Initial issues in the provided codebase:
The current BattleGame code has lots of problems first starting with its organization as all the logic is packed into the main method, making it hard to read, update, and expand. The enemy's strategy is randomly chosen but not actually used in the game making the isAggressive and enemyStrategy variables useless. Also, the player's defense option chosen by number 2 in the code doesn't reduce damage or provide any other advantage. Finally, there is no input validation for the player's choices, which means entering an invalid option could cause the game to crash or behave unexpectedly.

I decided to use the **Factory Method Creational Pattern** to handle the creation of different enemy strategies (aggressive or patient) in the game. The goblin enemy can behave in two different ways, sometimes it attacks aggressively, and other times it acts patiently. The Factory Method Pattern allows the game to decide which strategy to use at runtime.This makes the game unique and unpredictable as the Goblin's behavior can change each time the game is played. The assignment of enemy strategy was originally never being used in the initial code.

By using a factory, the logic for creating strategies is separated from the main game logic and allows me to update the factory and add a new strategy class without modifying the existing game or enemy code. The factory can be reused in other parts of the game or in other games to create different types of objects, not just strategies.The Factory Method is implemented by first defining an interface (Strategy) to ensure all strategies have the same structure. This allows the Enemy class to use any strategy interchangeably.

```
1   public interface Strategy {
2       void execute(Player player, Enemy enemy);
3   }
```

Then I created two concrete strategies for the enemy strategies:
AggressiveStrategy: Deals higher damage to the player.
PatientStrategy: Waits patiently for the player's turn.

```
import java.util.Random;

public class AggressiveStrategy implements Strategy {
    private final Random random = new Random();

    @Override
    public void execute(Player player, Enemy enemy) {
        int damage = random.nextInt(bound:5) + 2; // More aggressive attack
        player.takeDamage(damage);
        System.out.println("The Goblin strikes aggressively for " + damage + " damage!");
    }
}
```

```
public class PatientStrategy implements Strategy {
    @Override
    public void execute(Player player,Enemy enemy) {
        System.out.println(x:"Goblin waits patiently, preparing for the next move.");
    }
}
```

The next step would be creating the factory as the StrategyFactory class now uses a Random object to decide whether to create an AggressiveStrategy or a PatientStrategy. The createStrategy method generates a random boolean value (true or false) and uses it to determine which strategy to create.

```java
import java.util.Random;

public class StrategyFactory {
    private static final Random random = new Random();

    public static Strategy createStrategy() {
        boolean isAggressive = random.nextBoolean();
        Strategy enemyStrategy = isAggressive ? new AggressiveStrategy() : new PatientStrategy();
        return enemyStrategy;
    }
}
```

The Game class still uses the StrategyFactory.createStrategy() method to create the enemy's strategy when the game starts. This strategy is then passed to the Enemy object.

```java
public Game() {
    this.player = new Player();
    Strategy enemyStrategy = StrategyFactory.createStrategy();
    this.enemy = new Enemy(health:20, enemyStrategy);
    this.scanner = new Scanner(System.in);
    this.playerTurn = new PlayerTurn(player, enemy, scanner);
}
```

When the Game constructor is called, it creates a Player object. It then calls StrategyFactory.createStrategy() to create a random strategy (AggressiveStrategy or PatientStrategy). The created strategy is passed to the Enemy object, which uses it to define its behavior. The Enemy class has been updated to include health management and the ability to perform actions using its assigned strategy. The performAction method calls the execute method of the strategy, which determines how the enemy attacks the player.

```java
public class Enemy {
    private int health;
    private final Strategy strategy;

    public Enemy(int health, Strategy strategy) {
        this.health = 20;
        this.strategy = strategy;
    }

    public int getHealth() {
        return health;
    }

    public void takeDamage(int damage) {
        health -= damage;
        if (health < 0) {
            health = 0;
        }
    }

    public void performAction(Player player) {
        strategy.execute(player, this);


    }

    public boolean isAlive() {
        return health > 0;
    }
}
```

I chose the **Bridge Method structural pattern** to separate the actions (like Attack, Defend, and Heal) from the player and enemy classes. This allows the actions to be defined independently and reused across different parts of the game. If I want to add a new action, like a "Special Attack," I can simply create a new class that implements the Action interface without changing the player or enemy classes. It also allows the player and enemy to perform actions without

needing to know the details of how those actions work. In the code, the Action interface acts as the bridge between the player/enemy and the specific actions. The AttackAction, DefendAction, and HealAction classes implement this interface and define what happens when each action is executed. During the player's turn, the game creates an instance of the chosen action (like AttackAction) and calls its execute method, passing the player and enemy as parameters.

The Bridge Method Pattern is first implemented by the Action interface as it acts as the bridge between the player/enemy and the specific actions. It defines a common method execute that all actions must implement:

```java
public interface Action {
    void execute(Player player, Enemy enemy);
}
```

Next, each action (Attack, Defend, Heal) is implemented as a separate class that implements the Action interface. For example, in the AttackAction.java, DefendAction.java, & HealAction.java file.

```java
public class AttackAction implements Action {
    private final int damage;

    public AttackAction() {
        this.damage = new Random().nextInt(bound:5) + 1;
    }

    @Override
    public void execute(Player player, Enemy enemy) {
        enemy.takeDamage(damage);
        System.out.println("You slashed the " + "Goblin" + " for " + damage + " damage.");
    }
}
```

During the player's turn, the game uses the bridge to execute the chosen action. This happens in the PlayerTurn class:

```java
public class PlayerTurn {
    private final Player player;
    private final Enemy enemy;
    private final Scanner scanner;

    public PlayerTurn(Player player, Enemy enemy, Scanner scanner) {
        this.player = player;
        this.enemy = enemy;
        this.scanner = scanner;
    }

    public void execute() {
        System.out.println(x:"\nYour Turn:");
        System.out.println(x:"Choose an action: (1) Attack (2) Defend (3) Heal");
        int choice = scanner.nextInt();
        player.setDefending(defending:false);
        if (choice == 1) {
            Action action = new AttackAction();
            action.execute(player, enemy);
        } else if (choice == 2) {
            Action action = new DefendAction();
            action.execute(player, enemy);
        } else if (choice == 3) {
            Action action = new HealAction();
            action.execute(player, enemy);
        } else {
            System.out.println(x:"Invalid choice. Try again.");
        }
    }
}
```

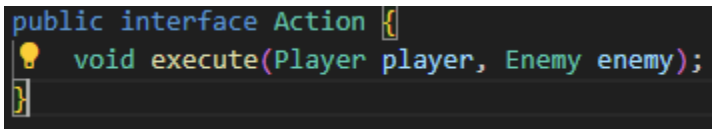
The player chooses an action (e.g., Attack, Defend, or Heal).The corresponding action class (e.g., AttackAction) is created.The execute method of that action is called, passing the player and enemy as parameters.The action performs its logic (e.g., dealing damage, setting defense, or healing) without the PlayerTurn class needing to know how the action works.

The same bridge pattern is used for the enemy's actions. The Enemy class has a Strategy object (which is similar to the Action interface) that defines its behavior. The Strategy interface is another bridge that allows the enemy to perform actions (like attacking or defending) without being tightly coupled to the specific logic of those actions. This makes it easy to change the enemy's behavior by swapping out different strategies.

```
public class Enemy {
    private int health;
    private final Strategy strategy;

    public Enemy(int health, Strategy strategy) {
        this.health = 20;
        this.strategy = strategy;
    }

    public int getHealth() {
        return health;
    }

    public void takeDamage(int damage) {
        health -= damage;
        if (health < 0) {
            health = 0;
        }
    }

    public void performAction(Player player) {
        strategy.execute(player, this);


    }

    public boolean isAlive() {
        return health > 0;
    }
}
```

I chose the **Command Method Behavioral Pattern** to handle the player's actions in the game. This pattern allows each action (like attacking, defending, or healing) to be encapsulated as an object, making the code more organized and flexible. Using the Command Pattern improves the game in several ways. First, it separates the logic of each action into its own class (like AttackAction, DefendAction, and HealAction). This makes the code easier to read and maintain as the original implementation was messy and hard to maintain. In the initial code, all the logic for player actions (attack, defend, heal) was combined into a single if-else block within the main game loop. This makes the code difficult to read, extend, and debug. In conclusion, the command pattern makes the player's turn logic cleaner, as the PlayerTurn class only needs to call the execute method on the chosen action, without worrying about the details of how each action works.

The command method is first implemented by the Action Interface (The Command).
It defines a single method, execute, which all actions must implement. This is like a contract that says, "Every action must have a way to execute itself." By defining a common interface, we ensure that all actions follow the same structure. This makes it easy to add new actions in the future without changing the existing code.



```
public interface Action {
    void execute(Player player, Enemy enemy);
}
```

Next, each action (attack, defend, heal) is implemented as a separate class that implements the Action interface. These classes contain the specific logic for what happens when the action is

executed. Each action is encapsulated in its own class, making the code modular and easy to maintain.The logic for each action is isolated, so changes to one action (e.g., modifying how damage is calculated) don't affect other actions.

```java
import java.util.Random;

public class AttackAction implements Action {
    private final int damage;

    public AttackAction() {
        this.damage = new Random().nextInt(bound:5) + 1;
    }

    @Override
    public void execute(Player player, Enemy enemy) {
        enemy.takeDamage(damage);
        System.out.println("You slashed the " + "Goblin" + " for " + damage + " damage.");
    }
}
```

```java
public class DefendAction implements Action {
    @Override
    public void execute(Player player, Enemy enemy) {
        System.out.println(x:"You raise your shield, bracing for impact.");
        player.setDefending(defending:true);
    }
}
```

```java
import java.util.Random;

public class HealAction implements Action {
    private final int healAmount;

    public HealAction() {
        this.healAmount = new Random().nextInt(bound:3) + 1;
    }

    @Override
    public void execute(Player player, Enemy enemy) {
        player.heal(healAmount);
        System.out.println("You heal yourself for " + healAmount + " HP.");
    }
}
```

The PlayerTurn class (The Invoker) is responsible for asking the player what action they want to take and then executing that action. It doesn't need to know how the action works—it just calls the execute method on the chosen action. The PlayerTurn class doesn't need to know the details of how each action works. It simply creates the appropriate action object and calls its execute method. The use of the Command Pattern allows the PlayerTurn class to be decoupled from the action logic, making the code more flexible and easier to extend as shown below:

```java
import java.util.Scanner;

public class PlayerTurn {
    private final Player player;
    private final Enemy enemy;
    private final Scanner scanner;

    public PlayerTurn(Player player, Enemy enemy, Scanner scanner) {
        this.player = player;
        this.enemy = enemy;
        this.scanner = scanner;
    }

    public void execute() {
        System.out.println(x:"\nYour Turn:");
        System.out.println(x:"Choose an action: (1) Attack (2) Defend (3) Heal");
        int choice = scanner.nextInt();
        player.setDefending(defending:false);
        if (choice == 1) {
            Action action = new AttackAction();
            action.execute(player, enemy);
        } else if (choice == 2) {
            Action action = new DefendAction();
            action.execute(player, enemy);
        } else if (choice == 3) {
            Action action = new HealAction();
            action.execute(player, enemy);
        } else {
            System.out.println(x:"Invalid choice. Try again.");
        }
    }
}
```

Finally, the Player and Enemy Classes (The Receivers) are the ones that actually perform the actions. For example: The Player class handles healing and defending. The Enemy class takes damage when attacked.

```java
import java.util.Random;

public class Player {
    private int health;
    private boolean defending;

    public Player() {
        this.health = 20;
        this.defending = false;
    }

    public void takeDamage(int damage) {
        if (defending) {
            damage /= 2; // Reduce damage by half when defending
        }
        health -= damage;
        if (health < 0) {
            health = 0;
        }
    }

    public void setDefending(boolean defending) {
        this.defending = defending;
    }



    public void heal(int healAmount) {
        Random random = new Random();
        int heal = random.nextInt(bound:3) + 1;
        health += heal;
    }



    public boolean isAlive() {
        return health > 0;
    }
}
```

```java
public class Enemy {
    private int health;
    private final Strategy strategy;

    public Enemy(int health, Strategy strategy) {
        this.health = 20;
        this.strategy = strategy;
    }

    public int getHealth() {
        return health;
    }

    public void takeDamage(int damage) {
        health -= damage;
        if (health < 0) {
            health = 0;
        }
    }

    public void performAction(Player player) {
        strategy.execute(player, this);



    }

    public boolean isAlive() {
        return health > 0;
    }
}
```

The Player and Enemy classes are the "receivers" of the commands. They are the ones that actually change state (like health or defense) when an action is executed. By separating the action logic (in the command classes) from the state management (in the Player and Enemy classes), clean separation is achieved..This makes the code easier to test and maintain, as each class has a single responsibility.

In conclusion, adding these patterns has helped the flow of the code and improved the overall game:

The Factory Method Pattern was chosen to improve the game by making the enemy's behavior more dynamic, organized, and easier to manage. In the initial source code, the enemy's strategy (aggressive or patient) was determined using a simple boolean flag (isAggressive). This approach made the code harder to read, maintain, and extend. For example, if you wanted to add more strategies or modify how strategies are chosen, you'd have to change the game logic, which could introduce bugs or make the code more confusing. By introducing the Factory Method Pattern, the creation of enemy strategies is moved to a separate StrategyFactory class. The factory decides which strategy to create (e.g., aggressive or patient) based on random logic, and the game simply uses the strategy without worrying about how it was created.

The Command Pattern was chosen to improve the game by making player actions more organized, and easier to extend. In the initial source code, the logic for player actions (attack, defend, heal) was directly embedded in the main game loop using a series of if-else statements. By introducing the Command Pattern, each action is encapsulated into its own class (e.g., AttackAction, DefendAction, HealAction), and the game simply creates and executes the appropriate action based on the player's choice. This separation makes the code cleaner, more flexible, and easier to extend without affecting the rest of the game logic.

The Bridge Pattern was chosen to improve the game by making the actions (like Attack, Defend, and Heal) more reusable, and easier to extend. In the initial source code, the actions were tightly coupled to the player and enemy logic, making it harder to add new actions or modify existing ones without changing the core game logic. For example, if you wanted to add a new attack action you'd have to modify the player and enemy classes, which could introduce bugs or make the code more complex. By introducing the Bridge Pattern, the actions are separated into their own classes (like AttackAction, DefendAction, and HealAction) that implement a common Action interface. This allows the player and enemy to execute actions without needing to know how they work, making the code cleaner, more organized, and easier to maintain. For instance, adding a new action now only requires creating a new class that implements the Action interface, without touching the player or enemy logic.