Code Review: Lab05

File: Action.java , Line: 50-71

The `Attack` class performs both damage calculation and management of the defender's health and shield. This could violate the Single Responsibility Principle, leading to less maintainable code.

```
50      public void takeDamage(int damage) {
51          hp = hp - damage;
52          if (isDefend) {
53              if (shield - damage <= 0) {
54                  hp = hp + shield;
55                  defender.setHp(hp);
56                  defender.setShield(0);
57                  if (hp <= 0) {
58                      defender.setHp(0);
59                  }
60              } else {
61                  shield = shield - damage;
62                  defender.setShield(shield);
63              }
64          } else {
65              if (hp <= 0) {
66                  defender.setHp(0);
67              } else {
68                  defender.setHp(hp);
69              }
70          }
71      }
```

File: Action.java , Line: 50-71

- utilize direct numeric values without clear explanations. Constants or named variables could enhance readability.
- Variable names like `hp`, `shield`, and `isDefend` are quite succinct but might benefit from more descriptive names.

```
50      public void takeDamage(int damage) {
51          hp = hp - damage;
52          if (isDefend) {
53              if (shield - damage <= 0) {
54                  hp = hp + shield;
55                  defender.setHp(hp);
56                  defender.setShield(0);
57                  if (hp <= 0) {
58                      defender.setHp(0);
59                  }
60              } else {
61                  shield = shield - damage;
62                  defender.setShield(shield);
63              }
64          } else {
65              if (hp <= 0) {
66                  defender.setHp(0);
67              } else {
68                  defender.setHp(hp);
69              }
70          }
71      }
```

File: Character.java , Line: 168-188

Consider adding comments to clarify the logic behind the weight thresholds and their respective speed reduction percentages

```
168         public void setRunSpeed() {
169 
170             runSpeed = 34;
171 
172             if (weapon[0] != null) {
173                 totalWeight += weapon[0].getWeight();
174             }
175 
176             if (weapon[1] != null) {
177                 totalWeight += weapon[1].getWeight();
178             }
179 
180             if (totalWeight > 150 && totalWeight <= 200) {
181                 runSpeed *= 0.75; // วิ่งช้าลง 25%
182             } else if (totalWeight > 200 && totalWeight <= 250) {
183                 runSpeed *= 0.7; // วิ่งช้าลง 30%
184             } else if (totalWeight > 250) {
185                 runSpeed *= 0.5; // วิ่งช้าลง 50%
186             }
187         }
188 
```

File: Character.java

There are no explicit checks for unexpected or invalid values that might cause issues, like null values or boundary conditions.Should add checks at the beginning of methods (e.g., `equipWeapon`, `setRunSpeed`) for unexpected null values or boundary conditions to prevent potential issues.

***fixing by give conditions when the equipment is null in public method is NOT A GOOD IDEA line 290-318**

```
208     public void equipWeapon(Equipment weapon1) {
209         if (weapon1.getType().equals("sword")||weapon1.getType().equals("dagger")||weapon1.getType().equals("wand")) {
210             weapon[0] = weapon1;
211             setRunSpeed();
212             setDamage();
213         } else if (weapon1.getType().equals("shield")) {
214             weapon[1] = weapon1;
215             setRunSpeed();
216             setShield();
217         }
```

```
233     public void equipWeapon(Equipment weapon1, Equipment weapon2) {
234         weapon[0] = weapon1;
235         weapon[1] = weapon2;
236         setRunSpeed();
237         setDamage();
238         setShield();
239     }
240 
```

```
254     public void equipAccessory(Equipment accessory1, Equipment accessory2) {
255         accessory[0] = accessory1;
256         accessory[1] = accessory2;
257         applyAccessoryEffects();
258     }
259 
```

File: Equipment.java , Line: 115-129

Refactor this method to avoid repeating similar printing logic for different equipment types. Create separate methods or utilize polymorphism to handle specific equipment types.

```
115     public void getWeaponInfo(){
116         if("sword".equals(type)){
117             System.out.println("Name: " + name + "\tLevel: " + level + "\tDamage: "
118             + damage + "\tWeight: " + weight);
119         }
120         if("shield".equals(type)){
121             System.out.println("Name: " + name + "\tLevel: " + level + "\tReduces Damage: "
122             + reducesDamage + "\tWeight: " + weight);
123         }
124         if("dagger".equals(type) || "wand".equals(type) || "accessory".equals(type)){
125             System.out.println("Name: " + name + "\tLevel: " + level + "\tDamage: "
126             + damage + "\tWeight: " + weight + "\tBounsHp: " + bonusHp + "\tBounsSpeed: " + bonusSpeed + "\tBonseMana: " + bonusMana);
127         }
128     }
129 }
```

File: Equipment.java , Line: 64,69-74

Variable names like `dmgOrRdmg` could be improved for better readability and clarity

```
64     public Equipment(String type, String name, int level, int dmgOrRdmg, int weight, int bonusHp, int bonusSpeed, int bonusMana) {
65         this.name = name;
66         this.level = level;
67         this.type = type.toLowerCase();
68 
69         if(this.type.equals("sword") || this.type.equals("dagger") || this.type.equals("wand")){
70             this.damage = dmgOrRdmg * (2 + level);
71         }
72         if(this.type.equals("shield")){
73             this.reducesDamage = dmgOrRdmg + (3 * level);
74         }
75         this.weight = weight;
76         this.bonusHp = bonusHp;
77         this.bonusSpeed = bonusSpeed;
78         this.bonusMana = bonusMana;
79     }
80 
```

Advantage

- The separation of `Defend` and `Attack` into nested classes helps in maintaining a clear separation of concerns, focusing on specific actions related to combat.
- Encapsulation is used effectively to group related functionalities (`Defend` and `Attack`) within the `Action` class, improving code organization and readability.