**Classe abstrata Item (Item.h e Item.cpp)**

Todo item, tanto equipamento e itens de cura quanto itens chaves, são derivados desta classe abstrata. Todos os itens possuem:

* Uma **string** , que será a sua forma de identificação no programa.
* Uma **string para descrição**, que seria uma breve explicação sobre o que faz e/ou sua importância para a estória.
* Um **inteiro para a durabilidade**, que seria quantas vezes o item seria utilizado até ser quebrado. Esta variável é mais voltada para os objetos da classe derivada “**Equipamento**” do que para os itens em si
* Uma **variável booleana para verificar se é consumível.** Caso sim, ao ser utilizado, o tem faz seu efeito e saí do inventario da classe “**Grupo**”. Caso não, o item pode ser utilizado diversas vezes (Ou até nem ser utilizado pelo jogador, só servindo como objeto chave para se ativar algum evento).

Este método possui os seguintes construtores:

* **Construtor default**, com valores padrão para o nome e a descrição.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Construtor inicializando e validando os valores** “nome”, “descrição”, “durabilidade” e “consumibilidade”.
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Diminuir\_Durabilidade()**, responsável por diminuir a durabilidade dos itens.
* **virtual void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.

**Classe concreta Kuraudo\_Mirrors (Kuraudo\_Mirrors.h e Kuraudo\_Mirrors.cpp).**

**Classe base: Item**

Esta classe não tem muita utilidade ao programa, servindo apenas como um item-chave (que tem a função de ativar algum evento, ou de simbolizar algum grande feito do jogador).

Kuraudo\_Mirrors possui:

* Atributos da classe **Item**( Nome, descrição, etc.).

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Item** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor vazio**, com valores pré-determinados.
* **Construtor de cópia**, para construir um objeto usando outro objeto desta mesma classe como referência.
* **Destrutor**.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.
* **Void Diminuir\_Durabilidade() :** Diminui a durabilidade do item. Esta função sobrescreve a mesma da classe base.

**Classe abstrata Equipamento (Equipamento.h e Equipamento.cpp).Classe base: Item**

**A Classe Equipamento** é a classe abstrata que corresponde a todos e qualquer equipamento (Arma, Armadura e Acessório). Ela é derivada da classe **Item** e é base para as classes abstratas **“Arma”, “Armadura”e “Acessórios”.**

Equipamento contém:

* Atributos da classe **Item**( Nome, descrição, etc.).
* **Inteiros para causar um bônus nos atributos ataque, defesa magia e velocidade**, que se somará ao valor total dos atributos do herói.

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Item** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor default**, com valores padrão para o nome e a descrição.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Construtor default**, com valores padrão para um único bônus.
* **Construtor default**, com valores padrão para dois bônus.
* **Construtor de cópia**, para copiar os valores de outro Equipamento. Não é necessário validar os dados.
* **Construtor inicializando e validando os valores** “nome”, “descrição”, “durabilidade” e “consumibilidade”, “bônus de ataque”, “bônus de defesa”,“bônus de magia”,“bônus de velocidade”.
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Add\_Bonus(),** que Adiciona um bônus com valor maior ou igual a 0 a um equipamento.
* **virtual void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.

**Classe abstrata Arma (Arma.h e Arma.cpp). Classe base: Equipamento**

Com poucas exceções (ex: Pokemon), quase todos os personagens jogáveis de RPG possuem uma arma, diferenciando-se na forma de como esta é trabalhada (se ela é possui atributos fixos, se ela pode ser trocada, se ela pode ser fundida com outra).

Arma contém:

* Atributos da classe **Item**e da classe **Equipamento**.

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Equipamento** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor default**, com valores padrão para um único bônus.
* **Construtor de cópia**, para copiar os valores de outra Arma. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os valores necessários.**
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Add\_Bonus(),** que Adiciona um bônus com valor maior ou igual a 0 a um equipamento.
* **virtual void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.

**Classe abstrata Armadura (Armadura.h e Armadura.cpp). Classe base: Equipamento**

Idem a descrição da classe **Arma**.

Armaduracontém:

* Atributos da classe **Item**e da classe **Equipamento**.

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Equipamento** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor default**, com valores padrão para um único bônus.
* **Construtor de cópia**, para copiar os valores de outra Armadura. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os valores necessários.**
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Add\_Bonus(),** que Adiciona um bônus com valor maior ou igual a 0 a um equipamento.
* **virtual void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.

**Classe abstrata Acessorio (Acessorio.h e Acessorio.cpp). Classe base: Equipamento**

Idem a descrição da classe **Arma e Armadura**, porém é mais comum de ser encontrada em Rpgs.

Acessorio contém:

* Atributos da classe **Item**e da classe **Equipamento**.

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Equipamento** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor default**, com valores padrão para um único bônus.
* **Construtor de cópia**, para copiar os valores de outro Acessório. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os valores necessários.**
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Add\_Bonus(),** que Adiciona um bônus com valor maior ou igual a 0 a um equipamento.
* **virtual void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.

**Classes concretas Espada, Cajado, Faca, Roupas\_Leves, Anel\_De\_Forca.**

**Classe Base: Arma(Espada, Cajado, Faca), Armadura(Roupas\_Leves), Acessorio(Anel\_De\_Forca)**

Classes concretas da hierarquia equipamento. Nota-se que elas também podem ser classes abstratas, servindo como base para outras classes (Ex: Espada de Fogo e/ou Armadura de Prata).

Todas estas classes possuem:

* Atributos da classe **Item**e da classe **Equipamento**.

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Arma ,Armadura ou Acessorio (dependendo doa classe base DAE cada uma)** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor default**, com valores padrão para um único bônus.
* **Construtor de vazio,** com valores pré-determinados**.**
* **Destrutor.**

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **void Add\_Bonus(),** que Adiciona um bônus com valor maior ou igual a 0 a um equipamento.
* **void Efeito() = 0**, método virtual puro indicando qual o efeito( Ex: Causar dano, adicionar atributos positivos a um herói, etc). Cada item terá seu próprio efeito, por isso ela é uma função virtual.

**Classe abstrata Magia (Magia.h e Magia.cpp).**

Magias utilizadas pelos magos e monstros.

Magia contém:

* Uma **string nome** , que será a sua forma de identificação no programa.
* Uma **string para descrição**, que seria uma breve explicação sobre o que faz e/ou sua importância para a estória.
* Um inteiro para o poder da magia, que, somada a magia do mago, será utilizada para causar dano a um monstro
* Uma **string para o elemento,** que verificaria a fraqueza ou a resistência de certo monstro a um tipo específico de magia.

Este método possui os seguintes construtores:

* **Construtor default**, com valores padrão para um único bônus.
* **Construtor de cópia**, para copiar os valores de outra Magia. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os valores necessários.**
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual int Efeito() = 0**, compartilhando o nome e a função com um método da hierarquia Item, retornando um inteiro com o resultado do cálculo para saber qual o poder de ataque da magia.

**Classe abstrata Trovao (Trovao.h e Trovao.cpp).**

Uma magia derivada da classe Magia.

Trovao contém:

* Mesmos atributos da classe abstrata **Magia**.

Este método possui os seguintes construtores:

* **Construtor Vazio,** iniciando os atributos com valores pré-determinados
* **Construtor de cópia**, para copiar os valores de outra Magia. Não é necessário validar os dados.
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **int Efeito()**, compartilhando o nome e a função com um método da hierarquia Item, retornando um inteiro com o resultado do cálculo para saber qual o poder de ataque da magia.

**Classe abstrata Heroi (Heroi.h e Heroi.cpp)**

Esta classe abrange o “representante virtual” do jogador em um RPG. Desta classe não surge só o personagem em si, mas as especializações que o usuário pode assumir (Guerreiro, Mago ou Ladino, por exemplo)

Todo Herói deve ter:

* Uma **string para nome**, que será a sua forma de identificação no programa.
* Uma **string para descrição**, que seria uma breve explicação sobre o que faz e/ou sua importância para a estória.
* Um **inteiro para o nível do jogador**, que representa quanto ele melhorou suas habilidades durante o jogo.
* Um **inteiro estático e global** para determinar quanto o herói pode evoluir no jogo.
* **Referências às classes abstratas “Arma”, “Armadura” e “Acessorio”,** representando os equipamentos que o herói possui.
* Um **inteiro representando os pontos de experiência do Herói**, e este serve para fazer o personagem subir de nível.
* Um **inteiro representando o HP e MP**, indicando o HP e o MP total que o jogador têm.
* Um **inteiro constante e estático representando o HP e MP máximo**, que é o máximo de HP e MP que um personagem pode ter.
* Um **inteiro representando o HP e MP atual**, que é o HP atual do Herói. Estes não podem ser maior que os atributos HP (para HP\_Atual) e MP (para MP\_Atual). Estes atributos são utilizados para batalhas.
* **Inteiros para os atributos do** herói, e estes atributos são: **Ataque, Defesa, Magia e Velocidade.**

Este método possui os seguintes construtores:

* **Construtor default**, com valores padrão para o nome e a descrição.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os atributos.**
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **virtual void Menu\_Ataque()**, será o menu que aparecerá quando for o turno do Herói.A função é virtual porque cada especialização podem ter opções diferentes (Mago usar magia, por exemplo).
* **virtual void Subir\_De\_Nivel(constint&),** recebendo a experiência do monstro derrotado, esta função diminui quanto falta para o próximo aumento de nível, caso a experiência necessária chegue a 0. O método é virtual pois cada classe vai ter sua forma de aumentar os atributos.
* **virtual void Equipar(const Equipamento&),** passando um equipamento, o método verifica se aquele equipamento é válido e se é compatível com a classe (Espada para Guerreiros, Cajado para magos, e assim vai). Método virtual, pois cada um vai ter sua própria validação.
* **StaticvoidDef\_Stats(),** define os atributos iniciais para todos as especializações.

**Classes concretas Mago, Guerreiro e Ladino.**

**Classe Base: Heroi**

Indicam as especializações dos heróis

Todas estas classes possuem:

* Atributos da classe **Heroi.**
* Um **vetor de Magias, exclusivo a classe Mago.**

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Heroi** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor vazio**, com valores pré-determinados**.**
* **Construtor que programa e valida todos os atributos.**
* **Construtor de cópia.**
* **Destrutor.**

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **void Menu\_Ataque()**, será o menu que aparecerá quando for o turno do Herói.
* **void Subir\_De\_Nivel(constint&),** recebendo a experiência do monstro derrotado, esta função diminui quanto falta para o próximo aumento de nível, caso a experiência necessária chegue a 0
* **void Equipar(const Equipamento&),** passando um equipamento, o método verifica se aquele equipamento é válido e se é compatível com a classe (Espada para Guerreiros, Cajado para magos, e assim vai).
* **Static void Def\_Stats(),** define os atributos iniciais para todos as especializações.
* **void Show\_Magics(), exclusiva a especialização Mago,** que mostra todas as magias desta classe
* **void Roubar(const Monstro&), exclusiva a especialização ladino,** que tenta pegar o item o monstro alvo .

**Classe abstrata Monstro (Monstro.h e Monstro.cpp)**

Apesar do nome, esta hierarquia determina qualquer vilão que o jogador encontre durante a sua jogatina.

Todo Monstro deve ter:

* Uma **string para nome**, que será a sua forma de identificação no programa.
* Uma **string para descrição**, que seria uma breve explicação sobre o que faz e/ou sua importância para a estória.
* Um **inteiro representando os pontos de experiência que o Herói ganhará**, e este serve para fazer o personagem subir de nível.
* Um **inteiro representando o HP e MP**, indicando o HP e o MP total que o monstro têm.
* Um **inteiro constante e estático representando o HP e MP máximo**, que é o máximo de HP e MP que um monstro pode ter.
* Um **inteiro representando o HP e MP atual**, indicando o HP e MP atual que o monstro possui. Utilizado normalmente nas batalhas.
* **Inteiros para os atributos do** herói, e estes atributos são: **Ataque, Defesa, Magia e Velocidade.**
* Um **objeto do tipo Item**, sendo este o item que a especialização Ladina pode conseguir caso use o comando roubar nesta classe.
* Um **vetor com as magias do monstro.**

Este método possui os seguintes construtores:

* **Construtor default**, com valores padrão para o nome e a descrição.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os atributos.**
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **voidAtacar()**, será o conjunto de ações que o monstro tomará durante o seu turno. Ele pode atacar, usar magia ou simplesmente não fazer nada.
* **Static void Def\_Stats(),** define os atributos iniciais para todos as especializações, similar ao método de mesmo nome da classe Heroi.

**Classes concretas Kuraudo e Camtasia.**

**Classe Base: Monstro**

Classes deste tipo éum monstro que o jogador encontrará durante o jogo.

Todas estas classes possuem:

* Atributos da classe **Monstro.**

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Monstro** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor vazio.**
* **Construtor que implementa e valida todos os atributos.**
* **Construtor de cópia.**
* **Destrutor.**

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **Static void Def\_Stats(),** define os atributos iniciais para todos as especializações, similar ao método de mesmo nome da classe Heroi.

**Classe abstrata Cidade (Cidade.h e Cidade.cpp)**

Cidades ou localidades seguras para os jogadores pertencerão a esta Hierarquia. Possuem normalmente moradores com quem o jogador pode conversar vendedores de equipamentos, locais para salvar o jogo, e outras funcionalidades.

Toda Cidade deve ter:

* Uma **string para nome**, que será a sua forma de identificação no programa.
* Um vetor de **string para armazenar as falas dos moradores.**

Este método possui os seguintes construtores:

* **Construtor default**, com valores padrão para o nome.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **void Add\_Npc()**,armazenará uma mensagem que será apresentada ao jogador quando ele for falar com o morador. Cada cidade tem um limite para o número de moradores.
* **virtual void Menu\_Cidade(),** será aonde o jogador poderá escolher se quer falar com o vendedor, ou se quiser ir para o calabouço. Como o método é virtual, cada cidade pode ter seu menu próprio.
* **Virtual void Menu\_Texto\_Cidade(),** mostrará um pequeno menu, funcionando como auxiliar da classe Menu\_Cidade().

**Classe concreta Sepolopolis.**

**Classe Base: Cidade**

Uma das cidades que o jogador encontrará. Não possui muita coisa.

Sepolopolis terá:

* Atributos da classe **Cidade.**

Este método possui os seguintes construtores. Todos eles chamam o construtor da classe **Cidade** para realizar o reaproveitamento de código, com exceção do destrutor:

* **Construtor default.**
* **Destrutor.**

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **void Add\_Npc()**,armazenará uma mensagem que será apresentada ao jogador quando ele for falar com o morador. Cada cidade tem um limite para o número de moradores.
* virtual void Menu\_Cidade(), será aonde o jogador poderá escolher se quer falar com o vendedor, ou se quiser ir para o calabouço. Como o método é virtual, cada cidade pode ter seu menu próprio.
* **Virtual voidMenu\_Texto\_Cidade(),** mostrará um pequeno menu, funcionando como auxiliar da classe Menu\_Cidade().
* **void Vendedor(Grupo&),** contém o que o menu de ações do vendedor, aonde o jogador pode comprar itens

**Classe abstrata Dungeon (Dungeon.h e Dungeon.cpp)**

Enquanto que a hierarquia superior representava os locais seguros aonde o jogador passará, esta hierarquia representa o inverso, indicando os calabouços, locais totalmente inseguros, e cheios de batalhas aleatórias.

Toda Dungeon deve ter:

* Uma **string para o** **nome**, que será a sua forma de identificação no programa.
* Um **inteiro** para armazenar o número de salas do calabouço.
* Um **inteiro constante e estático** para determinar o número máximo desalas que uma dungeon pode ter.

Este método possui os seguintes construtores:

* **Construtor default**, com valores padrão para o nome e o número de salas.
* **Construtor que implementa e valida todos os atributos da classe.**
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **Virtual void Salas(Grupo&)**, mostra as salas e suas opções. Cada sala não é nada mais do que um menu aonde o jogador escolhe se quer avançar ou não. A cada movimentação, existe uma chance de um encontro aleatório aparecer. É auxiliado pelo método **Mostrar\_Salas(const int&)**.
* **virtual void Mostrar\_Salas(const int&),** que evita a repetição de vários cout << no código, tornando-o mais limpo. Este método mostra o menu baseado no índice da sala, assim auxiliando o método **void Salas(Grupo&)**.
* **Static void Verificar\_Num\_Salas(constDungeon&)**., que mostra o número de salas que a dungeon possui.

**Classe concreta Caverna\_Monstro.**

**Classe base: Dungeon**

Umas das dungeons do jogo. Não possui nada de especial

Caverna\_Monstro possui:

* Atributos da classe base **Dungeon**.

Este método possui os seguintes construtores. É de se notar que esta classe chama o construtor da classe **Dungeon**, quando possível:

* **Construtor default**, com valores padrão para o nome e o número de salas.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Destrutor virtual**, para chamar os destrutores das classes derivadas.

Além dos métodos “get()”, “set()” e operad ores += e << para as variáveis acima, temos os métodos:

* **void Salas(Grupo&)**, mostra as salas e suas opções. Cada sala não é nada mais do que um menu aonde o jogador escolhe se quer avançar ou não. A cada movimentação, existe uma chance de um encontro aleatório aparecer. É auxiliado pelo método **Mostrar\_Salas(const int&)**.
* **Void Mostrar\_Salas(const int&),** que evita a repetição de vários cout << no código, tornando-o mais limpo. Este método mostra o menu baseado no índice da sala, assim auxiliando o método **void Salas(Grupo&)**.

**Includer.h**

É um header sem atributos ou funções. Só tem o objetivo de organizar todos as referências “#include” e um lugar, tornando o código limpo.

**Classe concreta Grupo (Grupo.h e Grupo.cpp)**

Em alguns Rpgs, o jogador pode controlar diversos heróis e possui um inventário. Esta classe representa ambos.

Todo Grupo deve ter:

* Um vetor de Heróis para armazenar os integrantes do grupo.
* Um vetor de Itens para armazenar o inventário do grupo
* Um inteiro para armazenar o dinheiro da equipe.
* Dois inteiros para armazenar o número máximo de heróis e o número máximo de itens.
* Uma **Typedef struct chamada Posicao**, que armazena a coordenada X e coordenada Y do herói em um mapa.

Este método possui os seguintes construtores:

* **Construtor vazio**, com os valores pré-implementados.
* **Construtor de cópia**, para copiar os valores de outro item. Não é necessário validar os dados.
* **Construtor inicializando e validando todos os atributos.**
* **Destrutor**.

Além dos métodos “get()”, “set()” e operadores += e << para as variáveis acima, temos os métodos:

* **void Atacar(Heroi\*, Monstro\*)**, será o menu que aparecerá quando for o turno do Herói.Através de um dynamic\_cast é identificado qual tipo de especialização o herói possui e qual menu de contexto deve ser chamado.
* **void Batalhar(Monstro\*),** recebendo um monstro como parâmetro, a batalha se desenvolve neste método, chamando os método ”Atacar” de “Grupo” quando a situação necessitar.
* **void Ordenar\_Herois(),** que ordena os heróis através da velocidade deles.
* **void Mostrar\_Herois(),** mostra todo os heróis do Grupo.
* void Mostrar\_Inventario(), mostra todos os Itens do inventario.
* **Void Usar\_Item(),** que chama um efeito de um item, retirando-o do inventario caso este seja consumível.
* **void Text\_Menu():** Mostra o menu do grupo, servindo de suporte para as outras classes.