IGL s1 2023-2024

Maksym Lytvynenko

Exercice 1

A) Is this framework adaptable to new hero/obstacle types?

The framework IS adaptable because:

- It uses abstract classes (`Heros` extends `Element`)
- It uses polymorphism (List<Obstacle>)
- New types can be added by extending existing classes
- The injection methods allow adding any subtype of Heros or Obstacle

B) Example of an extension:

```
public class SuperPacman extends Heros {
        @Override
        public String getDescription() { return "Super Pacman"; }
        @Override
        public void rencontrer(Obstacle o) {
        System.out.println(this.getDescription() + " has encountered " + o.getDescription()); } }
```

C)

1) The control inversion is in the method getDescription() which is abstract in Heros and defined in its child class
2) Easy extensibility when we don't need to modify the Heros class when adding new types of heroes, each different type of hero MUST provide its own description, so when calling `getDescription()` from Heros we knows it will get the right description

D)

1)
```
g.injectHeros( new Pacman() );
g.injectObstacle( new Fantome() );
```
2) Without polymorphic assignment, we would need:
    - Separate lists for each type of obstacle
    - Different methods for each hero-obstacle combination
    - Lots of type checking and casting

E)

1) `this` has the static type `Heros` because we're in the `Heros` class
   `o` has the static type `Obstacle` as declared in the parameter
2) For `this`:
    - Cannot be just a `Heros` since it's abstract
    - Could be a `Pacman` instance (as shown in listing 1's main method)
    - Could be any other subclass of `Heros` that we might create (like SuperPacman)

   For `o`:
    - Could be a `Fantome` instance
    - Could be a `Cerise` instance
    - Could be any other subclass of `Obstacle`

Exercice 2

A)

- Violates Open-Closed Principle (need to modify code for new types)
- Hard to maintain (growing if-else chain)
- More prone to errors
- Less type-safe

B)

1) When we write `h.rencontrer(f)`, we're sending a "message" to the object referenced by `h` to encounter the object referenced by `f`.

   For variable `h`:

   - Static type: `Heros`
   - Dynamic type: `Pacman`

   For variable `f`:

   - Static type: `Fantome`
   - Dynamic type: `Fantome`

   Thus. the rencontrer(Fantome f) being executed is from the Pacman class that redefines the abstract method of its parent Heros

2)

   The compiler would use the following logic:

   - It sees variable `h` of static type `Heros`
   - It looks for a method `rencontrer(Fantome)` in `Heros` class
   - Not finding this specific method, it would look for more general versions
   - It would find `rencontrer(Obstacle)` since `Fantome` is an `Obstacle`
   - Therefore, it would use this more general method instead
   - At runtime, it would print "Suis-je utile?" instead of the specific ghost encounter message

3)

   The compiler would use the following logic:

   - It looks at the static type of `f`, which is now `Obstacle`
   - It searches for a method that takes an `Obstacle` parameter
   - It finds `rencontrer(Obstacle)` in the `Heros` class
   - It doesn't consider the more specific `rencontrer(Fantome)` because it's working with the static type
   - The code would compile
   - At runtime, it would call `rencontrer(Obstacle)`
   - We would get "Suis-je utile?" printed instead of the specific ghost encounter message

Exercice 3

A) State Pattern Analysis:

Advantages:

1. Clear representation of different behavioral states
    - The normal state and invincible state would be distinct classes
    - Each state would have its own clear implementation of how to handle ghost encounters
    - State transitions would be explicit and easy to track
2. Clean separation of behaviors
    - Normal behavior stays in one class
    - Invincible behavior stays in another class
    - New states could be added without modifying existing ones

Disadvantages:

1. State management complexity
    - Need to handle state transitions carefully
    - Need to ensure proper state cleanup
2. Increased number of classes
    - Each state requires its own class
    - Can lead to class proliferation if many states are added

B) Decorator Pattern Analysis:

Advantages:

1. Dynamic behavior modification
    - Can add invincibility at runtime
    - Original Pacman remains unchanged
    - Easy to add and remove the behavior
2. Composition over inheritance
    - More flexible than inheritance-based solutions
    - Can combine multiple decorators if needed
    - Easier to test and modify
3. Clean separation of concerns
    - Base behavior stays in original class
    - Additional behaviors are cleanly separated
    - Easy to add new decorators

Disadvantages:

1. Object identity issues
    - Decorated object is not the same as original
    - Can cause problems with equality checks
    - Reference tracking becomes more complex
2. Potential complexity with multiple decorators
    - Order of decorators might matter
    - Interaction between decorators needs careful consideration

C) Making the Choice:
For this specific problem, I would choose the State pattern because the behavior change is a clear state transition (Normal → Invincible → Normal). The states are well-defined and mutually exclusive. The transition conditions are clear (eating cherry, time expiration)

```
public class Hero {
	private HeroState currentState;
	public Hero() { // Start in normal state
		this.currentState = new NormalState(this); }

	// Method to change state
	public void setState(HeroState newState) { this.currentState = newState; }

	// Delegate ghost encounters to current state
	public void rencontrer(Fantome f) {
		currentState.rencontrer(f);
		// After each action, check if state should change
		currentState.updateState(); }

	// Called when hero collects a cherry
	public void collectCherry() {
		// Transition to invincible state
		setState(new InvincibleState(this, 10000)); // 10 seconds of invincibility
		System.out.println("Became invincible!"); }
}
abstract class HeroState {
	protected Hero hero;
	abstract void rencontrer(Fantome f);
	abstract void updateState(); // Check if state should change
}
class NormalState extends HeroState {
	void rencontrer(Fantome f) { // Normal ghost encounter behavior }
	void updateState() { // Check if hero ate a cherry }
}
class InvincibleState extends HeroState {
	private long endTime;
	InvincibleState(long duration) {
	this.endTime = System.currentTimeMillis() + duration; }
	void rencontrer(Fantome f) { // Send ghost to prison }
	void updateState() { if (System.currentTimeMillis() > endTime) {
		hero.setState(new NormalState()); } }
}
```

Exercice 4

A) Static tape of o is Obstacle, so the compiler will choose the method that takes Obstacle as an argument

B) To solve this without modifying the existing methods, we need to use the Visitor pattern.

```
public abstract class Heros extends ElementJeu {
        public void rencontrer(Obstacle o) {
                // Make the obstacle accept the hero as visitor
                o.acceptVisitor(this); }
        // Specific encounter methods remain the same
}

public interface Obstacle { void acceptVisitor(Heros h); }

public class Fantome implements Obstacle {
        @Override
        public void acceptVisitor(Heros h) {
                // This calls the specific rencontrer method for Fantome
                h.rencontrer(this); }
}
public class Cerise implements Obstacle {
        @Override
        public void acceptVisitor(Heros h) {
                // This calls the specific rencontrer method for Cerise
                h.rencontrer(this); }}
```

C) Any new obstacle type just needs to implement the Obstacle interface, provide its acceptVisitor implementation, no changes needed to existing code

```
public class Banane implements Obstacle {
        @Override
        public void acceptVisitor(Heros h) {
        h.rencontrer(this); // Will call the specific method for Banane }
}
```

D) Yes, this solution is compatible with the State pattern because:

- When the game engine calls `h.rencontrer(o)`, the visitor pattern ensures the right specific method is called
- Once the right method is selected, the state pattern takes over
- State transitions still work as before
- Visitor pattern: Handles method selection (which rencontrer method to call)
- State pattern: Handles behavior implementation (what to do in that method)

E) This exemplifies the Open-Closed Principle: The system is open for extension (new obstacle types), but closed for modification (existing code doesn't change)