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

Gabriel Henrique Linke, Pedro Sodré dos Santos

gabriellinke@alunos.utfpr.edu.br, psantos.2000@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

### Universidade Tecnológica Federal do Paraná - UTFPR

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

**Resumo** – *Este documento apresenta um modelo para o texto do trabalho de Técnicas de Programação, além de instruções para o trabalho ele mesmo. Quanto ao resumo em si, ele deve trazer uma visão geral do trabalho. Mais precisamente, o resumo deve contemplar sucintamente a motivação e o contexto do trabalho, o seu objeto de estudo (um jogo de plataforma), o seu processo de desenvolvimento e os resultados obtidos. Um exemplo de resumo seria*: 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 Brasileirinho++, no qual o jogador enfrenta inimigos em um dado cenário. O jogo tem três fases que se diferenciam por dificuldades para o jogador. Para o desenvolvimento do jogo foram considerados os requisitos textualmente propostos e elaborado modelagem (análise e projeto) usando como recurso o Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language* - *UML*) usando como base um diagrama prévio proposto. Subsequentemente, em linguagem de programação C++, realizou-se o desenvolvimento que contemplou 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 pelos próprios desenvolvedores demonstraram sua funcionalidade conforme os requisitos e o modelagem elaborado. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado visado.

**Palavras-chave ou Expressões-chave** (máximo quatro, não excedendo três linhas)**:** 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 Prog. 2.

**Abstract** - *This document shows a model for the manuscript to the academic work of Técnicas de Programação as well as it presents general instructions about this academic work. With respect to the abstract contents, it must provide a general explanation about the work. Precisely, the abstract must shortly present the work motivation and context, its study object (a platform game), its development process, and the obtained results. An instance of abstract would be*:

**. . .**

**Key-words or Key-expressions** (maximum four, not exceeding three lines)**:** 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 é um trabalho de Técnicas de Programação que tem como objetivo ampliar a aplicação dos conceitos de programação orientada a objetos, com a devida coesão e desacoplamento, aprendidos em aula, assim como ampliá-los.

Este trabalho consiste na apresentação de um jogo de plataforma, incluindo inimigos e fases,implementado em C++, com a orientação do professor Jean Marcelo Simão.

Para a devida realização do trabalho, foi utilizado o uso do ciclo clássico da engenharia de software, ou seja, primeiramente foi feita a compreensão de requisitos passados (contidos na primeira tabela deste arquivo); a modelagem, feita através do diagrama de classes UML; a implementação feita em C++, com as IDEs do Visual Studio Community 2019 e CodeBlocks; assim como feito os testes para a verificação da ocorrência de erros na implementação.

Dado isso, será apresentado os devidos procedimentos, detalhadamente, de cada passo para a realização deste projeto, assim como a apresentação do jogo em si e dos devidos requisitos implementados.

# EXPLICAÇÃO DO JOGO EM SI

Lemurya é um jogo de plataforma que se passa na cidade submersa de Lemurya, contendo 3 fases, podendo ser jogados no modo solo (1 jogador) ou cooperativo (2 jogadores). Ao iniciar o jogo, o usuário se depara com o menu principal, onde poderá escolher entre começar um novo jogo ou carregar um jogo já salvo. Nesse menu, também há opção de ranking, onde o usuário poderá ver os 5 jogadores mais pontuados e suas respectivas pontuações.

**** 

Figura . Imagem do Menu Principal. Figura 2. Imagem do Ranking.

Após ser selecionada a opção de “New Game”, o usuário escolherá o número de jogadores (Solo – 1 jogador ou Coop – 2 jogadores), assim como a fase em que será jogada.

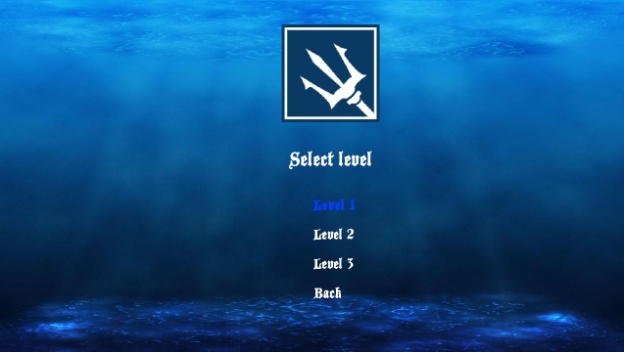
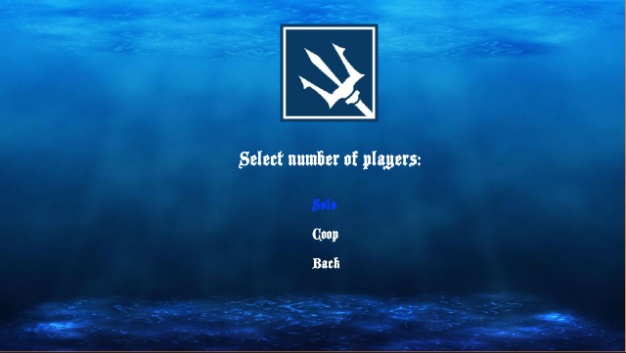


Figura 3. Seleção do número de jogadores. Figura 4. Seleção da fase.

Ao abrir a Fase 1 e 2, o jogador deverá chegar ao final dela, passando para a fase conseguinte. Na última fase (Fase 3), seu objetivo torna-se derrotar o chefão (Mago). Nessas fases, para atingir seu objetivo, o jogador controlará o rei reptiliano e terá de passar por diversos desafios, como superar inimigos e obstáculos.



Figura 5. Exemplo da 1° Fase Figura 6. Exemplo de 3° Fase, com chefão no topo.

Ao decorrer da fase, o jogador pode utilizar as teclas ‘WASD’ para se movimentar e a tecla ‘E’ para realizar seu ataque e derrotar inimigos (setas direcionais e ‘Enter’, respectivamente, no caso do 2° jogador), ganhando pontos que aparecem no canto superior direito da tela (Score), a pontuação quando há dois jogadores é conjunta. Ao fim da partida, estes pontos poderão aparecer no Ranking (Figura 2), após o jogador colocar seu nome na tela final de jogo, podendo ser a tela de ‘Game Over’, caso o jogador tenha morrido, ou de vitória (mesma tela com título diferente, neste caso apareceria ‘You Win’), desde que o jogador tenha vencido o jogo (derrotando o chefe na 3° fase).



Figura 7. Tela de fim de jogo

No decorrer da partida, o jogador deverá se preocupar com sua barra de vida, a qual aparece no canto superior esquerdo da tela (figuras 5 e 6). A vida diminui gradativamente quando o jogador encosta em um dos inimigos. Se a vida do jogador chegar a 0, ou caso caia em um dos abismos, o jogador morrerá acarretando na tela de ‘Game Over’ (figura 7). No caso de dois jogadores, aparecerá duas barras de vida, e esta tela aparece quando qualquer um dos dois jogadores morrer.



Figura 8. Exemplo de jogo com 2 jogadores.

No meio do jogo, também há opção de pause, que abre quando o jogador aperta a tecla ‘Esc’, ou quando a janela do jogo perde o foco. Nela, há as opções de retornar ao jogo, salvá-lo (onde poderá ser dado, posteriormente, o carregamento da fase na opção de ‘Load Game’), retornar ao menu principal e a opção de fechar o jogo.



Figura 9. Menu Pause.

# DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Tabela 1. Lista de Requisitos do Jogo e suas Situações.

|  |  |  |  |
| --- | --- | --- | --- |
| N. | Requisitos Funcionais | Situação | Implementação |
| 1 | Apresentar menu de opções aos usuários do Jogo. | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe Menu (abstrata) e os respectivos objetos de suas subclasses (Menu Principal, Menu Pause e Menu Morte). |
| 2 | Permitir um ou dois jogadores 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 Player cujos objetos são agregados em jogo, podendo ser um ou dois efetivamente. |
| 3 | Disponibilizar ao menos duas fases que podem ser jogadas sequencialmente ou selecionadas. | Requisito previsto inicialmente e realizado. | Requisito cumprido, inclusive com 3 fases, sendo através de objetos das classes (FaseAquatica1, FaseAquatica2 e FaseNoturna3). |
| 4 | Ter três tipos distintos de inimigos (o que pode incluir ‘Chefão’, vide abaixo), sendo que pelo menos um dos inimigos deve ser capaz de lançar projetil contra o(s) jogador(es). | Requisito previsto inicialmente e realizado. | Requisito cumprido, criando objetos das classes, Tritão, Esqueleto e Mago, sendo este último, o chefão capaz de lançar bolas de fogo (projétil). |
| 5 | Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias e sendo pelo menos 5 instâncias por tipo. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via método gerarInimigos, definido na classe abstrata Fase e redefinido nas suas classes derivadas. |
| 6 | Ter inimigo “Chefão” na última fase | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via objeto da classe Mago que é instanciado na última fase (FaseNoturna3). |
| 7 | Ter três tipos de obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classes Tronco, Pedra e Caixa cujos objetos são agregados na fase através da classe LemuryaPrototype. |
| 8 | Ter em cada fase ao menos dois tipos de obstáculos com número aleatório de instâncias (i.e., objetos) sendo pelo menos 5 instâncias por tipo. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via método gerarObstaculos definido na classe Fase e redefinido nas suas classes derivadas. |
| 9 | Ter representação gráfica de cada instância. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive com a utilização da biblioteca gráfica SFML e via classe GerenciadorGrafico. |
| 10 | Ter em cada fase um cenário de jogo com os obstáculos. | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe abstrata Fase e suas derivadas, com seus obstáculos sendo agregados através dos métodos gerarObstaculos, novoJogo e recuperarJogo. |
| 11 | Gerenciar colisões entre jogador e inimigos, bem como seus projeteis (em havendo). | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classes Collider, que verifica a colisão, e o Gerenciador de Colisões, que verifica as colisões percorrendo todas as listas de entidades que são colidíveis. |
| 12 | Gerenciar colisões entre jogador e obstáculos. | Requisito previsto inicialmente e realizado. | Idem para o requisito acima. |
| 13 | Permitir cadastrar/salvar dados do usuário, manter pontuação durante jogo, salvar pontuação e gerar lista de pontuação (*ranking*). | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classes GerenciadorDePontuação, atributo ranking do jogador, método do menu principal que gera a lista de pontuação. |
| 14 | Permitir Pausar o Jogo | Requisito previsto inicialmente e realizado | Requisito cumprido inclusive via classe MenuPause, que é instanciada caso o usuário pressione a tecla ESC. |
| 15 | Permitir Salvar Jogada. | Requisito previsto e realizado | Requisito realizado via classe MenuPause, e método checkSalvarJogo das classes derivadas de Fase, que permite salvar a jogada. |

<<Colocar Imagem do UML Principal>>

A principal classe do UML é a classe que dá o nome ao jogo (Lemurya), por meio de seu objeto o jogo é controlado através de uma pilha de states (State é outra classe do jogo), esta que é um stack da STL e um dos atributos da classe Principal. Nesse sentido, a classe State conhece o jogo, por meio disso são chamados os métodos ‘Push’ e ‘Pop’ da classe principal, fazendo a transição entre menus e fases do jogo.

<<Zoom no UML da relação entre Lemurya e State>>

No diagrama de classes, há o pacote das classes de gerenciadores, nela, há a classe ‘Animation’, que gerencia a animação de objetos; a classe ‘Collider’, que verifica a colisão entre objetos; a classe ‘GerenciadorDeColisoes’, a qual percorre as listas para verificar colisões; a classe ‘GerenciadorDePontuação’, que organiza a pontuação dos jogadores e a classe ‘GerenciadorGrafico’, que organiza as principais texturas do jogo. //Colocar esse trecho depois e colocar onde são instanciados.

<<Zoom no pacote de gerenciadores>>

Após isso há o pacote das classes mais gerais, que são as classes mais abstratas, nesse pacote contém as classes: Lemurya, State, Entidade e EntidadeColidivel. A partir delas que são feitas as subclasses que realmente serão encarregadas de fazer algo funcional no jogo.

<<Zoom no pacote de classes gerais>>

Após isso vem o pacote de Personagens, nela há a classe abstrata ‘Personagem’, juntamente com suas subclasses: ‘Player’, ‘Inimigo’, e derivadas desta última, vem as classes ‘Esqueleto’, ‘Tritao’ e ‘Mago’.



Figura 2. Diagrama de Classes de base em UML.

# TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

Nesta seção, em relação aos conceitos aprendidos, deve-se apresentar uma tabela de conceitos utilizados e não utilizados tal qual a Tabela 2, sendo que a coluna de ‘Conceitos’ dessa tabela **NÃO** pode ser alterada, absolutamente. Deve-se também apresentar outra tabela justificando o uso ou não uso, tal qual a Tabela 3.

Oportunamente, todas as tabelas que venham a ser utilizadas deverão ser numeradas sequencialmente com algarismos arábicos, conforme o exemplo abaixo:

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

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **N.** | | Conceitos | | Uso | | Onde / O quê | |
| **1** | | **Elementares:** | | | | | |
| - Classes, objetos. &  - Atributos (privados), variáveis e constantes. &  - Métodos (com e sem retorno). | | Sim | | Todos .h e .cpp | |
| - Métodos (com retorno *const* e parâmetro *const*). &  - Construtores (sem/com parâmetros) e destrutores | | Sim | | Todos .h e .cpp | |
| - Classe Principal. | | Sim | | main.cpp & Lemurya.h/.cpp | |
| - Divisão em .h e .cpp. | | Sim | | No desenvolvimento como um todo. | |
| **2** | | **Relações de:** | | | | | |
| - Associação direcional. &  - Associação bidirecional. | | Sim | | Bidirecional entre EntidadeColidivel e Collider.  Direcional entre BolaDeFogo e Player. | |
| - Agregação via associação. &  - Agregação propriamente dita. | | SIm | | Agregação via associação entre Lista de Entidades e Lista.  Agregação propriamente dita entre GerenciadorDeColisoes e ListaPlataformas, por exemplo. | |
| - Herança elementar. &  - Herança em diversos níveis. | | Sim | | Elementar em EntidadeColidivel e Entidade.  Em diversos níveis entre Mago e Entidade, por exemplo. | |
| - Herança múltipla. | | Sim | | Fase herdando de Entidade e State, por exemplo. | |
| **3** | | **Ponteiros, generalizações e exceções** | | | | | |
| - Operador *this*. | | Sim | | Em diversos momentos. Ao inicializar o prototype dentro de fase, por exemplo. | |
| - Alocação de memória (*new* & *delete*). | | Sim | | Em diversos momentos. Delete em destrutoras e new nos métodos de carregar um novo estado do jogo, como no carregarPause de fase. | |
| - Gabaritos/*Templates* criada/adaptados pelos autores (e.g. Listas Encadeadas via *Templates*). | | Sim | | Classe List. | |
| - Uso de Tratamento de Exceções (*try catch*). | | Sim | | Classe GerenciadorGrafico. | |
| **4** | | **Sobrecarga de:** | | | | | |
| - Construtoras e Métodos. | | Sim | | Construtoras nos objetos que podem ser clonados.  Métodos em FaseAquatica1, por exemplo. | |
| - Operadores (2 tipos de operadores pelo menos). | | Não | | - | |
| **Persistência de Objetos (via arquivo de texto ou binário)** | | | | | |
| - Persistência de Objetos. | | Sim | | Via métodos de gravarJogo e recuperarJogo das fases. | |
| - Persistência de Relacionamento de Objetos. | | Sim | | Persistência do relacionamento dos inimigos com os players, por exemplo | |
| **5** | | **Virtualidade:** | |  | |  | |
| - Métodos Virtuais. | | Sim | | Na classe Fase, por exemplo | |
| - Polimorfismo | | Sim | | Chamadas de update e Draw da classe Entidades, por exemplo. | |
| - Métodos Virtuais Puros / Classes Abstratas | | Sim | | Classes menu e fase, por exemplo. | |
| - Coesão e Desacoplamento | | Sim | | No projeto como um todo | |
| **6** | | **Organizadores e Estáticos** | | | | | |
| - Espaço de Nomes (*Namespace*) criada pelos autores. | | Não | | - | |
| - Classes aninhadas (*Nested*) criada pelos autores. | | Sim | | Classes List e Entidade | |
| - Atributos estáticos e métodos estáticos. | | Parcialmente | | Atributo estático VIEW\_HEIGHT em Lemurya | |
| - Uso extensivo de constante (*const*) parâmetro, retorno, método... | | Sim | | No projeto como um todo | |
| **7** | | Standard Template Library *(STL)* e String OO | | | | | |
| - 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 | | Em GerenciadorDePontuacao e ListaObstaculos, por exemplo | |
| - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa. | | Sim | | Em Lemurya(Pilha) e setRanking em MenuPrincipal(Multimap) | |
| **Programação concorrente** | | | | | |
| *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins*.* | | Não | | - | |
| *- Threads* (Linhas de Execução) no âmbito da Orientação a Objetoscom uso de Mutex, Semáforos, **OU** Troca de mensagens. | | Não | | - | |
| **8** | | **Biblioteca Gráfica / Visual** | | | | | |
| - Funcionalidades Elementares. &  - Funcionalidades Avançadas como:   * tratamento de colisões * duplo *buffer* | | Sim | | Representações gráficas e escrever texto através da biblioteca SFML. | |
| - Programação orientada e evento em algum ambiente gráfico.  **OU**  *- RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc). | | Sim | | Utilização dos Eventos da biblioteca SFML na implementação dos menus. | |
| **Interdisciplinaridades por meio da utilização de Conceitos de Matemática e/ou Física.** | | | | | |
| - Ensino Médio. | | Sim | | Conceitos de matemática e física básica, como velocidade, por exemplo. | |
| - Ensino Superior*.* | | Sim | | Norma de vetores, por exemplo. | |
| **9** | | **Engenharia de Software** | | | | | |
| - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. & | | Sim | | Durante reuniões com o professor e monitor. | |
| - Diagrama de Classes em *UML*. | | Sim | | No desenvolvimento como um todo. | |
| - Uso efetivo (quiçá) intensivo de padrões de projeto (particularmente GOF). | | Sim | | Classes  LemuryaPrototype(Prototype)State(State)  MenuPrincipal(Iterator) | |
| - Testes a luz da Tabela de Requisitos e do Diagrama de Classes. | | Sim | | Durante reuniões com o professor e monitor. | |
| **10** | | **Execução de Projeto** | | | | | |
| - Controle de versão de modelos e códigos automatizado (via SVN e/ou afins) **OU** manual (via cópias manuais). &  - Uso de alguma forma de cópia de segurança (backup). | | Sim | | Via GitHub e cópias manuais | |
|  | | - Reuniões com o professor para acompanhamento do andamento do projeto. | | Sim | | 3 Reuniões (verificar dias nos UMLs) | |
|  | | - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. | | Sim | | 8 Reuniões | |
|  | | - Revisão do trabalho escrito de outra equipe e vice-versa. | | (?) | | (?) | |

Ressalta-se que as legendas das figuras devem ser colocadas abaixo, enquanto que as legendas das tabelas devem ser colocadas acima das mesmas.

Tabela 3. Lista de Justificativas para Conceitos Utilizados e **Não** Utilizados no Trabalho.

|  |  |  |
| --- | --- | --- |
| No. | Conceitos | *Listar apenas os utilizados* Situação |
| 1 | **Elementares** | Os conceitos elementares como os de classe, objetos, atributos, métodos, entre outros, foram utilizados por conta de serem conceitos fundamentais da Programação Orientada a Objetos. |
| 2 | **Relações** | As relações de associação, agregação e herança foram utilizadas por serem partes fundamentais em um projeto desenvolvido com Programação Orientada a Objetos. |
| 3 | **Ponteiros, generalizações e exceções** | O operador this foi utilizado em alguns momentos onde era necessário retornar o ponteiro do objeto atual. A alocação de memória foi utilizada constantemente nos momentos em que se fez necessária a criação de objetos em tempo de execução. Templates foram utilizados para aumentar a reusabilidade do código. O tratamento de exceções (try catch) foi utilizado para impedir que o programa fosse inicializado caso houvesse a falta de algum arquivo essencial para o seu funcionamento. |
| 4 | **Sobrecargas e persistência de objetos** | A sobrecarga de construtoras e métodos foi utilizada para possibilitar a redefinição dos mesmos, para que assim, seja possível utilizá-los em diferentes situações. A sobrecarga de operadores não foi utilizada porque não foi possível encontrar uma situação em que ela fosse essencial. A persistência de objetos e relações foi utilizada para possibilitar salvar a atual versão do programa e posteriormente voltar-se para a última versão salva. |
| 5 | **Virtualidade** | Métodos virtuais, classes abstratas e polimorfismo foram utilizados, pois, são recursos que podem facilitar muito a Programação Orientada a Objetos. Coesão e desacoplamento foram utilizados pois são de suma importância na criação de um projeto. |
| 6 | **Organizador e Estáticos** | Foi realizado o uso extensivo de const para respeitar-se o princípio do menor privilégio¹. Classes aninhadas foram utilizados pois elas fazem com que uma classe não possa ser instanciada sem estar vinculada a uma classe de ordem superior. Atributos estáticos foram utilizados para valores que deveriam ser divididos por todas as instâncias de uma classe. Métodos estáticos não foram utilizados pois não foi encontrada uma forma de inseri-los no projeto. Namespace não foi utilizado pois a sua utilização só foi cogitada ao fim do projeto, quando o tempo para a finalização já era escasso. |
| 7 | **Componentes STL e programação concorrente** | Os componentes da STL foram utilizados para acelerar o desenvolvimento do projeto, pois são componentes prontos que já foram testados e otimizados, tendo baixíssimo risco de conterem algum erro de implementação. Programação concorrente não foi utilizada por conta de ser cogitada apenas próximo ao fim do projeto, momento em que o tempo para a implementação era escasso. |
| 8 | **Uso de biblioteca gráfica e interdisciplinaridade** | A matemática e a física foram essenciais para o desenvolvimento, pois estavam presentes no projeto como um todo. O uso da biblioteca gráfica SFML foi de suma importância para facilitar a implementação da parte gráfica do programa. |
| 9 | **Engenharia de Software** | O diagrama de classes foi utilizado majoritariamente para a projeção do software. A partir do diagrama foi possível analisar como estava a implementação do projeto, e verificar se ela estava cumprindo os requisitos. Padrões de projeto foram utilizados para melhorar a coesão e desacoplamento, assim, permitindo que o código se tornasse mais flexível e reutilizável. |
| 10 | **Execução de projeto** | O controle de versões foi utilizado para manter um histórico de todas as versões feitas no projeto, e para que houvesse uma cópia de segurança caso necessário. As reuniões com o professor e com o monitor da disciplina foram realizadas para que ambos pudessem opinar a respeito do projeto, dando dicas para a sua melhoria, e para que pudessem acompanhar o seu desenvolvimento. A revisão foi realizada para garantir a qualidade do trabalho. |

[1] - Princípio citado no Deitel

# REFLEXÃO COMPARATIVA ENTRE DESENVOLVIMENTOS

Nesta seção se deve apresentar uma reflexão comparativa (sucinta e suficiente) entre o desenvolvimento orientado a objetos e o desenvolvimento procedimental segundo a percepção dos desenvolvedores nos trabalhos realizados no âmbito da disciplina de Fundamentos de Programação 1 e Técnicas de Programação (ou até no âmbito de outras disciplinas ou experiências prévias se for pertinente).

# DISCUSSÃO E CONCLUSÕES

Esta seção deverá apresentar uma reflexão sobre o desenvolvimento e os resultados obtidos. Certamente uma conclusão bem elaborada ajuda na avaliação do Professor. Por sua vez, a avaliação do trabalho como um todo (pelo Professor) será baseada em:

* Quantidade e qualidade de conceitos utilizados na elaboração do *software*, envolvendo particularmente os bons princípios de Orientação Objetos, como organização, encapsulamento e reutilização, todos baseados no princípio de coesão e desacoplamento doutrinados em classe.
* Complexidade do problema e respectiva quantidade e qualidade de projeto e código o que, novamente, envolve a correção na aplicação dos princípios da Orientação a Objetos (i.e. coesão, desacoplamento, encapsulamento, organização e reutilização), o número de classes e objetos, o número e a forma de relacionamento, a complexidade algorítmica, o cumprimento de requisitos etc.
* Qualidade do trabalho escrito e da apresentação.

Em termos gerais, pode-se considerar o primeiro item com um peso de 35%, o segundo com um peso de 35% e o terceiro com um peso de 30%, lembrando que eles estão inter-relacionados (sobretudo o primeiro e o segundo). Não obstante, esta porcentagem é relativa, pois (por exemplo) um item muito bem desenvolvido pode até compensar (um pouco) outro não tão bem desenvolvido em proporções diferentes deste referencial dado. Lembrar ainda que fazer seu projeto ser acompanhado pelo professor e monitor (a medida que avança) é **fundamental**.

# CONSIDERAÇÕES PESSOAIS

Caso o aluno ou os alunos desejam expressar algum sentimento relativo, por exemplo, aos aprendizados e dificuldades encontradas, isto deve ser feita nesta seção opcional. Nesta seção pode-se até utilizar primeira pessoa, entretanto seria melhor a forma impessoal.

Neste sentido, **todas as demais seções devem ser escritas de forma impessoal** (o que significa não usar primeira (e mesmo segunda) pessoa singular ou plural – em suma não usar “eu” ou “nós” no texto). Ademais, salienta-se que o trabalho deve ser redigido em linguagem correta, na forma culta e sem exageros poéticos, com textos não prolixos e bem encadeados.

# DIVISÃO DO TRABALHO

Esta seção deverá ter uma tabela salientando quem desenvolveu cada classe/módulo do *software* e realizou demais atividades como as de ‘engenharia de *software’*, a redação do trabalho escrito, a revisão da redação do trabalho e a preparação da apresentação do trabalho.

Tabela 4. Lista de Atividades e Responsáveis.

|  |  |
| --- | --- |
| Atividades. | Responsáveis |
| Levantamento de Requisitos | Pedro e Gabriel |
| Diagramas de Classes | Pedro |
| Programação em C++ | Gabriel e Pedro em geral |
| Implementação de *Template* | Gabriel |
| Implementação da Persistência dos Objetos... | Gabriel |
| Implementação de State | Pedro |
| Implementação de Menus | Mais Pedro que Gabriel |
| Implementação de Prototype | Pedro e Gabriel |
| Implementação de Ranking | Pedro e Gabriel |
| Implementação de Projétil | Gabriel |
| Implementação de Fases | Mais Gabriel que Pedro |
| Implementação de Colisões | Gabriel |
| Implementação de Listas | Mais Gabriel que Pedro |
| Implementação do 2° Jogador | Gabriel |
| Escrita do Trabalho | Mais Pedro que Gabriel |
| Revisão do Trabalho | Pedro e Gabriel |

# AGRADECIMENTOS

Havendo agradecimentos de ordem profissional, estes deverão vir antes das referências.

Aqui se pode salientar e agradecer caso outra equipe tenha revisado o trabalho.

# REFERÊNCIAS CITADAS NO TEXTO

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

[2] STADZISZ, P. C. Projeto de Software usando UML. Apostila CEFET-PR 2002.

<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/EngSoftware/Apostila%20UML%20-%20Stadzisz%202002.pdf>

[3] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 09/09/2015, às 15:15:

<http://www.dainf.ct.utfpr.edu.br/~jeansimao/Fundamentos2/Fundamentos2.htm>.

# REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] BEZERRA, E. Princípios de Análise e Projeto de Sistemas com UML. Editora Campus. 2003. ISBN 85-352-1032-6.

[B] HORSTMANN, C. Conceitos de Computação com o Essencial de C++, 3ª edição, Bookman, 2003, ISBN 0-471-16437-2.