MONASH University
Information Technology

FIT3077 Software Engineering: Architecture & Design

Sprint 3

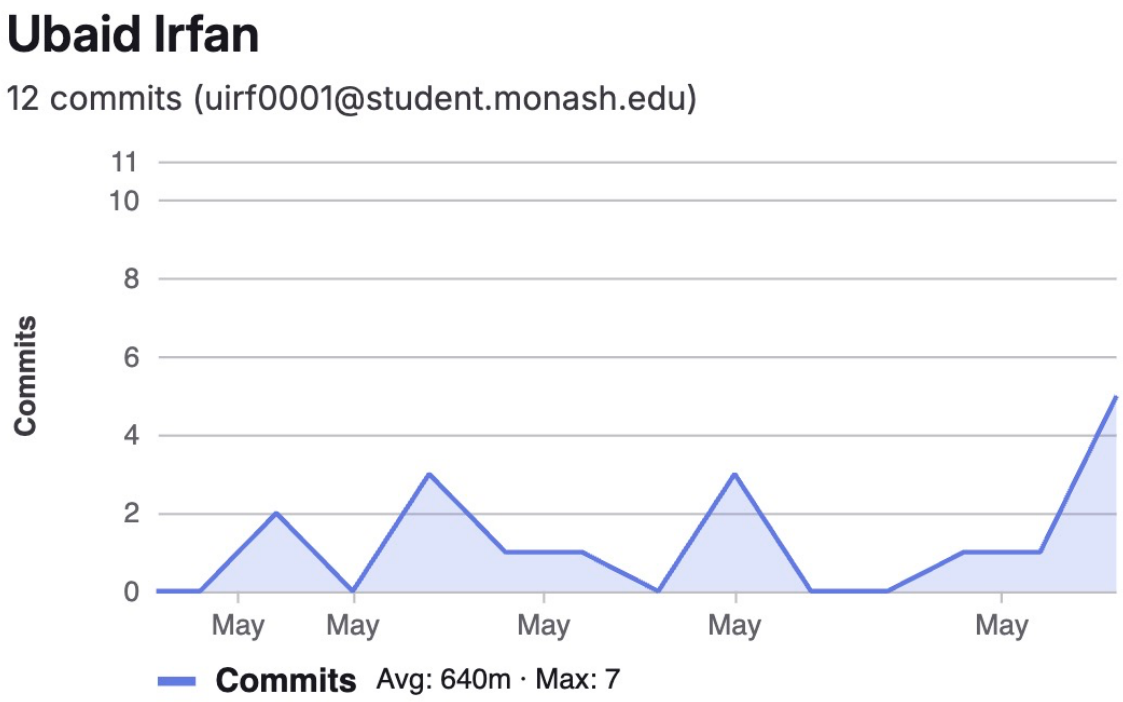
Name: Ubaid Irfan

Student ID: 33886466

Date: 31/05/2025

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

Contributor Analytics:



Accessible through git: https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-individual/uirf0001/-/graphs/main?ref_type=heads

Brief Description of Chosen Extensions:

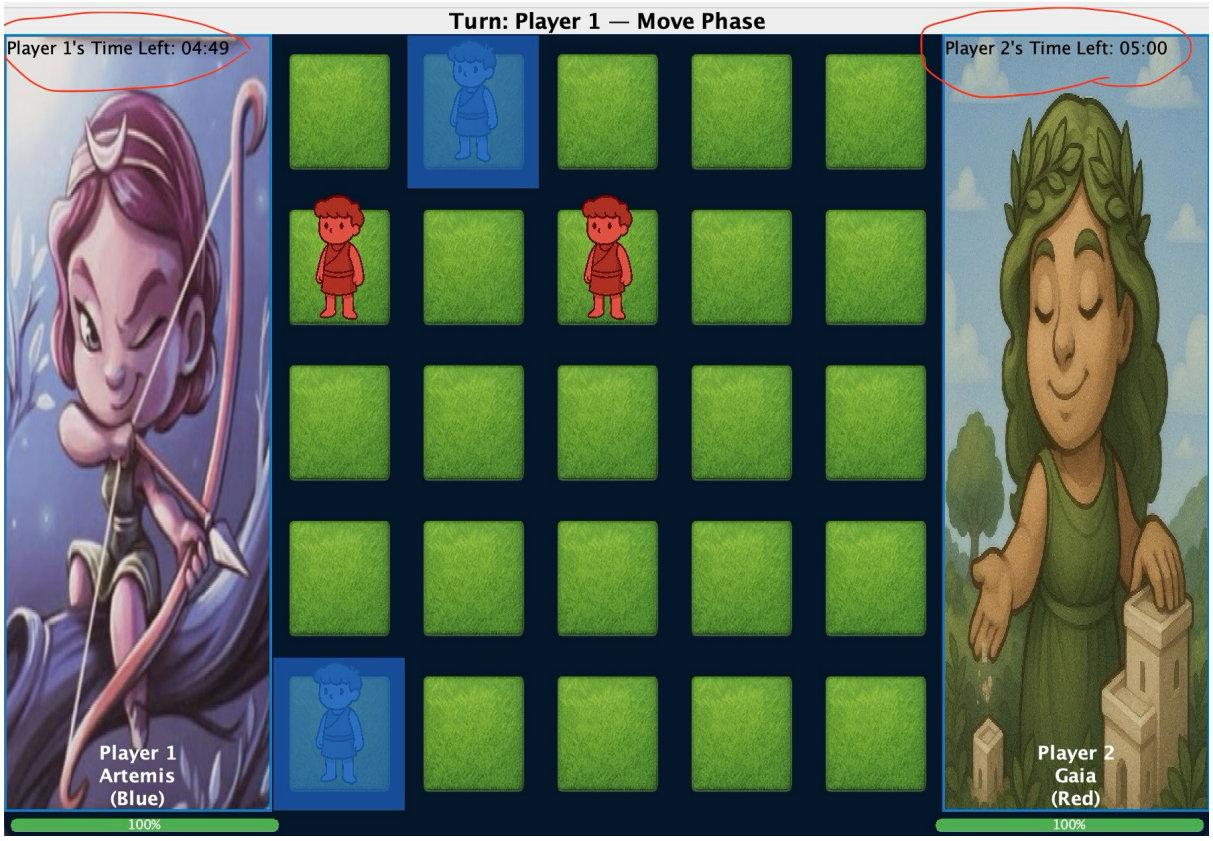
Extension 1: The addition of a new Zeus God/Power. This extension enables a player with Zeus to build a block underneath their current worker.

It is my interpretation from the game rules that a win condition (in this related scenario) should only occur when a worker **moves up** from a level 2 building to a level 3 building.

- As such, since no move occurs when a player places a building under a worker, I have enforced these below conditions.
    - If a player places a build under a worker who was previously on a level 2 building, that worker will be boosted to a level 3 building, however, a win condition will not trigger.
        - Similarly, if that worker on level 3 building moves to another level 3 building, a win condition will not trigger.
    - As per previous implementation, a win condition will trigger when a worker on level 2 building moves up to a level 3 building.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025
Extension 2: The addition of a timer functionality.

- Each player starts with 5 minutes for 'thinking' and executing their turns. Once a player has completed their turn, their clock pauses, and the clock starts for the next player.
- If a player runs out of time, they will instantly lose the game, with both player's timer's pausing indefinitely at that moment.
- Each player's timer appears at the top of their respective player panels, as illustrated below.



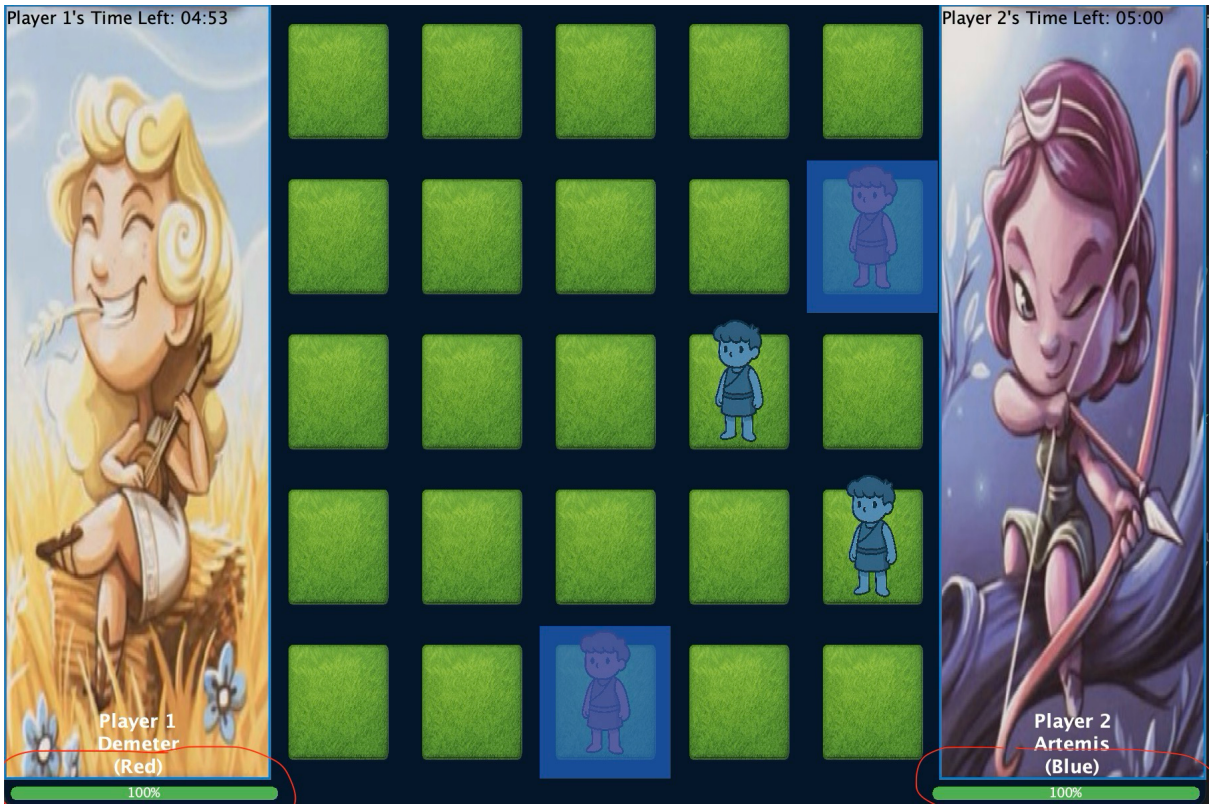
*Timers have been circled in red

Extension 3: Human Value – Protecting the Environment.

- **Remove existing blocks functionality** via the addition of a **new Gaia God/Power.**
  - A player with Gaia may optionally remove any block that is adjacent to, or on the same cell as their current worker.
  - The remove build phase becomes available after the build phase.
  - A remove may remove only 1 level at a time, e.g. removing a dome will update that cell with a level 3 building.
  - A remove action does not distinguish between a player's own blocks, meaning during a valid remove build action, a player may remove any block in their vicinity, including blocks which were placed by another player.

- A player in remove build phase may remove a block under a worker.

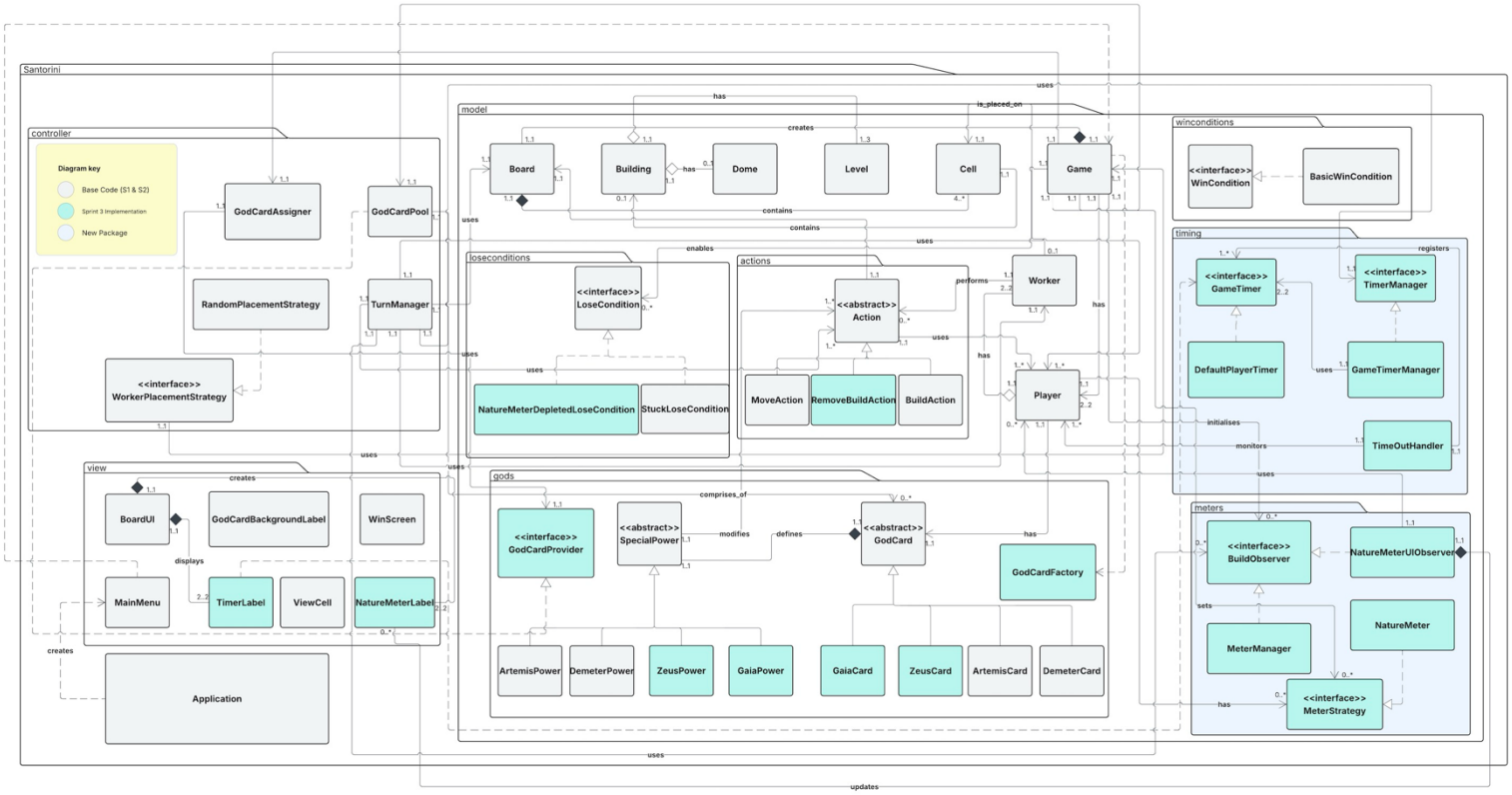
- **Nature meters** have been implemented.
  - Each player starts with max capacity (100%) on their nature meter.
  - Acts of destruction to the environment, e.g. block placement, will deplete the corresponding player's nature meter.
    - Placing a level 1 building causes a 5% reduction (impact) to the nature levels.
    - Placing a level 2 building causes a 10% reduction (impact) to the nature levels.
    - Placing a level 3 building causes a 15% reduction (impact) to the nature levels.
    - Placing a dome causes a 20% reduction (impact) to the nature levels.
  - Nature meter may be replenished via acts of grace towards the environment, i.e. removing an existing block.
    - The amount replenished will correspond to the initial impact caused by the block placement, e.g. removing a dome will replenish nature meter by 20%.
  - Currently, the only way to affect the nature meters are by placing/removing blocks.
  - If a player's nature meter is depleted to 0%, they will instantly lose the game.
  - Each Player's nature meter is displayed below their respective player panels, as per below.



*Nature meters have been circled in red

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

Updated UML Class Diagram (Rev. 5):



For a clearer image of the final iteration - access via link:
https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-individual/uirf0001//blob/main/docs/uml/Sprint%203/uml_class_final_S3.png?ref_type=heads

All revisions can be accessed through this link:
https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-individual/uirf0001/-/tree/main/docs/uml/Sprint%203?ref_type=heads

Brief Discussion Regarding the UML Class Diagram:

- The diagram includes all the necessary classes/interfaces for the required functionality of the Santorini project, as well as the extensions made in Sprint 3.
- A diagram key is provided to distinguish between the Sprint 2 classes/interfaces (grey) and the Sprint 3 classes/interfaces (cyan). Sprint 2 packages have been made white, whereas new packages added in Sprint 3 are highlighted in blue (as per diagram key).
- Unlike the Sprint 2 UML Class Diagrams, methods/attributes have been omitted, to favour relationships, class names, and minimum/maximum cardinalities, and to increase readability.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

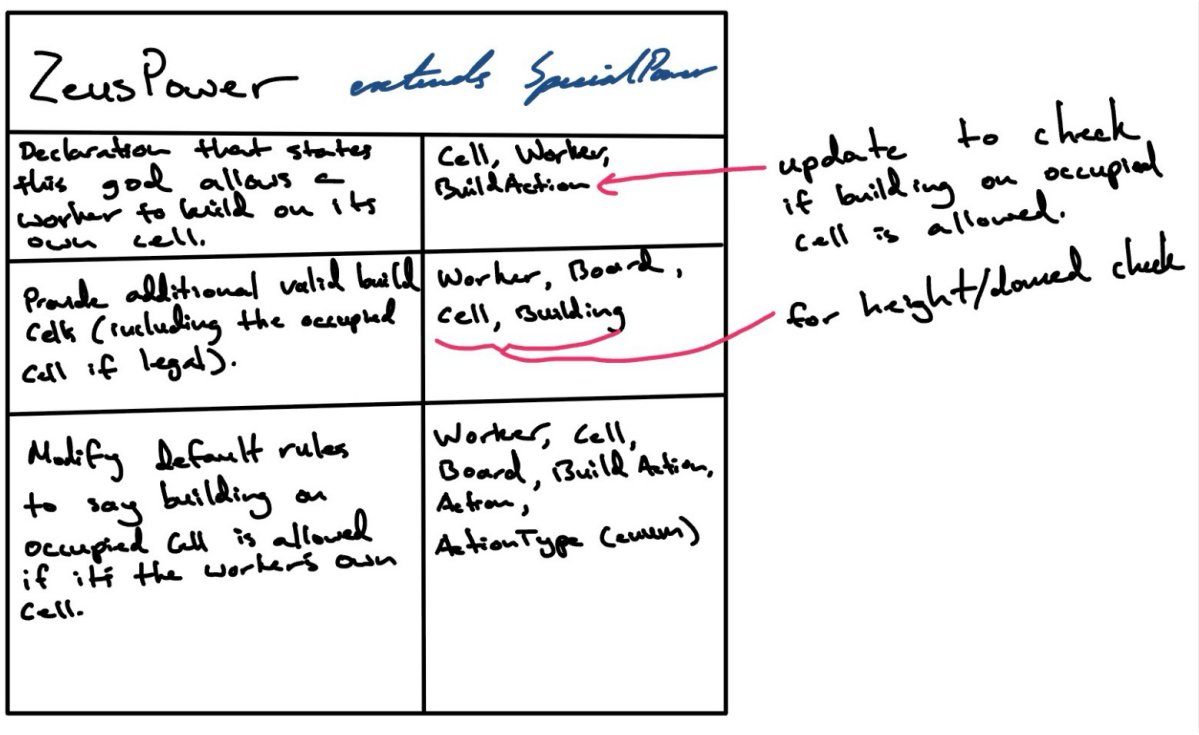
- Below is a comprehensive list of all the newly added classes/interfaces in the Sprint 3 design.
    - NatureMeterDepletedLoseCondtion – implements LoseCondition
    - RemoveBuildAction – extends Action
    - ZeusPower, GaiaPower – extends SpecialPower
    - ZeusCard, GaiaCard – extends GodCard
    - GodCardProvider (interface)
    - GodCardFactory
    - GameTimer (interface)
    - DefaultPlayerTimer– implements GameTimer
    - TimerManager (interface)
    - GameTimerManager – implements TimerManager
    - TimeOutHandler
    - TimerLabel
    - BuildObserver (interface)
    - MeterManager, NatureMeterUIObserver – implements BuildObserver
    - MeterStrategy (interface)
    - NatureMeter – implements MeterStrategy
    - NatureMeterLabel

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

CRC Cards:

All CRC cards (including drafts) are accessible via this link:
https://git.infotech.monash.edu/FIT3077/fit3077-s1-2025/assignment-individual/uirf0001/-/tree/main/docs/crc/sprint3?ref_type=heads

ZeusPower (final):

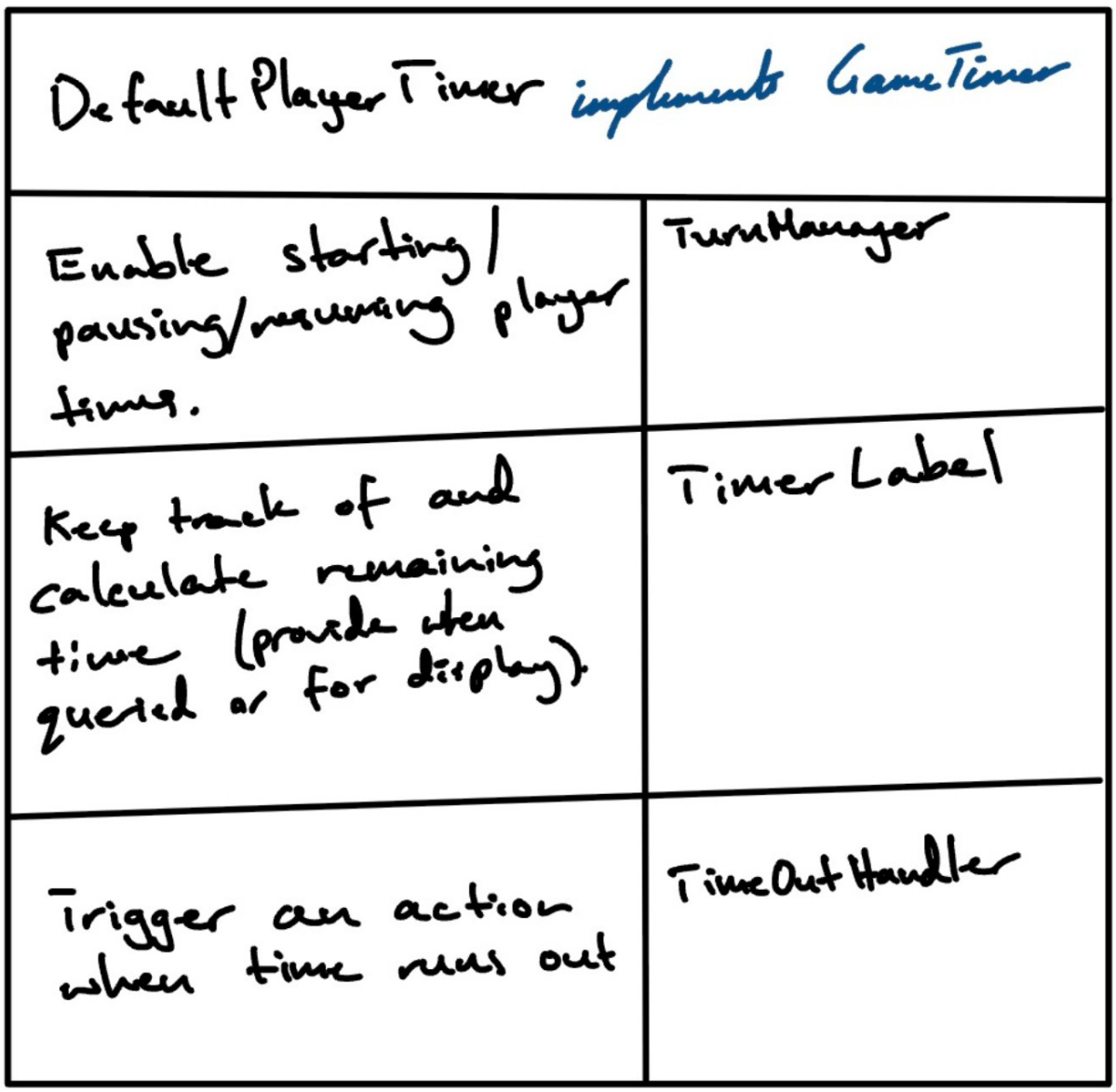


Description:

- The top row of this CRC card outlines the class we are focusing on (ZeusPower) and which class (if any) it is a subclass of (SpecialPower).
- Each row below the title/top row is comprised of two columns – the left column corresponds to a responsibility of the class, and the right corresponds to the collaborators.
- For each row, there is a new responsibility, and hence a corresponding collaborators list.
- Some arrows/extra text have been added to enhance understandability.
- This CRC card has been produced to cover a part of Extension 1.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

DefaultPlayerTimer:

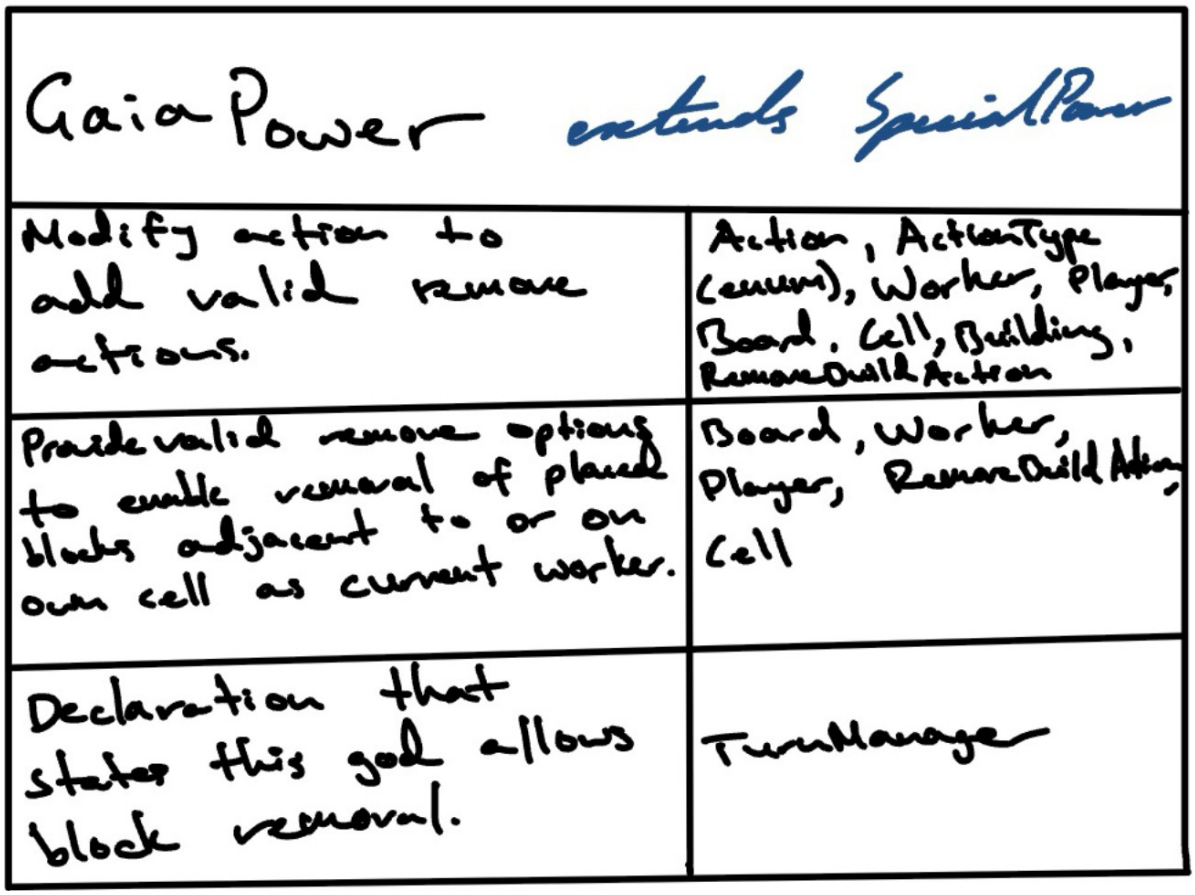


Description:

- The top row of this CRC card outlines the class we are focusing on (DefaultPlayerTimer) and which class (if any) it is a subclass of (GameTimer).
- Each row below the title/top row is comprised of two columns – the left column corresponds to a responsibility of the class, and the right corresponds to the collaborator/s.
- For each row, there is a new responsibility, and hence a corresponding collaborators list.
- This CRC card has been produced to cover a part of Extension 2.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

GaiaPower:



Description:

- The top row of this CRC card outlines the class we are focusing on (GaiaPower) and which class (if any) it is a subclass of (SpecialPower).
- Each row below the title/top row is comprised of two columns – the left column corresponds to a responsibility of the class, and the right corresponds to the collaborator/s.
- For each row, there is a new responsibility, and hence a corresponding collaborators list.
- This CRC card has been produced to cover a vital part of Extension 3.
- block = a building of any level, or a dome.

Design Rationales: Extension 1 – Zeus God/Power

To introduce Zeus, the existing structure is extended by creating two new classes:

- ZeusCard (subclass of existing GodCard) – to represent Zeus in the system.
- ZeusPower (subclass of existing SpecialPower) – to encapsulate the build-under-self behaviour.

This approach ensures god-specific logic is encapsulated, following Single Responsibility Principle. Reusing the god card architecture from Sprint 2 avoided bloating the base GodCard or BuildAction classes with Zeus-specific conditionals. This cleanly adheres to the Open/Closed Principle, where new behaviour is introduced without modifying existing logic.

Why not handle Zeus's logic in BuildAction or Player?

Embedding Zeus-specific logic inside BuildAction (e.g., checking if the cell is the same as the worker's current cell) would violate **Separation of Concerns** and introduce **god-specific branching**, leading to code smells like instanceof. Instead, letting ZeusPower override canBuildOnOccupiedCell() provides a clean, extensible mechanism.

To improve flexibility, I added a few methods to SpecialPower and overrided them in specific god powers, to avoid using hardcoded checks (e.g. if player has Demeter…), in favour of polymorphic behaviour, allowing TurnManager, Player, and BuildAction to query god powers generically without breaking abstraction.

This ensures **decentralisation of rule handling** and aligns with the **Dependency Inversion Principle** (DIP) - higher-level modules (Player, TurnManager, etc.) no longer depend on concrete god classes or switch statements.

Expanding BuildAction.isLegal() to give players **clear failure feedback** and support **occupied cell building** only when the assigned god allows it, will naturally support Zeus's logic. Similarly highlighting of buildable cells was expanded upon to ask god powers for extra valid build locations - enabling visual cues for Zeus's build-on-own-cell power.

- Responsibility for interpreting build options continues to lie with the Player because god powers are tied to players. This respects encapsulation and allows god logic to remain localized.

Zeus's core logic exposed a glaring misinterpretation in the win condition logic within BasicWinCondition – in Sprint 2, it was simply checking for the existence of a worker on a level 3 building. This had to be refactored to allow the existence of workers on level 3 without triggering a win condition (as Zeus logic emphasises building on own cell to reach a level 3 building shouldn't trigger a win), enabling players to (correctly) only win the game when their worker **moved up** to a level 3 building instead.

- Although this change was made to an existing class, it was more to fix an existing bug that was not realised due to prior implementation – resulting in a more flexible and extensible design.

Alternative Designs:

1. GodType Enum with Switch Statements

Storing the god's identity in an enum and using switch (godType) blocks to manage unique behaviour was **rejected** because:

- It violates **Open/Closed Principle**
- Requires editing the switch block every time a new god is introduced
- Leads to tight coupling between god logic and unrelated classes like BuildAction, TurnManager.
- Reduced scalability and verbosity when defining behaviour-rich gods like Zeus.

2. Injecting God Logic Directly into Action Classes (BuildAction)

Letting each action class directly check god-specific conditions, e.g., hardcoded checks for Zeus in BuildAction would pollute action classes with game rule logic and violate the **Single Responsibility Principle**, making maintenance and future extension difficult.

3. GodPowerManager Singleton

I briefly considered a central manager to route all god-related decisions (build/move/extra actions). This centralized design could simplify querying powers, but:

- It introduces a **God Object** anti-pattern
- Makes code harder to trace and debug
- Contradicts the existing, cleanly distributed polymorphic structure

Minimum/Maximum Cardinalities/Key Relationships Explanation:

- 1..1 GodCardAssigner uses 1..1 GodCardProvider (association)
- 1..1 Game has 1..1 GodCardPool (association)
    - Each game picks from a specified pool of god cards, where 1..1 GodCardPool is comprised of 0..* GodCard (association)
        - To facilitate game modes with different or no god cards in the future.
- ZeusPower extends SpecialPower (generalisation)
- ZeusCard extends GodCard (generalisation)

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

CRC Card Explanation:

Class: ZeusPower

Superclass: SpecialPower

Responsibilities & Collaborators:

1. **Declare that Zeus allows a worker to build on its own cell.**

   Collaborators:

- Cell: To check if the worker's current cell is legal for building (no dome, height < 3).
- Worker: To access current cell location.
- BuildAction: Used to create a valid action on the same cell as the worker.

2. **Provide additional valid build cells (including occupied cell if legal).**

   Collaborators:

- Worker: Used to determine the worker's position.
- Board: Provides context for retrieving the current cell.
- Cell: Checked for legality (no dome, height < 3).
- Building: Checked for height to prevent overbuilding.

3. **Modify the default building rules to allow building on an occupied cell if it is the worker's own cell.**

   Collaborators:

- Worker, Cell: Used to check if the cell is the same as the one occupied by the worker.
- Board: For board context during action execution.
- Action, ActionType: Used to intercept BUILD actions for modification.
- BuildAction: Instantiated if Zeus's build-under-self condition is valid.

Changes I made to improve the existing design (adjacent to Zeus God/Power implementation):

In Sprint 2:

- **GodCardPool** both *created* and *stored* God cards – violating the Single Responsibility Principle (SRP).
- **GodCardAssigner** handled both *shuffling* and *assigning* cards – similarly overloaded

In Sprint 3:

- I added a **GodCardFactory** to isolate the creation of GodCard instances (e.g., deciding which cards exist).
- **GodCardPool** now *only manages access* to God cards via dependency-injected suppliers – serves as a provider of god cards.
- **GodCardAssigner** now relies on a **GodCardProvider** interface to decouple source logic.

Sprint 2 - Tight Coupling

- Sprint 2's design made it hard to extend or test god cards individually.
- Adding Zeus in Sprint 3 required modification of the GodCardPool constructor, violating the **Open/Closed Principle (OCP)** - to add a new god, you'd need to modify the internal list directly.

Fix:

- GodCardFactory.getDefaultGodCards() returns a List<Supplier<GodCard>>, allowing plug-and-play extension (e.g., ZeusCard::new) without changing GodCardPool.

Sprint 2 - Poor Extensibility

- Adding or removing gods required modifying internal logic of GodCardPool and Assigner.

Fix:

- Now we can support:
  - Custom god card pools (e.g., for specific scenarios, UI choices)
  - Runtime configuration with minimal code changes

Instead, injecting a god card list supports:

- Lazy construction
- Dependency injection
- Reduced memory use
- Better testability

Design Patterns Used

1. Factory Pattern – GodCardFactory

- Centralizes creation logic of God cards – enforces a clean separation of concerns.
- Encapsulates instantiation details (e.g., constructor references).

2. Strategy Pattern (Induced via GodCardProvider)

- Allows switching god card sources (e.g., from file, UI, filtered sets).

Existing: SpecialPower allows dynamic god behaviours (build, move, win conditions) to vary independently, e.g. ZeusPower enables worker to build-under-self, DemeterPower enable worker to build twice in a turn, but not on the same cell.

3. Dependency Injection

- GodCardPool and GodCardAssigner receive dependencies via constructor.
- Promotes testability and modularity.

Adherence with SOLID Principles:

SRP	- **GodCardPool**: Only responsible for *holding and supplying* GodCards. - **GodCardFactory**: Only responsible for *creating* GodCard suppliers. - **GodCardAssigner**: Only responsible for assigning GodCards to players.
OCP	- Easy addition of new god cards can be made by simply adding [GodCard]::new in GodCardFactory – no need to modify GodCardPool or GodCardAssigner. - This means classes are open for extension, closed for modification.
LSP	- GodCard is a common superclass. ArtemisCard, ZeusCard, etc., can substitute it without breaking behaviur (Sprint 2 design).
ISP	- introduced GodCardProvider, which is a clean, focused interface. o Clients like GodCardAssigner only depend on what they need: getAvailableCards().
DIP	- GodCardAssigner **depends on GodCardProvider interface**, not the concrete GodCardPool.

	- High-level modules do not depend on low-level modules anymore. Both depend on abstractions.

Design Rationales: Extension 2: Timer

New Interfaces and Classes:

1. **GameTimer (Interface)**

- **Purpose:** Abstracts common timer operations – the concept of a timer (start, pause, resume, getRemainingTimeSeconds, setOnTimeout).
- **Why not in an existing class**: No previous abstraction for turn-based timers existed, and Timer responsibilities are unrelated to core gameplay mechanics, which are handled by TurnManager, Game. Mixing them would violate Single Responsibility Principle (SRP).
- Supports flexibility: Enables substitution of different timer implementations (e.g., countdown, delay-based, custom god-specific timers) without changing client code.

2. **DefaultPlayerTimer (Concrete Implementation)**

- **Purpose:** To encapsulates logic for countdown timing, thread management, and triggering timeouts.
- It is a thread based implementation that runs in parallel with gameplay, keeping the timer decoupled from turn logic. It manages the remaining time and timeout behaviour for an individual player.
- **Why not in an existing class**: Could not be part of Player or TurnManager, to enforce SRP and a clear separation of concerns - as those should not manage thread lifecycles or low-level timing logic.

3. **GameTimerManager (Implements TimerManager)**

- Purpose: Acts as a central controller to pause() and resume() timers based on player index. It manages a list of timers for multiple players.
- Encapsulates timer logic in one place, enhancing modularity and testability.
- **Why separate class:** TurnManager should coordinate gameplay logic, not handle individual timer mechanics. Our design improves maintainability by respecting this separation of concerns. Furthermore, we avoid code duplication this way (in accordance with DRY practices), and improves encapsulation.
- Time control operations have been abstracted away from TurnManager to ensure it remains focused on gameplay sequencing.

4. **TimeOutHandler**

- **Purpose:** To connect each player's timer to a loss condition via callback.

- **Why a new class:** Keeps timeout-specific behaviour modular. Prevents coupling timeout logic with TurnManager, which focuses on turn coordination – avoids adding even more unrelated methods/responsibilities to the TurnManager god class which was unfortunately created in the Sprint 2 design.
- Ensures the UI is updated properly using SwingUtilities.invokeLater, handling Swing thread-safety.
- The responsibility to declare a timeout and trigger the WinScreen is moved to a dedicated class, rather than handled within TurnManager – which should focus more on direct sequence of events in the game, not what happens upon timeout.

5. **TimerLabel**

- A UI component that polls timer status every second and updates the displayed time.
- **Purpose:** To decouple UI refreshing from core logic and allow future enhancements (e.g., colour changes, animations on low time). It connects a GameTimer to a visible UI component (JLabel) with regular updates.
- Avoids pollution in BoardUI or timer logic with update cycles, uses Swing Timer for lightweight updates.

Alternative Designs Considered

1. **Add timer logic inside Player or TurnManager**

- Rejected because it would violate SRP and lead to tightly coupled (between turn logic and timer control), hard-to-test classes.
- Player should be more of a data holder, as per the domain model, not a timer manager.
- TurnManager already controls flow, and adding low-level state tracking would increase complexity.

2. **Use javax.swing.Timer for core countdown**

- Rejected for logic timer because it's intended for UI updates, not long-running background countdowns.
- Swing Timer would limit flexibility and make time-based game rules (e.g., pausing/resuming based on turn logic) harder to implement.
- Instead, javax.swing.Timer is correctly used only for the TimerLabel UI updater.

3. **TimeOutLoseCondition implements LoseCondition, instead of a TimeOutHandler:**

While this sounded good on paper, as it was extending the existing design by implementing the LoseCondition interface, there were many problems with this approach:

- The check for a time-out **only happened at predefined points** (e.g. startTurn(), endTurn(), or during updates).
- **Polling** was needed: the system had to repeatedly *ask* if the condition was true.
- There was no real-time responsiveness - if a player's time ran out mid-turn, the game wouldn't detect it immediately.

- This undermines the gameplay, because players could take actions even **after** their time ran out until the next check occurred.

TimeOutHandler takes a push-based, event-driven approach. This enables:

- **Immediate Response (Reactive Design):**
  - The handler is triggered **as soon as the timer expires**, not at the next game loop iteration.
  - This ensures the game ends instantly and correctly, even in the middle of a player's turn.
  - This aligns with how real-time chess clocks work.
- **Separation of Concerns (SOLID):**
  - The LoseCondition interface is for game state-based rules (e.g. no legal moves, a worker on the 3rd level, etc.).
  - Timer-based events are external to core game logic and better handled via event callbacks, not game state polling.
  - TimeOutHandler does exactly one job: register a response when a timeout occurs, respecting the Single Responsibility Principle.
- **Cleaner TurnManager Logic:**
  - With TimeOutHandler, you don't need to clutter TurnManager with yet another unrelated method for timeout checking logic.

Design Patterns Used

1. **Strategy Pattern – Used via GameTimer interface**

- Enables different implementations of timers (e.g., standard countdown, god-based effects, time boosts).
  - DefaultPlayerTimer is one strategy; others could easily be plugged in – this pattern facilitates interchangeable timer behaviours without changing client code.

2. **Observer Pattern (Conceptual/Implicit)**

- TimeOutHandler acts like an observer to timer completion via setOnTimeout().
- Loose coupling allows DefaultPlayerTimer to be reused or tested independently.
- TimerLabel acts like an observer by polling the timer periodically and updating itself. Following this pattern decouples the timer logic from the display logic in BoardUI, improving modularity.

Rejected Design Patterns:

1. **State Pattern**

- Not used for timer states (running, paused, timeout).

- Why: The timer state transitions are simple and don't justify separate state classes. A few boolean flags (isRunning, etc.), as we have done, will suffice.

2. **Mediator Pattern**

- **Not suitable:** There is no complex inter-object communication to centralize; responsibilities are already cleanly divided.

3. **Decorator Pattern**

- Not used for enhancing timers (e.g., adding visual effects or bonuses).
- Why: No dynamic behaviour stacking is needed - timer enhancements would be better modelled via composition or subclassing instead.

To Summarise, this timer extension cleanly integrates into the game's architecture by:

- Respecting SOLID principles
- Keeping gameplay, timing, and UI logic separate
- Using proper abstraction and delegation

The design allows future extensibility, such as:

- Different god-specific time rules
- Time-based score penalties
- Multiplayer network time sync

Explanation of Minimum/Maximum Cardinalities:

- **1..1 GameTimerManager uses 2..2 GameTimer**
  - We need a timer for each player in the (currently) 2 player Santorini game.
  - The constructor expects a list of exactly 2 timers.
- **1..1 TimeOutHandler uses 1..* Player**
  - Each TimeOutHandler is created with at least one player, typically more (2 players for Santorini), as it needs to know who to declare as winner or loser if timeout occurs.
- **1..1 TimeOutHandler registers 1..* GameTimer**
  - Each TimeOutHandler sets up timeout callbacks on at least one, usually multiple GameTimer objects. It needs to handle timeout consequences for each player's timer.
- **1..1 TurnManager uses 1..1 TimerManager**
  - Each TurnManager always interacts with **exactly one** TimerManager, to pause/resume timers during turn transitions.
- **1..1 BoardUI displays 2..2 TimerLabel**
  - The UI always displays exactly two timer labels - one per player.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

CRC Card Explanation:

Class: DefaultPlayerTimer

Implements: GameTimer (Interface)

Responsibilities:

1. **Manage player time flow**
    - Starts, pauses, and resumes the player's countdown timer.
    - Ensures accurate time tracking even when paused/resumed multiple times.

    Collaborator/s:

- TurnManager – Calls start() at the beginning of a player's turn and pause() at the end, or when a win/lose condition is met.

2. **Track and calculate remaining time**
    - Maintains internal state to return the correct time remaining at any point.
    - Provides up-to-date values for real-time UI display.

    Collaborators:

    • TimerLabel – Queries getRemainingTimeSeconds() to display updated time each second.

3. **Handle timeout logic**
    - Triggers a specified action (e.g., declaring loss) when timer reaches zero.
    - Executes this action safely on the EDT to update the UI.

    Collaborators:

    - TimeOutHandler – Sets the timeout behaviour using setOnTimeout(Runnable) (e.g., showing a WinScreen when a player's time runs out).

Design Rationales: Extension 3 – Human Value (Protecting the Environment):

Value Alignment:
The human value embedded in this Santorini extension is "Protecting the Environment", which is classified under Universalism–Nature in Schwartz's refined theory of basic human values. This specific value emphasizes the importance of preserving and enhancing the natural environment, promoting environmental sustainability, and limiting human impact on nature. According to Schwartz, values under Universalism reflect a concern for the welfare of

all people and nature, transcending selfish interests in favour of broader, long-term well-being.

Importance:
In today's world, where environmental crises such as climate change, biodiversity loss, and resource depletion are escalating, fostering ecological awareness is more critical than ever. Games are increasingly recognized not just as entertainment, but as cultural artifacts that can influence behaviour, attitudes, and values. By embedding "Protecting the Environment" into gameplay, players are encouraged to reflect on the consequences of their decisions, even in a fictional world. This positions the game as a subtle educational tool that promotes environmental stewardship through systems of cause-and-effect that are emotionally and cognitively engaging.

Manifestation of the Value In-Game:

The value is not merely referenced or represented visually - it is **structurally and interactively embedded into the gameplay mechanics**. Specifically, this is achieved through:

1. **The Nature Meter**: A UI-based feedback system that visibly tracks each player's ecological footprint. Every BuildAction decreases the meter, simulating environmental degradation. The rate of depletion is tied to the scale of construction:
    - Level 1 block: Small impact (5%)
    - Level 2 block: Moderate impact (10%)
    - Level 3 block: High impact (15%)
    - Dome: Very high impact (20%)

To faithfully reflect the value, the game mechanics must simulate realistic consequences of different types of human activity on the environment. A one-size-fits-all penalty for every build, regardless of its size or impact, would ignore the nuanced reality that not all development is equally destructive. That's why in this implementation, larger or more "unnatural" constructions (domes) deplete the Nature Meter more significantly.

Small-scale construction (e.g. placing a Level 1 block) may represent minimal human intervention, like using existing resources with low energy cost. In contrast, a level 3 building or a dome represents high levels of modification to the natural environment - possibly involving resource-intensive construction, habitat destruction, or irreversible changes.

Reasoning:

- The *value* is not just "don't build," but build responsibly.
- A flat penalty would discourage all building equally, which undermines the subtlety of environmental ethics. The scaled impact system supports intelligent, mindful choices: players can build - but they must think carefully about how far to go, and whether a taller structure is *worth* the ecological cost.

Strategic Depth/Consequential Decision-Making:

- By scaling the impact, players are forced to prioritise, e.g. *'do I rush a Level 3 building and risk losing Nature Meter points quickly? Or do I proceed incrementally with lower levels to maintain balance?'*
    1. Pursuing aggressive building tactics may secure a positional advantage but could ultimately cost the game if the Nature Meter reaches 0%. This mimics real-world tensions between development and conservation, encouraging players to think twice before exploiting all resources for short-term gain.
- This introduces **moral tension** - a key factor in value-driven design, encouraging players to reflect on trade-offs between ambition and responsibility, just as societies must in real life.

Gameplay Balance:

- A constant penalty would make building a dome (which is necessary to win) just as harmful as building a Level 1 block, which is unrealistic and leads to skewed incentives.
- With scaled impact, players must carefully plan the timing and frequency of high-impact moves (like domes), possibly removing blocks earlier to restore meter before committing to final moves.
- It creates opportunities for the eco-god card to shine - offering alternatives and healing rather than brute-force escalation. A player may even be able to win a game without causing any destruction at all, e.g. by removing every build they place and simply using a naïve player's builds instead – that could be a possible achievement that we may implement into the game in future.

Eco-Conscious God Powers: A new god card, Gaia, reflects environmentally aware leadership by granting a power to **remove** one block per turn, restoring a portion of the Nature Meter. This introduces a new playstyle focused on restorative action rather than aggressive progression, promoting harmony and balance as viable strategic goals.

New Win/Loss Condition: A core game rule is modified - if a player's Nature Meter depletes entirely, they lose the game regardless of board position or prior progress. This rule adds urgency and gravity to ecological decisions, treating the environment not as a background theme but as a determinant of success or failure.

UI Feedback: Each player's panel now displays their current Nature Meter status via a visible progress bar, reinforcing real-time awareness of environmental health. This not only aids gameplay strategy but also keeps the ecological impact central to the player's attention throughout the game.

Gameplay Culture Shift: The new mechanics encourage players to:

- Build more efficiently and selectively.
- Reconsider aggressive strategies in favour of sustainable play.
- Coordinate building and removal for long-term balance.

- Engage in peaceful, restorative actions even when unnecessary for immediate advantage - reflecting genuine ecological responsibility.

Interfaces and Classes Added in Sprint 3 and Justifications:

BuildObserver (interface):

- **Purpose:** Decouples core game mechanics (event producers)from side effects triggered by build/remove actions (consumers – UI/meter logic); promotes reusability and modularity
- **Open/Closed Principle (OCP):** New observers (e.g., for pollution or health meters in the future) can be added without modifying existing build logic.
- **Dependency Inversion Principle (DIP):** High-level modules (e.g., TurnManager, MeterManager) depend on abstractions (BuildObserver) rather than concrete classes.
- **Justification:** Modifying TurnManager or Board directly to update the Nature Meter or UI would **violate SRP** and tightly couple game logic with environment tracking or UI updates. Instead, BuildObserver lets other components (like MeterManager or NatureMeterUIObserver) subscribe to relevant events while keeping concerns separated and extensible.

MeterStrategy (interface) & NatureMeter (Concrete Class):

- This implementation follows the Strategy Pattern.
- **Purpose:** Encapsulates behavior for different types of meters - not just Nature, but future extensions like PollutionMeter, HealthMeter, etc.
- I didn't choose to use NatureMeter alone as embedding environmental logic directly into NatureMeter would violate OCP, making the codebase rigid to future metrics.
- **MeterStrategy** allows us to plug-and-play different meter types using polymorphism, facilitating reuse and testing.

Future-Proofing Benefit (facilitating extensibility): A GameMeterFactory could instantiate meters dynamically using Strategy objects.

ImpactLevel (Enumeration):

- **Purpose:** Assigns an impact value to building levels in a centralized manner.
- **Design Benefit:** Encapsulation & Maintainability: Keeps magic numbers out of logic, avoiding scattered if statements.
- **DRY Principle:** Central logic avoids duplication across build operations.

Alternative Considered & Rejected:

- Hardcoding impact values into BuildAction or MeterManager would:
    - Scatter impact logic across files.
    - Complicate changes (e.g., increasing Dome's impact from 4% to 5%) requiring changes in multiple places.
    - Obscure the rationale behind numerical impact decisions.

MeterManager (Concrete Class) - implements BuildObserver

- **Purpose:** Calculates and applies changes to a player's NatureMeter based on the ImpactLevel of each building.
- **Justification:** Encapsulates the responsibility of managing meter changes, respecting **SRP**.
- Avoids mixing nature impact logic into unrelated systems like TurnManager or Board.
- **Design Role in Observer Pattern:**
    1. Subscribes to build/remove events from TurnManager.
    2. Applies changes independently of game mechanics, which don't need to know how meters are updated.

Facilitates Extensibility: Supports plugging in different meter strategies (NatureMeter, PollutionMeter) in future without changes to build logic.

NatureMeterUIObserver (Concrete Class) implements BuildObserver

- **Purpose:** Dynamically updates the Nature Meter UI bar whenever a build or remove action occurs.
- **Alternative:** Place inline in UI code.
    1. This would **violate SRP** and **tight-couple** logic with UI rendering.
    2. Observer pattern allows the UI to react to changes without needing access to internal game logic.

Facilitates **Testability:** Can be tested independently from the rest of the game or UI, simulating build events and observing UI changes.

GaiaCard/GaiaPower (Concrete Subclasses) extends GodCard, SpecialPower

- **Design Pattern: Strategy –** enables swappable/scalable behaviour.
- **Purpose:** Implements a new eco-friendly god card where players can optionally remove one block level per turn to restore the Nature Meter.

- **Alternative – modify existing logic to cater for Gaia implementation**
    1. Modifying TurnManager or the GodCard base class to handle Gaia specifically would require branching logic or instanceof, violating **OCP**.
    2. The **Strategy Pattern** allows GaiaPower to be swapped in via SpecialPower interface without changing existing code.

Facilitates Encapsulation: Gaia logic is localized to GaiaPower, without affecting unrelated gods like Artemis or Demeter.

RemoveBuildAction (Class) extends Action

- **Purpose:** Represents a reversible, nature restoring action where players remove a building level, enabled only for Gaia currently.
- **Design Justification:**
    1. Maintains parity with the Action hierarchy, preserving polymorphic behaviour across move, build, and remove actions.
    2. Keeps **removal logic separate** from build logic - respecting **SRP** and **Command-like semantics** in the existing system.

NatureMeterDepletedLoseCondition (Class) implements LoseCondition

- **Design Pattern:** Strategy
- **Purpose:** Introduces a new game-ending condition - if a player depletes their Nature Meter, they lose instantly.
- **Justification:**
    1. Follows Lose Condition Heirarchy implemented in Sprint 2.
    2. Avoids bloating WinCondition or TurnManager with additional logic.
    3. Strategy pattern lets this condition be added alongside others like standard height-based victory without interference.

NatureMeterLabel (UI Component)

- **Purpose:** UI component for displaying a nature meter for each player in the game.
- **Decoupling:** Handles rendering only - does not interact with game logic directly.
- **Design Benefit:**
    1. Allows future enhancement to visual customization (e.g., gradient bars, animations) without touching logic.

Rejected Design Patterns:

While we used Observer and Strategy patterns, we considered but did not use the following:

- **Decorator**: While it could be used to add dynamic behaviour to GodCards (e.g. nature variants), it would overly complicate the card hierarchy and introduce runtime overhead that isn't justified for the limited behaviour of Gaia.
- **Template Method**: Rejected because god powers vary too widely; enforcing a fixed method skeleton (e.g., applyPower()) would restrict flexibility and violate SRP.
- **Mediator**: Considered for centralizing communication between Board, UI, and MeterManager, but current Observer implementation already achieves loose coupling effectively.
- **Command:** already have an Action hierarchy which serves a similar purpose more directly and simply.

Minimum/Maximum Cardinalities Explanation:

- **1..1 Player has 0..* MeterStrategy**
    1. The Player class uses a Map<MeterType, MeterStrategy> to manage meters. This allows a single Player to have **zero or more meters** (e.g., NatureMeter, future HealthMeter, etc.), all accessed via a strategy interface for extensibility.
    2. 1..1 Player: Every MeterStrategy must be linked to some player.
    3. 0..* MeterStrategy: A player can have no meters (e.g., creative mode), one meter, or multiple meters.

This design enforces extensibility, as instead of Player being concerned with specific implementations of meters in the game (NatureMeter), it interacts with them through this interface, enabling the addition of different game modes in future with players having additional meters, such as a health bar, or no meters at all.

- **1..1 TurnManager uses 0..* BuildObserver**
    1. TurnManager has a List<BuildObserver> and calls notifyBuild() and notifyRemove() to inform all observers about changes (e.g., a build action).
    2. 1..1 TurnManager: There is one manager coordinating the turn sequence.
    3. 0..* BuildObserver: It can operate without any observers (no UI tracking), or with multiple observers listening to build events (e.g., Gaia god effects, NatureMeter UI).
- **1..1 NatureMeterUIObserver updates 0..* NatureMeterLabel**
    1. The NatureMeterUIObserver has a Map<Player, NatureMeterLabel> which can store any number of labels, one per player.
    2. 1..1 Observer exists: There is one instance of the UI observer.
    3. 0..* NatureMeterLabel: There may be no players (and thus no labels), or many players, each with their own NatureMeterLabel.
- **1..1 Game sets 0..* MeterStrategy**

1. The Game class, during initialization, sets up meters for players via player.setMeter() – currently only sets nature meter.
2. 1..1 Game: There is one central game instance doing the setup.
3. 0..* MeterStrategy: Depending on the game mode, the Game is open to assigning zero, one, or many meters per player (e.g., nature meter, health meter, mana meter, etc.).

- **1..1 NatureMeterUIObserver uses 0..* Player**
  1. The NatureMeterUIObserver's map is keyed by Player, meaning it references multiple players' nature meters to update their labels.
  2. 1..1 Observer: A single observer processes updates.
  3. 0..* Player: It may track no players, or all players, depending on game setup.
- 1..1 BoardUI creates 2..2 NatureMeterLabel
  1. BoardUI creates a NatureMeterLabel object for each player in the (currently) 2 player Santorini game, via the SetEcoMeters method

Reassigned Responsibilities from Sprint 2:

- TurnManager was extended to accommodate the optional remove build phase after a build when Gaia is active.
- Player was modified minimally to include a NatureMeter, but logic stayed encapsulated in NatureMeter.
- No previously implemented god powers were changed - the Strategy pattern enabled Gaia integration without side effects.

Alternative Designs Considered:

Placing impact logic in Player or Board:

- Violates SRP, leads to bloat and tight coupling

Centralizing model and UI in a monolithic NatureManager:

- violates **SRP** and **DIP**; decreases testability and increases rigidity.

Using hardcoded values in BuildAction for impact calculation

- breaks encapsulation, leads to duplication (violation of DRY practices).

Using instanceOf for GaiaPower handling:

- Violates Open/Closed Principle; not scalable for future god powers

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

CRC Card Explanation:

Class: GaiaPower

Superclass: SpecialPower

Responsibilities:

1. Modify actions to allow valid block removal (REMOVE_BUILD) actions.
2. Provide legal remove options for the current worker - includes adjacent and own cell if they contain blocks/domes.
3. Declare god capability for allowing removal actions via allowsRemove().

Collaborators

- Action / RemoveBuildAction: To encapsulate the removal operation.
- ActionType (enum): To identify REMOVE_BUILD action type.
- Worker: To determine position and ownership.
- Player: For associating valid players for actions.
- Board: To query neighboring cells and current game state.
- Cell: To inspect for buildings/domes and determine remove eligibility.
- Building: For height checks to validate removable structures.
- TurnManager (indirectly): Uses allowsRemove() to determine if removal is an option for the god card.

To summarise, this extension demonstrates how game mechanics can promote environmental values without sacrificing depth or competitiveness. It required thoughtful architectural changes:

- Observers for decoupling.
- Strategy pattern for extensibility.
- SRP-aligned components for logic, UI, and state.
- A scalable meter system for future values (e.g., pollution, health).

By embedding *Protecting the Environment* into the strategic and visual fabric of the game, this extension meaningfully brings human values into play, altering both the strategic landscape and the ethical dimension of Santorini, enriching the gameplay with moral reflection and diversity of play styles.

Executable Instructions:

Executable Description:

- The executable produced is in the form of a .jar file. Opening it by double clicking on the file initiates the MainMenu screen of the Santorini game, at which point the game may be initiated via the 'Play' button or exited via the 'Quit' button.
- The executable enables the running of the complete implementation of the Santorini game produced by Team Santorinians (005) in Sprint 2. It also includes all the proprietary extensions made by me in Sprint 3.

Platform Compatibility:

- This process has been tested as functional on MacOS, a very similar can be followed on Windows.

How to Create an Executable from Source Code:

- In Intellij, click on File --> Project Structure
- In Project Structure window, select Artefacts --> Plus Symbol (+) --> JAR --> from modules with dependencies
- Set main class in the artefact. This is the class that contains the psvm method --> OK
- Set an output directory, e.g. output executable JAR to the desktop
- Select Build --> Build Artefacts --> select artefact you want to build --> Build

OpenJDK version "22.0.2" (Corretto 22.0.2.9.1) and Ensure you are using Java 17 or above to run the project successfully.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

Reflection:

Extension 1: God Card/Power

This extension was partially easy to implement due to the existence of abstract structures like GodCard and SpecialPower, which we introduced in Sprint 2. These abstractions allowed me to add new god cards without modifying existing god implementations—adhering to the Open/Closed Principle (OCP) in most places.

However, significant difficulty arose in integrating new god powers into the existing TurnManager, which had evolved into a God Class in Sprint 2. It contained large, complex methods responsible for handling movement, building, and turn logic. Due to poor documentation and the fact that I hadn't worked on this class that much in Sprint 2, understanding and working with it was time-consuming. Additionally, code duplication and tightly coupled logic introduced code smells that made it hard to extend cleanly.

If I were to redo Sprint 2, I would introduce a PhaseHandler interface with concrete implementations like MovePhase, BuildPhase, and RemoveBuildPhase. This would enable clear separation of responsibilities and make it easy to plug in new god powers that involve new phases, without needing to touch TurnManager directly. This also better supports OCP and Single Responsibility Principle (SRP).

Another design flaw from Sprint 2 was the way god cards were managed. God creation and assignment logic were tightly coupled within GodCardPool, which made introducing new gods like Zeus harder than it should have been:

- Implementing Zeus required changes across multiple classes, violating OCP.
- The system lacked a plug-and-play architecture for new gods.
- God logic was not testable in isolation due to low modularity.

To address this, I introduced a GodCardFactory and GodCardProvider, which decouples god creation from usage. This has greatly improved extensibility going forward.

Furthermore, Sprint 2 design assigned the responsibility of interpreting build highlights to the Player class. To support Zeus (who builds under himself), I had to modify highlightBuildableCells() and getBuildableCells() in Player to account for special god logic. This is a clear violation of OCP, as each new god that alters cell selection will now require modifying the Player class. In hindsight, the Player class should have delegated this responsibility to its GodCard, allowing each god to override methods like getValidBuildCells() as needed. This would preserve polymorphism and improve modularity.

To summarise, if Sprint 2 had:

- Delegated move/build logic to GodCard/SpecialPower
- Made Player agnostic to god-specific rules
- Avoided making TurnManager a central controller

…then Sprint 3 god extensions (Zeus, Demeter, Gaia) would have been much easier, and entirely free of code modification in existing classes.

Extension 2: Timer

This was the most straightforward extension to implement. Since no timer functionality existed previously, I had full control to design the solution cleanly from scratch using SOLID principles. I created:

- GameTimer and GameTimerManager to handle timer logic
- TimeOutHandler to register player timeouts and display game-over conditions
- TimerLabel to display time in the UI

The only modification to existing code was in TurnManager where I added start() and pause() calls in the startTurn() and endTurn() methods respectively. These were small, isolated changes and did not introduce any design violations.

This extension highlighted the benefit of designing new functionality from the ground up with modularity in mind. Because the rest of the game logic was largely decoupled from timing logic, the timer system could plug in smoothly.

Extension 3: Human Value Extension

After the work I did to make Zeus work correctly, the Gaia god power (representing a human value extension with environmental themes) was very easy to implement. I followed the ZeusPower model:

- Gaia's logic was encapsulated in a new GaiaPower class
- I implemented a RemoveBuildAction to support cell removal
- All cell-removal logic was triggered in its own method, following the structure already used for build and move actions

However, I once again had to add another method (handleWorkerSelectionForRemoveBuild) in TurnManager to process the new remove phase. This repeated the same issue from Zeus: each new phase requires modifying TurnManager.

If I were to redo Sprint 2, I would build a PhaseHandler framework with a Phase interface, which each game phase (move, build, remove) would implement. This would remove phase-handling logic from TurnManager, allowing us to easily add or modify phases in the future.

Despite this flaw, the existing abstraction work done for Zeus made Gaia's implementation clean and quick. This illustrates that small refactors can have compounding benefits for extensibility.

The other aspect of the human value extension – nature meters, had not been considered in Sprint 2 design so, while it gave me a 'fresh plate' to work with in the fact that I didn't have to sift through any existing code to implement this feature, I did have to implement an

entirely new SOLID design, which took considerable time – though it was preferable to refactoring code.

Updated File Structure (Git):

- Added timing, meters packages to facilitate placement of some newly added classes for Extensions 2/3.
- Added docs/crc package to docs. Within that, a sprint 3 package containing the drafts/final of each CRC card produced.
- Docs/deliverables package – within sprint 3 directory, deliverables document for sprint 3 is placed.
- Docs/uml/Sprint 3 contains all drafts/final of UML class diagram for sprint 3 implementation.
- The rest of the structure has remained the same from Sprint 2, for further guidance, please review the README on the repository.

FIT3077 – Software Engineering: Architecture & Design
Name: Ubaid Irfan
Student ID: 33886466
Date: 31/05/2025

References:

[1]
"Adding a countdown timer to a JFrame?" *Reddit.com*, 2018. https://www.reddit.com/r/javahelp/comments/8ezvxw/adding_a_countdown_timer_to_a_jframe/ (accessed May. 22, 2025).
[2]
CodeNMore [YouTube Channel] 2015. https://www.youtube.com/watch?v=w1aB5gc38C8&ab_channel=CodeNMore (accessed May. 22, 2025).
[3]
"How to Use Swing Timers (The Java™ Tutorials > Creating a GUI With Swing > Using Other Swing Features)," *Oracle.com*, 2024. https://docs.oracle.com/javase/tutorial/uiswing/misc/timer.html (accessed May. 22, 2025).
[4]
M3832 [YouTube Channel], 2025. https://www.youtube.com/watch?v=s9flVs2ByXQ&ab_channel=M3832 (accessed Jun. 01, 2025).

AI acknowledgement: For the code base, ChatGPT o3 (https://chat.openai.com) and lesser models were used to come up with initial ideas as well as bug fix over roughly 50 iterations. Here's the list:

- Suggesting an alternate design decision: using TimeOutHandler instead of a TimeOutLoseCondition which I implemented prior. This was better as my design required checking timeouts at various stages of the game, which proved to be inconsistent/unreliable.
- Fixing math logic in DefaultPlayerTimer causing early timeouts
- Fixing timeout label resetting after the game was over

- An initial strategy to follow for the Timer functionality, in conjunction with other methods I found online (mentioned).
- Designing the NatureMeterLabel – badly, I had to make major revisions so it looked good.
- Introducing impact level enum, NatureMeterUIObserver
- I also used it to generate an initial design plan to follow to implement the human value extension – mostly to conceptualise the amount of work it would take.
- Fixing a logic flaw which allowed workers on a level 3 building to advance onto a dome
- Fixing logic flaw in BasicWinCondition which was previously triggering wins when there was a worker on a level 3 build, instead of checking if that worker moved up to it.

Reference (from Sprint 2 base code) – I moved from each class to here, as it was cluttering up the project files.

```
Our team used chatGPT(https://chatgpt.com) to speed up the process of
writing JavaDocs.
The tool was used to replace old comments on functions and classes
that we used to explain how the function/class works to the team.
We modified some misunderstanding of the functions of the generated
documents.
```

I used GitHub Copilot to generate javadocs for most public/protected classes/methods. I just used the first generated iteration and refactored accordingly.

AI generated snippets were carefully reviewed, then implemented into our existing files, and manually verified that every patch compiled cleanly and passed our informal play tests.

Artwork was also generated using ChatGPT – below is a list of each newly introduced piece of artwork which was generated:

- Main Menu screen (‘default_bg.jpg’)
- WinScreen background (‘game_over.png’)
- Gaia god background label (‘gaia.jpg’)

Generative AI (ChatGPT) was used in quickly giving me a rough template to follow for the rationale section of this document, although I had written down many notes/plans throughout development. For this rough template, it was only used with 0-2 iterations max, where I fed it some code snippets and my design intentions. With the reflection part I followed the opposite approach – where I wrote the whole thing up first and then fed it to AI to ensure my grammar and reasoning was correct – I wanted to ensure ChatGPT had the proper context, though I refactored after this 1 iteration generation anyway.