GROUP PROJECT

Arkham Horror. Board game

Presented by Polina Batova, Rustem Zhumabek, Igor Letunovsky

Course Instructor: Nurtas Nadirov

Course: Software Design Patterns

Introduction

- Arkham Horror is a popular board game based on books of H.P. Lovecraft.
- Player (investigator) should figure out what happens in the city Arkham and save it

- Investigators can travel through the city, fight with monsters, gather resources, update own skills, research clues and ward.
- Each investigator has own attributes like health and sanity, skills, and cards representing their asset (items, spells, allies).

Tools and Design Patterns

1. Coding Language: Java, Java FX

2. Interface: GUI

3. Patterns: Strategy, Observer, Facade, State, Singleton, Builder, Factory

DISCLAIMER

This board game itself used one of the most complicated mechanics among other board games. This is why a FULL port is a big challenge. Even this little we've touched required huge effort. Understand our position that we've implemented only major or most impactful features of the game. For example, we completely omitted 2 of 4 turn stages, did not implement some of basic actions and made only basic and rough reasisation of item usage.

Project Structure

Game & Monsters

Publisher

abstract addMonster()
abstract removeMonster()
abstract notifyMonsters()

Game

implements Publisher

Subscriber

abstract update()

Monster

implements Subscriber
abstract play()

MonsterFactory

createMonster()

MonsterType (enum)

HUNTER,
LURKER,
PATROL

Hunter

Lurker

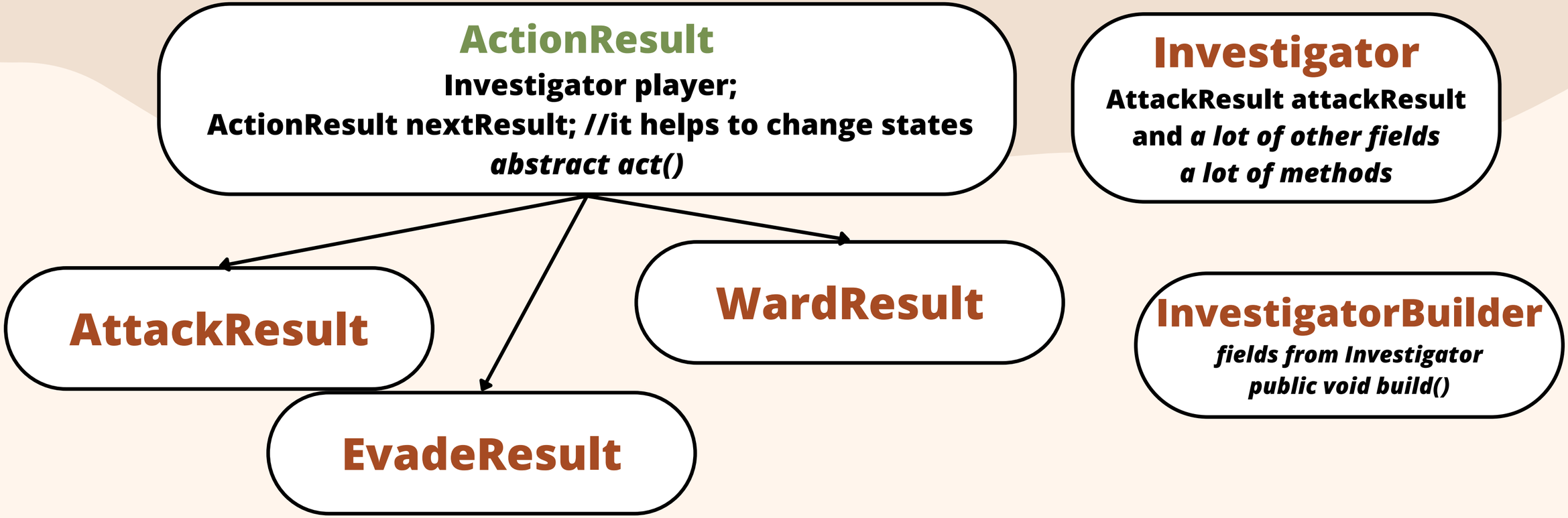
Patrol

Here we can see the Observer and the Factory patterns

Project Structure

Investigator & ActionResult

ActionResult implements the State Pattern. It is used to define the result of the player's action



Here we can see the Builder and the State patterns

GameFacade makes work with the class Game easier. Moreover, GameFacade is a singleton. As we don't need other instances of this class.

Project Structure

Game & Assets

Action
abstract use(Investigator i)

AssetCard
private Action actionStrategy
@Override public void use(inv i)

For implementation of different actions of cards we used the Strategy pattern. It allowed us to give objects of the same class different behavior

GameFacadeInterface
abstract void startGame();

GameFacadeIAzatoth
implements GameFacadeInterface
static GameFacadeIAzatoth gameFacadeIAzatothInstance;
Game game;
MonsterFactory monsterFactory
@Override public void startGame()
public static GameFacadeIAzatoth createGameFacadeIAzatoth()

Game
implements Publisher

Item

Ally

ShotgunStrategy
private int damage;
@Override public void use(inv i)

MagnifierStrategy
private int plus;
@Override public void use(inv i)

Here we can see the Facade, Singleton and Strategy pattern

Builder Pattern

A lot of build options available

```
1 usage  PollyBreak
public InvestigatorBuilder name(String name) {
    this.name=name;
    return this;
}

no usages  PollyBreak
public InvestigatorBuilder startSpace(Node node) {
    this.startNode = node;
    return this;
}

1 usage  PollyBreak
public InvestigatorBuilder skills(int lore, int influence, int observation, int strength, int will) {
    this.lore = lore;
    this.influence = influence;
    this.observation = observation;
    this.strength = strength;
    this.will = will;
    return this;
}
```

Builder Pattern

```
no usages  PollyBreak
public InvestigatorBuilder asset(AssetCard assetCard) {
    if (assetCard instanceof Item) {
        this.assets.getItems().add((Item) assetCard);
    }
    else if (assetCard instanceof Spell) {
        this.assets.getSpells().add((Spell) assetCard);
    }
    else if (assetCard instanceof Ally) {
        this.assets.getAllies().add((Ally) assetCard);
    }
    return this;
}

no usages  PollyBreak
public InvestigatorBuilder talent(Action talent) {
    this.talent = talent;
    return this;
}

1 usage  PollyBreak
public InvestigatorBuilder health(int health) {
    this.health=health;
    return this;
}
```

A lot of build options available

Builder Pattern

```
1 usage   PollyBreak
public InvestigatorBuilder focusLimit(int limit) {
    this.focusLimit=limit;
    return this;
}

1 usage   PollyBreak
public InvestigatorBuilder game(Game game) {
    this.game = game;
    return this;
}

1 usage   PollyBreak
public InvestigatorBuilder money(int money){
    this.money=money;
    return this;
}

1 usage   PollyBreak
public Investigator build(){
    Investigator newHero = new Investigator( investigatorBuilder: this);
    return newHero;
}
```

A lot of build options available

Builder Pattern

```
Investigator hero = new InvestigatorBuilder().name("Daniela Reyes").game(game).health(7)
        .sanity(5).skills( lore: 3, influence: 3, observation: 1, strength: 3,  will: 3).focusLimit(3)
        .money(3).build();
```

We use builder pattern in order to create investigators with any parameter customised at any step of initialization. This allows us to create in the most convenient way.

Strategy Pattern

```
1 usages  PollyBreak
public class Item extends AssetCard{
    no usages
    private int value;
    no usages
    private int hands;
    no usages
    private int health;
    no usages
    private int sanity;
    no usages
    private ArrayList<Integer> traits;

    1 usage
    private Action actionStrategy;
    3 usages  PollyBreak
    @Override
    public void use(Investigator investigator) { actionStrategy.use(investigator); }
}
```

```
1 usages  4 implementations  PollyBreak
public interface Action {
    3 usages  3 implementations  PollyBreak
    public void use(Investigator investigator);
}
```

Strategy Pattern

```java
public class ShotgunStrategy implements Action {
    1 usage
    private int damage;
    1 usage
    @Override
    public void use(Investigator investigator) {
        investigator.test(damage);
        int countExtra = 0;
        for (int i=0; i<investigator.getLastTest().size(); i++){
            if (investigator.getLastTest().get(i) == 6) {
                countExtra++;
            }
        }
        for (int i=0; i<countExtra; i++){
            investigator.getLastTest().add(6);
        }
    }
}
```

We use strategy pattern to specify different behaviors for different items.

State Pattern

```
public abstract class ActionResult {
    15 usages
    protected Investigator player;
    4 usages
    protected ActionResult nextResult;

    1 usage   4 implementations
    public abstract void act();

    4 usages
    public ActionResult(Investigator player, ActionResult nextResult) {
        this.player = player;
        this.nextResult = nextResult;
    }
}
```

State Pattern

```java
public class WardResult extends ActionResult{

    2 usages  PollyBreak +1
    @Override
    public void act() {
        int successes = player.countSuccesses();
        int count = 0;
        for (int i=0; i< successes; i++) {
            if (player.getSpace().getDoom() > 0){
                player.getSpace().setDoom(player.getSpace().getDoom() - 1);
                count++;
            }
        }
        if (count > 1) {
            player.setClues(player.getClues()+1);
        }
        player.setActionResult(nextResult);
        player.setTEMP_true_successes(count);
    }

    1 usage  PollyBreak
    public WardResult(Investigator player, ActionResult nextResult) { super(player, nextResult); }
}
```

We use state pattern to determine

Observer Pattern

```java
public interface Publisher {

    no usages   1 implementation
    public void addMonster(Subscriber subscriber);
    no usages   1 implementation
    public void removeMonster(Subscriber subscriber);
    no usages   1 implementation
    public void notifyMonsters();
```

```java
public interface Subscriber {
    1 usage   1 implementation
    public void update(Game game);
}
```

There are subscribers and publishers. Our main game class notifies its subscribers to update.

```java
    no usages
    @Override
    public void addMonster(Subscriber subscriber) { monstersSubscribers.add(subscriber); }

    no usages
    @Override
    public void removeMonster(Subscriber subscriber) { monstersSubscribers.remove(subscriber); }

    no usages
    @Override
    public void notifyMonsters() {
        for(Subscriber monster : monstersSubscribers){
            monster.update( game: this);
        }
    }
```

Observer Pattern

And here is our patrolling monster that gets notified to move to different destination every time there is a change in game class.

```
public class Patrol extends Monster implements Subscriber {
    2 usages
    private Node destination;

    1 usage
    @Override
    public void update(Game game) { this.destination = game.getUnstableSpace(); }

    1 usage
    @Override
    public void play() { move(destination); }

}
```

Factory Pattern

```java
public class MonsterFactory {
    4 usages
    public Monster createMonster (MonsterType typeGame, Game game, String name, String spawn,
                                  int damage, int horror, int health) {
        Monster monster = null;

        switch (typeGame) {
            case HUNTER:
                monster = new Hunter(game, name, spawn, damage, horror, health);
                monster = (Hunter)monster;
                ((Hunter) monster).setPlayer(game.getPlayers().get(0));
                break;
            case LURKER:
                monster = new Lurker(game, name, spawn, damage, horror, health);
                break;
            case PATROL:
                monster = new Patrol(game, name, spawn, damage, horror, health);
                break;

        }

        return monster;
    }

}
```

There is a switch so that we can manipulate with constructor args!

Factory Pattern

```
monsterFactory.createMonster(MonsterType.PATROL,game,  name: "Robbed Figure", spawn: "Independance Square",
        damage: 1,  horror: 0,  health: 1);
monsterFactory.createMonster(MonsterType.PATROL,game,  name: "Robbed Figure", spawn: "Black Cave",
        damage: 1,  horror: 0,  health: 1);
monsterFactory.createMonster(MonsterType.LURKER, game,  name: "Lurker", spawn: "Independance Square",
        damage: 2,  horror: 1,  health: 2);
monsterFactory.createMonster(MonsterType.HUNTER, game,  name: "Swift byakhee", spawn: "Independance Square",
        damage: 2,  horror: 2,  health: 3);
```

Factory classes are similar to builder pattern, but this one can create instances of different subclasses, it possess more flexibility but more concrete.

Singleton Pattern

```
private GameFacadeIAzatoth() {
}

1 usage
public static GameFacadeIAzatoth createGameFacadeIAzatoth(){
    if (gameFacadeIAzatothInstance == null){
        gameFacadeIAzatothInstance = new GameFacadeIAzatoth();
    }
    return gameFacadeIAzatothInstance;
}
```

We use singleton pattern in order to make sure that we have only one instance of an ongoing game at the same time.

Facade Pattern

```java
public class GameFacadeIAzatoth implements GameFacadeInterface{
    3 usages
    private static GameFacadeIAzatoth gameFacadeIAzatothInstance;
    19 usages
    private Game game;
    4 usages
    private MonsterFactory monsterFactory= new MonsterFactory();
    1 usage  PollyBreak +1
    @Override
    public void startGame() {
        game = new Game();
        BoardBuilder boarder = new BoardBuilder();
        game.setBoard(boarder.build( scenario: "Azatoth"));
        game.setPlayers(new ArrayList<>());
        game.setCurrentPhase(Phases.ACTION_PHASE);
        MonsterPhaseLogic monsterPhaseLogic = new MonsterPhaseLogic();
        game.setMonsterPhaseLogic(monsterPhaseLogic);
        Investigator mainHero = new InvestigatorBuilder().name("Daniela Reyes").game(game).health(7)
                .sanity(5).skills( lore: 3, influence: 3, observation: 1, strength: 3,  will: 3).focusLimit(3)
                .money(3).startSpace(game.getBoard().fetchNode( name: "Arkham Advertiser")).build();
```

We set up our game step by step...

Facade Pattern

...step by step.

```
        Item shortgun = new Item(new ShotgunStrategy( damage: 3));
        mainHero.getAssets().getItems().add(shortgun);

    game.getPlayers().add(mainHero);
    game.getBoard().placePlayer(mainHero.getSpace().getName(),mainHero);

    monsterFactory.createMonster(MonsterType.PATROL,game,  name: "Robed Figure", spawn: "Independence Squa
            damage: 1,  horror: 0,  health: 1);
    monsterFactory.createMonster(MonsterType.PATROL,game,  name: "Robed Figure", spawn: "Black Cave",
            damage: 1,  horror: 0,  health: 1);
    monsterFactory.createMonster(MonsterType.LURKER, game,  name: "High Priest", spawn: "Independence Squa
            damage: 2,  horror: 1,  health: 2);
    monsterFactory.createMonster(MonsterType.HUNTER, game,  name: "Swift Byakhee", spawn: "Independence S
            damage: 2,  horror: 2,  health: 3);

    game.subscribeMonsters(game.getMonsters().toArray(new Monster[0]));
    game.setUnstableSpace(game.getBoard().fetchNode( name: "Arkham Advertiser"));
    game.notifyMonsters();
}
```

Facade Pattern

Facades are used to quickly setup a large thing if there are different realisations on the same request.

```
1 usage   1 implementation   PollyBreak
public interface GameFacadeInterface {
    1 usage   1 implementation   PollyBreak
    public void startGame();
}
```

Interface

Arkham Horror!

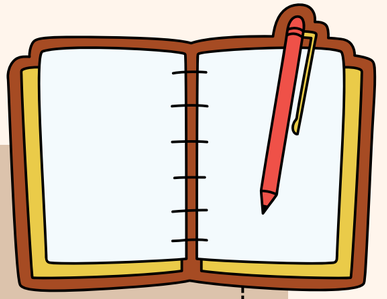
Arkham Advertiser
Curiositie Shoppe
Train Station
Independence Square
Arkham Asylum
La Bella Luna
Velma's Dinner
Police Station
Hibb's Roadhouse
Unvisited Island
River Docks
Tick-Tock Club
Black Cave
General Store
Graveyard

Ward
Gather Resources
Finish test

Name - Daniela Reyes
Health - 7
Sanity - 5
Money - 2

You are about to hit Robed Figure
Inventory:

Conclusion



We created the working release of the Arkham Horror board game with not all mechanics.

We implemented a lot of software design patterns:

1. **The Builder pattern - for a creation of an investigator.**
2. **The Singleton pattern - to have only one game facade.**
3. **The Strategy pattern - to allow items have different behavior.**
4. **The Observer pattern - to notify monsters about unstable zone.**
5. **The Façade pattern - to manipulate with the Game class.**
6. **The State pattern - to set different result of different hero’s actions.**
7. **The Factory method - for creating different monsters;**

Thanks for Attention!