and unit testing jobs on each commit. Results are stored as badges in the repository README for transparency.

5.3 Traceability Between Test Cases and Requirements

[Provide a table that shows which test cases are supporting which requirements. —SS]

6 Unit Test Description

[This section should not be filled in until after the MIS (detailed design document) has been completed. —SS]

[Reference your MIS (detailed design document) and explain your overall philosophy for test case selection. —SS]

[To save space and time, it may be an option to provide less detail in this section. For the unit tests you can potentially layout your testing strategy here. That is, you can explain how tests will be selected for each module.

For instance, your test building approach could be test cases for each access program, including one test for normal behaviour and as many tests as needed for edge cases. Rather than create the details of the input and output here, you could point to the unit testing code. For this to work, you code needs to be well-documented, with meaningful names for all of the tests. —SS]

6.1 Unit Testing Scope

[What modules are outside of the scope. If there are modules that are developed by someone else, then you would say here if you aren't planning on verifying them. There may also be modules that are part of your software, but have a lower priority for verification than others. If this is the case, explain your rationale for the ranking of module importance. —SS]

6.2 Tests for Functional Requirements

[Most of the verification will be through automated unit testing. If appropriate specific modules can be verified by a non-testing based technique. That can also be documented in this section. —SS]

6.2.1 Module 1

[Include a blurb here to explain why the subsections below cover the module. References to the MIS would be good. You will want tests from a black box perspective and from a white box perspective. Explain to the reader how the tests were selected. —SS]

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

2. test-id2

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input:

   Output: [The expected result for the given inputs —SS]

   Test Case Derivation: [Justify the expected value given in the Output field —SS]

   How test will be performed:

3. ...

6.2.2 Module 2

...

6.3 Tests for Nonfunctional Requirements

[If there is a module that needs to be independently assessed for performance, those test cases can go here. In some projects, planning for nonfunctional tests of units will not be that relevant. —SS]

[These tests may involve collecting performance data from previously mentioned functional tests. —SS]

6.3.1 Module ?

1. test-id1

   Type: [Functional, Dynamic, Manual, Automatic, Static etc. Most will be automatic —SS]

   Initial State:

   Input/Condition:

   Output/Result:

   How test will be performed:

2. test-id2

   Type: Functional, Dynamic, Manual, Static etc.

   Initial State:

   Input:

   Output:

   How test will be performed:

6.3.2 Module ?

...

6.4 Traceability Between Test Cases and Modules

[Provide evidence that all of the modules have been considered. —SS]

References

Author Author. System requirements specification. https://github.com/..., 2019.

7 Appendix

This is where you can place additional information.

7.1 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

7.2 Usability Survey Questions?

[This is a section that would be appropriate for some projects. —SS]

Appendix — Reflection

[This section is not required for CAS 741 —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Lifelong Learning.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing "what you think the evaluator wants to hear."

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. What knowledge and skills will the team collectively need to acquire to successfully complete the verification and validation of your project? Examples of possible knowledge and skills include dynamic testing knowledge, static testing knowledge, specific tool usage, Valgrind etc. You should look to identify at least one item for each team member.
4. For each of the knowledge areas and skills identified in the previous question, what are at least two approaches to acquiring the knowledge or mastering the skill? Of the identified approaches, which will each team member pursue, and why did they make this choice?