Apresentação dos Conceitos de Programação Orientada a Objeto e da Premissa de Programa Utilizando Tais Conceitos

Adolfo S. Cardoso, Eric F. Schmiele, Gabriel H. G. Siqueira

Departamento de Engenharia – Universidade Federal de Lavras (UFLA) – Lavras, MG – Brasil

Abstract. *This paper describes the basic concepts involved in Object Oriented Programming, a recurring type of programming that has a strong presence in the market nowadays, and presents an introduction for a program using these concepts. The program consists of a simple RPG fighting style game, with a user interface and functionalities designed for entertainment. With this program, we also intend to enhance the programming capabilities of its developers.*

Resumo. *Este artigo descreve os conceitos básicos envolvidos na Programação Orientada a Objeto, um tipo de programação que tem tido uma forte presença no mercado ultimamente, e apresenta uma introdução de um programa que utiliza estes conceitos. O programa consiste de um simples jogo de lutas no estilo RPG, com uma interface para usuário e funcionalidades desenvolvidas para entretenimento. Com este programa, também desejamos aprimorar as capacidades de programação de seus desenvolvedores.*

1. Conceituação de Orientação a Objeto e elementos fundamentais

Orientação a Objeto (OO) é um dos paradigmas existentes no desenvolvimento de softwares, sendo atualmente o mais difundido, *The RedMonk Programming Language Rankings: June 2015*. Fatores preponderantes na grande utilização deste preceito são a facilidade no reaproveitamento de código, a modelagem mais aproximada do mundo real (pelo próprio conceito de objeto), além da robustez do código e expansibilidade, Cleidson Souza.

Os dois conceitos mais básicos e fundamentais da OO, são os conceitos de Classe e Objeto.

1.1. Classe

Classe é a generalização (gabarito) de um objeto, ou seja, é a definição por meio de código de seus atributos e métodos, sendo a função da classe garantir que os objetos advindos desta estejam em formatos validos para a aplicação, além de definir o comportamento geral do objeto. Para realizar isto a classe é subdividida em três partes bem definidas pela linguagem: nome da classe, atributos da classe e métodos da classe, Ivan Luiz Marques Ricarte.

1.1.1. Nome da Classe

O nome da classe é um identificador para a classe, pelo qual se dará a referência desta em momentos posteriores, como na criação de um objeto, Ivan Luiz Marques Ricarte.

1.1.2. Atributos

Atributos definem as propriedades da classe, cada atributo é definido por um nome e um tipo, sendo o tipo o nome de uma classe ou um tipo primitivo de dado, e o nome o jeito de referenciar-se a esse dentro da classe, Ivan Luiz Marques Ricarte.

1.1.3 Métodos da Classe

Métodos definem a funcionalidade da classe, ou seja, como os objetos da classe podem atuar, cada método é especificado por uma assinatura, sendo a assinatura composta pelo identificador (nome) do método, o tipo de valor para o seu retorno e a lista dos parâmetros necessários para a realização do método (sendo esta lista composta pelo tipo e identificador interno para cada parâmetro), Ivan Luiz Marques Ricarte.

1.2. Objeto

Objetos são a instanciação de uma classe, ou seja, são uma particularização da classe, ARMES, David J.; KÖLING, Michael. Além desta definição geral de objeto e importante sedimentar que cada objeto possui atributos únicos e válidos (estado válido), e uma posição de memória única (sendo esta a forma mais simples e direta de referenciar-se a um objeto, sendo utilizada na comparação de igualdade padrão em JAVA). Vale citar que Objetos são a principal estrutura da OO, já que neste paradigma de programação todas as tarefas são realizadas por objetos, Ivan Luiz Marques Ricarte. Objetos se comunicam através de mensagens.

1.2.1. Mensagem

Mensagem é a forma que os objetos interagem entre si, na prática, a troca de mensagens se traduz na aplicação dos métodos existentes em dado objeto, Ivan Luiz Marques Ricarte.

2. Instanciação de Objetos

Objetos são instanciados por meio de construtores em suas classes, os quais devem ser feitos de modo com que todos os objetos que sejam criados por esta classe tenham estado inicial válido, podendo depender ou não de parâmetros externos. Estes parâmetros externos precisam ser checados, para garantir o estado inicial da classe, essa aferição deve ser feita por meio de Exceções. Também vale destacar que os construtores podem ser sobrescritos para dar maior versatilidade aos estados inicias da classe.

Esta maior versatilidade de estados iniciais tem de ser pensada para satisfazer as diversas necessidades do usuário. Para a sobrecarga ocorrer é necessário garantir que os construtores possuam parâmetros diferentes em quantidade ou variação de ordem dos tipos dos parâmetros.

Por fim vale destacar que construtores não são herdados, independentemente do que se é desejado.

3. Pilares Fundamentais da OO

A OO, se baseia em quatro pilares fundamentais, sendo eles: Abstração, Encapsulamento, Herança e Polimorfismo, sendo os 3 últimos específicos da OO e a Abstração de qualquer forma computacional de resolução de problemas.

3.1. Abstração

Abstração é um dos principais conceitos em qualquer tipo de programação, mas neste trabalho iremos trata-la na forma utilizada pela OO. De forma geral, abstrair é o processo de representar um grupo de entidades através de seus atributos comuns, feita a abstração cada instância do grupo é considerada somente pelos seus atributos particulares, Fábio Protti.

Com isso é possível ver que a abstração é o principal passo para a passagem do mundo real para o mundo computacional, e para a modelagem do sistema em si. Pois dada a abstração é possível definir as classes existentes em dado sistema e seus relacionamentos, para assim conseguir com que os objetos e seus métodos atinjam o fim desejado.

3.2. Encapsulamento

Encapsulamento é um dos principais conceitos de OO, neste trabalho trataremos como encapsulamento a junção dos conceitos de Encapsulação e Ocultamento de Informação.

3.2.1. Encapsulação

Encapsulação é um princípio de projeto, no qual cada parte de um programa deve agregar em si toda informação relevante para sua manipulação, Ivan Luiz Marques Ricarte.

3.2.2. Ocultamento de Informação

Ocultamento de Informação é o princípio pelo qual cada parte de um programa deve manter oculta sob sua guarda as informações que possui. Para a utilização do mesmo, apenas o mínimo e necessário para sua operação deve ser revelado, Ivan Luiz Marques Ricarte.

Com a junção destes conceitos chegamos ao conceito de Encapsulamento utilizado no trabalho, no qual o estado objeto deve ser mantido oculto (atributos), e a utilização do objeto deve ser feita apenas pela utilização de seus métodos públicos, assim tendo de revelar apenas as assinaturas de seus métodos públicos, sendo estas assinaturas a definição de interface operacional, Ivan Luiz Marques Ricarte.

Com isso conseguimos visualizar a importância do Encapsulamento devido a sua importância para a manutenção de objetos e classes robustas, tendo em vista que os objetos de dada classe apenas podem ser alterados pelos métodos da própria classe. Com isto evitando a alteração de forma descuidada por terceiros, assim aumentado a

segurança do sistema como um todo, dificultando a existência de objetos em estados inválidos e o uso de forma descuidada de classes já existentes (não há como gerar objetos desta classe sem seguir os padrões definidos nela).

3.3. Herança

Herança é um dos principais conceitos de OO, este mecanismo permite que características comuns a classes distintas, denominadas subclasses, sejam fatoradas em uma classe base, denominada superclasse. Utilizando-se deste conceito cada subclasse possui a estrutura e métodos existentes na superclasse além de adicionar métodos específicos pertinentes ao escopo da subclasse. A herança pode se subdividir em várias formas, sendo elas: Extensão, Especificação e Combinação de Extensão e Especificação, Ivan Luiz Marques Ricarte.

3.3.1. Extensão

A subclasse estende a superclasse, acrescentando novos membros. A superclasse permanece inalterada, motivo pelo qual este relacionamento é normalmente referenciado como herança estrita Ivan Luiz Marques Ricarte. No escopo deste trabalho este tipo de herança é tratado como a utilização de Implements no Java.

3.3.2. Especificação

A superclasse apenas define uma interface mínima para as subclasses, com isto é dito que apenas a interface da superclasse é herdada pela subclasse, Ivan Luiz Marques Ricarte. No escopo deste trabalho este tipo de herança é tratado como a utilização de Extends no Java e é responsável pela definição de uma interface básica para as subclasses.

3.3.3. Combinação de Extensão e Especificação

A subclasse herda a interface e uma implementação padrão de métodos da superclasse. A subclasse pode então redefinir métodos para especializar seu funcionamento, tal qual requerido, além de ter de implementar os métodos que a superclasse tenha apenas declarado, mas não implementado. Geralmente este tipo de herança e denominado herança polimórfica, Ivan Luiz Marques Ricarte. No escopo deste trabalho e em Java isto é realizado pela utilização da combinação de Implements e Extends.

Destacando que construtores nunca são herdado, independentemente do método utilizado.

Java se utilizou de Combinação de Extensão e Especificação para contornar a limitação de herança múltipla (permitida em C++), devido ao fato de Java apenas permitir a Implementação de uma superclasse, mas a Extensão de N superclasses. Com isso possibilitando a existência de N Interfaces mínimas implementadas, mas apenas um padrão de comportamento de superclasse.

Além disso vale-se destacar a existência de hierarquia de herança em Java, ou seja, caso exista uma subclasse que seja superclasse de outra (qualquer tipo de herança

segue esta logica caso visto de forma mais profunda, uma vez visto que Object é superclasse de todos as demais classes), a nova subclasse herdara tanto da superclasse quanto da superclasse de sua superclasse, assim possibilitando a existência de herança direta (quando se herda diretamente da superclasse) e herança indireta (quando se herda de uma superclasse da superclasse da classe atual).

Assim sendo a herança e muito importante para a expansibilidade do código, utilizando-se de classes já feitas. Para a confiabilidade, pois se baseia em classes previamente testadas, além de definir uma interface mínima para as subclasses, sendo esta a interface da superclasse, assim sabendo que todas as partes do sistema conseguirão se comunicar minimamente nos padrões estabelecidos.

Concluindo, herança é a principal ferramenta para gerar expansibilidade de código, pois utilizando-se de superclasses já feitas e bem especificadas, é possível se criar uma grande quantidade de subclasses (sendo estas previstas ou não no projeto inicial). Também é uma ferramenta primordial para a confiabilidade, pois baseia-se em classes previamente testadas para a criação de novas classes. Além de padronizar a interface mínima das subclasses, assim sabendo que o sistema poderá se comunicar ao menos nos padrões do projeto (sendo ao menos compatível).

Por fim, deve-se definir o que é interpretado como Interface para Java, logo o que é passível de ser implementado (possuir uma subclasse que Implementa esta).

3.3.4. Interface

Interfaces são classes as quais por definição são abstratas e em sua forma mais comum são apenas um grupo de métodos relacionados com “corpo” vazio. Implementar uma interface permite a classe se tornar mais formal quanto ao comportamento que deve realizar. Sendo assim Interfaces podem ser vistas como contratos entre a classe e o mundo externo a classe, pois quando uma classe implementa uma interface esta é obrigada a sobrescrever todos os métodos da Interface, para o código ser compilado com sucesso, Oracle (concepts/interface).

3.4. Polimorfismo

O polimorfismo é o último dos principais conceitos de OO definidos neste trabalho, sendo que polimorfismo é o princípio pelo qual duas ou mais classes derivadas de uma mesma superclasse podem invocar métodos com a mesma assinatura, mas comportamentos distintos e especializados para cada classe derivada, usando para isto a referência a um objeto da superclasse. Sendo este mecanismo fundamental na OO, permitindo definir funcionalidades que funcionem genericamente com objetos, desconsiderando seus detalhes particulares quando estes não forem necessários, Ivan Luiz Marques Ricarte. Em Java esta forma de chamada de métodos se torna ainda mais poderosa devido a todos os objetos herdarem de Object, logo todos terem uma superclasse comum caso visto de forma geral.

Para o polimorfismo ser utilizado, apenas é necessário que os métodos que estejam sendo definidos nas subclasses tenham a mesma assinatura da superclasse, utilizando-se para isso de Overriding, Ivan Luiz Marques Ricarte.

Desta forma é impossível para o compilador decidir qual método será executado, afinal pelo princípio de substituição um objeto da subclasse pode estar sendo referenciado por um objeto da superclasse, com isso a decisão sobre qual método será utilizado apenas pode ocorrer em tempo de execução, utilizando para isso o mecanismo de ligação tardia (também denominado como late binding, dynamic binding ou run-time binding), Ivan Luiz Marques Ricarte. Assim sendo agora é necessário explicar a diferença entre ligação tardia e (ligação dinâmica) e ligação estática.

3.4.1. Ligação Dinâmica

Na ligação dinâmica a mensagem apenas é ligada a um método em tempo de execução, com a herança advinda do polimorfismo não se sabe qual será o método utilizado com certeza até se saber a classe dinâmica do objeto em tempo de execução, Reinaldo Gomes. Assim sendo o programa irá executar o método de seu tipo dinâmico em tempo de execução, independentemente do tipo estático declarado na compilação, tudo isto devido a junção de herança e polimorfismo.

3.4.2. Ligação Estática

Na ligação estática a ligação entre a chamada de um subprograma e sua implementação é estabelecida em tempo de compilação, sendo o existente durante o tempo de compilação, Reinaldo Gomes. Assim sendo o compilador apenas compilara o código sem erro caso seja seguida a lógica de Ligação estática, ou seja, se o polimorfismo for bem utilizado (devido a uma subclasse ser do tipo de sua superclasse).

4. Visibilidade de Métodos e Classes

O conceito de visibilidade diz respeito a capacidade de utilização e visualização de atributos e métodos dentre os objetos existentes em dado sistema. Em Java a visibilidade e subdividida em 4 casos, separados por 3 keywords básicas: Public, Protected e Private, além é claro do default, que seria o caso sem keyword modificadora utilizada.

4.1. Public

O modificador Public, faz com que o método ou atributo possua visibilidade externa total, o que significa que este poderá ser acessado por qualquer outro objeto do sistema, pertencente ou não ao mesmo pacote, Ivan Luiz Marques Ricarte.

4.2. Private

O modificador Private, faz com que o método ou atributo possua nenhuma visibilidade externa, o que significa que nenhum outro objeto poderá acessa-lo, Ivan Luiz Marques Ricarte.

4.3. Protected

O modificador Protected, faz com que o método ou atributo possua visibilidade limitada, ou seja, apenas para objetos de classes que sejam derivadas dessa através do mecanismo de herança Ivan Luiz Marques Ricarte. Além disso, classes que dividem o mesmo pacote também podem acessar o atributo ou método definido como Protected, Oracle (acesscontrol).

4.4. Default

O modificador Default (utilizado quando não se especifica a visibilidade do método ou atributo diretamente), faz com que o método ou atributo possua visibilidade limitada, mas diferentemente do método Protected, no método default apenas as classes que dividem o mesmo pacote podem acessar o atributo ou método definido como Default, Oracle (acesscontrol).

Além do citado, vale destacar que para todos os casos a própria classe pode acessar seus métodos ou atributos, independentemente do tipo de visibilidade existente, Oracle (acesscontrol).

Finalmente será definido o que é considerado um pacote, para melhor entendimento das definições passadas.

4.5. Pacotes

Pacote é uma unidade de organização de código que condensa, classes, interfaces e exceções relacionadas. O código-base de Java é estruturado em pacote. Com isso aplicações desenvolvidas em Java devem ser assim organizadas também. Isto é feito para uma maior facilidade de compilação (com menor gasto computacional), devido a buscar-se uma classe apenas em um local especifico do classpath e não em todo ele, além de combater o problema de nome de classes iguais, pois classes podem ter nomes iguais desde que pertençam a pacotes diferentes, Ivan Luiz Marques Ricarte. Concluindo, assim, se facilita a modularização do código (cada parte especifica fica contida a seu próprio pacote) e reaproveitamento de código (import de pacotes específicos).

5. Relações entre classes

Os relacionamentos que duas ou mais classes podem ter são: Dependência, Generalização e Associação, cada um destes relacionamentos possui características específicas, além de representação única no diagrama UML, Robinson Vida Noronha.

5.1. Dependência

Dependência é um relacionamento entre objetos, é representado no diagrama UML por uma reta tracejada com terminal em seta. Duas classes se relacionam por dependência quando uma alteração em uma das classes pode afetar a outra e o inverso não é verdade. Ou seja, uma classe utiliza a outra como argumento na assinatura de algum de seus métodos Ex: Dvd player precisa de um Dvd mídia, ou seja, o método play da classe Dvd

player, usa como parâmetro um objeto ou instancia da classe Dvd mídia, Robinson Vida Noronha.

5.2. Generalização

Generalização é um relacionamento entre objetos, é representado no diagrama UML por uma reta não tracejada com terminal em triângulo. Duas classes se relacionam por generalização quando uma das classes especializa ou detalha a outra. Sendo a classe genérica denominada superclasse e a outra subclasse Ex: Engenheiro e Músico são pessoas, Robinson Vida Noronha. Assim sendo a relação de generalização nada mais séria do que a Herança previamente definida. Além disto vale destacar que em Java também se utiliza a Implementação de Interfaces (como simulação de herança múltipla presente em outros softwares), como forma de relacionamento entre classes, está também é um tipo de Generalização e é representada no Diagrama UML como uma reta tracejada com terminal em triângulo.

5.3. Associação

Associação é um relacionamento entre objetos, é representado no diagrama UML por uma reta não tracejada com ou sem terminação, levando-se em conta o tipo de Associação. Associação é uma relação estrutural, assim sendo ela informa que uma classe faz parte de outra. O relacionamento de associação pode ser subdividido em: Associação Plana e Agregação, Robinson Vida Noronha.

5.3.1. Associação Plana

Associação Plana representa uma relação estrutural onde as classes possuem a mesma importância, uma linha ligando duas classes representa esta relação em UML. Ex: Saxofonista toca com Baterista, Baterista e Saxofonista são classes distintas que se relacionam e possuem mesma importância, Robinson Vida Noronha.

5.3.2. Agregação

Agregação representa a estrutura todo-parte. É representada em UML por uma linha ligando ambas as classes com um diamante na parte do todo (preto ou branco dependendo do tipo de Agregação). Agregação ainda é subdividida em Agregação Simples e Composição, Robinson Vida Noronha.

5.3.2.1. Agregação Simples

Agregação simples representa a estrutura todo-parte, sem relação de posse entre as classes. Ex: Músico toca em orquestra, nesse caso a orquestra é mais importante que o músico, sendo o todo e configurando a relação todo-parte, Robinson Vida Noronha.

5.3.2.2. Composição

Composição representa a estrutura todo-parte**,** mas além disto também caracteriza a relação de posse. Assim sendo um objeto da classe parte pertence apenas e exclusivamente ao objeto da classe todo. Com isto a parte todo se torna responsável pela criação e destruição das partes que a compõe. Ex: Um tabuleiro de xadrez possui casas, Robinson Vida Noronha.

6. Exceções

Exceção é um sinal que indica que um tipo de condição excepcional ocorreu durante a execução do programa. Com isto, exceções estão associadas a condições de erros impossíveis de serem checadas no momento da compilação, sendo geradas na classe em que ocorrem e capturadas pela classe que se deseje e seja possível ocorrer o tratamento destas Ivan Luiz Marques Ricarte.

6.1. Geração

A sinalização de que a condição excepcional ocorreu, Ivan Luiz Marques Ricarte. Geralmente geradas pelo usuário com inputs inválidos, ou por mal funcionamento do sistema.

6.2. Captura

A manipulação da situação excepcional, onde as ações necessárias para a recuperação da situação de erros são definidas. Capturadas por blocos de Try-Catch ou finally, e tratada por exception handler, Ivan Luiz Marques Ricarte.

A exceção se propaga a partir do bloco de código onde ocorreu através de toda a pilha de métodos até que ela seja capturada, até encontrar um exception handler capaz de trata-la, caso isto não ocorra a Exceção se propagará até o main e abortará o programa, com uma mensagem de erro para o usuário, Ivan Luiz Marques Ricarte.

7. Eventos

Para implementar a funcionalidade de uma interface gráfica, pode-se fazer uso de uma repetição sem fim. Esse tipo de programação é considerado extremamente ineficiente, pois é necessário explicitamente “ler” os periféricos para tratar as ações do usuário. Por isso é utilizada a implementação de eventos. A implementação por eventos faz com que cada vez que ocorra um gatilho de evento o sistema gere um sinal de evento, que atuará em todas as classes que estejam ouvindo este evento, Marcelo Grohen. Eventos funcionam como interrupções no sistema, e fazem com que o processamento atual seja parado em favorecimento da classe que se utiliza deste evento.

Para que um evento seja utilizado corretamente é necessário a implementação de um ouvinte tal qual citado, fornecido pelo Java como Event Listener. Esta é a interface específica para tratar a ativação de eventos e realizar a chamada da classe ouvinte. Quando não for possível a utilização de tal classe cabe ao programador realizar a implementação de um Event Listener particular.

Com os conceitos teóricos abordados, se finaliza o referencial teórico do trabalho, seguindo-se o desenvolvimento do mesmo.

8 – Padrões de Projeto:

Padrões de Projeto servem para solucionar problemas recorrentes em projeto de software, estabelecendo soluções elegantes para este tipo de problemas, também sabendo as vantagens e desvantagens existentes na utilização de tal padrão. Além disso são importantes para uma melhor comunicação entre trabalhadores da área, Alessandro Garcia. Neste trabalho foram utilizados os padrões de projeto Observer e Factory.

8.1 – Observer: É utilizado quando necessário acoplar objetos entre si, sem que estes apenas se conheçam em runtime, e que seja possível criar e desfazer este acoplamento em runtime. Em JAVA um objeto (source) envia informações para outro (listener) pela chamada de um método do listener, Jacques Philippe Sauvé.

A solução para isto é baseada principalmente em interfaces. Para solucionar o problema é necessário gerar uma classe separada de evento para cada tópico principal, encapsulando a informação a ser propagada. Seguidamente é necessário criar uma interface para cada categoria de evento, contendo um método para cada evento, o qual servirá como gatilho da propagação de informação. O próximo passo é criar a classe observável, para isto é necessário criar um par de métodos, um para adicionar e outro para remover listeners, além de possuir a propagação dos eventos, engatilhada no passo anterior. Finalmente define-se os listeners, para fazer isso basta-se implementar a interface listener previamente criada, Jacques Philippe Sauvé.

O padrão Observer dever ser utilizado quando a abstração possui dois aspectos, dependentes, a encapsulação de tais aspectos aumenta a redutibilidade de código, ao poderem serem utilizados separadamente. Além disso quando alterações em um objeto refletem em alterações em outros objetos. E quando não é necessário o acoplamento forte entre os objetos, Jacques Philippe Sauvé.

Como consequências dessa implementação temos que o source ou listener podem variar independentemente, pois o acoplamento entre estes é mínimo. Além disso o suporte de transmissão de informação em broadcast. Mas em contrapartida a mudança de estado do source possui grande custo para propagação de informação caso existam muitos listeners, Jacques Philippe Sauvé.

8.2 – Factory: É utilizado quando a classe é incapaz de antecipar o tipo dos objetos que está precisa criar, assim sendo precisa adiar a instanciação de objetos para suas subclasses. A solução do problema se baseia em criar duas interfaces, uma com os Produtos (Product) e outra com o Criador de produtos (Creator), Eduardo Figueiredo.

Explicando de forma mais detalhada as superclasses não conhecem o produto em específico (gerando independência do código), o método fabrica cria e retorna o objeto desejado quando necessário (eliminando a necessidade de antecipação citada), e por fim o produto geral pode ser usado devido aos produtos específicos herdarem de

Produto (se aproveitando do polimorfismo). Com isso é eliminada a dependência de classes especificas, programando-se para interfaces e não para classes, Eduardo Figueiredo.

Com os conceitos teóricos abordados, se finaliza o referencial teórico do trabalho, seguindo-se o desenvolvimento do mesmo.

9. O Jogo Coliseum

Para colocar todos estes conceitos em prática, desenvolvemos o projeto Coliseum. Um jogo simples de luta baseado em encontros de um role-playing game (RPG).

Dentro de um RPG normal o jogador tem encontros com inimigos, seja por eventos ou por simplesmente estar explorando o mapa, nos quais o jogador deve batalhar contra o inimigo. Baseado neste conceito criamos o Coliseum, um jogo que apenas trata as batalhas destes encontros, ignorando eventos ou mapas. No jogo, o usuário deve escolher o nome do personagem, a qual classe o seu personagem pertence - arqueiro, cavaleiro ou mago - e a dificuldade do jogo. Feito isto, o jogador começa a lutar contra vários inimigos consecutivamente - como se estivesse dentro de uma arena de batalhas. A cada luta o jogador pode receber novos itens para equipar e ficar mais forte e também recebe pontos de experiência para elevar o seu level na determinada classe escolhida e se tornar mais forte. O usuário vencerá o jogo quando o seu personagem chegar ao level 10.

Para efeitos de mecânica de jogo, escolhemos os valores que determinam a "força" do personagem - seja ele o jogador ou o inimigo - como os seguintes atributos, os quais chamamos de stats: força, defesa, velocidade, magia, sorte e vida. Estes são utilizados para determinar quem ataca primeiro, qual a força do ataque, qual será o dano e muitos outros detalhes internos do jogo. Vale destacar como funciona a mecânica de ataques: cada ataque tem como base um stat atacante (que determina a força com que o ataque será deferido), um defensor (que determina a força com que o ataque será evitado) e um stat danificado (stat que será danificado caso o valor de defesa seja menor que o atacante).

9.1. Mudanças desde a Última Versão

Desde a última versão apresentada várias mudanças foram efetuadas no projeto, que agora já está em estado executável com o mínimo de seu funcionamento e com melhorias em funcionalidade. Estas são listadas e explicadas a seguir em ordem cronológica:

- Foi criada uma classe chamada GerarArquivoItens para criar itens e guardá-los em arquivos da maneira correta. Possui o atributo todoOsItens do tipo ArrayListItem contendo todos os itens que foram criados internamente. Possui o método salvarEmArquivo do tipo void com modificadores private static final que cria um arquivo para cada parte do corpo na qual os itens podem ser equipados e salva cada item no seu respectivo arquivo (*.itm). Esta classe não é

utilizada no jogo, serve simplesmente para iniciar os arquivos que serão utilizados durante o jogo.

- Foi criada uma classe chamada TodosOsItens para ler os arquivos onde os itens estão salvos e guardá-los em várias listas para fácil acesso. Possui um atributo chamado todosOsItens do tipo ArrayList<ArrayList<Item>> que é uma lista de listas de itens (uma lista para cada tipo de item, sendo os itens separados pela parte do corpo na qual eles devem ser equipados). Possui os métodos:
  - iniciaListas do tipo void com visibilidade privada final que chama os outros métodos envolvidos na abertura de cada um dos arquivos (chama os arquivos pelos seus nomes corretos).
  - abreArquivoItens do tipo void com visibilidade privada final para abertura de um arquivo de itens específico e inserir os itens deste na lista correta. Recebe um parâmetro String com o nome do arquivo e um parâmetro ParteDoCorpo indicando em qual lista os itens deste arquivo serão inseridos.
  - getItem do tipo Item com visibilidade pública para retornar um item específico de uma lista específica. Recebe um parâmetro int com a posição do item na lista e um parâmetro ParteDoCorpo indicando de qual lista o item será retirado.
- Foi criada uma interface chamada Observador e outra chamada Observavel para seguir o padrão de projeto Observer para uma comunicação correta entre a Engine e as possíveis interfaces de usuário.
- A interface Observador tem o método update do tipo void que recebe como parâmetro o Observador que chamou este método.
- A interface Observavel tem os métodos adicionarObservador e removerObservador do tipo void que recebem um parâmetro Observador específico a ser adicionado ou retirado da lista de observadores, e o método notificarObservadores do tipo void que deve ser implementado de maneira que utilize o método update de cada um dos observadores da lista de observadores (que também deve ser criada pela classe que implementar esta interface).
- Na classe Engine:
  - Agora a Engine implementa a interface Observável para que ela possa enviar mensagens à interface gráfica de usuário sem ter que saber exatamente como esta interface gráfica funciona. Além de poder informar mais de uma classe do tipo Observador ao mesmo tempo.
  - Os atributos nomeInimigo, classeInimigo, cenario, tentarNovamente, jogadorDerrotado, inimigoDerrotado, fuga e todasAsPartes foram deletados, pois não são mais necessários devido às mudanças de lógica na classe Egine.
  - O atributo statsComNovoItem do tipo Stats foi criado para enviar os valores dos novos stats do jogador ao equipar um item deixado pelo inimigo.
  - O atributo todosOsItens foi mudado de ArrayList<Item> para TodosOsItens, pois esta nova classe já se encarrega da leitura dos arquivos de itens.

- O atributo parteDoCorpoDoInimigo do tipo int foi criado para uma referência de qual item do inimigo estamos verificando se foi deixado ou não.
- Os atributos moverJogador, jogadorErrou, moverInimigo e inimigoErrou do tipo booleano foram criados para indicar aos observadores o que ocorreu durante uma ação de luta.
- O atributo janelaAtual do tipo Janela foi criado para informar aos observadores qual tela deve ser apresentada ao usuário.
- O atributo listaDeObservadores do tipo ArrayList<Observador> para guardar a lista de objetos que receberão informações como se fossem interfaces de usuário.
- Os atributos novoJogador, novoInimigo, acaoDeLuta e escolherItem do tipo booleano foram criadas para indicar aos observadores qual foi a mudança no jogo (deixando que eles interpretem quais devem ser as variáveis que devem ser atualizadas).
- Todos os métodos privados passaram a ser protegidos.
- O construtor agora deve receber um parâmetro String com o nome do jogador, um parâmetro Mensagem com a classe que o jogador terá, um parâmetro int com a dificuldade que o jogo terá e um parâmetro com o Observador da Engine (pois a engine necessita de um observador para funcionar corretamente).
- O método iniciaNovoJogo foi deletado, pois não há mais necessidade dele.
- O método inicia jogador agora deve receber um parâmetro String com o nome do jogador e um parâmetro Mensagem com a classe que o jogador terá.
- O método abreArquivoitens foi deletado pois a classe TodosOsItens faz esta atividade agora.
- Os métodos ataqueJogador e ataqueInimigo mudaram de boolean para void, pois não há mais necessidade de um retorno sendo que todas as mensagens para os observadores são enviadas por outros métodos.
- Os métodos jogadorDerrotado, inimigoDerrotado e fugir mudaram de void para boolean, pois não há necessidade de guardar esta informação em uma variável.
- Os métodos loopDeJogo, atualizarCenario, monitoraCenario e main foi deletado, pois a lógica do funcionamento da Engine mudou.
- O método flagsParaCenario do tipo void com visibilidade protegida foi criado para atualizar as flags necessárias para a notificação dos observadores (cenários). Recebe como parâmetros quatro valores booleanos com os valores a serem escritos nos atributos novoJogador, novoInimigo, acaoDeLuta e escolherItem respectivamente.
- O método mensagemDoCenario do tipo void com visibilidade pública foi criado para receber uma mensagem de alguma classe que contenha um objeto do tipo Engine (geralmente deverá ser uma classe encarregada de ser uma interface de usuário). Recebe um parâmetro do tipo Mensagem que informa qual foi a decisão tomada pelo usuário, desta forma a classe segue a rotina de atividades devido a essa decisão.

- Os métodos getStatJogador e getStatInimigo do tipo int[] com visibilidade pública foram criados para enviar os valores atual e total respectivamente do jogador ou do inimigo. Recebe um parâmetro PosStats indicando qual stat que se deseja adquirir os valores.
- O método getLevel jogador do tipo int com visibilidade pública foi criado para enviar o level atual do jogador.
- O método getNomeItem do tipo String com visibilidade pública foi criado para enviar o nome dos itens utilizados pelo jogador atualmente. Recebe um parâmetro ParteDoCorpo indicando qual dos itens do jogador está sendo requerido.
- O método getNomeAtaque do tipo String com visibilidade pública foi criado para enviar o nome de cada um dos ataques do jogador. Recebe um parâmetro int indicando qual ataque é requerido.
- Os métodos getAcaoJogador e getAcaoInimigo do tipo boolean[] com visibilidade pública foram criados para indicar se o jogador ou o inimigo deferiram um ataque e se este ataque foi efetivo ou não (respectivamente representados por dois valores booleanos).
- O método getNomeInimigo do tipo String com visibilidade pública foi criado para enviar o nome do inimigo.
- O método getClasseInimigo do tipo Mensagem com visibilidade pública foi criado para enviar a classe do inimigo.
- O método getStatComNovoItem do tipo int com visibilidade pública foi criado para enviar o valor de um stat específico utilizando o item deixado pelo inimigo. Recebe um parâmetro PosStats indicando um stat específico a ser enviado.
- O método getNomeItemDeixado do tipo String com visibilidade pública foi criado para enviar o nome do item deixado pelo inimigo.
- O método getJanelaAtual do tipo Janela com visibilidade pública foi criado para enviar a janela que o jogador deve estar observando atualmente.
- O método getFlagsParaCenário do tipo boolean[] com visibilidade pública foi criado para enviar os valores atuais dos atributos novoJogador, novoInimigo, acaoDeLuta e escolherItem.
- Os métodos setAcaoJogador e setAcaoInimigo do tipo void com visibilidade protegida foram criados para mudar os valores das variáveis que indicam as ações do jogador e do inimigo durante uma luta. Recebem dois parâmetros booleanos dizendo se um golpe foi deferido e se ele foi efetivo.
- O método setJanelaAtual do tipo void com visibilidade protegida para mudar o valor do atributo janelaAtual.
- Sobrescreve os métodos adicionarObservador, removerObservador e notificarObservador da interface Observavel.
- O método fabricarClasse do tipo ClassePura com visibilidade pública foi criado para gerar a classe específica do jogador e do inimigo. Recebe um parâmetro

Mensagem informando a classe específica desejada. Possui uma sobrecarga que recebe um parâmetro int ao invés de Mensagem.

- Na classe GUI
  - Foram feitas várias mudanças para que o jogo funcionasse corretamente. Principalmente a criação de um método main, um atributo do tipo Engine e vários métodos envolvidos com a atualização dos valores a serem mostrados ao usuário. Além de implementar a interface Observador e sobrescrever o método update presente nesta interface (responsável pelo recebimento das informações do objeto Observavel).
- Na classe Personagem
  - O segundo parâmetro do construtor mudou de String para ClassePura, pois a classe Personagem não é mais responsável por criar o objeto ClassePura.
  - O método setCLasse foi deletado, pois não há mais necessidade dele.
  - O método getClasse foi mudado de String para Mensagem para padronizar a comunicação entre as classes.
  - O método setItem mudou de nome para equiparItem.
  - O método getAtaque retorna um objeto do tipo Ataque agora e não mais um PosStats[], para facilitar a comunicação e interpretação entre classes.
  - O método atualizarStats passou a ter visibilidade pública final ao invés de privada.
  - O método curarStatsTotalmente do tipo void com visibilidade pública foi criado para deixar os valores do atributo statsAtuais iguais aos do atributo stats.
- Na classe Stats
  - Agora implementa a interface Serializable para poder ser escrita em um arquivo binário.
  - O atributo todasPosicoes foi deletado, pois agora é utilizado o método values da enum PosStats.
- Na classe ClassePura
  - O atributo nome mudou de String para Mensagem para facilitar a comunicação entre classes.
  - O atributo fraqueza foi deletado, pois não há mais necessidade para ele.
  - O atributo ataques mudou de Ataque[] para ArrayList<Ataques> para facilitar a possibilidade de uma extensão futura do jogo.
  - O construtor tem um parâmetro a menos pois a fraqueza não é mais utilizada, e o primeiro parâmetro mudou de String para Mensagem para facilitar a comunicação entre as classes.
  - O método getNome mudou de String para Mensagem para facilitar a comunicação entre as classes.

- O método getFraqueza foi deletado, pois não há mais necessidade dele.
- O método getAtaque mudou de PosStats[] para Ataque para facilitar a comunicação e interpretação entre as classes.
- O método setAtaque agora recebe apenas um parâmetro do tipo Ataque para facilitar a comunicação e interpretação entre classes.

- Na classe Item
  - Agora implementa a interface Serializable para poder ser escrita em um arquivo binário.
  - O atributo classe mudou de String para Mensagem para facilitar a comunicação entre classes.
  - O décimo quarto parâmetro do construtor mudou de String para Mensagem para facilitar a comunicação entre classes.
  - O método getClasse mudou de String para Mensagem para facilitar a comunicação entre classes.
- Na enum ParteDoCorpo
  - Agora implementa a interface Serializable para poder ser escrita em arquivo binário.
- Na enum Mensagem
  - Agora implementa a interface Serializable para poder ser escrita em arquivo binário.
  - Possui novos valores de enum: ATAQUE("ATAQUE"), ATAQUE0("ATAQUE0"), ATAQUE4("ATAQUE4"), ATAQUE5("ATAQUE5"), ATAQUE6("ATAQUE6"), ATAQUE7("ATAQUE7"), ATAQUE8("ATAQUE8"), ATAQUE9("ATAQUE9") e ATAQUE10("ATAQUE10"), para uma possível futura expansão do jogo em questão do número de ataques de uma classe.
- Foi criada a enum PosAtaque com os valores ATACANTE(0), DEFENSOR(1) e DANIFICADO(2) para acesso a cada uma destas informações na classe Ataque, facilitando assim a comunicação entre as classes. Possui o atributo posição do tipo int e um método chamado valor que retorna o valor deste atributo.
- Na classe Ataque
  - O método getAtaque do tipo PosStats com visibilidade pública foi criado para enviar um dos valores do vetor de PosStats do objeto ataque específico. Recebe um parâmetro PosAtaque especificando qual valor é desejado (atacante, defensor ou danificado).
  - O método clone do tipo Ataque com visibilidade pública foi criado para criar e enviar um clone idêntico a este objeto.
- Foi criada uma interface chamada ClassePuraFactory para iniciação das classes de um personagem utilizando o Padrão de Projeto Factory Method. Esta

interface possui o método criaClassePura do tipo CLassePura com o parâmetro Mensagem (que deve especificar qual classe deve ser iniciada). Ela também possui uma sobrescrita deste mesmo método, porém com o parâmetro do tipo int ao invés de Mensagem.

- Foi criada uma classe chamada ClassePuraFactoryBasica que implementa a interface ClassePuraFactory para as três classes desta versão do jogo (Arqueiro, Cavaleiro e Mago).
- Foi criada a classe EngineBasica para sobrescrever os métodos abstratos da classe Engine utilizando a classe ClassePuraFactoryBasica para a iniciação das classes dos personagens.

9.2. Situação Atual

O jogo não está totalmente pronto, porém está funcional. O jogador já pode escolher seu nome, classe e dificuldade para começar a jogar. O sistema de ataques já funciona parcialmente.

Neste jogo um personagem pode ter até três ataques, porém todas as classes iniciam com apenas um ataque. Os novos ataques são liberados em diferentes levels para diferentes classes. Vale mencionar que se o jogador tentar atacar com um dos ataques que ainda não foram liberados ele perderá o seu turno. O mesmo ocorre quando o jogador tenta fugir do inimigo e não consegue. Se o jogador consegue fugir um novo inimigo é criado.

Ao final de cada inimigo derrotado é possível equipar novos itens. É possível chegar ao level máximo e terminar o jogo e então iniciar uma nova partida caso o usuário deseje isso.

9.3. Mudanças Futuras

As mudanças necessárias são:

Calibrar corretamente os valores dos ataques tanto do jogador quanto do inimigo.

Calibrar os valores de stats dados pelos itens.

Calibrar a porcentagem de acertos e erros de ataques.

Fazer melhorias na GUI.

Gerar novos itens para que se tenha uma gama maior de itens deixados pelo o inimigo.

Adicionar novos nomes de inimigos.

Verificar o porquê do jogo estar lento em execução.

10. Coliseum: Janelas do jogo

O jogo ainda está em desenvolvimento, porém já tem um protótipo funcional com todas as janelas necessárias para a interface com o usuário.

10.1. Nome Personagem

Nesta janela o jogador deverá escolher o nome pelo qual o seu personagem será identificado no jogo.

Possui uma caixa de texto para que o usuário insira o nome, e um botão para aceitar o nome e continuar o jogo.

10.2. Classe Personagem

Nesta janela o jogador deverá escolher a classe que o seu personagem terá (arqueiro, cavaleiro ou mago).

Possui três botões, um para cada classe. Cada botão tem como identificação a imagem do personagem da respectiva classe.

10.3. Dificuldade

Nesta janela o jogador deverá escolher a dificuldade do jogo (fácil, médio difícil).

Possui uma caixa de escolhas para as dificuldades e um botão para aceitar a dificuldade e continuar o jogo.

10.4. Cenário Principal

Esta janela apresenta as informações básicas do jogador (nome, classe, level e vida atual), do inimigo (nome e barra de vida), uma imagem para o jogador (à esquerda) e uma para o inimigo (à direita), e opções para atacar, fugir ou visualizar o menu.

Possui cinco botões. Um para cada um dos três ataques. Um para tentar fugir do inimigo. E um para visualizar o menu do jogador.

10.5. Menu Jogador

Nesta janela o jogador pode visualizar todas as suas informações: todos os valores de seus stats e os nomes dos seus itens atuais.

Possui um botão para retornar ao cenário principal.

10.6. Escolher Item

Esta janela ocorre quando um inimigo é derrotado e deixa um item para o jogador. Nela o jogador pode visualizar os valores de seus stats atuais e o valor de seus stats com o novo item. Ele também pode visualizar o nome do item e deve decidir se aceita equipar o novo item ou não. Depois de fazer a escolha o jogador é enviado ao cenário principal para enfrentar um novo inimigo.

Possui dois botões, um para aceitar o novo item e outro para rejeitar.

10.7. Inimigo Derrotado

Esta janela informa ao jogador que ele derrotou um inimigo e mostra uma imagem do jogador vitorioso e do inimigo derrotado.

Possui um botão para aceitar e continuar o jogo.

10.8. Vitória

Esta janela informa que o jogador venceu o jogo (atingiu o level 10) e mostra uma imagem do jogador vitorioso.

Possui um botão para iniciar um novo jogo.

10.9. Derrota

Esta janela informa ao jogador que ele foi derrotado por um inimigo e mostra uma imagem do jogador derrotado.

Possui um botão para tentar lutar contra o mesmo inimigo novamente e um botão para iniciar um novo jogo.

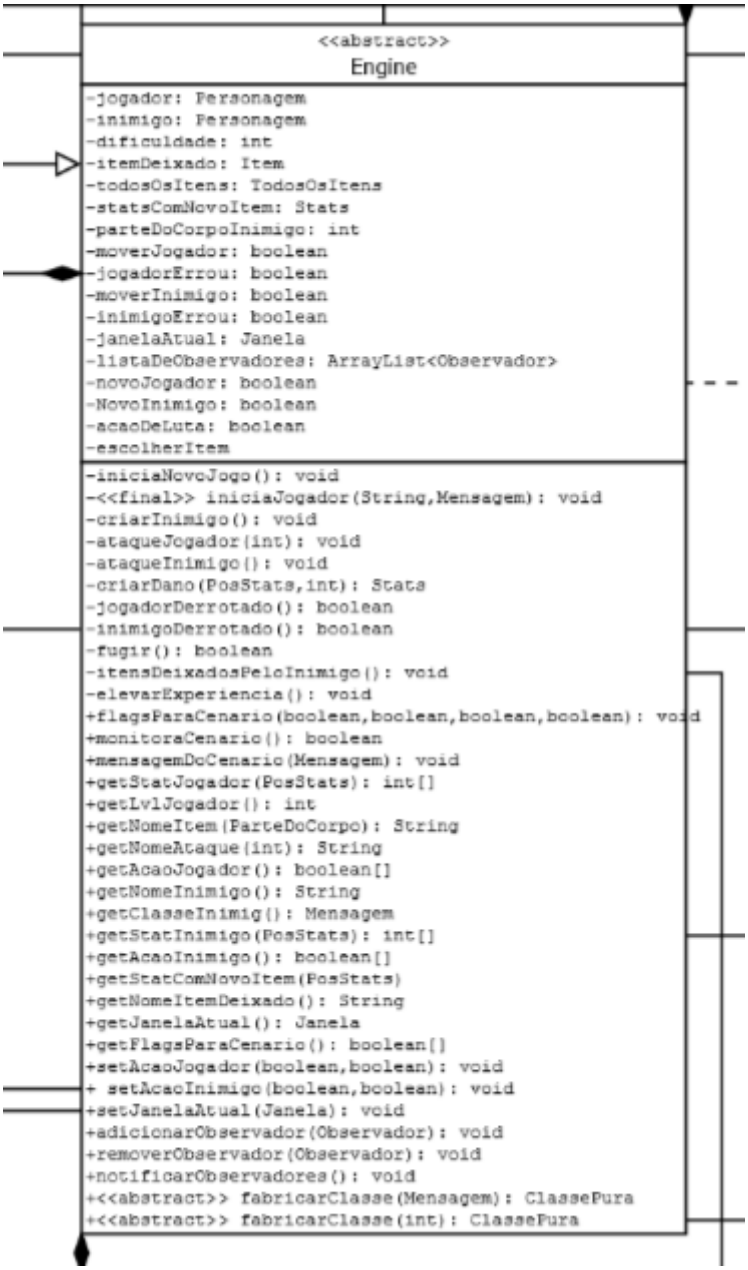
10.10. Fuga

Esta janela informa ao jogador que ele conseguiu fugir do inimigo e mostra uma imagem do jogador.

Possui um botão para aceitar e continuar o jogo.

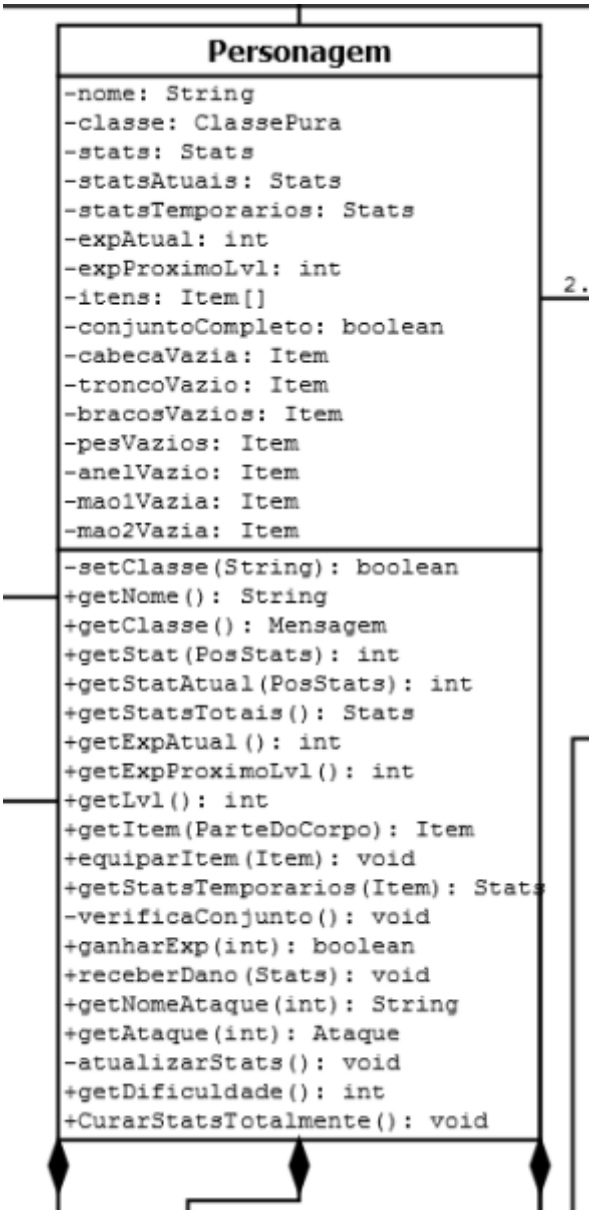
11. Coliseum: UML

Aqui serão apresentadas, separadamente, as classes presentes no projeto como um todo no esquema UML. O diagrama UML completo está presente no anexo A.



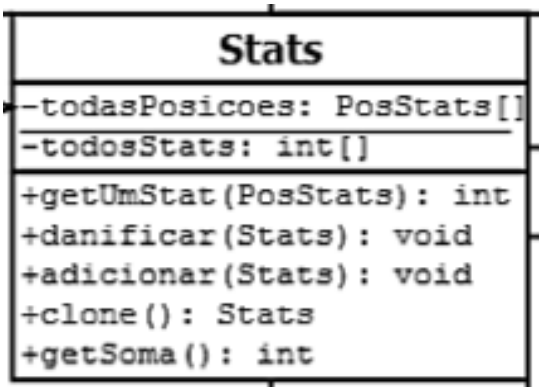
A Engine é responsável por gerenciar e organizar todas as informações do programa. Gerência os dados vindos da Observador e do usuário e aplica a lógica para controlar o personagem, e gera uma reação do inimigo. Implementa Observável.

Possui composição com as classes Personagem, ArrayList, e Item. Por fim, deve-se dizer que possui associação com Janela, TodosOsItens, Mensagem, PosStats, ParteDoCorpo e Dado. É uma classe abstrata.



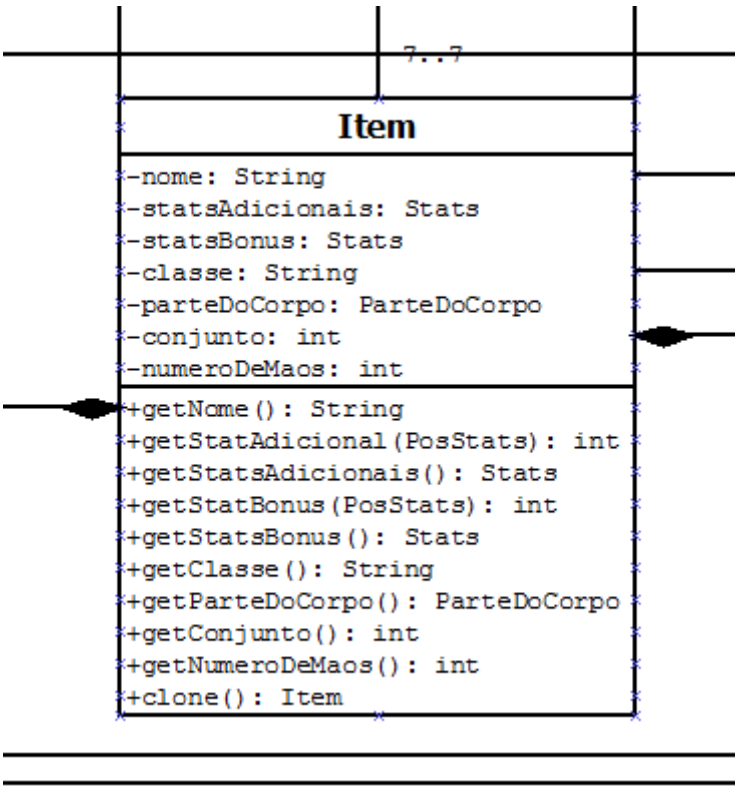
Contêm os atributos que compõem um personagem deste jogo. Também possui métodos que informam os valores destes atributo no momento que eles forem requisitados por outra classe e métodos que são capazes de modificar os valores dos atributos.

Existe relação de composição com as classes Item, ClassePura e Stats. E por fim, deve-se dizer que possui associação com PartesDoCorpo, Mensagem e Dado. É uma classe abstrata.



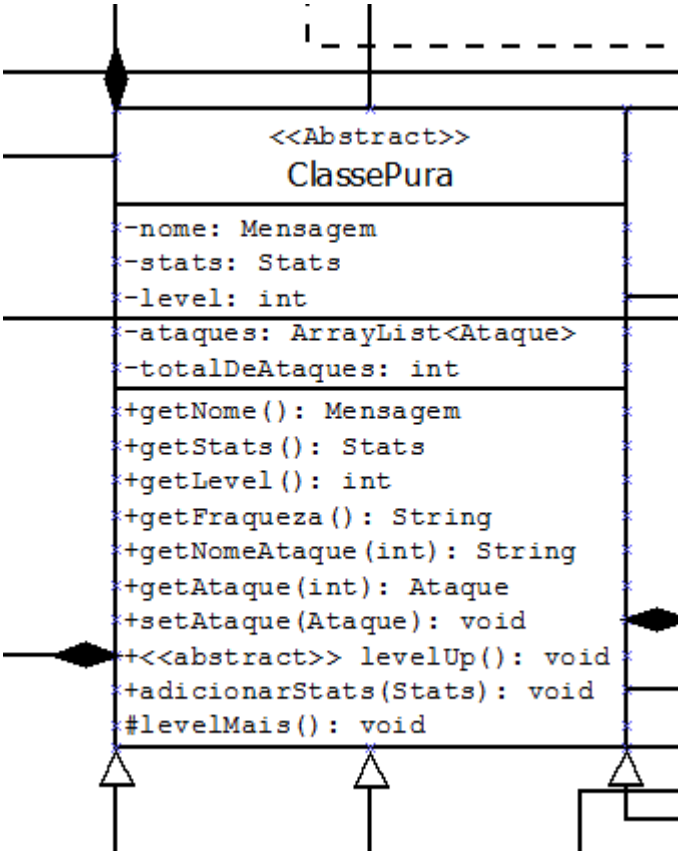
É responsável por definir os Stats dos personagens. Os Stats são: força, defesa, velocidade, magia, sorte e vida. Implementa Serializable.

Existe relação de composição com as classes Item, Personagem e ClassePura. Por fim, deve-se dizer que possui associação com Arqueiro, Cavaleiro, PosStats e Mago.



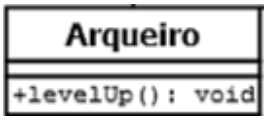
Contêm todas as características dos itens, define como o item será usado e diz qual o bônus que ele proporciona quando está com o conjunto a qual ele pertence. Implementa Serializable

Existe relação de composição com a classe Stats, Engine e TodosOsItens. Por fim, deve-se dizer que possui associação com Mensagem, PosStats e PartesDoCorpo.



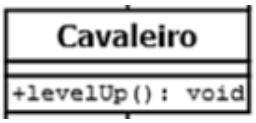
Define os atributos e o métodos principais de cada classe e que serão herdados pelas subclasses.

Existe relação de composição com a classe Stats, Ataque e ArrayList. Por fim, deve-se dizer que possui associação com PosStats, ClassePuraFactory e ClassePuraBasica.



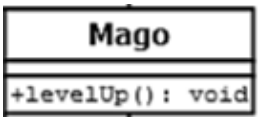
Herda da superclasse ClassePura e implementa o método que realiza o aumento de nível do personagem.

Possui relação de herança com ClassePura e associação com PosStats, PosAtaque, Stats, Dado e Stats.



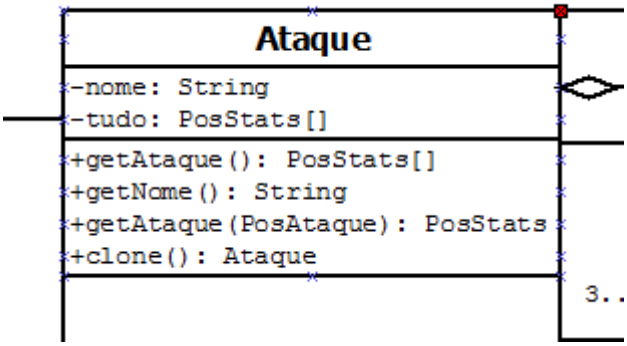
Herda da superclasse ClassePura e implementa o método que realiza o aumento de nível do personagem.

Possui relação de herança com ClassePura e associação com PosStats, PosAtaque, Stats, Dado e Stats.



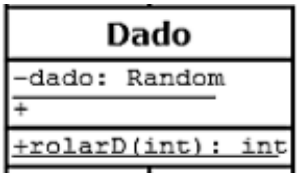
Herda da superclasse ClassePura e implementa o método que realiza o aumento de nível do personagem.

Possui relação de herança com ClassePura e associação com PosStats, PosAtaque, Stats, Dado e Stats.



É responsável por gerenciar os ataques e os identifica no vetor ataque.

Existe relação de Agregação com a classe PosStats e de Composição com ClassePura e associação Mago, Cavaleiro, Arqueiro, Personagem e PosAtaque.

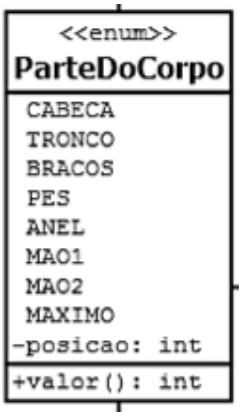


É responsável por gerar um número randômico dentro de um limite definido.

Há uma relação de associação com as classes Personagem, Engine, Mago, Cavaleiro e Arqueiro e também relação de composição com Random.

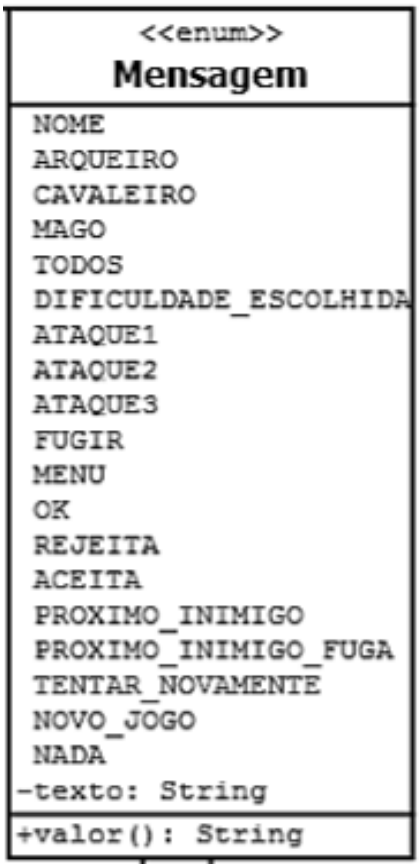
Gera identificadores para que se possa acessar o vetor que contém os Stats corretamente.

Possui associação com Engine, Arqueiro, Mago, Cavaleiro, ClassePura e Item.



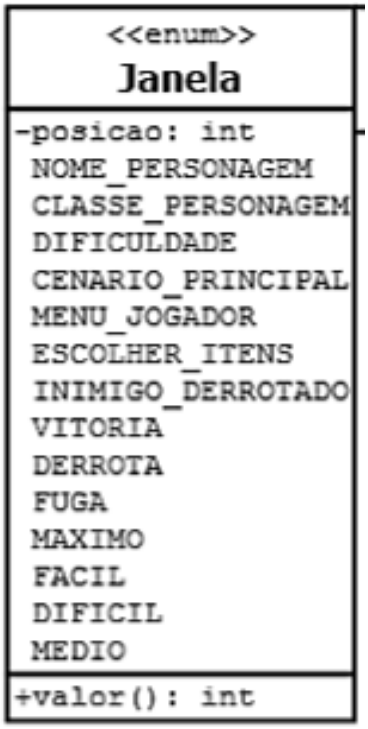
É responsável por gerar identificadores que organizam o acesso do vetor de itens, e assim, especifica-se em qual parte do corpo cada item deve ser utilizado.

Possui relação de associação com a Engine e Personagem.



A classe é utilizada para comparar qual foi a mensagem gerada pela ação do usuário na interface para então se toma uma decisão correta.

Possui relação de associação com a Personagem e GUI.

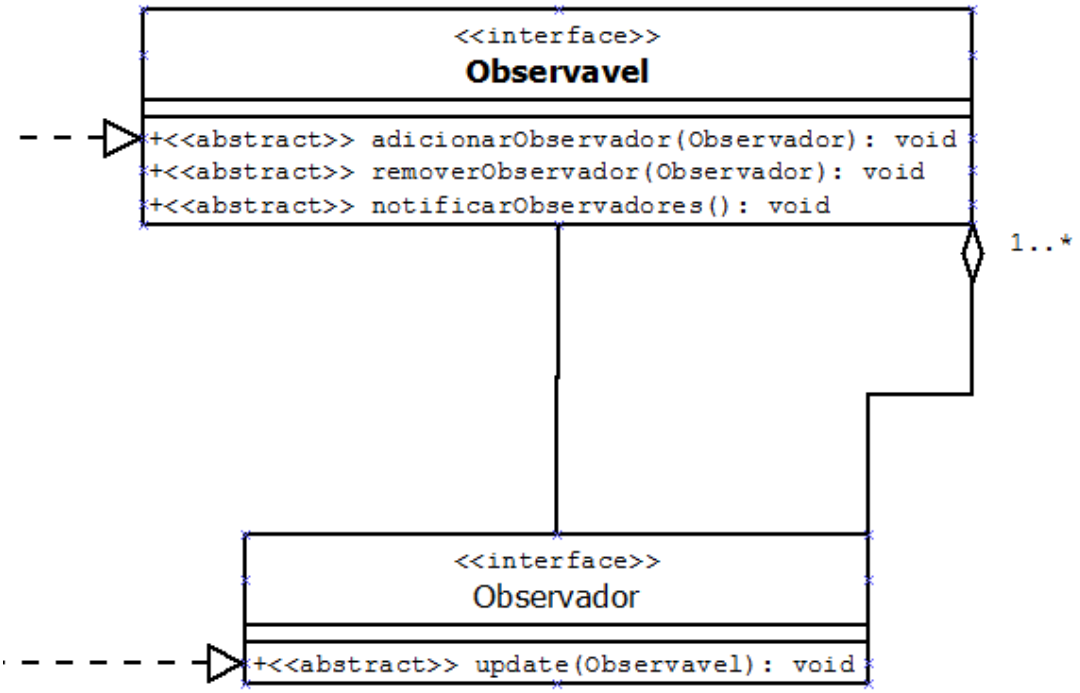


Gera identificadores que facilita o acesso do vetor que contêm as janelas da interface. Também possui os valores de referência para a dificuldade escolhida pelo usuário. Utilizado para a comunicação entre as classes Engine e GUI.

Possui relação de associação com a Engine e de composição com a GUI.

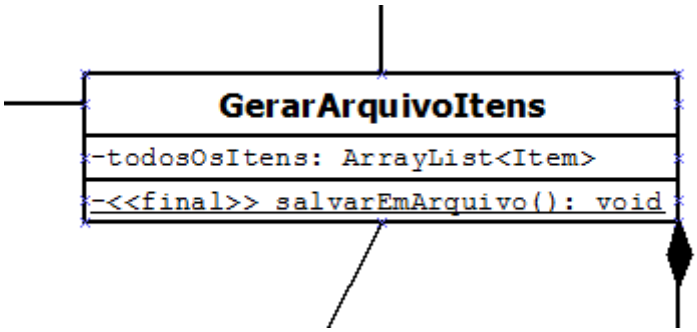
Responsável por mostrar as informações necessárias para o usuário através de uma interface gráfica. Também pede e recebe informações do usuário. Ela implementa a interface ActionListener para monitorar as ações produzidas pelos eventos gerados por cada botão da interface. Implementa Observador

Ela contêm relação de composição com as classes JButton, JFrame, JPanel, JLabel, JComboBox, BufferedImage, Color e Font e Engine.



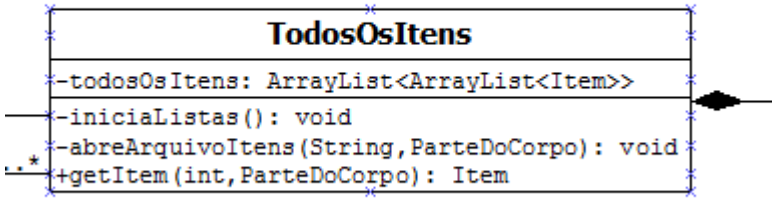
Essa duas classes criam o padrão de projeto do tipo Observador.Com ele a Engine consegue atualizar seus estados de acordo com a GUI, porém GUI e Engine não possuem relação entre si.

Observador se relaciona com observável com agregação e composição e o mesmo vale para o observável.



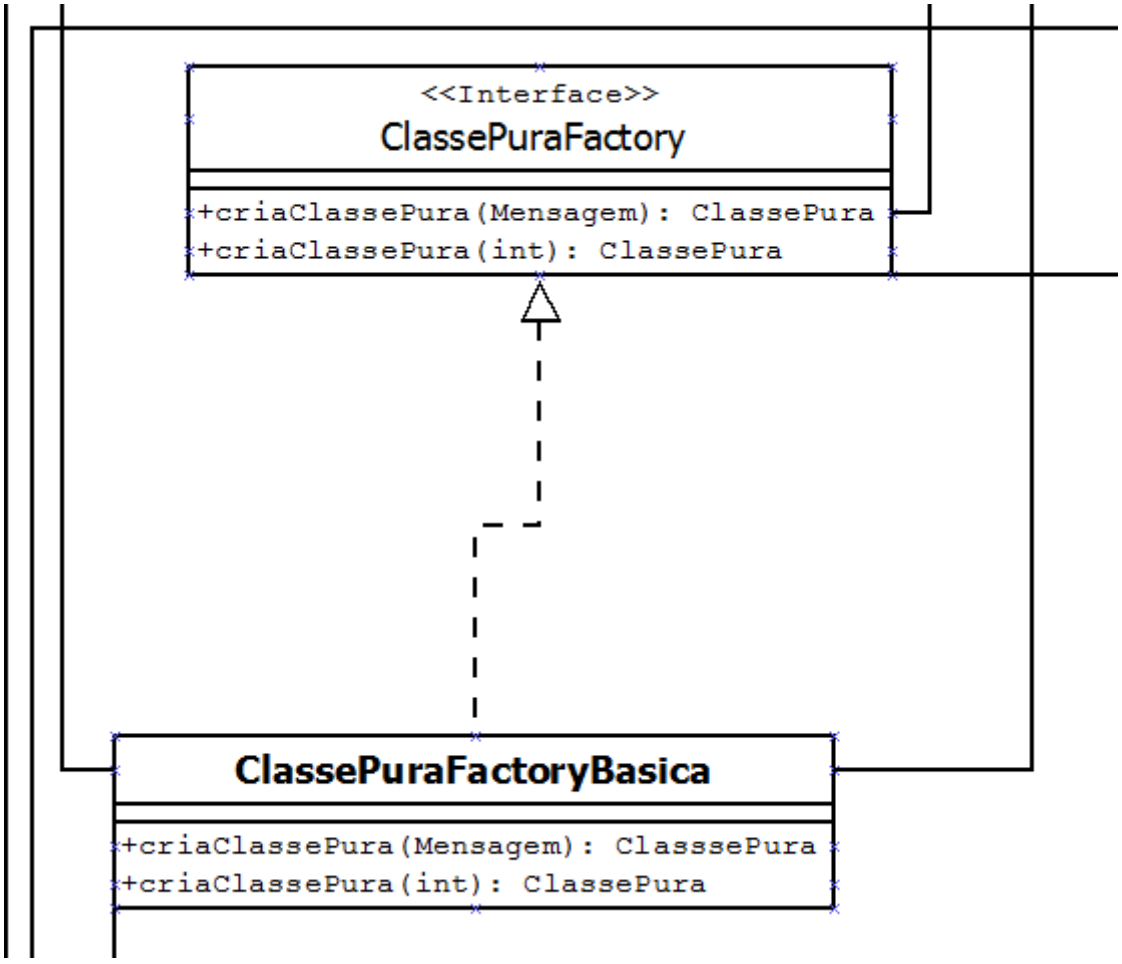
Guarda os itens em Arquivos.

Se relaciona com composição com ArrayList e associação com Mensagem, Item e PartedoCorpo.



Cria uma matriz de Itens para posteriormente abrir o arquivo de itens e passar este Item adiante.

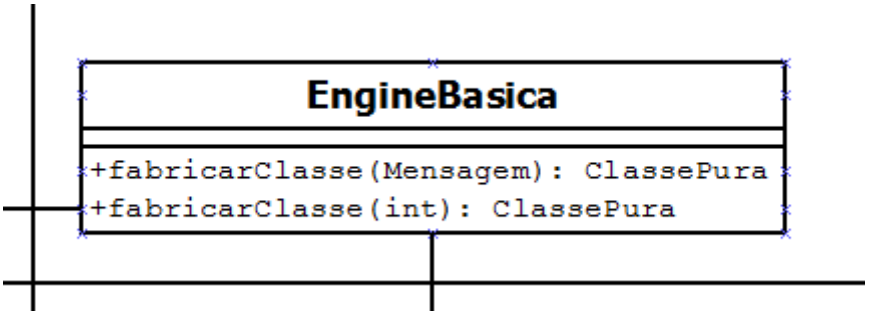
Se relaciona como composição com ArrayList e Item e como associação com a Engine.



Estas duas Classes implementam o padrão de projeto Factory.

ClassePuraFactoryBasica implementa ClassePuraFactory e tem associação com Mensagem, EngineBasica, Arqueiro, Cavaleiro e Mago.

ClassePuraFactory tem associação com EngineBasica e ClassePura.



É uma classe concreta que onde é possível criar um objeto engine e faz parte do padrão Factory.

Tem associação com ClassePuraFactory e herda de Engine.

12. Conclusão

Foram apresentados os conceitos necessários para aplicar a Programação Orientada a Objeto. Primeiramente estes conceitos foram utilizados para iniciar a programação de um jogo (estilo RPG arena). Foram apresentadas as mudanças feitas na versão anterior do jogo a situação atual de seu funcionamento e as perspectivas de mudanças para a próxima versão. Buscou-se consertar os erros cometidos na versão anterior e adicionar e implementar todos os métodos e atributos necessários para o funcionamento mínimo do jogo. Como resultado esperado para este trabalho, e que ao final de todas as etapas, é ter uma base sólida dos conceitos de OO e produzir algo para se entreter.

13. Referências

https://redmonk.com/sogrady/2015/07/01/language-rankings-6-15/ - Ranking de linguagens mais utilizadas para programação. Acessado pela última vez em 01/02/2016 – 08:00.

http://www.ufpa.br/cdesouza/teaching/es/3-OO-concepts.pdf - Slides professor Prof. Dr. Cleidson Souza, UFPA. Acessado pela última vez em 01/02/2016 - 08:00.

http://www.dca.fee.unicamp.br/cursos/PooJava/Aulas/poojava.pdf - Apostila "Programação orientada a objetos: Uma abordagem com Java" – Prof. Dr. Ivan Luiz Marques Ricarte, 2001, FEEC UNICAMP. Acessado pela última vez em 01/02/2016 – 08:00

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html - Explicação dos tipos de visibilidade para Java, Oracle. Acessado pela última vez em 01/02/2016 – 08:00.

http://www.professorvida.com.br/if62c/material/relacionamentos.pdf - Apostila com Tema:" Relacionamento entre Classes" – Prof. Dr. Robinson Vida Noronha, UTFPR. Acessado pela última vez em 01/02/2016 – 08:00.

https://docs.oracle.com/javase/tutorial/java/concepts/interface.html - Explicação do conceito de interface para Java, Oracle. Acessado pela última vez 01/02/2016 – 08:00

BARMES, David J.; KÖLING, Michael. Programação orientada a objetos com Java: uma introdução prática usando o BlueJ. 4. ed. São Paulo, SP: Pearson Prentice Hall, 2009. 455 p. ISBN 9788576051879.

http://www.cin.ufpe.br/~rcmg/cefet-al/proo/aulas/proo_05_heranca_ligdinamica_6pp.pdf - Slides – Prof. Dr. Reinaldo Gomes – UFPE, atualmente UFCG.Acessado pela última vez 01/02/2016 – 08:00

www2.ic.uff.br/~fabio/CAP10.ppt – Slides – Prof. Dr. Fábio Protti – UFF. Baixado pela última vez 01/02/2016 – 08:00

http://www.inf.pucrs.br/~flash/lapro2/lapro2_eventos.pdf - Prof. Dr. Marcelo Cohen - PUCRS. Acessado pela última vez 01/02/2016 – 08:00

http://www.inf.puc-rio.br/~inf1301/docs/INF1301_Aula24_Modular_Programming_Design_Patterns_2010.pdf - Prof. Dr. Alessandro Garcia – PUC-Rio. Acessado pela última vez 06/03/2016 – 08:00

http://homepages.dcc.ufmg.br/~figueiredo/disciplinas/2012a/reuso/reuso-aula03.pdf - Prof. Dr. Eduardo Figueiredo – UFMG. Acessado pela última vez 06/03/2016 – 08:00

http://www.dsc.ufcg.edu.br/~jacques/cursos/map/html/arqu/observer.htm - Prof. Dr. Jacques Philippe Sauvé – UFCG. Acessado pela última vez 06/03/2016 -08:00