Instituto Federal de Educação, Ciência e Tecnologia do Espírito Santo

PROGRAMAÇÃO ORIENTADA A OBJETOS II

**THE MAZE**

Leonardo Camilo Ribeiro

João Carlos Pandolfi

SERRA

2015

**Sumário**

(i) Diagrama de classe.....................….......................................................................... 3

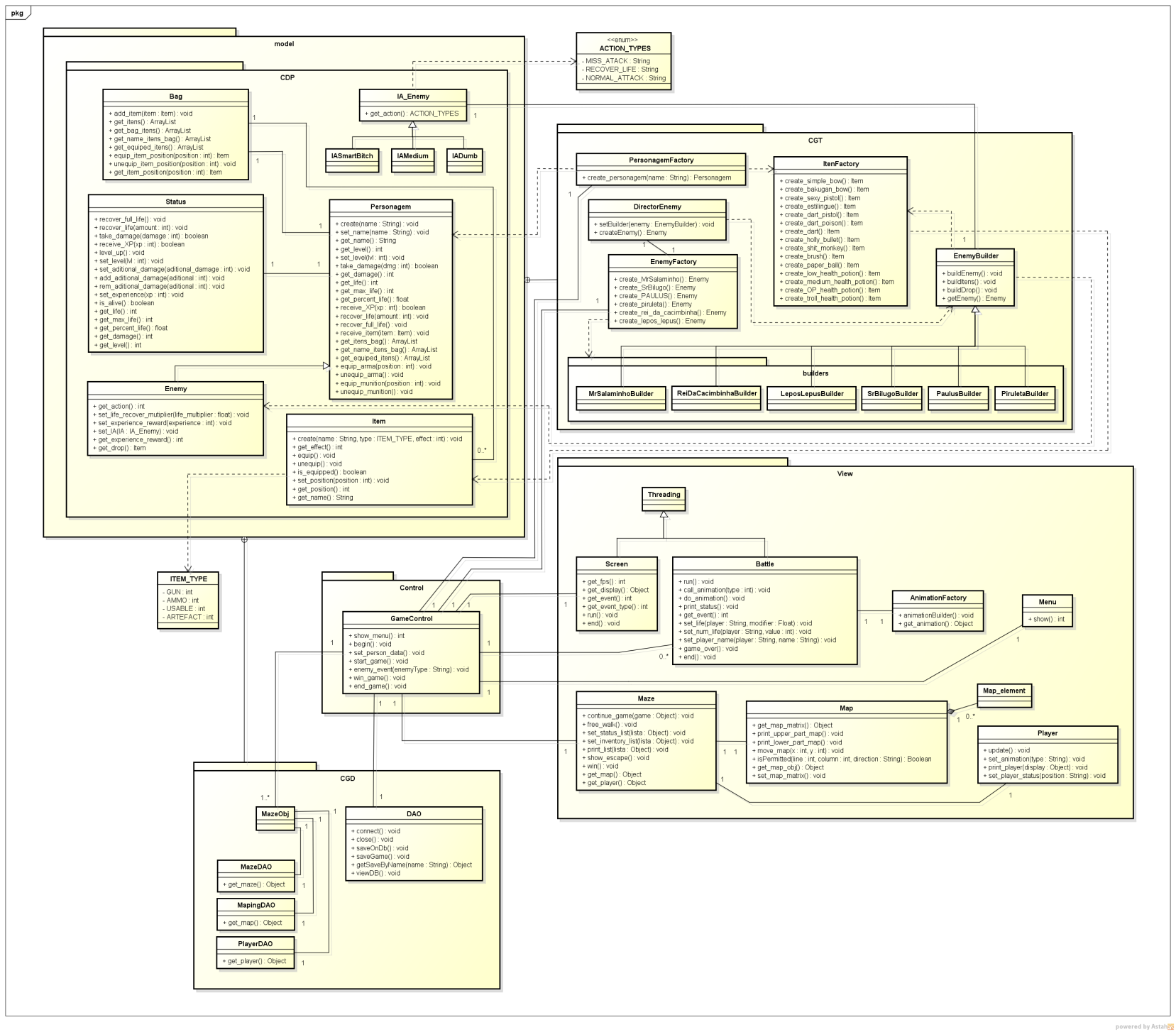
(ii) Explicação das tecnologias utilizada....................................................................... 8

(iii) Explicação do uso de cada padrão ......................................................................... 9

(iv) Resultado do Sonar................................................................................................. 10

(v) Conclusão ............................................................................................................... 12

**Diagrama de Classe:**

****

**Link github para melhor visualização:** [**link**](https://github.com/leoCamilo/The_Maze_Game/blob/master/documentation/Class%20Diagram.png)

Este trabalho foi divido inicialmente em três packages principais, seguindo o modelo MVC abordado posteriormente, o packageview, o control, e o model, onde cada um é responsável por uma área do programa.

1. VIEW:

O View é o package responsável por toda exibição de interface, foi projetada de modo que não possua dependências em outros packages diminuindo o acoplamento e assim melhorando a manutenabilidade e inteligibilidade do programa.

* 1. Screen: É uma das principais classes do view, pois tem como função a exibição de todas as telas das outras classes do view, herda da classe thread e instancia uma janela (display), e o disponibilizar para as outras classes do view essa instancia, cada uma das outras classes gera o seu conteúdo, seja lá ele qual for, e o manda exibir nessa instancia, essas características foram necessárias por conta de dois motivos, o primeiro é que, se cada classe criar sua janela, tornaria o programa confuso, o segundo fato é que como a classe precisa trabalhar o tempo todo por conta dos eventos de teclado, era necessário que roda-se em outra thread, tornando melhorando assim também o acoplamento, pois ao invés de todas as classes precisarem conhecer os comandos de entrada de teclado, somente uma precisa, podendo retornar o resultado pra quem requerer.
  2. Maze: É a classe responsável pela exibição do labirinto, ela recebe em seu construtor uma instancia de tela, gerada pela classe Screen, e é responsável por, exibir o mapa, exibir o player, exibir os inimigos, fazer controle de colisão, entre outras coisas, ela retorna a classe de controle abordada afrente um evento de mapa, como por exemplo, inimigo encontrado ou game over, de modo que o controle tome alguma decisão baseada no retorno.
  3. Map: É um elemento da classe Maze, um mapa é uma instancia visual de algum labirinto do jogo, como os mapas são arquivos de texto, são perfeitamente editáveis, por isso existe uma classe para tratar de algumas funções como, por exemplo, um parser de txt, funções de exibição do mapa (prints), sprites (imagens) que representam cada parte do mapa, etc.
  4. Map\_element: Como o sugestivo nome diz, é um elemento do mapa, um mapa é composto por células, como em uma matriz, cada célula dessas é mapeada para uma instancia de map\_element.
  5. Player: Um player, é a representação visual de um personagem, possui informações visuais básicas como, por exemplo, posição no mapa, quais sprites (imagens) que o desenha e gera suas animações, como se exibir (print), entre outras coisas.
  6. Battle: A classe Battle, assim como a Screen, também herda de thread, isso, pois a batalha é algo muito mais dinâmico que o labirinto, pois o labirinto pode executar enquanto o controle espera, porém, na batalha, as informações referentes a quase todas as ações residem na camada Model do modelo MVC, como por exemplo, o dano do personagem, a experiência de cada monstro, a ação tomada em cada rodada pelo monstro, o funcionamento será explicado mais detalhadamente na classe Controler, porém, ao ser instanciado, uma batalha recebe uma janela de Screen, e como se trata de uma thread, enquanto o loop principal fica exibindo a batalha, o controle chama funções para executar ações visuais na batalha.
  7. AnimationFactory: Esta por sua vez é responsável por retornar um elemento de animação para a classe batalha, isso pois uma animação é uma sequencia de imagens, porém defini-la não é algo complexo, portanto para facilitar na futura possível troca de sprites para animação, e na redução do acoplamento, foi criada essa classe.
  8. Menu: A classe menu tem como objetivo mostrar as opções básicas para o usuário e retornar a sua opção para a classe controle, também recebe uma instancia de Screen para exibir o menu.

1. MODEL:

Este por sua vez, é o maior pacote que existe no software, é composto pelos seguintes sub-pacotes, cdp, cgt, cgd, cada um com um foco especial explicado a seguir.

* 1. CDP:

A camada de domínio do problema, também chamada de cln (camada logica de negocio) é responsável basicamente por conter as regras de negocio do produto, ela não executa nenhuma funcionalidade, porém possui tudo o que é necessário para executa-las:

* + 1. Item: A classe Item é onde se define todas as coisas “usáveis” do programa, tendo um tipo, que define se ele é uma arma, uma munição, uma porção ou um artefato, cada item possui um efeito o qual resulta em alguma melhora ou piora de resultado final dos que o usam.
    2. Bag: A classe Bag é onde todos os itens ficam guardados, uma Bag pode possuir muitos Itens, possui funções como equipar algum determinado item no jogador, retornar o nome de todos os itens, entre outras coisas.
    3. Status: É a classe que define como está o Personagem no exato momento, ela possui atributos como level, vida máxima, vida atual, experiência atual, entre inúmeras outras coisas, sua principal função é adicionar uma característica para o personagem, seja ele o player ou o enemy.
    4. Personagem: É onde se caracteriza um Player ou um Enemy, possui uma Bag, um Status, uma arma, uma munição, que por sua vez são itens, um nome, entre outras coisas.
    5. Enemy: Herda de Personagem, possui algumas funcionalidades a mais, por isso a herança, estas são por exemplo, recompensa de experiência, drop de itens, poder dizer umas sua inteligência, entre outras coisas.
    6. IA\_Enemy: Define como o inimigo deve tomar uma decisão, cada herança sua define o nível de inteligência da decisão visto que é algo probabilístico, por isso não vamos abordas as heranças que são (IADumb, IAMedium, IASmartBitch) pois estas são muito simples, somente alterando índices.
  1. CGT:

A camada de gerencia de tarefas é responsável por entender a camada de domínio do problema e disponibilizar funcionalidades para os outros pacotes.

* + 1. EnemyBuilder: possui os métodos básicos de criação de um builder, esta classe serve de pai para todos os builders de inimigos, estes por sua vez somente implementam os métodos aplicando valores diferentes para cada um, sendo assim não abordados.
    2. ItemFactory: responsável por instanciar cada tipo de item, não sendo necessário definir os mesmos valores toda vez que se quiser instanciar um item do tipo desejado, utilizado no EnemyBuilder e no PersonagemBuilder.
    3. EnemyFactory: Possui os métodos e valores para instanciar cada um dos tipos de inimigos, facilitando assim a criação dos mesmos.
    4. PersonagemFactory: assim como a fabrica de inimigos, tem como função gerar uma instancia de Personagem.
    5. DirectorEnemy: Esta classe tem como função receber um builder de um inimigo e retornar o instancia de Enemy referente aquele builder.
  1. CGD

Este pacote possui classes referentes a persistência de dados, como funcionalidades com banco de dados, salvamento de estado, etc...

* + 1. DAO: esta é a classe operacional do pacote, ela é responsável por todas as funções de comunicação, inserção, recuperação, deleção do banco de dados, possui comandos para criação do banco caso não exista, abrir conexão com o mesmo, entre outras coisas.
    2. PlayerDAO: retorna um objeto de persistência de Player para ser salva no banco.
    3. MapingDAO: retorna um objeto de persistência de Mappara ser salva no banco.
    4. MazeDAO: retorna um objeto de persistência do Labirinto para ser salva no banco.

1. CONTROL

O pacote de controle é o mais simples, possui somente uma classe, devido ao sistema possuir somente um modo de visualização (view) ele possui somente um controler, porém isso não o torna menos complexo.

* 1. GameControl: Esta classe controla todo o fluxo do jogo, ela inicia exibindo um menu do packageview, ao retornar a opção escolhida, a partir disso existem três possibilidades, sair do jogo, continuar um jogo, ou começar um novo, ao continuar, é feito uma chamada da classe DAO para recuperação dos dados e a partir disso ser repassado para a classe Maze e continuar o jogo, caso contrario é instanciado um novo Maze, ao ser retornado um evento de mapa a classe encaminha para seu determinado tratamento, por exemplo em uma batalha, dependendo do tipo de inimigo encontrado é instanciado um novo enemy e é feito um loop para se definir o desfeche da batalha, chamando funções da camada CGT para pegar os dados como por exemplo o dano dado pelo personagem e fazer o inimigo recebe-lo, e por exemplo chamar animações da classe Battle.

**Explicação das tecnologias utilizadas:**

**PYTHON**

Para o desenvolvimentodesse jogo utilizamos a linguagem de programação Python. Essa é uma linguagem Multiparadigma, ou seja, suporta o paradigma orientado a objetos, imperativo, funcional e procedural, assim como pode ser tanto interpretada como linguagens como javascript, php, etc... pode também se gerar um arquivo com extensão .pycpré-compilado para acelerar a interpretação, assim como o Java.

Python não possui separados de blocos convencionais como por exemplo o “;”, ao invés disso utiliza a própria indentação como separador de escopo, forçando assim uma melhor legibilidade do código, pois prioriza a clareza sobre o desempenho, é uma linguagem fracamente tipada, sendo que todas as variáveis são objetos, prejudicando ainda mais seu desempenho, porém é bastante prática, podendo se fazer coisas em poucas linhas, que em outras linguagem precisariam muito mais.

**PyGame**

O framework utilizado para o desenvolvimento do trabalho foi o pygame, este por sua vez é uma biblioteca exclusiva para python que tem como objetivo facilitar o desenvolvimento de jogo para python pois incorpora funções que de outra maneira seriam complexas, ou até mesmo demoradas de se fazer, agilizando o processo e tirando o foco de sua preocupação de partes já bem consolidadas, possui funções básicas como exibição de imagens na tela, captura de eventos de teclado, controla de frame\_rate, entre outras coisas.

**Unittest**

Para a execução de testes automatizados foi utilizado o framework do unittest, já incorporado ao python por padrão, não possui saida visual porém gera uma porentagem de acertos e erros, exibida no terminal.

**SQLite**

Para a camada de persistencia foi utilizada a framework sqlite também incorporada ao python, se trata de uma classe gerenciadora de banco de dados, não possui comandos complexos como os outros bancos, porém como a aplicação é simples, e ele é rapido por possuir poucas funcionalidades, tornando seu uso estremamente viavél.

**Explicação do uso de cada padrão:**

**Fábrica Abstrata:** O padrão Fábrica Abstrata foi utilizado em varias partes do código para facilitar a manutenabilidade e a legibilidade, ele faz isso agrupando todas as instanciações de um mesmo tipo de objeto dentro de uma classe, fazendo assim que, caso se deseje adicionar ou remover uma dessas instancias, não será tão complexo o refatoramento do código, outro motivo de se usa-lo é a padronização de instancias, pois assim só existira objetos que nós definirmos nas fabricas.

**Diretor e Builder:** O padrão Diretor foi aplicado para criação de inimigos, pois, ao utilizar este padrão ao criar um novo inimigo, precisa-se somente criar um builder, a classe que possui comportamento e peculiaridades para criação do mesmo, isto faz com que seja facilitada a manutenção e a criação de novos inimigos com dados diferentes. O diretor, sabe usar o builder de forma que qualquer enemyBuilder inserido ele saberá criar o inimigo com suas respectivas características.

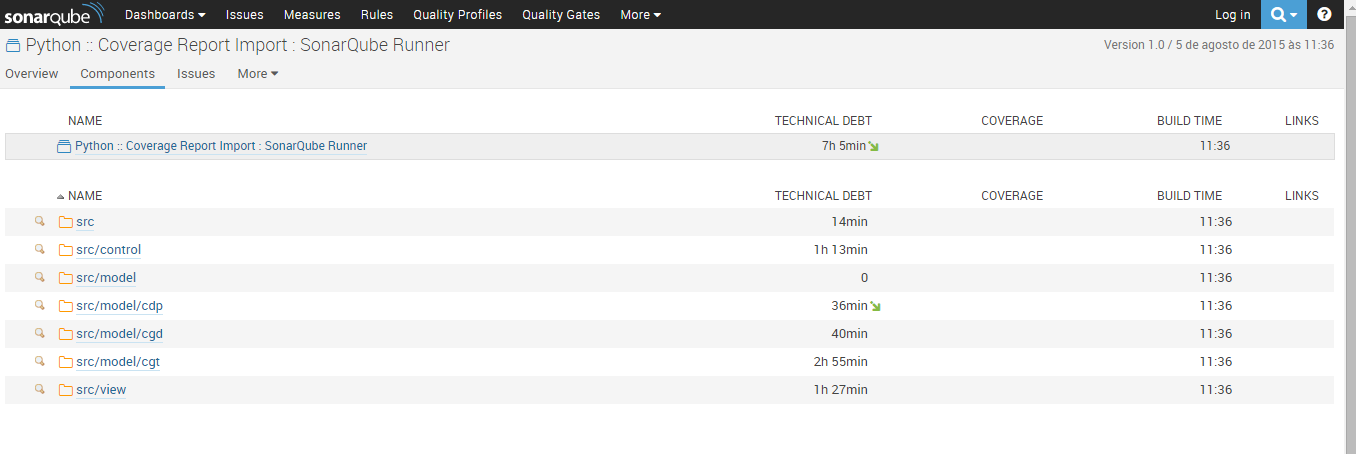
**DAO:** Utilizamos o padrão DAO para separar a camada de persistência da camada de domínio do problema, isso tem como vantagem o desacoplamento do código e a melhora da manutenabilidade, de modo que, por exemplo, seja necessária à troca do sistema de arquivos, não impacte no restante do software, nesse padrão fizemos uma classe genérica DAO que possui funções básicas de persistência como salvar, recuperar, etc... E fizemos uma DAO para cada um dos dois elementos que compõe um labirinto, o mapa, e o player, a classe mappingDAO sabe, ao receber um objeto do mapa, instanciar um objeto de persistência de mapa, o mesmo ocorre com o PlayerDAO, assim, com esses dois objetos, compõe-se o objeto MazeDAO, este por sua vez é passado para a classe genérica que realiza o trabalho de persistir o dado.

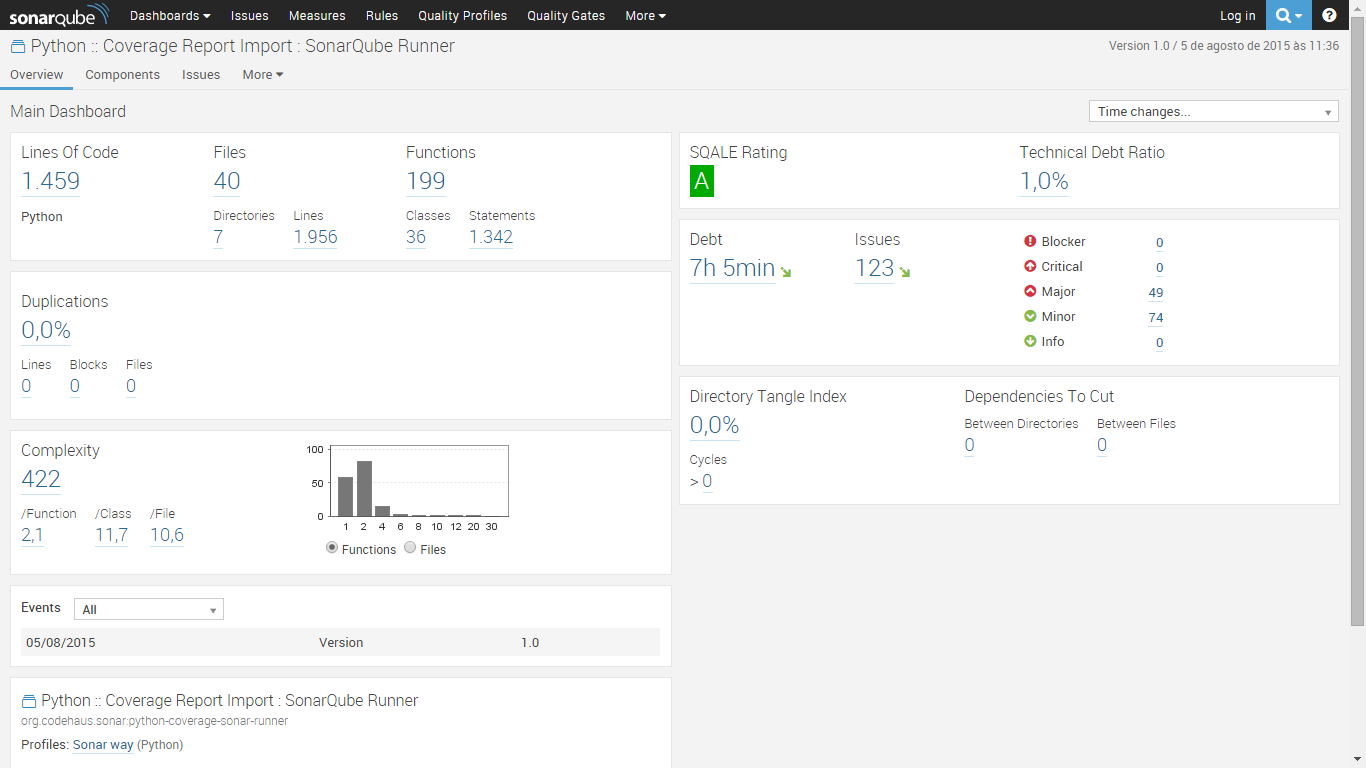
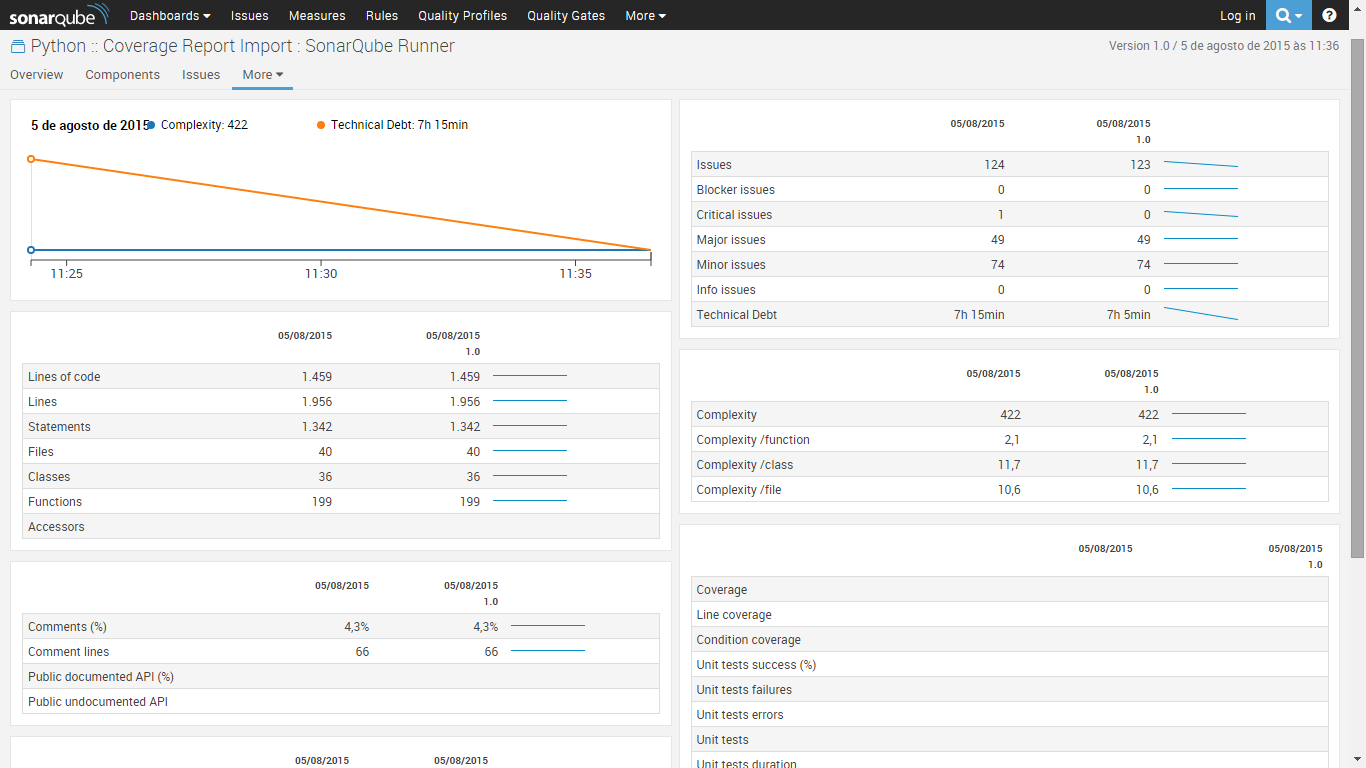
**Estratégia:** O padrão strategy foi usado no comportamento dos monstros, foi criada uma classe IA, onde dela herdam três tipos de IA, a burra, a mediana, e a inteligente, atualmente cada uma calcula a ação baseada em probabilidade simples, portanto não fazendo uso total do comportamento, porém, caso seja requerido alguma mudança, a alteração não seria tão complexa, por exemplo, deseja que a IA inteligente retorne a ação, porém ao invés de probabilidade simples ela faça uma analise em arvore do status do player e veja qual o movimento que mais favorece ela segundo os pesos definidos pelo programador, para isso somente é necessária o override do método get\_action() na classe filha. De modo que não seja necessárioa alteração nas demais classes.

**Template Method:**Este padrão foi utilizado na inteligência dos monstros, cada monstro tem uma probabilidade de errar o ataque, recuperar vida, ou acertar o ataque, o calculo dessa probabilidade é sempre o mesmo, porém a única coisa que muda são os índices, fazendo-se assim um perfeito caso para uso do método template, melhorando em dois quesitos, o primeiro é que não é necessário a repetição de código, o segundo que como a função só foi escrita uma vez elimina o fator de erro de escrita caso fosse necessário a cópia dela para as outras classe, sem falar que caso seja necessária uma alteração dessa função, como por exemplo inserção de uma nova ação, é feito de uma maneira muito mais pratica, pois só altera um lugar.

**MVC:** O padrão MVC foi utilizado pois ele consegue separar muito bem todas as partes envolvidas no sistema, ajudando por exemplo o trabalho em equipe, onde uma camada do MVC só precisa saber como vai ser a saída ou entrada da outra, podendo assim se desenvolver paralelamente, além do mais, ele permite uma manutenção mais prática pois, como as camadas são bem definidas, os erros não geram uma arvore de problemas tão grande, sendo fácil a sua localização.

**Resultado do Sonar:**





Relatorio sonar:

A complexidade. O ideal é 1.0, levando em conta que a complexidade das funções atingiu uma média de 2.1, significa que o código está em uma complexidade aceitável, porém não quer dizer que não possa ser melhorado.

O Rating: A nota alcançada foi A, simbolizando que, levando em conta todos todos o indices, a qualidade do software, no que tange a qualidade técnica, está boa, não excluindo a melhora, é claro.

technical debt Ratio

Foi alcançado uma taxa de 1.0%, significa dizer que a taxa de "erro" (pois erro não é necessáriamente um problema, porém uma falha de clean code, uma função muito complexa, entre outras coisas) no sistema atinge 1.0% de todo o código, analizando os erros apresentados, encontramos 49 major erros, no qual realmente, somente 22 deles são possíveis erros, os outros 27 são artifícios do python que o sonar não identifica muito bem.

Codigo duplicado

Foi alcançada uma taxa de 0% de codigo duplicado, o que significa dizer que não existe nenhuma linha repetição que possa ser modularizada.

technical debit

Tendo em vista que o sonar define um tempo fixo para cada tipo determinado de erro, ao contabilizar todos os encontrados, ele Pressupõeo tempo estimado para correção de todos os erros em cada pacote, que foi de aproximadamente 8 horas, que é considerado um tempo aceitável.

**Conclusão**:

O uso de padrões apesar de parecer um “desperdicio” de codigo a primeira vista, ajuda bastante no que diz respeito a organização de codigo, graças a isso foi posivel a criação de um “grande” projeto, e a facilitar a manutenção, um exemplo disso foi que foram feitas três atualizações no jogo, sendo que o trabalho para alterações entre cada uma foi cada vez menor, mostrando que era possivel fazer a manutenção de modo pratico e sem complicações, melhorando tambem a capacidade do software de evoluir.

Link para o cod. Fonte do trabalho: <https://github.com/leoCamilo/The_Maze_Game>