# **Week 4 - Homework 3: Advanced Creational Patterns – Builder & Prototype**

## **Overview**

This document provides documentation for **Assignment 1** and **Assignment 2** of Week 4 Homework, which involve implementing the **Builder** and **Prototype** design patterns in Java. The implementation focuses on constructing complex objects and efficiently replicating them when needed.

## **Assignment 1: Implementing a Builder for a Complex Object**

### **Objective**

The goal is to create a **Dungeon Builder** that facilitates the step-by-step creation of a **Dungeon** object with rooms and NPCs.

### **Classes and Interfaces**

#### **Dungeon.java (The Product Class)**

Represents a **Dungeon**, which contains:

* name: A string representing the dungeon's name.
* rooms: A list of **Room** objects.
* npcs: A list of **NPC** objects.

**Key Methods:**

* showDungeon(): Displays dungeon details.
* Getter and Setter methods for name, rooms, and npcs.

package Assignment1;

import java.util.List;

class Dungeon {

private String name;

private List<Room> rooms;

private List<NPC> npcs;

public Dungeon(String name, List<Room> rooms, List<NPC> npcs) {

this.name = name;

this.rooms = rooms;

this.npcs = npcs;

}

public Dungeon() {

}

public void showDungeon() {

System.*out*.println("Assignment1.Dungeon: " + name);

System.*out*.println("Rooms: " + rooms.size());

System.*out*.println("NPCs: " + npcs.size());

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public List<Room> getRooms() {

return rooms;

}

public void setRooms(List<Room> rooms) {

this.rooms = rooms;

}

public List<NPC> getNpcs() {

return npcs;

}

public void setNpcs(List<NPC> npcs) {

this.npcs = npcs;

}

}

#### **IDungeonBuilder.java (Builder Interface)**

Defines the builder methods for constructing a Dungeon object step by step.

**Key Methods:**

* setDungeonName(String name): Sets the dungeon name.
* addRoom(Room room): Adds a **Room** object.
* addNPC(NPC npc): Adds an **NPC** object.
* build(): Returns the final **Dungeon** object.

package Assignment1;

interface IDungeonBuilder {

IDungeonBuilder setDungeonName(String name);

IDungeonBuilder addRoom(Room room);

IDungeonBuilder addNPC(NPC npc);

Dungeon build();

}

#### **SimpleDungeonBuilder.java (Concrete Builder)**

Implements IDungeonBuilder to construct a **Dungeon** step by step.

**Key Features:**

* Implements methods to set name, add rooms, and NPCs.
* Uses method chaining for a fluent interface.
* Returns a fully constructed Dungeon object.

package Assignment1;

import java.util.ArrayList;

import java.util.List;

class SimpleDungeonBuilder implements IDungeonBuilder {

private String name;

private List<Room> rooms = new ArrayList<>();

private List<NPC> npcs = new ArrayList<>();

public IDungeonBuilder setDungeonName(String name) {

this.name = name;

return this;

}

public IDungeonBuilder addRoom(Room room) {

rooms.add(room);

return this;

}

public IDungeonBuilder addNPC(NPC npc) {

npcs.add(npc);

return this;

}

public Dungeon build() {

return new Dungeon(name, rooms, npcs);

}

}

#### **Room.java**

Represents a room inside the dungeon.

package Assignment1;

class Room {

private String description;

public Room(String description) {

this.description = description;

}

}

#### **NPC.java**

Represents a non-player character inside the dungeon.

package Assignment1;

class NPC {

private String name;

private String description;

private double health;

public NPC(String name, String description, double health) {

this.name = name;

this.description = description;

this.health = health;

}

}

#### **MUDBuilderDemo.java (Demonstration)**

Demonstrates how to use the **SimpleDungeonBuilder** to construct a dungeon.

package Assignment1;

public class MUDBuilderDemo {

public static void main(String[] args) {

Dungeon dungeon = new SimpleDungeonBuilder()

.setDungeonName("Dark Cave")

.addRoom(new Room("Entrance Hall"))

.addRoom(new Room("Treasure Chamber"))

.addNPC(new NPC("Goblin King", "Boss", 1000))

.build();

dungeon.showDungeon();

}

}

## **Assignment 2: Implementing a Prototype for Cloning Objects**

### **Objective**

The goal is to implement the **Prototype Pattern** to enable cloning of entities like **NPC** and **Room**.

### **Classes and Interfaces**

#### **CloneableGameEntity.java (Prototype Interface)**

Defines the prototype method for cloning objects.

package Assignment2;

class Room {

private String description;

public Room(String description) {

this.description = description;

}

public Room cloneEntity() {

return new Room(this.description);

}

}

#### **NPC.java (Prototype Implementation)**

Represents a **Non-Player Character (NPC)** that implements cloning.

package Assignment2;

class NPC implements Cloneable {

private String name;

private String description;

private double health;

public NPC(String name, String description, double health) {

this.name = name;

this.description = description;

this.health = health;

}

public NPC cloneEntity() {

return new NPC(this.name, this.description, this.health);

}

public String getName() {

return name;

}

public void setName(String name) {

this.name = name;

}

public String getDescription() {

return description;

}

public void setDescription(String description) {

this.description = description;

}

public double getHealth() {

return health;

}

public void setHealth(double health) {

this.health = health;

}

}

#### **Room.java (Prototype Implementation)**

Represents a **Room** that supports cloning.

package Assignment2;

class Room {

private String description;

public Room(String description) {

this.description = description;

}

public Room cloneEntity() {

return new Room(this.description);

}

}

#### **MUDPrototypeDemo.java (Demonstration)**

Demonstrates how to use the **Prototype Pattern** to create and modify cloned NPC objects.

package Assignment2;

public class MUDPrototypeDemo {

public static void main(String[] args) {

NPC prototypeNPC = new NPC("Orc Warrior", "Boss", 100);

NPC clonedNPC1 = prototypeNPC.cloneEntity();

NPC clonedNPC2 = prototypeNPC.cloneEntity();

clonedNPC1.setHealth(clonedNPC1.getHealth()-50); *// Уменьшаем здоровье одного клона*

System.*out*.println("Original NPC: " + prototypeNPC.getName());

System.*out*.println("Clone 1 (damaged): " + clonedNPC1.getHealth());

System.*out*.println("Clone 2 (normal): " + clonedNPC2.getHealth());

}

}

## **Conclusion**

This documentation now covers both **Assignment 1 (Builder Pattern)** and **Assignment 2 (Prototype Pattern)**. Future improvements may include:

* Implementing **deep cloning** for complex objects with nested lists.
* Combining both patterns to allow **template-based cloning of structured objects**.