Testing Plan

One of the ideas we adopted from test-driven development is that the design of the tests almost define the functionality of the program.

This is the thought process that we go through.

1. Identify relevant specifications
2. Generalise and expand the specification, noting assumptions
3. Define tests based on the described behavior
4. A functioning implementation is written almost verbatim from the tests

In general, testing should be done at a high level, through playing the game or via DungeonManiaController which ensures that features are well-integrated. Unit testing of internal objects can be useful, however, writing unit tests do not demonstrate that the end product is functional.

We have written an example on how this process can be followed.

---

1. Relevant Specification

The requirements of a Boulder:

1. > `(a)` Can be pushed by a player to an adjacent square, `(b)` but not if the path is blocked by a wall or another boulder
2. > `(c)` When a boulder is pushed onto a floor switch, it is triggered. Pushing a boulder off the floor switch untriggers.

There are actually 3 testable features defined (a, b, c) in the specification.

2. Expand and Generalise

We can then develop these ideas into (debatable) assumptions about the behaviour in general.

- `(b)` The boulder should probably also be blocked by doors, exits and zombie toast spawners which we might categorise as `"solid"`
- `(c)` The boulder is not blocked by floor switches and might not be blocked by `"soft"` objects like portals or collectibles.

3. Defining Tests

The spec and the assumptions together define the tests

`(a)`. Boulder moves left when player moves into the boulder's square AND boulder's is empty.

`(b)`. Both the boulder and the player do NOT move when the boulder's adjacent square is blocked by a `"solid"` object.

`(c)`. Building on test (a). The boulder's movement is not blocked by `"soft"` objects.

4. Implementation using tests as a reference.

Ideas `(b)` and `(c)` imply that all Entities should be either `"solid"` or `"soft"`.

```
abstract class Entity {
+   abstract boolean blocksBoulder();
}
```

Then this might come together looking like

```
class Boulder ... {
    // return true if the boulder can move in the given direction
    boolean move(Direction direction, ...) {
        // get Entity at (position + direction)

        // if empty or blocksBoulder() is false, boulder can move

        // else boulder cannot move
    }
}
```