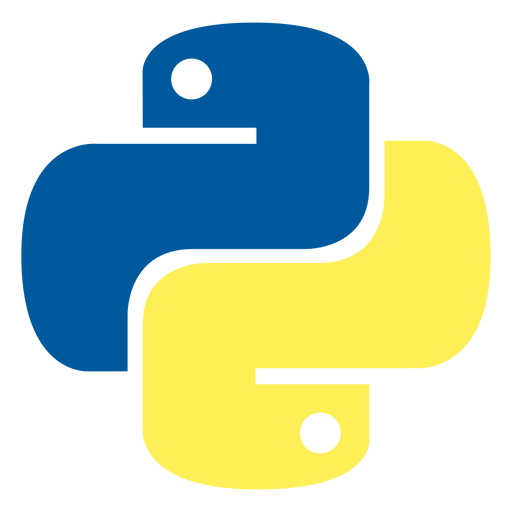
## Revisions

* The original design proposal for the tower defense game was mostly correct in its descriptions of game mechanics and programming requirements, however their implementation could be made more clear/organised.
  + State management has been suggested in this design.
* Error Handling/testing was not present in the previous design proposal, which has been added in this proposal.
* Pseudocode and flow chart extract are included in this proposal to provide further detail on how the program should be designed and on logic flow.
* System architecture is included in this proposal to guide the organization of modules in the program.
* The original design proposal did not mention ethical/legal considerations, included in this proposal.

## Ethical/Legal Considerations

* In the designing of my tower defense game I should ensure that all assets (e.g., images, music) used in the game are either original or properly licensed to avoid copyright infringement, and the ethical/legal consequences that follow.
* Fair gameplay should be prioritized by avoiding pay-to-win mechanics or manipulative design choices, to avoid associated ethical concerns.



Python logo - Python Software Foundation

## Pseudocode for checking grid tiles

* Demonstration of the ‘check\_tile’ method inside the Grid class; checks the number (tile) at given coordinates inside a 2D array, returning a label (string) representing the tile type.

CLASS Grid:

# Define constants for different tile types

DEFINE FREE\_TILE AS 0 # Represents a free (empty) tile

DEFINE PATH AS 1 # Represents a path

DEFINE TOWER AS 2 # Represents a tower

METHOD \_\_init\_\_(grid):

# **Encapsulation**: The grid is stored as a private attribute (data hiding)

SET self.\_grid TO grid

ENDMETHOD

METHOD check\_tile(grid\_coords):

# **Abstraction**: The method provides a meaningful string/label instead of raw numbers

# **Validation**: Ensure the input format is valid

IF grid\_coords IS NOT a tuple OR LENGTH(grid\_coords) IS NOT 2 THEN

RETURN "Invalid input" # Abstraction: Returns a user-friendly error message

ENDIF

# Extract x and y coordinates

SET grid\_x TO grid\_coords[0]

SET grid\_y TO grid\_coords[1]

# **Validation**: Ensure coordinates are valid (not None)

IF grid\_x IS None OR grid\_y IS None THEN

RETURN "Invalid coordinates"

ENDIF

# **Validation**: Check if coordinates are within grid boundaries

IF grid\_x < 0 OR grid\_x >= LENGTH(self.\_grid[0]) THEN

RETURN "Out of bounds" # Encapsulation: Prevents direct access to invalid memory

ENDIF

IF grid\_y < 0 OR grid\_y >= LENGTH(self.\_grid) THEN

RETURN "Out of bounds"

ENDIF

# Retrieve the tile value from the grid

SET tile\_value TO self.\_grid[grid\_y][grid\_x]

# **Validation**: Check tile value to ensure it's within expected types

IF tile\_value NOT IN {self.FREE\_TILE, self.PATH, self.TOWER} THEN

# **Error handling**: Prevents unexpected tile values from breaking the program

THROW ERROR "Invalid tile value!"

ENDIF

# Determine the tile type and return a string

IF tile\_value == self.PATH THEN

RETURN "path" # **Abstraction**: Converts raw numbers into meaningful names

ELSEIF tile\_value == self.TOWER THEN

RETURN "tower"

ELSEIF tile\_value == self.FREE\_TILE THEN

RETURN "free space"

ENDIF

ENDMETHOD

ENDCLASS

## Testing Strategies/Error Handling

**Error Handling:**

* Invalid User Inputs: Validate inputs (e.g., coordinates for tower placement) and provide clear error messages for invalid actions.
* File I/O Errors: Handle file-related issues (e.g., loading/saving game data) using try-except blocks to prevent crashes.
* Runtime Exceptions: Use try-except to catch unexpected errors (e.g., out-of-bounds grid access) and provide fallback error messages.
* General Error Feedback: Provide user-friendly error messages and log errors for easier debugging.

**Testing Strategies:**

* Unit Tests: Test individual components (e.g., tower placement, enemy movement) using Pytest to ensure correct behaviour.
* Integration Tests: Test interactions between components (e.g., placing towers, enemy movement) with Pytest to ensure seamless functionality.
* System Tests: Test the entire game flow (e.g., start-to-finish gameplay) to ensure the game works wholly.

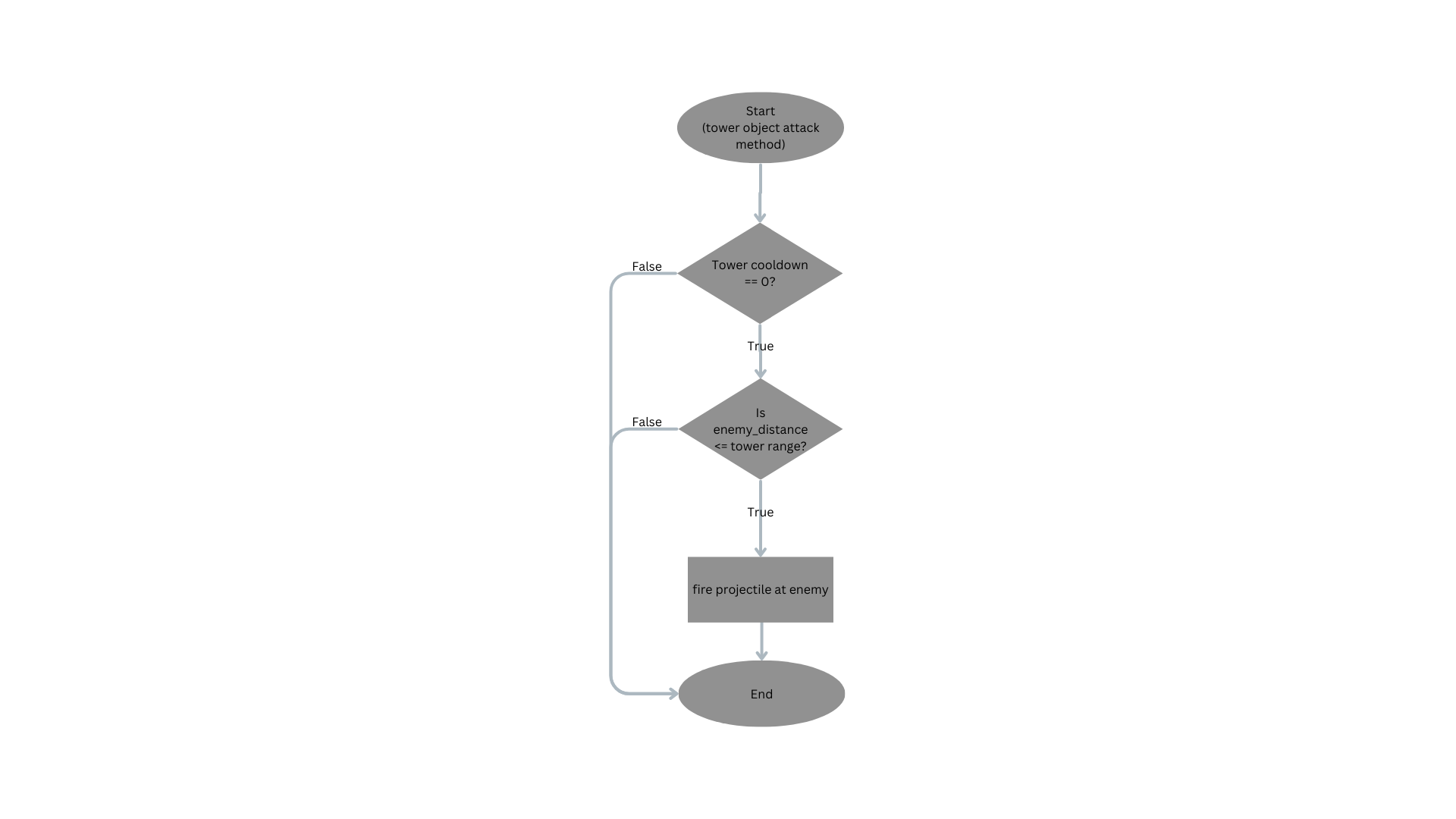
**Justification:**

* Error Handling ensures a smooth user experience and prevents crashes.
* Testing (unit, integration, system) ensures functionality, reliability, and performance, making the game maintainable and bug-free.



For example, this function ensures that the check\_tile method (as shown in pseudocode earlier) works as expected; i.e. it correctly returns that at the given grid and coordinates, a tower is present.

## Flowchart



* This is the attack method inside the base\_tower class.
* It checks for if:
  + The tower’s cooldown is 0 (meaning it can shoot)
  + The enemy\_distance is smaller than/equal to (within) the tower’s range, meaning the tower can reach it.
* If both conditions are satisfied, the tower will shoot (shoot is another method within the base\_tower class). Otherwise, the method ends there.
* This method is called every frame within the games update loop (in game\_state class; through update(self, events) → tower.update(self.enemies) → **self.attack(enemies)**)

## System Architecture

**/tower\_defense\_game**

**/Assets # Folder for all static assets used in the game**

**/Maps # Contains map files or configurations**

**/sprites # Holds image files for characters, enemies, and UI elements**

**/ui # UI-related assets like button graphics, menu backgrounds, etc.**

**/Constants # Folder for constants that manage the configuration of the game**

**/config # Game configuration settings like screen size, difficulty, etc.**

**/sprites # Sprite constants like default dimensions, asset paths, etc.**

**/Entities # Folder for game entities (objects and entities in the game world)**

**/enemies # Contains enemy-related classes and behaviours**

**/base\_enemy # Base class for all enemies, contains common enemy properties and methods**

**/towers # Contains tower-related classes and behaviours**

**/base\_tower # Base class for all towers, contains common tower properties and methods**

**/Game # Folder for core game logic, classes, and mechanics**

**/game # Contains core game engine logic, managing game state, and looping**

**/map # Class or files related to the map generation and management**

**/grid # Grid class for handling the 2D grid of the tower defence map**

**/maps # Specific map files or map configurations for each level**

**/mouse # Handling mouse input and interactions within the game**

**/States # Folder for various game states that control the flow of the game**

**/base\_state # Base class for game states (used for common state behaviour)**

**/game\_state # Class representing the game state (active gameplay)**

**/menu\_state # Class representing the menu state (menu navigation)**

**/pause\_state # Class for the pause state when the game is paused**

**/state\_manager # Manages state transitions (switching between states)**

**/UI # Folder for user interface components and layout**

**/Menus # Folder for menu-related UI components and screens**

**/menu # General menu components (buttons, titles, etc.)**

**/mainmenu # Main menu components**

**/optionsmenu # Options menu components (audio, graphics settings, etc.)**

**/pausemenu # Pause menu components (resume game, restart, etc.)**

**/button # Classes or components for rendering buttons in the UI**

**/tower\_selection\_panel # Panel for selecting and placing towers in the UI**

**/game\_buttons # Buttons that are part of the in-game interface (e.g., pause, quit)**

**main.py # Main entry point of the game, initializes and runs the game**

**README.md # Documentation file with game details, installation instructions, and more**

## Refined Class Diagrams

***Class Diagram for Game\_state and related classes***

|  |
| --- |
| **Game** |
| - mouse: Mouse |
| - running: bool |
| - debug: bool |
| - map: Map |
| - state\_manager: StateManager |
| - screen: surface |
| + run(): void |

**Composed of**

**Composed of**

**Composed of**

**Composed of**

|  |
| --- |
| **State\_manager** |
| - states: Dict[str, State] |
| + add\_state(): |
| + change\_state(): |
| + update(): |
| + draw(): |
| + exit\_state(): |

|  |
| --- |
| **Base\_State** |
| - game: Game |
| + update(): |
| + draw(): |
| + exit(): |
| + enter(): |

**Inherets From**

**Inherets From**

**Composed of**

|  |
| --- |
| **Game\_state** |
| - towers: List[Tower] |
| - enemies: List[Enemy] |
| - score: int |
| - money: int |
| - current\_wave: int |
| - lives: int |
| + update(): void |
| + draw(): void |
| + spawn\_enemy(): Enemy (object) |
| + place\_tower(): Tower |
| + game\_over(): bool |

|  |
| --- |
| **Enemy** |
| - reward: int |
| - health: int |
| - speed: float |
| - position: Tuple[int, int] |
| - path: List[Tuple[int, int]] |
| + move(): void |
| + render(): void |
| + take\_damage(): void |
| + is\_dead(): bool |
| + update(): void |

|  |
| --- |
| **Tower** |
| - damage: int |
| - range: int |
| - cooldown\_timer: float |
| - cost: int |
| - position: Tuple[int, int] |
| + attack\_enemy(enemy: Enemy): |
| + upgrade(): void |
| + render(): void |
| + update(): void |

**Indirectly Dependant On**

**Indirectly Dependant On**

|  |
| --- |
| **Tank\_Enemy** |
| - Defense: int |
| + take\_damage(): void |
| + move(): void |
| + render(): |
| + enrage(): void |

|  |
| --- |
| **Map** |
| - music: str |
| - sprite: str |
| - grid: Grid |
| + check\_tile(): string |
| + reset\_map(): void |
| + render(): void |
| + place\_tower(): void |
| + remove\_tower(): void |

|  |
| --- |
| **Flamethrower\_Tower** |
| - Defense: int |
| + attack\_enemy(): void |
| + upgrade(): void |
| + render(): void |

**Composed Of**

**Inherets From**

|  |
| --- |
| **Grid** |
| - grid: list [list[int]] |
| + check\_tile(): int |
| + set\_tile(): void |
| + render(): void |