Assignment 3

Game: Temple Run

by

Group 9

Course: TI2206

Software Engineering Methods

Maarten Sijm
Robin van Heukelum
Mathias Meuleman
Mitchell Dingjan
Maikel Kerkhof

TA: Jurgen van Schagen

Teacher: Alberto Bacchelli

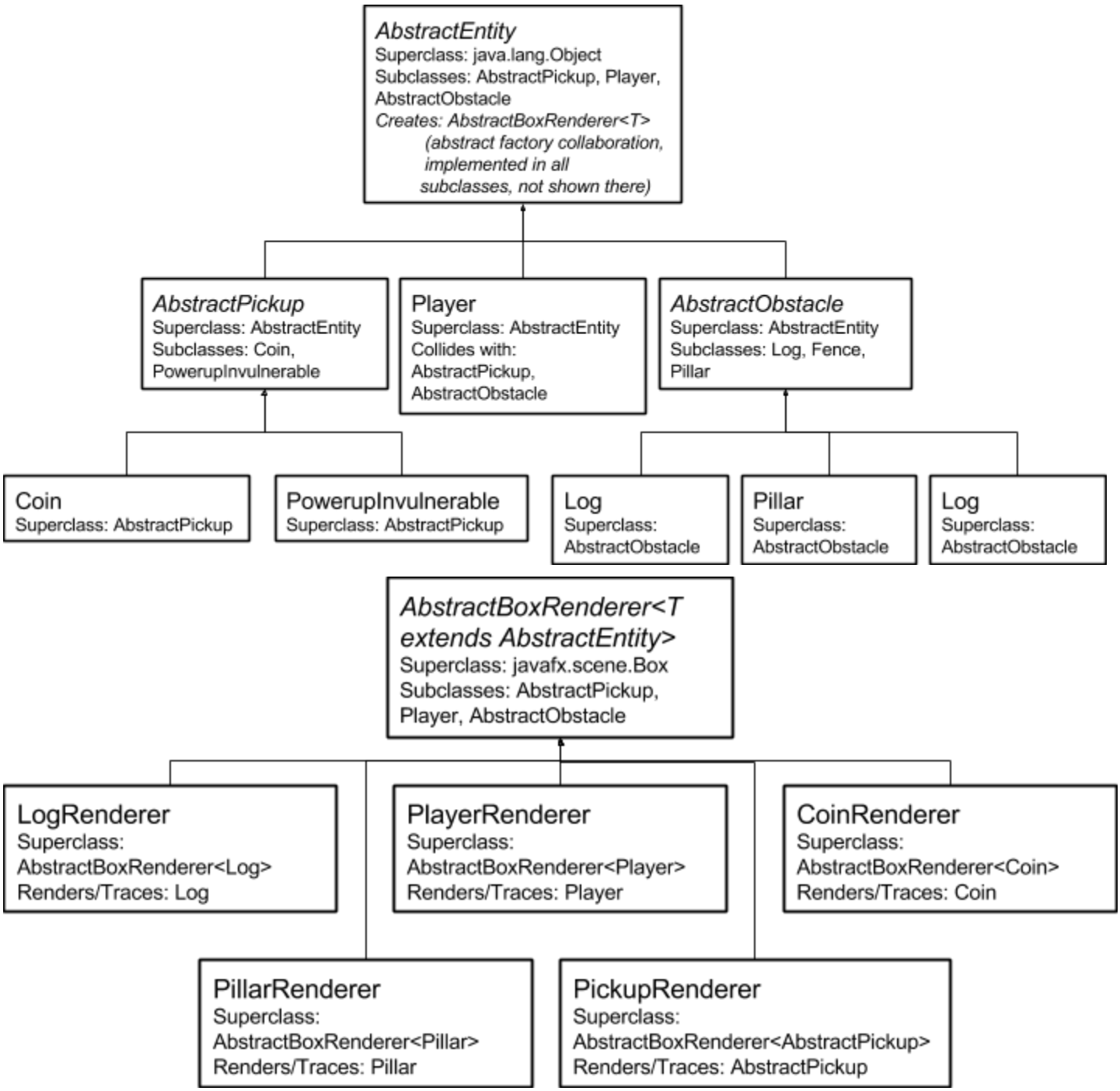
09-10-2015

Exercise 1 – 20-Time, Reloaded

Requirements

- The player shall be able to pick up power-ups.
    - Power-ups are placed on the track, just like coins.
    - When a power-up is picked up, the action belonging to the power-up is executed immediately.
    - All actions will last for a yet to be determined amount of time.
    - Some of the actions include: invulnerability, coins moving towards the Player, slowness.

CRC cards



Responsibility Driven Design

The CRC cards show also a bit our refactoring of the GUI packages. The classes added in this functionality are AbstractPickup, PowerupInvulnerable and PickupRenderer. CoinRenderer has been altered to display a coin texture instead of a question mark box, which is now being displayed by the PickupRenderer.
The Single Responsibility is clearly visible in our structure. The back-end Entities know only how to interact with each other, the only thing they know about their rendering is the type of Renderer they should have. This is to avoid a lot of `instanceof` switches. The Renderers on their side know nothing about how the Entities interact, but they do know their position and size, so they can be rendered with the proper texture.

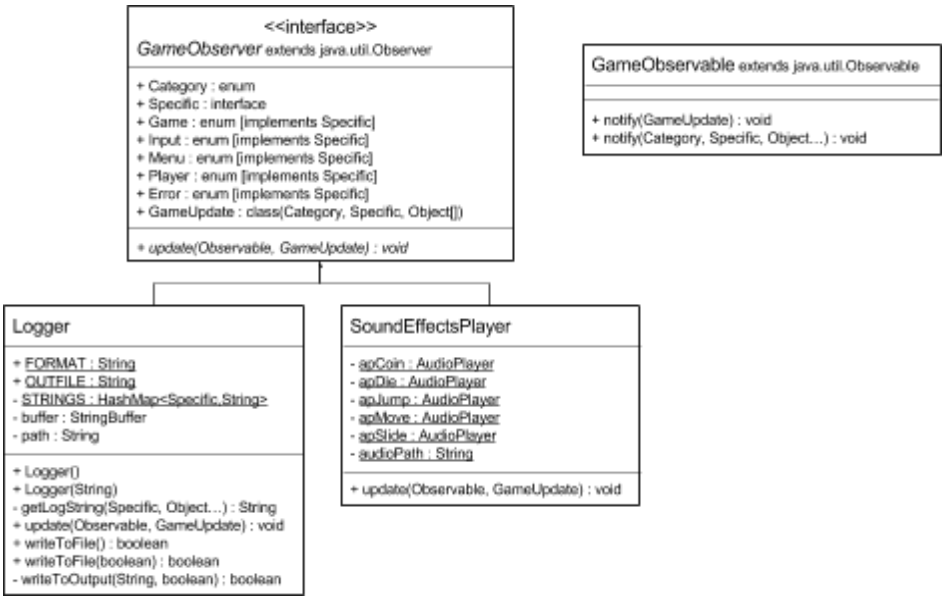
Exercise 2 – Design Patterns

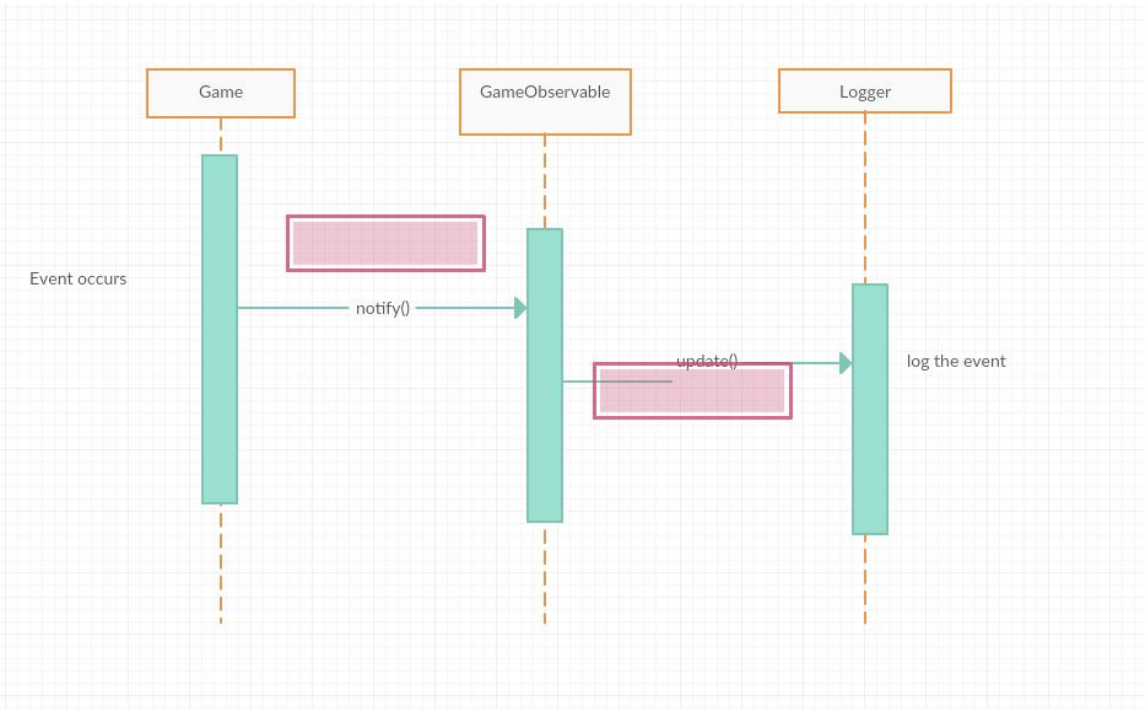
We've chosen to implement the following 2 design patterns: The Observer Pattern and the Decorator Pattern.

Observer Pattern

This pattern was a very logical choice for us. We've used the pattern in our logger extension. The logger has to log all actions that happen during the game. Therefore it has to listen to the game and log all that it has to say. This listening is simplified with the Observer Pattern. It enables us to log to the console, because the observer has a one-to-many dependency between the objects. We could even log our actions to a server located somewhere else in the world. There could be many more observers, for example the sound effects player, which is interested in Player events.

We simply call GameObservable.notify() with a set of arguments defining the action that happened. The GameObserver interface has an extensive list of enumerations for all important events that can happen during the game. The logger gets called after each important event in the game to log this to a file, as well as the sound engine, which plays the sound effects at player actions.

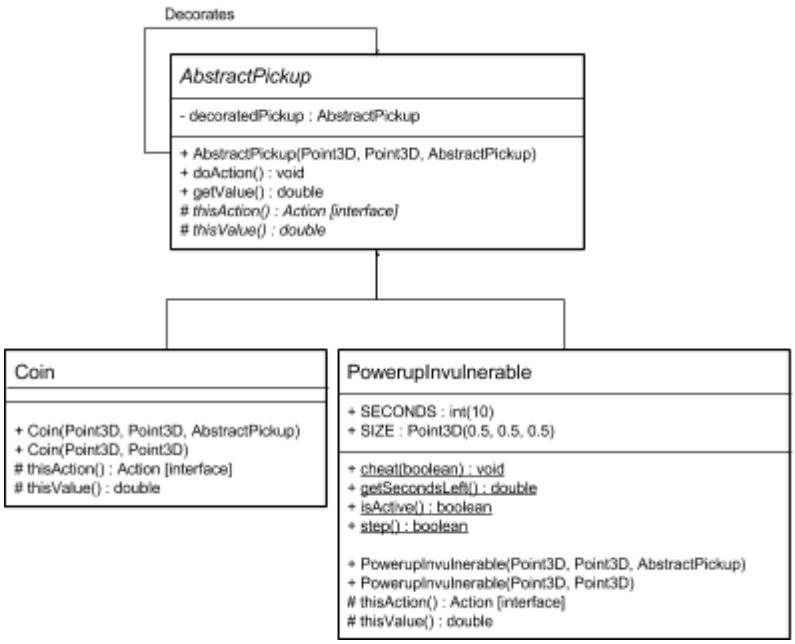


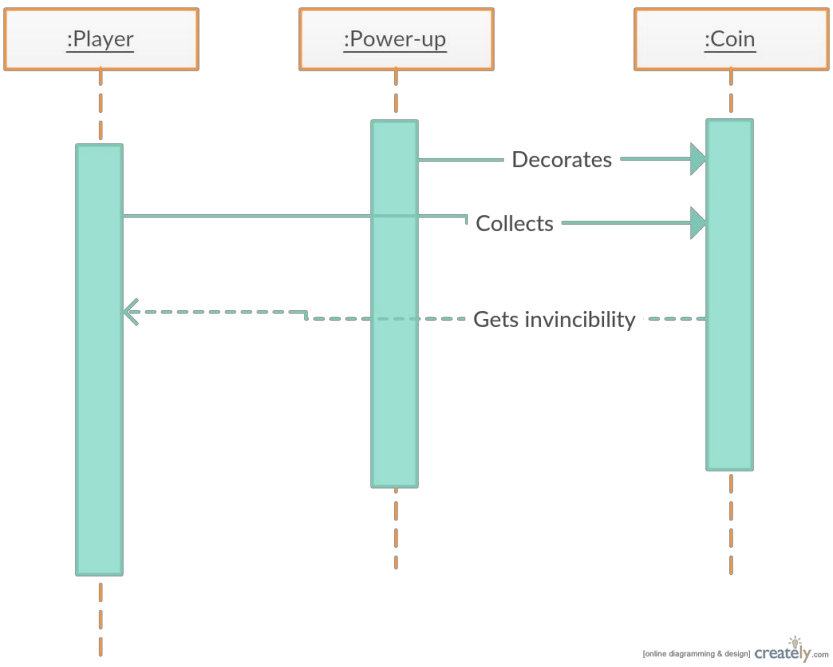


Decorator Pattern

An example for this pattern was given in class: use it for your player power ups. This was indeed a very good way implement this feature. Pickups can be decorated with other Pickups to increase functionality without having to create a quadratic amount of pickups relative to each possible pickup.

Since the start of this project we've maintained an AbstractEntity class to serve as a superclass for all of our entities. The AbstractPickup extends from this class and decorates itself. Therefore it contains an AbstractPickup as a field. The AbstractPickup defines two protected abstract methods: `thisValue()` and `thisAction()`. There are also two public methods: `getValue()` and `doAction()`. These two methods calculate the total value resp. execute all actions that belong(s) to all the decorated pickups. In a sense, you could create a chain of pickups this way. E.g: an invulnerability pickup decorating a coin, meaning that the player becomes invincible ánd gets a coin when it is picked up.





Exercise 3 – Software Engineering Economics

1. *Explain how good and bad practice are recognised.*
   According to the paper (2.4.2):
   "A practice is considered as a good practice when the performance on both cost and duration is better than the average cost and duration corrected for the applicable project size."
   On the other hand, "a practice is considered as a bad practice when the performance on both cost and duration is worse than the average cost and duration again corrected for the applicable project size."
   So in other words, if there is a practice that seems to reduce the costs or (inclusive or) duration compared to the average costs and duration (and taken into account the applicable project size), then the practice is a good practice. And then of course, if it increases costs or duration, then the practice is a bad practice.
   In the research this was done by *classifying the projects in a Cost/Duration Matrix*, in which project performance is challenged against the performance of the whole sample. Then the 56 project factors that had been defined were analyzed on how they are related to the four quadrants of the Cost/Duration matrix. This resulted in two percentages; a percentage based on the *number of projects per quadrant for every factor* and a percentage based on *the cost per quadrant for every factor*. Furthermore, the *significance* of the two outcomes was calculated (with a chi-square test[1]). Based on these numbers, factors were identified that are strongly related for the composition of software engineering project portfolios, by analyzing specific subsets per research aspect. In relation to this, factors are strongly related if the significance percentage Good Practice/Bad Practice was *at least 50%.* Finally a practice was recognised by being 'good' or 'bad', *by generating two hypothesises* (the factor does influence or doesn't) and *testing them by a probability function* with a binomial distribution.

2. *Explain why Visual Basic being in the good practice group is a not so interesting finding of the study.*
   What is expected to be interesting about to the results of this study is probably related to useful methods/tools that can be applied in order to increase the chance for a new project to be successive. The (largest) target group of the study is therefore related to the relative complex projects. The programming language Visual Basic is generally considered as a relatively easy programming language for less complex projects. Thus applying the programming language Visual Basic in relative complex projects can be hard / undesired, as requirements cannot be reached with this language. Thus the fact that Visual Basic is a good practice according to this paper, is not interesting for the target group of this paper.

[1] https://en.wikipedia.org/wiki/Chi-squared_test

3. *Enumerate other 3 factors that could have been studied in the paper and why you think they would belong to good/bad practice.*

- The use of Static Analysis Tools (e.g. PMD, Checkstyle, Findbugs) in projects. We think that this is a good practice, as Static Analysis Tools contribute to increase code quality and decrease the amount of defects. This could result in a decrease of duration and cost, as developers are (directly) notified by the warnings of the Static Analysis Tools.
- Not using UML (diagrams) (for e.g. modelling the system, communicating with project members). We think that this is a bad practice, as UML diagrams contribute to thinking about how the developers want to design a system in an early stage and are effective when developers explain system relative issues among each other. Besides, UML diagrams can be used as a lookup source for the developers later on. Of course UML diagrams have to be manually updated and might be complex for large systems, but we think that the pros are more valuable than the cons, as UML increases the insight on the system for developers. Thus not using UML could prevent developers form a decrease in time and costs.
- Applying Test Driven Design in projects. We think that this is a good practice, as via this way programmers are forced to both write tests and take into account how they want to implement their features. In our opinion, this would result in less defects and thus in a decrease of cost and duration, because developers save time by finding defects in an early stage of the project.

4. *Describe in detail 3 bad practice factors and why they belong to the bad practice group.*

- Rules and Regulations driven is a bad practice, as rules and regulations reduce the freedom of developers on designing and programming their system. Developers have to take rules and regulations into account, which could result in more complex systems. For example, in The Netherlands we have the rule that websites should ask the user for permission to use cookies. This makes the website more ‘complex’ in terms of they have to do more programming in order to satisfy the rules and regulations. Thus duration is increased by this factor (and thus costs).
- Dependencies with other systems is a bad practice, as software engineers have less freedom in creating their system; they have to take their dependencies into account. This could result in having to create some kind of ‘adapter’ in order to be able to make the system cooperate with the system that is designed by the engineers. This of course takes more time, as engineers have to develop this adapter. Thus duration and costs increase because of this factor.
- Once-only projects is a bad practice. This can be the result of using the waterfall model. For medium/large systems it is recommended to not use the waterfall model, as getting early feedback is very useful for the project and showing to the stakeholders progress results of course in confidence. For smaller systems, the waterfall model might not be a bad practice, as applying a model like scrum, might be overkill and can eventually result in an increase of duration.
  Furthermore, in relation to once-only projects, software engineers might have a too easy attitude towards the project; they have to create a system with just one deadline, and then they’re done. This can delay the project, as the engineers take it easy at the beginning of the project and as the deadline approaches, they ask for more time.
  Because of the reasons described above, once-only projects can result in an increase of duration and costs.