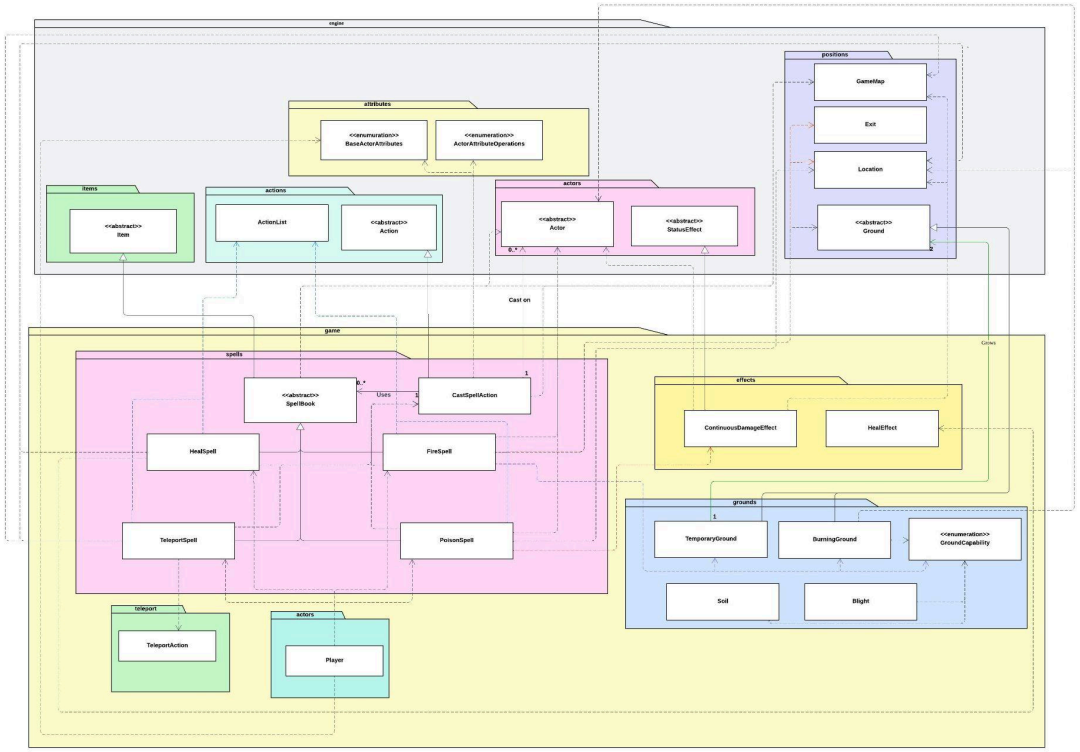
Design Rationale: Spellcasting System (REQ3)



1. Introduction

This document outlines the design for the spellcasting system. It allows the Player to cast spells **(FireSpell, HealSpell, PoisonSpell, TeleportSpell)** from SpellBook items in their inventory, consuming a new Mana attribute. The design supports various targeting paradigms (self-cast, single-target, area-of-effect) and is architected to prioritise extensibility and modularity.

2. Design Goals

- Implement a Mana-based spellcasting system where the Player can cast spells from their inventory.
- Support diverse spell types, including self-targeted, enemy-targeted, and area-of-effect (AoE) spells.
- Decouple the logic of a spell's effect from the action of casting it, promoting high cohesion and low coupling.

3. Design Analysis & Decisions

3.1 Spell Activation and Mana Management

This feature addresses the fundamental mechanism for how a spell is cast and how its Mana cost is handled.

Option A: Centralised Spell Logic in a SpellManager Class

- **Description:** A single **SpellManager** or the **Player** class would contain a large method with a **switch** or **if-else** block. This block would identify the spell being cast and execute the appropriate logic.

Pros	Cons
Conceptually simple for a very small, static set of spells.	Violates SRP, as the manager class assumes the responsibilities of every spell, leading to a "god class" with low cohesion.
	Violates OCP, as this central class must be modified every time a new spell is added.
	Exhibits strong connascence of algorithm. The SpellManager is intimately tied to the internal implementation of every spell, making the system rigid and fragile.

Option B: Abstracted SpellBook with a Generic CastSpellAction (Chosen Design)

- **Description:** An abstract SpellBook class provides an interface and a contract for all spells, defining methods like **getManaCost()** and **activate()**. Each concrete spell extends SpellBook and encapsulates its unique logic within the **activate()** method. A generic **CastSpellAction** are responsible for enforcing the contract's preconditions (checking Mana) before executing the action.

Pros	Cons
CastSpellAction is only responsible for the act of casting, while each **SpellBook** subclass is responsible for its specific behaviour this adhering to SRP.	Increases the number of classes in the project.
New spells can be introduced by simply creating new subclasses of **SpellBook**; no modifications are needed (OCP), **CastSpellAction** or **Player**.	
Transforms strong connascence into weaker forms. The system now relies on connascence of name (**getManaCost**, **activate**) and connascence of Type (**SpellBook**), which are low-strength, static forms of connascence.	

- **Decision & Justification: Option B** was chosen. Its adherence to SRP and OCP creates a clean architecture. By leveraging abstraction (DIP), it strategically refactors the highly problematic connascence of algorithms found in Option A into the weakest forms of static connascence.

3.2. Spell Effect Implementation

Option A: Effect Logic Hardcoded in Concrete Spell Classes

- **Description:** Each SpellBook subclass would directly implement the full logic for its effect. Like, **PoisonSpell.activate()** would contain the code to apply damage and manage a turn counter.

Pros	Cons
Simple for effects that are unique and not intended for reuse.	Violates the DRY principle. If a "Poisoned Arrow" item were introduced later, the poison logic would be duplicated.
	This duplication creates an undesirable connascence of algorithms across different parts of the codebase.

Option B: Delegating to a Reusable Effect System (Chosen Design)

- **Description:** Spells delegate their effects to a dedicated, reusable system of **Effect** and **StatusEffect** classes. When **PoisonSpell.activate()** is called, it instantiates and applies a ContinuousDamageEffect to the target.

Pros	Cons
The logic for "poison" is implemented once in **ContinuousDamageEffect** and can be reused by any scenario, thus adhering to DRY.	Introduces an additional layer of abstraction and a greater number of small, focused classes.
The **PoisonSpell** class's responsibility is to initiate the poison effect, while the ContinuousDamageEffect's responsibility is to be a poison effect (SRP).	
Lowers connascence. The PoisonSpell now only has connascence of name with the **ContinuousDamageEffect** class; it must know its name to instantiate.	

- **Decision & Justification: Option B** was chosen. It avoids the connascence of algorithms for similar effects across different game items by encapsulating the effect logic in one place.

3.3. Environmental Spell Effects and Temporary Ground

Option A: Monolithic Temporary Ground Classes

- **Description:** A **BurningGround** class would be created that extends **Ground**. It would contain its internal timer and a reference to the **Ground** type it should revert to.

Pros	Cons
Requires fewer classes for a single temporary effect.	Violates SRP. The **BurningGround** class has two distinct responsibilities: its burning behaviour and its temporary lifecycle management.
	Violates DRY. If an **IcyGround effect** was needed later, the timer logic would have to be reimplemented, creating a connascence of algorithms.

Option B: Generic TemporaryGround Decorator (Chosen Design)

- **Description:** This design utilises the Decorator Pattern. A generic **TemporaryGround** class wraps a **Ground** object. The decorator's sole responsibility is managing the duration and revision. It delegates the per-turn effect logic by calling the **tick()** method of the **Ground** object it is decorating.

Pros	Cons
TemporaryGround is responsible only for the lifecycle. **BurningGround** is responsible only for the burning effect, which follows (SRP).	The Decorator pattern is a more complex concept than simple inheritance.
Maximises reusability (DRY) and extensibility (OCP).	
Exhibits very low-strength connascence. **TemporaryGround** only has connascence of name with the **tick()** method of the **Ground** interface.	

- **Decision & Justification: Option B** was chosen. The Decorator pattern provides an elegant and highly modular solution. If a new temporary effect, such as

ConsecratedGround that heals actors, is required, it can be implemented as a new **Ground** subclass and instantly made temporary by wrapping it with the existing **TemporaryGround** class, demonstrating perfect adherence to OCP.

4. Conclusion

The spellcasting system has been architected as a modular and extensible feature. The design, which utilises an abstract **SpellBook** class, a generic **CastSpellAction**, and a reusable **Effect** system, successfully decouples the core components of the system. This separation of concerns enhances cohesion and strategically manages dependencies. By transforming potentially strong dependencies (connascence of algorithm) into weaker, more manageable forms (connascence of name), the architecture is fortified against future changes.