# Relatório

Nesse relatório justificamos decisões de implementação tomadas em nosso programa.

## Índice



## Implementação

### Dungeon

Inicialmente, havíamos implementado `Dungeon`, a classe que mantém a informação de salas, formato do mapa e entidades, utilizando uma classe normal. Porém, nessa implementação encontramos dificuldades no desenvolvimento pois várias classes dependiam do mapa:

- Várias classes utilizavam o cálculo de visibilidade que existia na `Dungeon`;
- Várias spells precisavam de informações do mapa, como *Teleport* e *Fireball*;
- Cálculo de movimento dos inimigos;

Com isso em mente, consideramos melhor utilizar um Singleton que poderia ser acessado de qualquer parte do programa.

### Salas

Num primeiro momento, havíamos decidido por criar uma classe `Room` que representava uma sala convexa e uma classe `Corridor`, que herdaria de `Room` e teria uma lista de `Room`, já que corredores podem ser côncavos. Essas classes mantinham uma lista de todas as entidades que estavam numa sala a qualquer dado momento. Nessa implementação, as paredes não ocupariam espaço no mapa, se mantendo fiel ao mapa original de *Hero's Quest*, visto abaixo:

Porém, encontramos dificuldades implementando paredes e portas que não ocupavam espaço, principalmente quando fomos implementar passagem de uma sala para outra e linha de visão. Como a especificação não exige que esse aspecto

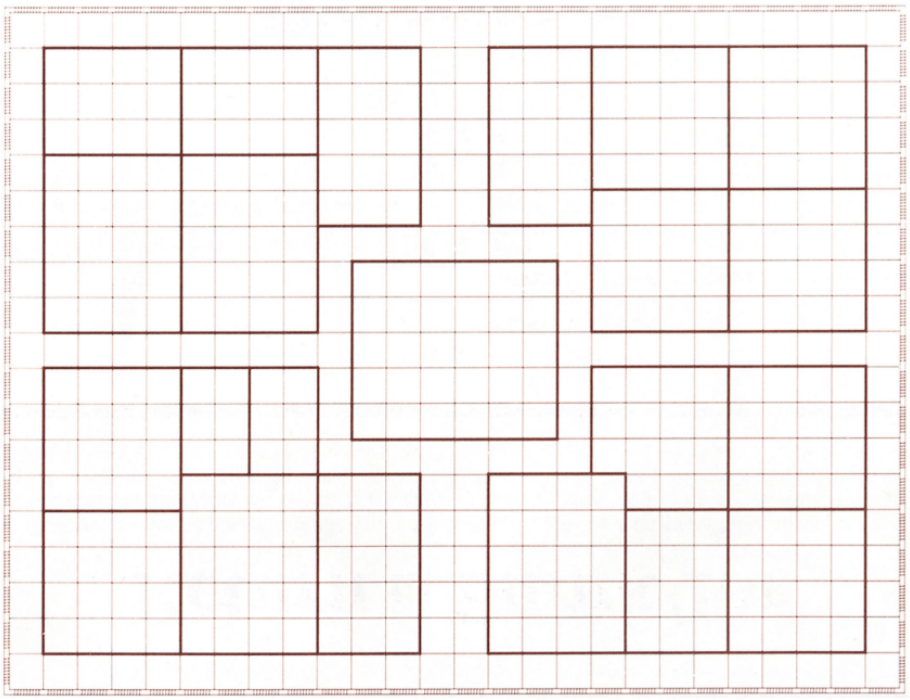

Figure 1: Mapa do Hero Quest

do mapa original seja preservado, optamos por desenvolver salas com paredes e portas que ocupam espaço no mapa.

Nesse novo sistema, se quiséssemos fazer uma sala 2x2, por exemplo, ela teria que ser 4x4, pois a primeira e a última linha e a primeira e a última coluna seriam paredes.

Assim, em nossa nova versão o mapa original de *Hero's Quest* teria o seguinte aspecto:

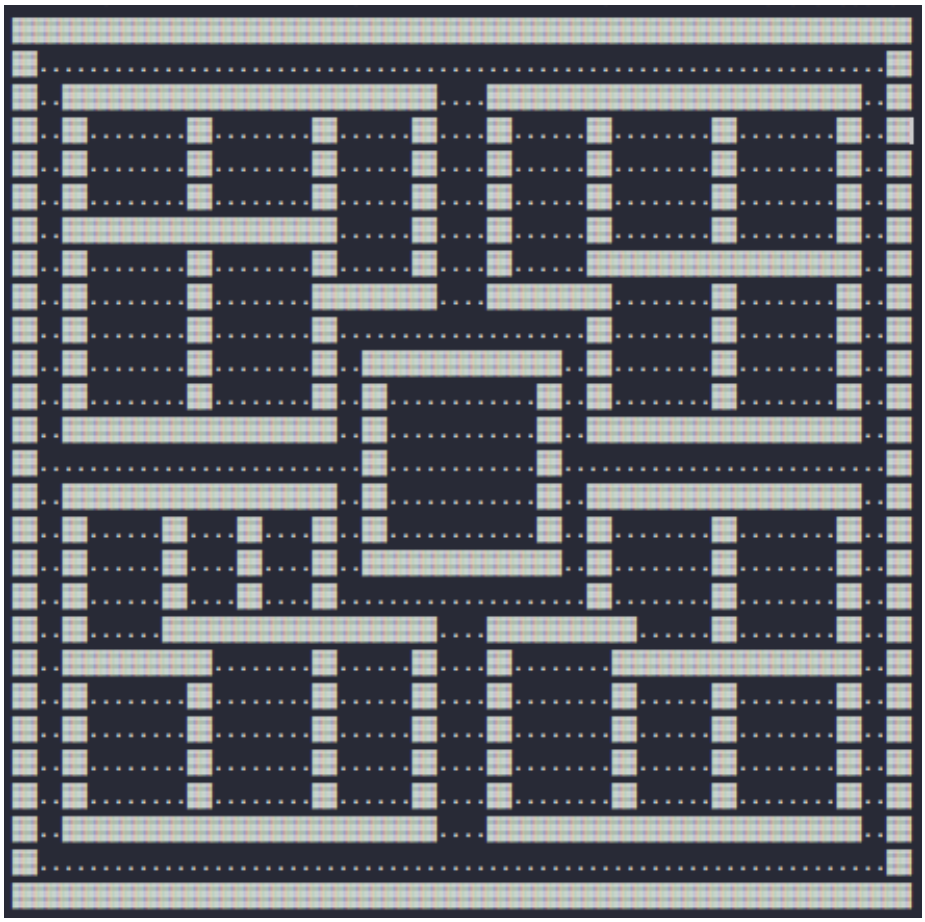

Figure 2: Output do tabuleiro no terminal do jogo

Note que todas as salas se mantiveram iguais, salvo a sala no canto superior esquerdo do bloco inferior esquerdo.

Para implementar a linha de visão, usamos o Algoritmo de Bresenham para detectar o que é visível para o jogador, nos baseando na implementação disponível no RogueBasin. Também usamos a melhoria explicada nesse site, que trata

alguns casos em que o algoritmo de Bresenham se comporta de forma estranha próximo a paredes.

### Inimigos

Durante a implementação dos inimigos, percebemos que eles tinham muitos atributos em comum com os heróis, então fizemos ambos herdarem de uma classe concreta `Character`. Porém, como cada inimigo tinha uma inteligência artificial própria, ficamos em dúvida sobre duas possibilidades de como implementar os diferentes inimigos: uma possibilidade era criar uma classe para cada inimigo, enquanto outra era criar um getter para cada inimigo, baseando-nos no design pattern `Factory`, e usar Interfaces funcionais para definir os comportamentos dos inimigos, como no `Strategy`.

Criando uma classe para cada inimigo, eles podem ser mais versáteis, definindo comportamentos próprios para funções de `Character`, ao custo de que seria difícil reutilizar o código. Usando Interfaces Funcionais, por outro lado, os inimigos são representados somente por instâncias, porém suas inteligências são intercambiáveis, facilitando o reuso de código por composição. Pesando esses prós e contras, optamos pela segunda implementação. ### Spells

A primeira implementação de `Spell` era bem parecida com a de `Enemy`, pois havia um `BiConsumer<Point, Point> castSpell` que implementava a lógica da magia. Porém, a vantagem de usar essa implementação em `Enemy` era que vários inimigos compartilhavam a mesma lógica, então com o reuso por composição diminuíamos os códigos repetidos. Diferentes Spells, porém, não compartilham implementação e, portanto, achamos melhor implementá-las usando herança.

Outro motivo para usarmos herança é que, nas raras ocasiões em que Spells compartilham código, a funcionalidade base continua a mesma, só alterando parâmetros, como uma Fireball 2 que cause mais dano, por exemplo. Nesse cenário, seria mais fácil compartilhar funcionalidades utilizando herança do que composição.