SEM - ASSIGNMENT 5

Group 42

Exercise 1 - Anonymous peer suggestions

The anonymous peer review found a couple of good points where our code was bad. Luckily, most of the points they picked we had already covered last week. There were some points left however that we had to consider:

KeyboardInputManager does not need to be an observable

KeyboardInputManager was implemented as a hybrid between an observer and an utility class. This worked but actually did not make sense: The observer pattern needed an instance, so the class could not really be a utility class. Also the functionality gained by making KeyboardInputManager an observable was never utilized as such.

We changed the KeyboardInputManager class to be just a utility class. Now other classes can just check whether a key is pressed, down, released, or up during their update cycle which happens approximately 60 times / second.

Input interface

We decided not to go for an input interface. This is because the input controls are already configurable as much as we would like. Also, keyboard input alone is sufficient to play the current game and any extensions we foresee anytime soon.

Test methods are commented, have dubious names and have useless implementations

This is a reasonable point. The previous week, we already refactored some tests and got the coverage above 75%. This week we will continue on doing so. Also we will try to remove all commented tests.

Exercise 2 - Software Metrics

When we ran inCode, it did not give us any design flaws. This is great, but it did not help us improve our code, while it was clear that our code was not optimal at all yet.

Our TA Gijs helped us by applying stricter PMD rules. This revealed some flaws in the code which did not seem like much, but after all led to quite a lot of changes in the codebase.

No abstract methods in AbstractPowerUp

This first design flaw, was the easiest to fix. The only thing that needed to be done was creating the method “activate” in AbstractPowerUp. This method is then overridden in DuringPowerUp and InstantPowerUp. This solves the flaw.

Empty methods in abstract classes should be abstract

The AbstractEntity class had some methods which were empty yet not abstract. To fix this we created two interfaces for entities: DynamicEntity, CollidingEntity.

- The DynamicEntity interface corresponds to entities which have custom behaviour that must be executed each update cycle.
- The CollidingEntity interface corresponds to entities which have custom behaviour when they collide with another entity.

This made the class structure a bit more complex, but it improved the efficiency of the overall game:

A level used to loop over all the entities that are in there. Now it holds a separate list of DynamicEntities - which is way smaller than the original - and only loops over these.

Classes contain too many methods

This was by far the hardest problem to tackle. It required a lot of rework and resulted in the most improvements. Two big results are: more use of class composition and the implementation of a state pattern.

The Character class

The Character class used to be a single class which was composed of only one vine and one shield. This resulted in a lot of methods which did not directly had to do with character itself.

By delegating parts of functionality of the Character to different classes, the Character class held way less responsibility and thus way less methods.

The first step was to delegate the character’s sprite choice into a CharacterSprites class.

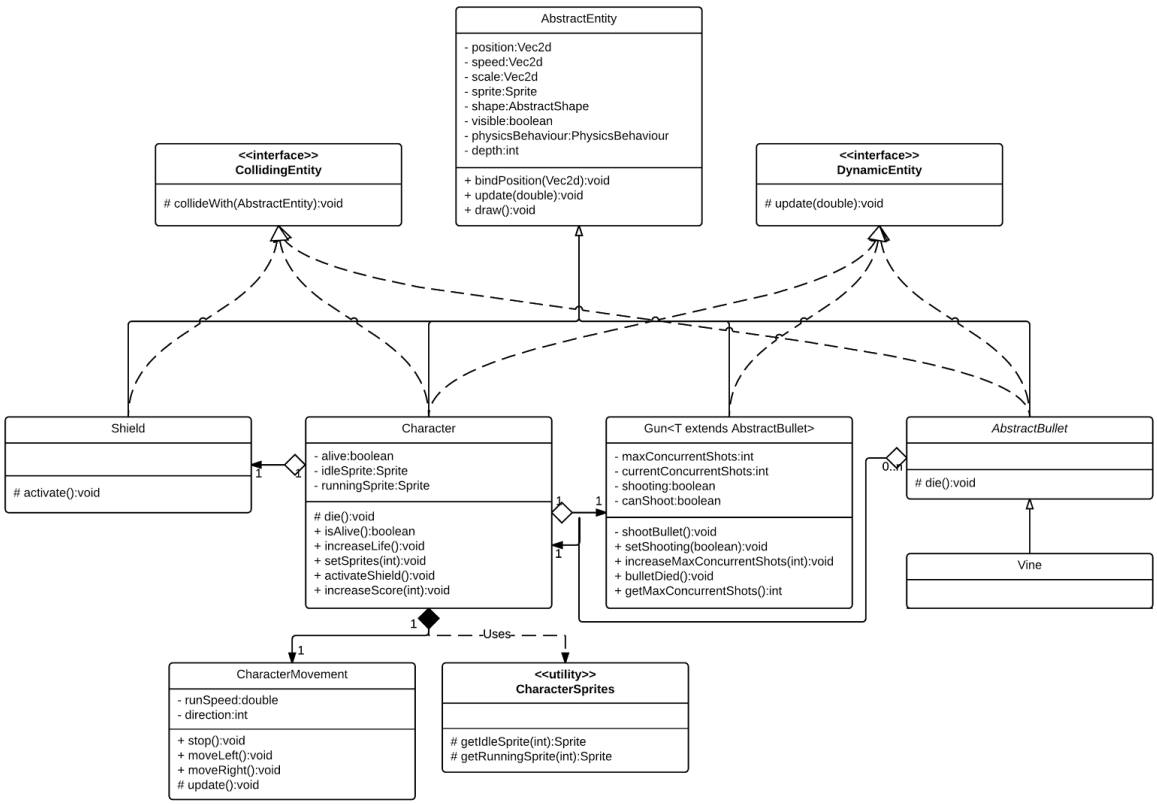
The second step was to delegate all the character’s shooting behaviour to a Gun class. This includes:

- The amount of bullets a character can shoot concurrently.
- Which bullets a character shoots.
- When a character can shoot a bullet.

The third step was to delegate the character's movement behaviour to a CharacterMovement class. This includes:

- The character's run speed.
- The direction the character is running in.
- Whether the character is moving or not.

This results in the following class structure:

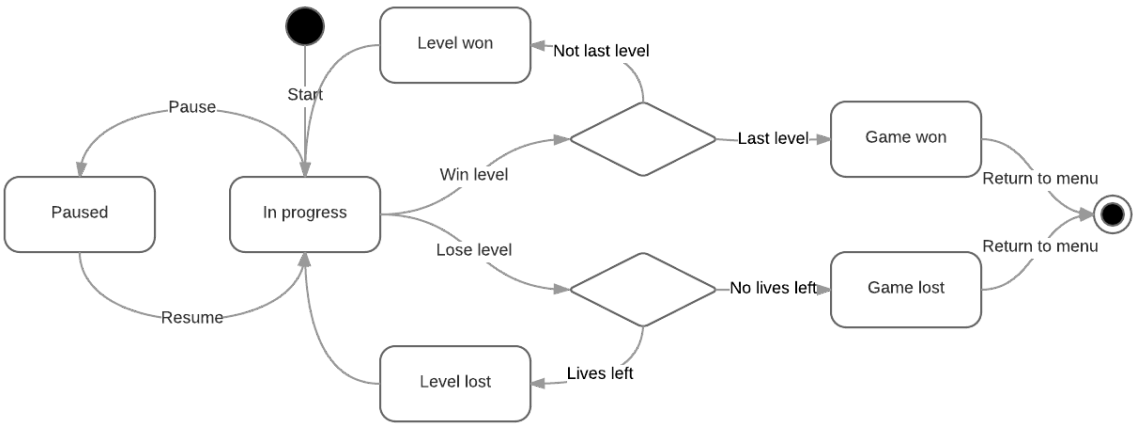


The GameState class

The Game class used to aggregate a GameState object. This GameState object however was not really a state machine. It was just a single class containing all of the state of the game, which was a lot.

Transforming this into a real state machine was a lot of work, but also made a lot of sense. After this change each state held its own class implementing the GameState interface.

The finished state machine looks like this:



The Level class

By transforming the state of the game into a real finite state machine, already a lot of functionality could be removed from the Level class. This alone however was not enough.

Further improvements were made by composing the Level class of an EntityManager and a LevelTimer.

The EntityManager class handles all behaviour that has to do with entities in a level. This includes:

- Depth sorting entities.
- Adding and removing entities.
- Updating entities.
- Drawing entities.

The LevelTimer class handles all behaviour that has to do with timing in a level. This includes:

- Playing a sound when the time is nearly up.
- Increasing the time left when an ExtraTime power-up is activated.

In the future a LevelBuilder class might remove the need for getters and setters for name, background, size in Level. This would simplify the Level class even more, but for now, this is enough.

Exercise 3 - Teaming up

This monday we suggested some features to our TA Gijs. From there we figured two main features to implement for the week. A ceiling for all levels and a gate in level 5. The ceiling exist of spikes, and if a ball collides with the ceiling the ball splits in two smaller balls, of it disappears if it already is the smallest type of ball. The gate is a green pipe in the middle of a level and above it is a wall. The gate disappears when the left ball is gone.

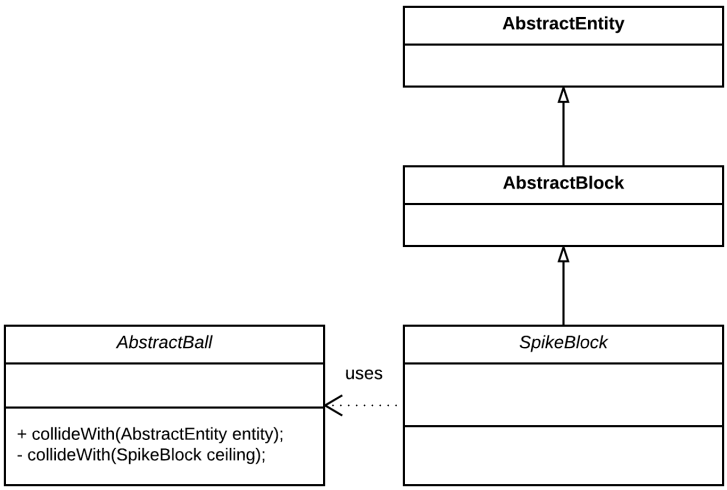
Feature 1: Ceiling

Requirements

The requirements can be found at:

https://github.com/MatthijsKok/TI2206/wiki/Requirement-documents#ceiling

UML

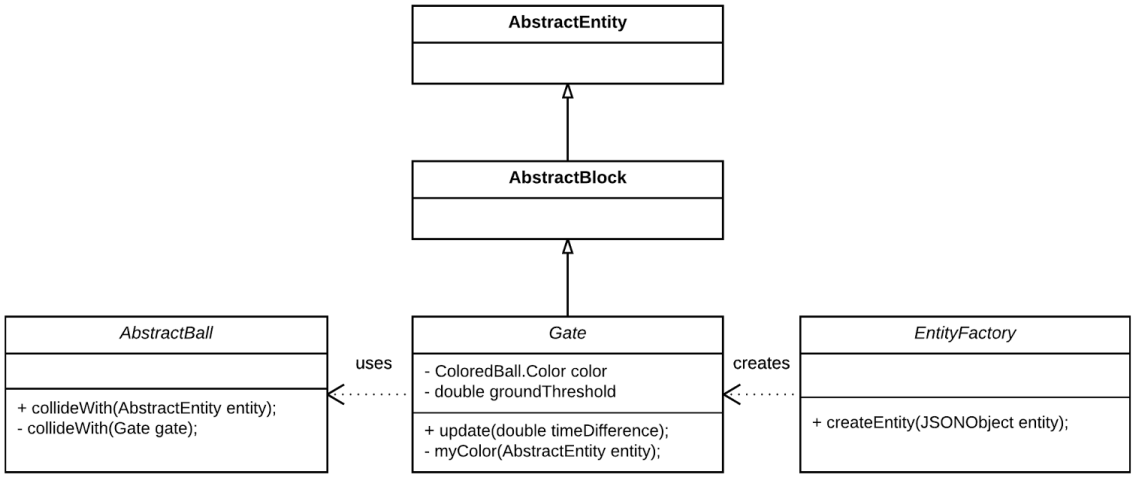


Feature 2: Gate

Requirements

The requirements can be found at:
https://github.com/MatthijsKok/TI2206/wiki/Requirement-documents#gate

UML


AbstractEntity
AbstractBlock
AbstractBall
+ collideWith(AbstractEntity entity);
- collideWith(Gate gate);
uses
Gate
- ColoredBall.Color color
- double groundThreshold
+ update(double timeDifference);
- myColor(AbstractEntity entity);
creates
EntityFactory
+ createEntity(JSONObject entity);