POLYTECHNIQUE MONTRÉAL

LOG8430E - Software Architecture and Advanced Design

Travail pratique 1 (TP1)

Professor: Zohreh Sharafi

Group Name: Atwater

Team members:

Student ID	Name
2313663	Xi-Zhen Wang
2078987	Thomas Lusignan
2317239	Roberto Molina

Montréal, Quebec
2023

Contents

Question 1: Software Architecture Analysis (20 points)

Architecture of Pixel Dungeon

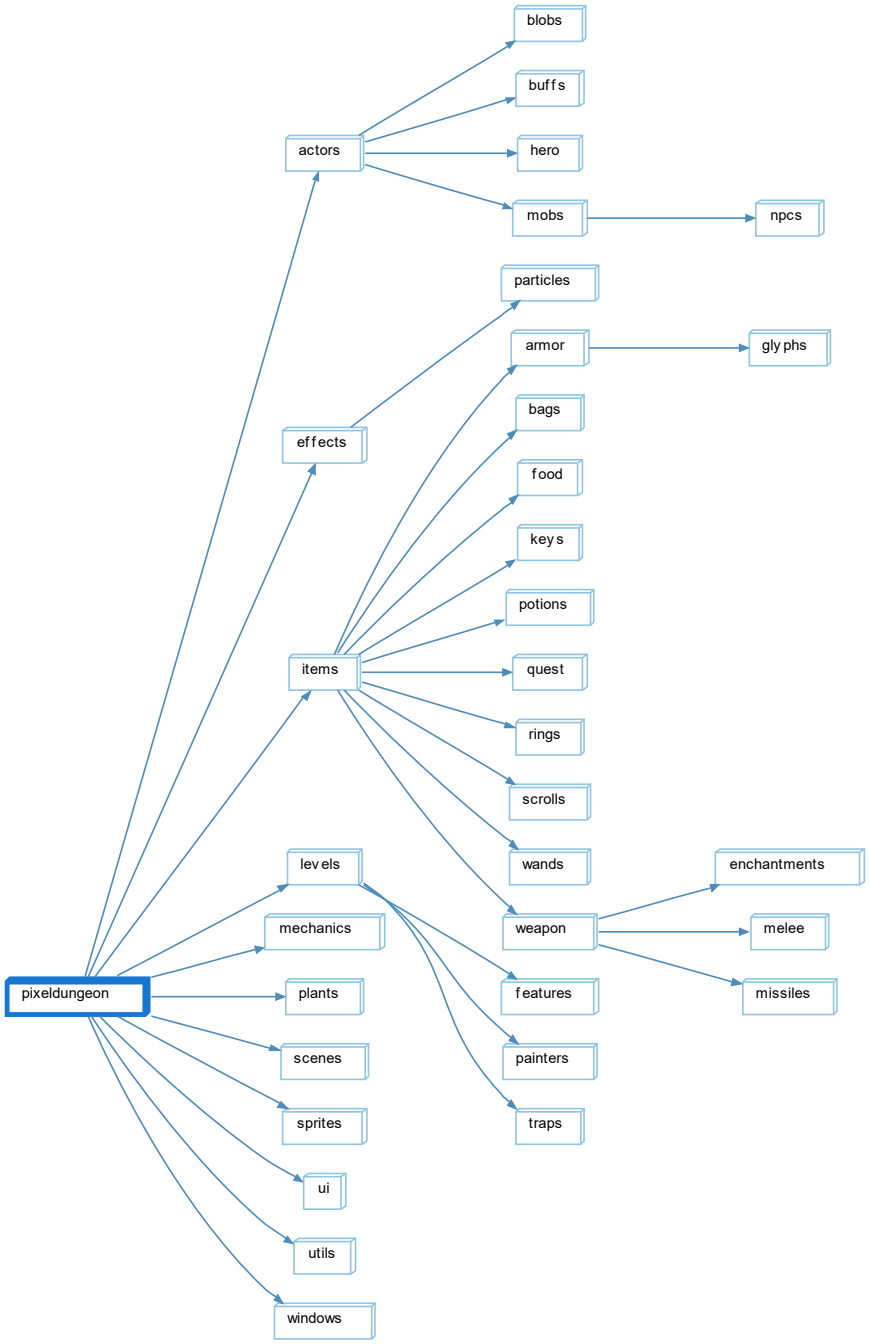


Figure 1: Pixeldungeon src module package diagram extracted by Understand

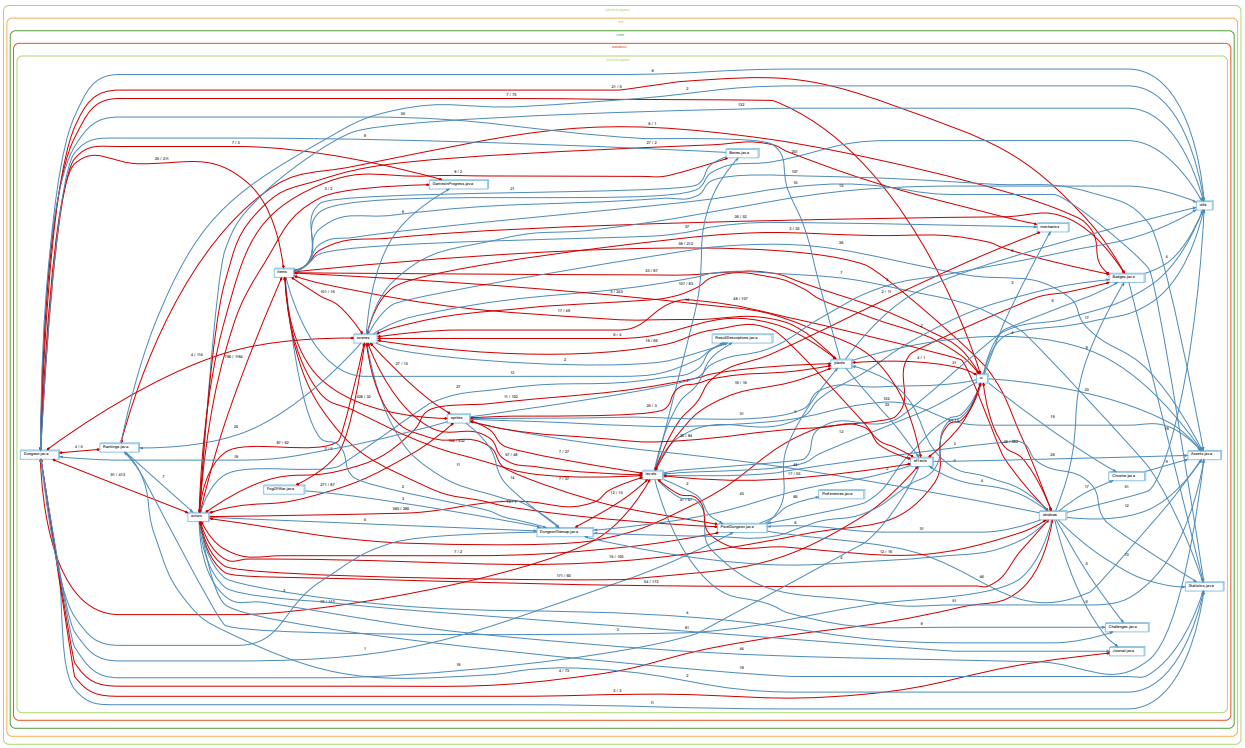


Figure 2 : Pixel Dungeon dependency diagram extracted using Understand. The red arrows represent mutual dependencies, and the blue arrows represent one-way dependencies.

Architectural Style

As explained during the class, there are three main categories of architecture styles: monolithic, modular, and distributed architectures. Each have their defining characteristics.

First, the distributed style is characterized by having different modules running on multiple machines, connected through the network. This is evidently not the case for Pixel Dungeon, as the source code is contained in a single directory and compiles to a single application. We can therefore exclude this category of styles.

Therefore, we must figure out whether Pixel Dungeon is a monolithic or a modular architecture. This is a harder task because it has some characteristics of both styles. For instance, a monolithic architecture is usually developed by a small team of experts with a clear ownership. Pixel Dungeon was mainly developed by one person who goes by the pseudonym “watabou”. However, it is also an open-source project, which means anyone can participate to its development, and its ownership is therefore also open.

The main difference between monolithic and modular architectures is their organization. A monolith is a single program with little to no distinction between elements like UI, logic, and data, and keeps the logic and data in a single place. On the other hand, a modular architecture separates the code in independent modules that interact with each other, that can be developed and compiled separately. In this regard, Pixel Dungeon leans more towards a monolithic architecture, albeit a very well organized one. As discussed in the next section, many classes have the responsibility to both hold the data about a component and handle the logic surrounding it. Most modular architectures for a program of this type (variants of layered or MVC architectures, since neither pipe-filter nor blackboard architectures would be appropriate here) would keep these responsibilities in separate, well-defined layers. All in all, a monolithic architecture is the model that fits Pixel Dungeon the best.

Role of Each Module

Pixel Dungeon is organized as a series of directories, each containing a category of objects in the game such as levels, characters, items used by characters, or visual effects. The code is object-oriented, and as such the modules are mainly categories of abstractions and their derived classes.

Members of each category interacts regularly with objects in other categories, as demonstrated by the numerous arrows in Figure 2. Although there is no clear overarching layer separation, such as data, logic, and presentation layers, each class has specific responsibilities, which we will discuss here, in the order in which they are presented in Figure 1.

Pixeldungeon

Pixeldungeon is the base module that contains all the others. It also contains the base class that starts the game (PixelDungeon) as well as the infrastructure of the game, such as the dungeon and links to the asset files.

Actors

In Pixel Dungeon, actors are any object that can move on the map, such as the player's character (hero), their enemies (mobs), effects that can affect them (buffs), or areas with a special effect on other actors, such as fire (blobs). Their logic and data are contained in their classes.

Effects

Effects are the logic behind the visual animations of special effects in the game, such as spells or fire. They routinely interact with Sprites to create the visible effects on screen.

Items

Items are things the player can use, attack with, find, wear, or consume as they interact with the game. The classes in this directory are responsible for the logic and data about these objects.

Levels

Levels contain the logic for the creation as well as the data of each procedurally-generated level of the game's dungeon. It also contains Painters to create the visual elements of each level.

Mechanics

Mechanics are special objects that contain the logic for some game mechanics such as projectiles and shadows.

Plants

Plants are a category of objects placed around the levels with which the player can interact in multiple ways (harvesting, planting, eating, etc.).

Scenes

The scenes are sequences that happen throughout the game to help communicate the story to the player. These classes comprise both the data about the scene content and the logic of how they are displayed.

Sprites

Sprites control how all item animations in the game are displayed. They are the link between the code and the assets.

Ui

The ui directory contains the building blocks of the user interface of the game: menus, buttons, etc. They are the components that fill windows. The classes control both the display and the logic of the components.

Utils

Similarly to Mechanics, utils are a collection of utility methods used across the code to process strings, save a game log, etc.

Windows

Windows are the different screens of the game, from the main menu to the in-game screen. Windows contains various ui components with which the player can interact. They act as a view in the sense of an MVC architecture, but also implement the logic behind the actions triggered by their components.

Question 2: Design Patterns (30 points)

In this question we were asked to look for three different design patterns in Pixel Dungeon, we searched for them using the IDE intellij and the software program Understand.

It was a complicated task as the application does not apply 100% correct the patterns found, so for the patterns that were not 100% implemented we explained what parts were missing.

We found two creational patterns that are singleton and factory method and one behavioral that is the template pattern.

Singleton

The first design pattern we identified is Singleton:

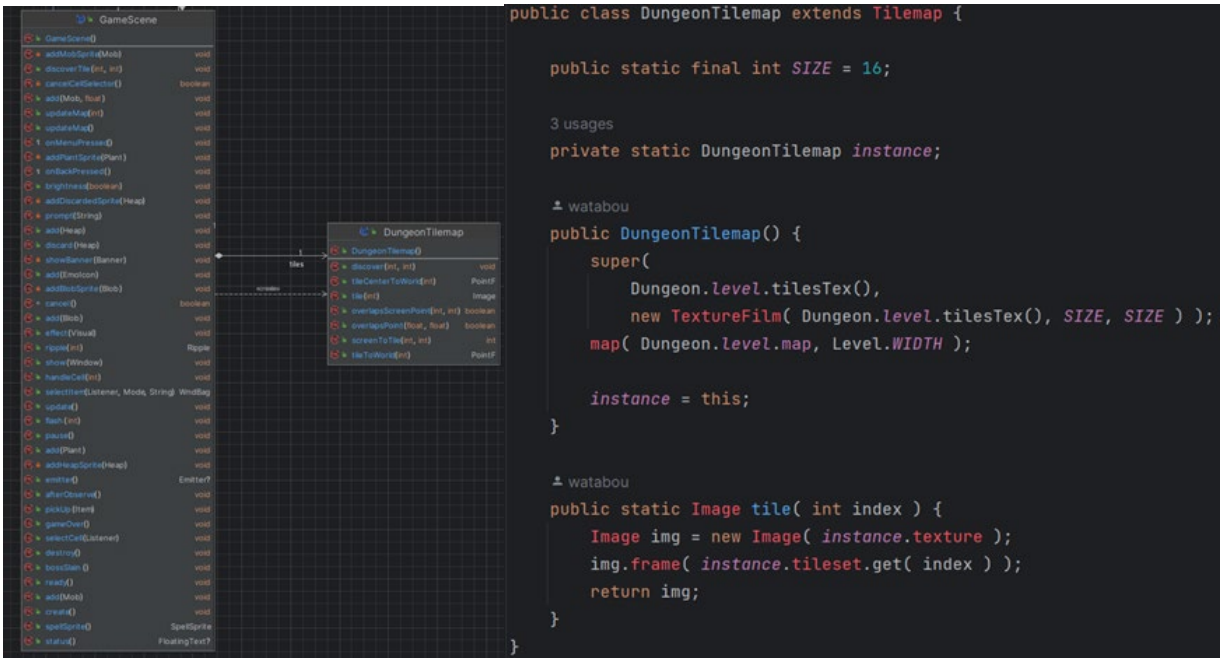


Figure 3. UML of DungeonTilemap and Its relations and Code of DungeonTilemap (some code and classes were omitted because they were not relevant)

Singleton design pattern is used when we want to create only one instance of a class and it provides a global point to access this instance.

The pattern is used in this case to create, represent and manipulate the tile map of a game scene of Pixel Dungeon. To do this DungeonTilemap has only one instance that represents this tile map.

This class follows the pattern as it only has one instance and it is declared private and static, also it has one global access (the function tile), where other classes can access a part of the instance.

The problem is that it does not apply 100% of the pattern because the constructor should be private. Also, the instance should be created when tile is called and there is not an instance rather than be created outside of the class (in this case in game scene as is shown in the UML).

There are two classes involved one is DungeonTilemap that it contains the instance and creates it and the second one is GameScene that is the one that calls the constructor, this class should not be involved in the singleton as the instance should be created by the global access point (tile).

Template

The second design pattern we observed is the template:

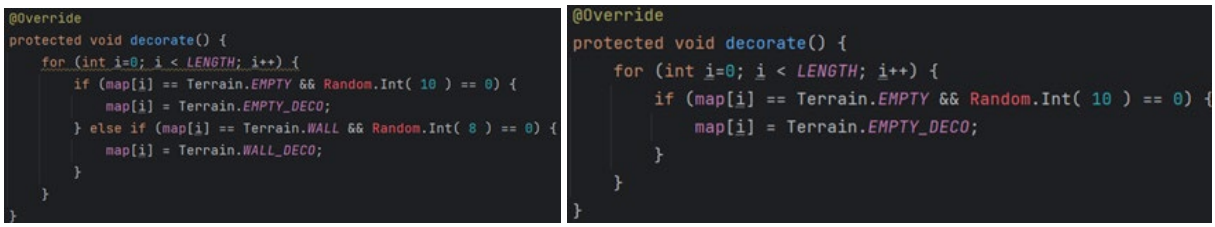
Figure 4. ULM of level and the different subclasses it has (some classes were omitted because they were not relevant)

```
public void create() {
    resizingNeeded = false;
    map = new int[LENGTH];
    //...

    if (!Dungeon.bossLevel()) {
        addItemToSpawn( Generator.random( Generator.Category.FOOD ) );
        if (Dungeon.posNeeded()) {
            addItemToSpawn( new PotionOfStrength() );
            Dungeon.potionOfStrength++;
        }
        //...

    }
    boolean pitNeeded = Dungeon.depth > 1 && weakFloorCreated;
    do {
        //...
    } while (!build());

    decorate();
    buildFlagMaps();
    cleanWalls();
    createMobs();
    createItems();
}
```

Specific part

```
abstract protected boolean build();
13 implementations  watabou
abstract protected void decorate();
11 implementations  watabou
abstract protected void createMobs();
11 implementations  watabou
abstract protected void createItems();
```

Figure 5. Code of the algorithm create in the class level and the different abstract methods it has for the subclasses to implement (some code was omitted because it was not relevant)



```
@Override
protected void decorate() {
    for (int i=0; i < LENGTH; i++) {
        if (map[i] == Terrain.EMPTY && Random.Int( 10 ) == 0) {
            map[i] = Terrain.EMPTY_DECO;
        } else if (map[i] == Terrain.WALL && Random.Int( 8 ) == 0) {
            map[i] = Terrain.WALL_DECO;
        }
    }
}
```

```
@Override
protected void decorate() {
    for (int i=0; i < LENGTH; i++) {
        if (map[i] == Terrain.EMPTY && Random.Int( 10 ) == 0) {
            map[i] = Terrain.EMPTY_DECO;
        }
    }
}
```

Figure 6. Implementation of decorate in deadendlevel and in lastlevel (some code was omitted because it was not relevant)

The Template Method pattern is applied when a general algorithm is defined in a superclass, and subclasses are responsible for overriding specific steps of that algorithm. In the context of this code, the Level class implements an algorithm to create game levels. While the creation of levels follows a common skeleton, certain aspects vary depending on the specific type of level.

So, in this case the pattern helps in the creation of the different types of levels where all levels have a global skeleton but each of them has its own specific parts.

In this pattern, multiple classes are involved. The Level class serves as the base class, containing the overarching logic of the create algorithm. Additionally, there are various subclasses (like deadend), each responsible for implementing specific functions of the create algorithm, such as decorate, cleanWalls, createMobs, and createItems. These subclasses provide custom implementations of these functions, allowing for diverse and specialized level creation.

Factory method

```
public class Emitter extends Group {
    protected Factory factory;

    public void start( Factory factory, float interval, int quantity ) {
        this.factory = factory;
    }
    1 override
    protected void emit( int index ) {
        if (target == null) {
            factory.emit(
                emitter: this,
                index,
                x: x + Random.Float( width ),
                y: y + Random.Float( height ) );
        } else {
            factory.emit(
                emitter: this,
                index,
                x: target.x + Random.Float( target.width ),
                y: target.y + Random.Float( target.height ) );
        }
    }
    36 inheritors
    abstract public static class Factory {
        36 implementations
        abstract public void emit( Emitter emitter, int index, float x, float y );
        //...
    }
}
```

```
public class ShaftParticle extends PixelParticle {

    watabou
    public static final Emitter.Factory FACTORY = new Factory() {
        watabou
        @Override
        public void emit( Emitter emitter, int index, float x, float y ) {
            ((ShaftParticle)emitter.recycle( ShaftParticle.class )).reset( x, y );
        }
        no usages  watabou
        @Override
        public boolean lightMode() { return true; }
    };
```

Figure 7. Implementation of the class emitter and implementation of the shaft particle (some code was omitted because it was not relevant)

The Factory Pattern is employed to define an interface for creating objects in a superclass, allowing subclasses to determine the specific type of objects to create. In this context, the

Emitter.Factory serves as the factory interface, enabling the recycling of various particle types, with each particle type acting as a subclass that can be recycled.

This pattern is utilized to efficiently recycle and configure different types of particles within the system. However, it's important to note that in this implementation, new particles are not created but rather recycled and reset so the pattern is not 100% implemented.

The various classes involved in this pattern include Emitter, which initiates particle emission and utilizes the factory, Emitter.Factory, the interface that defines the abstract emit() method implemented by different particle types, and the individual particle classes (like shaft particle), each of which implements the emit() method to recycle and emit particles of their specific type.

Question 3: SOLID Design Principles (30 points)

Single Responsibility Principle

We can easily find there are many designs that comply with the Single Responsibility Principle (SRP) in Pixel Dungeon. SRP defined that a class should have only one reason to change, which means it should have only one responsibility or job.

Take `Toolbar` as an example. It contains three different components: `Tool` creates buttons, `PickedUpItem` is responsible for animation, and `GoldIndicator` displays in-game currency information. `Toolbar` itself uses the values returned by those classes to create the user interface (UI) elements of the in-game toolbar. It is obvious that all classes are separated appropriately and have a clear and distinct purpose.

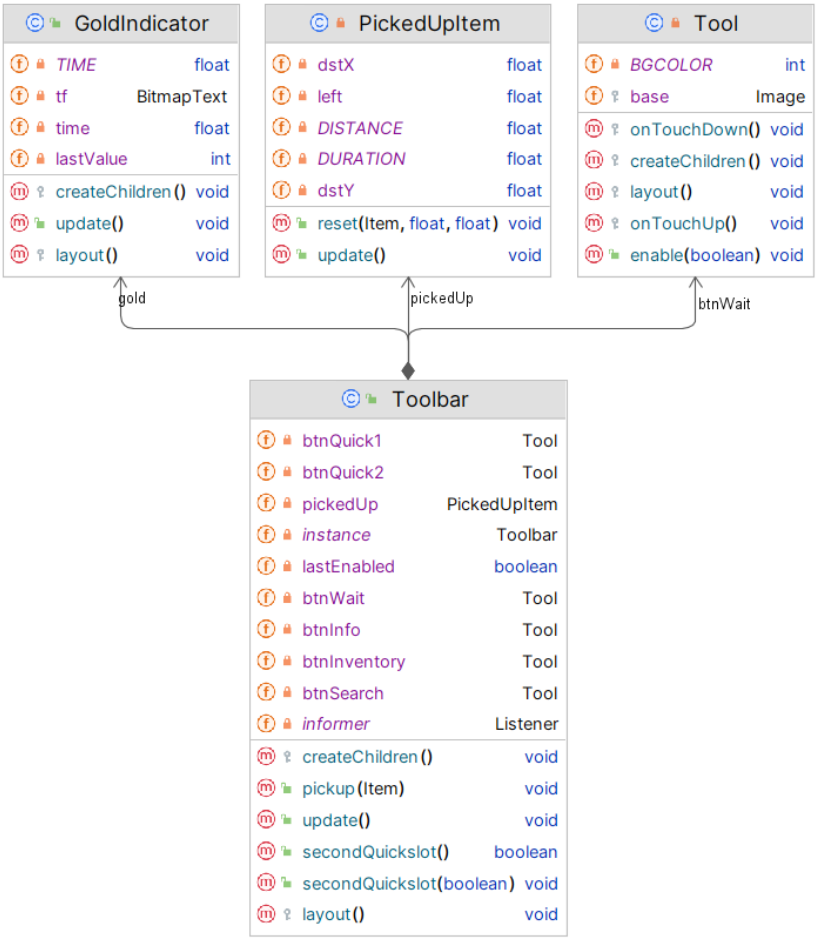


Figure 8. UML diagram for Single Responsibility Principle found at `Toolbar` file

Let me briefly describe the roles each object is responsible for:

- ***Toolbar***: Provides the toolbar user interface (UI) in the game and manages the layout and appearance of toolbar elements.
- ***Tool***: The base class for various buttons in the toolbar. It creates buttons, handles touch events for buttons, and decides whether the button's state is enabled or disabled.
- ***PickedUpItem***: Responsible for animating and displaying when item is picked up by the player.
- ***GoldIndicator***: Display the player's in-game currency (gold) in the toolbar and update the gold

Open-Closed Principle

One concrete example of a SOLID principle we can find in Pixel Dungeon is the Open-Closed Principle (OCP). OCP suggests that software entities, such as classes or modules, should be open for extension but closed for modification.

According to the UML Diagram below, there is one interface `AiState` and the other five classes implement it. Therefore, we can add a new class to create a new state by implementing `AiState` without modifying `AiState` itself.

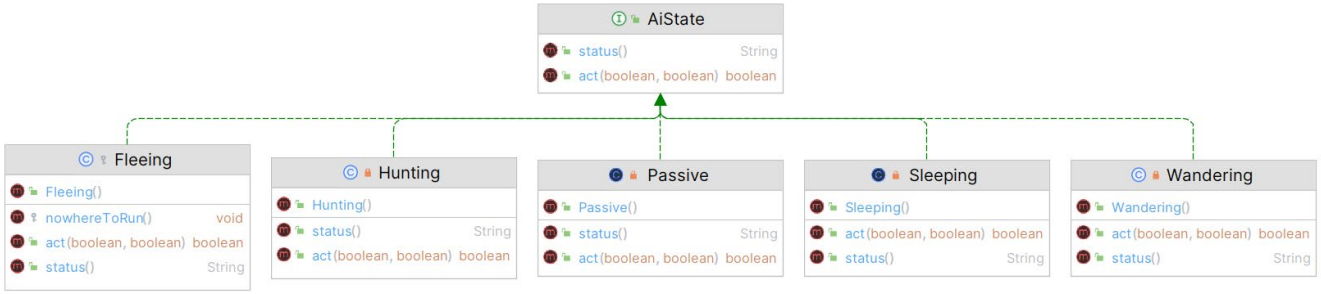


Figure 9. UML diagram for Open-Closed Principle found at `AiState` Class

Let me explain the roles and responsibilities of each class:

- ***AiState***: This is an interface that defines different states in the system. It defines two methods:
    - `act` is responsible for taking action based on the enemy's position and whether to get alert.
    - `status` returns a string describing the current state.

The following classes all override methods in the `AiState` interface:

- ***Sleeping*** (implement `AiState`): Represents the entity's state as sleeping. If an enemy is in the field of view and certain conditions are met, the state changes to "Hunting". Otherwise, it remains in the "Sleeping" state and starts to tick time.

- ***Wandering*** (implement `AiState`): Represents the entity's state as wandering. If an enemy is detected, the state changes to "Hunting". Otherwise, it will move to a random destination.
- ***Hunting*** (implement `AiState`): Represents the entity's state as hunting. It checks it can attack the enemy and takes some actions. If the enemy is not in the field of view, the state changes to the "Wandering".
- ***Fleeing*** (implement `AiState`): Represents the entity's state as fleeing. It will try to flee from the enemy if the enemy is in the field of view. Otherwise, it will start timing and trigger `nowhereToRun` method.
- ***Passive*** (implement `AiState`): Represents the entity's state as passive, where the entity does nothing but spends time.

Dependency Inversion Principle

Another principle we also find in `AiState` is the Dependency Inversion Principle (DIP). DIP states that high-level modules should not import anything from low-level modules, but both should depend on abstractions (e.g., interfaces).

According to the UML Diagram below, we notice that the higher-level `Mob` class does not depend on specific lower-level classes such as `Hunting` but depends on an intermediate interface `AiState`.

Figure 10. UML diagram for Dependency Inversion Principle found at `Mob` file

Let me supply Mob's role and responsibility:

- ***Mob***: Represents the behavior of game mobs based on their current state and conditions, which includes functionality for movement, combat, status effects, etc.

Question 4: Violation of SOLID Design Principles (20 points)

Violation of SOLID principle

Code in `Item` Class and other Classes that inherit from `Item` violates **Liskov Substitution Principle (LSP)**. LSP states that objects of a derived class should be able to replace objects of the base class without affecting the correctness of the program.

UML diagram and code snippets

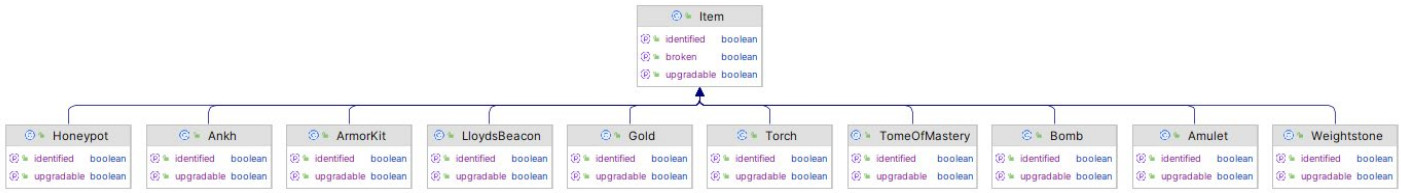


Figure 11. UML diagram for Liskov Substitution Principle violation

As you can see, there are multiple classes extending from `Item` and all of them override the `isUpgradable` and `identified` method.

Take the `Gold` as an example, the problem occurs here: the sub-class (`Gold`) overrides the `isUpgradable` method to return false, while the same method in the base class (`Item`) returns true. In this case, the `Gold` class is altering the behavior defined in the base class.

```
public class Item implements Bundlable {
    public boolean isUpgradable() {
        return true;
    }
};
```

```
public class Gold extends Item {
    @Override
    public boolean isUpgradable() {
        return false;
    }
};
```

Figure 12. Code snippets for `isUpgradable` methods from the `Item` and `Gold` class

Justification

This behavior violates the Liskov Substitution Principle because LSP requires that subclasses should be substitutable for their base class without changing the behavior or expectations of the code using the base class.

According to Design by Contract, complying with LSP means complying with the following three conditions:

1. Preconditions cannot be strengthened in the subtype.
2. Postconditions cannot be weakened in the subtype.
3. Invariant cannot be weakened in the subtype.

In this case, the post-condition is clearly violated. In the base class (`Item`), the code inherits from `Item` class expects that `isUpgradable` should return true for all items, indicating that items are generally upgradable.

Violating LSP will have a significant negative impact on both maintainability and understandability of the program. First, it can make writing unit tests difficult. Subclasses that don't adhere to the expected behavior of the base class can lead to unexpected test failures and make it harder to ensure the correctness of the code. In addition, it will cause inconsistent behavior between base class and subclasses, which may confuse developers and make it challenging to understand how different parts of the program interact, ultimately leading to unpredictable results.

How to fix the violation

In order to comply with LSP, we need to ensure that derived classes do not alter the behavior which defined in the base class. If gold items are inherently non-upgradable, it would be more appropriate to handle this in a composition way, rather than inheriting from the base class.

Therefore, we create the `Upgradability` class that contains the `upgradable` method, which always returns true, and the `nonUpgradable` method, which returns false.

In the `Item` class, we add a private `Upgradability` field named `upgradability` and delegate the `upgradable` method to it. Other subclasses inherited from `Item`, such as `Gold`, there is also a private `Upgradability` field and delegates the `nonUpgradable` method to it.

By doing this, the subclass can fully follow the behavior of the base class without having to override methods.

```
public class Upgradability {
    boolean upgradable() {
        return true;
    }

    boolean nonUpgradable() {
        return false;
    }
}
```

```java
public class Item implements Bundlable {
    private Upgradability upgradability = new Upgradability();

    public boolean isUpgradable() {
        return upgradability.upgradable();
    }
}
```

```java
public class Gold extends Item {
    private Upgradability upgradability = new Upgradability();

    public boolean isUpgradable() {
        return upgradability.nonUpgradable();
    }
}
```

Figure 13. Code snippets for `Upgradability` class is used in the `Item` and `Gold` class

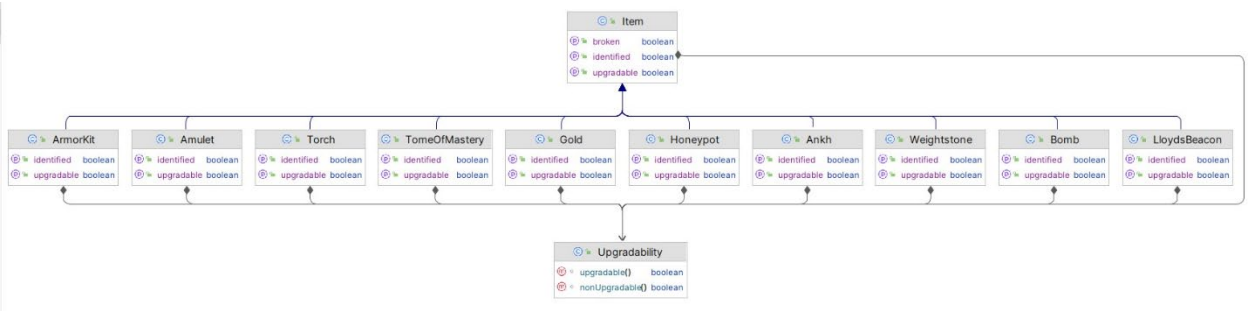


Figure 14. UML diagram for Liskov Substitution Principle violation refactoring

This approach increases the flexibility and readability of the program by using the composition instead of inheritance way, maintaining the Liskov Substitution Principle.

References

Shvets, A. (2023). Factory Method. Refactoring Guru https://refactoring.guru/design-patterns/factory-method

Shvets, A. (2023). Singleton. Refactoring Guru https://refactoring.guru/design-patterns/singleton

Shvets, A. (2023). Template. Refactoring Guru https://refactoring.guru/design-patterns/template-method