**SPRING 2025**

**COMP 302**

***KU Tower Defense***

**Final**

TeamD

Ahmet Taha Ekim - 79234

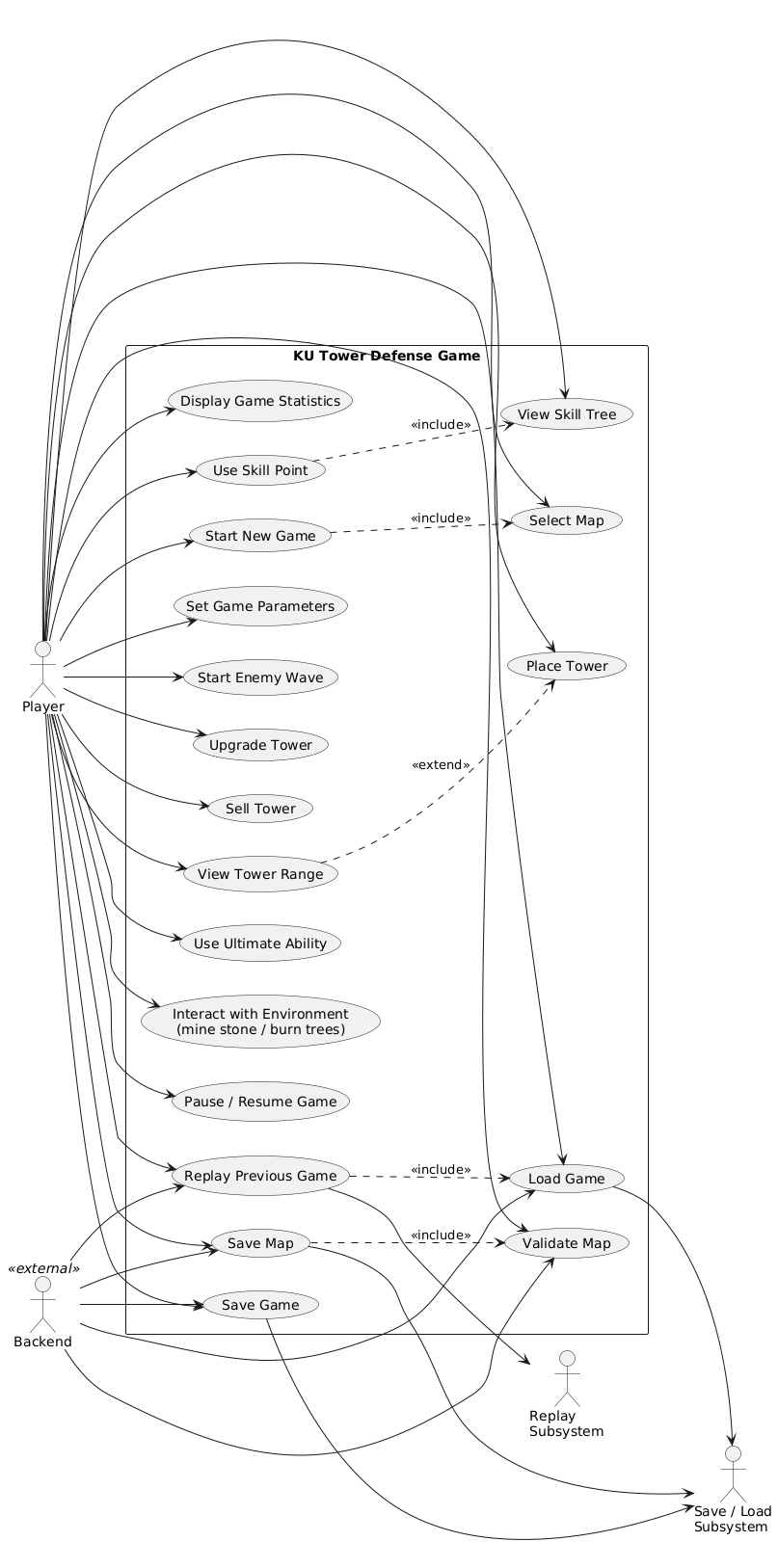
Muhammed Talha Sarıtepe - 79264

Mert Tuna – 80363

Doğu Arslan – 76649

Emir Göcen – 79634

# **Use Case Diagram**



***Figure 1: Use Case Diagram***

# **Use Case Narratives**

### **Use Case 1: Start New Game**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants to quickly start a new game.
* **Game System**: Must trigger transition to level selection and handle state change.

**Preconditions:**

* Game is in Main Menu scene.
* At least one map exists under resources/Levels/.

**Postconditions:**

* Game state changes to LEVEL\_SELECTION.
* LevelSelectionScene is rendered and functional.

**Main Success Scenario:**

1. Player clicks the “New Game” button in Menu.
2. SetGameState(GameStates.LEVEL\_SELECTION) is called.
3. LevelSelectionScene is initialized and displayed.
4. Available maps are listed dynamically.
5. Player proceeds to select a map (See Use Case 2).

**Extensions:**

* 1a. No maps found: System disables map selection UI or shows a warning.

**Special Requirements:**

* Seamless state transition without delay.
* Menu.mousePressed and Menu.update must handle this without lag.

### **Use Case 2: Select Map**

**Scope:** KU Tower Defense  
 **Level:** Subfunction  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants to choose a playable map.
* **Game System**: Should load valid map files and avoid crashes.

**Preconditions:**

* Game state is LEVEL\_SELECTION.
* Level files exist in resources/Levels.

**Postconditions:**

* A valid map is loaded and game switches to PLAYING scene.

**Main Success Scenario:**

1. LevelSelectionScene lists all .json maps.
2. Player clicks a map thumbnail or name.
3. LevelData.loadLevel(String mapName) is invoked.
4. If map is valid, it is queued and passed to Playing.
5. Game screen and logic are initialized with selected map.

**Extensions:**

* 3a. File not found or invalid format: Show popup error and remain in scene.
* 2a. Player presses back: Return to Main Menu with SetGameState(MENU).

**Special Requirements:**

* Live file listing; map directory is read each time scene is loaded.

### **Use Case 3: Place Tower**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants to place towers tactically.
* **Game System**: Should ensure resources and place valid towers.

**Preconditions:**

* Game is in PLAYING state.
* A tile is empty and buildable.
* Player has sufficient gold.

**Postconditions:**

* Tower is placed on tile.
* Gold is deducted.
* Tower becomes active.

**Main Success Scenario:**

1. Player clicks an empty tile.
2. Playing.setSelectedTile(Tile tile) highlights it.
3. Tower menu opens.
4. Player selects a tower type.
5. System checks if playerGold >= towerCost.
6. Tower is instantiated and added to tower list.
7. Gold is reduced accordingly.

**Extensions:**

* 3a. Player exits menu: Action is cancelled.
* 4a. Not enough gold: Show showWarning("Insufficient Gold").

**Special Requirements:**

* Only one tower per tile (Tile.isOccupied()).
* UI menu must be responsive even during wave spawn.

### **Use Case 4: Sell Tower**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants to regain gold from towers.
* **Game System**: Removes tower and processes refund.

**Preconditions:**

* A tower exists on a tile.

**Postconditions:**

* Tower is removed from list.
* Partial gold is refunded.
* Tile becomes buildable.

**Main Success Scenario:**

1. Player clicks a tower tile.
2. UI shows Sell Tower option in popup.
3. Player confirms sell.
4. Tower is removed from Playing.towers.
5. Refund is calculated: refund = basePrice \* refundRate.
6. Gold is added back to player balance.

**Extensions:**

* 2a. Player cancels menu: No changes made.
* 3a. Tower reference is invalid: Show error, do not crash.

**Special Requirements:**

* Refund depends on upgrade level.
* Consistent with stats in tower data JSON.

### **Use Case 5: View Tower Range**

**Scope:** KU Tower Defense  
 **Level:** Subfunction  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants visual feedback for range planning.
* **Game System**: Displays real-time overlays.

**Preconditions:**

* A tower is placed.

**Postconditions:**

* A visual range circle appears on tower hover or click.

**Main Success Scenario:**

1. Player clicks or hovers over a tower.
2. Playing.drawTowerRange(Graphics g) is called.
3. A translucent circle is drawn using tower.range.
4. When mouse moves away, overlay disappears.

**Extensions:**

* 1a. Range not loaded properly: Default range used.
* 2a. Graphics fail to render: Skip overlay, no crash.

**Special Requirements:**

* Overlay must match real attack range (in pixels).
* Must render within one frame for responsiveness.

### **Use Case 6: Set Game Parameters**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants to customize difficulty and game variables.
* **Game System**: Applies valid options and updates game behavior.

**Preconditions:**

* Game is in OPTIONS scene (GameStates.OPTIONS).

**Postconditions:**

* Options are stored in memory and reflected in future gameplay.

**Main Success Scenario:**

1. Player enters Options from main menu.
2. Options.initInputs() loads current values from config.
3. Player modifies values like wave delay, enemy health multiplier, etc.
4. Player clicks "Save".
5. System calls GameOptions.writeToFile() with validated input.
6. Confirmation is shown.

**Extensions:**

* 5a. Value is outside allowed bounds (e.g., delay < 0): Show error, block save.
* 6a. Disk write fails: Show Save Failed, keep current settings in memory.

**Special Requirements:**

* Inputs validated client-side via NumericInputField.validate().
* File path: resources/config/options.json.

### **Use Case 7: Save Game Options**

**Scope:** KU Tower Defense  
 **Level:** Subfunction  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants settings to persist.
* **Game System**: Ensures config durability across sessions.

**Preconditions:**

* Player has changed or reviewed values in the Options scene.

**Postconditions:**

* Config is saved as JSON and reloaded on startup.

**Main Success Scenario:**

1. Player presses “Save” after editing options.
2. Options.saveOptions() is called.
3. GameOptions.writeToFile() writes settings in JSON format.
4. A "Settings Saved!" message appears.
5. Settings are read back correctly on next launch via GameOptions.load().

**Extensions:**

* 2a. Validation error: Inputs highlighted in red.
* 3a. IOException: Show dialog, prevent file corruption.

**Special Requirements:**

* JSON format (key: value), human-readable and editable.

### **Use Case 8: Save Map File**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Player  
 **Supporting Actor:** File System

**Stakeholders and Interests:**

* **Player**: Wants to keep and replay custom maps.
* **System**: Must store playable, valid map structure.

**Preconditions:**

* Player is in Map Editor (MapEditing scene).
* Map contains start and end points and a valid path.

**Postconditions:**

* Map is saved under resources/Levels/ as a .json file.

**Main Success Scenario:**

1. Player clicks "Save" in map editor.
2. MapEditing.validateMap() checks for:  
   * Start and end tiles
   * Road connectivity
3. If valid, MapEditing.saveMapToFile() is called.
4. Player is prompted for a filename or given a default.
5. JSON is written to disk and appears in map list.

**Extensions:**

* 2a. Validation fails: List all errors and block save.
* 4a. File system error: Show alert and retry option.

**Special Requirements:**

* File must show in LevelSelectionScene.

### **Use Case 9: Validate Map**

**Scope:** KU Tower Defense  
 **Level:** Subfunction  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants assurance that map is playable.
* **System**: Prevents invalid maps from being saved.

**Preconditions:**

* Custom map is being edited.

**Postconditions:**

* Validation feedback is shown.
* Save button is enabled only if valid.

**Main Success Scenario:**

1. Player clicks "Validate" or "Save".
2. MapEditing.validateMap() checks:  
   * Presence of start and end
   * Continuous road path using BFS
3. If valid, allow save.
4. Otherwise, show errors like “Disconnected path” or “Missing start point.”

**Extensions:**

* 2a. Multiple issues: All listed in pop-up dialog.
* 2b. Blocking errors: Save disabled entirely.

**Special Requirements:**

* Validation occurs on every significant change (autoValidate=true).
* Visual indicators for issues (e.g., blinking tiles).

### 

### **Use Case 10: Start Enemy Wave**

**Scope:** KU Tower Defense  
 **Level:** System-level User Goal  
 **Primary Actor:** System Timer

**Stakeholders and Interests:**

* **Player**: Expects smooth progression of waves.
* **System**: Spawns enemies on time and applies delays from options.

**Preconditions:**

* Game is in PLAYING state.
* Grace period has ended or Start Wave button pressed.

**Postconditions:**

* Enemies spawn in timed intervals.
* Next wave loads after current ends.

**Main Success Scenario:**

1. Player starts a wave (or timer auto-triggers).
2. WaveManager.update() spawns enemy group after delay.
3. Enemies enter path at intervals based on GameOptions.waveDelay.
4. When all enemies are defeated, a new grace period starts.
5. Cycle repeats.

**Extensions:**

* 1a. Game is paused: Enemy spawning is suspended.
* 3a. Enemy queue fails: Wave restarts or fallback spawn used.

**Special Requirements:**

* Retry logic if spawn fails.
* Timing constants stored in GameOptions.

### 

### **Use Case 11: Upgrade Tower (R2M2D2)**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Player

**Stakeholders and Interests:**

* **Player**: Wants to strengthen towers to handle tougher enemies.
* **Game System**: Validates upgrades, maintains balance, and updates tower behavior.

**Preconditions:**

* A tower is already placed.
* Player has enough gold to afford the upgrade.

**Postconditions:**

* Tower stats are enhanced based on type.
* Player's gold is reduced.
* UI reflects upgraded state immediately.

**Main Success Scenario:**

1. Player clicks on an existing tower.
2. System shows tower UI with upgrade option and current level.
3. Player clicks "Upgrade".
4. Playing.upgradeTower(tower) is called.
5. System checks playerGold >= upgradeCost.
6. If true:  
   * Gold is deducted.
   * Tower stats are modified based on tower type:  
     + **Archer**: +50% range, 2x fire rate
     + **Artillery**: +20% range, +20% AOE
     + **Mage**: Adds slowing effect (20%, 4 sec)
   * Tower gets a visual indicator (badge, glow, animation).
7. Tower behavior updates in-game instantly.

**Extensions:**

* 1a. **Insufficient Gold**:  
  + Show warning popup: *"Not enough gold."*
  + Upgrade is cancelled.
* 2a. **Max Level Reached**:  
  + Upgrade button disabled.
  + Tooltip: *"Maximum upgrade reached."*

**Special Requirements:**

* Changes apply in real-time during ongoing waves.
* Visual upgrade feedback: particle, badge, or sound.
* Tower data (TowerManager, Tower.java) must track levels and effects accurately.
* All stat upgrades must reflect in projectile logic and tower targeting.

### **Use Case 12: Display Game Statistics on Game Over Screen(R2M2D2)**

**Scope:** KU Tower Defense  
 **Level:** User Goal  
 **Primary Actor:** Game System

**Stakeholders and Interests:**

* **Player**: Wants detailed feedback about performance.
* **System**: Closes the game loop and optionally offers replay or score submission.

**Preconditions:**

* Game has ended: player lost or completed all waves.

**Postconditions:**

* GameOverScene is activated and stats are visible.
* Player can choose to replay, exit, or submit score.

**Main Success Scenario:**

1. Game loop detects final wave completion or base health reaches 0.
2. GameStates.GAME\_OVER is triggered.
3. GameOverScene initializes and reads game stats from GameStatistics object.
4. The following values are shown:  
   * Total Gold Earned
   * Enemies Spawned
   * Enemies Reached End
   * Enemies Defeated
   * Total Damage Done
   * Time Played (seconds)
5. Player sees two buttons:  
   * “Main Menu” → resets game context.
   * “Replay Game” → restarts with the same map.
6. Clicking either routes the player accordingly.

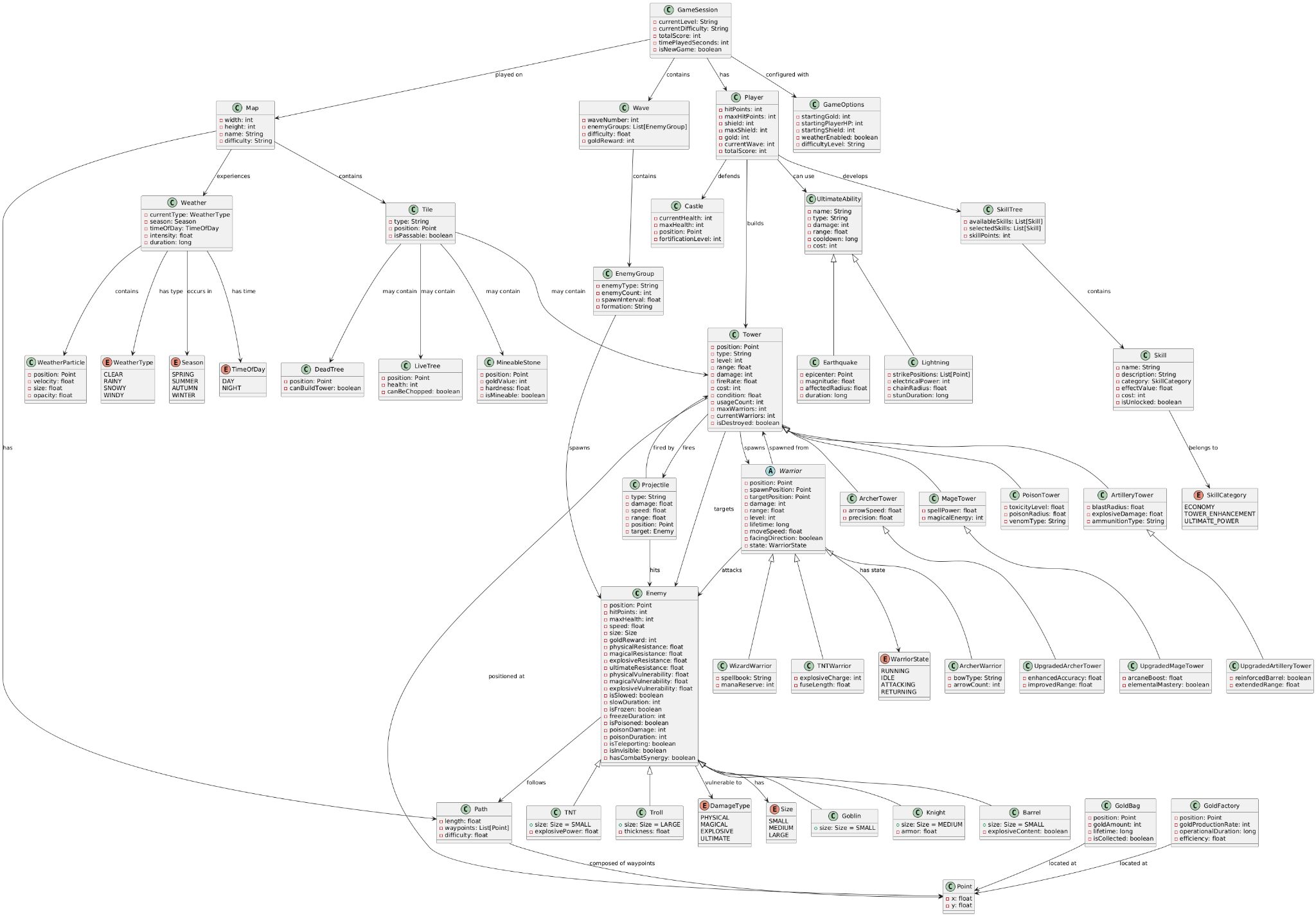
**Extensions:**

* 1a. **Victory Condition**:  
  + GameOverScene plays victory music and animation.
  + Optional confetti or badge.
* 1b. **Defeat Condition**:  
  + Different music, color palette, and message ("You were overwhelmed").
* 2a. **Submit Score (Optional Feature)**:  
  + "Submit Score" button appears.
  + Calls ScoreUploader.upload(stats) if available.
  + Success or failure is shown.

**Special Requirements:**

* Stats must sync with in-game counters (GameStatsTracker, EnemyManager, TowerManager).
* UI layout should be responsive and adaptive to resolution.
* Background audio, visuals, and transitions must match win/lose state.
* Data should optionally be saved for replay or leaderboard use.

# **Domain Model**

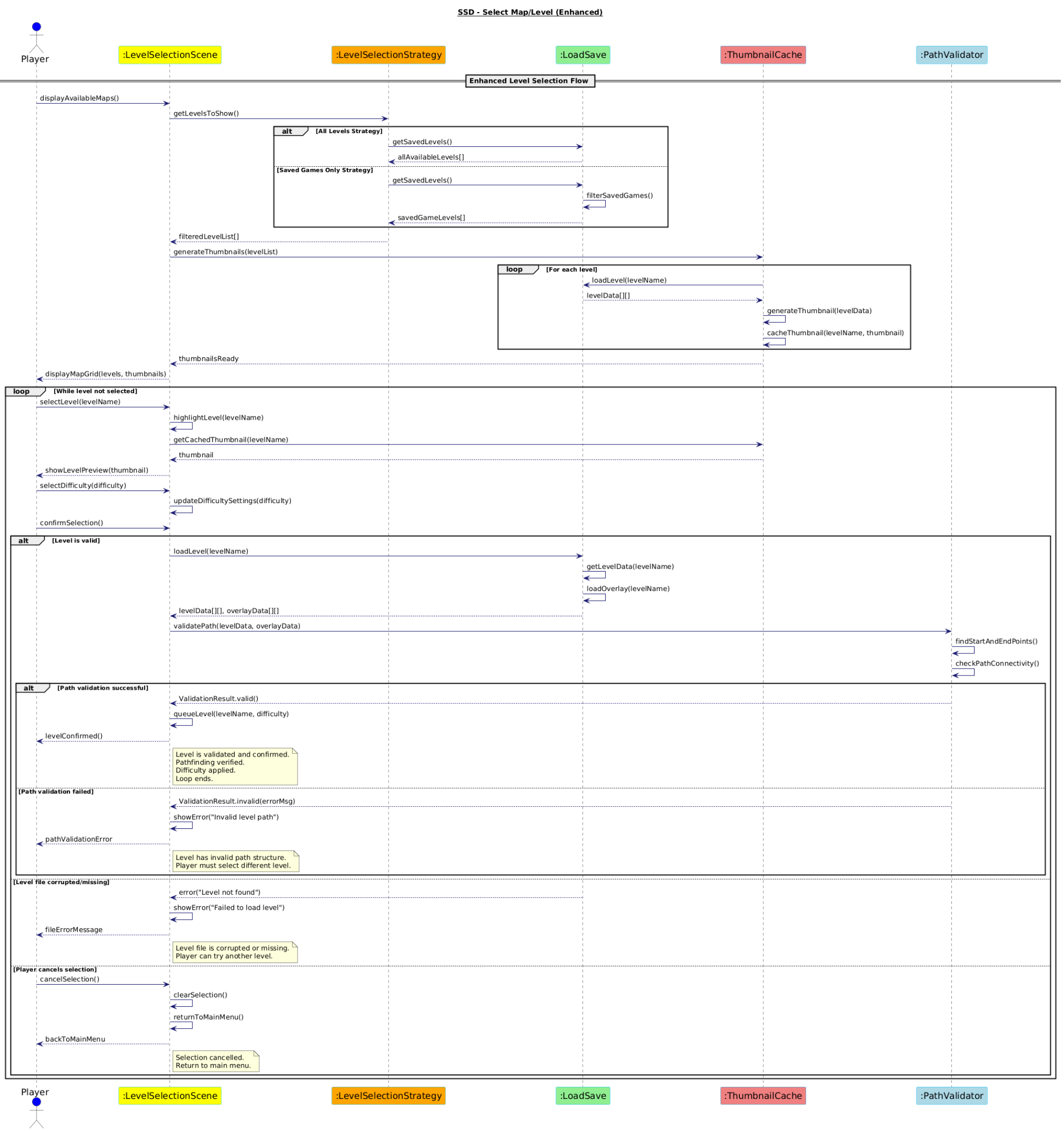


***Figure 2: Domain Model Diagram***

# **System Sequence Diagrams**

### **SSD 1: Start New Game**

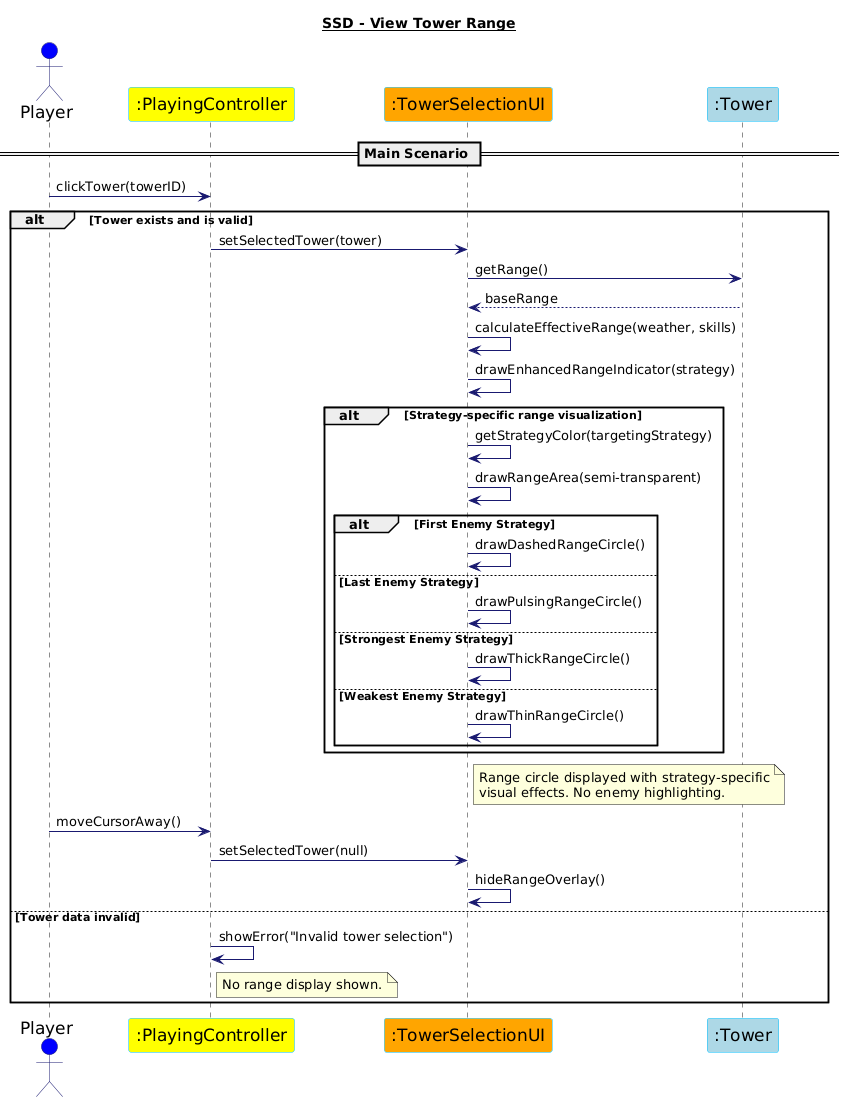
### **SSD 2: Select Map**



### **SSD 3: Place Tower**

### **SSD 4: Sell Tower**

### **SSD 5: View Tower Range**

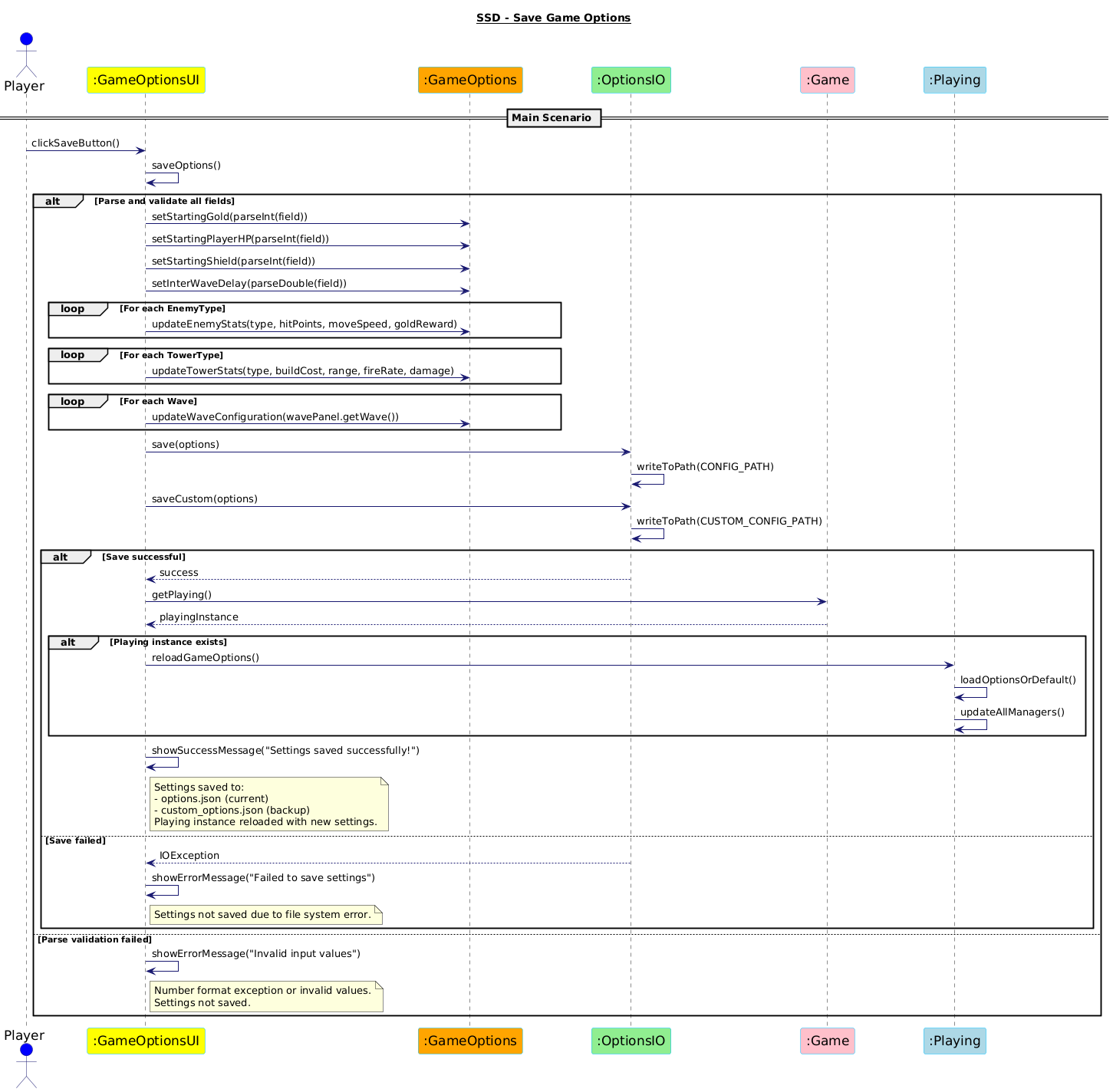


### 

### 

### **SSD 6: Set Game Parameters**

### **SSD 7: Save Game Options**

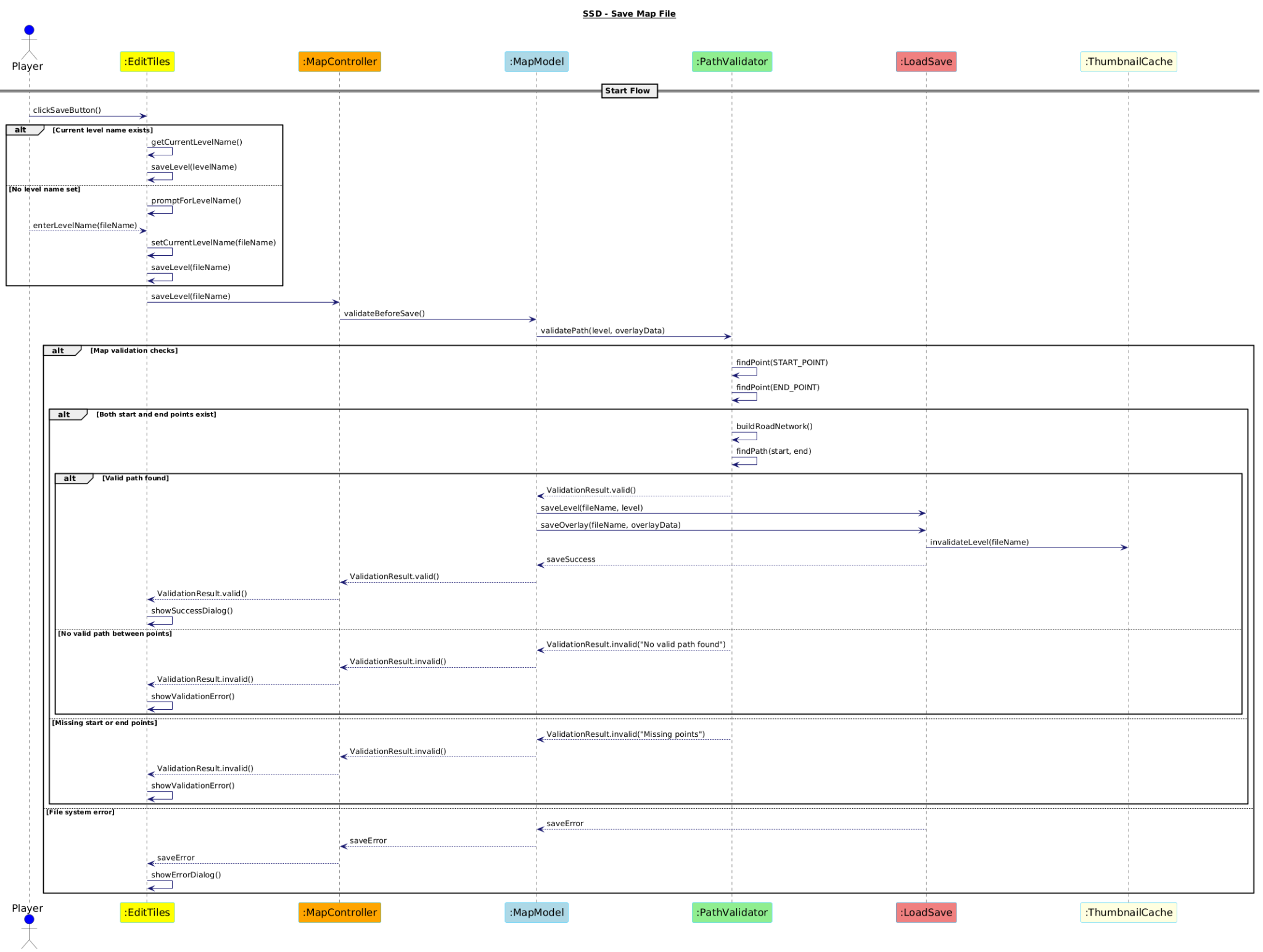


### 

### 

### 

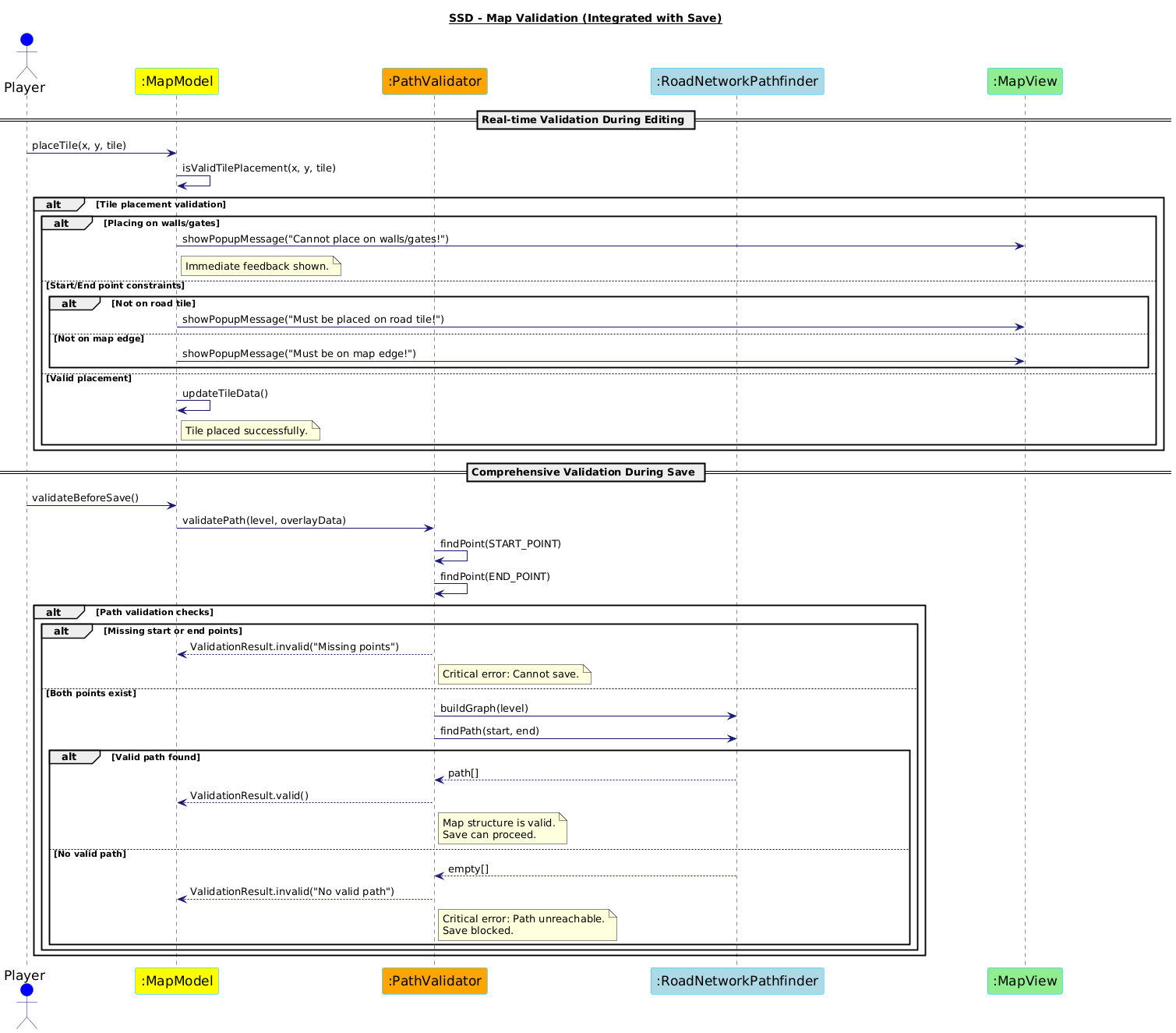
### **SSD 8: Save Map File**



### 

### 

### **SSD 9: Validate Map**



### 

### 

### 

### 

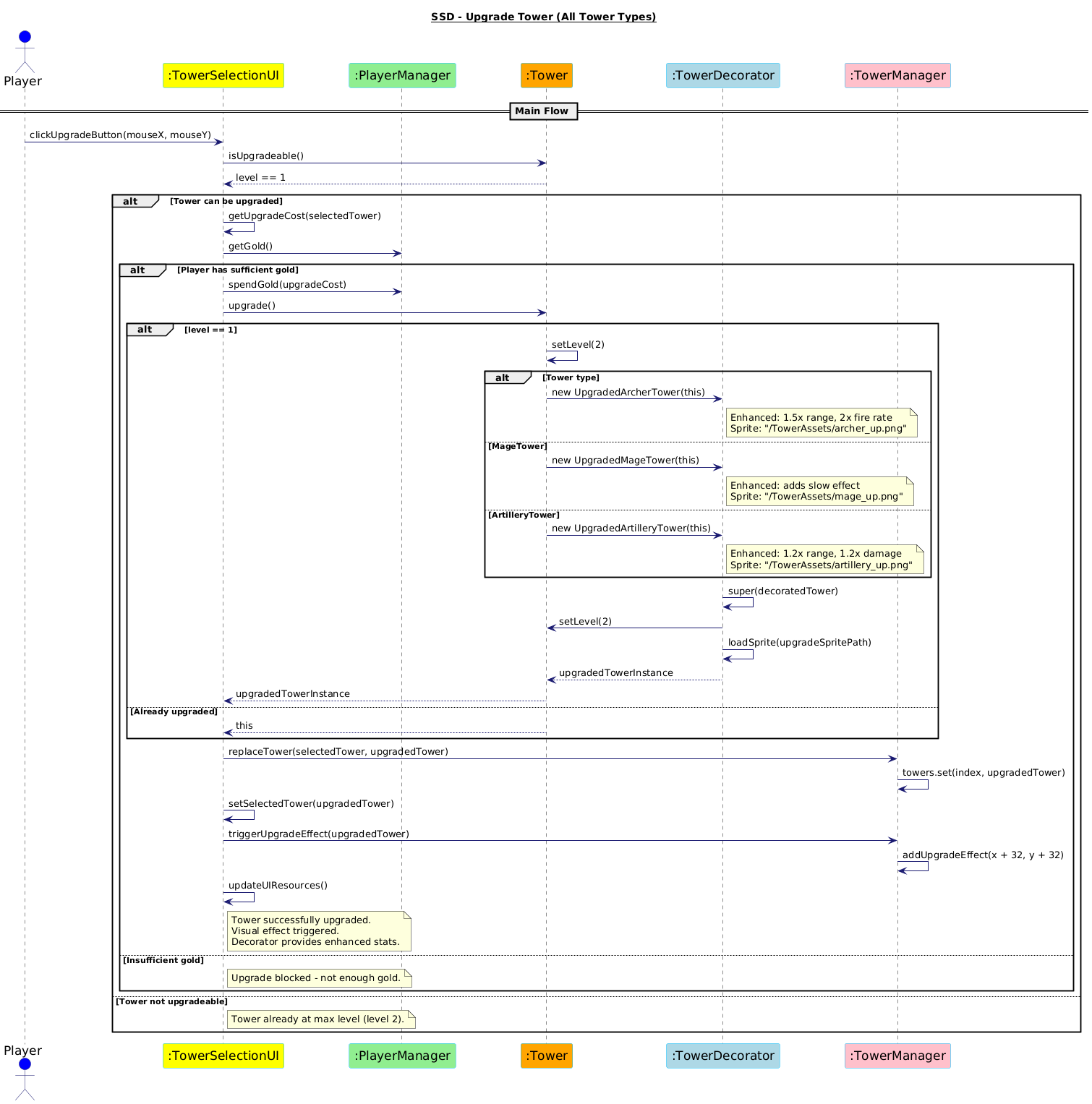
### 

### **SSD 10: Start Enemy Wave**

### 

### 

### **SSD 11: Upgrade Tower (R2M2D2)**

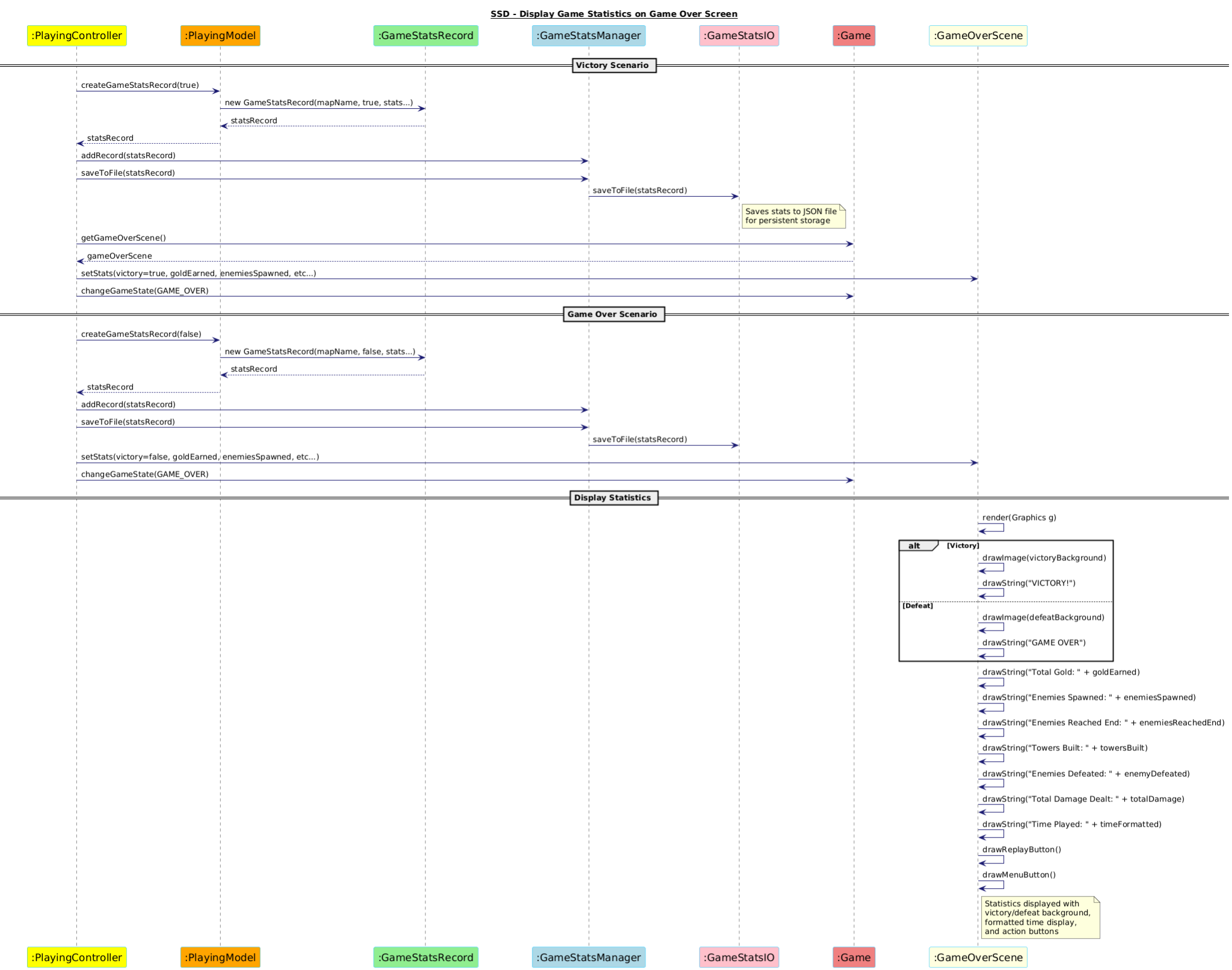


### 

### 

### 

### **SSD 12: Display Game Statistics on Game Over Screen(R2M2D2)**



# **Operation Contracts**

**Contract CO1-Updated — saveGameState(filename: String)**

**Preconditions**

1. Game is in a valid state for saving.
2. Player has initiated the save operation.
3. Game is not in a transition state (between waves or during animations).

**Postconditions**

1. Complete game state persisted, including:  
    • Player stats (health, shield, gold)  
    • Wave progress  
    • All tower states (condition, targeting strategy)  
    • Weather conditions and seasonal progression  
    • Skill-tree progress and selected skills  
    • Environmental-object states (DeadTrees, LiveTrees, MineableStones)  
    • Wave-start checkpoints for restart capability
2. Save file created with timestamp and metadata.

**Constraints**

1. Save file must be valid and uncorrupted.
2. All manager states must be serialisable.
3. File system must be writable.

**Contract CO2 — clickNewGame()**

**Preconditions**

1. Player is on the Main Menu.
2. Game system is initialised.

**Postconditions**

1. List of available maps displayed.
2. Game flagged as new (not loaded).
3. Default game options loaded.

**Constraints**

1. Valid map files must be present.
2. Default configuration must exist.

**Contract CO3 — selectMap(mapFileName: String, difficulty: String)**

**Preconditions**

1. Player has chosen a map.
2. Difficulty level selected.
3. System has received the map file.

**Postconditions** If map is valid

* Map data loaded.
* Game options applied according to difficulty.
* Game screen and weather system initialised.
* Game-stat counters reset.  
   If map is invalid
* Error displayed; player prompted to select another map.

**Constraints**

1. Map file must contain valid paths and tiles.
2. Difficulty settings must exist.
3. Map version must match game version.

**Contract CO4 — loadMapData(mapFileName: String, gameOptions: GameOptions)**

**Preconditions**

1. Valid map file selected.
2. Game options available.

**Postconditions**

1. Map layout, tiles, paths and environmental objects loaded.
2. Environmental objects positioned.
3. Weather system and seasonal progression initialised.
4. Enemy and tower stats configured for difficulty.

**Constraints**

1. Map file correctly formatted.
2. Environmental objects at valid positions.
3. Waypoints must form a valid enemy route.

**Contract CO5-Updated — placeTower(towerType: TowerType, position: Point, targetingStrategy: TargetingStrategy)**

**Preconditions**

1. Player has selected a tower type.
2. Position is a valid placement tile (or DeadTree).
3. Player has sufficient gold.
4. Position is unoccupied.

**Postconditions**

1. Tower created (condition 100 %, usage 0).
2. Targeting strategy assigned.
3. DeadTree removed if applicable.
4. Tile data updated.
5. Player gold reduced.
6. Tower added to warrior-spawning system if applicable.

**Constraints**

1. Tower must be placeable at the position.
2. Cost calculated from game options.
3. Placement must not block paths.

**Contract CO6-Updated — updateGameOptions(gameOptions: GameOptions)**

**Preconditions**

1. Player has modified parameters.
2. Values are within acceptable ranges.

**Postconditions**

1. Enemy stats refreshed for existing and future enemies.
2. Tower stats and costs updated.
3. Weather system toggled per options.
4. Starting resources adjusted.
5. Difficulty effects applied.

**Constraints**

1. All values must be valid.
2. Existing game state must remain consistent.
3. Options must be storable.

**Contract CO8 — triggerWaveStart(waveNumber: int)**

**Preconditions**

1. Time to start a new wave.
2. Previous wave finished (or first wave).
3. Game not paused.

**Postconditions**

1. Wave-start snapshot captured.
2. Enemy groups queued.
3. Wave timers and counters initialised.
4. Player notified.

**Constraints**

1. Wave data must be valid.
2. No active pause.
3. Prior wave’s enemies cleared.

**Contract CO9 — wait(spawnInterval: float, gameSpeedMultiplier: float)**

**Preconditions**

1. Group spawning in progress.
2. Spawn interval defined.

**Postconditions**

1. System waited for spawnInterval ÷ gameSpeedMultiplier.
2. Next enemy ready for spawn.

**Constraints**

1. Interval adjusted by speed multiplier.
2. Wait must pause if game is paused.

**Contract CO10-Updated — spawnEnemyGroup(enemyGroup: EnemyGroup, gameOptions: GameOptions)**

**Preconditions**

1. Wave active; group’s spawn time reached.
2. Enemy group data valid.
3. Spawn location clear.

**Postconditions**

1. Enemies spawned with stats from current options.
2. Resistances and vulnerabilities applied.
3. Group synergy effects initialised.
4. Size and movement set.
5. Enemies attached to path-finding system.

**Constraints**

1. Spawn location valid.
2. Stats loaded from options.
3. Valid movement path required.

**Contract CO11 — closeMapSelection()**

**Preconditions**

1. Map-selection menu open.
2. Player opts to close.

**Postconditions**

1. Menu closed; player returned to Main Menu.
2. No game initialisation performed.

**Constraints**

1. Menu must close cleanly.
2. Return to Main Menu must succeed.

**Contract CO12-Updated — checkPlayerStatus()**

**Preconditions**

1. Game state needs evaluation.

**Postconditions**

1. Player health and shield assessed.
2. Castle health verified.
3. Game Over triggered if health ≤ 0 and shield ≤ 0.
4. Victory triggered if final wave complete and no enemies remain.
5. Game statistics updated.

**Constraints**

1. Health and shield tracking accurate.
2. Victory and game-over mutually exclusive.

**Contract CO13 — clickGameOverScreen()**

**Preconditions**

1. Game-Over screen displayed.
2. Player clicks.

**Postconditions**

1. Statistics recorded.
2. Player choice (restart, main menu, exit) processed.

**Constraints**

1. Screen must be active.
2. Click must be valid.

**Contract CO14 — resetGameState()**

**Preconditions**

1. Player chooses to restart.
2. A current session exists.

**Postconditions**

1. All variables reset to defaults.
2. Managers re-initialised.
3. Map data reloaded.
4. Environmental objects restored.
5. Weather reset.
6. Main Menu displayed.

**Constraints**

1. Complete reset required.
2. No residual state.
3. No memory leaks.

**Contract CO15 — closeGame()**

**Preconditions**

1. Player decides to exit.
2. Game can close safely.

**Postconditions**

1. Resources cleaned.
2. Unsaved progress handled.
3. Game process terminated cleanly.

**Constraints**

1. Threads and timers terminated.
2. File handles closed.
3. Memory released.

**Contract CO16 — clickPauseButton()**

**Preconditions**

1. Game running and pauseable.
2. Player initiates pause.

**Postconditions**

1. Game updates halted.
2. Pause menu prepared.
3. Game speed multiplier set to 0.

**Constraints**

1. Game must be in playable state.

**Contract CO17 — returnToMainMenu()**

**Preconditions**

1. Player opts to leave current game.
2. Game paused or interruptible.

**Postconditions**

1. Session ended; progress saved or discarded per choice.
2. Main Menu displayed.
3. Resources cleaned.

**Constraints**

1. Confirmation required if progress may be lost.
2. Transition must be clean.

**Contract CO18 — gameContinues()**

**Preconditions**

1. Game is paused.
2. Player selects resume.

**Postconditions**

1. Game resumes from exact previous state.
2. Timers and animations continue.
3. Speed multiplier restored.
4. Pause menu closed.

**Constraints**

1. All elements must resume correctly.

**Contract CO\_Weather1 — updateWeatherState(gameTime: long, speedMultiplier: float)**

**Preconditions**

1. Weather system is enabled.
2. Game is running.

**Postconditions**

1. Weather type advanced based on season and game-time progression.
2. Day/night cycle progressed.
3. Weather effects and particle systems updated.

**Constraints**

1. Seasonal rules must be obeyed.
2. Effects applied consistently.
3. Particle workload must remain within performance limits.

**Contract CO\_Weather2 — applyWeatherEffects(weatherType: WeatherType)**

**Preconditions**

1. A weather type is currently active.

**Postconditions**

1. Enemy speed, tower range and archer accuracy adjusted according to the active weather.
2. Corresponding visual effects displayed.

**Constraints**

1. Effects must be reversible when weather changes.
2. Multiple weather effects must not stack improperly.

**Contract CO\_Warrior1 — spawnWarrior(tower: Tower, targetX: int, targetY: int, warriorType: WarriorType)**

**Preconditions**

1. Tower can spawn warriors (current < max).
2. Spawn and target positions are valid.
3. Tower is intact.

**Postconditions**

1. Warrior spawned with a 30 s lifetime.
2. Tower’s warrior count incremented (+1).
3. Warrior state set to *RUNNING*.

**Constraints**

1. Maximum warriors per tower enforced.
2. Spawn distance limits respected.
3. Warrior lifetime correctly tracked.

**Contract CO\_Warrior2 — updateWarriorState(warrior: Warrior, speedMultiplier: float)**

**Preconditions**

1. Warrior is active.
2. Game is not paused.

**Postconditions**

1. Warrior position updated.
2. State transitions handled (RUNNING → IDLE → ATTACKING → RETURNING).
3. Lifetime decremented; warrior removed on expiry, tower count decremented.

**Constraints**

1. Transitions must be logical.
2. Timers respect game-speed multiplier.
3. Resources fully freed on removal.

**Contract CO\_Skill1 — selectSkill(skillType: SkillType, player: Player)**

**Preconditions**

1. Skill is available and prerequisites met.
2. Player has sufficient skill points.

**Postconditions**

1. Skill marked as selected.
2. Skill effects applied immediately.
3. Skill points deducted.
4. Dependent skills unlocked if conditions met.

**Constraints**

1. Selection is permanent.
2. Effects applied globally and consistently.

**Contract CO\_Skill2 — applySkillEffects(selectedSkills: List<SkillType>)**

**Preconditions**

1. Selected skills list is populated.

**Postconditions**

1. Economy bonuses (gold, interest, starting resources) applied.
2. Tower enhancements (damage, range, special abilities) applied.
3. Ultimate-ability bonuses (cool-down, potency) applied.

**Constraints**

1. Stacking and override rules obeyed.
2. Performance impact kept minimal.

**Contract CO\_Ultimate1 — useUltimateAbility(abilityType: UltimateType, targetArea: Area, player: Player)**

**Preconditions**

1. Cool-down expired.
2. Target area valid.
3. Player initiated ability use and has required resources.

**Postconditions**

1. Ability effect executed on target area.
2. Cool-down timer reset (including skill bonuses).
3. Visual and audio effects triggered.
4. Game statistics updated.

**Constraints**

1. Cool-down strictly enforced.
2. Area effects calculated accurately.

**Contract CO\_Ultimate2 — processEarthquake(epicenter: Point, magnitude: float, duration: long)**

**Preconditions**

1. Earthquake ability activated.
2. Epicenter coordinates valid.

**Postconditions**

1. Damage applied to all enemies within radius (distance-scaled).
2. Towers in radius may be destroyed per condition rules.
3. Debris and ground-shake visual effects shown.

**Constraints**

1. Damage scales correctly with distance.
2. Tower-destruction logic honoured.
3. Performance impact acceptable.

**Contract CO\_Ultimate3 — processLightning(strikePositions: List<Point>, chainRadius: float)**

**Preconditions**

1. Lightning ability activated.
2. Strike positions valid.

**Postconditions**

1. Primary lightning strikes executed.
2. Chain lightning propagated within radius; stuns applied.
3. Lightning visuals rendered.

**Constraints**

1. Chain radius respected.
2. Stun duration tracked accurately.
3. Multiple strikes coordinated properly.

**Contract CO\_Environment1 — interactWithEnvironmentalObject(objectType: EnvironmentalObjectType, position: Point, player: Player)**

**Preconditions**

1. Environmental object exists at position and is interactable.
2. Player initiated the interaction.

**Postconditions**

1. Interaction resolved based on object type (MineableStone, LiveTree, DeadTree).
2. Tile data updated.
3. Player resources adjusted accordingly.

**Constraints**

1. State changes are permanent.
2. Rewards balanced with gameplay.
3. Map integrity preserved.

**Contract CO\_Environment2 — updateEnvironmentalObjects(speedMultiplier: float)**

**Preconditions**

1. Environmental objects present on map.
2. Game is running.

**Postconditions**

1. Object states updated (growth, decay, expiry).
2. Temporary objects (GoldBags) removed when expired.
3. Visual state kept in sync.

**Constraints**

1. Object lifecycle properly managed.
2. Expired objects fully cleaned up.

### 

**Contract CO\_Resistance1 — calculateDamage(rawDamage: int, damageType: DamageType, enemy: Enemy)**

**Preconditions**

1. Damage event targeting an enemy.

**Postconditions**

1. Final damage calculated using resistance/vulnerability modifiers.
2. Enemy health reduced.
3. Combat feedback triggered if applicable.

**Constraints**

1. Formula consistent and balanced.
2. Minimum-damage thresholds respected.

**Contract CO\_Status1 — applyStatusEffect(enemy: Enemy, effectType: StatusEffectType, duration: int, intensity: float)**

**Preconditions**

1. Enemy is alive.
2. Effect parameters valid.
3. Effect applicable to enemy type.

**Postconditions**

1. Status effect applied and timer started.
2. Visual indicators activated.
3. Enemy behaviour modified according to effect.

**Constraints**

1. Multiple effects stack or override correctly.
2. Timers account for game-speed multiplier.

**Contract CO\_Status2 — updateStatusEffects(enemy: Enemy, speedMultiplier: float)**

**Preconditions**

1. Enemy has active status effects.
2. Game is not paused.

**Postconditions**

1. Effect timers decremented.
2. Periodic effects (e.g., poison) processed.
3. Expired effects removed; enemy stats restored.

**Constraints**

1. Timers accurate.
2. Modifications reversible.

**Contract CO\_Save1 — createGameStateMemento()**

**Preconditions**

1. Game is stable.
2. All managers initialised.

**Postconditions**

1. Snapshot captured, including: player, towers, enemies, weather, skills, wave/timing data.

**Constraints**

1. Snapshot must be consistent and serialisable.
2. Capture must not stall gameplay.

**Contract CO\_Load1 — restoreGameStateMemento(memento: GameStateMemento)**

**Preconditions**

1. Valid memento provided.
2. Game systems ready to accept state.

**Postconditions**

1. Entire game state restored atomically.
2. Managers re-initialised.
3. Visual state matches saved snapshot.

**Constraints**

1. Restore is all-or-nothing.
2. Save-format compatibility maintained.
3. No data corruption allowed.

# 

# **Class Diagrams**

Since the files are large we have added them as additional files in our submission.

# **Interaction Diagrams**

Since the files are large we have added them as additional files in our submission.

# 

# **Vision**

**1.1 Overview**

KUTowerDefense is an expandable tower-defense game that combines strategic base building with a dynamic environment. The final form introduces a day–night cycle, four-season weather engine, new enemy archetypes, and a player skill tree, delivering replayable depth and accessibility for a diverse player base.

**1.2 High‑Level Goals**

· Deliver engaging, session‑based gameplay with lasting progression.

· Achieve smooth performance targets: 60 FPS on mid‑range hardware (<100 ms input latency).

· Support modding and future expansion through data‑driven configuration.

· Maintain full localisation (Turkish & English) and colour‑blind accessibility.

**1.3 Key Features**

· Dynamic Day–Night Cycle & Seasonal Weather System

· Pre‑game Skill‑Tree layer (economy, tower, ultimate perks)

· New enemy archetypes: Barrel, TNT, Troll – each with unique behaviours

· Player Shield mechanic for strategic mitigation

· Detailed Statistics Scene with persistent match history

**2. Solution Description**

**2.1 Architecture Overview**

The codebase is organised into modular subsystems: Rendering, Physics, AI, UI, and Data. Gameplay rules are expressed in JSON; the Weather system injects shader parameters at runtime.

**2.2 Major Components**

· WeatherManager – orchestrates day–night & season shaders.

· SkillTreeManager – applies perk effects before waves spawn.

· EnemyFactory – spawns archetypes with data‑driven stats.

· ShieldSystem – absorbs damage before player health.

**2.3 Tunable Parameters**

All balancing variables are externalised in config/\*.json. Key parameters include:

· weatherPreset (summer, winter, etc.)

· shieldValue (int, default = 100)

· interWaveDelay (seconds)

· enemyHealthMultiplier (float)

**3. Supplementary Specification**

**3.1 Performance**

Target 60 FPS on mid‑range GPUs; garbage‑collection pauses < 5 ms.

**3.2 Reliability**

Crash‑recovery mementos saved every wave and on exit.

**3.3 Usability**

In‑game tutorial overlays and controller‑friendly input scheme.

**3.4 Portability**

Built with OpenJDK 17; verified on Windows, macOS, Ubuntu 22.04.

**3.5 Accessibility & Localisation**

Colour‑blind palette toggle; full Turkish/English localisation via i18n JSON bundles.

**4. Glossary**

| Term | Definition |
| --- | --- |
| Shield | Temporary buffer that absorbs incoming damage. |
| Weather System | Subsystem managing environmental effects (rain, snow, etc.). |
| Season | Preset defining weather patterns and visuals for a quarter of the in‑game year. |
| Skill Tree | Progression graph enabling players to unlock perks. |
| Statistics Scene | Menu showing historical performance metrics. |
| Inter‑Wave Delay | Pause between enemy waves during which players may build. |
| Barrel / TNT / Troll | Enemy archetypes with explosive, area‑damage, or regeneration behaviours. |

**5. Revision Log**

| Version | Date | Author | Key Changes |
| --- | --- | --- | --- |
| v1.0 | 27 Mar 2025 | Core Team | Initial draft of vision & specs. |
| v2.0 | 17 Jun 2025 | Core Team | Aligned with final source‑code features; added weather, skill tree, glossary, performance targets. |