Colosseum Duel

Software Design Document

Adrian Quintero

December 12, 2022

CS 225, Fall 2022

Embry-Riddle Aeronautical University

Daytona Beach campus

1 Aerospace Boulevard

Daytona Beach, FL 32114

Introduction:

This Software Design Document (SDD) describes the design and process of an AI-controlled 1v1 dueling game. For the sake of this SDD, the game will be referred to as “Colosseum Duel” (CD). In CD, two AI-controlled combatants take turns attacking each other, with the goal of reducing their enemy’s health points (HP) to zero. This CD project allows a human player to create and customize their own combatant, which is then pit against several automatically generated enemy combatants in successive 1v1s. Note: Although a player combatant, it is still controlled by the computer during combat.

This CD project was created as a showcase project for Embry-Riddle Aeronautical University’s (ERAU) Computer Science II class (CS 225). It showcases everything that is expected of a student to know by the end of the term.

This SSD details the CD project in two parts. The first is the Problem Description, which explains the workings of a CD-based game and rules to follow. The second is the Problem Solution, taking all the information shared in the previous part and describing its implementation in software.

Problem Description:

This project creates a CD game where two AI-controlled combatants duel against each other. The game is entirely text-based, with information being transmitted to the player as the game progresses. The player is given the ability to create a combatant of their own.

Every combatant has a Base Stat Total (BST). This is used to determine multiple calculations during a duel. Additional stats are also used, although they are not selected by the player or automatically generated by the AI (Table 1).

Table 1: Stats Directly Attached to Combatant

Stat Name	Included in BST?	Value Range (disregarding weapons and armor)	Usage
Health points (HP)	Yes	30 to 50	Determine how much damage a combatant can take.
Attack (ATK)	Yes	10 to 40	Determines the minimum damage that can be output.
Speed (SPD)	Yes	15 to 40	Determines move order and whether a combatant can make a follow-up attack.

Defense (DEF)	Yes, additionally found on armor	10 to 35	Determines how much damage is negated.

The customizable stats found on combatants are only one group of stats found throughout CD. Table 2 lists the four other stats found on other sources, such as weapons and armor which are further outlined below.

Table 2: Stats Found on Outside Sources

Stat name (abbreviation)	**Found on:**	**Usage**
Might (MT)	All weapons	
AVOID	Certain weapons	Used in attack accuracy formula.
HIT	All weapons	Used in attack accuracy formula.
CRIT	Certain weapons	Used as a percentage to determine chance of critical damage.
Weight (WGT)	Certain armor	Used in attack accuracy formula.

Table 3: Skill Effects

Skill effects	**Description**
[Poison]	Inflicts -3HP to foe after combat.
[Heal]	Heals user for +3HP after combat.

Multiple weapon variants are available to select, each of which have qualities differentiating them, as shown in Table 4. Combatants can select one weapon for use throughout the rest of the game. Certain weapons have associated skill effects previously outlined in Table 3.

Table 4: Weapon Types

Weapon Type	**Basic stat values**	**Special behaviors**
Sword	14MT, 90 HIT	None
Lance	12MT, 80 HIT	Applies [Heal] to unit after combat.
Axe	16MT, 50 HIT	Grants +20 CRIT.
Bow	8MT, 70 HIT	Grants +10 AVOID and inflicts [Poison] on foe.

Along with weapons, multiple armor variants are also available to select, as shown in Table 5. Armor is a more streamlined gear relative to weapons, only offering basic stat values. However, certain sets of armor reduce combatants' total attack accuracy, meaning that the player must be mindful when making their selection.

Table 5: Armor Types

Armor Type	**Basic stat values**
Leather Armor	0 WGT, 2 DEF
Soldier's Gear	20 WGT, 9 DEF
Rogue's Attire	10 WGT, 5 DEF

When a combatant attacks, attack accuracy must be considered. Attack accuracy determines the odds of whether the combatant's attack will land, dealing damage to the enemy combatant. The formula for attack accuracy is as follows:

$$\textbf{atkAccuracy} = \text{combatant1 } \textbf{HIT} \textbf{ – } \text{combatant1 } \textbf{WGT} \textbf{ – } \text{combatant2 } \textbf{AVOID} \qquad (1)$$

After determining atkAccuracy, the total damage dealt by a combatant to another is determined with the following formula:

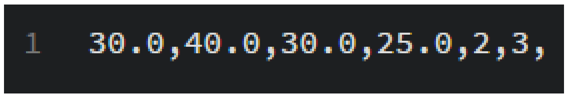
$$\textbf{damageDealt} = \text{combatant1 } \textbf{totalATK} - \text{combatant2 } \textbf{totalDEF} \qquad (2)$$

Note that combatant1 and combatant2 are roles taken by the player and enemy combatants. Whether combatant1 is the player or enemy and vice versa depends on who's turn it is. Further explanation is found within the Problem Solution.

The rules for CD are as follow:

1. Combatants must have a BST that equals 125. See Table 1 for more information.
2. Only two combatants are active at a time.
3. The player must be able to customize their combatant in some way.
4. All combatants are controlled only by the computer.
5. The game is turn-based; each combatant takes turns attacking each other.
6. Attacks are not guaranteed to hit. See (1).
7. Duels continue until either combatant is left with zero HP.
8. Duels must be successive. In other words, the player's combatant cannot be customized once the first duel begins.
9. The player's combatant can heal up to a maximum of three times in between duels.
10. The game is over when either the player's combatant loses, wins five duels, or a duel results in a stalemate.

After the duels are over, the player is given the opportunity to save their combatant's data. This data includes the stats found in Table 1 and the selections the player made when choosing a weapon and set of armor. Formatting of the text file, named "PlayerStats.txt" when created, is found in Figure 1.



```
30.0,40.0,30.0,25.0,2,3,
```

Figure 1: "PlayerStats.txt" Format

PROBLEM SOLUTION:

This section provides a description of the software implementation of CD. An overview of the software design is provided in a Unified Modeling Language (UML) class diagram (Figure 2), followed by a detailed description of each class and its methods. "Setters and getters" for class attributes are assumed to exist as needed, and thus are not represented within Figure 2.

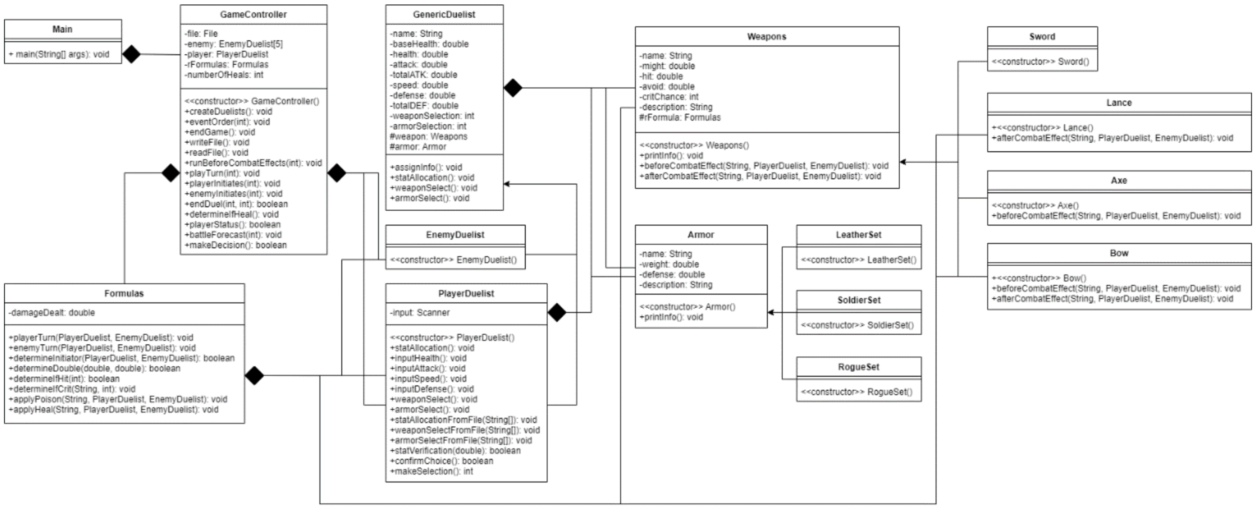

Main
+ main(String[] args): void
GameController
-file: File
-enemy: EnemyDuelist[5]
-player: PlayerDuelist
-rFormulas: Formulas
-numberOfHeals: int
<<constructor>> GameController()
+createDuelists(): void
+eventOrder(int): void
+endGame(): void
+writeFile(): void
+readFile(): void
+runBeforeCombatEffects(int): void
+playTurn(int): void
+playerInitiates(int): void
+enemyInitiates(int): void
+endDuel(int, int): boolean
+determineIfHeal(): void
+playerStatus(): boolean
+battleForecast(int): void
+makeDecision(): boolean
GenericDuelist
-name: String
-baseHealth: double
-health: double
-attack: double
-totalATK: double
-speed: double
-defense: double
-totalDEF: double
-weaponSelection: int
-armorSelection: int
#weapon: Weapons
#armor: Armor
+assignInfo(): void
+statAllocation(): void
+weaponSelect(): void
+armorSelect(): void
EnemyDuelist
<<constructor>> EnemyDuelist()
PlayerDuelist
-input: Scanner
<<constructor>> PlayerDuelist()
+statAllocation(): void
+inputHealth(): void
+inputAttack(): void
+inputSpeed(): void
+inputDefense(): void
+weaponSelect(): void
+armorSelect(): void
+statAllocationFromFile(String[]): void
+weaponSelectFromFile(String[]): void
+armorSelectFromFile(String[]): void
+statVerification(double): boolean
+confirmChoice(): boolean
+makeSelection(): int
Formulas
-damageDealt: double
+playerTurn(PlayerDuelist, EnemyDuelist): void
+enemyTurn(PlayerDuelist, EnemyDuelist): void
+determineInitiator(PlayerDuelist, EnemyDuelist): boolean
+determineDouble(double, double): boolean
+determineIfHit(int): boolean
+determineIfCrit(String, int): void
+applyPoison(String, PlayerDuelist, EnemyDuelist): void
+applyHeal(String, PlayerDuelist, EnemyDuelist): void
Weapons
-name: String
-might: double
-hit: double
-avoid: double
-critChance: int
-description: String
#rFormula: Formulas
<<constructor>> Weapons()
+printInfo(): void
+beforeCombatEffect(String, PlayerDuelist, EnemyDuelist): void
+afterCombatEffect(String, PlayerDuelist, EnemyDuelist): void
Armor
-name: String
-weight: double
-defense: double
-description: String
<<constructor>> Armor()
+printInfo(): void
LeatherSet
<<constructor>> LeatherSet()
SoldierSet
<<constructor>> SoldierSet()
RogueSet
<<constructor>> RogueSet()
Sword
<<constructor>> Sword()
Lance
+<<constructor>> Lance()
+afterCombatEffect(String, PlayerDuelist, EnemyDuelist): void
Axe
<<constructor>> Axe()
+beforeCombatEffect(String, PlayerDuelist, EnemyDuelist): void
Bow
+<<constructor>> Bow()
+beforeCombatEffect(String, PlayerDuelist, EnemyDuelist): void
+afterCombatEffect(String, PlayerDuelist, EnemyDuelist): void

Figure 2: Colosseum Duel UML Class Diagram

The general flow of action and information for the game is listed below.

1. The Main class creates an instance of the GameController class and requests the GameController to initiate and run the game.

2. The GameController class creates an instance of the PlayerDuelist class, five instances of the EnemyDuelist class, and an instance of the Formulas class.
3. The GameController class prompts the user whether they wish to load a save data file of a pre-existing combatant.
   3.1. If yes, the GameController class uses its readFile() method to read from the save data file, then passing the information to PlayerDuelist's statAllocationFromFile(), weaponSelectFromFile(), and armorSelectFromFile() methods to automatically set the information.
   3.2. If no, the GameController will load PlayerDuelist's statAllocation(), weaponSelect(), and armorSelect() methods, prompting the user with various questions and places to input values.
4. Once the player has finished customizing their combatant, the Main class runs GameController's eventOrder() method up to five times.
   4.1. The Main class runs this loop until either the player combatant has no more HP or the five-duel limit is surpassed.
5. Upon completion of the loop, the Main Class runs GameController's endGame() method to prompt the user if they wish to save their combatant's data, and then ends the software.

Further details of each class are provided below.

Main Class –
This class provides the initial starting point to run CD. This class is responsible for the following actions:
1. Create an instance of the GameController class.
2. Run GameController's eventOrder() method in a do-while loop until either the player combatant's HP falls to 0, or the five-duel limit is reached.
3. Once out of the do-while loop, run GameController's endGame() method, ending the software.

GameController class –
This class initializes the game by creating five instances of the EnemyDuelist class, and one instance of both the PlayerDuelist and Formulas classes. It is responsible for ensuring that duels run correctly. This class is responsible for the following actions:
1. Creates instances for all the combatants.
2. Allows the user the option to import save data and forego manual combatant creation.
3. Calls the PlayerDuelist class to run the statAllocation(), weaponSelect(), and armorSelect() methods if user is doing manual combatant creation.
4. Prints a battle forecast prior to the start of each duel.
5. Runs any effects from the player and enemy that occur before combat begins.

6. Runs the playTurn() method in a while loop, with the endDuel method used as its condition.
    6.1. Calls the Formulas class to determine whether the player or enemy is the initiator, and runs either playerInitiates() or enemyInitiates() based on the returned value.
    6.2. Runs any effects from the player and enemy that occur after each round of combat.
7. Once the loop is exited, prints out a line varying based on whether the player is successful or not.
8. Before the start of each duel after the initial one, as long as the player's combatant is alive, prompts the user if they would like to heal their combatant, up to a maximum of three times.

<u>GenericDuelist class</u> –
This class is the base template for player and enemy combatant creation, holding various attributes representing stats and selection choices. Refer to Table 1 for specific stats. This class is not directly accessed by the player.

<u>EnemyDuelist class</u> –
This class extends the GenericDuelist class and is an exact copy, only used to differentiate between the enemy and player. The program refers to this class when creating the enemy combatants. This class is responsible for the following actions:

1. Upon being called by the GameController class, runs the assignInfo() method.
    1.1. Runs the statAllocation() method, randomly generating values within the set ranges per each stat attribute, as shown in Table 1, until the sum is equal to the required BST.
    1.2. Runs the weaponSelect() method, randomly generating a value for the selection choice and assigning a weapon based on said value.
        1.2.1. Adds the enemy combatant's current ATK stat with the selected weapon's MT stat.
    1.3. Runs the armorSelect() method, randomly generating a value for the selection choice and assigning a set of armor based on said value.
        1.3.1. Adds the enemy combatant's current DEF stat with the selected set of armor's DEF stat.

<u>PlayerDuelist class</u> –
This class extends the GenericDuelist class and is used to contain all the player's customization regarding their combatant. This class is responsible for the following actions:

1. If player chose to load save data from file, runs the statAllocationFromFile(), weaponSelectFromFile(), and armorSelectFromFile() methods to automatically assign values to the necessary attributes. Otherwise, proceeds as follows:

2. Allows the player to manually input values for the stats listed in Table 1 with the statAllocation() method running the inputHealth(), inputAttack(), inputSpeed(), and inputDefense() methods.
    2.1. Verifies that the inputs are valid by either catching InputMismatchException errors or determining that the values are out of range.
    2.2. Verifies that the values add up to the required BST with the statVerification() method.
3. Allows the player to manually select a weapon of choice with the weaponSelect() method.
    3.1. Runs the makeSelection() method to get the user's input and verify that the input is valid with exception handling.
    3.2. Allows the player to confirm their selection by running the confirmChoice() method, using it as the condition for a do-while loop.
    3.3. Adds the player combatant's current ATK stat with the selected weapon's MT stat.
4. Allows the player to manually select a set of armor of choice with the armorSelect() method.
    4.1. Runs the makeSelection() method to get the user's input and verify that the input is valid with exception handling.
    4.2. Allows the player to confirm their selection by running the confirmChoice() method, using it as the condition for a do-while loop.
    4.3. Adds the player combatant's current DEF stat with the selected set of armor's DEF stat.

<u>Formulas class</u> –
This class determines multiple conditions necessary for duels to run correctly, as well as holding the order in which events occur per combatant's turn. This class also runs various special calculations depending on if a combatant has a special effect. This class is responsible for the following actions:

1. When called by the GameController class, runs the determineInitiator() method to determine whether the player or enemy combatant begins the duel.
2. Whenever it is the player combatant's turn, runs the playerTurn() method.
    2.1. Determines if the combatant lands their hit by running the determineIfHit() method.
    2.2. Sets the damage dealt based on Equation 2. If less than zero, the value is defaulted to zero.
    2.3. Determines if the combatant lands a critical hit by running the determineIfCrit() method.
    2.4. Prints out lines showcasing the action the player combatant took, and the damage, if any, that the enemy combatant received.
        2.4.1. If the enemy combatant's health is less than zero, defaults the value to zero.
    2.5. When called by the GameController class, runs the determineIfDouble() method to determine whether the player or enemy combatant can make a follow-up attack.

3. Whenever it is the enemy combatant's turn, runs the enemyTurn() method. The process is the same as outlined in 2.
4. After each round of combat, whenever called by the classes, first called by the GameController class, which extend the Weapons class, runs any skill effects that occur after combat, such as the applyPoison() and applyHeal() methods.

<u>Weapons class</u> –
This class holds the attributes necessary for each weapon. The values are either null or set to zero and replaced upon construction of one of four extended classes. This class is not directly accessed by the user.

<u>Sword class</u> –
This class extends the Weapons class and holds the Sword weapon. Upon creation, attribute values are replaced with values as shown in Table 4. This class is responsible for the following actions:
1. Upon player selection, run printInfo(), showing the stats and description of the weapon contained.

<u>Lance class</u> –
This class extends the Weapons class and holds the Lance weapon. Upon creation, attribute values are replaced with values as shown in Table 4. This class is responsible for the following actions:
1. Upon player selection, run printInfo(), showing the stats and description of the weapon contained.
2. Runs the afterCombatEffect() method after being called by the GameController class, applying [Heal] to its user by calling the Formulas class to run the applyHeal() method.

<u>Axe class</u> –
This class extends the Weapons class and holds the Axe weapon. Upon creation, attribute values are replaced with values as shown in Table 4. This class is responsible for the following actions:
1. Upon player selection, run printInfo(), showing the stats and description of the weapon contained.
2. Runs the beforeCombatEffect() method after being called by the GameController class, setting user's CRIT to 20.

<u>Bow class</u> –
This class extends the Weapons class and holds the Bow weapon. Upon creation, attribute values are replaced with values as shown in Table 4. This class is responsible for the following actions:

1. Upon player selection, run printInfo(), showing the stats and description of the weapon contained.
2. Runs the beforeCombatEffect() method after being called by the GameController class, setting user's AVOID to 10.
3. Runs the afterCombatEffect() method after being called by the GameController class, applying [Poison] to the user's foe by calling the Formulas class to run the applyPoison() method.

<u>Armor class</u> –
This class holds the attributes necessary for each set of armor. The values are either null or set to zero and replaced upon construction of one of three extended classes. This class is not directly accessed by the user.

<u>LeatherSet class</u> –
This class extends the Armor class and holds the Leather Armor set. Upon creation, attribute values are replaced with values as shown in Table 5. This class is responsible for the following actions:
1. Upon player selection, run printInfo(), showing the stats and description of the armor set contained.

<u>SoldierSet class</u> –
This class extends the Armor class and holds the Soldier's Gear set. Upon creation, attribute values are replaced with values as shown in Table 5. This class is responsible for the following actions:
1. Upon player selection, run printInfo(), showing the stats and description of the armor set contained.

<u>RogueSet class</u> –
This class extends the Armor class and holds the Rogue's Attire set. Upon creation, attribute values are replaced with values as shown in Table 5. This class is responsible for the following actions:
1. Upon player selection, run printInfo(), showing the stats and description of the armor set contained.

REFERENCES:
No references are needed for this document.

APPENDICES:

No appendices are needed for this document.