Reflection

Alex Vosnakis 743946

The project design was changed after the submission of Project 2A. Some of the key changes were the addition of the `Position`, `Explosion`, `Timer`, `Notifier`, `GameUtils`, and `Direction` classes, as well as the `Destructible` interface.

Due to the multiple entities requiring the usage of a timer (for example, the `Explosion`, `Ice`, `Skeleton`, and the `World`'s undo system, a `Timer` class was created to support the abstraction of these timing functions.

The `Explosion` made sense to abstract into a class, as it had its own functionality that could be considered independent of the `Tnt`.

The `Position` class was created midway through development to alleviate the amount of attributes that the `Sprite` had which were just coordinates in some way. It became too difficult to manage up to four coordinates at once (the `Sprite`'s grid coordinates and window coordinates), so the generic and immutable `Position` class was made with the specification that `T extends Number`. The class was made immutable to make all `Sprite` movement easier to understand; an immutable class must be created every time the `Sprite` moves, making it easier to debug movement issues.

`Notifier` was implemented as `SwitchNotifier extends Observable` as its purpose was to alert the `Switch` or `Tnt`'s corresponding `Door` or `CrackedWall` respectively to when its state changed while preserving encapsulation.

`GameUtils` was created as a static class containing useful methods that needed to be called in many different places.

`Direction` is an `enum` made to represent the different directions possible to move in. This was made as the `Directions` need to be referred to in many places so representing them as a `public enum` rather than as `static` variables made sense.

One of the main challenges of the project was the requirement to have multiple classes on the same level of the hierarchy needing to make changes or refer to other classes without direct access to each. This was first encountered early in the project, when making the functionality to push blocks. I saw a few ways to approach this:

1. A `Sprite` could poll a `public static` data structure recording the positions of other `Sprites`. This was attempted but mutability issues made it difficult to maintain, and making it `public static final` meant I could not actually change the data structure.

2. The `World` could be passed as an argument to the `Sprites` during an `update`. For example, `sprite.update(input, delta, this)` when calling from the `World`, giving the `Sprites` access to all `public` methods, making all attributes `private`. This has the tradeoff of reduced encapsulation and more coupling but makes implementation simpler.
3. I could have the `Sprites` only update frame-by-frame, meaning that should a `Player` attempt to move a `Block`, the `Player` would occupy the same position as the `Block` for a frame until the `Block` reads the player's position from a `static` variable. However, this resulted in many bugs that were difficult to fix given the inconsistent update state of the blocks, including a lot of unexpected behaviour.

Approach 1 was abandoned early in, as well as approach 3, leaving the next two options for implementing approach 2:

1. Then, the `World` itself could record the positions of each Sprite in a 3D array (given that some `Sprites` can share the same (x,y) coordinate as other `Sprites`), allowing fast access using a `Position<Integer>`.

2. Or, every time the position of another `Sprite` needed to be polled, the World would just iterate over the `ArrayList<Sprite>` of sprites, finding the `Sprites` that matches the specified conditions (usually an equal position and/or a category or type).

This first version of approach 2 was taken for the majority of the project, but having to maintain the `Positions` of every `Sprite` as well as this 3D array made it very difficult to debug issues with the undo functionality, as then the 3D array had to be maintained and recorded separately to the `Sprites`' `Positions`. This approach was abandoned and the project was refactored to approach 2ii, which has a far simpler implementation at the (negligible) trade-off of performance.

A key piece of knowledge gained was the use of the subscriber-publisher model, which was implemented for the `Switch` and `Door` classes, as well as the `Tnt` and `CrackedWall` classes. This allowed for the total encapsulation of their functionality by using the `Observer` interface and implementing `Notifier extends Observable`. The `Switch` only knows about the `Door` to the point that the `Door` is added as an `Observer` of the `Switch`, and whenever the `Switch` changes its state it alerts its `Observers`, of which the `Door` is the only one. Therefore the functionality for these two classes are totally encapsulated.

Also used were functional programming utilities in Java 1.8, such as lambdas, streams, and predicates, which were used mainly in the `World` class to make the various methods relying on iteration more abstract and simple to understand.

In the future, if I was to work on a similar project, I would re-evaluate my approach of the `World` class. Given it had access to all the data that many of the `Sprites` needed, it is larger than it should be, given that is has methods for finding references to certain `Sprites` that

other `Sprites` require knowledge of. Additionally, passing the `World` to the `Sprites` in their `update` method makes the two classes highly coupled, however I could not think of a better way to implement the required behaviour without using what essentially amount to global variables. There is likely a better approach, most likely using an Entity-Component system or a more fleshed-out Subscriber-Publisher model, however we did not learn these in time to fully implement them. I am still hesitant to use a `static` data structure though since that seems to defeat the entire purpose of using object oriented programming.