**TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO DA UTFPR:**

**MOUNT**

Vitor Kenji Zoppo Yamada

vitoryamada@alunos.utfpr.edu.br

Disciplina: Técnicas de Programação – CSE20 / S71 – Prof. Dr. Jean M. Simão

Departamento Acadêmico de Informática – DAINF - Campus de Curitiba

Curso Bacharelado em: Engenharia da Computação / Sistemas de Informação

Universidade Tecnológica Federal do Paraná - UTFPR

Avenida Sete de Setembro, 3165 - Curitiba/PR, Brasil - CEP 80230-901

**Resumo** – A disciplina de Técnicas de Programação exige o desenvolvimento de um software de plataforma, no formato de um jogo, para fins de aprendizado de técnicas de engenharia de software, particularmente de programação orientada a objetos em C++. Para tal, neste trabalho, escolheu-se o jogo Mount, no qual o jogador enfrenta inimigos em um dado cenário. O jogo possui duas fases que se diferenciam por dificuldades para o jogador, exigindo habilidades e raciocínio. Para o desenvolvimento do jogo foram considerados os requisitos textualmente propostos e elaborado modelagem (análise e projeto) via Diagrama de Classes em Linguagem de Modelagem Unificada (Unified Modeling Language - UML) usando como base um diagrama genérico e previamente proposto. Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento contemplando os conceitos usuais de Orientação a Objetos como Classe, Objeto e Relacionamento, bem como alguns conceitos avançados como Classe Abstrata, Polimorfismo, Gabaritos, Persistências de Objetos por Arquivos, Sobrecarga de Operadores e Biblioteca Padrão de Gabaritos (Standard Template Library - STL). Depois da implementação, os testes e uso do jogo feitos pelo próprio desenvolvedor demonstraram sua funcionalidade conforme os requisitos e a modelagem propostos. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado.

**Palavras-chave ou Expressões-chave:** Artigo-Relatório Modelo para o Trabalho em Fundamentos de Programação 2, Trabalho Acadêmico Voltado a Implementação em C++, Normas Internas para Elaboração de Trabalho, Exemplo de Conteúdos de Trabalho de Fundamentos de Programação 2.

**Abstract:** The Programming Techniques discipline requires the development of platform software, in the form of a game, for the purpose of learning software engineering techniques, particularly object-oriented programming in C++. For such, in this work, the Mount game was chosen, in which the player faces enemies in a given scenario. The game has two phases that differ by difficulty for the player, requiring skills and reasoning. For the development of the game, the textually proposed requirements were considered and modeling (analysis and design) was elaborated via Class Diagram in Unified Modeling Language (UML) using a generic and previously proposed diagram as a basis. Subsequently, in C++ programming language, development was carried out contemplating the usual Object Oriented concepts such as Class, Object and Relationship, as well as some advanced concepts such as Abstract Class, Polymorphism, Templates, Object Persistence by Files, Operator Overloading and Standard Template Library (STL). After the implementation, the tests and use of the game made by the developer demonstrated its functionality according to the proposed requirements and modeling. Finally, it should be noted that the development in question allowed fulfilling the learning objective.

**Key-words or Key-expressions:** Paper Model to the Academic Work of Programming Course, Academic Work Related to C++ Implementation, Internal Rules for Work Elaboration, Examples of Elements for the Work of a Programming Course.

# **INTRODUÇÃO**

Este documento é um relatório do trabalho para a disciplina de Técnicas de Programação do curso de Engenharia de Computação da UTFPR, referente ao jogo “Mount”, desenvolvido em programação orientada a objetos em C++, pelo aluno Vitor Kenji Zoppo Yamada. Para esse propósito, são apresentados e explicados os tópicos contextualizados no âmbito da elaboração de um jogo.

O trabalho objetiva principalmente a ampliação e aplicação dos conceitos aprendidos em classe e extraclasse, ao longo do período acadêmico, desafiado no desenvolvimento de um jogo de plataforma, previamente acordado com o professor, com requisitos mínimos e premissas, juntamente com a utilização de diversos recursos da linguagem de programação. Isso propicia a ampliação dos conceitos e aprimoramento das habilidades através do cumprimento dos desafios expostos.

O método utilizado no desenvolvimento desse trabalho, em suma, é o ciclo de Engenharia de Software de forma simplificada. Inicialmente, as fases abordam o planejamento – com seu devido ambiente de UML – em que há o levantamento de requisitos, seguindo para a análise/projeto. Logo após, as outras fases retratam a execução – com seu ambiente de programação para codificação –, as quais operam a implementação/codificação, finalizando em estágio de testes.

Para as seções subsequentes, a explicação do jogo em si se transcorrerá preliminarmente, dando seguimento à parte de desenvolvimento do jogo na versão orientada a objetos. Na sequência, haverá a exposição de tabelas de conceitos utilizados, reflexões e considerações do projeto, além da divisão de tarefas. Portanto, uma vez apresentado do que se trata o conteúdo presente neste documento, segue-se o secionamento dos tópicos desse projeto.

# **EXPLICAÇÃO DO JOGO EM SI**

O jogo “Mount” foi idealizado com a temática de luta de heróis contra inimigos com superação de obstáculos. O objetivo do jogador é conduzir os heróis em ambientes repletos de plataformas, obstáculos e inimigos em cada fase, até a luta e eliminação dos líderes dos inimigos no final da última fase. O jogo propicia a participação de até dois jogadores que assumem personagens nos papéis de heróis - um denominado de guerreiro e outro de samurai. Eles são capazes de se movimentar – correr horizontalmente e saltar verticalmente, com o intuito de avançar em seus caminhos para passar de fase e ir em busca dos líderes dos inimigos, os feiticeiros. Os heróis poderão saltar em cima dos inimigos, eliminando-os. Os heróis têm um nível de vida a ser preservada. Cada tiro de projétil do inimigo que os atinjam, ou o próprio contato com o inimigo, reduzirá aos poucos o seu nível de vida durante o jogo. Esse nível de vida deverá ser preservado ao máximo, caso contrário, o seu esgotamento total o levará a óbito e finalizará o jogo sem o atingimento do objetivo.

Desenho de personagem de desenho animado

Descrição gerada automaticamente com confiança média

Figura 1. Heróis: Guerreiro e Samurai, respectivamente.

Inicialmente, o jogo é aberto através de um menu que direciona opções tal como: seleção da primeira fase, seleção da segunda fase e ranking.

Interface gráfica do usuário

Descrição gerada automaticamente

Figura 2. Menu com as opções de início de jogo.

A fase 1 é constituída por 6 andares ou plataformas. Dentre os inimigos, estão inclusos monstros, nomeados de goblins, que atiram balas de armas em direção aos heróis. Adicionalmente, há os esqueletos, que, ao contrário dos goblins, podem se movimentar horizontalmente ao longo do cenário e, ao colidirem horizontalmente com os heróis, reduzem o nível de vida deles. Os esqueletos não atiram. Os obstáculos compreendem entre caixas, espinhos, e poças de água. As caixas se dispõem no cenário, e dificultam a livre circulação dos heróis, que deverão passar por cima através de um salto. O espinho é gerado no meio das plataformas, e reduz o nível de vida dos heróis durante o tempo em que eles ficarem em cima de tal obstáculo. As poças de água, assim como os espinhos, se dispõem no meio das plataformas, mas reduzem a velocidade dos heróis, e impedem o pulo deles. A fase é encerrada com sucesso quando todos os inimigos são eliminados.

Tela de jogo de vídeo game

Descrição gerada automaticamente com confiança média

Figura 3. Cenário fase 1, e Inimigos: esqueleto e goblin, respectivamente.

A seguir, a fase 2 é constituída por 8 andares ou plataformas. Dentre os inimigos, novamente estão os esqueletos, que realizam as mesmas ações usadas na primeira fase. Entretanto, na segunda fase haverá certo número de líderes dos inimigos, os feiticeiros. Tais inimigos, além de se movimentarem aleatoriamente pelo cenário, atiram projéteis, assim como os globins, disparados contra os heróis com uma frequência definida. Os obstáculos são os mesmos da primeira fase, mas todos gerados em menor número comparado à fase anterior. Esta fase também é concluída com sucesso quando os heróis eliminarem todos os globins e vencerem os feiticeiros existentes.

Tela de jogo de vídeo game

Descrição gerada automaticamente com confiança médiaDesenho de uma flor

Descrição gerada automaticamente com confiança média

Figura 4. Cenário fase 2, e o Lider dos Inimigos: Feiticeiro, respectivamente.

# **DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS**

A seguir, estruturam-se as ideias do jogo alinhadas às exigências designadas dentro do âmbito avaliativo do que a disciplina institui. Dessa maneira, o trabalho foi desenvolvido iniciando-se com a compreensão dos requisitos fornecidos para o desenvolvimento do jogo. Oportunamente, esse conjunto de requisitos está apresentado na Tabela 1 tal qual foram apresentados inicialmente, sendo que também consta a situação deles e onde foram contemplados. Para tanto, segue-se abaixo a tabela com as respectivas informações.

Tabela 1. Lista de Requisitos do Jogo e Situações.

|  |  |  |  |
| --- | --- | --- | --- |
| **N.** | **Requisitos Funcionais** | **Situação** | **Implementação** |
| 1 | Apresentar graficamente menu de opções aos usuários do Jogo, no qual pode se escolher fases, ver colocação (ranking) de jogadores e demais opções pertinentes. | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe Menu e seu respectivo objeto, com suporte da SFML. |
| 2 | Permitir um ou dois jogadores com representação gráfica aos usuários do Jogo, sendo que no último caso seria para que os dois joguem de maneira concomitante. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classe Jogador cujos objetos são agregados em jogo, podendo ser apenas um jogador, entretanto. |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas, via menu, nas quais jogadores tentam neutralizar inimigos por meio de algum artifício e vice-versa. | Requisito previsto inicialmente e realizado. | Requisitos foi realizado completamente porque a classe Fase deriva as classes Fase 1 e Fase 2, que constituem diferentes cenários. |
| 4 | Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es) e um dos inimigos dever ser um ‘Chefão’. | Requisito previsto inicialmente e realizado. | Requisito foi realizado completamente como se observa na hierarquia da classe Inimigo em que há três tipos de inimigos, sendo um deles que atira projéteis. |
| 5 | Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias e sendo pelo menos 3 instâncias por tipo. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via função rand() no instante de inicializar inimigos em cada Fase. |
| 6 | Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa danos em jogador se colidirem. | Requisito previsto inicialmente e realizado. | Requisito foi realizado completamente como se observa na hierarquia da classe Obstáculo há três tipos de obstáculos, sendo um deles o Espinho que causa danos ao colidir. |
| 7 | Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (i.e., objetos), sendo pelo menos 3 instâncias por tipo. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via função rand() no instante de inicializar obstáculos em cada Fase. |
| 8 | Ter em cada fase um cenário de jogo constituído por obstáculos, sendo que parte deles seriam plataformas ou similares, sobre as quais pode haver inimigos e podem subir jogadores. | Requisito previsto inicialmente e realizado. | Requisito cumprido ao inicializar plataformas em cada fase. |
| 9 | Gerenciar colisões entre jogador para com inimigos e seus projeteis, bem como entre jogador para com obstáculos. Ainda, todos eles devem sofrer o efeito da gravidade no âmbito deste jogo de plataforma vertical e 2D. | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe Colisao, que checa o contato entre as entidades do jogo, quando seu método checaColisao() é chamada pela classe Fase. |
| 10 | Permitir: (1) salvar nome do usuário, manter/salvar pontuação do jogador (incrementada via neutralização de inimigos) controlado pelo usuário e gerar lista de pontuação (ranking). E (2) Pausar e Salvar Jogada. | Requisito previsto inicialmente e parcialmente realizado. | Requisito parcialmente cumprido, sendo que é possível pausar o jogo. |
| **Total de requisitos funcionais apropriadamente realizados.**  *(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)* | | | **95%** (noventa e cinco por cento). |

Após a apresentação dos requisitos com os acompanhamentos pertinentes, foi elaborado um projeto do sistema. Em verdade, além dos requisitos já havia um modelo genérico de classes, que também serviu de base para o projeto em questão. O projeto se trata de um diagrama de classes detalhado em UML, o qual correlaciona as ideias convenientes à OO, estabelecendo classes com seus devidos relacionamentos, características e hierarquias. A implementação do jogo é descrita, por conseguinte, através dos componentes retratados nesse diagrama de classe.

Analisando o diagrama em UML, a classe Principal é tida como a principal (possuindo uma tela e um visualizador, administrados pela classe Grafico). Além disso, ressalta-se a classe abstrata Ente, que deriva três outras classes abstratas as quais atuam como pilares no sistema: Menu, Fase e Entidade. Dessa maneira, a classe principal contém objetos de classes derivadas delas: os tipos de menus (para fins de administração ou seleção de estados de jogo), os personagens jogáveis, bem como as duas fases. Seguem-se nos próximos itens mais detalhes relacionados a essas classes pilares, tal como suas relações de herança com suas devidas propriedades.

Enfatiza-se a relevância da classe Entidade, que será classe base de diversas outras classes. Assim, a derivada Personagem – caracterizada por se atualizar e possuir níveis específicos de complexidade – é outra abstrata a qual deriva as classes Jogador e Inimigo, sendo este também classe base para tipos de inimigos diferentes. Já a entidade Obstáculo particulariza-se por conter derivadas não atualizáveis que interagem com os jogadores ao colidir, entretanto. Ademais, outras entidades isoladas consistem na classe Projétil, o qual também se atualiza (não possuindo complexidades como a Personagem).

Com finalidades sistemáticas, os menus de jogo coordenam a ordem do que é inicializado de acordo com eventos e estados pré-definidos no Jogo. À vista disso, atribui-se uma associação bidirecional à relação entre Menu e Jogo.

Logo que é selecionado um novo jogo, inicializa-se individualmente a fase, conhecendo os jogadores criados pela classe principal. Cada entidade de fase é inicializada de acordo com o solicitado, sendo que todas são adicionadas em uma classe ListaEntidade – enquadrada consoante ao seu template Lista – a qual é responsável principalmente por atualizar, desenhar ou salvar cada entidade. Os personagens e obstáculos são inseridos em listas do tipo List do STL, a qual estão dentro da classe Colisao, conhecida pela própria fase. Os inimigos e obstáculos são instanciados aleatoriamente, enquanto o jogador é posicionado no início da fase, finalizando a etapa de inicialização.

A classe Colisao calcula o módulo das distâncias entre os centros das entidades e a intersecção de coordenadas dos personagens e entidades para estabelecer a colisão deles de diferentes maneiras. Quando um personagem colide com a entidade, é chamado um método virtual das entidades que define sua reação à colisão. Uma plataforma colidindo em um personagem irá impedir com que ele a atravesse, enquanto um inimigo colidindo em um jogador irá decrementar a vida deste. Assim, estabelece-se uma lógica a qual permite simular efeitos físicos entre entidades.

Os projéteis são disparados por dois personagens derivados da classe Inimigo, sendo eles o Goblin e o Feiticeiro. Caso a vida de um jogador chegue a zero, a janela do jogo fecha.

Por fim, cumpriu-se o desenvolvimento do jogo, que esteve constantemente sendo testado a cada implementação de funcionalidade ou conceito dado. É nesse âmbito que a quarta etapa do processo de Engenharia de Software esteve presente, uma vez que foram utilizadas ferramentas de debug a fim de conferir dados relativos aos elementos do sistema, além de inúmeras discussões com o monitor para direcionar a resolução dos problemas enfrentados. Isto posto, ressalta-se a importância da fase de testes para refinar imprecisões no código, bem como avaliar métodos e mecanismos.

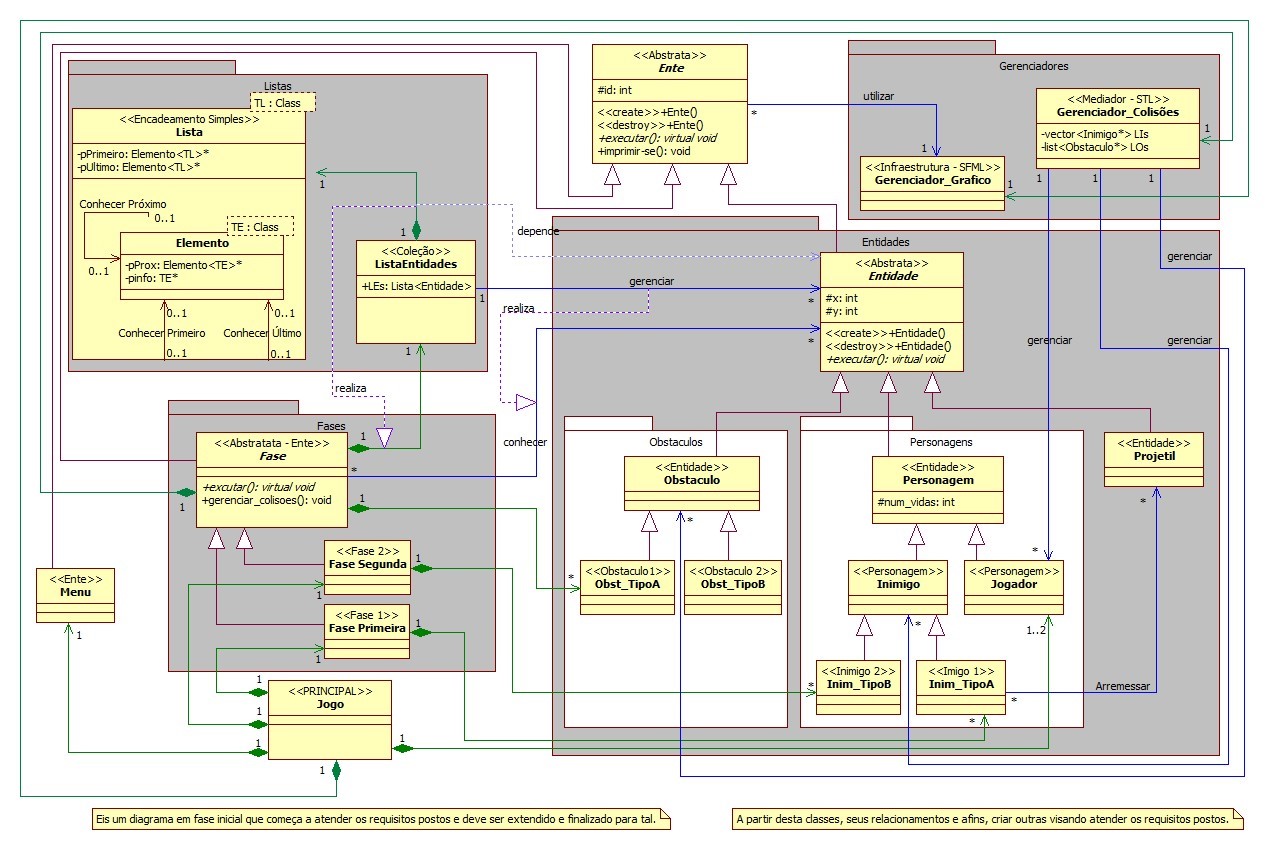


Figura 5. Diagrama de Classes de base em UML.

# **TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS**

A seguir, estruturam-se numa tabela os conceitos aprendidos classificados em utilizados e não utilizados de acordo com as exigências designadas e propostas pelo trabalho em questão.

Tabela 2. Lista de Conceitos Utilizados e Não Utilizados no Trabalho.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **N.** | **Conceitos** |  | **Uso** | **Onde / O quê** |
| **1** | **Elementares:** |  | |  |
| 1.1 | * Classes, objetos. & * Atributos (privados), variáveis constantes. & * Métodos (com e sem retorno). | e | Sim | Todos .h e .cpp, como nas classes ListaEntidade e Entidade. |
| 1.2 | * Métodos (com retorno const e parâmetro const). & * Construtores (sem/com parâmetros) e destrutores | | Sim | Na maioria dos .h e .cpp, como na classe Ente. |
| 1.3 | - Classe Principal. | | Sim | Main.cpp & Principal.h/.cpp |
| 1.4 | - Divisão em .h e .cpp. | | Sim | No desenvolvimento como um todo, como nas classes Fase e Esqueleto. |
| **2** | **Relações de:** | | | |
| 2.1 | - Associação direcional. & - Associação bidirecional. | | Sim | Em vários dos .h e .cpp, como na classe Goblin. |
| 2.2 | * Agregação via associação. & * Agregação propriamente dita. | | Sim | Em vários dos .h e .cpp, como na classe Fase. |
| 2.3 | * Herança elementar. & * Herança em diversos níveis. | | Sim | Em alguns dos .h e .cpp, como nas classes Entidade e Personagem |
| 2.4 | - Herança múltipla. | | Não |  |
| **3** | **Ponteiros, generalizações e exceções** | | | |
| 3.1 | - Operador this para fins de relacionamento bidirecional. | | Sim | Precisamente nos .h e .cpp, da classe Ente. |
| 3.2 | - Alocação de memória (new & delete). | | Sim | Inclusão e exclusão de entidades via classe ListaEntidades. |
| 3.3 | - Gabaritos/Templates criada/adaptados pelos autores (e.g., Listas Encadeadas via Templates). | | Sim | Template de Lista Encadeada. |
| 3.4 | - Uso de Tratamento de Exceções (try catch). | | Não |  |
| **4** | **Sobrecarga de:** | | | |
| 4.1 | - Construtoras e Métodos. | | Sim | Método push() na classe Lista. |
| 4.2 | - Operadores (2 tipos de operadores pelo menos – Quais?). | | Sim | Na classe Coordenada (operator + e operator -) |
| **---** | **Persistência de Objetos (via arquivo de texto ou binário)** | | | |
| 4.3 | - Persistência de Objetos. | | Não |  |
| 4.4 | - Persistência de Relacionamento de Objetos. | | Não |  |
| **5** | **Virtualidade:** | |  |  |
| 5.1 | - Métodos Virtuais Usuais. | | Sim | Método executa() |
| 5.2 | - Polimorfismo. | | Sim | Método prejudicaJogador() |
| 5.3 | - Métodos Virtuais Puros / Classes  Abstratas. | | Sim | Classe Fase |
| 5.4 | - Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto. | | Sim | Classe Grafico |
| **6** | **Organizadores e Estáticos** | | | |
| 6.1 | - Espaço de Nomes (Namespace) criada pelos autores. | | Sim | Classe Coordenada. |
| 6.2 | - Classes aninhadas (Nested) criada pelos autores. | | Sim | Classe Lista e Elemento. |
| 6.3 | - Atributos estáticos e métodos estáticos. | | Sim | Classe Grafico |
| 6.4 | - Uso extensivo de constante (const) parâmetro, retorno, método... | | Sim | No desenvolvimento como um todo |
| **7** | **Standard Template Library (STL) e String OO** | | | |
| 7.1 | * A classe Pré-definida String ou equivalente. & * Vector e/ou List da STL (p/ objetos ou ponteiros de objetos de classes definidos pelos autores) | | Sim | Classe Menu utilizando strings.  List na classe Colisao. |
| 7.2 | - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa. | | Sim | Map para associar strings a cores, texturas e fontes do SFML dentro da classe Grafico. |
| **---** | **Programação concorrente** | | | |
| 7.3 | - Threads (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time OU Win32API ou afins. | | Não |  |
| 7.4 | - Threads (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, OU Troca de mensagens. | | Não |  |
| **8** | **Biblioteca Gráfica / Visual** | | | |
| 8.1 | - Funcionalidades Elementares. & - Funcionalidades Avançadas como:   * tratamento de colisões * duplo buffer | | Sim | Classe Grafico trata janela do jogo. Classe Colisao trata as colisões. |
| 8.2 | * Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico.   OU   * RAD – Rapid Application Development (Objetos gráficos como formulários, botões etc). | | Sim | Classe Grafico trata os eventos. |
| **---** | **Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.** | | | |
| 8.3 | - Ensino Médio Efetivamente. | | Sim | Implementação de gravidade da Cinemática, para reproduzir movimentos análogos à realidade para o Personagens. |
| 8.4 | - Ensino Superior Efetivamente. | | Sim | Módulo e versor de vetores da Geometria Analítica para direcionar projéteis à posição do Jogador. |
| **9** | **Engenharia de Software** | | | |
| 9.1 | - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. & | | Sim | No desenvolvimento como um todo. |
| 9.2 | - Diagrama de Classes em UML. | | Sim | Plataforma StarUML |
| 9.3 | - Uso efetivo e intensivo de padrões de projeto GOF, i.e., mais de 5 padrões. | | Não |  |
| 9.4 | - Testes à luz da Tabela de Requisitos e do Diagrama de Classes. | | Sim | No desenvolvimento como um todo. |
| **10** | **Execução de Projeto** | | | |
| 10.1 | * Controle de versão de modelos e códigos automatizado (via github e/ou afins). & * Uso de alguma forma de cópia de segurança (i.e., backup). | | Sim | Git e GitHub do repositório do Aluno. |
| 10.2 | - Reuniões com o professor para acompanhamento do andamento do projeto. | | Sim | Reuniões dias 04/11, 11/11; 18/11 e 25/11.  Total de reuniões: 4. |
| 10.3 | - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. | | Não | Foi realizada apenas uma. Não foram realizadas as 10 reuniões propostas. |
| 10.4 | - Revisão do trabalho escrito de outra equipe e vice-versa. | | Sim | Ajuda do colega Thomas Missiato Pinton |
| **Total de conceitos apropriadamente utilizados.**  *(Cada grande tópico vale 10% do total de conceitos. Assim, por exemplo, caso se tenha feito metade de um tópico, então valeria 5%.)* | | | | 85% (Oitenta e cinco por cento). |

A tabela a seguir, lista as justificativas para os conceitos utilizados.

Tabela 3. Lista de Justificativas para Conceitos Utilizados.

|  |  |  |
| --- | --- | --- |
| **No.** | **Conceitos** | **Listar apenas os utilizados Situação** |
| **1** | **Elementares** |  |
| 1.1 | * Classes, objetos. & * Atributos (privados), variáveis e constantes. & * Métodos (com e sem retorno). | Classe, Objetos, Atributos e Métodos foram utilizados porque são conceitos elementares na orientação a objetos. |
| 1.2 | * Métodos (com retorno const e parâmetro const). & * Construtores (sem/com   parâmetros) e destrutores | A constância de atributos, métodos e afins, quando pertinente, evita mudanças equivocadas.  Construtores são finalmente mandatórios para bem inicializar os atributos, enquanto destrutores .... |
| 1.3 | - Classe Principal. | Ter uma classe Principal permite se alcançar um melhor ‘purismo’ em termos de OO. |
| 1.4 | - Divisão em .h e .cpp. | Permite melhor organização das classes e afins que compõem o sistema. |
| **2** | **Relações** |  |
| 2.1 | - Associação direcional. & - Associação bidirecional. | Herança e associações foram utilizadas para criar uma relação entre as classes. |
| 2.2 | * Agregação via associação. & * Agregação propriamente dita. | Herança e associações foram utilizadas para criar uma relação entre as classes. |
| 2.3 | * Herança elementar. & * Herança em diversos níveis. | Herança e associações foram utilizadas para criar uma relação entre as classes. |
| 2.4 | - Herança múltipla. | Herança e associações foram utilizadas para criar uma relação entre as classes. |
| **3** | **Ponteiros, generalizações e exceções** |  |
| 3.1 | - Operador this para fins de relacionamento bidirecional. | O operador this foi utilizado em métodos a fim de apontar para um objeto. |
| 3.2 | * Alocação de memória (new & delete). | A alocação de memória dinâmica foi utilizada para criar objetos em tempo de execução, como projéteis. |
| 3.3 | * Gabaritos/Templates criada/adaptados pelos autores (e.g. Listas Encadeadas via Templates). | O template foi utilizado como uma alternativa à sobrecarga de métodos. |
| **4** | **Sobrecarga de:** |  |
| 4.1 | - Construtoras e Métodos. | A sobrecarga de métodos foi utilizada para permitir diferentes tratamentos para diferentes parâmetros passados para uma função. |
| 4.2 | * Operadores (2 tipos de operadores pelo menos). | A sobrecarga de operadores foi utilizada para auxiliar na simplicidade e clareza do código. |
| **5** | **Virtualidade:** |  |
| 5.1 | - Métodos Virtuais. | A virtualidade foi utilizada para tratar todos os objetos derivados de uma mesma classe de forma geral, a partir de apontamentos para tal, sendo todos tratados genericamente em listas. |
| 5.2 | * Polimorfismo. | A virtualidade foi utilizada para tratar todos os objetos derivados de uma mesma classe de forma geral, a partir de apontamentos para tal, sendo todos tratados genericamente em listas. |
| 5.3 | * Métodos Virtuais Puros / Classes Abstratas. | A virtualidade foi utilizada para tratar todos os objetos derivados de uma mesma classe de forma geral, a partir de apontamentos para tal, sendo todos tratados genericamente em listas. |
| 5.4 | - Coesão e Desacoplamento. | A virtualidade foi utilizada para tratar todos os objetos derivados de uma mesma classe de forma geral, a partir de apontamentos para tal, sendo todos tratados genericamente em listas. |
| **6** | **Organizadores e Estáticos:** |  |
| 6.1 | - Espaço de Nomes (Namespace) criada pelos autores. | Namespaces foram utilizados para melhor organizar o código e torná-lo mais compreensível. |
| 6.2 | * Classes aninhadas (Nested) criada pelos autores. | Classes foram utilizados para melhor organizar o código e torná-lo mais compreensível. |
| 6.3 | * Atributos estáticos e métodos estáticos. | Atibutos foram utilizados para melhor organizar o código e torná-lo mais compreensível. |
| 6.4 | - Uso extensivo de constante (const) parâmetro, retorno, método... | Constantes foram utilizados para melhor organizar o código e torná-lo mais compreensível. |
| **7** | **Standard Template Library (STL) e String OO** |  |
| 7.1 | - A classe Pré-definida String ou equivalente. &  - Vector e/ou List da STL (p/ objetos ou ponteiros de objetos de classes definidos pelos autores) | Para melhor organizar o código e facilitar a alocação e  desalocação de memória. Strings foram efetivadas para manipulação de textos, e atuaram em conjunto com Mapas. |
| 7.2 | * Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa. | Para melhor organizar o código e facilitar a alocação e  desalocação de memória. |
| **8** | **Biblioteca Gráfica / Visual** |  |
| 8.1 | - Funcionalidades Elementares. &  - Funcionalidades Avançadas como:   * tratamento de colisões * duplo buffer | A biblioteca gráfica foi utilizada na implementação visual do jogo, como a impressão de entidades na tela, além do tratamento de eventos. |
| 8.2 | * Programação orientada e evento em algum ambiente gráfico.   OU   * RAD – Rapid Application Development (Objetos gráficos como formulários, botões etc). | A biblioteca gráfica foi utilizada na implementação visual do jogo, como a impressão de entidades na tela, além do tratamento de eventos. |
| 8.3 | * Ensino Médio. | A implementação e conhecimento da física do Ensino Médio ensina a compreensão e melhor entendimento dos movimentos análogos à realidade. |
| 8.4 | - Ensino Superior. | A implementação e conhecimento da física do Ensino Superior ensina a compreensão e melhor entendimento dos movimentos análogos à realidade. |
| **9** | **Engenharia de Software** | O ciclo da engenharia de software foi utilizado para melhor planejar o desenvolvimento, tornando-o mais organizado e poupando tempo. |
| **10** | **Execução do projeto** | O versionador de arquivos foi utilizado para registrar o progresso do desenvolvimento e tornar fácil a restauração do código em caso de erros graves. As reuniões com o professor foram úteis para mensurar os avanços do projeto, bem como o esclarecimento de dúvidas. |

# **REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS**

Um desenvolvimento procedimental, devido a sua simplicidade de execução, demandaria um grande grau de dificuldade quando se trata de implementações de código em escalas mais robustas e complexas, tal como foi no jogo. Por outro lado, um desenvolvimento voltado a OO é muito mais consistente com suas múltiplas funcionalidades, em virtude das características referentes às propriedades de organização, encapsulamento e reutilização. Ademais, o uso de classes auxilia na capacidade de abstração do código, já que a relação entre elas possui comportamentos análogos à realidade. Portanto, evidencia-se que um projeto realizado em prol de OO provou ser mais conveniente do que em uma versão procedimental.

# **DISCUSSÃO E CONCLUSÕES**

Dentre os resultados obtidos, é ressaltado que o desenvolvimento do trabalho trouxe muito aprendizado acadêmico e preparação profissional na condução e desenvolvimento de um projeto. O efeito alcançado constituiu o cumprimento das expectativas mediante a um período tão curto de atividades acadêmicas, as quais englobam uma grande quantidade de assuntos.

Evidencia-se que este trabalho auxiliou no desenvolvimento de novas aptidões e qualificações. Além do mais, ao longo da execução houve avanços na performance de técnicas tal como a disciplina propõe. Dessa forma, foram consolidadas experiências pertinentes em projetos as quais irão implicar na formação de competências não só na parte de OO, mas também generalizadamente na área de computação, visto que há grande demanda vinculada às qualificações de cooperação e disposição relacionadas. Por conseguinte, o caráter disciplinador relacionado a este trabalho será benéfico para o desenvolvedor no escopo profissional.

# **CONSIDERAÇÕES PESSOAIS**

A grande dificuldade aqui encontrada foi a pouca ou quase nenhuma ajuda e colaboração do parceiro designado, que culminou na sua desistência da disciplina. Todo o trabalho final ficou a cargo do único aluno.

# **DIVISÃO DO TRABALHO**

A seguir, a lista de atividades em que toda a responsabilidade foi de apenas um aluno.

Tabela 4. Lista de Atividades e Responsáveis.

|  |  |
| --- | --- |
| **Atividades** | **Responsável** |
| Compreensão de Requisitos | Vitor Kenji Zoppo Yamada |
| Diagramas de Classes | Vitor Kenji Zoppo Yamada |
| Programação em C++ | Vitor Kenji Zoppo Yamada |
| Implementação de Template | Vitor Kenji Zoppo Yamada |
| Implementação da Persistência dos Objetos | Vitor Kenji Zoppo Yamada |
| Gerenciador gráfico | Vitor Kenji Zoppo Yamada |
| Menus | Vitor Kenji Zoppo Yamada |
| Colocação (ranking) | Vitor Kenji Zoppo Yamada |
| Inimigos, obstáculos e jogadores | Vitor Kenji Zoppo Yamada |
| Escrita do Trabalho | Vitor Kenji Zoppo Yamada |
| Revisão do Trabalho | Vitor Kenji Zoppo Yamada |

* Vitor Kenji Zoppo Yamada trabalhou sozinho em 100% das atividades ou as realizando ou colaborando nelas efetivamente.

# **AGRADECIMENTOS**

Agradecimentos ao Prof. Dr. Jean M. Simão pelas reuniões e pelos conteúdos disponibilizados em sua disciplina. Agradecemos também o colega Thomas Missiato Pinton pela atenção e ajuda.

# **REFERÊNCIAS CITADAS NO TEXTO**

Não há referências citadas no texto.

# **REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO**

DEITEL, H. M.; DEITEL, P. J. C++ Como Programar. 5ª Edição. Bookman. 2006.