Introdução

Dungeon Fighter é um jogo de RPG, ambientado em uma era medieval, onde você percorre um tabuleiro 5x10, desviando de armadilhas, derrotando monstros e coletando elixires ao longo do caminho. Você joga como um bárbaro, paladino ou guerreiro, cada classe com suas particularidades, e o seu objetivo é derrotar o monstro no fim do tabuleiro.
Esse projeto foi feito em Java, com a biblioteca gráfica Swing, para a disciplina de POO.
O projeto utiliza o *spritepack* 32rogues, disponibilizado gratuitamente por Seth Boyles (https://sethbb.itch.io/32rogues)

Compilando

Para compilar o projeto, é necessário explicitar o *classpath* tanto na hora de invocar o compilador, quando na hora de rodar o aplicativo.

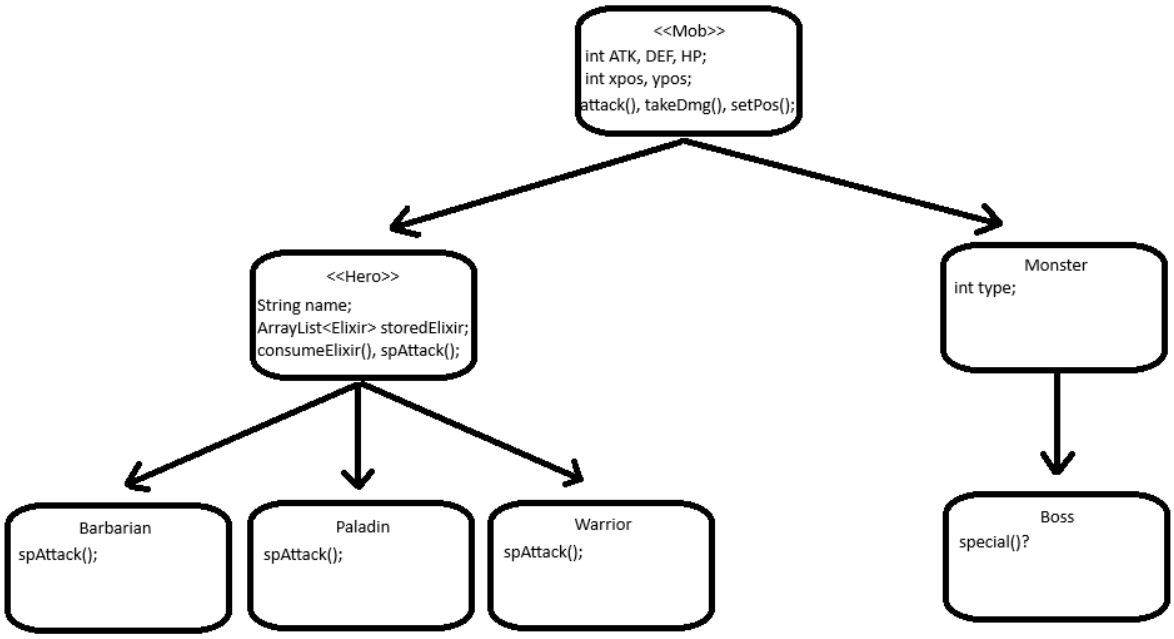
1. Navegue até o diretório /src/
2. Compile com `javac -d bin -cp bin *.java`
3. Execute com `java -cp bin App`
   O projeto foi estruturado desta maneira a fim de esconder arquivos `.class` tanto do usuário quanto do desenvolvedor que for trabalhar no código.

Como jogar

1. Para começar o jogo, clique em **PLAY** (clicar no botão **DEBUG** irá alternar entre o modo DEBUG ou modo normal, o estado atual será printado no console)
2. Escolha sua classe (obrigatório) e insira seu nome (opcional).
3. Distribua seus atributos da maneira que achar mais conveniente (máximo de 20 atributos). Note que as classes já vem com o balanceamento ideal.
4. Uma vez na tela de jogo você terá três opções:
   - **hnt** (*hint*, te dá até 3 dicas sobre as armadilhas na coluna);
   - **mov** (*move*, ao clicar nesse botão, os movimentos possíveis serão mostrados e você poderá se mover para cada um deles);
   - **hlt** (*halt*, para o jogo, conta como uma derrota).
5. Durante a batalha, você terá quatro opções:
   - **atk** (*attack*, ataca o inimigo)
   - **ex** (*extra*, ou ataque especial)
   - **elx** (*elixir*, recupera vida)
   - **esc** (*escape*, tenta escapar da batalha, deixando o monstro de lado)
6. Se você derrotar o chefão, você ganha. Se você morrer em qualquer ponto, você perde.

Diagrama de classes

Para começar esse projeto, é interessante ter uma ideia de como as classes serão organizadas, bem como das interações entre elas. Com a proposta de criar um RPG, o conceito de diversas classes de personagens é óbvio, então comecei a traçar um diagrama preliminar.



Olhando pro diagrama, notei que o chefão não se diferenciava do monstro normal, portanto, foi implementado um especial para o chefão, onde ele regenera 1HP a cada dois ataques que ele não conseguir infligir dano ao jogador.
Também foi criado classes para o elixir e as armadilhas, além de duas classes UI e Engine para separar a interface gráfica da lógica do jogo, que serão discutidas abaixo.

Criando a janela

O primeiro passo para começar o desenvolvimento deste jogo foi criar uma janela utilizando os componentes da biblioteca de interface gráfica Swing. De começo, foi implementada uma janela de início, com os botões "Play", "Debug" e "Quit".

```
// Start screen
JFrame startScreen = new JFrame("Dungeon Fighter");
startScreen.setSize(800, 600);
startScreen.setLayout(null);

// Container
Container screen = startScreen.getContentPane();

// Play button
JButton playButton = new JButton("PLAY");
playButton.setBounds(350, 100, 100, 50);
screen.add(playButton);

// Debug button
JButton debugButton = new JButton("DEBUG");
debugButton.setBounds(350, 200, 100, 50);
screen.add(debugButton);

// Quit button
JButton quitButton = new JButton("QUIT");
quitButton.setBounds(350, 300, 100, 50);
screen.add(quitButton);
```

Como o projeto terá muitas telas, decidi optar por utilizar um LayoutManager.
Após adicionar os botões, precisamos deixar a tela visível:

```
// Set visible
startScreen.setVisible(true);
```

```
startScreen.setLocationRelativeTo(null);
```

Esse processo foi repetido para todas as outras janelas.

Primeiro obstáculo

O primeiro bug que encontrei no meu programa foi justamente com essa janela. Após inicializar ela no *main* e definir seu comportamento de "fechar", o esperado é que a janela feche e o processo seja parado. Isso não aconteceu pois estava criando um objeto ***gameUI*** na main e setando o parâmetro por lá. Ao invés disso, passei a só instanciar ***gameUI*** e configurei o padrão dentro da classe UI:

```
//UI.java
startScreen.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

Após isso, comecei a fazer as próximas telas.

Trocando telas

Após concluir a primeira tela, decidi que iria fazer todas as telas dentro da classe UI, a fim de tentar separar a interface gráfica do jogo em si. Para realizar a lógica de trocar entre JFrames, utilizei um switch case que possui o código de cada tela dentro do seu próprio caso. Os casos são definidos em um enum para facilitar a compreensão do código:

```
public enum windowState{
  START,
  CHAR_SELECTION,
  STAT_MENU,
  INGAME,
  POSTGAME
}
```

Com isso pronto, fiz com que o método `createWindow()` recebesse como argumento um *windowState* e desenhe a tela baseado neste argumento:

```
public void createWindow(windowState currentWindow){
  System.out.println(getCurrentWindow());
  switch(currentWindow){
  case START:
  // Start screen
  // ...
```

Assim, fui criando a interface gráfica específica de cada tela, controlando a troca de telas chamando a função quando algum botão for pressionado:

```
@Override
public void actionPerformed(ActionEvent e){
  if(e.getSource().equals(playButton)){
    setCurrentWindow(windowState.CHAR_SELECTION);
    createWindow(currentWindow);
  }else if(e.getSource().equals(nextButton)){
  // ...
  }
```

Com isso feito, já temos a maior parte da interface gráfica pronta.

Instanciando uma engine

Agora que a interface gráfica foi criada, comecei a implementar a lógica do jogo. Decidi tentar separar ao máximo a interface gráfica da lógica, portanto criei uma classe chamada *Engine.java* que vai fazer a maioria das funções relacionadas ao funcionamento do jogo.
Como o jogador monta o herói antes do jogo começar, tive que instanciar um herói na classe UI e mandar ele para a classe Engine após a seleção dos atributos.

```
case STAT_MENU:
// Instanciar uma classe com atributos base
switch(chosenClass){
  case "Paladin":
    hero = new Paladin(6, 6, 8, 0, 2, heroName);
  break;
```

Após o jogador mexer nos atributos do personagem com botões de incremento/decremento, instanciamos uma engine quando o jogo começa:

```
case INGAME:
// code...
gameEngine = new Engine(hero, debugMode, retryGame);
gameEngine.startGame();
```

Com isso, estamos prontos para começar a programar a lógica do jogo.

Funcionamento da engine

Como mencionei anteriormente, o objetivo dessa classe é realizar o trabalho "pesado", como calcular posições de elementos no tabuleiro e inicializar os monstros. A função `startGame()` é a que realiza esses comandos.

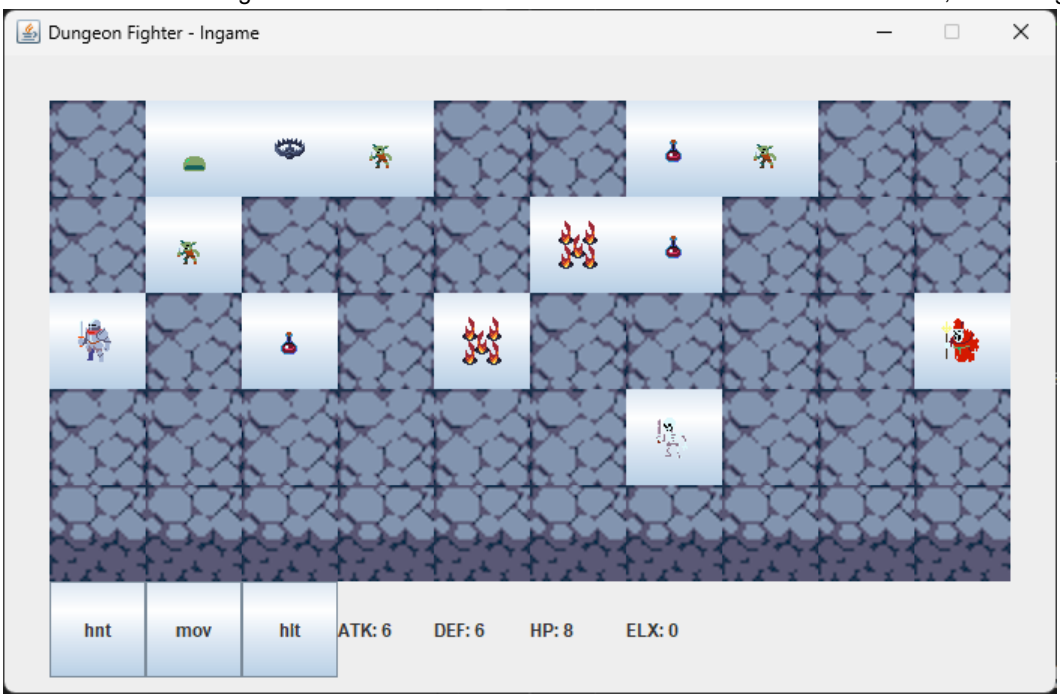
```
public void startGame(){
  if(!retryGame) getSeed();
  spawnPlayer();
  spawnMobs(mobNumber);
  spawnTraps(trapNumber);
  spawnElixir(elixirNumber);
  spawnBoss();
}
```

Quando o jogo começa, várias funções são executadas. Vamos começar pela mais simples: `spawnPlayer()` define a posição do jogador, tira todos elixir da bolsa dele e desenha o sprite no tabuleiro.
A função `getSeed()` inicializa as arrays de posições de todos os outros elementos, e os coloca em uma posição única (não habitada por outros elementos), além de também inicializar incrementos aleatórios nos atributos dos monstros (uma implementação básica de dificuldade). Ela só é executada quando a flag `retryGame` não é ativada, pois quando o jogador escolhe a opção de rejogar, os elementos não mudam de posição.
As funções `spawnMobs()`, `spawnTraps()`, `spawnElixir()` e `spawnBoss()` instanciam de fato os objetos e atribuem suas posições com as posições calculadas na `getSeed()`. Dentro dessas funções, o método `drawSprite()` só é chamado se a flag `debugMode` está ativada, caso contrário, os monstros ficam "invisíveis".
Abaixo segue a implementação de `spawnMobs()`:

```
public void spawnMobs(int mobNum){
  monArr = new Monster[mobNum];
  for(int i = 0; i < mobNum; i++){
    Random rand = new Random();
    monArr[i]  = new Monster(
      base_monATK + monATK,
      base_monDEF + monDEF,
      base_monHP + monHP,
```

```
        mobXPos[i],
        mobYPos[i],
        rand.nextInt(3) //type
      );
      if(debugMode) UI.drawSprite(monArr[i]);
    }
  }
```

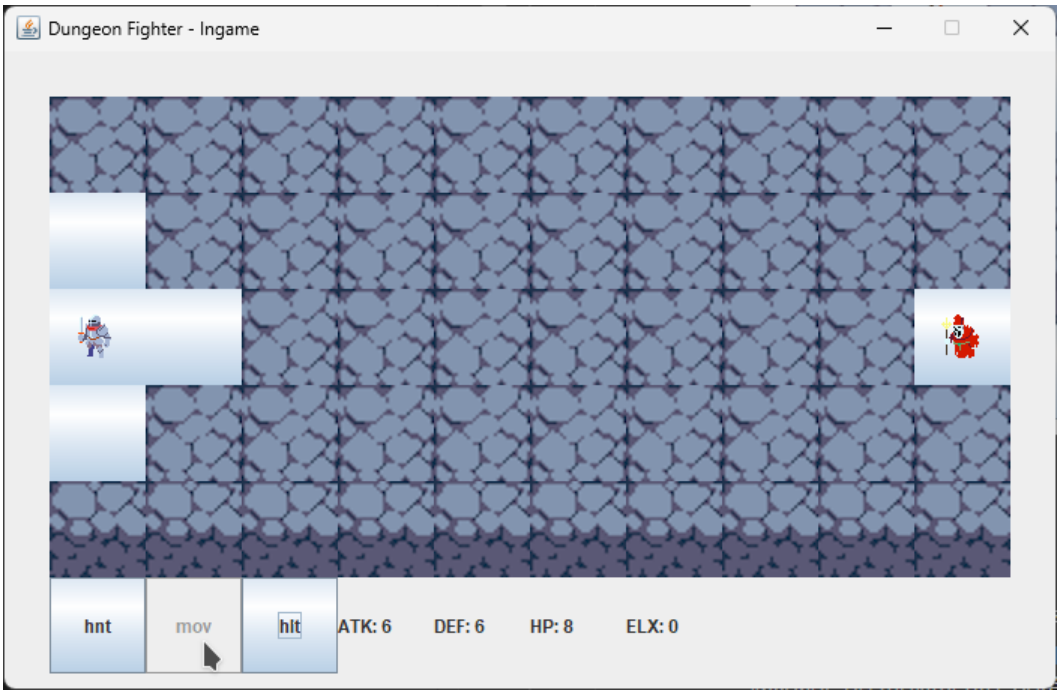
Os outros métodos seguem uma estrutura similar. Abaixo é como o tabuleiro é mostrado, com a flag de debug habilitada.



Movimentação e dificuldades

Com todos os monstros no tabuleiro, agora é a hora de implementar a movimentação do personagem. Primeiramente pensei em deixar todos os botões clicáveis, checando apenas a validade do movimento, porém decidi fazer um sistema onde o jogador clica no comando de movimento e é apresentado os possíveis movimentos, adicionando *ActionListeners* neles. Isso provou ser um pouco mais complicado, porém eu gostei do resultado final.

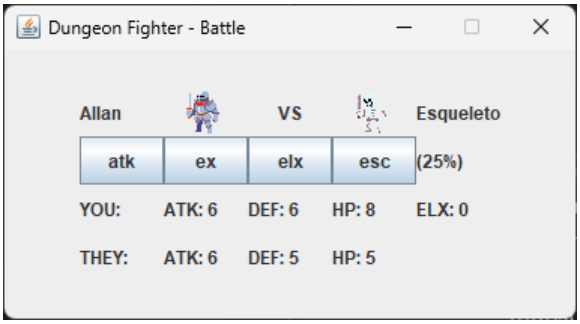
Uma das dificuldades foi detectar quais botões no vetor a função `highlightAvailableMoves()` deveria acessar. No começo, estava acessando botões fora do vetor, o que causava um *ArrayOutOfBounds* exception. Porém, após tratar os *edge cases*, a movimentação se comportou bem. O tratamento desses casos se deu com 4 checagens, cada uma delas se referindo a um canto da tela. Por exemplo: se o personagem está na parede da esquerda (como retratado na imagem abaixo), 3 dos 4 condicionais passam - o que não passa é o que checa se o personagem está na parede da esquerda - portanto apenas 3 movimentos são possíveis, e consequentemente destacados.

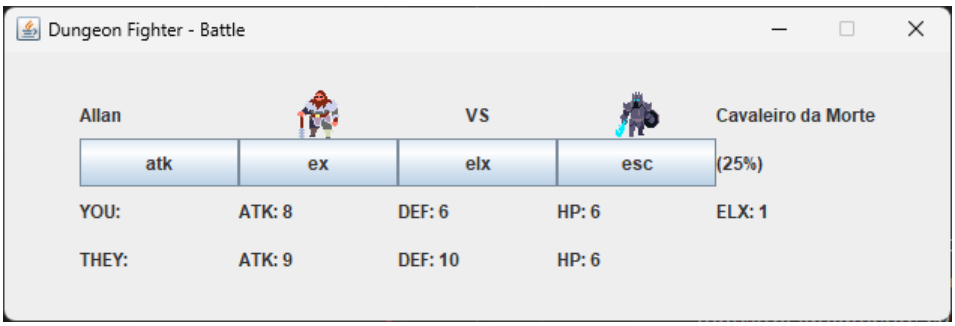


Outra dificuldade no movimento do personagem foi a remoção de *ActionListeners* previamente adicionados. Se o personagem se mover para a direita, por exemplo, as posições de cima e de baixo continuarão com *Listeners* nelas, tornando possível acessar botões fora do vetor. Para contornar esse problema, toda vez que o botão de movimento é pressionado, uma função retira todos os *ActionListeners* do tabuleiro.

Batalha!

A batalha foi simples de implementar: fiz uma função que é chamada quando o personagem se movimenta para uma posição que esteja ocupada com um *Monster*. Nessa função, a tela do tabuleiro é escondida (não chamo o *dispose*, só o *setVisible*) e uma nova tela contendo o herói e o monstro, juntamente com os seus atributos, é criada.
Como cada mob possui seu sprite, o visual da batalha fica diferente a cada encontro.





Aqui, você pode escolher entre atacar, usar a sua habilidade especial, usar elixir e tentar escapar. O combate foi implementado do jeito usual: você rola um número aleatório e soma com o seu ataque, o monstro rola um número aleatório e soma com sua defesa, o número maior ganha, calcula-se a diferença entre os números, que é infligido no que perdeu.
A função também checa se é o chefão ou não, uma vez que, ao derrotar o chefão, o jogo precisa ir para a tela de pós jogo.

Conclusão

Realizar esse trabalho foi uma ótima atividade para mim, pois nunca tinha feito algo em uma escala tão grande. Durante o desenvolvimento do jogo aprendi muito sobre como estruturar um projeto, como aplicar os conceitos de POO em aplicações grandes e como organizar o pensamento para lidar com tantas classes e métodos. Além disso, aprendi muitos conceitos de interface gráfica ao longo do caminho, assim como de Java em geral.