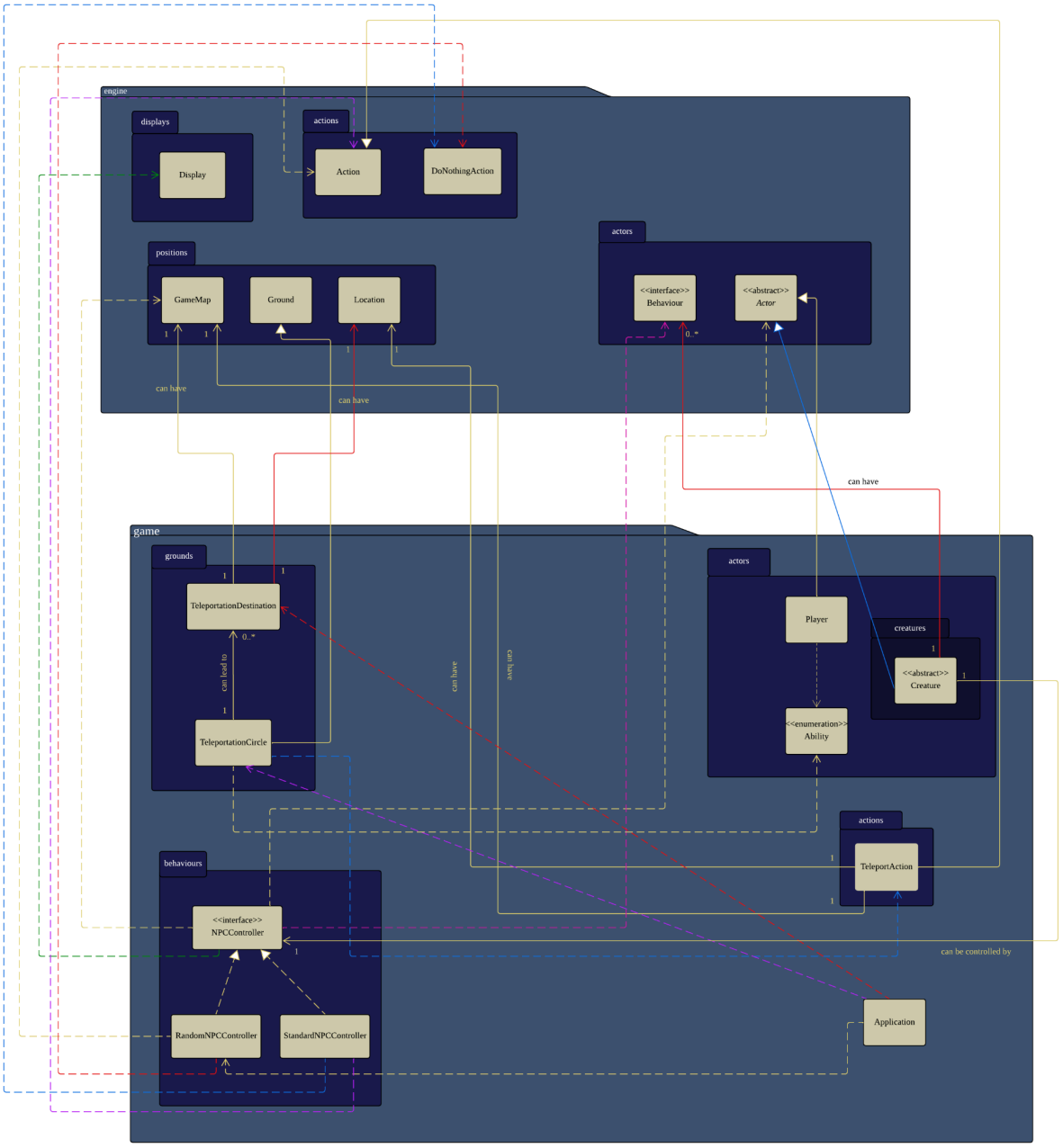
Design Rationale 3 - ELDEN THING: The Valley of the Inheritree

FIT2099 Assignment 3

Adji Ilhamhafiz Sarie Hakim	**33212422**
Anfal Muhammad Ahsan	**34309225**
Arielle Dela Cruz	**34086315**
Mohanad Al-Mansoob	**34303715**

REQ 1 - Limveld



Design Goals

1. Efficiently model teleportation such that destination mapping and actor relocation are encapsulated within dedicated classes, following single responsibility.

2. Avoid downcasting by relying on polymorphism and interfaces to ensure type-safe behavior selection and execution.

3. Enable reusable behavior selection logic across NPCs by abstracting control flow into a configurable NPCController, supporting both priority-based and randomized decision-making without duplicating code.

Feature summary

1. This requirement introduces teleportation, allowing actors with the appropriate capability to move instantly between linked destinations, either within the same map or across different maps.

2. Additionally, it enables flexible NPC behavior control: NPCs can now act based on either a fixed priority order or a randomized selection of behaviors, configurable per instance. This supports more dynamic and varied NPC behavior across the game world.

Design Decision

We chose a modular and extensible design that adheres to core object-oriented principles. NPC behavior is managed through the NPCController interface, with two concrete implementations: StandardNPCController (priority-based) and RandomNPCController (random behavior selection). Teleportation functionality is encapsulated via the TeleportDestination and TeleportationCircle classes, keeping concerns separated.

Pros	**Cons**
Separation of Concerns: Each class has a single responsibility e.g., TeleportationCircle manages allowable actions, while TeleportDestination simply stores destination data.	Increased Complexity: More classes and interfaces mean more boilerplate and a slightly steeper learning curve.

Adheres to SOLID principles: Single Responsibility Principle: Each class does one thing Open/Closed Principle: New NPCController strategies can be added without modifying existing code Dependency Inversion Principle: High-level actor logic can depend on the abstract NPCController interface	Overhead for Simple NPCs: For simple use cases, the NPCController abstraction may be unnecessary.
Easily Extendable: New teleport mechanics or behavior controllers can be added with minimal impact.	

Justification

By introducing separate classes like TeleportationCircle and TeleportDestination, we achieve clear separation of concerns. The teleportation logic is not tightly coupled with the actor or map classes, which keeps the design modular and easier to manage. TeleportationCircle dynamically manages its valid destinations through an internal ArrayList, eliminating the need to hardcode teleportation links. This makes it easier to add, remove, or modify destinations without altering core logic, improving maintainability and scalability.

Moreover, using an NPCController interface with StandardNPCController and RandomNPCController implementations allows per-instance behavior flexibility. This enables different NPCs to act with varied strategies, some selecting behaviors in order of priority, others choosing randomly without modifying the NPC class itself. This aligns with the Open/Closed and Single Responsibility Principles, as new behavior strategies can be introduced independently. Altogether, the design promotes flexibility, extendability, and a cleaner separation of logic that supports future features or modifications.

Alternative Design and Comparison

Design	Pros	Cons
Chosen Design: Uses an NPCController interface with separate implementations (StandardNPCController and RandomNPCController) and a distinct TeleportDestination class.	- Extensible and easily supports new behavior strategies - Adheres to SOLID principles, especially Single Responsibility and Open/Closed.	- More complex with additional classes and interfaces
Alternative Design: No NPCController abstraction; random behavior handled by a dedicated random NPC subclass. No separate TeleportDestination class; destinations stored directly in TeleportationCircle.	- Simpler and easier to implement initially	- Less flexible, behavior logic duplicated or hardcoded in subclasses - Violates SOLID principles, reducing maintainability and extensibility

SUMMARY

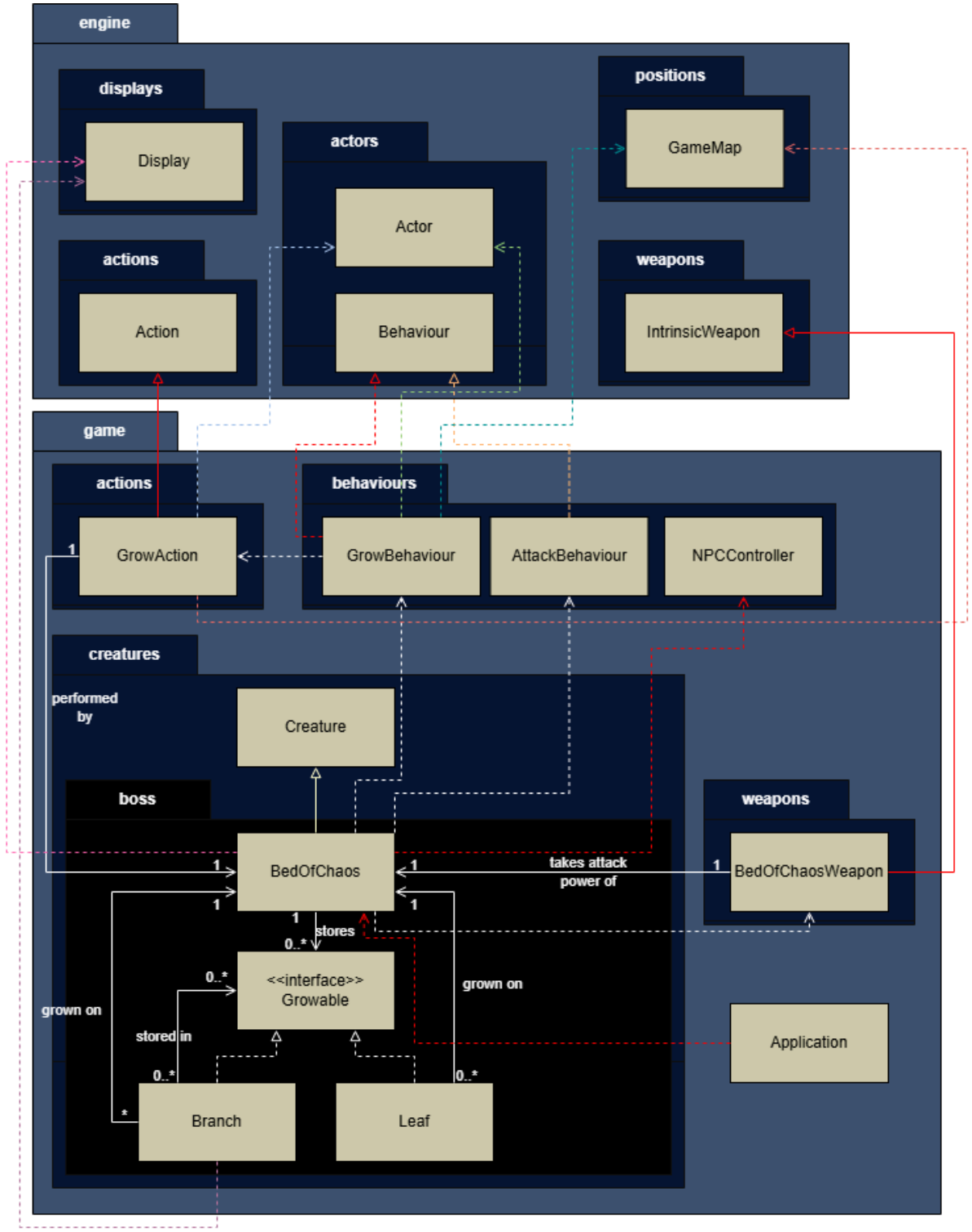
Our chosen design effectively balances modularity, clarity, and flexibility, making it well-suited for managing teleportation and NPC behavior in the game. By separating teleportation logic into distinct classes such as TeleportationCircle and TeleportDestination, the system remains clean and easy to maintain. Additionally, the use of the NPCController interface with multiple implementations allows different NPCs to behave according to different strategies without code duplication or invasive changes to existing classes.

The implementation relies on the TELEPORT capability enum to control which actors are allowed to teleport, ensuring clear permission handling. Teleportation actions are encapsulated in the TeleportAction class, which handles moving actors across maps seamlessly. Meanwhile, NPCs' behaviors are driven by either the StandardNPCController or `RandomNPCController`, offering predictable or randomized action selection. This design keeps responsibilities well distributed and the system adaptable.

Several SOLID principles are followed in this design: the Single Responsibility Principle is honored by assigning distinct responsibilities to teleportation and behavior control classes; the Open/Closed Principle is observed as new behavior controllers or teleportation features can be added without modifying existing code; and the Interface Segregation Principle is implemented through the NPCController interface, allowing flexibility in behavior management.

Looking ahead, this design supports future extensibility well. New teleportation features, such as cooldowns or teleport traps, can be introduced by extending the teleportation classes without affecting unrelated code. Similarly, new NPC behavior controllers can be added easily to enrich gameplay dynamics. This modular approach ensures that the system can grow organically as new game mechanics are conceived.

REQ 2: Bed Of Chaos



Design Goals

1. Model Bed of Chaos's turn-based growth as a n-ary tree of branches and leaves.
2. Separate responsibilities between grow behaviour, damage calculation etc in their own classes.
3. Make use of the existing classes such as IntrinsicWeapon to incorporate logic.
4. Avoid any use of downcasting.
5. Enable potential addition of new growable items (eg. Flower) without modifying existing code.

Feature Summary

This feature introduces a boss, "Bed Of Chaos", that grows at each turn such that:

- The boss itself always grows one new branch or leaf (50% branch or 50% leaf).
- Every branch that already exists in that turn grows once.
- Any branch created in this turn does not get to grow until the next turn.
- The boss grows when an actor is not adjacent to it. If an actor is adjacent to it, it does not grow and attempts to attack the actor.

Design Decision

We chose to implement a SOLID-based design that introduces a Growable interface, separate Branch and Leaf classes, GrowBehaviour and GrowAction with manage growth logic.

Pros	**Cons**
Clear separation of concerns: Boss only orchestrates growth but actual grow and damage calculation is handles in Branch and Leaf classes.	Adds many classes which may be more difficult to track and manage.
Easy extension: To add a new Growable type such as Flower or RottingLeaf, there is no need to modify existing classes.	Growth behaviour is spread across several classes which may be more complex than just having a single class.
Adheres to SOLID principles: Branch/Leaf have just one responsibility, growth scheduling is handled in GrowthBehaviour and GrowAction.	Distributed logic in separate classes may make it harder to track.

Justification

By introducing a Growable interface, we are able to keep branch and leaf logic separate from boss. If a new growable is to be added, we only need to create a new class that implements Growable. The boss's code remains unchanged.

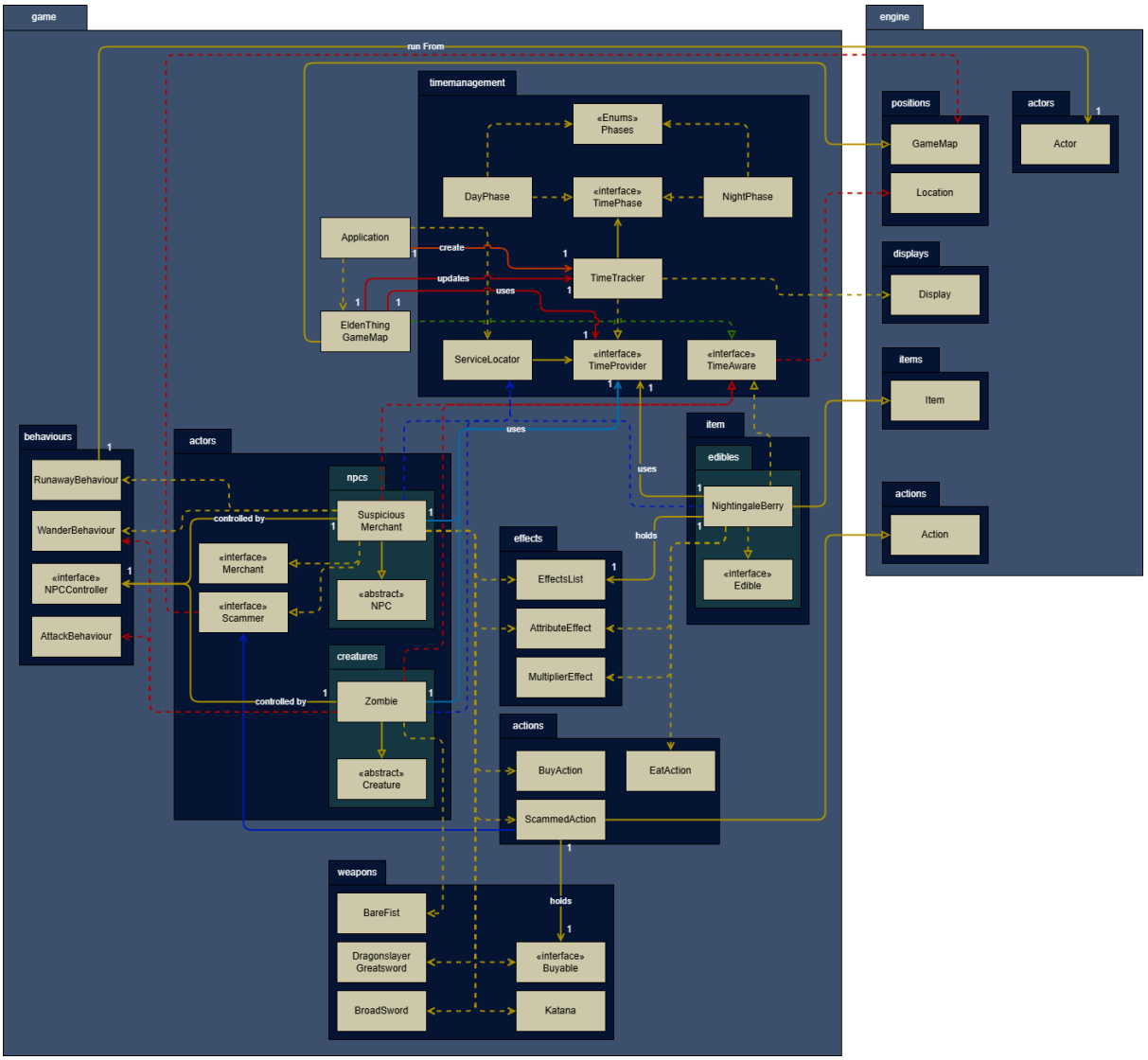
Design Alternatives and Comparison

Alternative Design	**Pros**	**Cons**
Chosen design: Growable interface + Branch + Leaf + GrowBehaviour + GrowAction	- Full separation of concerns: - Easily add new Growable types without changing existing code	- Need to introduce more classes as well as interface - Growth behaviour is spread across many classes, may be difficult to track
BedOfChaos + Branch + GrowBehaviour classes only to handle growth	- Very few classes, easy to read and understand - Growth logic lives directly in BedOfChaos and Branch - No extra classes needed	- Violates SRP as BedOfChaos handles many different things at once. - Hard to extend new growable types.
Let BedOfChaos implement Growable	- Fewer total classes - No need for separate Branch and Leaf classes	- Boss becomes a God class, handling many different things at once - Difficult to introduce new growable types - Violates Open-Closed (hard to extend)

Summary

Our chosen design builds an n-ary tree of Growable nodes under a single root (BedOfChaos). Each Branch recursively sums its own subtree's attack power and grows once per turn. Each Leaf heals the boss when it grows.This approach fully adheres to the SOLID principles, by separating concerns in each class such as Branch managing its own children, Leaf and its healing logic, GrowAction executing growth following SRP, and being able to introduce a new Growable without modifying existing classes (OCP), as well as high-level boss code only depending on the Growable interface and not on concrete Branch or Leaf classes. This design makes it easy to extend, test, and avoids downcasting.

REQ 3: Day / Night System



Design Goal

To implement a turn-based day-night system that would change the behaviour of the game based on the state or time it is currently (e.g. when night arises, zombies would appear from the cursed ground) and implement special game entities features such as NPCs that spawn at night and disappear at day.

Feature Summary

This implementation would introduce a detailed system of time tracking, where we previously only progressed through time by using ticks, where every tick is a single turn. We wanted to modify it such that it would have a sense of Day/Night, which would allow extra features to be implemented in the game.

Core Feature: Day/Night implementation

The core concept of this requirement is to allow the game to experience the progress of days, where after a certain number of turns, it will turn to Night or Day. Which would help us implement more advanced features, such as special events during the Day or Night. To implement the feature that holds the core of this functionality, we would need to create a TimeTracker, which would store the current number of turns and what state of time phase is currently followed, with an interface that would make the objects aware of the time. Then we would need some form of variable or something that we can track that tells the time.

Design Alternatives

The design to implement the core features can be separated into the following methods:

Method	Pros	Cons
Extend the GameMap and modify the tick function such that it calls the TimeTracker tick method to increase the time	High-level integration ensures time advances regardless of specific entities	Requires modifying core game loop logic (if tightly coupled)
	Keeps TimeTracker centralised and decoupled from game entities	Difficult to share the variable that holds the current phase or turn
	Fits naturally if GameMap is already managing turns	
Attach the TimeTracker to a single GameEntity that would essentially be a Clock Entity	Easy to save time phase state	Time won't progress if the entity dies or is removed
	Keeps the core logic clean (GameMap or World)	Risk of bugs if entity isn't ticked consistently
	Easier to add or remove the feature from the game	Less intuitive to debug compared to central tracking
Extend the World class itself and make a custom world class where we could override the tick method, such that the tracker will increase its turn on the highest level possible	Very high-level control ensures time is always synced with the world tick	Tight coupling to engine/world lifecycle — might be hard to maintain or test
	Centralised time control without polluting entities or the map	More invasive changes, especially since World Class is part of the engine
	Most robust if the world handles turn logic	Might not be portable to other systems/maps if each needs different logic

Justification

Referring to the UML diagram, we've decided to use a custom GameMap (EldenThingGameMap) that extends the original GameMap. This is because the children who need the core feature are unrelated entities, which would require a high-level approach, such that the tick would be called for everyone. However, this method ran into the issue that the Tracker is quite hidden from its lower classes that need it. To mitigate this issue, we used a ServiceLocator which acts similarly to a Singleton but it merely stores the Tracker, where in this case a TimeProvider, since we also need to maintain SOLID principles where the ServiceLocator are able to be used by similar TimeTracker (such as a test TimeTracker or other type). By having a ServiceLocator we can pass it along to all the other classes that need to "see the time" without the need for an injection via a constructor or a setter, which would increase the complexity of the codebase.

By implementing the features inside these classes, we can maintain our SOLID principle since every class still has its responsibilities, such as the GameMap would call the tick for all GameEntities, we simply add one more thing to call the tick. Followed by the TimeTracker that manages the time and keeps track of whether it's Day or Night, and is passed along using ServiceLocator to remove the need for constructor or setter injection to the classes that need to look at the time.

Sub-Features: Implementing Higher Class to Child Classes

Once we've implemented the core features, we need to find a way how to implement the higher class into the Child classes, as all of them needed to hold the same TimeTracker object yet its difficult with the restriction to the edit access of the engine and the increased complexity if we used constructor or setter injection.

Design Alternatives

Method	Pros	Cons
Use a ServiceLocator to access a shared TimeProvider interface	Clean and minimal integration for child classes	Global access can hide dependencies, harder to track what a class uses
	Works well when the engine doesn't support dependency injection	Slight violation of Dependency Inversion Principle (since classes pull dependencies rather than receive them)
	Can depend on an abstraction (TimeProvider), preserving SOLID	
Pass the TimeProvider or TimeTracker directly to child classes via constructor	Fully SOLID-compliant, clear dependencies and fully testable	Requires engine or factory control over object creation

	No global state, explicit usage improves maintainability and debugging	It can be difficult or impossible in some engines where entities/components are created dynamically or via scripting
	Easy to mock for unit testing	
Use an Observer or Event system where TimeTracker notifies registered listeners	Decouples TimeTracker from its consumers, no hard dependency required	Adds complexity — need to manage registration/unregistration
	Objects only react when something changes	Not great if child classes need to check the current TimePhase anytime
	Allows for more dynamic reactions (e.g. event-driven game mechanics)	Harder to track when and where notifications happen, especially in large games

Justification

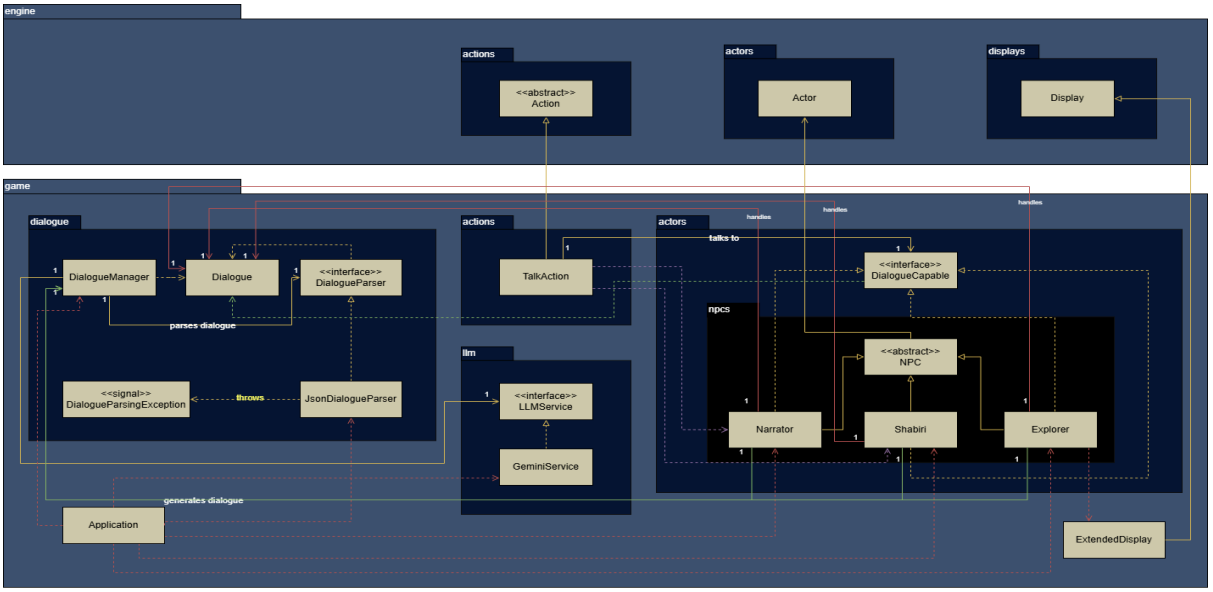
Continuing the core Feature justification. We use the ServiceLocator method since we needed the classes that are time-aware to be able to see the current time phase. We did not use the other two methods as we would require access to the engine, considering the time constraint; however, given time, it might be possible to use injection, though we could not tell at this moment. We tried to use an Event system during our first iteration of the implementation; yet we're met with a challenge where the number of class that are registered to the TimeTracker would be huge bordering the whole GameEntity which if the game would continue its development, the class would has the complexity of all the objects created in the game (e.g. grounds, actors, items, etc)

By using a ServiceLocator, we can maintain SOLID principles within the time constraints where each class has their responsibility; ServiceLocator provide TimeProvider for the child to use, TimeTracker updates and holds its time-phase, and each child class has their implementations that need the core features. Moreover, with the way ServiceLocator is implemented, it follows Open/close to TimeProvider, which is an interface, so once we needed another type of TimeTracker (FakeTime or DebugTime), we could just use the same interface, which allows us to modify the Tracker without changing TimeTracker itself.

Conclusion

By making the decisions that we made, our team are able to implement a feature that follows SOLID principles using ServiceLocator and placing the high-level class inside a custom GameMap where we are able to manipulate time within TimeTracker and allow the children to use that implementation to create advanced features.

Req 4: LLM-Powered NPC Dialogue System



Design Goals:

1. Enable immersive, natural language conversations between players and NPCs using an LLM (e.g., Gemini)
2. Ensure modularity and extensibility, especially for supporting new LLMs or dialogue formats.
3. Preserve clean separation of concerns and alignment with SOLID principles.
4. Ensure individual NPCs can have distinct personalities, prompts, and conversation logic.

Feature Summary:

This system integrates a Gemini LLM API to allow dialogue between player and NPCs. The dialogue logic is encapsulated in a Dialogue class specific to each NPC. The system supports different NPC behaviours and personalities, customizable prompt structures, and extensible support for both dialogue data formats (via parsers) and LLM providers.

Feature 1: LLM Service Abstraction

At the heart of this requirement is the integration with a large language model (LLM) API, currently Gemini, which generates natural language responses for NPC dialogue. This LLM service is responsible for handling communication with the API, sending prompts, receiving responses, and possibly managing aspects like tokens, or model versions.
Since AI models and APIs evolve quickly, and other providers (e.g. OpenAI) might be used in the future, it is important to isolate this LLM-specific logic into a dedicated component. This keeps the rest of the dialogue system independent from the details of any one API.

Design Decision

An LLMService interface defines the contract for generating text, while a concrete GeminiService implements the Gemini-specific API logic.

Justification

This allowed future LLMS (e.g., OpenAi, Claude, etc.) to be plugged in by simply implementing the same interface. This adheres to the Open/Closed Principle (OCP) and promoted Dependency Inversion Principle (DIP). This also adheres to Liskov Substitution Principles, as any future LLMService implementation can safely be used in place of GeminiService without altering how DialogueManager works.

Design Alternatives and Comparison

Approach	**Pros**	**Cons**
LLMService interface (chosen)	Swapabble LLMs, testable, scalable	Slightly more setup
Hardcoded Gemini logic in NPC	Simpler to start with	Tightly coupled, harder to switch LLM later

Feature 2: Dialogue Parsers

While the current implementation uses JSON format to generate a dialogue, designers may want to use different file formats such as XML, or CSV in the future. To keep the system flexible, parsing dialogues is delegated to parser classes.

Each parser reads the dialogue format and converts it into usable data structure (Dialogue) that the game logic can use. This separation means adding new formats requires writing a new parser, without touching core dialogue or game code.

Design Decision

Dialogue content is parsed using a DialogueParser interface, with a concrete JsonDialogueParser.

Justification

The system can easily support new dialogue formats by implementing additional parsers. This keeps Dialogue free from format-specific parsing logic, aligning with Open/Closed Principle. The system also adheres to LSP by ensuring that each interface (LLMService, DialogueParser) can be substituted with any of its concrete implementations without affecting program correctness. For instance, a JsonDialogueParser can be replaced with an XMLDialogueParser, as long as they follow the expected behaviour. This enables flexibility and safe extension of functionality.

Approach	Pros	Cons
Parser interface (chosen)	Easy to add new formats	Requires interface setup
Hardcoded JSON parsing	Simple, fast for one format	Not reusable, format-locked

Feature 3: Dialogue System and NPC Modularity

To support rich, personalized conversations, each NPC may require its own tone, behaviour, and conversation logic. This necessitates a modular system where NPCs can define custom prompts and manage independent dialogue rounds.

Design Decision

A Dialogue class models structured conversations between the player and NPCs, including options, responses and dialogue progression. The Dialogue Manager acts as a controller handling prompt construction, interaction with the LLM, and parsing the response into usable Dialogue objects.

Each NPC can define their own prompts, and optionally provide different dialogue flow structures, all without modifying the core engine.

Justification

This modular design allows NPC behaviour to vary significantly without requiring any changes to dialogue mechanics, For example, an angry guard NPC can respond, while a friendly merchant may offer more cooperative dialogue that is all determined by custom prompts.

This aligns with:

- Single Responsibility Principle (SRP): Dialogue, DialogueManager, and parser classes each handle a specific aspect of the system.
- Strategy Pattern: Different prompts can define distinct dialogue “strategies” per NPC.

Design Alternatives and Comparison

Approach	Pros	Cons
Modular DialogueManager + prompt design (chosen)	Highly flexible, easy to extend NPC behaviour	Slightly more complexity in prompt management.
Hardcoded NPC dialogue trees	Easy to control and test	Not scalable, harder to personalise

Extended Display for Free-Text Input

The standard Display class provided by the FIT2099 engine only supports reading a single character using readChar(). However, the Explorer NPC requires full natural language input from the player (i.e., entire lines of text). This is because the player's input acts as a question that is passed to the LLM for generating immersive, lore-rich dialogue.

Design Decision

To meet this requirement, a subclass ExtendedDisplay was created, which extends Display and adds a readline(String prompt) method for reading full lines of user input.

Justification

This approach maintains backward compatibility with the existing Display system while adding the required functionality. It adheres to the Liskov Substitution Principle (LSP). ExtendedDisplay can be used anywhere a display is expected without breaking behaviour, while still offering more capabilities. It also follows SRP by isolating the user input enhancement into a focused subclass, rather than modifying the engine's core Display class. This separation ensures that components like Explorer can handle natural language dialogue seamlessly without tightly coupling to low-level I/O mechanisms.

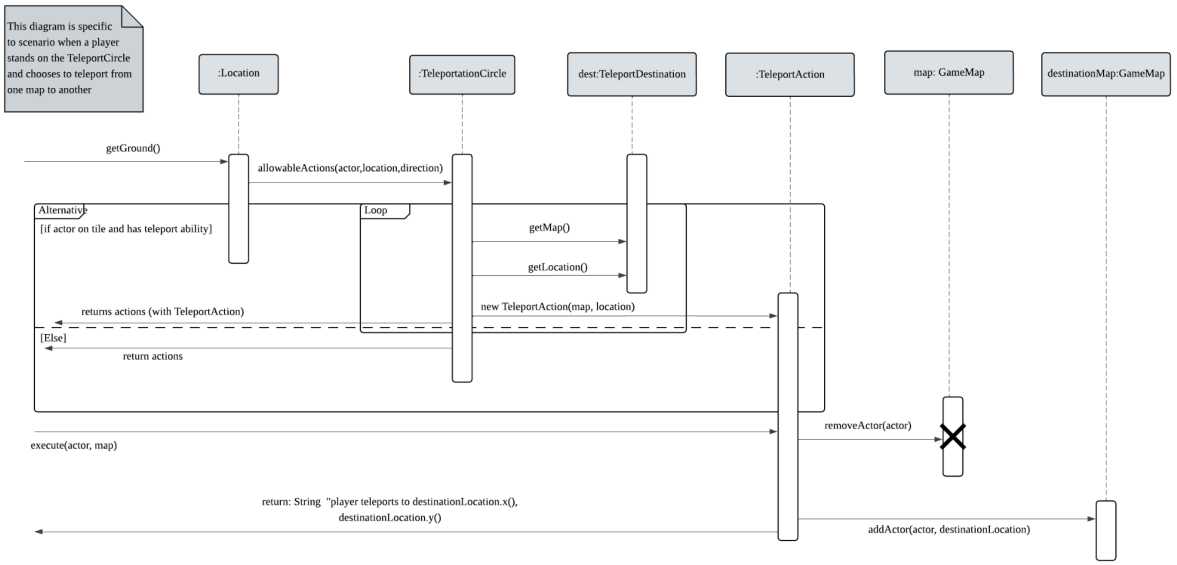
Summary

This dialogue system is modular, extensible, and aligns with SOLID principles. Each NPC is empowered with unique personality-driven conversations. The architecture supports future expansion, such as:

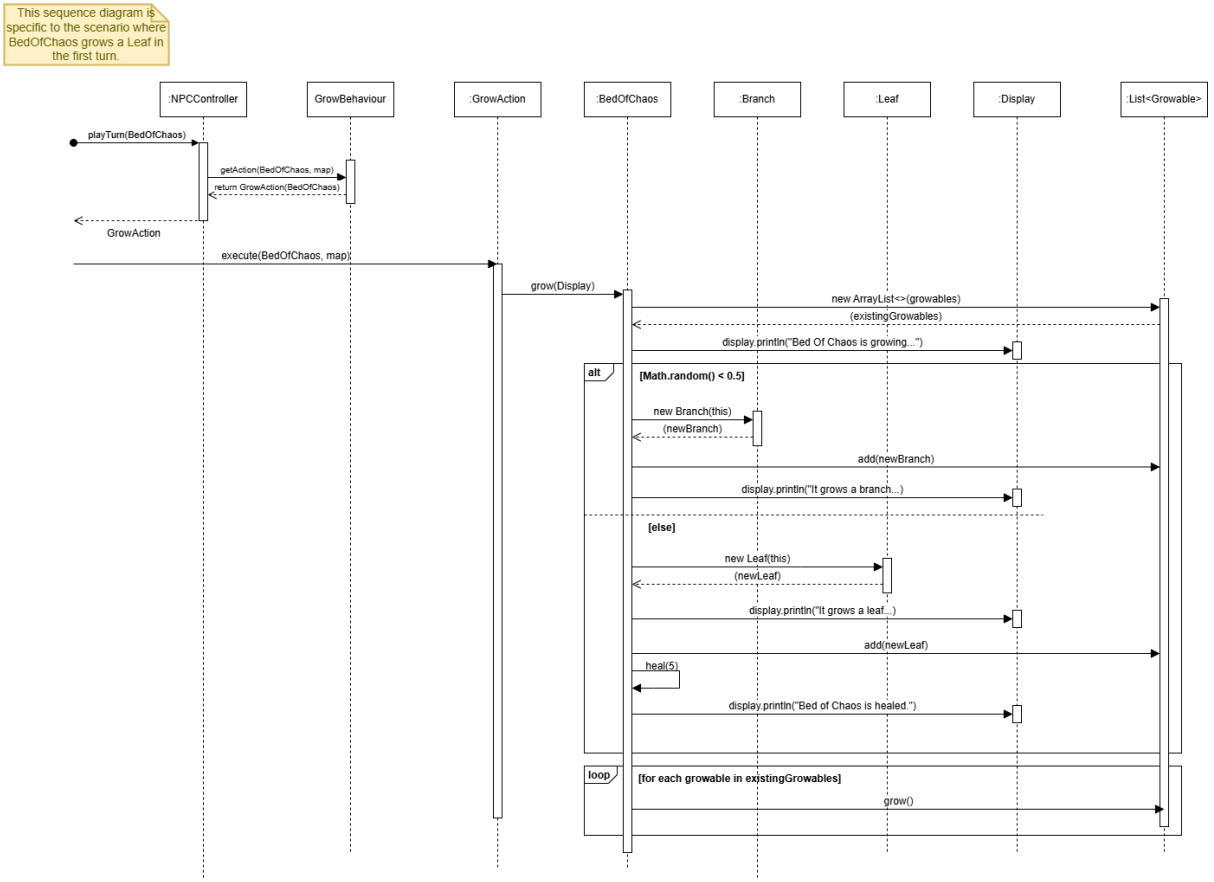
- Switching or adding new LLM providers via the LLMService interface.
- Supporting new dialogue file formats via the DialogueParser interface
- Introducing new DialogueCapable NPCs with minimal changes to existing code

Addendum

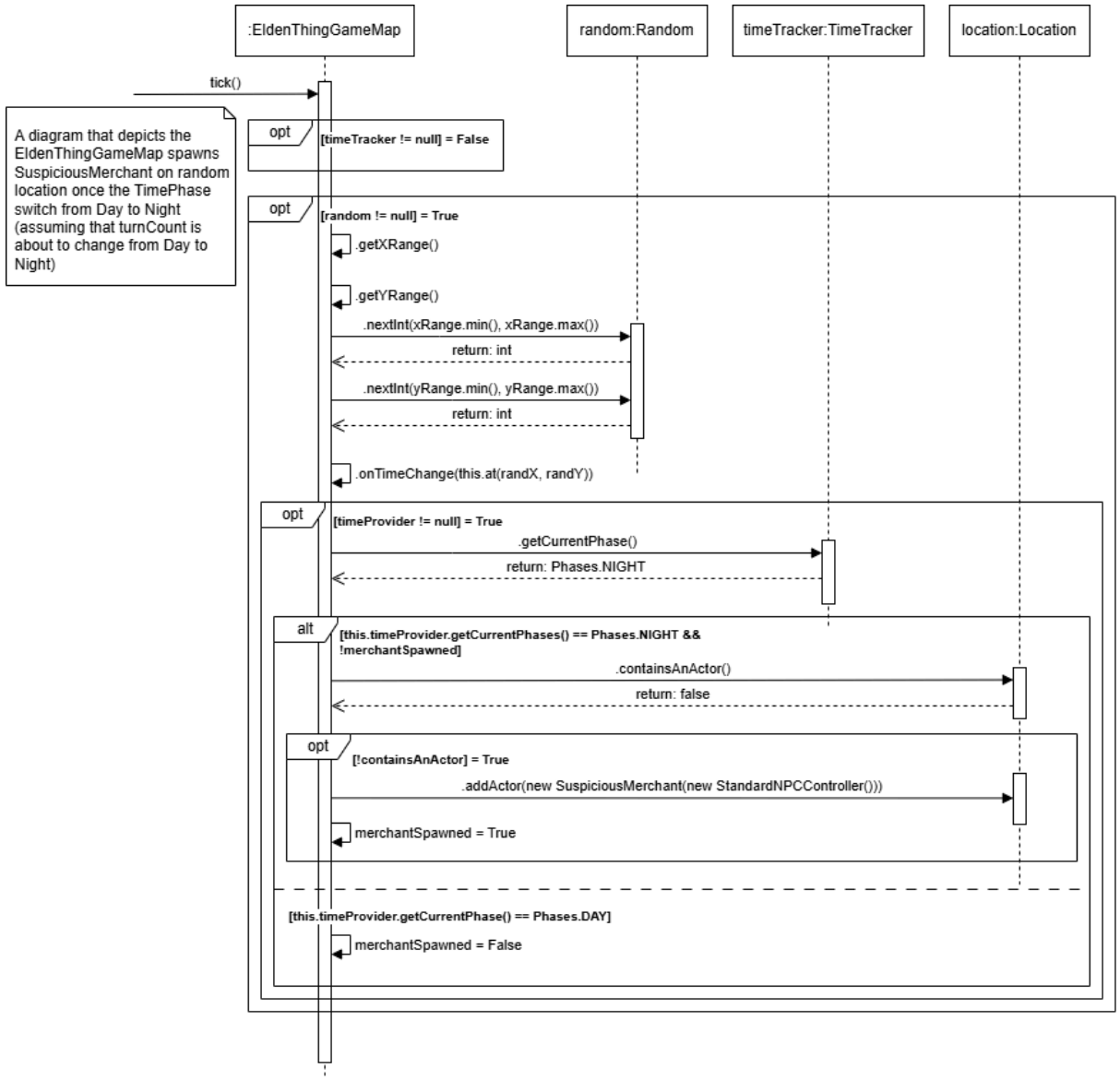
Req1 Sequence Diagram



Req2 Sequence Diagram



Req3 Sequence Diagram



Req4 Sequence Diagram