SC2002 Object-Oriented Design & Programming

Turn-Based Combat Arena

Group Assignment Report

Module	SC2002
Academic Year	AY25/26 Semester 2
Lab Group	FCSE
Group Members	NEOH TIAN POK (LIANG ZHANBO) NESTOR ZHANG RUIZHE NG QI YING (HUANG QIYING) PATEL MADHAVAN PARESHBHAI
GitHub Repository	https://github.com/liang799/SC2002-Project
Date	April 19, 2026

Table of Contents

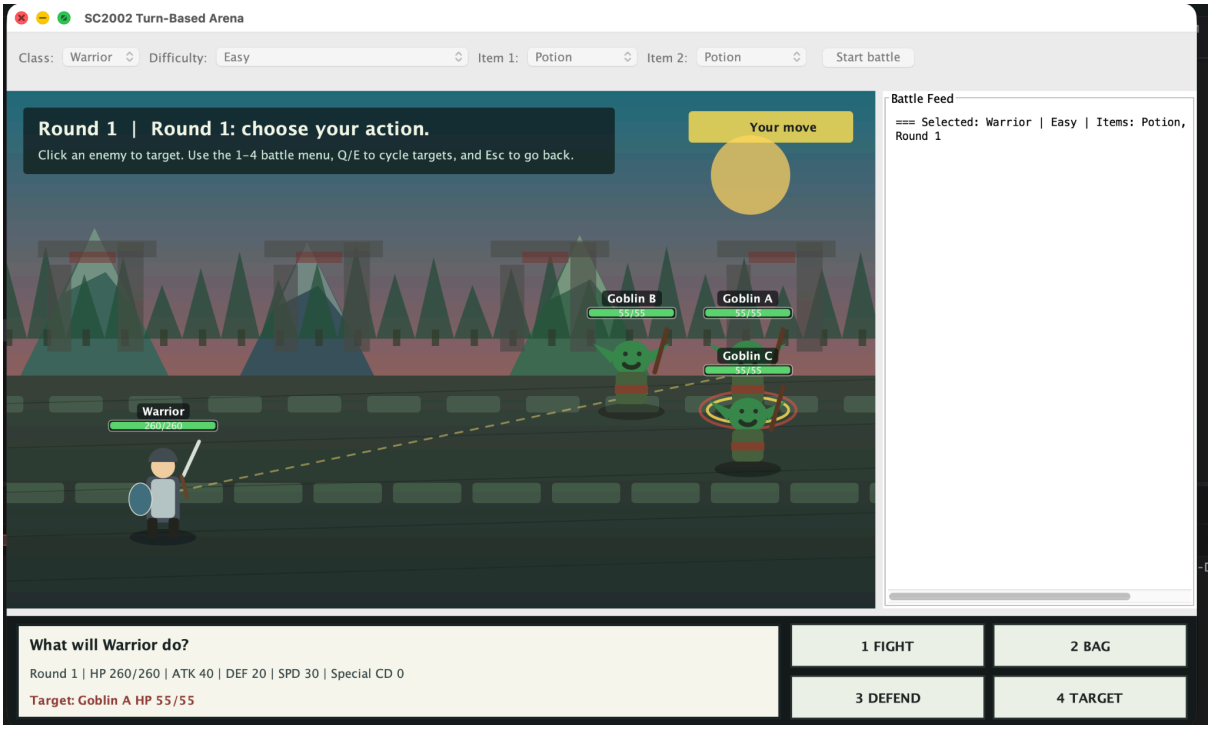
Appendix A

a) Additional Features

Beyond the core assignment requirements, our group implemented several additional features. Each is described below with its design motivation and connection to object-oriented principles.

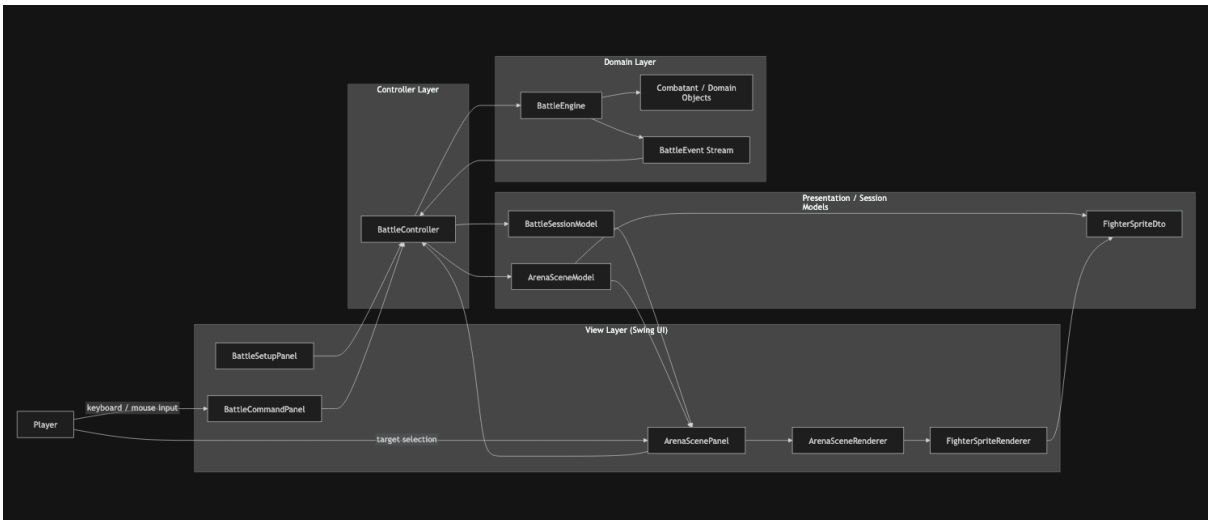
Swing GUI Front End

We implemented a comprehensive Java Swing graphical user interface to enhance interactivity and user experience. Moving beyond basic console-based interactions, the system now features a 2D arena complete with animated sprites, environmental backgrounds, dynamic health bars, and floating combat text. The interface provides a polished command menu, allowing players to navigate the battlefield via keyboard, manage target selection through mouse interactions, and execute strategic actions through an intuitive command hierarchy.



This enhancement was motivated by a desire to improve visual clarity and engagement. While the CLI requires players to parse text logs to understand battle state, the GUI projects the current actor, active targets, and real-time HP metrics directly onto the viewport. To further support readability, we implemented an animation subsystem that paces the delivery of narration and combat events, ensuring the player can follow the flow of battle without being overwhelmed by rapid execution.

MVC-inspired Architecture



We structured the GUI in an MVC-inspired way so that each part of the interface has a clear role. The `BattleController` connects the engine to the display, while `BattleSessionModel` and `ArenaSceneModel` keep track of temporary UI state like turns and sprite changes. Components such as `ArenaScenePanel`, `BattleCommandPanel`, and `BattleSetupPanel` are then responsible for displaying information and receiving user input.

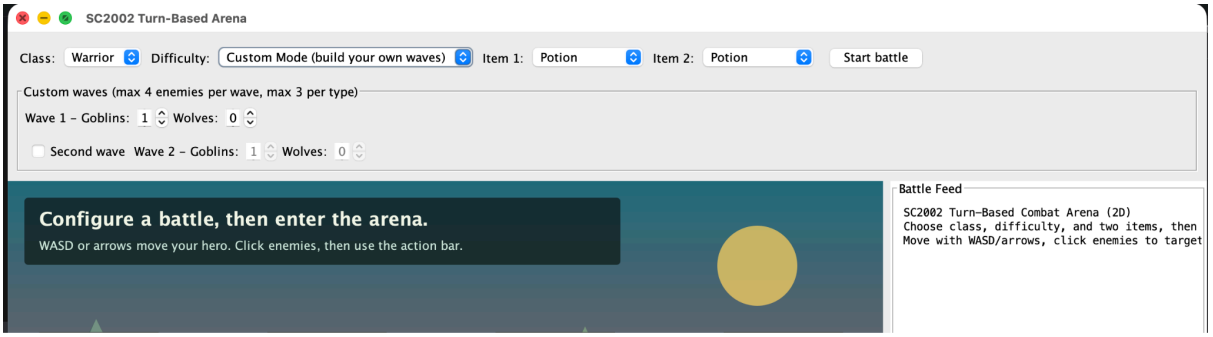
We also tried to follow the Single Responsibility Principle carefully. Instead of putting everything into one class, `ArenaScenePanel` acts mainly as the top-level container, while other classes handle specific jobs such as state management, scene coordination, and character rendering. By moving sprite-related data into `FighterSpriteDto`, we kept rendering details separate from the actual combat logic, so the domain layer does not depend on Swing.

More importantly, the battle engine remains the only place where game rules and outcomes are decided. The GUI simply listens to the engine's event stream and reflects those updates visually. This keeps the interface cleanly separated from the battle logic, so changes to visuals or animations do not affect the engine.

Finally, we refactored the rendering pipeline so it is easier to extend. Different renderers handle different fighter archetypes, such as the Warrior, Wizard, and Goblin. This means new fighter types can be added by introducing new renderer classes, without changing the existing drawing logic.

Custom Game Mode With Configurable Waves

We designed the project to go beyond the preset Easy, Medium, and Hard modes by also allowing players to customise their own battle setup. Instead of being limited to fixed difficulty options, we can configure either a one-wave or two-wave battle and choose the enemy composition more directly. Before the battle begins, the system checks that the setup is valid, so we can ensure the custom configuration makes sense and does not break the game flow.



This custom mode includes:

- one or two waves
- configurable enemy composition per wave
- validation of invalid wave setups
- automatic naming of repeated enemies across waves

This is a major functional addition because it transforms battle setup from a fixed preset system into a configurable battle-construction system, demonstrating the Factory pattern and the value of separating configuration from execution. Hence, validating the robustness of our code.

Richer Status-Effect System

We built the status effect system as a more advanced part of the project, instead of treating effects as simple boolean flags inside `Combatant`. Rather than just marking whether an effect is active or not, we represent each status effect as its own object with its own behaviour and lifecycle. This gives us much better control over how effects are applied, how long they last, and how they influence the battle.

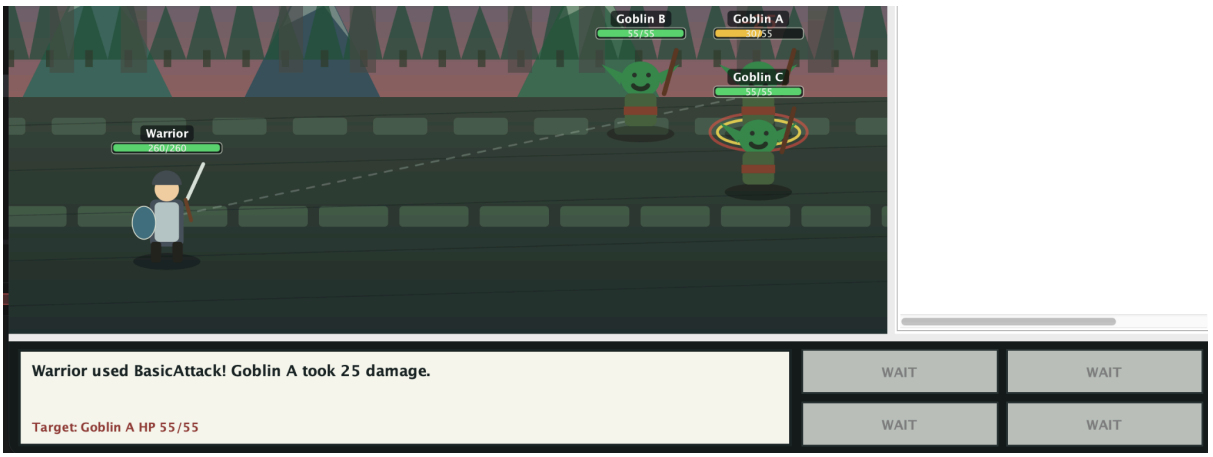
The effects we implemented include stun, defend buff, smoke bomb, arcane power boost, and strength boost. Each of these is defined as its own class that implements the `StatusEffect` interface, and each one is managed through `StatusEffectRegistry`. This helps us keep effect-related logic organised and makes it easier to add new effects in the future without cluttering the combat classes.

- Turn blocking via StatusEffect.getTurnBlockReason and StatusEffectRegistry.getTurnBlockReason
- Stat modification via StatusEffect.modifyStats
- Incoming damage modification via StatusEffect.modifyIncomingDamage (registry calls this and aggregates modifiers)
- Expiry at the correct timing via isExpired
- Effect merging via mergeWith in StatusEffectRegistry
- Structured effect outcomes via StatusEffectOutcome and CombatantStatusOutcome, produced by StatusEffectRegistry and reported through battle events

This is a significant extension because it makes the combat model more expressive and easier to expand by adding new StatusEffect implementations without changing existing classes.

Live Event-Driven Battle Output

The battle engine now emits structured battle events that are rendered live during gameplay. Rather than only printing a final transcript after everything is finished, the system can show round starts, actions, skipped turns, status-effect changes, summaries, and outcome narration as they happen. BattleEventPublisher forwards events to BattleEventListener.onEvent() during BattleEngine.runRounds().[1]



- RoundStartEvent, ActionEvent, SkippedTurnEvent, NarrationEvent, RoundSummaryEvent all emitted in real time
- CLI and GUI both receive the same event stream through different BattleEventListener implementations
- BattleConsoleFormatter converts events to readable text outside the engine
- Same events reused for automated test validation

This improves both gameplay and architecture. From the player's perspective, the battle is easier to follow. From the implementation perspective, it cleanly separates battle logic from output formatting and allows the same events to be reused for debugging and validation.

Modularized Engine Architecture

[1] Used AI for fluency. Original sentence was "we changed the game so it reports what is happening live, while the battle is going on, instead of waiting until the very end to dump everything at once"

The battle engine has been split into smaller responsibilities instead of keeping all orchestration inside one large class. The current implementation separates concerns such as:

- DefaultTurnProcessor: processes one combatant turn
- DefaultWaveManager: handles wave transitions and backup spawning
- DefaultRoundLifecycle: manages round start and end hooks
- DefaultBattleOutcomeReporter: determines and reports battle result
- BattleState: tracks shared mutable battle state
- BattleEventPublisher: forwards events to registered listeners

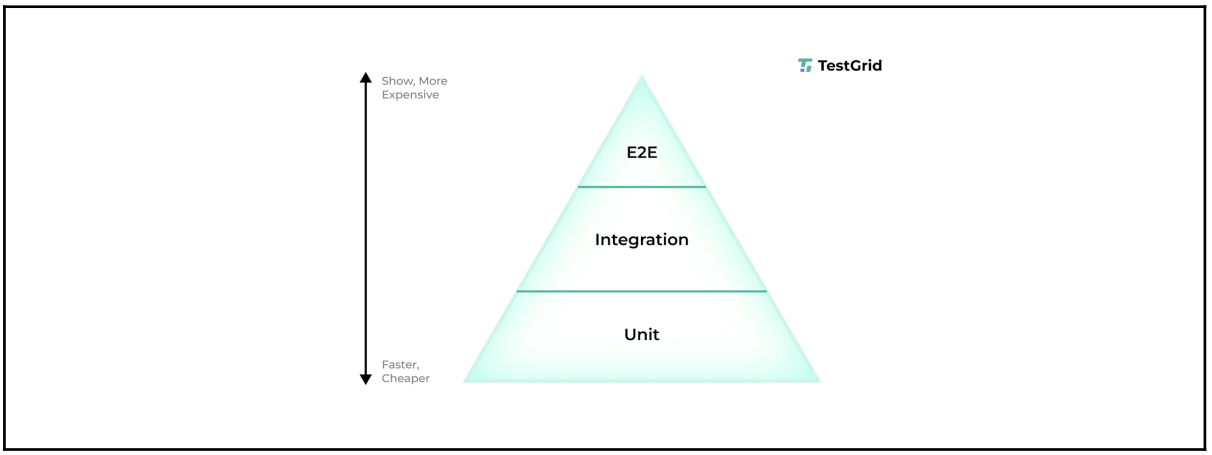
This is an important additional feature because it represents a structural enhancement to the codebase itself. It makes the engine easier to extend, easier to test, and easier to maintain as more mechanics are introduced.

Extensive Automated Testing Infrastructure

The project now includes a much stronger automated testing system than a basic scenario checker. The test suite covers multiple layers of the application, including:

- domain-level behavior
- status effects
- actions
- engine components
- appendix scenario integration
- custom battle flow end-to-end validation
- UI decision-provider behavior

The tests are also annotated according using `@Tag` to better capture the testing pyramid:



We also used reusable test-support infrastructure for building combatants, capturing rounds, asserting battle state, and running scenarios consistently. This is one of the most important additional features because it directly supports correctness, refactoring safety, and confidence in the more advanced mechanics added to the project.

b) Design Considerations

Object Oriented Concepts

Abstraction

We use abstraction in the codebase to keep the overall flow of the system stable, while still allowing specific rules and behaviours to evolve independently. This helps us separate what changes often from what should remain consistent. For example, more volatile concerns like the user interface are kept at the outermost layer, while the core business logic is expressed through domain-level abstractions.

At the centre of the design, we model `Combatant` as an abstract class that captures behaviour shared by all battle participants, such as resolving attacks, handling damage, computing stats, and interacting with status effects. We use `BattleAction` to define a common contract for actions that can be executed, including their name, targeting behaviour, cooldown handling, and event generation. We also use `StatusEffect` to represent how effects behave throughout their lifecycle, from application and stat modification to damage adjustment, round-end processing, and expiry.

To support flexibility in gameplay flow, we abstract turn sequencing through `TurnOrderStrategy`, and we separate decision-making through `PlayerDecisionProvider`, which allows the same game flow to work whether decisions come from a real player or from an automated test harness. On the GUI side, we use `BattleView` to define what the interface needs to present, while `BattleController` coordinates view updates and connects interactive commands to the engine's blocking APIs. This lets us support a graphical interface without leaking Swing-specific concerns into the engine or domain layers.

Encapsulation

We also make sure that internal collections are not exposed as mutable maps. For example, `Inventory` keeps item counts inside a private `EnumMap` and only exposes a small, controlled set of operations such as `add`, `countOf`, `use`, and `snapshot`. This allows us to manage inventory state safely without letting other parts of the system modify it freely.

In a similar way, `HitPoints` encapsulates health-related transitions such as taking damage and healing, so `Combatant` does not need to work directly with raw integers or manually clamp values. We use `StatusEffectRegistry` to manage the active effects and pending outcomes for a single owner, including the logic for merging effects and pruning expired ones. For player abilities, `SpecialSkill` encapsulates both the underlying special-skill `BattleAction` and its cooldown state, which keeps this behaviour together instead of scattering related fields across unrelated classes.

Within the GUI layer, we follow the same principle of encapsulation. Swing components and temporary UI state are kept behind panels such as `BattleSetupPanel`, `BattleCommandPanel`, and `ArenaScenePanel`. This helps us keep rendering and input handling separate from the actual battle rules, so the interface remains easier to change without affecting the core game logic.

Inheritance

We use inheritance only when there is a genuine subtype relationship. In this design, `PlayerCharacter` and `EnemyCombatant` both extend `Combatant` because they represent different kinds of battle participants while still sharing the same core behaviour. Each one then specialises how it takes part in a turn.

We also separate archetype data from concrete instances. `PlayerType` and `EnemyType` define the underlying character or enemy templates, while `CombatantFactory` is responsible for creating the correct `PlayerCharacter` or `EnemyCombatant` objects with the appropriate skills and registries already configured. This helps us keep object creation consistent and avoids spreading setup logic throughout the codebase.

For enemy behaviour, `EnemyCombatant` composes a `BattleAction` field to represent what it does on its turn. This meets the current requirement that enemies always perform `BasicAttack`, while still giving us room to extend the design later if we want different enemies to use different action strategies.

Polymorphism

We design the engine and turn processor to work with actors through shared abstractions rather than concrete implementations. This means the core engine does not need to check class names or branch on specific action types. Instead, different `BattleAction`

implementations can be invoked in a uniform way, which keeps the engine logic cleaner and easier to extend.

We apply the same idea to status effects. Rather than placing all effect-handling logic inside `Combatant` through one large central switch, we process `StatusEffect` implementations through the hooks managed by the registry. This allows each effect to define its own behaviour while keeping the combat flow organised.

For turn sequencing, `TurnOrderStrategy` gives us a replaceable way to determine turn order without having to rewrite the round loop itself. In the same way, `PlayerDecisionProvider` allows the same `BattleEngine` to run across different input sources, whether decisions come from the CLI, from the GUI through the controller bridge, or from scripted inputs used for automated testing. This gives us flexibility without forcing changes into the engine each time we support a new way of driving the game.

No Side Effects

From reading Kent Beck's TDD book, we learned the importance of immutability and side effects. For example, instead of mutating integer hitpoints within combatant, we extracted hitpoints to a value object, just like the Money example from his book.

DDD (Domain Driven Design)

After watching uncle bob, we believe our code should not be domain anemic. At the very core domain, the entities should be expressive and indicative of business rules. This in turn, also makes testing easy to read:

```
@Test  liang799
void receiveDamageAndHeal_WhenHitPointsChange_UpdatesHitPointsValueObject() {
    PlayerCharacter warrior = TestDependencies.warrior();

    warrior.receiveDamage(90);
    HitPoints damagedHitPoints = warrior.getHitPoints();
    warrior.heal( amount: 500);
    HitPoints healedHitPoints = warrior.getHitPoints();

    assertEquals(new HitPoints( current: 170,  max: 260), damagedHitPoints);
    assertEquals(new HitPoints( current: 260,  max: 260), healedHitPoints);
}
```

In this example, `PlayerCharacter` exposes meaningful domain behaviour such as taking damage and healing, while `HitPoints` acts as a value object that preserves HP invariants. This also makes the test easy to read, because it reflects a domain scenario rather than low-level state manipulation.

SOLID Design Principles

Single Responsibility Principle (SRP)

Each class has exactly one reason to change. SRP is applied by splitting responsibilities across small, focused types.

Boundary

Our code at this layer only exists to translate and pass messages. It doesn't know business logic.

Control

Our code at this layer only has a single responsibility, which is to take input from a Boundary, figure out which Entities need to be involved, execute the sequence of operations required for that specific use case, and return the result to the Boundary.

BattleEngine coordinates rounds but delegates turn processing, wave transitions, round summaries, and outcome reporting to dedicated collaborators; BattleState, DefaultTurnProcessor, DefaultWaveManager, DefaultRoundLifecycle, and DefaultBattleOutcomeReporter; keeping each unit easier to reason about than a single monolithic engine class.

Evidence: When we added the Swing GUI, new boundary and controller layer classes were introduced, including TurnBasedArenaGui, BattleController, and view panels such as BattleSetupPanel, BattleCommandPanel, and ArenaScenePanel. The battle rules remained in the engine and domain layers, and the GUI integrates by consuming BattleEvent output and providing decisions through PlayerDecisionProvider.

Entity

Our domain objects only encapsulate business rules. It doesn't handle UI or the execution of business logic.

Open/Closed Principle (OCP)

Extension happens by adding new classes implementing existing interfaces, while keeping the overall battle loop structure stable.

Interface	Existing implementations	How to extend
BattleAction	BasicAttackAction, DefendAction, ShieldBashAction, ArcaneBlastAction, UsePotionAction, UseSmokeBombAction, UseSpecialSkillAction, UsePowerStoneSkillAction	Add new class implementing BattleAction

StatusEffect	StunStatusEffect, DefendStatusEffect, SmokeBombStatusEffect, ArcanePowerStatusEffect, StrengthBoostStatusEffect	Add new class implementing StatusEffect
TurnOrderStrategy	SpeedTurnOrderStrategy	Add new class implementing TurnOrderStrategy
BattleEventListener	CLI listener, GUI listener	Add new class implementing BattleEventListener
PlayerDecisionProvider	CliPlayerDecisionProvider, GuiPlayerDecisionProvider, ScriptedDecisionProvider	Add new class implementing PlayerDecisionProvider

Evidence: ShieldBashAction and ArcaneBlastAction were added without modifying BattleEngine. The GUI was added as a new UI boundary that consumes the same BattleEventListener event stream without changing the engine loop.

Liskov Substitution Principle (LSP)

Subtypes honour the Combatant contract for life state, turn blocking reasons, and participation in the shared turn pipeline so DefaultTurnProcessor can treat all actors uniformly except where player input is required.[2]

- PlayerCharacter instances are substitutable regardless of PlayerType: the engine processes all player characters through the same turn pipeline without checking the underlying archetype
- EnemyCombatant instances are substitutable regardless of EnemyType: DefaultTurnProcessor handles all enemy combatants uniformly through the shared turn pipeline
- Both CLI and GUI use the same BattleEngine and BattleEventListener; only the decision provider and presentation layer differ (CLI uses ConsoleBattleUi and CliPlayerDecisionProvider, GUI uses BattleController and a GUI decision provider)
- ScriptedDecisionProvider and CliPlayerDecisionProvider are both valid PlayerDecisionProvider: BattleEngine cannot distinguish them

Interface Segregation Principle (ISP)

[2] AI was used for fluency. Original sentence was “Players and enemies more or less work the same in the turn system, so the game can handle both using the same flow. Only difference is player need user input.”

All interfaces are lean and focused; no class is forced to implement methods it does not need.

- BattleEventListener exposes one method: onEvent(BattleEvent)
- TurnOrderStrategy exposes one method: determineOrder(combatants)
- PlayerDecisionProvider exposes one method: decide(roundNumber, player, livingEnemies); unrelated UI methods are not forced into this type
- BattleAction stays small and uses defaults where behaviour is common across actions

Dependency Inversion Principle (DIP)

High level modules depend on abstractions, not concrete classes.

High level class	Depends on (abstraction)	NOT on (concrete)
BattleEngine	TurnOrderStrategy	SpeedTurnOrderStrategy
BattleEngine	PlayerDecisionProvider	CliPlayerDecisionProvider
BattleEngine	BattleAction	BasicAttackAction etc
BattleEngine	BattleEventListener	BattleConsoleFormatter
TurnBasedArenaCli	BattleEventListener and PlayerDecisionProvider interfaces	ConsoleBattleUi or TurnBasedArenaGui directly

Formatting is applied outside the engine by composing BattleConsoleFormatter inside the listener lambda in TurnBasedArenaCli or the GUI transcript path in TurnBasedArenaGui; BattleEngine never calls console or Swing APIs directly.

Design Patterns

Strategy Pattern

TurnOrderStrategy with SpeedTurnOrderStrategy is the Strategy pattern for turn ordering. Enemy behaviour is composed using a BattleAction field on EnemyCombatant; a practical way to vary enemy turns without embedding a large conditional per species inside the engine.

Observer Pattern

BattleEventPublisher forwards BattleEvent instances to BattleEventListener.onEvent() during BattleEngine.runRounds(). This decouples narration and UI from engine internals and supports multiple consumers simultaneously. The CLI transcript, the GUI display, and any future observer can all receive the same event stream.

Factory Pattern

BattleSetupFactory builds BattleSetup from configuration records. CombatantFactory and CombatantFactories centralise construction of players and enemies with consistent status registries and skills. This ensures BattleEngine receives ready made objects and never directly constructs domain entities.

Trade offs and Alternatives Considered

Decision	What we chose	Alternative considered	Why we chose ours
BattleEngine structure	Delegate to DefaultTurnProcessor, DefaultWaveManager, DefaultRoundLifecycle, DefaultBattleOutcomeReporter	Single monolithic BattleEngine class	Reduces collision risk when multiple features are edited; each collaborator is easier to reason about

Player/Enemy archetypes	PlayerType and EnemyType enums plus CombatantFactory	Separate Warrior, Wizard, Goblin, Wolf subclasses	Reduces duplicated subclass boilerplate; factory shows clearly how archetypes differ
Event system	BattleEventListener Observer pattern	Direct print statements in BattleEngine	Separates display from logic; easier to add new output targets
Player decisions	PlayerDecisionProvider interface	Hardcoded Scanner input in BattleEngine	Enables scripted testing, GUI input, and future AI without engine changes
Enemy AI	BattleAction field on EnemyCombatant	Switch statement in BattleEngine per enemy type	OCP: new enemy behaviours without touching engine

Coupling and Cohesion

Coupling from engine to UI is kept indirect through events and decisions; BattleEngine never imports or references any UI class. Cohesion is strengthened by keeping reporting types in the report package, action implementations in the actions package, orchestration in the engine package, and domain rules in the domain package including domain.status for status effects.

Hence, none of the code is tightly coupled.

c) Reflection

Difficulties Encountered

Initially we were very overwhelmed. We decided to adopt a bottom up approach: we just jump straight into coding.

We only roughly knew that we had to split the project into layers because of ECB (entity, control, boundary).

We were somewhat doing pair programming. The 3 of us were surrounding Nestor, watching him code out and giving feedback accordingly and taking turns coding on his laptop

Then as the project size grew, we realise that it was unsustainable to continue doing so, we are facing several difficulties.

As the codebase became larger, it was harder to make changes without accidentally breaking something else. Because we had begun coding before properly planning the structure, some parts of the system became tightly coupled, which made the code harder to understand, test, and maintain. It also became difficult for team members to work independently, since much of the project depended on a shared understanding formed during live coding sessions rather than clear documentation or well-defined responsibilities

How Difficulties Were Overcome

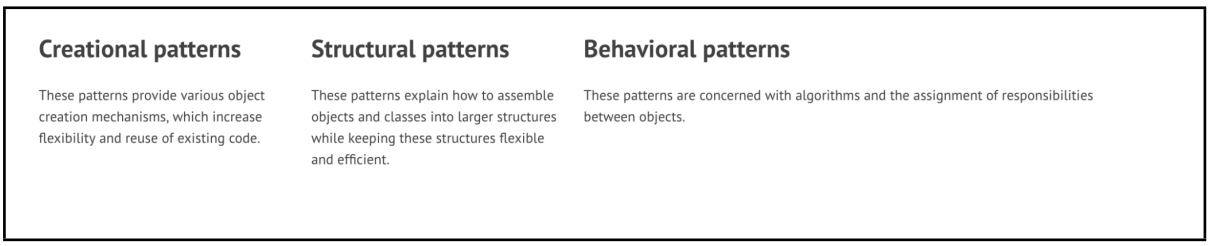
We revisited the lecture notes and paid closer attention to the SOLID principles. As a team, we analysed the codebase file by file to identify classes that violated these principles, and we refactored them accordingly.

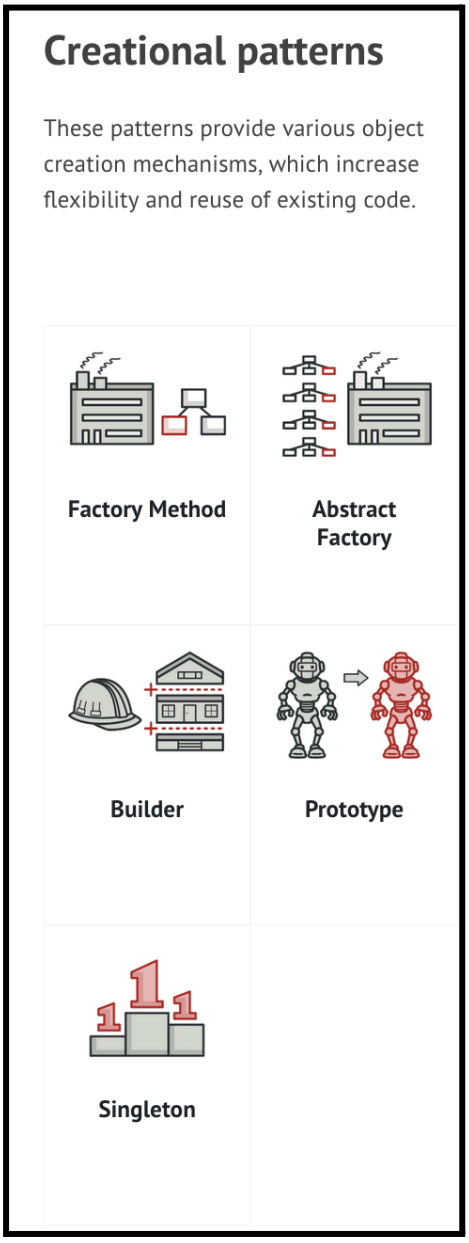
During the school holidays, Neoh also deepened his understanding of software design by watching Uncle Bob's videos, which introduced ideas from Extreme Programming. This led him to briefly explore Kent Beck's concept of Test-Driven Development (TDD). Based on this, we decided to adopt the testing pyramid so that we could make changes with greater confidence and reduce the risk of breaking existing functionality.

One idea that particularly influenced our design was Kent Beck's example of the `Money` object, which encapsulates core domain logic in an immutable value object. Inspired by this, we refactored the primitive integer representation of HP into a dedicated value object. This allowed the domain rules around HP to be expressed more clearly and safely. We also wrote tests to verify that HP behaved according to the intended domain logic.

We then took this a step further by exploring design patterns after reading more about clean code practices: https://refactoring.guru/

It became much more manageable once we identified the category of the design problem and matched it to one of these 3 patterns:





After identifying pattern (e.g. creational)

We identified the appropriate pattern by reviewing the creational patterns on Refactoring Guru one by one and considering whether each suited our problem.

For example, during testing, we found that the creation of `Combatant` objects was becoming increasingly difficult to read. The constructor required many parameters, which made the test setup tedious and cluttered.

Since the issue was related to object creation, we focused specifically on creational patterns. By going through them individually, we found that the Builder pattern was the most suitable choice. It allowed us to construct test objects in a much clearer and more readable way. For example:

```
Combatant attacker =
TestCombatantBuilder.aCombatant()
    .named("Warrior")
    .withAttack(40)
    .build();
```

We then built a fake builder on top of this, which initialised sensible default values for testing.

This made our unit tests much easier to write and understand. In particular, for attack-related tests, we could focus directly on the attack value under test without being distracted by unrelated attributes such as defence or speed.

And because we adopt testing, the test also serves as documentation. We no longer have to huddle up around one laptop to do all the work:

```
@Test  liang799
void attack_WhenCombatantAttacksTarget_ResolvesCombatInsideCombatant() {
    Combatant attacker = TestCombatantBuilder.aCombatant()
        .named("Warrior")
        .withAttack(40)
        .build();
    Combatant target = TestCombatantBuilder.aCombatant()
        .named("Goblin")
        .withCurrentHp(70)
        .withDefense(15)
        .build();

    AttackResolution attackResolution = attacker.attack(target);

    assertEquals( expected: 25, attackResolution.damage());
    assertEquals( expected: 40, attackResolution.attackUsed());
    assertEquals( expected: 15, attackResolution.targetDefense());
    assertEquals( expected: 70, attackResolution.hpBefore());
    assertEquals( expected: 45, attackResolution.hpAfter());
    assertEquals( expected: 45, target.getCurrentHp());
}
```

The code has become significantly more intuitive, allowing members of the group to understand more clearly how the `Combatant` class functions. This is especially important because `Combatant` is a core domain object.

Domain objects have a far-reaching impact on the system. From our experience, changes to the domain model often propagate throughout the codebase, whereas changes to the user interface usually require only limited modifications. As such, a strong understanding of the domain provides understanding of a substantial portion of the system. The remaining parts of the codebase largely consist of supporting layers, such as domain integration and end-to-end interactions where users engage with the system.

To strengthen reliability, we also introduced additional tests, including integration and end-to-end tests, to ensure that the system behaves correctly in practice. This reduced the need to repeatedly run the application manually and simulate user actions in the RPG game by hand.

As a result, the adoption of the testing pyramid greatly improved our productivity. We now have a much higher level of confidence in the codebase, as each commit only needs to pass the GitHub CI pipeline before being accepted.

Knowledge Learned

- The importance of the Testing Pyramid
- The importance of SOLID principles in making the code easier to change
- How value objects can model domain concepts more clearly
- How object-oriented design patterns help solve recurring design problems
- How unit tests can also serve as documentation

Further Improvement Suggestions

In hindsight, we believe the current solution may have become more complex than necessary. In our effort to apply SOLID principles, design patterns, and layered design, we introduced many abstractions that improved flexibility but increased complexity and overengineering. For a project of this size, a simpler design might have been easier to maintain while still meeting the requirements.

Learning Points and Insights

Through this project, we learned that effective software engineering is not just about producing a working system, but about designing one that can continue to evolve as requirements and code size grow. In the early stages, it was easy to focus mainly on implementation, but as the project expanded, we came to appreciate the importance of maintainability, modularity, and clear separation of responsibilities. Principles such as SOLID, together with the use of value objects, testing, and design patterns, helped us build a system that was easier to reason about and improve.

We also learned that testing plays a much broader role than simple bug detection. Unit, integration, and end-to-end tests gave us greater confidence during refactoring, reduced the risk of unintended regressions, and helped us move faster as a team. In many cases, the tests also acted as documentation by making the intended behaviour of the system more explicit and easier for team members to understand.

At the same time, this project taught us that good design requires balance. While abstraction and design patterns can improve flexibility and extensibility, applying too many of them can introduce unnecessary complexity.[3] As a result, one of our most important insights was that clean architecture should be guided not only by theory, but also by the actual scale and needs of the project. A well-designed system is therefore not simply one with many abstractions, but one that is both structured and appropriately simple.

[3] AI was used for fluency. The original sentence was less formal: “our code got too complicated because of overengineering abstractions”

Appendix B

Declaration of Original Work for SC2002/CE2002/CZ2002 Assignment

We hereby declare that the attached group assignment has been researched, undertaken, completed, and submitted as a collective effort by the group members listed below.

We have honored the principles of academic integrity and have upheld Student Code of Academic Conduct in the completion of this work.

We understand that if plagiarism is found in the assignment, then lower marks or no marks will be awarded for the assessed work. In addition, disciplinary actions may be taken.

Name	**Course(SC2002/CE 2002 CZ2002)**	**Lab Group**	**Date**
NEOH TIAN POK (LIANG ZHANBO)	SC2002	FCSE	16 April
NESTOR ZHANG RUIZHE	SC2002	FCSE	16 April
NG QI YING (HUANG QIYING)	SC2002	FCSE	16 April
PATEL MADHAVAN PARESHBHAI	SC2002	FCSE	16 April

Important notes:

1. Name must EXACTLY MATCH the one printed on your Matriculation Card.
2. Student Code of Academic Conduct includes the latest guidelines on usage of Generative AI and any other guidelines as released by NTU.

Appendix C

NEOH TIAN POK (LIANG ZHANBO)

Declaration on Use of GAI (Generative Artificial Intelligence) Assistance in relation to Assignment/Project (to be submitted individually even for group projects)

I, NEOH TIAN POK (LIANG ZHANBO), (student name),
neoh0045@e.ntu.edu.sg (NTU email) honestly and sincerely make the following declaration in relation to the following course submission:

1. Name of course: OBJECT ORIENTED DES & PROG
2. Course Code: SC2002
3. Instructor: Li Fang
4. Title of Assignment/Project Submission: SC2002 Turn-based Combat Assignment

In relation to the foregoing I hereby declare that, fully and properly in accordance with the Assignment/Project Instructions I have (check where appropriate):
i. Used GAI as permitted to assist in generating key ideas only. ☐
ii. Used GAI as permitted to assist in generating a first text only. ☐
And/or
iii. Used GAI to refine syntax and grammar for correct language submission only.
☐
Or
iv. As it is not permitted: Not used GAI assistance in any way in the development or generation of this assignment or project.
☐
I also declare that I have :
a. Fully and honestly submitted the digital paper trail required under the assignment/project instructions; and that
SC2002 Group Assignment AY25/26 Semester 2
b. Wherever GAI assistance has been employed in the submission in word or paraphrase or inclusion of a significant idea or fact suggested by the GAI assistant, I have acknowledged this by a footnote; and that,
c. Apart from the foregoing notices, the submission is wholly my own work.

Tian Pok
19 April

_____
Student Name & Signature Date

(Optional) Sample Reflection Questions:
In 150 words or less, briefly state up to 5 key ideas that you formulated that the GAI tools alone could not have generated.

NESTOR ZHANG RUIZHE

Declaration on Use of GAI (Generative Artificial Intelligence) Assistance in relation to Assignment/Project (to be submitted individually even for group projects)

I, NESTOR ZHANG RUIZHE, (student name),
nest0001@e.ntu.edu.sg (NTU email) honestly and sincerely make the following declaration in relation to the following course submission:

1. Name of course: OBJECT ORIENTED DES & PROG
2. Course Code: SC2002
3. Instructor: Li Fang
4. Title of Assignment/Project Submission: SC2002 Turn-based Combat Assignment

In relation to the foregoing I hereby declare that, fully and properly in accordance with the Assignment/Project Instructions I have (check where appropriate):
i. Used GAI as permitted to assist in generating key ideas only. ☐
ii. Used GAI as permitted to assist in generating a first text only. ☐
And/or
iii. Used GAI to refine syntax and grammar for correct language submission only.
☐
Or
iv. As it is not permitted: Not used GAI assistance in any way in the development or generation of this assignment or project.
☐
I also declare that I have :
a. Fully and honestly submitted the digital paper trail required under the assignment/project instructions; and that
SC2002 Group Assignment AY25/26 Semester 2
b. Wherever GAI assistance has been employed in the submission in word or paraphrase or inclusion of a significant idea or fact suggested by the GAI assistant, I have acknowledged this by a footnote; and that,
c. Apart from the foregoing notices, the submission is wholly my own work.

Nestor
19 April

_____
Student Name & Signature Date

(Optional) Sample Reflection Questions:
In 150 words or less, briefly state up to 5 key ideas that you formulated that the GAI tools alone could not have generated.

NG QI YING (HUANG QIYING)

Declaration on Use of GAI (Generative Artificial Intelligence) Assistance in relation to Assignment/Project (to be submitted individually even for group projects)

I, NG QI YING, (student name),
c250199@e.ntu.edu.sg (NTU email) honestly and sincerely make the following declaration in relation to the following course submission:

1. Name of course: OBJECT ORIENTED DES & PROG
2. Course Code: SC2002
3. Instructor: Li Fang
4. Title of Assignment/Project Submission: SC2002 Turn-based Combat Assignment

In relation to the foregoing I hereby declare that, fully and properly in accordance with the Assignment/Project Instructions I have (check where appropriate):
i. Used GAI as permitted to assist in generating key ideas only. ☐
ii. Used GAI as permitted to assist in generating a first text only. ☐
And/or
iii. Used GAI to refine syntax and grammar for correct language submission only.
☐
Or
iv. As it is not permitted: Not used GAI assistance in any way in the development or generation of this assignment or project.
☐
I also declare that I have :
a. Fully and honestly submitted the digital paper trail required under the assignment/project instructions; and that
SC2002 Group Assignment AY25/26 Semester 2
b. Wherever GAI assistance has been employed in the submission in word or paraphrase or inclusion of a significant idea or fact suggested by the GAI assistant, I have acknowledged this by a footnote; and that,
c. Apart from the foregoing notices, the submission is wholly my own work.

Larry
19 April

_____
Student Name & Signature Date

(Optional) Sample Reflection Questions:
In 150 words or less, briefly state up to 5 key ideas that you formulated that the GAI tools alone could not have generated.

PATEL MADHAVAN PARESHBHAI

Declaration on Use of GAI (Generative Artificial Intelligence) Assistance in relation to Assignment/Project (to be submitted individually even for group projects)

I, PATEL MADHAVAN PARESHBHAI, (student name),
madhavan004@e.ntu.edu.sg (NTU email) honestly and sincerely make the following declaration in relation to the following course submission:

1. Name of course: OBJECT ORIENTED DES & PROG
2. Course Code: SC2002
3. Instructor: Li Fang
4. Title of Assignment/Project Submission: SC2002 Turn-based Combat Assignment

In relation to the foregoing I hereby declare that, fully and properly in accordance with the Assignment/Project Instructions I have (check where appropriate):
i. Used GAI as permitted to assist in generating key ideas only. ☐
ii. Used GAI as permitted to assist in generating a first text only. ☐
And/or
iii. Used GAI to refine syntax and grammar for correct language submission only.
☐
Or
iv. As it is not permitted: Not used GAI assistance in any way in the development or generation of this assignment or project.
☐
I also declare that I have :
a. Fully and honestly submitted the digital paper trail required under the assignment/project instructions; and that
SC2002 Group Assignment AY25/26 Semester 2
b. Wherever GAI assistance has been employed in the submission in word or paraphrase or inclusion of a significant idea or fact suggested by the GAI assistant, I have acknowledged this by a footnote; and that,
c. Apart from the foregoing notices, the submission is wholly my own work.

Madhavan
19 April

_____
Student Name & Signature Date

(Optional) Sample Reflection Questions:
In 150 words or less, briefly state up to 5 key ideas that you formulated that the GAI tools alone could not have generated.