Tower Defense Gameplay Requirements (All of these requirements are prerequisites in order for your game to be graded.)	**Priority**	**Done (RED - Not Done, GREEN - Done)**	**Brainstorming and general outlines for the start**	**Documentation (Mandatory)**	**Justification of design patterns, leave as blank if not applicable**	**Reflection and/or improvement after code review(s), leave as blank if no code review was done for this requirement.**	
Example requirement(DON'T DO THIS IN YOUR OWN ASSIGNMENT!): The game can generate a dungeon procedurally, the size and number of rooms of the dungeon can be adjusted without hard coding.			To think of the most important things to be implemented for this feature to be working: aka must-haves!	Example documentation: The main class for this requirement is DungeonGenerator.cs, which is attached to the DungeonGenerator prefab, you can set up the size and number of generated rooms in Unity inspector without any hard coding.	Example justification: I applied the strategy pattern for this requirement so that the dungeon generator can easily switch between using Dijkstra and using A star.	Example improvement: A code review pointed out that the dungeon should also be able to save a random seed to be used to generate the exact same dungeon in different sessions, this was implemented in a later version, now you can click 'save seed' button to save the seed to a json file.	
Wave requirements							
• The game has at least 5 waves of enemies, each wave is more difficult than the previous one.	Low	Code for this is done	*Determine what is the difficulty based on: amount of HP, speed, additional abilities?*	Main Script(s): **WaveManager.cs**, which references a list of **WaveDefinition** objects. Each index in the waves array represents one wave. Usage: Attached **WaveManager** to a central GameObject. It initializes currentWaveIndex and increments it each time a wave completes.	**The Observer pattern** is used via **WaveEventsBus** to broadcast wave start/end events. This decouples wave-related logic (spawning) from listeners such as **LevelManager**, which responds to wave completions without direct dependencies on **WaveManager**.	**Pure Improvement:** I noted that scaling enemy stats per wave could be made more flexible, since right now the difficulty of the wave is mainly dependent on amount of enemies. As an improvement, **simultaneous/sequential spawn option** was added to each WaveDefinition so that we have more designer-friendly ground for level design.	
• Properties of a wave can be configured in the Unity Editor without changing the code, e.g. one or multiple of: enemy types, enemy amounts, enemy combinations, delay between enemies, percentage of each enemy type, chance of spawning, etc.	Very High	Waves can be created and configured easily - done	*Designer-friendly editor for tweaking - should be grown from start*	Main Script(s): **WaveDefinition**.cs and **SpawnInstruction**.cs. Usage: Each **WaveDefinition** exposes properties (like **spawnInstructions**, postWaveDelay, or whether to spawn in parallel) so you can tweak them directly in the Unity Inspector. **SpawnInstruction** includes fields for enemy type, amount, spawn delay, etc.	Uses **ScriptableObject pattern** (via **EnemyDataSO** referenced in **SpawnInstruction**) to store and configure enemy properties outside of code. This separates data from logic and keeps wave parameters easily adjustable.	**Pure Improvement:** We can define spawn point for every spawn instruction by inputting any empty gameobject transform. They spawn, and, by using **NavMesh** go to **Start Node -> Middle Node -> End Node**. This extends the designer-friendly environment.	
• In between waves the players have a short building phase to sell/destroy, build and upgrade the towers.	Very High	Build phase duration value is introduced, together with game states, but no connection logic is made yet	*Check if there are no remaining enemies; introduce timer; start another wave for the enemies to appear*	Main Script(s): **LevelManager**.cs controls the game's overall state. When a wave finishes, the state changes to Building and starts a build timer (buildPhaseDuration). Players can place or upgrade towers during this phase before the next wave begins.	The **State Machine** approach is partially demonstrated by the **LevelState enum** and ChangeState() method in **LevelManager**.cs. While it's not a formal "State" design pattern class structure, it follows a similar principle: the game transitions through Building, SpawningEnemies, Win, Lose, etc.	**Code Review:** After one code review, my tester was not glad with the fact that towers coudn't be built during SpawningEnemies state. **allowTowerBuildingDuringWave** was introduced to have such possibility and can be toggled on/off inside **LevelManager** GObject. Besides, a small overlook was fixed where player could get a **WIN state** when all waves were completed, but **not all the enemies** were killed.	
Enemy requirements							
• Enemies are spawned from spawn point(s).	Very High	Start node is existing and can be moved anywhere	*Make sure they are spawned above the ground, and not stuck in the floors. If they are going to have collisions - that might cause multiple issues. Decide if they need to have rigidbodies and*	Main Script(s): **WaveManager** (handles spawning via **WaveDefinition**), which instantiates Enemy objects. Each **Enemy** uses **EnemyMovementController**.cs, which delegates path logic to either **GroundMovementStrategy** or **FlyingMovementStrategy**. Usage: In the Unity Inspector, you set up **spawn points** and **WaveDefinition** entries. The **EnemyMovementController** calls StartMovement() on spawn, guiding enemies along node paths or directly toward a final goal.	**Strategy Pattern**: The **MovementStrategyFactory** picks either **GroundMovementStrategy** or **FlyingMovementStrategy**, letting you easily add or swap additional movement behaviors without modifying **EnemyMovementController**.	**Code Review + Pattern Implementation:** Upon code review, I was suggested to have an interface **IMovementStrategy** to accommodate new specialized movement classes (such as ground or flying strategies), keeping the code open for extension without heavy refactoring.	
• Enemies follow a non-straight path to the end goal.	High	Node-path + nav-mesh system was created: multiple nodes are created, obstacles are avoided	*Create waypoints system using NavMeshSurface?*	Main Script(s): **Node**.cs with a **NodeSelector** (e.g., **FixedNodeSelector**) that defines the **sequence of waypoints** for each enemy. The **EnemyMovementController** calls ProceedToNextNode() whenever an enemy enters a node's trigger, ultimately leading to the goal. Usage: You place multiple **Node objects** in the scene and link them together in the Inspector. Ground-based enemies will follow these nodes in order, resulting in non-linear paths.	The **node-based path** remains flexible, letting to add or remove nodes for different layouts without rewriting code.	Having a possibility to attach **different spawn points** inside the **spawn instruction** allows having the enemies follow different routes from different start points.	
• When x enemies reach the end goal it is game over, x is a tweakable level property.	High	X is a tweakable level property, but is not connected to any logic yet	*Introduce counter for the game over event to be triggered*	Main Script(s): **LevelManager**.cs tracks enemiesPassed and compares it to **maxEnemiesAllowedToPass** (adjustable in the Inspector). When the limit is reached, PlayerLost() sets **LevelState** to Lose. Usage: In the LevelManager's Inspector, set maxEnemiesAllowedToPass to any desired value.	**Observer/Event Bus**: **EnemyEventsBus**.**OnEnemyReachedGoal** notifies the LevelManager so it can increment its counter. This decouples the enemy from the level state logic.		
• Enemies have at least three properties: health, speed and carried money.	Very High	Properties are there, but logic web is still not there	*Health used for being damaged and killed; speed is used for adding pressure onto the player; carried money is what helps to build more towers in the game*	Main Script(s): **EnemyDataSO**.cs defines maxHealth, moveSpeed, and deathReward for each enemy type. For example: Usage: Create multiple **EnemyDataSO** assets, each with different stats and assigned prefabs for distinct visuals.	**ScriptableObject**: All enemy attributes (including visuals) live in an EnemyDataSO, so no code changes are required to introduce new enemy types or tweak existing ones.	**Code Review + Pattern Implementation:** Upon another code review and reflection I have moved from pure MonoBehaviour scripts to using Scriptable Objects (which I have never used before) to define the characteristics of enemies. Scriptable Object allowed me to test enemy characteristics during play mode and iterate on them.	
• There are at least two types of different enemies with different visuals and properties.	High	Prefabs can be changed easily, and tweaked both with the properties and visuals: done	*At least having two prefabs with different skin, but almost the same logic.*	**Ground Enemies**: standard path-following, moderate speed/health. **Flying Enemies**: skips middle nodes, faster but with lower health.	**ScriptableObject**: All enemy attributes (including visuals) live in an EnemyDataSO, so no code changes are required to introduce new enemy types or tweak existing ones.	**Ground and Flying Enemies** use their own configured **NavMesh surfaces** to walk around the level.	
• When an Enemy unit is destroyed, the amount of money gained should be clearly visible near the Enemy unit.	High	No UI is done yet	*World-space UI appearing near the enemy upon being killed*	Main Script(s): **Enemy**.cs calls **ShowRewardPopup**() in **Die**(). This triggers the **FloatingRewardManager** to spawn small floating icons indicating the gained coins at the enemy's position. Usage: Adjust the popup style, lifetime, or icon sprite in the UI prefab. FloatingRewardManager allows adjusting spacing between icons (since they use horizontal layout group) and pool size.	**Observer**: **EnemyEventsBus**.**RaiseEnemyDied** can also notify any extra UI logic if needed, ensuring the money popup logic is separate from the enemy's main script.	**Code Review + System Implementation**: From group projects I have learned about using pool systems in the game to minimize resource usage. Even though, pool systems are very useful when one wants to deal with enemy instantiation, I have gone for a simple thing to try out. Finally I made a floating reward **Pool System** which allows to reuse the same floating reward UI elements, and not destroy/instantiate them every time.	
• Attributes and visuals of enemies can be set up in the Unity Editor without hard coding.	Very High	Visuals can be changed only via prefab mode, and not in the unity editor.	*3D model or properties can be tweaked easily*	Main Script(s): **EnemyDataSO**.cs includes references to the enemy's prefab (enemyPrefab), base stats (maxHealth, moveSpeed), and reward. Simply create or duplicate an **EnemyDataSO**, assign a different model or speed, and the new enemy is ready. Usage: To change visuals, drag a different mesh or prefab into enemyPrefab. To change its stats, refer to the assigned SO.	**Open/Closed Principle**: Introducing new enemy variations is as simple as adding new **ScriptableObjects** with unique properties and models, requiring no changes to the underlying Enemy scripts.		
Tower requirements							
• Towers can be bought and built on grid cell along the enemy path in exchange for money.	Very High	No Grid placement done yet	*Creating a unity grid cell system for building or just using tile editor package for that? :)*	Main Script(s): **TowerPlacementManager, TowerPlacementGrid** and **CurrencyManager.** When a player selects a tower to buy, **TowerPlacementManager** spawns a "ghost" (**TowerPlacementGhost**.cs) and checks if the player has enough funds. **TowerPlacementGrid** ensures each grid cell is free before allowing placement and deducts currency upon confirmation.	**Observer Pattern**: **TowerEventsBus** broadcasts events (e.g., **OnTowerPlaced**), so UI panels or other systems can respond when a tower is built, without tight coupling to the placement logic.	**Code Review:** Important improvement I have made on the grid placement is introducing **ghost tower** which appears under user's mouse, upon clicking on the shop item button. This has enhanced the UX of my tester drastically, since he can visually see where the tower is placed now. Additionaly **grid snapping** was introduced, so the towers are placed accordingly to the provided space.	

• The tower's damage, range, attack interval and attack type should be displayed to the players when they select the tower they want to buy.	High	No UI info at all	*UI canvas with icons, numbers and general information.*	Main Script(s): **TowerLevelData**.cs stores each tower's stats (e.g., towerDamage, towerRange, towerFireRate). **ShopItemDataSO** provides a quick way to reference the Tower prefab (and thus its first-level **TowerLevelData**). **ShopItemButton** shows the tower's cost and invokes **TowerInfoDisplayPanel**.**ShowShopItemInfo**(shopItem) when clicked. **TowerInfoDisplayPanel** reads from **shopItem.towerHolderPrefab.levels[0].towerLevelData** to display stats such as damage, range (via DPS), fire rate, and debuff/attack type in the UI before purchase.	**ScriptableObject**: All stats are defined in TowerLevelData as a **ScriptableObject**, allowing you to easily adjust damage, range, etc. in the Unity Inspector without changing any code.	**TowerLevelData** was very well connected and integrated to the UI system in my game, regarding **Upgrading** and **Displaying** Information.	
• One tower-gridcell can only hold one tower (towers can't overlap with each other).	High	No grid system yet	*^^ using the tilemap package here could help a lot*	Main Script(s): **TowerPlacementGrid**.cs uses an internal bool[,] availableCells to track which cells are occupied. When you place a tower, the relevant cells get marked as occupied.	**Single Responsibility** Principle (SOLID): **TowerPlacementGrid** focuses solely on positioning/occupancy logic, while **TowerPlacementManager** handles the higher-level process of initiating placement and checking currency.	A peer review suggested highlighting cells if they're already occupied. I implemented a quick visual tile system (**PlacementTile**.cs) that changes material to a **"filled" material** (upon tower placement), or **"empty" material** (upon tower sell).	
• There are at least three different types of towers:	Very High	In process	*Creating prefab variants here with different visuals and damage-dealing features.*	Main Script(s): Each **TowerLevel** is configured with a **TowerLevelData** that uses different **ProjectileEffectSO** classes. Usage: In the Unity Editor, you create three separate tower prefabs, each referencing the appropriate **TowerLevelData** and thus the wanter **projectile effect.** Additionally all the settings regarding the projectile can be configured in the TowerLevelData.	**Strategy Pattern**: The **ProjectileEffectSO** is a classic **Strategy** approach, letting each projectile use a different effect (SingleTarget, AoE, Debuff) by simply assigning a different **ScriptableObject** to the tower.	At start, all the projectile logic was written in one place, but using Strategy Pattern allowed me to extend the base, and actually not be concerned about failing at Open/Closed principle.	
• Single target attack tower	High	Basic tower is done and shoots one target at a time	*Selects the first enemy in the line within the tower range (think of how to implement this, since what is being first one)*	Single-target uses **SingleTargetProjectileEffectSO**	Overrides ProjectileEffectSO's abstract void ApplyEffect() in dedicated script		
• AOE attack tower	High	AOE property is introduced in the Scriptable objects, but logic is not there	*Multiple enemies attack at the same time (if there are "enemies" in the zone, deal damage to them)*	AoE uses **AOEProjectileEffectSO**	Overrides ProjectileEffectSO's abstract void ApplyEffect() in dedicated script		
• Debuff attack tower (slows downs enemies, or another effect, the same effect never stacks)	High	Debuff property is introduced in the Scriptable objects, but logic is not there	*Reducing speed of the enemies which are being shot by the tower, usually AOE as well)*	Debuff uses **DebuffProjectileEffectSO**	Overrides ProjectileEffectSO's abstract void ApplyEffect() in dedicated script		
• All projectiles/AOE/Debuff attacks are visible in the Scene.	High	Projectiles should be added for Debuff and AOE attacks	*Having an attack effect visible*	Main Script(s): **Projectile**.cs handles movement and calls the assigned **ProjectileEffectSO** on impact. The AoE overlap or debuff effect is triggered when reaches the enemy. Usage: Mesh of the projectile is assigned inside the **projectilePrefab** and then inside every **TowerLevel** object.	**Open/Closed Principle (SOLID)**: Each new projectile effect is simply a new prefab or new **ScriptableObject**. We don't modify Projectile.cs for each new effect. We only configure the visuals in the Editor, or create new scripts for new projectile logic.		
• Which towers can be bought is adjustable without changing the code.	High	No building phase done yet	*Which towers can be bought depends on money amount?*	Main Script(s): **Tower**.cs plus the relevant ScriptableObjects (**TowerLevelData**). **ShopItemDataSO** assets list which tower prefab to spawn and its cost (derived from the tower's first level). **ShopManager** loads a list of **ShopItemDataSO objects** in the Inspector. You can add, remove, or reorder entries to change the available towers without any code changes. **ShopItemButton** is dynamically created for each **ShopItemDataSO**, so the UI automatically updates to reflect newly added towers.	**ScriptableObject pattern** again: each Tower's availability and cost are set in the **Editor**, so designers can tweak or add new tower types as needed. The code simply loops over whichever towers are present in the shop's array.	I added a **buttonBackground** sprite field in **ShopItemDataSO** to help visually differentiate the buttons in the shop, further removing the need for code modifications.	
• Towers can be upgraded with money.	High	No upgrade feature yet, however there is tower level introduced	*By spending money, you can increase area, dmg or speed of attacking*	Main Script(s): **Tower**.cs includes a method **UpgradeTower**() which swaps the current level prefab for the next. **TowerUpgradeManager**.cs checks if the player can afford the upgrade via **CurrencyManager**.cs before calling UpgradeTower(). Usage: The player selects a tower in-game; if sufficient funds are available, **TryUpgradeTower**() is invoked, deducting the upgrade cost and updating the tower's level.	**Single Responsibility Principle**: The currency and upgrade logic is cleanly separated. **TowerUpgradeManager** handles upgrade validation and cost deduction, while **Tower**.cs only knows how to swap to the next visual/level.	For further extendability: introduced **TryUpgradeTowerToLevel**() to handle **cumulative** costs and skip intermediate upgrade clicks, improving the user experience.	
• The game clearly shows which towers can be bought and upgraded with the current amount of money.	High	No ghosting is done in that regard	*UI allows or doesn't allow to buy specific towers with the amount of money you have*	Main Script(s): **ShopManager**.cs dynamically creates buttons for each tower type in the Shop Panel and monitors **CurrencyEventsBus**.**OnMoneyChanged** to update button interactivity. The upgrade UI also checks **CurrencyManager**.**CanAfford**() each time the player selects a tower to upgrade. Usage: Players see a highlighted (or grayed-out) buy/upgrade button. When they lack the funds, the button is disabled.	**Observer Pattern**: The **CurrencyEventsBus** broadcasts money changes so that **UI elements** instantly reflect whether the user can afford to buy or upgrade.	A **Tower Selection Handler** (basic raycast) was introduced to allow the user to click on the towers in the scene and check if theycan be upgraded. Upon clicking, a **TowerUpgradePanel** appears with all the necessary information.	
• The upgrade system can be adjusted in the Unity Editor without changing the code so that at least two of the following are tweakable for the upgrade: damage/debuff power, range, attack interval.	High	I click on the tower - it gets upgraded to the next level.	*Showcasing that I can change upgrading features in the unity editor*	Main Script(s): **TowerLevelData**.cs allows each tower level to have configurable towerDamage, projectileDebuffMultiplier, towerRange, and towerFireRate. You can tweak these values for each level in the Editor without touching code. Usage: For example, you can set level 2 to have higher damage and slightly faster firing, or change the debuff slow percentage for a Debuff Tower, all via the TowerLevelData Scriptable Object asset.	**ScriptableObject Configuration**: This approach cleanly separates the tower's behavior from its data, letting you create multiple variants of towers.	A lot of rebalancing was done after testing the towers. Since the main goal of this course is to show the functionality over game feel, I didn't bother spending much time on tweaking it to perfect level :)	
• Different levels of towers have different visuals: they have different colors/scales/models to indicate that they are the same tower but at different level.	High	Done - I have two different levels with two different SOs attached	*Having different models for the tower, or adding stars (ui) above them*	Main Script(s): **Tower**.cs holds an array of **TowerLevel prefabs**. When you upgrade, the old level prefab is destroyed and replaced by the new one (with its own mesh, materials, etc.). Usage: In the Inspector, each **tower's levels[]** references a series of prefabs (**TowerLevel**.cs), each with distinct appearance while retaining the same overall functionality.	**Factory Method**: By instantiating different TowerLevel prefabs, the visuals and data for each level are kept modular. This also aligns with the **Open/Closed Principle**, since adding or changing a level prefab doesn't require changing the base Tower script.	**Tower Information Panel** (upon clicking on tower) reveals all the necessary information regarding this specific tower with its own level, stats, etc. That concerns overall **UX**.	

GUI Requirements, the following information should be displayed in clear ways in the game:					**Observer Pattern**: A unified UICore listens to game-wide events (wave start, money changed, etc.), decoupling the UI from gameplay scripts. Each UI element (e.g., HUDManager) only updates the relevant text or visual fields, staying modular and maintainable.		
• Wave number.	Normal		*What wave is it now*	Main Script(s): **UICore** listens to WaveEventsBus.OnWaveStarted and passes the wave index to **HUDManager**.**SetWaveNumber**(). Usage: The wave count is displayed at the top of the HUD and updates automatically when a new wave starts.			
• Total money.	Normal		*Money gained from killing enemies and winning*	Main Script(s): **UICore** subscribes to CurrencyEventsBus.OnMoneyChanged and updates **HUDManager**.**SetMoney**(). Usage: The current money is shown in HUDManager, enabling players to see if they can afford a new tower or upgrade.			
• Time left for the player to build/upgrade towers before next wave begins.	High		*Introducing timer for the building stage*	Main Script(s): **LevelManager** tracks a build timer, while **HUDManager** displays the remaining time each frame (BuildPhaseDuration - BuildTimer). Usage: When the game is in LevelState.Building, the **HUD** shows a countdown.			
• Where can the towers be built.	High		*Using the grid system to showcase white and red zones*	Main Script(s): **TowerPlacementGrid** (with optional visual tiles in **PlacementTile**.cs) highlights valid cells. The "ghost" tower (**TowerPlacementGhost**) also changes materials or color to indicate valid vs. invalid placement.			
• Health bar of each enemy.	High		*World-space UI element floating near the enemy at all times*	Main Script(s): **EnemyUIController**.cs spawns a **EnemyHealthBarUI**.cs prefab on each enemy. The **EnemyHealthBarUI** listens to EnemyEventsBus.OnHealthChanged to update a red/white health bar in real-time. Usage: Can tweak the bar's colors, animation speed, or offset in the Inspector.		Health bars (**EnemyHealthBarUI**.cs) use **billboard** logic so they **face the camera** at any angle.	
• How many enemies can enter the end point before game over.	Normal		*Having a number which showcases the game-over number*	Main Script(s): **LevelManager**.cs keeps track of **maxEnemiesAllowedToPass** and increments **enemiesPassed** every time an enemy reaches the end. When enemiesPassed >= maxEnemiesAllowedToPass, the **PlayerLost**() method is triggered, transitioning the game to the **Lose state**.	The Observer/Event Bus pattern again: **EnemyEventsBus**.**OnEnemyReachedGoal** notifies **LevelManager** that an enemy has passed. This event-based approach reduces direct interdependencies.		
• Final game state (win/lose)	High		*You WON! You LOST overlay*	Main Script(s): **LevelManager**.cs changes state to LevelState.Win after all waves complete, or to LevelState.Lose if too many enemies pass. **GameUIManager**.cs listens for these state changes and displays appropriate win/lose screens.	The Observer pattern is evident with **LevelEventsBus**.**OnLevelStateChanged**. **GameUIManager** subscribes to show/hide Win/Lose panels without embedding direct logic inside the **LevelManager**.	Additionaly, **PAUSE screen** was introduced by manipulating the game timescale. It allows to restart the level, or continue playing it.	
• The UI is **resolution independent**	High		*UI is adjusted by the screen size*	Main Script(s): **HUDManager**, **GameUIManager**, etc. rely on Unity's Canvas system with **anchors and scaling**. All text uses TextMeshPro (TMP), which scales appropriately.			
Other (non code) requirements							
Visual requirements:							
• The game uses either the suggested visual assets in the manual or other free visual assets that are suitable for a tower defense game.	High	Using Kenney assets from the manual.		No documentation needed if done.			
• The game can be presented as a suitable portfolio item (a game made with placeholders and default buttons is not a suitable portfolio item).				No documentation needed if done.			
Technical requirements:							
• Project uses a clear hierarchy and asset structure, with descriptive (Scene) names and a clear GameManager in the Hierarchy				No documentation needed if done.			
• Game speed can be easily adjusted at runtime. (https://docs.unity3d.com/ScriptReference/Time-timeScale.html)			***USE LEFT AND RIGHT ARROW KEYS IN-GAME***	No documentation needed if done.			
• The project has no exceptions or warnings.				No documentation needed if done.			
• Game can be restarted without closing and reopening.				No documentation needed if done.			