Assignment 1

TI2206: Software Engineering Methods

Exercise 1 - The Core

Description of making the CRC cards

From the requirements we made for the Gudeballs game, five objects stand out the most: Ball, Receptor, Track, Level and Nexus. Since a ball is able to travel over a Receptor, Track and Nexus, we introduce a traversable which indicates a ball is able to travel over this entity.

Other nouns like Score Counter and Timer fit more as attributes of (for example) the level class, so we don't view these as candidate classes.

Furthermore, objects like Locks and One way tracks can be found in the requirements. From this we form the candidate classes:

1. Ball
2. Traversable
3. Receptor
4. Track
5. Level
6. Nexus
7. LockTrack
8. One way Track

Responsibilities

The Ball class is an object that gets passed around between the Traversable entities in the game (Receptor, Track, Nexus). It has a Type, which holds its color.

A Traversable entity should be able to accept balls to allow them to travel over the entity. It should also be able to release the ball to pass it on to the next entity.

A Receptor must be able to have its slots rotated. A Track should interconnect two Receptors and allow balls to travel over it. A Nexus should spawn (random) balls and hand them over to a Receptor.

With these responsibilities, we have created CRC cards which can be found in the same directory as this document.

Comparison with the actual implementation

The difference with the design that came up with the creation of the CRC cards that there is no notion of a grid on which entities are placed. This notion is used in our implementation and simplifies traveling of the balls over these entities, since the four tiles on this grid where balls can go to or be received from are known.

Design of implementation

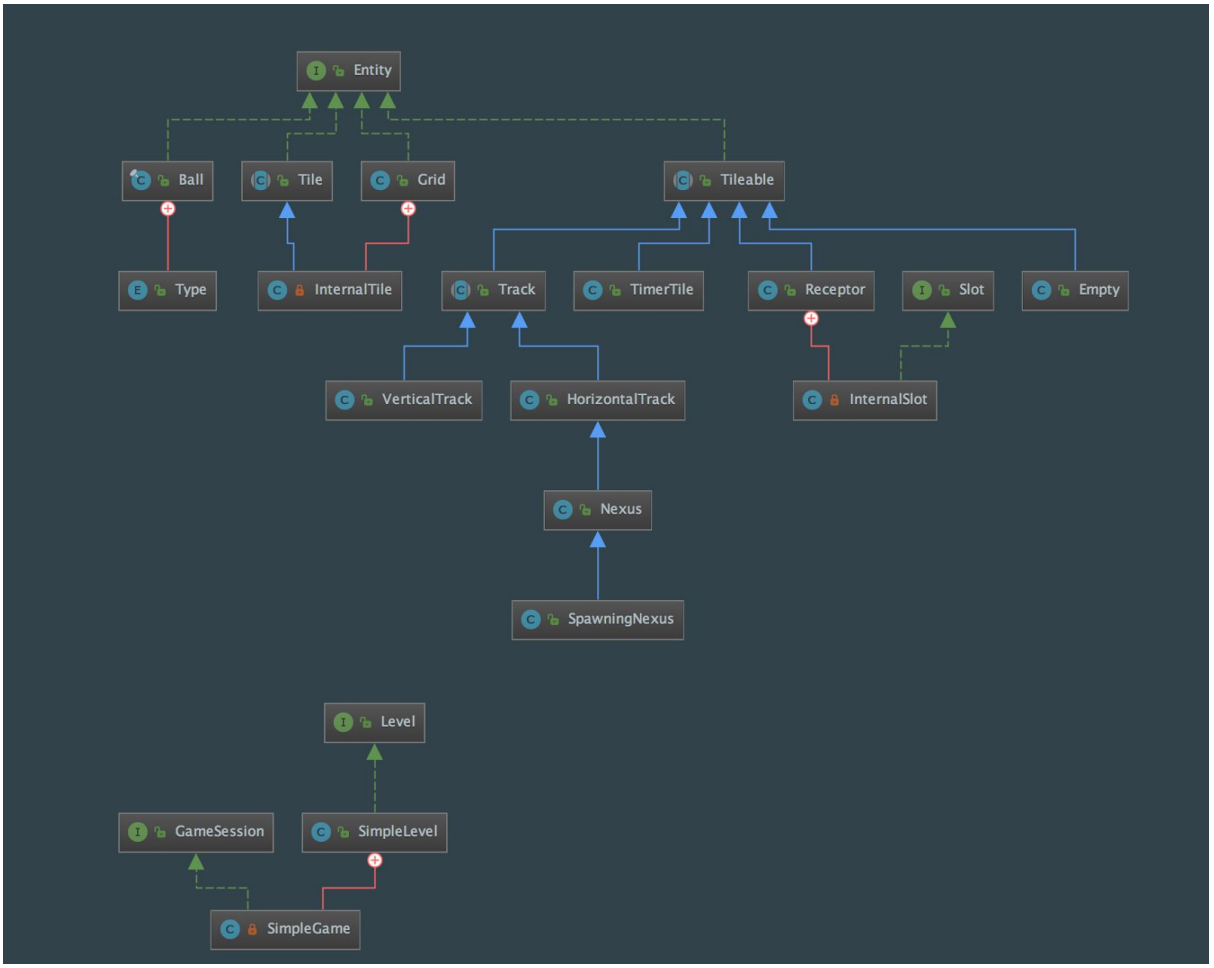
The main classes we use in the implementation mostly align with the design described above. These main classes are:

- Grid - A grid of tiles on which tileable entities are placed
- Tileable - Abstract class which is implemented by entities which can be placed on a tile
- Receptor - A tileable entity which accepts balls in empty slots and allows release of balls
- Track - A tileable entity which accepts balls from its two endpoints
    - HorizontalTrack
    - VerticalTrack
- Nexus - A horizontal track which delivers balls to a receptor if the receptor is placed below the nexus
    - SpawningNexus
- Empty - The empty tileable, which shows a texture of a tile with no tracks or receptors
- Level - A description of the grid of a game

The non-main classes of the project are Entity, Ball, Direction, GameSession, NexusContext and Slot. We could have refactored these classes away, but we think the current approach is clearer.

- Ball does not have many responsibilities, but is simply used a dumb object that is passed around.
- Direction enum could be converted to a simple integer, but we think using an enum conveys the meaning better and prevents errors.
- Entity interface does not have any responsibilities, since it does not define any methods. This means the interface is basically useless and could be removed in future releases.

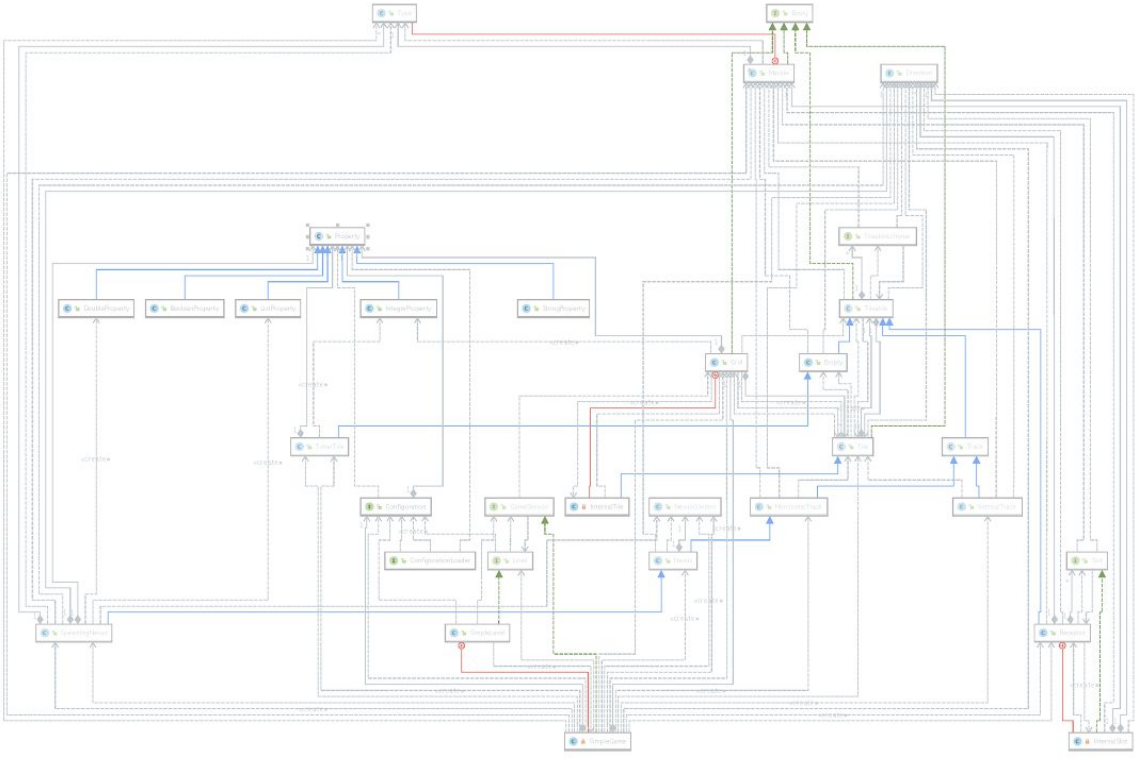
Exercise 2 - UML in practice



1. We use composition for the construction of a *Level*. In our architecture, the *Level* is responsible for (and the owner of) all entities within that level, meaning that when the level is disposed, we assume the entities of this level are also disposed, since they serve no purpose anymore (because they are tightly bound to a Level). Aggregation is a form of association with the meaning of a part-whole relationship (such as a car-exhaust and a car), where parts of a whole can be shared with other wholes. An example of this is the *Ball* class, since this class is not tightly bound to a *Level* and can be used by multiple levels at the same time. Therefore, disposal of the ball should not be done b
2. The code base currently contains several parameterized classes. One instance where we use a parameterized class is the *Property<T>* class which represents a

generic configuration property of type *T*. In this way, we have a *Configuration* object that unwraps a *Property<T>* to *T* in case the property is valid or returns the default value of the property. The benefit of parameterized classes is that is stimulates code reuse while still guaranteeing type safety.

3. Polymorphism is the provision of a single interface to entities of different types. In other words, it means that one single action, can be used in different ways. An Is-a relationship is based on the concept of inheritance. For example, lasagna is a type of Italian food, a banana is a fruit and a house is a building. This however is a one way lane; a fruit is not a banana in Object Oriented Programming. With these definitions, we can have a look at our class diagram. We can see again that all classes but Level, GameSession, SimpleGame, SimpleLevel and Slot are implemented by Entity and by that reason do not inherit something from Entity. However, SimpleLevel and SimpleGame do inherit some of Level and SimpleGame also inherits some of GameSession. The only four classes that don't inherit anything are Slot, Entity and Level and GameSession. Looking at polymorphism, and back at the class diagram, we see 4 interfaces, Entity, Slot, GameSession and Level. We see that all have subobjects and are therefore polymorphing



Exercise 3 - Game configuration

We decided to abstract away the specific library we received and put it in a separate module, partially due to its bad design, but also to the benefit of being able to quickly swap the configuration library with another one by just implementing some interface. The abstraction we created allows for simple type safe property retrieval by some key, falling back to a

default value if either the key does not exists in the configuration or the configuration cannot be parsed.
We also decided the core components of the system should not have knowledge of a *Configuration* object, but instead use parameters passed via the constructor. The conversion of configuration to object parameters is instead done by the selected *Level* implementation.

Also, in the diagram above this section, the UML design is shown including the *config* package which has as base classes *Property<T>*, *Configuration*, *ConfigurationLoader*.

Exercise 4 - Maintaining Traceability Links

Traceability Matrix

1.
We decided to construct our traceability matrix based on knowledge-based links, which means we have references to the requirements from the source code. We chose not to make structural-based links as we had already coded everything and we found the knowledge-based links enough traceability.

For making the traceability matrix we chose to just use a spreadsheet, as we didn't need a very big matrix and making it was still doable. The only disadvantage is that the matrix was a bit too big to properly fit into this document.

2.

Requirement ID	R1 CL-Ball	R1 TC-Ball	R2 CL-Receptor	R2 TC-Receptor	R2 CL-Slot	R3 CL-Track	R3 CL-HorizontalTrac	R3 TC-HorizontalTrac	R3 CL-VerticalTrack	R3 TC-VerticalTrack	R4 CL-Nexus	R4 TC-Nexus	R4 CL-SpawningNexu	R4 TC-SpawningNexu	R5 CL-Level	R5 CL-SimpleLevel	R5 TC-SimpleLevel	R5 CL-GameSession
R1 CL-Ball		x	x	x									x	x		x		
R1 TC-Ball	x																	
R2 CL-Receptor				x												x		
R2 TC-Receptor			x															
R2 CL-Slot			x	x														
R3 CL-Track							x		x									
R3 CL-HorizontalTrac								x								x		
R3 TC-HorizontalTrac																		
R3 CL-VerticalTrack										x						x		
R3 TC-VerticalTrack																		
R4 CL-Nexus												x	x			x		
R4 TC-Nexus																		
R4 CL-SpawningNexu														x		x		
R4 TC-SpawningNexu																		
R5 CL-Level																x		x
R5 CL-SimpleLevel																		
R5 TC-SimpleLevel															x	x	x	
R5 CL-GameSession																		