Lab 3: Abstraction

Instruction

1. Click the provided link on MyCourseVille to create your own repository.
2. Open Eclipse and then "File > new > Java Project" and set project name in this format **2110215_Lab3_2024_2{ID}_{FIRSTNAME}**
    - Example: **2110215_Lab3_2024_2_6731234521_Jolyne**.
3. Initialize git in your project directory
    - **Add .gitignore.**
    - Commit and push initial codes to your GitHub repository.
4. Implement all the classes and methods (copying initial code from the given lab file) following the details given in the problem statement file which you can download from CourseVille.
    - You should create commits with meaningful messages when you finish each part of your program.
    - Don't wait until you finish all features to create a commit.
5. Test your codes with the provided JUnit test cases, they are inside package **test.student**
    - **You have to write some tests by yourself**. Put them inside package **test.student**
    - Aside from passing all test cases, your program must be able to run properly without any runtime errors.
    - There will be additional test cases to test your code after you submit the final version, **make sure you follow the specifications in this document**.
6. After finishing the program, create a UML diagram and put the result image (**UML.png**) at the root of your project folder.
7. Export your project into a jar file called **Lab3_2024_2_{ID}** and place it at the root directory of your project. **Include your source code in the jar file**.
    - Example: **Lab3_2024_2_6731234521.jar**
8. Push all other commits to your GitHub repository.

1. Problem Statement: Card Game Mechanic

(This exercise is based on the game Inscryption by Daniel Mullins Games, released in October 2021. As well as Legends of Runeterra, released in April 2020)

Now that you understand how to create a deck manager for card game, now it's time to play. However, you still don't understand that there are more to card than just attack point, health point, and bloodcost. This game has 4 different types of unit cards, and 2 different types of spell cards.

1.1 Gameplay

(This image of the game is not entirely accurate to the game of this lab)

In this game, a player duels a fixed opponent who never plays any card, which means most of the card functions will only be used by the player.

1.1.1 Game Component

1.Player

- At the start of the game, each player will have a damage point set to 0. If the player's damage point is 5 or more than opponent's damage point, you win. You lose if the opposite happens.

2. Field

- Both players can summon a unit card from their hand on to a field, each player has 4 spaces to place their units (Player use lower row, opponent use upper row). You may replace a unit on the board with a different unit from your hand.

3.Deck

- Contains as many cards as you want, you can insert and remove any card at the card shop. However, you can only have up to 4 of the same cards.

4. Card

4.1 Unit Card

- A Unit card has Name, Flavor text, Blood cost, Attack power, and Health point.
- Player may play a unit card on to a field on any of 4 spaces, but player has to sacrifice a number of units on the field equal to the blood cost (0 blood cost unit can be summoned for free).
- Unit card on the field will attack a unit on the same column once on player's turn, dealing damage equal to its attack power. If there is no unit on the same column, this unit will deal damage to the opposite player.
- When a unit card takes damage (from a unit's attack or damage spell) and the unit's health point reduces to 0, that unit card will be removed from the field.
- There are 4 different types of unit cards.

4.1.1 Normal Unit Card

- A normal unit card with nothing special.

4.1.2 Leader Unit Card

- This card also contains Buff power and Buff health value.
- When this card enters a field, all the owner's units on the field will get power and health increase from Buff power and Buff health value.

4.1.3 Debuff Unit Card

- This card also contains Debuff power value.
- When this card attacks a unit, that unit's attack power will get reduced equal to Debuff power value (attack point can't go below 0).

4.1.4 Venom Unit Card

- When this card attacks a unit, that unit's health point will reduce to 0.
- When this card dies (health point reduces to 0 and leaves the field), it deals damage to its owner equal to its attack power.

4.2 Spell Card

- Each Spell card has Name, Flavor text, Blood cost, speed.
- Player may play a spell card on to a unit that is on a field, but player has to sacrifice a number of cards in their hand equal to the blood cost (0 blood cost spell can be casted for free).
- Spell card can have 2 speed values, Burst or Slow (Not Burst). Casting Slow speed spell will end the player's turn like summoning a unit card. Casting Burst speed spell will not end a turn, and player can play additional card.
- There are 2 different types of spell card.

4.2.1 Buff Spell Card

- This card also contains Power Increase value.
- This card can be casted on to player's unit to increase its attack power equal to Power Increase value.

4.2.2 Damage Spell Card

- This card also contains Damage value.
- This card can be casted on an opponent's unit to deal damage.

1.1.2 Game Flow

At the start of the game, the player shuffles his deck and draws 5 cards. While the opponent has a unit set on the field as you have chosen before the game.

1) Start with the player turn, the player draws a card.
2) Player chooses to play one card and pick its action.
    2.1) If the player plays a burst speed spell, return to 2)
3) Unit on the player's field executes attack from left to right, dealing damage to opponent's unit on the same column, or dealing damage to opponent's damage point.
4) Start the opponent turn, Unit on the opponent's field may/ may not execute attack from left to right the same way as player.
5) Repeat until the winning condition is met by either player.

2. Implementation Details:

To complete this assignment, you need to understand **Abstract Classes** and **Junit Test Cases**.

To test your understanding about abstraction, **we will not provide class diagrams for this assignment, and we will not indicate which methods and classes are abstract.** Try your best to figure it out. There are **three** abstract classes and **three** abstract methods.

There are **five** packages in the provided files: `application, player, card, deck` and `test.`

You will be implementing most of the class in the `card` and `deck` package (Every class is partly given, while PremadeDeck class doesn't need to be modified).

There are some test cases given in package `test.student`. These will help test your code. However, **some conditions are not tested** in these test cases. **Look for those conditions in the class details**. **You must create your own test cases for such cases.**

You can define any additional number of <u>private</u> (but not public, protected or package) fields and methods in addition to the fields and methods specified below. You are encouraged to try to group your logic into private methods to **reduce duplicate code as much as possible**.

** Noted that Access Modifier Notations can be listed below*

+ (public), # (protected), - (private)

2.1 package `deck`

1.1.3 **Class: `Deck`**

This class represents a deck the player will use. Deck has 3 attributes: **name, deckSize, and deckList**. Players can assign deck to use in gameplay. **Deck can only have at most 4 of the same cards.**

- name = name of the deck
- deckSize = number of cards the deck contains, need to change when Deck is created, when card is inserted into the Deck, and when card is removed from the Deck.
- deckList = list of the cards this deck contains. Each card must be assigned to an array without any empty slot.

2.1.1.1 Constructors

+ Deck(String name, Card[] deckList)	/*Fill Code*/ Construct a Deck object with the given name, deckList, and initialize `deckSize` to be the same size as the given deckList. (Hint: You should use Array (`Card[]`) to implement inventory.).

2.1.1.2 Methods

+ int insertCard(Card card) throws InsertCardFailedException	/*Fill Code*/ Insert the given card into **the bottom of card list (Recommend: create a new copy of deckList array with one more slot)** and modify deckSize to be accurate to the new deckList. Throw an InsertCardFailedException with message "`You can only put 4 of the same cards into the deck`" if there are already 4 of the same cards in the deck. Otherwise, return the new deckSize. (Given, but can be changed or adjusted according to how you code insert card).
+ Card removeCard(int slotNumber) throws RemoveCardFailedException	/*Fill Code*/ Unequip a card in the given slot number from the deck, rearrange the card from every slot after it to replace the empty slot. Modify deckSize to be accurate to the new deckList, and return the removed card. Throw an RemoveCardFailedException with message

	"`Number you insert exceed deck size`" when the slotNumber is greater than or equal to deckSize. (Already Implemented)
+ String toString()	You do **not** have to edit this method.
+ int Card[] getDeckList() + int String getName() + int getDeckSize()	/*Fill Code*/ Getter methods.
+ void setDeckSize(int deckSize)	/*Fill Code*/ Setter method.

2.2 package `card.base`

2.2.1 Class `Card`

This class is the base class of all cards in the card shop. Each card has its own name and description. `Card` **should never be instantiated into objects**, as it is only designed to be a base class so that consumer classes (like `Deck`) can easily use their subclasses.

2.2.1.1 Constructors

+ Card(String name, String flavorText, int bloodCost)	/*Fill Code*/ Initialize the card with the given name, flavor text, and blood cost.

2.2.1.2 Methods

+ String toString()	**This method should never be called from this class, it can only be called from subclasses.**
+ String getName() + String getFlavorText() + String getBloodCost()	/*Fill Code*/ Getter methods.
+ Object clone()throws CloneNotSupportedException	You do **not** have to edit this method.

2.2.2 Class `UnitCard`

This class is a base class of unit card, used to summon on to a field and attack in gameplay. It has power and health. `UnitCard` **should never be instantiated to objects,** as it is only

designed to be base class so that consumer classes (like the `Deck and Player`) can easily use their subclasses.

2.2.2.1 Constructors

+ UnitCard(String name, String flavorText, int bloodCost, int power, int health)	/*Fill Code*/ Initialize the character card with the given name, flavor text, bloodcost, power, and health.

2.2.2.2 Methods

+ int attackUnit(UnitCard u)	/*Fill Code*/ Called when using this UnitCard, this, to attack another UnitCard, u. **This method should never be called by this class. It can only be called from subclasses.**
+ int attackPlayer (Player opponent)	/*Fill Code*/ Deal damage to the opponent equal to this unit's power. Return the damage value. Hint: Use `takeDamage(int)` from Player class.
+ String toString()	You do **not** have to edit this method.
	/*Fill Code*/ Getter methods.
+ void setPower(int power)	/*Fill Code*/ Setter for power. If power is lower than 0, set this Unitcard's power to 0.
+ void setHealth(int health)	/*Fill Code*/ Setter for health. If health is lower than 0, set this Unitcard's health to 0.

2.2.3 Class `SpellCard`

This class is a base class of spell card, used to cast on units on the field. It has boolean name "isBurstSpeed". `SpellCard` **should never be instantiated to objects,** as it is only designed to be a base class so that consumer classes (like the `Deck and Player`) can easily use their subclasses.

2.2.2.1 Constructors

+ SpellCard(String name, String flavorText, int bloodCost, boolean isBurstSpeed)	/*Fill Code*/ Initialize the spell card with the given name, flavor text, blood cost, and isBurstSpeed

2.2.2.2 Methods

+ void castSpell (UnitCard unitCard)	/*Fill Code*/ Called when using SpellCard on a UnitCard. **This method should never be called by this class. It can only be called from subclasses.**
+ String toString()	You do **not** have to edit this method.
	/*Fill Code*/ Getter & Setter methods.

++++++++++++++++++++++++++++++++++++

2.3 package `card.type`

This package contains implementation of the **concrete** card in card shop. Some classes should be extended from `card.base` package while some should be extended from this package. In any case, every class must be extended from Card class at its root.

2.3.1 Class `NormalUnitCard`

This class represents "normal unit card" that has no additional logic or properties

2.3.1.1 Constructors

+ NormalUnitCard(String name, String flavorText, int bloodCost, int power, int health)	/*Fill Code*/ Initialize the normal unit card with the given name, flavor text, and other attributes.

2.3.1.2 Methods

+ int attackUnit(UnitCard unitCard)	/*Fill Code*/ Called when using NormalUnitCard to attack another unitCard. **Reduce unitCard health point by this normal unit card power.**

	Return a real damage dealt to this unitCard. **(You must also consider a case of this card power exceeding unitCard health)**

2.3.2 Class `DebuffUnitCard`

This class represents "Debuff unit card". Very similar to UnitCard but with 1 more property (**debuffPower**) and 2 more methods. Debuff Unit card's attack reduces the power of a unit.

2.3.2.1 Constructors

+ DebuffUnitCard(String name, String flavorText, int bloodCost, int power, int health, int debuffPower)	/*Fill Code*/ Initialize the debuff unit card with the given name, flavor text, blood cost, and other attributes.

2.3.2.2 Methods

+ int attackUnit(UnitCard unitCard)	/*Fill Code*/ Called when using DebuffUnitCard to attack another unitCard. **Reduce unitCard health point by this debuff unit card's power.** **Reduce unitCard power point by this debuff unit card's debuffpower.** **Return a real damage dealt to this unitCard.** **(You must also consider a case of this card power exceed unitCard health)**
+ int getDebuffPower()	/*Fill Code*/ Getter method.
+ void setDebuffPower(int level)	/*Fill Code*/ Setter method.

2.3.3 Class `VenomUnitCard`

This class represents "Venom unit card". Very similar to UnitCard but with 1 more method. Venom Unit card's attack kills a unit card regardless of its stat. But when a Venom unit dies, the player will take damage equal to its power.

2.3.3.1 Constructors

+ VenomUnitCard(String name, String flavorText, int bloodCost, int power, int health)	/*Fill Code*/ Initialize the venom unit card with the given name, flavor text, blood cost, and the attribute.

2.3.3.2 Methods

+ int attackUnit(UnitCard unitCard)	/*Fill Code*/ Called when using VenomUnitCard to attack another unitCard. **Reduce unitCard health point to 0** **Return a real damage dealt to this unitCard** **(Damage should be equal to unitCard's health before taking damage)**
+ int dead(Player player)	/*Fill Code*/ Called when VenomUnitCard's health becomes 0. **Player takes damage equal to venom unit card's power** Hint: Use `takeDamage(int)` from Player class **Return a damage dealt to player.**

2.3.4 Class `LeaderUnitCard`

This class represents "Leader unit card". Very similar to UnitCard but with 2 more properties **buffPower** and **buffHealth**.

2.3.4.1 Constructors

+ LeaderUnitCard(String name, String flavorText, int bloodCost, int power, int health, int buffPower, int buffHealth)	/*Fill Code*/ Initialize the leader unit card with the given name, flavor text, blood cost, and the attribute.

2.3.4.2 Methods

+ int attackUnit(UnitCard unitCard)	/*Fill Code*/ (Exactly like Normal Unit Card) Called when using LeaderUnitCard to attack another unitCard. **Reduce unitCard health point by this normal unit card power.**

	Return a real damage dealt to this unitCard. **(You must also consider a case of this card's power exceeding unitCard health)**
+ void buffUnit(UnitCard[] alliesCard)	/*Fill Code*/ Called when LeaderUnitCard is summoned on to the field. **Increase the power and health of every unit in alliesCard according to buffPower and buffHealth** (**You must check if there is a card on that slot of alliesCard array, if any slot is == null, do not attempt to increase stat as it will cause error during gameplay**)
+ int getBuffPower() + int getBuffHealth()	/*Fill Code*/ Getter methods.
+ void setBuffPower(int buffPower) + void setBuffHealth (int buffHealth)	/*Fill Code*/ Setter methods.
+ String toString()	You do **not** have to edit this method.

2.3.5 Class `BuffSpellCard`

This class represents "buff spell Card" Very similar to SpellCard but with 1 more property (**powerIncrease**) and 2 more methods. Buff Spell Card has to be casted on Player's unit on the field.

2.3.5.1 Constructors

+ BuffSpellCard(String name, String flavorText, int bloodCost, boolean isBurstSpeed, int powerIncrease)	/*Fill Code*/ Initialize the buff spell card with the given name, flavor text, blood cost, and the attribute.

2.3.5.2 Methods

+ void castSpell(UnitCard unitCard)	/*Fill Code*/ Called when cast this card on to a unit card **Increase unitCard Power by this spell's powerIncrease value.**
+ int getPowerIncrease()	/*Fill Code*/ Getter methods.
+ void setPowerIncrease(int powerIncrease)	/*Fill Code*/

	Setter methods. **If powerIncrease is lower than 1, set this BuffSpellcard's powerIncrease to 1**

2.3.6 Class `DamageSpellCard`

This class represents "Damage spell Card" Very similar to SpellCard but with 1 more property (**damage**) and 2 more methods. Damage Spell Card has to be casted on Opponent's unit on the field, dealing damage to its health point.

2.3.6.1 Constructors

+ DamageSpellCard(String name, String flavorText, int bloodCost, boolean isBurstSpeed, int damage)	/*Fill Code*/ Initialize the spell card with the given name, flavor text, blood cost, and the attribute.

2.3.6.2 Methods

+ void castSpell(UnitCard unitCard)	/*Fill Code*/ **Reduce unitCard Health point by this spell's damage value.**
+ int getDamage ()	/*Fill Code*/ Getter methods.
+ void setDamage(int damage)	/*Fill Code*/ Setter methods. **If damage is lower than 1, set this DamageSpellcard's damage to 1.**

2.4 package `test.student`

2.4.1 Class `TestDeck`

This class test Card class from card.base package. All constructors and some test cases have already been implemented, **except for 2 test cases which student needs to implement.**

2.4.1.1 Methods (Test cases)

void testRemoveCard()	/*Fill Code*/ **This test case must test 2 scenarios.** 1. Remove any card from the deck, then check if that card inside the deck is no longer there, the number of cards is reduced, and other cards are rearranged

	correctly. 2. Repeat scenario 1, but with a different deck.
void testRemoveNonExsistanceCard ()	/*Fill Code*/ Use assertThrows to check RemoveCardFailedException by removing card form the deck on the slot that doesn't have card in it.

2.4.2 Class `TestNormalUnitCard`

This class test NormalUnitCard class from card.type package. All constructors and test cases have already been implemented. You don't have to implement any test case in this class, but you can use it as a reference.

2.4.3 Class `TestDebuffUnitCard`

This class test DebuffUnitCard class from card.type package. All constructors and some test cases has already been implemented, **except for 2 test cases which student needs to implement.**

2.4.3.1 Methods (Test cases)

void testSetDebuffPower()	/*Fill Code*/ Test `setDebuffPower` method to see if it works correctly, especially when getting a negative value.
void testAttack()	/*Fill Code*/ Have 3 different DebuffUnitCards attack oppoUnit1, and check the damage dealt value, oppoUnit1 health point and attack after each attack.

2.4.4 Class `TestVenomUnitCard`

This class test VenomUnitCard class from card.type package. All constructors and some test cases has already been implemented, **except for 1 test case which student needs to implement.**

2.4.4.1 Methods (Test cases)

void testAttack()	/*Fill Code*/ Have 3 different VenomUnitCards attack oppoUnit1 **and change oppoUnit1's health point after every attack**, and check the damage dealt value, and oppoUnit1 health point after each attack.

2.4.5 Class `TestLeaderUnitCard`

This class test LeaderUnitCard class from card.type package. All constructors and some test cases has already been implemented, **except for 3 test cases which student needs to implement.**

2.4.5.1 Methods (Test cases)

void testSetBuffHealth()	/*Fill Code*/ Test if setBuffHealth works correctly, especially when given an out of range value.
void testAttack()	/*Fill Code*/ Have 2 different LeaderUnitCards attack oppoUnit1 total of 3 or more times, and check the damage dealt value, and oppoUnit1 health point after each attack. **(You must test a case of oppoUnit1 take damage exceed its health, not including this case will only give you half a score)**
void testBuffUnit()	/*Fill Code*/ Have 2 different LeaderUnitCards buff a unit in unitList. And check each unit's power and health point after each buff.

2.4.6 Class `TestBuffSpellCard`

This class test BuffSpellCard class from card.type package. All constructors and test cases has already been implemented. You don't have implement any test case in this class, but you can use it as a reference.

2.4.7 Class `TestDamageSpellCard`

This class test DamageSpellCard class from card.type package. All constructors and some test cases have already been implemented, **except for test cases which students need to implement.**

2.4.7.1 Methods (Test cases)

void testConstructor()	/*Fill Code*/ Check all attributes of one of the DamageSpellCard (Name, Flavor text, Blood cost, isBurstSpeed, and damage) (You may need another test case for when values go out of range.)
void testSetDamage()	/*Fill Code*/ Check if the setDamage method works correctly, especially when

	setting an out-of-range value.
void testCastSpell()	/*Fill Code*/ Have 3 different DamageSpellCards deal damage to unit. And check the unit's health point after each cast.

Score Criteria (UML score is 9 marks. Incorrect Abstract class will get -5 each.)
The total score is 55 (will be scaled down to 2.5)

Lab 3	Abstract		26	Lab 3	Junit		20
	test.student.TestBuffSpellCard		5		test.student.TestBuffSpellCard		0
		testConstructor	1				
		testConstructorNegativeValue	1				
		testSetPowerIncrease	1				
		testCastSpell	2				
	test.student.TestDamageSpellCard		0		test.student.TestDamageSpellCard		5
						testConstructor	2
						testSetDamage	1
						testCastSpell	2
	test.student.TestDebuffUnitCard		2		test.student.TestDebuffUnitCard		3
		testConstructor	1			testAttack	2
		testConstructorNegativeValue	1			testSetDebuffPower	1
	test.student.TestDeck		6		test.student.TestDeck		4
		testConstructor	1			testRemoveCard	2
		testInsertCard	3			testNonExsistanceRemoveCard	2
		testInsertCardMoreThan4Card	2				
	test.student.TestLeaderUnitCard		3		test.student.TestLeaderUnitCard		6
		testConstructor	1			testSetBuffHealth	1
		testConstructorNegativeValue	1			testAttack	3
		testSetBuffPower	1			testBuffUnit	2
	test.student.TestNormalUnitCard		6		test.student.TestNormalUnitCard		0
		testConstructor	2				
		testNonPositiveConstructor	1				

		testAttack	3				
	test.student.TestVemonUnitCard		4		test.student.TestVemonUnitCard		2
		testConstructor	1			testAttack	2
		testConstructorNegativeValue	1				
		testDead	2				