This system is a turn-based card battle game implemented in Java, using object-oriented programming principles. The system allows a player to use various types of cards, such as attacks, creatures, and status effects, to defeat an enemy creature. The system is implemented to allow the user to continuously battle different creatures, which deal greater damage and have more health in accordance to the user's number of wins. This is achieved as a result of different classes utilizing each other's information, as well as enumerated types. The system supports both a command-line interface for testing and a graphical user interface using JavaFX.

Object-Oriented Design (OOD):
This game follows object-oriented design principles such as polymorphism, enumeration encapsulation, inheritance, and separation of concerns. The codebase is modular, with classes representing gameplay concepts like cards, creatures, the player, and the battle system. Inheritance is used to define reusable behavior, while composition connects objects like the player's hand, deck, and summoned creatures. The design centers around modular, extensible classes that represent core gameplay entities.

Card
Consumable
Effect
Image
Type ENUM
AttackCard
CreatureCard
CreatureCard

AttackCard
Extends Card
Attack Power
Card

CreatureCard
Extends Card
Max Health
Health
Card
Creature

StatusCard
Extends Card
Effect
Card

Creature
Name Damage Health Health Status
CreatureCard
Card

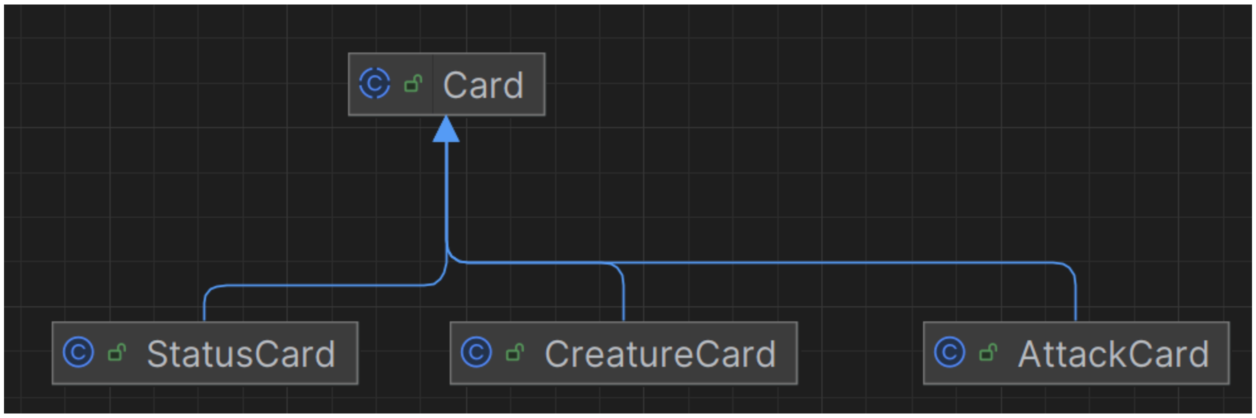
Player
Health
Current Hand
Player Deck
Health Status
DiscardPile
PlayerHand
PlayerDeck
Health Status

Battle Manager
Player
Enemy Creature
Victory Status
Summoned Creatures
Current Effects
Player
Creature

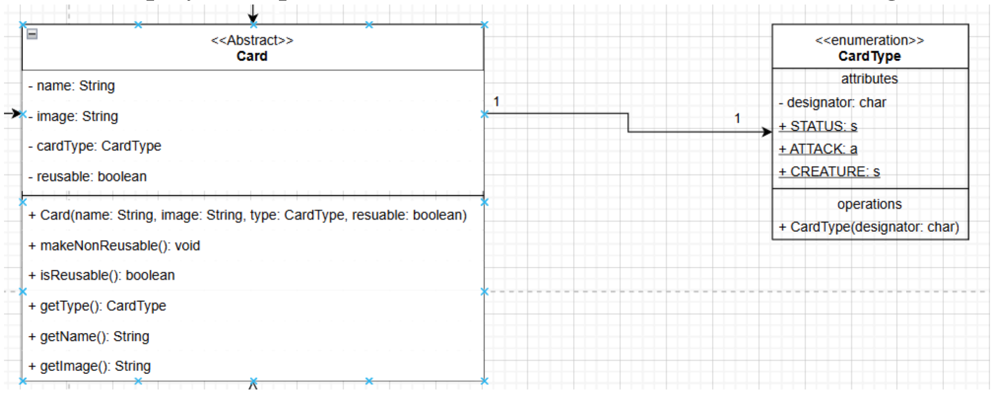
Effects
Type ENUM
Animation
Card
Card Pile

The Class-Responsibility-Collaboration (CRC) cards detail what information objects/classes are supposed to have, and how they interact with each other. The Player class holds information that pertains to a card game and turn-based combat such as a deck of cards, a hand, and health. In BattleManager, it interacts with the Player and Creature classes most of all. This is because the Player and enemy Creature need a way to interact with each other. The BattleManager class is a way to handle the card usage, combat logic, and status effects that happen between the Player and Creature. Thus, while Player and Creature exist separately, both are required to fulfil

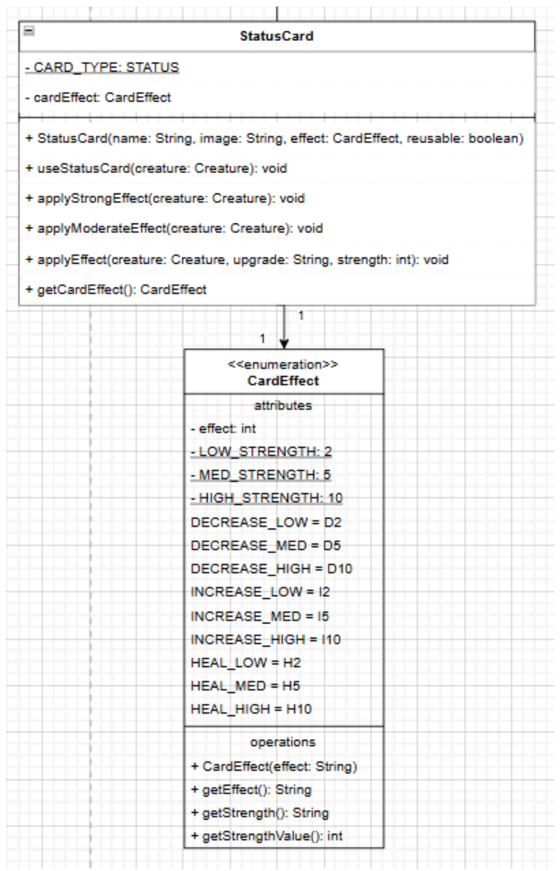
BattleManager's functions. More interactions can be observed when looking at Card, AttackCard, CreatureCard, and StatusCard.



As seen in this section of the UML class diagram, Card is an abstract superclass representing all cards in the game. It has subclasses like AttackCard, CreatureCard, and StatusCard, each handling specific card behavior. Card is abstract, because it is never, and does not need to be, an instantiated object. Card only handles all common behaviors between each card type, which includes a constructor and multiple getter methods, specifically for name, image name, CardType, and reusability. Furthermore, Card and BattleManager interact with each other in order to determine what effects are played for each card. Each card type triggers unique effects when played. Attack cards deal direct damage to the enemy creature, while Creature cards spawn new, allied creatures. Status cards can modify stats such as damage or health. The game ends when the enemy creature's health reaches zero, the player's health reaches zero, or the player runs out of playable options. Effects are determined thanks to the usage of enumerated types.

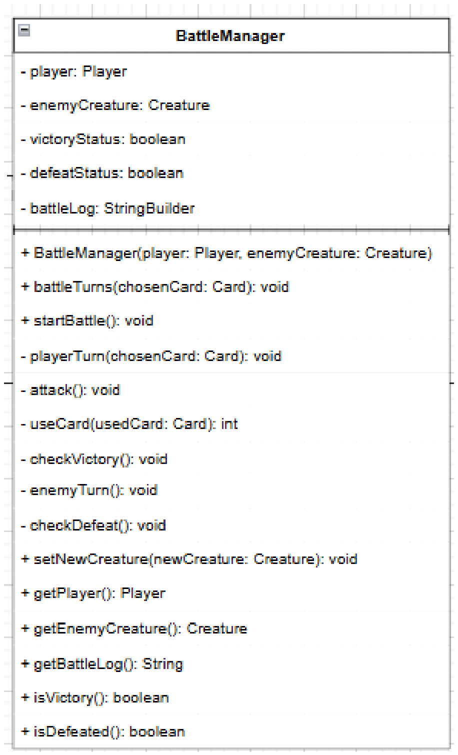


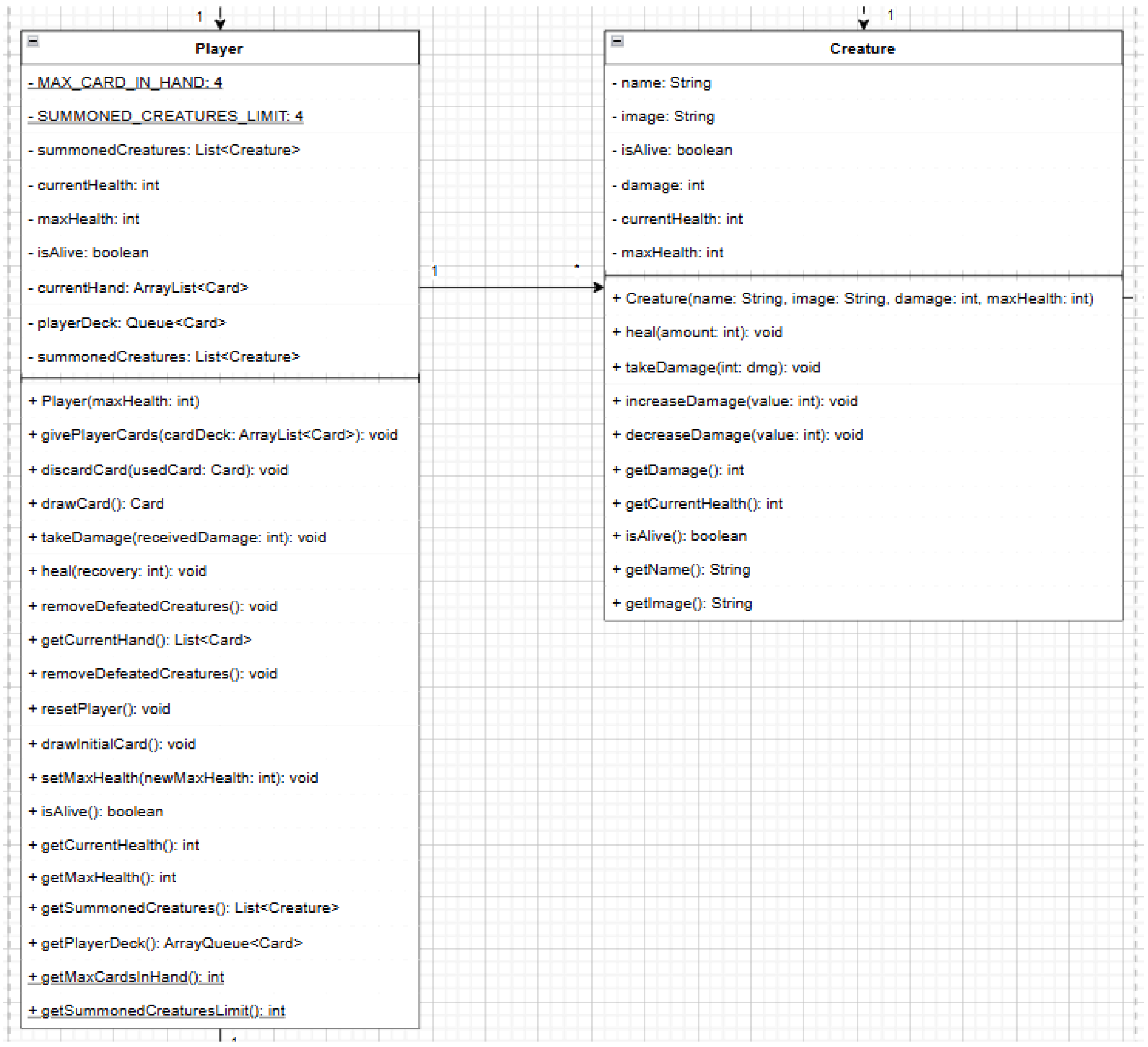
CardType is an enumerated type given to cards, which is used to determine how a card should be used. Each card has its own use() method (useAttackCard() for example), and which one to use is determined in BattleManager using the selected card's CardType.


StatusCard
- CARD_TYPE: STATUS
- cardEffect: CardEffect
+ StatusCard(name: String, image: String, effect: CardEffect, reusable: boolean)
+ useStatusCard(creature: Creature): void
+ applyStrongEffect(creature: Creature): void
+ applyModerateEffect(creature: Creature): void
+ applyEffect(creature: Creature, upgrade: String, strength: int): void
+ getCardEffect(): CardEffect
1
1
<<enumeration>>
CardEffect
attributes
- effect: int
- LOW_STRENGTH: 2
- MED_STRENGTH: 5
- HIGH_STRENGTH: 10
DECREASE_LOW = D2
DECREASE_MED = D5
DECREASE_HIGH = D10
INCREASE_LOW = I2
INCREASE_MED = I5
INCREASE_HIGH = I10
HEAL_LOW = H2
HEAL_MED = H5
HEAL_HIGH = H10
operations
+ CardEffect(effect: String)
+ getEffect(): String
+ getStrength(): String
+ getStrengthValue(): int

Furthermore, CardEffect is an enumerated type given to StatusCards, which represent the type of effect that card has and its strength. After determining that the selected card is a StatusCard, BattleManager then looks at the Card's CardEffect to determine who should get the effect.

In a similar vein with class interactions, the system utilizes encapsulation in a way to ensure certain methods are only accessible within their own class. This way, it makes sure no information is being necessarily tampered with. As seen to the right, BattleManager has multiple private methods, because they are only used in the scope of BattleManager. The methods playerTurn() and enemyTurn() are private, because outside classes do not need individual access to these methods. In contrast, all of Player and Creature's methods are public, because they all have purposes for being called in outside classes. For example, both takeDamage() methods are public because BattleManager needs them to reduce the health of Player and Creature objects during attack() and enemyTurn().


BattleManager
- player: Player
- enemyCreature: Creature
- victoryStatus: boolean
- defeatStatus: boolean
- battleLog: StringBuilder
+ BattleManager(player: Player, enemyCreature: Creature)
+ battleTurns(chosenCard: Card): void
+ startBattle(): void
- playerTurn(chosenCard: Card): void
- attack(): void
- useCard(usedCard: Card): int
- checkVictory(): void
- enemyTurn(): void
- checkDefeat(): void
+ setNewCreature(newCreature: Creature): void
+ getPlayer(): Player
+ getEnemyCreature(): Creature
+ getBattleLog(): String
+ isVictory(): boolean
+ isDefeated(): boolean


1
1
Player
- MAX_CARD_IN_HAND: 4
- SUMMONED_CREATURES_LIMIT: 4
- summonedCreatures: List<Creature>
- currentHealth: int
- maxHealth: int
- isAlive: boolean
- currentHand: ArrayList<Card>
- playerDeck: Queue<Card>
- summonedCreatures: List<Creature>
+ Player(maxHealth: int)
+ givePlayerCards(cardDeck: ArrayList<Card>): void
+ discardCard(usedCard: Card): void
+ drawCard(): Card
+ takeDamage(receivedDamage: int): void
+ heal(recovery: int): void
+ removeDefeatedCreatures(): void
+ getCurrentHand(): List<Card>
+ removeDefeatedCreatures(): void
+ resetPlayer(): void
+ drawInitialCard(): void
+ setMaxHealth(newMaxHealth: int): void
+ isAlive(): boolean
+ getCurrentHealth(): int
+ getMaxHealth(): int
+ getSummonedCreatures(): List<Creature>
+ getPlayerDeck(): ArrayQueue<Card>
+ getMaxCardsInHand(): int
+ getSummonedCreaturesLimit(): int
1
*
Creature
- name: String
- image: String
- isAlive: boolean
- damage: int
- currentHealth: int
- maxHealth: int
+ Creature(name: String, image: String, damage: int, maxHealth: int)
+ heal(amount: int): void
+ takeDamage(int: dmg): void
+ increaseDamage(value: int): void
+ decreaseDamage(value: int): void
+ getDamage(): int
+ getCurrentHealth(): int
+ isAlive(): boolean
+ getName(): String
+ getImage(): String

The architecture of the system separates gameplay logic from UI, allowing battle logic to be tested in isolation via the CLI while maintaining an engaging interface through JavaFX. The code is divided into packages based on the functions of each class. For example, card classes like Card and its children and the previously mentioned enums CardEffect and CardType are grouped together in a package called “cards”. The classes in this package are solely focused on the implementation of card objects. Furthermore, the classes in the “application” package generate the game’s UI, but the game itself is run by classes in the “game” method. Classes implementing creatures and the player are also each held in their own package, which also contains the BattleManager. All class dependencies are clear, favoring encapsulation and single responsibility, with BattleManager coordinating all gameplay flow and interaction.

User Stories

We used three user stories to guide our development process. Each of these stories prioritized different interests and required different code to meet those interests.

First, we recognized that developers like us would want to test the game logic without using the GUI. As a result, we created the class RunGame. Like the name suggests, it is a class to run the game, but it does so through text input instead of using the JavaFX UI, giving developers more flexibility in testing the game.

Next, we recognized that some players like challenges. There are a few ways for us, the game developers, to make the game more challenging. First, using the same deck of cards repeatedly can quickly get boring. To alleviate this, we give the player a new card after every victory. This new card is inserted into the deck randomly, changing the order of the cards every game. Additionally, adding more card effects makes for a more complex game. As a result, when we generate cards in CardGameView to create the player's initial deck or add new cards after wins, we make sure to generate cards of every available type. Finally, players may want increasing difficulty after every game to prevent it from becoming stale. As a result, the chooseRandomCreature() method in CardGameView, which generates the enemy creature, increases the damage and health of enemies depending on how many times the player has won in the current round.

Finally, graphics are very important to some players. First, players would want to see their health and that of the enemy creature. This is handled in different ways depending on if the game is being run through the GUI or the CLI. In the GUI, we use Labels to display the health, while in the CLI we use console messages. Players also want to see cards arrayed like a physical hand and click on one to play it. This is implemented in the GUI, where each card in the hand has its own unique image, and the cards are all clickable buttons that trigger the battleTurns() and updateBattleScreen() methods. Players also want to be able to track the progress of the game through turn-by-turn messages, which we implemented using BattleManager's battleLog method. This prints console messages in the CLI or displays the log in a box in the GUI and allows the player to track the damage output from themselves and the enemy creature, which is critical information that can affect their strategy. Finally, we recognized that players might want to see unique animations for each card. We originally had plans to set up unique animations for each Card object when they were played. Unfortunately, due to time constraints, we were unable to implement this.