TDA367 Peer Review of Bestprogrammers_eu

By Group C7

Code Reusability, Open-closed, and Design Principles

Most of the code is single-use only and created on a case-by-case basis. However, model contains some elements of reuse. One example of this is how an Attack's effect is created by stringing *IEffects* together. This makes new Attacks easy to implement due to their effect being a reuse of *IEffects*.

The project attempts to use the factory pattern in multiple places but lacks one critical part of the pattern. The *Attack*- and *ItemFactory* returns a concrete class, *Attack* and *Item* respectively, instead of an interface. Both classes are thus not encapsulated since they have public access modifiers. A factory is supposed to hide the internal parts of a package, but if every class in the package is public the factory loses some of its meaning. Likewise, if the dependency the factory returns is concrete instead of an interface, the code is not as open to modification as it could be if the factory returned an interface.

The observer pattern is also used in this project in the *EffectAnimationHandler* inside the view package. This class notifies any listener that an effect should be displayed. However, the class is problematic in other ways which will be discussed in the MVC portion.

However, one relatively modular portion of the application is the controller. The controller functions as an aggregate with its root in the *InputController*. Every sub-controller implements the *IController* interface. Because of this use of an interface, one could easily add a new controller to the aggregate root.

Documentation, Readability, and Consistency

The code lacks almost any documentation or comments, except for a few TODO:s and a comment or exceptionally rare java doc here and there. Notably, the few comments that do exists are, however, inconsistent in its usage of language; some are in Swedish (e.g. in IEffect) and most are in English. The lack of comments affects the readability of the code since some classes, such as the *alterCurrentStats* method in the class *Puckemon* (see figure 3, appendix). The readability is even worse in classes containing magic numbers, such as the *EffectHelper's typeChart*. This 2d double array filled of magic constants and has no description for what its purpose is.

Class names are generally sufficiently intuitive to make their purpose comprehensible despite the lack of documentation, but there are a few examples of poor naming, such as the implementation of the graphics library's *Game* being named "*Boot"* and the player's opponent in the *Combat* class being simply named "*fighter*".

The consistency of the code in the project could be improved. Some classes have package-private modifiers for no apparent reasons while others have only private modifiers. Some methods use boxed Booleans while others use primitives, and again for no apparent reason. To see an example of this one could compare *HealAmount* and *ModifyAttackPower* which reside in the same package. Similarly, some interfaces have explicit public methods (e.g. *IFighter*) while other interfaces have implicit public methods (e.g. *ITrainer*).

MVC Structure

The most obvious violation of the MVC pattern is that the model package has dependencies on the view package. The *IEffects* in the *effects.effectTypes* package uses the singleton *EffectAnimationsHandler* for telling the view when to display an effect (see figure 1, appendix). Secondly, in the *ListItem* class in the model uses the imports images and stores them. This implies that the model is not independent from the view. All of this implies that the model has a hard dependency on the graphics library and view. In addition to this, the *PartyScreen* view class modifies the model by switching *Puckemons* in the *switchPuckemon* method. Preferably the view should not mutate the model and just observe it.

In the view class *InvertoryView* model data (inventory items) is saved in instance variables. Optimally one would not save any model data in any view class.

The controller follows the MVC principles for the most part. It is separated form the view and model and has a clear responsibility of sending input and data to the model.

Dependencies and Abstractions

The code has circular dependencies. For example: the *Boot* class depends on the *InputController* and the *InputController* depends on the *Boot* class via the *setController* and *setView* methods. However, there are many more on the package level, and these can be seen in figure 1 and 2 in the appendix.

The concept of depending on abstraction rather than concrete implementation could be followed better. While the interface List is used in favor of ArrayList in most places and most instance variables and return types are interfaces. The project lacks interfaces for some relatively large components such as the class Attack or the Item class. Because of this concrete dependencies are used instead of dependencies on interfaces.

How is the performance?

There does not seem to be any performance issues with the application. However, the performance and stability of the application could be improved by only loading the excel file in *ExcelReader* once and then caching it instead of loading the excel file on every method call to the class. It is best to minimize IO operation during runtime.

Tests

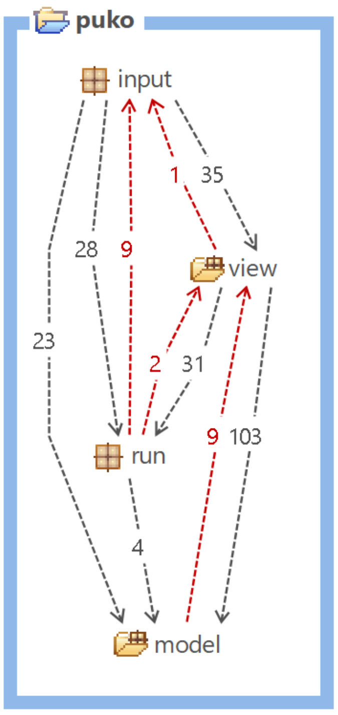
The codebase has a low test-coverage in the model. Only the *ExcelReader* and *MonRegisterInterperator* have tests. Preferably all of the model would have tests so that bugs such as the "*GoldenNuggie*" *Item* crashing the application when used.

Improvements

Here is a list of improvements in no particular order which would increase the quality of the code.

- Use interfaces more so that concrete dependencies are minimized, especially for factories since that is a core part of the design pattern.
- Use an observer pattern or some other form of event bus the view can tap into instead of *EffectAnimationsHandler* for notifying the view of effects from the model. This removes the model's dependency on the view.
- Remove code which modifies the model from view. This makes the model independent.
- Use package-private modifiers for classes which should not be accessed outside a package. This modularizes packages more and forces customers to not depend on internal parts of the package.
- Document and use functional decompositioning more int code. This improves its readability, and thus its maintainability.
- Move IO operations, such as the excel reader, outside the model and into a service or utility package. The model should also preferably not store file paths.
- Make an excel database singleton class instead so that the excel sheet only needs to be read once and then cached. This decreases IO operations.

Appendix



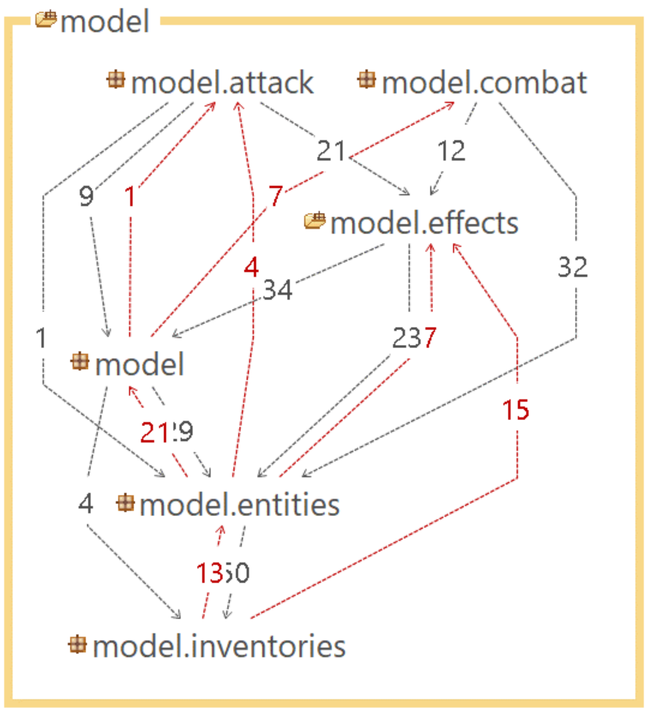
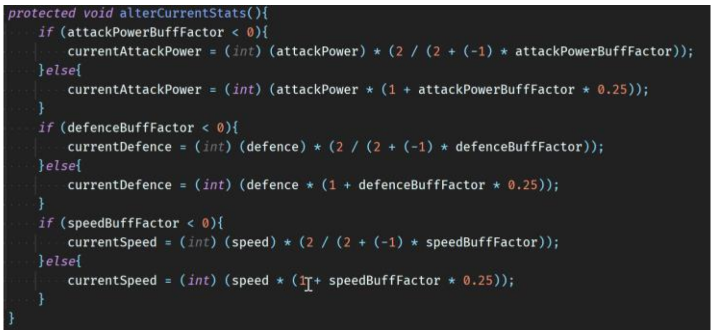


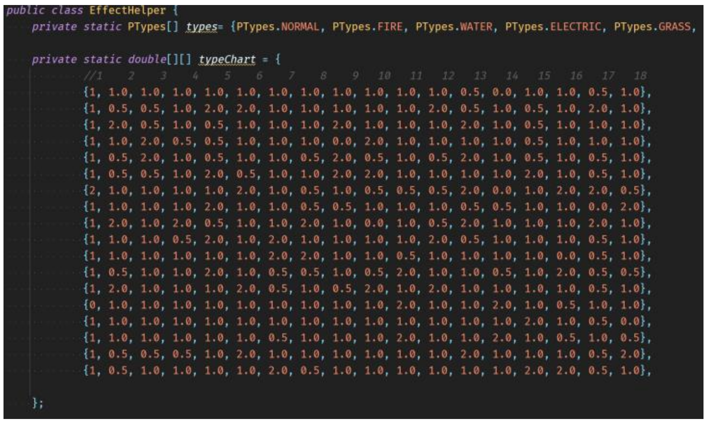
Figure 1: Root folder

Figure 2: Model package



```java
protected void alterCurrentStats(){
    if (attackPowerBuffFactor < 0){
        currentAttackPower = (int) (attackPower) * (2 / (2 + (-1) * attackPowerBuffFactor));
    }else{
        currentAttackPower = (int) (attackPower * (1 + attackPowerBuffFactor * 0.25));
    }
    if (defenceBuffFactor < 0){
        currentDefence = (int) (defence) * (2 / (2 + (-1) * defenceBuffFactor));
    }else{
        currentDefence = (int) (defence * (1 + defenceBuffFactor * 0.25));
    }
    if (speedBuffFactor < 0){
        currentSpeed = (int) (speed) * (2 / (2 + (-1) * speedBuffFactor));
    }else{
        currentSpeed = (int) (speed * (1 + speedBuffFactor * 0.25));
    }
}
```

Figure 3: "alterCurrentStats" method in model class Puckemon. Because of the lack of comments the code is harder to understand.



```java
public class EffectHelper {
    private static PTypes[] types= {PTypes.NORMAL, PTypes.FIRE, PTypes.WATER, PTypes.ELECTRIC, PTypes.GRASS,

    private static double[][] typeChart = {
            //1     2    3    4    5    6    7    8    9    10   11   12   13   14   15   16   17   18
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5, 0.0, 1.0, 1.0, 0.5, 1.0},
            {1, 0.5, 0.5, 1.0, 2.0, 2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 0.5, 1.0, 2.0, 1.0},
            {1, 2.0, 0.5, 1.0, 0.5, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 1.0, 1.0},
            {1, 1.0, 2.0, 0.5, 0.5, 1.0, 1.0, 1.0, 0.0, 2.0, 1.0, 1.0, 1.0, 1.0, 0.5, 1.0, 1.0, 1.0},
            {1, 0.5, 2.0, 1.0, 0.5, 1.0, 1.0, 0.5, 2.0, 0.5, 1.0, 0.5, 2.0, 1.0, 0.5, 1.0, 0.5, 1.0},
            {1, 0.5, 0.5, 1.0, 2.0, 0.5, 1.0, 1.0, 2.0, 2.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0},
            {2, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 0.5, 0.5, 0.5, 2.0, 0.0, 1.0, 2.0, 2.0, 0.5},
            {1, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 0.5, 0.5, 1.0, 1.0, 1.0, 0.5, 0.5, 1.0, 1.0, 0.0, 2.0},
            {1, 2.0, 1.0, 2.0, 0.5, 1.0, 1.0, 2.0, 1.0, 0.0, 1.0, 0.5, 2.0, 1.0, 1.0, 1.0, 2.0, 1.0},
            {1, 1.0, 1.0, 0.5, 2.0, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 1.0, 1.0, 0.5, 1.0},
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 1.0, 1.0, 0.5, 1.0, 1.0, 1.0, 1.0, 0.0, 0.5, 1.0},
            {1, 0.5, 1.0, 1.0, 2.0, 1.0, 0.5, 0.5, 1.0, 0.5, 2.0, 1.0, 1.0, 0.5, 1.0, 2.0, 0.5, 0.5},
            {1, 2.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 0.5, 2.0, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 0.5, 1.0},
            {0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 1.0},
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 0.5, 0.0},
            {1, 1.0, 1.0, 1.0, 1.0, 1.0, 0.5, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 2.0, 1.0, 0.5, 1.0, 0.5},
            {1, 0.5, 0.5, 0.5, 1.0, 2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 1.0, 1.0, 1.0, 0.5, 2.0},
            {1, 0.5, 1.0, 1.0, 1.0, 1.0, 2.0, 0.5, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 2.0, 2.0, 0.5, 1.0},

    };
```

Figure 4: double array filled with magic constants. The lack of comments, except for row index, makes the purpose of this array an enigma wrapped in a riddle.