Software Requirements Document

Color Clash – A Two-Player Fighting Game

TOBB ETU – BIL 482
February 2026

Team Members
Mustafa GÖZÜTOK
Mehmet Umur ÖZÜ
Mehmet Yasin TOSUN

Table of Contents

1. Product Perspective

Color Clash is a standalone web-based two-player fighting game developed as an educational project for the TOBB ETU BIL 482 course. The application demonstrates the effective use of object-oriented software design patterns within an interactive game development context. The game runs entirely in the browser using HTML5 Canvas and TypeScript, requiring no server-side components or external dependencies beyond the build toolchain (Vite).

Context

Aspect	Description
Platform	Web browser application (desktop/laptop focused)
Project Type	Educational demonstration of design patterns in game development
Deployment	Self-hosted static web application
Technology Stack	TypeScript, HTML5 Canvas, Vite build system

Key Characteristics

- Local multiplayer game with two players sharing a single keyboard
- Real time combat mechanics with visual feedback
- Local tournament mode for group play with multiple participants
- Pattern-driven architecture showcasing professional software design practices
- No persistence layer sessions are ephemeral
- Child friendly, non-violent, and colorful game design

2. Product Functions

The system provides the following core functions:

Two Player Combat System: Simultaneous control of two characters on a shared arena with real time collision detection, hit registration, health tracking, and victory condition detection.

Character Specific Combat Mode System: A scalable strategy system where each character possesses unique fighting modes (e.g., Fire/Water for one character, Earth/Wind for another). Players can dynamically switch between these modes during combat to adapt their strategy, with each mode offering distinct attributes such as damage output, defense multipliers, movement speed, and visual effects.

Character State Management: Idle, Move, Attack, and Hit states with smooth transitions. Physics based movement with gravity and boundaries. Attack cooldowns and hit stun mechanics.

Input Handling: Dual keyboard input for two players (WASD/Arrows + action keys). Command pattern for flexible input mapping. Prevention of input conflicts between players.

Visual Feedback System: Particle effects for attacks (themed per mode), health bars with damage animations, mode indicators showing current stance, and victory screen overlay.

Game Flow Control: Pause/resume functionality, restart mechanism, and frame rate independent game loop.

Local Tournament Mode: Registration of multiple players by name, automatic match scheduling with elimination bracket structure, sequential match play, and progression through rounds to a final match.

3. User Characteristics

Primary Users

Casual gamers, children, young players, and university students interested in local multiplayer party games, understanding game development patterns, or quick accessible gaming sessions.

User Skill Levels

User Type	Characteristics	Needs
Casual Player	Familiar with basic keyboard controls, no technical knowledge required	Clear visual feedback, intuitive controls, minimal learning curve
Children / Young Players	Seeking fun, non-violent fighting game	Child friendly design, simple mechanics, colorful visuals
CS Student / Developer	Programming background, interested in design patterns	Well documented code, clear pattern implementations, extensible architecture
Game Enthusiast	Experienced with fighting games	Responsive controls, balanced gameplay, strategic depth

Accessibility Considerations

- No color blindness accommodations currently (uses distinct color coding per mode)
- Two player local only (requires physical presence)
- Keyboard only input (no gamepad/touch support)

4. Constraints

Platform Constraints

Constraint	Description
Browser Compatibility	Requires modern browsers supporting ES6+ and HTML5 Canvas (Chrome 90+, Firefox 88+, Edge 90+, Safari 14+)
Display Requirements	Minimum 1024×576 pixels viewport
Input Device	Physical keyboard required (mobile/touch not supported)

Technical Constraints

Constraint	Description
Language	TypeScript (strict mode enabled)
Build System	Vite (specified in package.json)
No Backend	Pure frontend application, no server communication
No Persistence	Game state is not saved between sessions
Zero Production Dependencies	No game engines, no UI frameworks, no utility libraries pure TypeScript + Browser APIs

Development Constraints

Constraint	Description
Academic Timeline	Designed for BIL 482 course project submission
Pattern Requirements	Must demonstrate at least 5 design patterns (Strategy, State, Observer, Command, Factory, Singleton, Object Pool)
Code Quality	Must maintain clean architecture principles
Team Size	Three person development team

Performance Constraints

Constraint	Target
Frame Rate	60 FPS on standard hardware
Input Latency	< 16ms (1 frame)
Memory	Efficient particle pooling to prevent memory leaks

5. System Features (Use Case Based)

5.1 Use Case Diagram

The following UML Use Case Diagram represents the primary system functionality. The system has two main actors (Player 1 and Player 2) who interact with four core use cases.

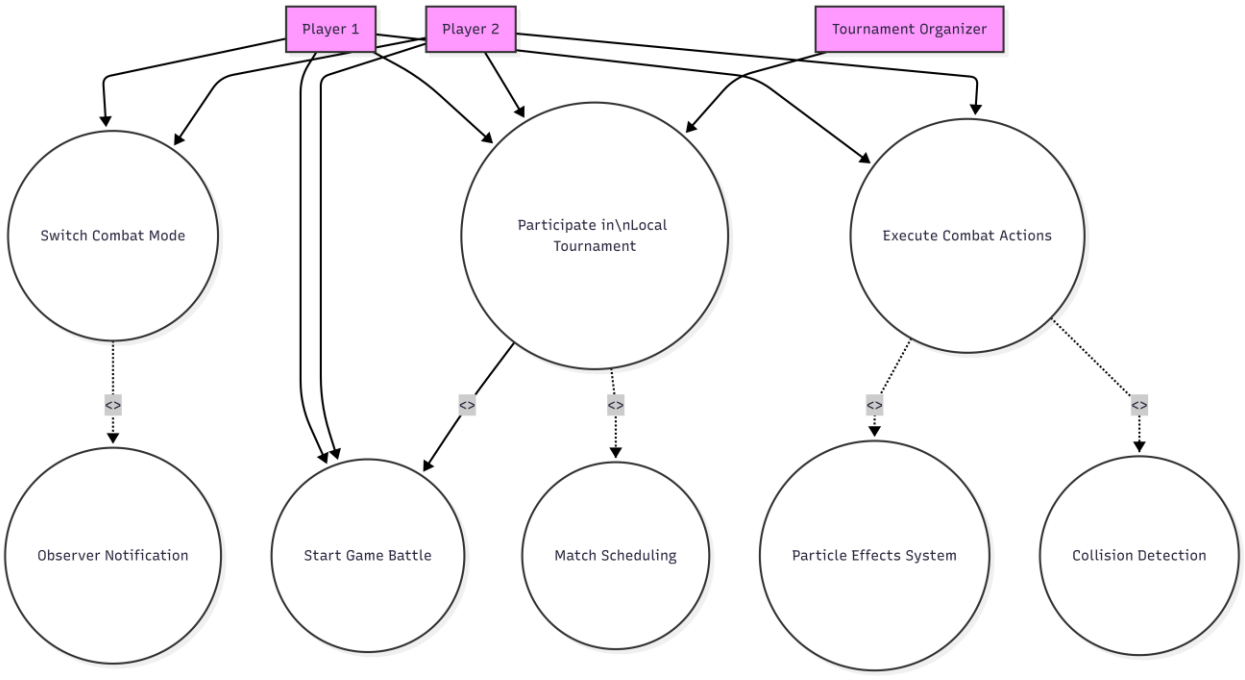
Actors: Player 1, Player 2

Use Cases:

- UC1: Start Game Battle – Both players initiate a game session
- UC2: Execute Combat Actions – Players perform movement, attacks, and defense
- UC3: Switch Combat Mode – Players toggle between their character's specific combat modes
- UC4: Participate in Local Tournament – Players register and compete in a tournament bracket

Included Sub-Systems:

- Collision Detection System (included by UC2)
- Particle Effect System (included by UC2)
- Observer Notification System (included by UC3)
- Match Scheduling System (included by UC4)



5.2 Use Case 1: Start Game Battle

Field	Description
Title	Start Game Battle
Main Actor	Player 1 and Player 2
Goal	Initialize and begin a fighting game session with two characters ready for combat

Preconditions

- User has opened the application in a web browser
- Browser supports HTML5 Canvas
- Canvas element is successfully rendered
- Both players have access to the keyboard

Main Flow

1. User navigates to the game URL in their browser.

2. System loads HTML page and executes main.ts entry point.

3. GameEngine singleton is instantiated with canvas reference.

4. CharacterFactory creates two Character instances (e.g., FireMage vs WaterGuardian, or default characters with specific modes).

5. Characters are initialized in their Primary Mode.

6. InputHandler registers keyboard bindings: Player 1 (WASD + Action Keys), Player 2 (Arrow keys + Action Keys).

7. GameHUD subscribes to both characters as Observer.

8. CollisionSystem initializes hitbox tracking.

9. ParticleSystem creates object pool for visual effects.

10. Game loop starts at 60 FPS.

11. System renders initial game state with both characters in idle state.

12. Control instructions are displayed on screen.

13. Both players can now input commands.

Postconditions

- Both characters are alive with full health (100 HP)
- Characters are in Idle state
- Game is in "Playing" state

- All systems are actively updating

Alternative / Exception Flows

Canvas not found: System logs error and game does not initialize.

Browser incompatibility: Canvas rendering fails; user is alerted to upgrade browser.

5.3 Use Case 2: Execute Combat Actions

Field	Description
Title	Execute Combat Actions
Main Actor	Player 1 or Player 2 (initiator)
Goal	Perform combat maneuvers including movement, attacking, and dealing/receiving damage

Preconditions

- Game is in "Playing" state
- Character is not defeated (health > 0)
- Character is not in hit stun state (for attack actions)

Main Flow – Movement Scenario

1. Player presses movement key (W/A/S/D or Arrow keys).

2. InputHandler captures keydown event and creates MoveCommand.

3. MoveCommand is executed; character inputFlags are updated.

4. Current state checks for transition; character transitions to MoveState.

5. MoveState.update() applies velocity based on input direction.

6. Character.update() applies gravity and checks boundaries.

7. Character position is updated and rendered at new position.

Main Flow – Attack Scenario

1. Player presses attack key (F or K).

2. InputHandler creates AttackCommand; checks cooldown and stun state.

3. Character transitions to AttackState; attack hitbox is created using current active mode.

4. CollisionSystem checks for hitbox intersection with opponent.

5. If hit detected: attacker's activeStrategy.attack() calculates damage based on mode multipliers.

6. Defender's activeStrategy.defend() calculates damage reduction (specific to their current mode).

7. Opponent's health is reduced; opponent transitions to HitState.

8. ParticleSystem spawns character-specific particles at hit location.

9. GameHUD HealthBar is notified via Observer pattern; health bar animates damage.

10. Attacker's cooldown starts; attacker returns to Idle after attack duration.

11. Defender exits HitState after hit stun duration expires.

Postconditions

- Character position is updated (for movement)
- Opponent health is reduced (if attack connected)
- Visual feedback displayed (particles, health bar animation)
- Attack cooldown and hit stun timers are active

Alternative / Exception Flows

Attack on cooldown: No state transition; character remains in current state.

Attack misses: No damage dealt; attack animation and cooldown still play.

Boundary reached: Position clamped to canvas bounds; velocity zeroed.

Fatal damage: Opponent health reaches 0; victory condition triggered.

5.4 Use Case 3: Switch Combat Mode

Field	Description
Title	Switch Combat Mode
Main Actor	Player 1 or Player 2
Goal	Toggle character's combat strategy between available modes (e.g., Attack Focus ↔ Defense Focus)

Preconditions

- Game is in "Playing" state
- Character is not currently attacking
- Character is not in hit stun
- Player has not held the switch key (single press detection)

Main Flow

1. Player presses switch mode key (G for Player 1, L for Player 2).

2. InputHandler detects keydown and checks switchModePressed flag to prevent repeat.

3. SwitchModeCommand is created and executed.

4. Character checks preconditions (not attacking, not stunned).

5. Character toggles active strategy reference to the next available mode (e.g., Mode A → Mode B).

6. New mode properties take effect immediately (Updated damage/defense/speed multipliers).

7. Character notifies all Observers via Observer pattern with ModeChangeEvent.

8. GameHUD updates mode indicator (e.g., changing icon from 🔥 to 💧 to ⚔️).

9. Character visual rendering updates (glow color, primary color).

10. On next attack/defense, new mode statistics are applied.

Postconditions

- Character's activeStrategy reference points to new mode
- Combat statistics changed according to new mode
- Visual appearance reflects new mode
- GameHUD displays updated mode indicator
- All Observers have been notified

Alternative / Exception Flows

Character is attacking or hit stunned: Mode switch exits early; no change occurs.

Key held down: InputHandler prevents repeated toggling.

5.5 Use Case 4: Participate in Local Tournament

Field	Description
Title	Participate in Local Tournament
Main Actor	Players (2 or more participants)
Goal	Register participants and compete through a structured elimination bracket until a tournament winner is determined

Note: Tournament mode is a planned feature defined in the Project Definition Document. It is currently under development and will be implemented as part of the project roadmap.

Preconditions

- Application is loaded in the browser
- At least 2 players are available to participate

- Players have access to the keyboard
- Game is in "Menu" state (tournament not yet started)

Main Flow

1. Players select "Tournament Mode" from the main menu.

2. System displays the player registration screen.

3. Each participant enters their name into the registration form.

4. Once all players are registered, the organizer confirms the participant list.

5. System generates a tournament bracket (elimination format) based on the number of participants.

6. System displays the bracket and schedules the first match.

7. The two players for the current match take positions at the keyboard.

8. A standard game battle is initiated (see UC1: Start Game Battle).

9. Players compete using combat actions (see UC2) and mode switching (see UC3).

10. When a player wins the match, the system records the result and updates the bracket.

11. System advances the winner to the next round.

12. Steps 7–11 repeat for each scheduled match until the final match is played.

13. System declares the tournament winner and displays the final results screen.

Postconditions

- A tournament winner has been declared
- The complete bracket with all match results is displayed
- Players can choose to start a new tournament or return to the main menu

Alternative / Exception Flows

Fewer than 2 players: System requires a minimum of 2 participants to start.

Player disconnects (closes tab): Match is forfeited; opponent advances.

Odd number of players: One player receives a bye and advances automatically to the next round.

6. Non-Functional Requirements

6.1 Usability

Requirement	Description	Metric
Learning Curve	New players should understand basic controls within 1 minute	Time to first intentional attack < 60s
Visual Clarity	Health bars, mode indicators, and effects must be immediately recognizable	90% can identify mode/health without instruction
Control Responsiveness	Input commands should feel instantaneous	Input-to-visual latency < 16ms (1 frame)
Instructions	On-screen control guide visible at all times	Keyboard shortcuts displayed in HUD
Error Prevention	No invalid state transitions possible	State machine prevents illegal transitions

6.2 Performance

Requirement	Target	Acceptance Criteria
Frame Rate	60 FPS	Maintain ≥55 FPS on mid-range hardware
Input Latency	< 16ms	Keypress to state change < 1 frame
Particle Count	100+ particles	No FPS drop below 50 with max particle load
Memory Usage	< 100MB heap	No memory leaks during 30+ min play
Load Time	< 2 seconds	Page load to playable state
Delta Time	Frame-rate independent	Identical behavior at 30–120 FPS

6.3 Portability

Platform	Support Level	Notes
Windows	Fully Supported	Primary development platform
macOS	Fully Supported	Tested on Safari and Chrome
Linux	Fully Supported	Tested on Firefox and Chromium
Mobile/Tablet	Not Supported	No touch controls, keyboard required

Browser Compatibility: Chrome 90+, Firefox 88+, Edge 90+, Safari 14+. Internet Explorer is not supported.

6.4 Reliability

Aspect	Requirement	Measure
Crash Resistance	No crashes during normal gameplay	MTBF > 10 hours
State Consistency	Game state always valid	All state transitions validated
Error Handling	Graceful degradation	Try-catch in initialization, console logging
Recovery	Restart always works	R key restart succeeds 100% from GameOver
Tab Switching	Survives tab switch	Game pauses on tab blur event
Memory Leaks	No accumulating memory	Heap stable after 1000 particles spawned/destroyed

7. External Interface Requirements

7.1 User Interfaces

Main Game Canvas

- Dimensions: 1024×576 pixels (16:9 aspect ratio)
- Rendering: HTML5 Canvas 2D context
- Background: Gradient from dark purple (#2d1b4e) to dark blue (#1a1a2e)
- Arena: Ground line at y=480 with visual decorations

HUD Elements

Element	Position	Description
Player 1 Health Bar	Top-left	Red bar, 300px width, shows damage animations
Player 2 Health Bar	Top-right	Blue bar, 300px width, mirrored from P1
Player 1 Mode Indicator	Below P1 health bar	Circle with Fire/Water emoji and color
Player 2 Mode Indicator	Below P2 health bar	Circle with Fire/Water emoji and color
Center Title	Top-center	"⚔ FIGHT ⚔"
Control Instructions	Bottom-center	Keyboard shortcuts, small font

Victory Screen Overlay

Triggered when a player's health reaches 0. Displays a semi-transparent black overlay with winner announcement text, player color theme, and glowing effects.

Tournament Screens (Planned)

- Player Registration Screen: Input fields for participant names with add/remove functionality.
- Tournament Bracket View: Visual bracket showing all scheduled matches and results.
- Match Result Screen: Displays winner of each match with option to proceed to next match.
- Tournament Winner Screen: Final celebration screen displaying the overall tournament champion.

7.2 Software Interfaces

Browser APIs

API	Purpose	Usage
Canvas 2D Context	Rendering	canvas.getContext("2d") for all drawing operations
requestAnimationFrame	Game loop	Smooth 60 FPS updates synced with display refresh
KeyboardEvent	Input	keydown/keyup listeners for player controls
Console API	Debugging	Logging startup info, errors, warnings

Design Pattern Interfaces

Pattern	Interface	Location
Strategy	IModeStrategy	/src/patterns/strategy/IModeStrategy.ts
State	ICharacterState	/src/patterns/state/ICharacterState.ts
Command	ICommand	/src/patterns/command/Command.ts
Observer	IObserver<T>, ISubject<T>	/src/patterns/observer/Observer.ts
Factory	CharacterFactory	/src/patterns/factory/CharacterFactory.ts
Object Pool	ObjectPool<T>	/src/patterns/pool/ObjectPool.ts

Build System Interface

Vite Configuration: Development server on port 5173 with HMR enabled, TypeScript compilation integrated.

Package Scripts: "dev" (vite), "build" (tsc && vite build), "preview" (vite preview).

DevDependencies: typescript ~5.6.2, vite ^6.0.5, @types/node ^22.0.0. Zero production dependencies.

8. Software Architecture

8.1 High Level Architecture

Color Clash follows a layered monolithic architecture with clear separation of concerns organized by design patterns. The architecture is structured as a single page application with a pattern driven game engine core.

Architectural Layers

Layer	Responsibility	Key Components
Presentation Layer	User interface and visual rendering	HTML5 Canvas, GameHUD, HealthBar
Game Engine Core	Central orchestration (Singleton)	GameEngine, Game Loop, Renderer
Game Systems Layer	Domain specific subsystems	InputHandler (Command), CollisionSystem, ParticleSystem (Object Pool)
Entity Layer	Core game objects	Character (Strategy+State+Observer), Hitbox, Particle
Pattern Implementations	Reusable design pattern abstractions	Strategy, State, Observer, Command, Factory, Object Pool

Architecture Diagram Description

The system uses a top-down layered architecture:

1. Presentation Layer: HTML5 Canvas UI → GameHUD → HealthBar Components
2. Game Engine Core (Singleton): GameEngine → Game Loop (60 FPS) → Rendering System
3. Game Systems Layer: InputHandler (Command Pattern) | CollisionSystem | ParticleSystem (Object Pool)
4. Entity Layer: Character Entities (Strategy + State) | Hitbox | Particle
5. Pattern Implementations: Strategy (Mode A/Mode B) | State (Idle/Move/Attack/Hit) | Observer | Command | Factory | Pool

Communication flows: HTML loads Engine → Engine drives Game Loop → Loop calls Input → Characters → Particles → Renderer. Input creates Commands. Characters use Strategy and State patterns. Characters notify Observers (HUD/HealthBar). Engine creates Characters via Factory. ParticleSystem uses Object Pool.

8.2 How Architecture Facilitates Use Cases

UC1: Start Game Battle: Singleton GameEngine ensures single instance initialization. Factory Pattern creates diverse character instances. Observer Pattern connects UI to entities. InputHandler registers bindings. Object Pool pre-allocates particles.

UC2: Execute Combat Actions: Command Pattern translates keypresses into actions. State Pattern manages character transitions. Strategy Pattern applies mode specific damage/defense (extensible for multiple characters). CollisionSystem detects hits. Observer notifies UI.

UC3: Switch Combat Mode: Strategy Pattern allows dynamic switching of character-specific combat behaviors. Command Pattern handles switch input. Observer Pattern notifies HUD. State Machine validates mode switch is allowed.

UC4: Participate in Local Tournament: Tournament Manager orchestrates the bracket and match scheduling. Factory Pattern creates fresh Character instances per match. GameEngine restart mechanism resets state between matches. Observer Pattern notifies tournament UI of match results.

8.3 Architectural Benefits

Design Goal	Architectural Solution
Modularity	Layered architecture with clear interfaces between layers
Extensibility	Strategy pattern allows unlimited new character modes without changing Character class
Testability	Decoupled systems, dependency injection of observers
Performance	Object pooling, efficient collision detection, delta-time game loop
Maintainability	Pattern-based organization, single responsibility classes
Scalability	Easy to add new characters, states, commands, or particle types

8.4 Design Patterns Summary

Pattern	Usage	Location	Purpose
Singleton	1 instance	GameEngine	Ensure single game orchestrator
Strategy	Multiple strategies	IModeStrategy implementations	Define unique combat behaviors per mode/character
State	4 states	IdleState, MoveState, AttackState, HitState	Manage character behavior transitions
Observer	2 observers	HealthBar, GameHUD	Decouple UI from game logic
Command	3 commands	MoveCommand, AttackCommand, SwitchModeCommand	Encapsulate input actions
Factory	1 factory	CharacterFactory	Centralize character

			creation with specific configs
Object Pool	1 pool	ParticleSystem for Particle	Reuse particles for performance