**MUMBRA**

**The Game Design Document**

|  |  |  |
| --- | --- | --- |
| Luiz Fernando Rodrigues da Fonseca | RA: 156475 | Equipe F.I.R.S.T.:  Módulo Mob |
| Lucas Alves Racoci | RA: 156331 |
| Isadora Sophia Garcia Rodoupolos | RA: 158018 | Equipe Them Crooked Vultures  Módulo BattleManager |
| Matheus Mortatti Diamantino | RA: 156740 |
| Lauro Cruz e Souza | RA: 156175 | Equipe Centro-Oeste Represent:  Módulo Célula |
| Lucas Silva Schanner | RA: 156412 |
| Henrrique Noronha Facioli | RA:157986 | Equipe Megazorde:  Módulo Mapa |
| Thiago Silva de Farias | RA: 148077 |
| Felipe Viglioni | RA:106665 | Equipe Torchwood:  Módulo Manager |
| Gabriel Siqueira | RA: 155446 |
| Daniel Cunha | RA:140561 | Equipe BDouble:  Módulo Puzzle |
| Victor Hugo Barreiro | RA: 118876 |

**Conceito:**

O jogo se passa em um longo corredor que se estende através da escuridão. Um vulto persegue o personagem vagarosamente, fazendo com que sua visão se embaralhe. Ao longo do corredor existem monstros que ficam mais fortes quanto mais perto do fim do corredor o personagem fica. O objetivo principal é chegar ao fim intacto.

O corredor não é contínuo, fazendo com que o jogador tenha que entrar nas salas que existem ao longo dele para resolver puzzles que permitem que o jogador avance. Dentro dessas salas não existem monstros, contudo o vulto pode alcançar o jogador.

**Mecânicas:**

**Combate:** assim que o jogador encontra um monstro, o jogo entra em Battle Mode, dando uma rápida descrição do oponente e uma lista de itens que o jogador pode escolher e usar como arma única durante a batalha. Assim que essa fase passa, a batalha em si começa.

A batalha é separada por turnos, nos quais o jogador e o monstro alternam ataques. O jogador deve escolher uma parte do corpo do monstro que quer atacar, porém quanto maior o dano causado nessa parte, menor a chance do jogador acertar o ataque.

**Puzzles:** Dentro das salas, haverão diversos puzzles à serem resolvidos. Para isso, será necessário interagir com objetos na cena, bem como coletar itens para uso na mesma ou em outras salas.

**Corredor:** única parte visual do jogo, onde aparecerá um mapa com a posição do jogador e um campo de visão que pode aumentar se ele estiver utilizando um item que emana luz. Ao longo do corredor haverão monstros espalhados que se aproximam do jogador a cada movimento que este faz, tendo uma chance dele desviar caso não queira entrar em combate.

**Salas:** em cada uma das salas, além dos puzzles, poderá ter um conjunto de itens para o jogador achar. Tais itens podem ser descartáveis (fósforos, velas) ou fixos (lanterna, armas em geral)

**Vulto:** o vulto aproximará do jogador a cada movimento feito, mas pode ser retardado quando o personagem está segurando algum tipo de item que emite luz (fósforo, vela, lanterna, etc). O tempo que ele demora para alcançar o jogador dependerá do nível escolhido pro jogo e do progresso do jogador ao longo do corredor.

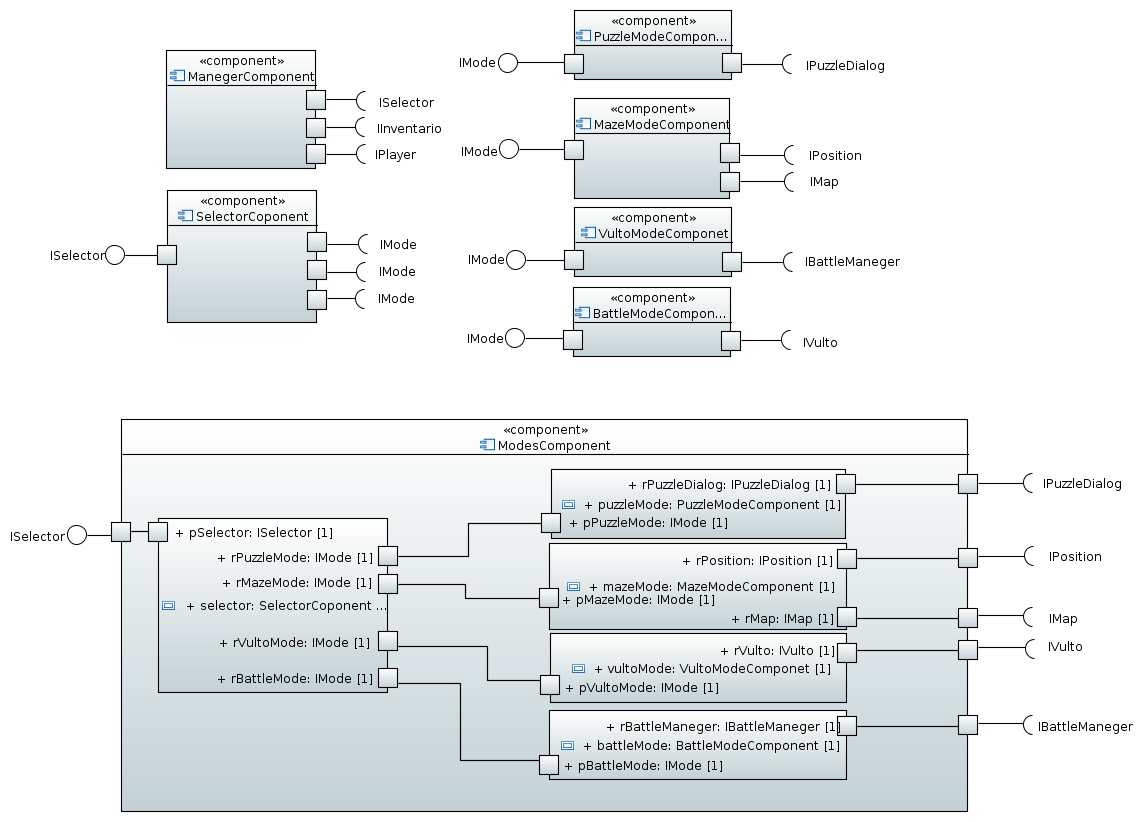
**Jogador:** o jogador terá um número de 'Stats' que compõem seu personagem. Estes são: Strength, Luck, Insanity, Current HP, Max HP, Dexterity, Speed, Attack, Defense e Evasiveness. Dessas, apenas Insanity não dependerá do level do jogador.

**Módulos:**

**Main Game**:

Responsável pelo gerenciamento do jogo, esse módulo irá se encarregar de realizar as principais comunicações entre os demais e a comunicação final com o usuário, bem como gerenciar os principais recursos da framework libgdx utilizada.

A classe principal requer as interfaces IInventario, IPlayer e ISelector. Esse módulo, porém, apresenta o componente Modulos, que irá prover a interface ISelector e requer as interfaces IPuzzleDialog, IPosition, IMap, IVulto e IBattleManeger.



**Princípios de Projeto utilizados:**

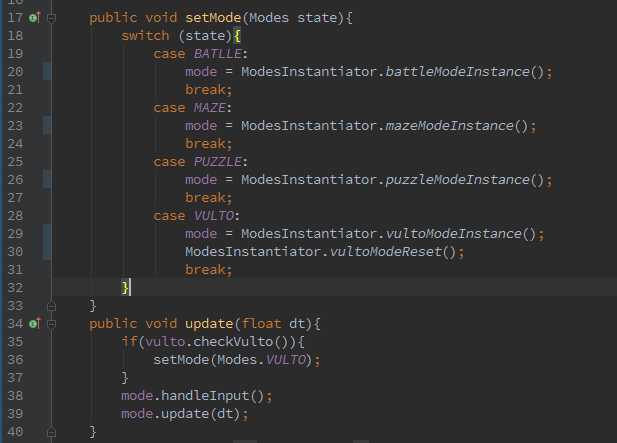
Dependency Inversion Principle: Toda a comunicação do módulo é feita através de interfaces, tendo objetos que, mesmo se comportando diferente internamente, oferecem os mesmos métodos com um comportamento genérico.

Open Closed Principle: Novos modos podem ser acrescentados sem que haja a necessidade de alterar os demais.

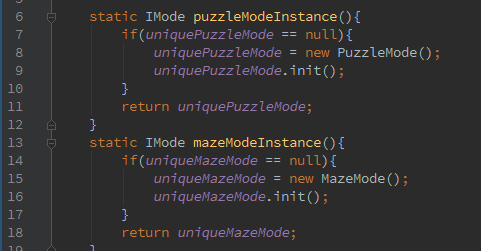
Interface Segregation Principle: Cada interface carrega pouca responsabilidade, como por exemplo há uma interface apenas para enviar as informações para o usuário, outra apenas para selecionar e passar informações para o modo correto e assim por diante.

**Padrões de Projeto utilizados:**

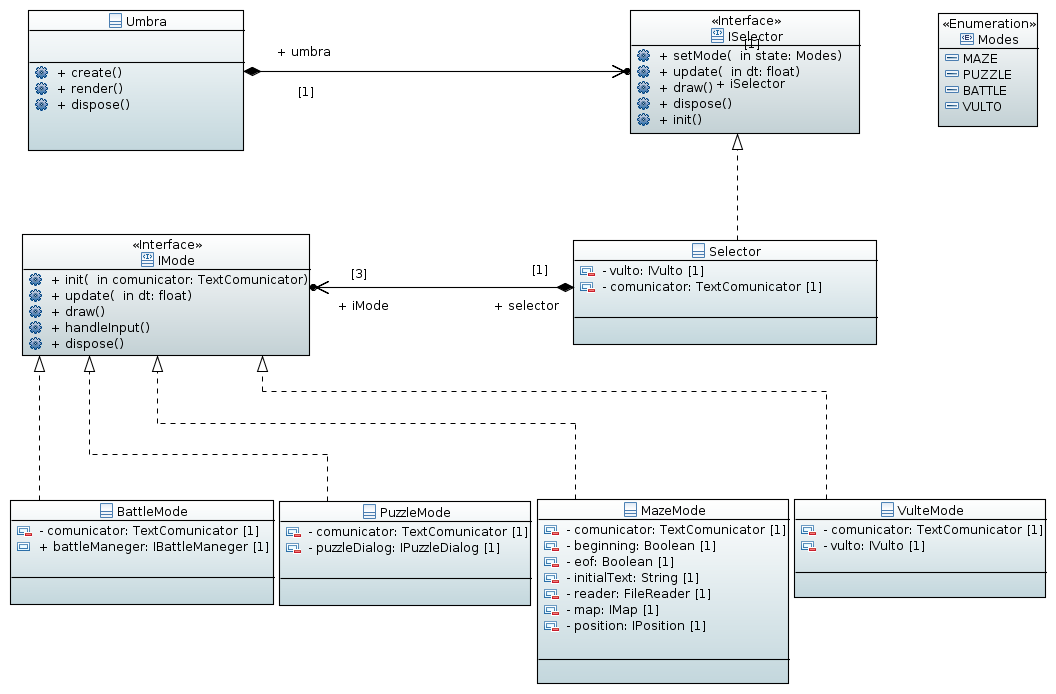
State: O padrão State é adotado para alterar o comportamento do jogo de acordo com seu modo. Isso é feito ao trocar a instância de modo em Selector.

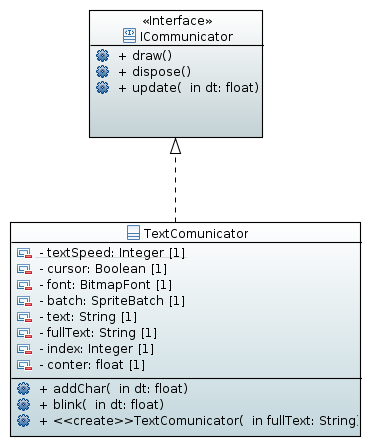


Singleton: O padrão Singleton é utilizado para se garantir que só haja uma única instância de cada modo.



**Diagrama de classes:**





**Sub componentes:**

SelectorComponent: Componente responsável pelo controle dos modos de jogo e suas interações.

Interfaces:

* Provida: ISelector
* Requeridas: Um IMode para cada modo do jogo (no caso quatro IModes).

PuzzleModeComponent: Cuida da comunicação com o componente Puzzle.

Interfaces:

* Provida: IMode.
* Requerida: IPuzzleDialog.

MazeModeComponent: Cuida da comunicação com o componente Map

Interfaces:

* Provida: IMode.
* Requerida: IPosition, IMap.

VultoModeComponent:Cuida da comunicação com o componente Vulto.

Interfaces:

* Provida: IMode.
* Requerida: IVulto.

BattleModeComponent:Cuida da comunicação com o componente Battle.

Interfaces:

* Provida: IMode.
* Requerida: IBattleManeger.

**Interfaces:**

ISelector: interface para seleção e gerenciamento do módulo correto, além da transição entre modos.

Métodos:

* init: chamado apenas uma vez, na inicialização. Inicializa os atributos e chama setMode para o modo Maze.
* setMode: recebe o modo a ser selecionado e muda para esse modo.
* update: chamado a cada atualização do jogo, chama update e handleInput do modo selecionado.
* draw: chamado a cada atualização do jogo, chama draw do modo selecionado.
* dispose: chamado apenas uma vez, quando o selector não for mais utilizado. Descarta o modo atual.

IMode: Interface de comunicação com os modos do jogo.

Métodos:

* init: chamado na inicialização do modo, inicializa os atributos e realiza outras configurações iniciais que sejam necessárias.
* update: chamado a cada atualização do jogo, é responsável por atualizar o valor dos atributos e envia e recebe informações necessárias dos demais componentes.
* draw: chamado a cada atualização do jogo, é responsável por atualizar a imagem da tela vista pelo usuário.
* handleInput: Verifica a entrada fornecida pelo usuário e a redireciona para o componente que irá utilizá-la.
* dispose: chamado apenas uma vez, quando o modo não for mais utilizado. Descarta os atributos que precisam ser descartados e realiza outras ações finais.

**Implementação interna dos componentes:**

IComunicator: Classes que implementam esta interface são responsáveis pela comunicação direta com o usuário:

Métodos:

* update: atualiza a mensagem a ser impressa, retorna verdadeiro quando a mensagem estiver completa.
* draw: imprime a mensagem.
* dispose: para o descarte da instância.

TextCommunication: Implementa IComunicator, usado para imprimir textos.

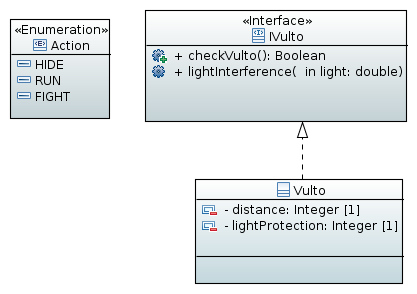
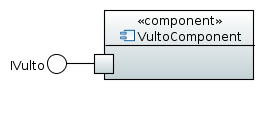
Métodos internos:

* addChar: adiciona o próximo caractere da mensagem.
* blink: faz o cursor piscar alterando o estado dele para apagando ou aparecendo.

ModesInstanciator: Responsável por instanciar os modos e os reiniciar quando necessário.

**Vulto**

Esta módulo é responsável por guardar o estado do vulto e simular as alterações que ocorram nele no decorrer do jogo.



**Princípios de Projeto utilizados:**

Common Reuse Principle: Embora este componente seja muito simples, sua independência em relação ao resto do programa fez com que ele seja mantido separado.

**Padrões de Projeto utilizados:**

Singleton: Garante que haja apenas uma instância de vulto.

**Interfaces:**

IVulto: Interface para controle do vulto e ações relacionadas a ele.

Métodos:

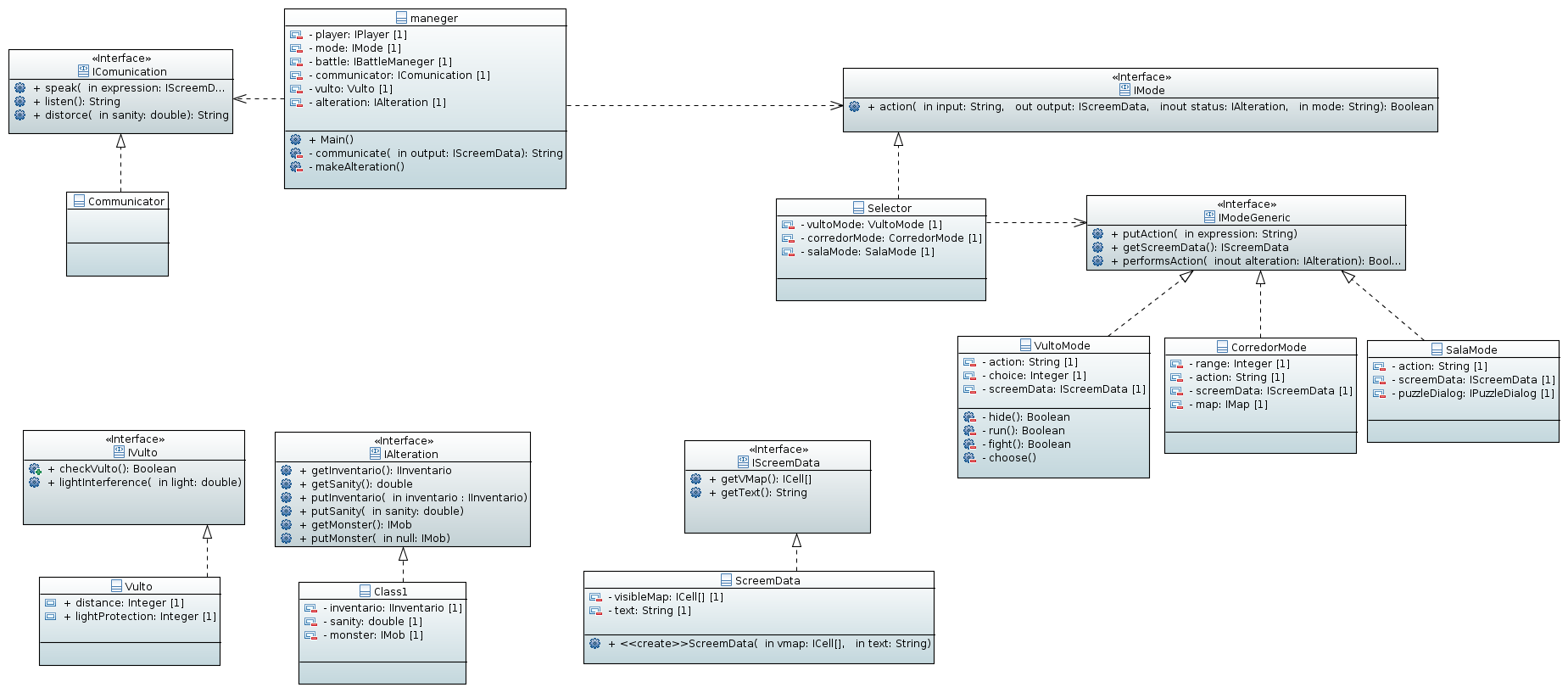
* checkVulto: verifica se o vulto alcançou o jogador.
* lightInterference: aplica a interferência da luz originada de determinados itens do jogador na posição do vulto.
* chooseAction: chamado quando o vulto alcança o jogador, tem como parâmetro a ação escolhida pelo usuário.

**Implementação interna do component:**

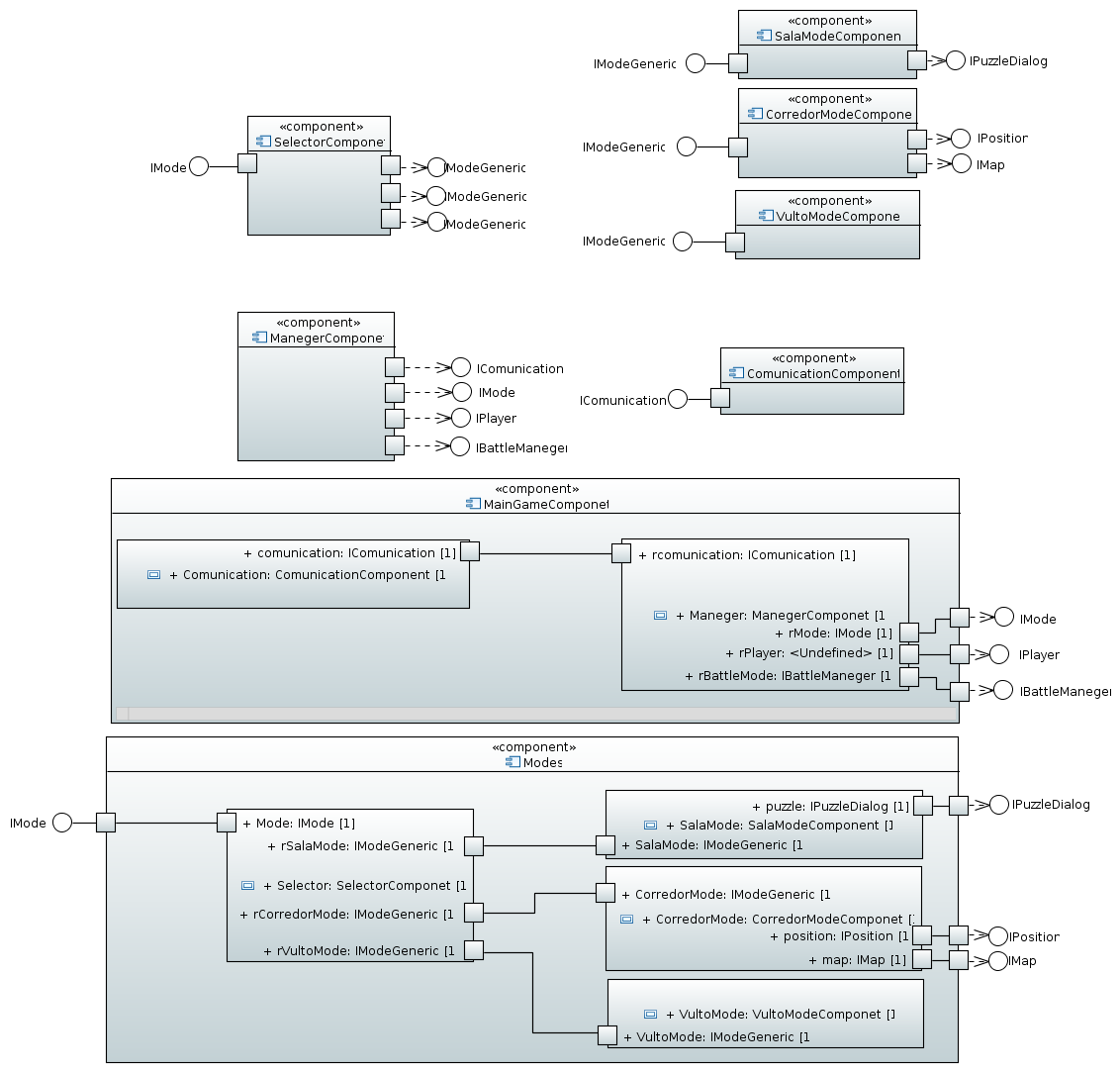
VultoSingleton: Classe responsável por instanciar Vulto.

**Evolução Main Game e Vulto**

**Antigo diagrama de classes:**



**Antigo diagrama de componentes:**



**Principais alterações:**

Adoção do Padrão State:

Na versão anterior uma mudança de modo do jogo consistia apenas em alterar a classe responsável por se comunicar com os outros componentes, deste modo ficava a cargo do selector e do manager checar constantemente o modo atual e então realizar as ações relativas a ele.

Na nova versão, cada modo é um estado diferente do selector, desta forma para alterar o modo do jogo basta mudar o modo instanciado em selector. Assim todo o comportamento do jogo se adéqua ao novo modo.

Separação do componente Vulto:

Na versão anterior, o VultoModeComponente era responsável integralmente pelo vulto, diferente dos outros Modes, que apenas se encarregavam de gerenciar os componentes que implementam os demais objetos do jogo. Na nova versão existe um componente separado para simular o vulto.

Criação do BattleModeComponent:

Na versão anterior não havia um componente interno do Main Game para conectar o Componete Battle com o restante do jogo, desta forma o Componete Battle ficava com responsabilidades que deveriam pertencer ao MainGame, como por exemplo se comunicar com o usuário.

Na nova versão, criou-se o componente BattleModeComponent, responsável por gerenciar o componente Battle.

Alteração na abordagem do Comunicator:

Na versão anterior, a comunicação com o usuário consistia em reunir os dados gerados por cada modo em um único objeto e os levar ao Comunicator, onde eles eram processados e a comunicação com o usuário era efetuada.

Na nova versão, os objetos que implementam IComunicator são capazes de se comunicar com o usuário e os modos possuem uma instância própria para realizar a comunicação.

**MapModule**

O MapModule foi originalmente formado pelo Henrique Noronha Facioli e pelo Thiago Silva de Farias e tinha como objetivo o gerenciamento de todos os eventos que ocorrem no mapa do jogo (Corredor). O MapModule deve criar uma matriz de iCells, gerenciar o que acontece e o alcance de visão do jogador no mapa.

Inicialmente ele deve posicionar os iPuzzles, utilizando um algorítimo aleatório (Java.Random) nas paredes do mapa, e garantir que elas sempre estejam no mesmo local. Deve, também, posicionar os iMob's presentes no jogo (monstros) de uma maneira aleatória, instanciando eles para aumentar a dificuldade conforme a posição da celula na matriz e, por fim, posicionar o jogador principal em alguma célula da matriz definida estaticamente.

Para que seja garantido que o mapa seja o mesmo durante uma partida, decidimos usar como design pattern o Singleton, fazendo assim que, mesmo que o jogador entre em algum modo de jogo que não o corredor, quando voltar a ele, volte todas as configurações do jeito que ficaram e também não tenha perigo de se instanciar duas vezes o mapa.

Durante o jogo, o MapModule deve gerenciar as colisões entre iMob's e entre iMob e iRoom. Caso dois iMobs se encontrem ele deve verificar se são monstros, e se forem apenas ignorar; Caso seja o personagem principal e um monstro, deve retornar a string de "BattleMode" para que o ManagerModule possa acionar este módulo; Caso seja um iMob e iRoom, deve verificar se é o personagem principal e se sim, retornar a string "RoomMode" para que o ManagerModule possa acionar o módulo necessário.

**Especificações Interface iMap**

* ●  O método “move” recebe um ‘iMob’ e uma direção na forma de ‘char’, e retorna um ‘boolean’. Assim, caso o ‘iMob’ seja movido na direção indicada com sucesso, retornamos ‘true’, e, caso contrário, retornamos ‘false’.
* ●  O método “getMap” retorna, a partir do campo de visão do personagem, uma string contendo o mapa que é possível observar a partir do ponto em que o jogador está.  **Classe Map**
* ●  Os atributos da classe Map são: ‘TAM\_X’ e ‘TAM\_Y’, que são do tipo ‘static int’, com a finalidade de determinar o tamanho do mapa; ‘instance’, que é do tipo ‘Map’ e é utilizada no método de ‘Singleton’ especificado abaixo; uma matriz de ‘iCells’, que representa o corredor.
* ●  Os métodos “move” e “getMap” simplesmente implementam os métodos da interface, especificados acima.
* ●  O método “getInstance” é um ‘Singleton’, garantindo que o mapa será implementado apenas uma vez. Ele recebe um ‘iMob’, que representa o ‘player’, e, caso o mapa ainda não tenha sido instanciado, cria um mapa com o construtor especificado abaixo. Caso contrário, ele simplesmente retorna o mapa que já havia sido criado.
* ●  O construtor do mapa, ‘Map’, é do tipo privado. Ele recebe um ‘iMob’, que representa o ‘player’, e então ele o posiciona, além de determinar posições para os ‘iPuzzles’ e os montros, no formato ‘iMob’. Em ambos os casos, a escolha para as posições será randômica, portanto fazemos uso do ‘Java.Random’.

**CellModule**

O CellModule foi originalmente criado separado do MapModule e o grupo encarregado é o do Lucas Silva Schanner e do Lauro Cruz e Souza. Com o andamento do projeto percebemos que o CellModule e o MapModule eram intrínsecos e não dava para separar, e portanto acabamos unindo os grupos. O CellModule deve conter as células e as posições.

As células podem armazenar em si um jogador ou um monstro (Mob) ou uma porta (Puzzle). Cabe a célula realizar as operações de retirar e adicionar em si mesma esses objetos.

**Especificações**

**Classe Cell**

* ●  Método removeMob: retira o Mob presente na célula (se houver um) e retorna esse mob retirado.
* ●  Método setMob: coloca um Mob na celula (player ou monstro) e retorna true se foi possível colocá­lo e false se não foi (ou seja, já há um mob ou uma porta na célula)
* ●  Método getMob: Retorna o Mob presente na célula, retorna null caso não tenha nenhum.
* ●  Método getDoor: Retorna a porta IPuzzle presente na célula. Retorna null caso nao tenha nenhuma.
* ●  Método setParede: Define se a célula é uma parede, recebendo como parâmetro um boolean.
* ●  Método getParede: retorna se a célula e ou nao uma parede (true ou false).
* ●  Método getDescription: retorna um caracter que representa o que existe naquela célula. Usado para imprimir o mapa na tela.
* ●  Atributos privados que armazenam o Mob (IMob), a porta (IPuzzle) e se é ou não parede (boolean).  **bdModule**

Com a união do CellModule e do MapModule, separamos os grupos e decidimos criar o bdModule que ainda está em fase embrionária. Ele será responsável pela criação, leitura e remoção de informações em algum banco de

dados. Estamos estudando a possibilidade de usar JDBC com mySQL ou criarmos o nosso próprio utilizando algum formato próximo do CSV.

# BattleModule:

# O cluster “BattleModule” é responsável por realizar toda a administração e desenvolvimento que envolve o modo batalha do jogo. Segue a modelagem em UML de componentes:

# Macintosh HD:Users:MatheusDiamantino:Downloads:UMBRA-master:Documentation:Módulo 3 - BattleModule:New:ComponentDiagram.png

# Como pode-se notar, o componente é formado basicamente por: uma classe estática “BattleExecuter” e uma classe principal “BattleModule", provida de uma interface IBattleModule. A comunicação entre o componente e o resto do jogo se dá pelo pattern Model-view-controller (MVC), a partir da chamada de funções pelo controller (no caso, o cluster GameManager ficará responsável por isso), que renderizará as informações providas pelo BattleModule (que fará o papel de Model) na tela (View). O pattern foi adotado para manter a organização no jogo, garantindo, também, a segmentação dos clusters.

# Segue a modelagem de classes, que propõem uma noção mais detalhada do módulo:

# Macintosh HD:Users:MatheusDiamantino:Downloads:UMBRA-master:Documentation:Módulo 3 - BattleModule:New:ClassDiagram.png

# O IBattleManager, interface para o BattleManager, compõe o básico para que o módulo possa ser executado: uma função initialize (IPlayer player, Imonstro monstro), responsável por inicializar o componente ao início de uma batalha - os parâmetros das funções são interfaces requeridas a partir de outro cluster, responsável pela organização de entidades no jogo. A outra função, processInput (String input), é responsável por processar um determinado input e processar um novo output para a tela. Como a mecânica do jogo se baseia em inputs e outputs, é uma forma viável de manter o workflow do jogo.

# O restante do BattleManager se trata da especificação de funções e funcionalidades, em que há a realização do encapsulamento: as únicas funções que o restante dos clusters poderão ter acesso serão, além das providas na interface, a função que resgata o valor do output (getStatus()) e uma booleana que devolve se o jogo foi finalizado (getDone()).

# O BattleExecuter é uma classe auxiliar para o BattleManager: se trata das questões mais técnicas quanto ao desenvolvimento da batalha. Embora o princípio de Singleton também pareça fazer sentido nesse contexto, foi decidido pela equipe realizar a classe como estática, uma vez que o acesso a ela se restringiria à realização de funções - não haveria necessidade de manter propriedades e outras especificações da batalha, sendo o uso de Singleton indiferente. Assim, a classe é estática e seu acesso se restringe aos módulos do componente (no caso, BattleManager).

# Outro design pattern importante para o planejamento do cluster foi o Observer, utilizado na interação do BattleManager com o BattleExecuter, para relacionar informações relevantes na batalha: se ela foi finalizada e se o jogador perdeu ou ganhou. A vantagem do pattern também se deu para prezar a organização ao componente.

# O plano de execeções, até então, consta em validar inputs feito pelo jogador. Caso ela seja inválida no contexto do jogo, o input é descartado e é reinicializado o último comando do jogo.

# O diagrama abaixo apresenta o antigo plano de modelagem do jogo, em que constava por interfaces distintas e funções ainda vagas, uma vez que o plano do MVC ainda não havia sido adotado e o aprofundamento do jogo ainda estava a ser realizado. O plano de interfaces continua basicamente o mesmo, a principal diferença exterior foi a perda de acesso do BattleExecuter: sem a necessidade de uma interface e visível apenas ao BattleManager. A mudança ocorreu bem para a organização e melhor visão do andamento de como jogo ocorrerá, e é provável que mais dessas mudanças ocorram ao longo da finalização do projeto.

# 

MobModule

O módulo MobModule fica responsável por prover interfaces relativas ao uso de um Mob, de um Item ou de Attribute conforme será especificado a seguir.

Plano de Interfaces

Interfaces Providas:

Todos os componentes e subcomponentes, exceto o inventory, têm uma interface Manager que será exportada, contendo métodos de administração geral do componente. Além disso, as interfaces que são exportadas por todo o módulo são as dos componentes: Componentes:

1.De mobComponent: 1.1. IMobManager:

Contém um método para criar um único Player, já que este é um Singleton, e outro para criar Monstros, retornando suas instâncias.

1.2. IMob:

Possui métodos que serão utilizadas especificamente neste jogo, e é uma composição das interfaces generalizadas que serão explicadas posteriormente.

1.3. IPlayer:

É uma das duas folhas da árvore de herança de interfaces que começa em IMobGeneric, sendo que a partir de IPlayerGeneric, só contêm funções relativas a entidade Player. Ela é uma composição de interfaces generalizada. É importante mencionar que a única classe que implementa essa interface, Player, só se refere a métodos de suas classes superiores, ou seja, na prática poderíamos apenas exportar a Interface superior, IPlayerGeneric, mas por requisição de funções específicas, isto é, que quebram a generalidade de se usar atributos do tipo IAttribute de forma

genérica, foi implementada também esta interface, contendo métodos mais específicos como getXp(), setXp(), getNivel(), etc;

1.4. IMonstro:

Assim como IPlayer, IMonstro é a outra folha da árvore de herança de interfaces que começa em IMobGeneric e é uma composição de várias interfaces generalizadas do módulo. Por enquanto essa classe não tem implementação, porque nenhum método especifico para Monstro foi requisitado, mas a estrutura para o caso de surgir esta requisição já foi montada, para manter a simetria com relação a Player.

2.De itemComponent:

2.1. IItemManager:

Assim como o Manager de Mob, esta interface contém apenas métodos para instanciar cada uma das folhas da árvore de herança de Itens.

2.2. IItem:

É exportado para permitir administração e armazenamento de um Item genérico, o que é feito pelo componente inventory por exemplo.

2.3. IItemBattle:

Uma das três folhas da árvore de herança de interfaces que começa em IItemGeneric, contém métodos para gerenciar os Modificadores de Atributos (modAtts) e possui métodos das interfaces generalizadas, que são subcomponentes gerenciados por IModAttManager, que serão detalhados mais adiante na parte de sub componentes.

2.4. IItemPuzzle:

Também uma das três folhas da árvore de herança de interfaces que começa em IItemGenerics e generaliza as interfaces gerais, contém métodos para gerenciar os adjectives, que são implementados apenas como Strings que serão usadas pelo PuzzleModule.

2.5. IItemIlumination:

Mais uma folha da árvore anteriormente citada, este Item contém métodos para gerenciar um atributo especial que apenas esse item tem, que é a iluminação, e também é composto das interfaces generalizadas.

3.De attComponent:

3.1. IAttManager:

Assim como os outros Managers, este apenas contêm métodos para instanciar IAttribute.

3.2. IAttribute:

Contém métodos de leitura e escrita do valor do atributo e de leitura do nome do atributo, além do método clone que é útil ao modificador de atributo. Subcomponentes:

1.De mobComponent

1.1. mobManager:

Exporta as mesmas interfaces de mobComponent.

1.2. inventoryComponent:

1.2.1. IInventory:

Contém os métodos para gerenciamento de itens dentro do inventário.

2.De itemComponent

2.1. itemManager:

Exporta as mesmas interfaces de itemComponent.

2.2. modAtt:

2.2.1. IModAttManager:

Métodos para criar modificadores de atributos.

2.2.2. IModAtt:

Contém métodos para modificar e desmodificar atributos do personagem durante o momento da batalha.

Interfaces Requeridas:

1. De mobComponent

2.De itemComponent

2.1. De itemManager

2.1.1. IModAtt 2.1.2. IInventory 2.1.3. IModAttManager 2.1.4. IPosition

2.2. De modAtt

2.2.1. IAttribute

2.3. Do próprio itemComponent

2.3.1. IPosition 2.3.2. IMob 2.3.3. IAttribute

1.1.

1.2. 1.3.

De mobManager

1.1.1. IInventory 1.1.2. IAttribute

De inventory

1.2.1. IItem

Do próprio mobComponent

1.3.1. IPosition

Interface provida pelo mapModule que indica a posição do mob.

1.3.2. IItem 1.3.3. IPlayer

Interfaces Generalizadas

Percebeu-se que as interfaces mais gerais de alguns componentes implementavam métodos iguais, por isso decidiu-se usar o Interface Segregation Principle (ISP) como pode ser visto nesse trecho do diagrama:

1. IClonable:

Método para clonar um tipo genérico, implementado com generics.

2. IDescriptionReadable:

Contém um método que retorna a descrição de um objeto.

3. IGetTypeAble:

Método que retorna um tipo enumerado que lista todos os tipos de mob e item, para poder identificá-los mais facilmente nos outros componentes.

4. INameReadable:

Lê o nome do objeto.

5. IPositionable:

Lê e modifica a posição.

Principais Classes

As principais classes de cada componente são:

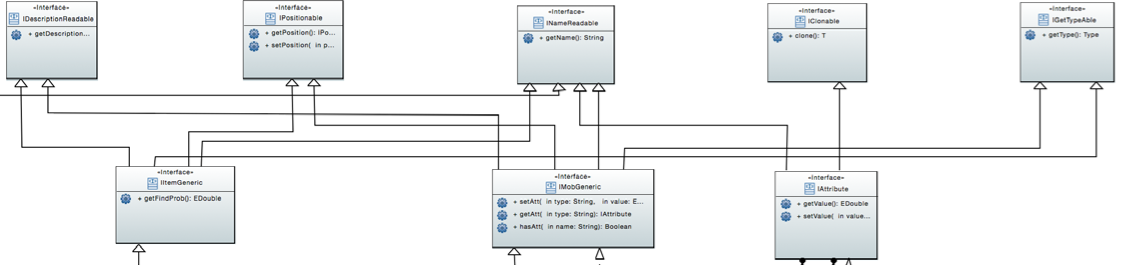
1.De mobComponent:

1.1. MobFactory:

Classe abstrata que se utiliza do design pattern Abstract Factory para criar monstros e player.

1.2. PlayerInstantiator e FabricaDeMonstro:

Fábricas que implementam a fábrica abstrata e criam os mobs.



1.3. MobGeneric:

Classe abstrata que tem atributos para guardar nome, posição, descrição e Attributes genéricos de cada mob. Também possui métodos para acessar Attributes e os atributos. Classes mais específicas como Player e Monstro herdam dessa classe.

1.4. Player:

Classe que será usada por todo o jogo, que herda de um PlayerGeneric que implementa, pela árvore de heranças, as interfaces generalizadas e possui os atributos e métodos de MobGeneric. Foi implementada utilizando o pattern singleton.

2.De itemComponent:

2.1. Item:

Classe abstrata que tem atributos de nome, descrição, probabilidade de ser encontrado e posição. Possui métodos para guardar e modificar esses atributos.

2.2. ItemManager:

Classe principal do componente, que é uma fábrica que instancia os vários tipos de itens.

2.3. ItemBattle:

Item que possui uma lista com os modificadores de atributo que modificam atributos de um mob. Possui métodos para adicionar um modificador, atualizar e desatualizar um mob.

3.De attComponent:

3.1. AttCreator:

Cria os Attributes genéricos.

3.2. Attribute:

Possui atributos de nome, valor, valor máximo e mínimo, e possui métodos para guardar e retornar esses atributos.

4.De inventoryComponent:

4.1. Inventory:

Possui um atributo para guardar o tamanho do inventário e uma hashtable com os items guardados. Possui métodos para gerenciar esses itens.

5.De modAttComponent:

5.1. ModAttCreator:

Possui métodos para criar modificadores de atributos.

5.2. ModAtt:

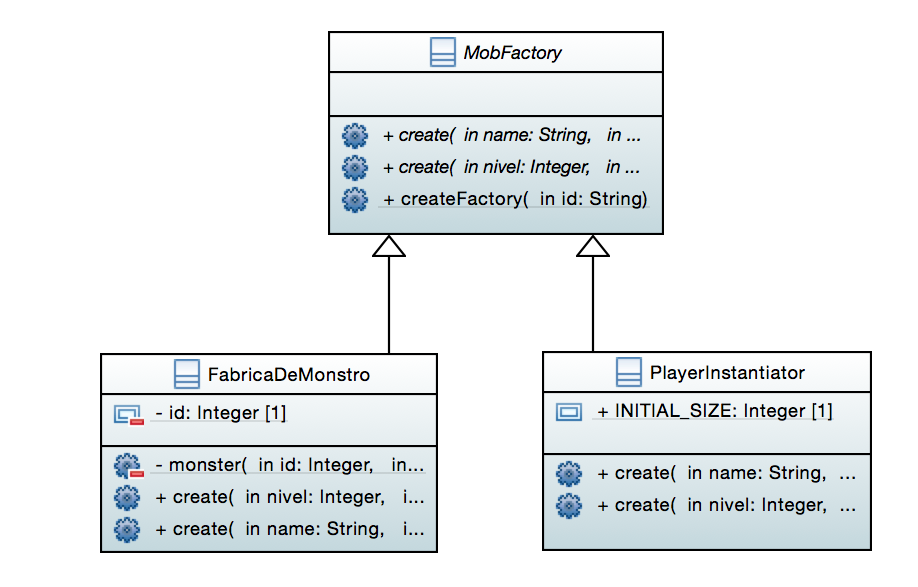
Contém atributos como nome, operação (que é um objeto que implementa a interface IModificator), parâmetros e o atributo que ele modifica. Tem métodos para modificar e desmodificar o atributo.

5.3. Addicionator e Multiplicator:

Classes que implementam a interface IModificator, e realizam diferentes tipos de operações com o atributo.

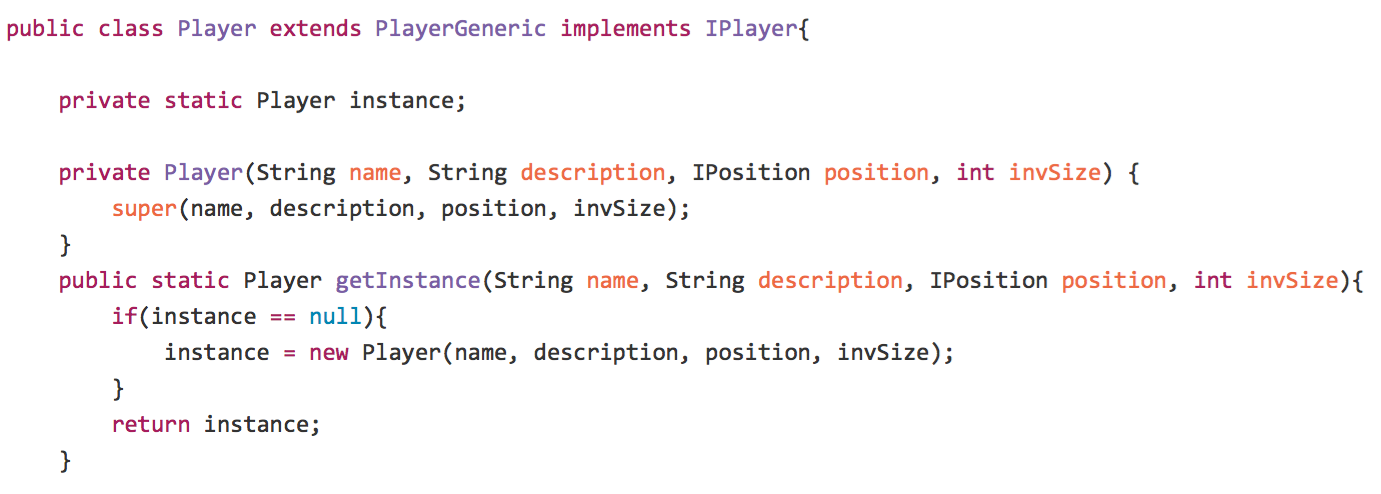
Patterns e Princípios

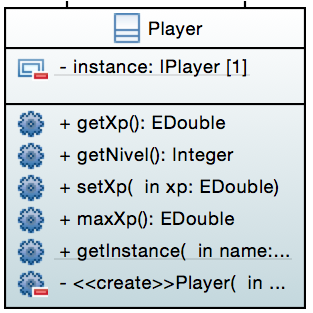
1.Abstract Factory:

O Pattern foi adotado para instanciar os objetos, tornando o código extensível, caso outros mobs sejam criados.

2. Singleton:





O pattern foi adotado pois no jogo só existirá um Player, e isso não permitirá o instanciamento por acidente de um segundo Player.

3.Interface Segregation Principle (ISP):

O Princípio foi adotado, como já foi citado, na montagem da árvore de herança com as interfaces menores generalizadas, que diminuem os métodos nas interfaces exportadas, e as transforma numa composição de interfaces.

4.Open Closed Principle (OCP):

O Princípio foi adotado basicamente não permitindo que os componentes fossem modificados, mas que pudessem ser abertos para extensão. Isso foi feito com as generalizações de Mob e Item, para poderem ser extendidos quando se precisassem de métodos diferentes caso o componente fosse vendido para outro projeto.

Plano de Exceções

O Plano de exceções ainda está sendo desenvolvido, então pode sofrer alterações. Até o momento, existem 5 tipos de exceções que podem ser tratadas pelo mobModule, e elas são:

1. BadArgumentExcetion:

Esta exceção é disparada se, na hora de criar um modificador de atributo que divide o valor, o valor passado para a divisão for zero.

2. BadAttributeModificationException:

Esta exceção é uma generalização das exceções que podem ser lançadas quando uma modificação ou desmodificação de um atributo da errado.

3. BadConstructorException:

Esta exceção é disparada quando por algum acaso for chamado o construtor do Monstro com os parâmetros do Player ou o construtor do Player com os parâmetros do Monstro.

4. CannotDoubleModifyAttributeException:

Esta exceção é jogada caso se tente modificar o mesmo atributo duas vezes com o mesmo modificador de atributo quando o Player entra em batalha e equipa seus itens.

5. CannotUnmodifyWhatHasNotBeenModifiedException:

Esta exceção é disparada quando se chamou a função de desmodificar os atributos do Player, mas os atributos ainda não tinham sido modificados ainda.

**Puzzles/Salas:**

O PuzzleDialog funciona como um enquirer para o componente Puzzle, de forma que para cada instância de um dado Puzzle, existe um PuzzleDialog associado a ele, pensei ser útil isso caso quiséssemos fazer um Puzzle depender do outro (numa relação de continuidade, por exemplo). Ainda, uma instância de cada "Enquirer" poderia ser justificada pelo estado independente de cada puzzle, que ficaria salvo no componente PuzzleDialog que gerencia o Puzzle a ele associado.

Cada um dos 2 componentes implementa uma interface, com os seus respectivos nomes (IPUzzleDialog e IPuzzle). Assim do diagrama de componentes podemos ver que a função da interface implementada pelo PuzzleDialog é se comunicar com o player (mundo externo ao modo sala), recebendo os inputs digitados pelo player e outputs advindos do puzzle.

Vamos aos métodos:

**Componente PuzzleDialog** -> Implementando a interface IPuzzleDialog, ele define:

*Atributos:*

**- int**maxTime: para cada puzzle um tempo máximo se for caso.

**-** **int**dPuzzleId: Um modo de identificar este componente quando for instanciado, numericamente, puzzle 1, 2, 3, ... quantos houverem .

*Métodos:*

+inputMsg(String msgIn): recebe o comando digitado pelo jogador, e retorna de acordo com a situação do puzzle (pode fazer isso, não pode, etc etc..)

+outputMsg(String puzzleInString): recebe uma string do vetor de strings existentes no puzzle para repassá-la ao ao player, podendo ser aplicada na mensagem a função da zoeira, que iria embaralhar a mensagem original a critério da engine do game, ou simplesmente adicionar algo à mensagem do puzzle em função de um evento ocorrido durante a resolução do puzzle.

+setMaxTime(): Altera o valor de maxTime.

+setDPuzzleId(): Altera o valor de dPuzzleId.

+getDPuzzleId(): Método para pegar o numero do Puzzle.

+getCorridorPosition(Int gameProgress): Método que vai decidir qual puzzle o player vai entrar em função da posição dele no mapa, podemos definir valores fixos, posicao 15 do mapa, puzzle 1, 35 puzzle 2, e assim por diante. Ou se o mapa for gerado proceduralmente, então percentualmente (pos.atual/pos.final), quando chegar a x% será carregado o enésimo puzzle).

+getMaxTime(): Acessa o tempo limite estabelecido.

**Componente Puzzle** -> Implementa a interface IPuzzle, e instância cada Puzzle de acordo com o tipo dele implementa os métodos da interface conforme for conveniente.

*Atributos:*

**-** **int**puzzleId: Número que representa o puzzle corrente.

**-** **HashTable<String, Boolean>** Tasks**:** Permite a alocação de um pequeno texto descritivo das tarefas demandadas pelo puzzle e ainda associa a cada uma delas (HashTable extendsDictionary) um valor booleano (inicialmente false para todos). Independentemente do tipo do puzzle criado, ele tem uma sequência de tarefas que pode ser abstraída e armazenada dessa forma.

**- Vector<String>** messages: Armazena, de forma sequencial, as mensagens que seriam enviadas ao player referentes a cada etapa do puzzle. As mensagens viriam de um banco de dados conforme demanda do puzzle em corrente funcionamento.

*Métodos:*

+getRoomPos(): retorna a posição atual do jogador dentro da room para o puzzle, caso seja um puzzle labiríntico ou algo do gênero.

+getPuzzleId(): Método de acessar o puzzleId.

+getItems(): *Este método pode não ser necessário a depender de como será dado o acesso aos items de puzzle, que foram criados como classes e podem assim ser acessados diretamente, o método pode no entanto servir para padronizar o acesso aos itens para todos os tipos de puzzles.*

+combineItems(): em puzzles no estilo first times, items podem precisar ser combinados, o que a depender da implementação dos itens pode ser apenas a mudança de um status do item, ou um adjetivo, como está sugerido pela equipe dos Itens.

+checkCurrTask(): Pode ser usado para dar alguma dica ao jogador sobre a tarefa que está correntemente realizando, e para verificar internamente no programa se a tarefa foi terminada por completo, essencialmente verifica já pode trocar o estado de uma task da HashTable para true (= Terminada).

+setSequenceOfTasks(): Este método basicamente adiciona a descrição e o estado das tarefas que seriam realizadas sequencialmente. Ele pode ser implementado pela manipulação de um arquivo externo (.txt) que teria as definições do puzzle.

+setNewStatus(): método para efetivamente alterar a Hastable(que é private).

+Timer(): implementa o temporizador (tempo transcorrido dentro do puzzle) se o puzzle necessitar, e também implementa uma verificação periódica se o tempo do Timer não ultrapassa o tempo definido no PuzzleDialog, por meio do método getMaxTime(). O método provavelmente chamará o timer da biblioteca javax.swing.Timer definindo os parâmetros corretamente do temporizador.

+addMessage(): Insere as mensagens de cada puzzle no Vector<String>. Ele pode ser implementado pela manipulação de um arquivo externo (.txt) que teria as definições do puzzle.

**Interface IPuzzleDialog** -> Demanda a implementação dos métodos inputMsg(String msgIn) e outputMsg(), que em resumo serão responsáveis por fazer a comunicação com o modo sala, definido como um dos 5 macro módulos do jogo. Este modo sala necessita receber as mensagens digitas pelo jogador no console ou terminal (será desenvolvido graficamente por meio da manipulação da biblioteca gráfica LibGDX). As mensagens serão recebidas pelo componente que efetivamente implementa estes métodos e possui função de gerenciar todos os parâmetros necessários à instanciação do Puzzle. Este por sua vez poderá gerar os Puzzles em si por meio do Pattern Factory.

**Interface IPuzzle** -> Demanda a implementação dos 4 métodos que formam um conjunto de características mínimas para qualquer tipo de puzzle, estas características são:

- Sequência de tarefas a ser realizada para a conclusão do puzzle. (método setSequenceOfTasks());

- Informações tanto para o player quanto para o macro módulo Sala em si sobre a tarefa que está pendente no momento (não são esperadas tarefas concorrentes num mesmo Puzzle) (checkCurrTask(String msgIn));

- Método para alterar a HashTable que guarda o estado atual da tarefa corrente (estados: completa = true; incompleta = false; dependendo do estado desta tarefa uma mensagem diferente pode ser passada ao usuário (Hints) ) (setNewStatus());

- Um timer deve ser implementado internamente para cada Puzzle, o tempo será contabilizado independentemente do puzzle possuir um tempo para finalização, de modo que ao final do jogo uma forma de Scores de conclusão de cada Puzzle possa ser exibida ao término do jogo. (Timer() – implementará um temporizador da biblioteca *javax.swing.Timer*);