

# **TRABALHO PARA A DISCIPLINA DE TÉCNICAS DE PROGRAMAÇÃO DO CURSO DE ENGENHARIA DE COMPUTAÇÃO DA UTFPR: *Gangster – MODELO & ESPECIFICAÇÃO DO TRABALHO***

Gustavo Padovam Ferreira, Gustavo Henrique Bruno dos Santos  
gustavopadovam@alunos.utfpr.edu.br , gustavohenriquesantos@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** – *Este documento descreve o desenvolvimento completo de um jogo de plataforma, atendendo às exigências da disciplina de Técnicas de Programação, com o propósito de aplicar, na prática, as técnicas essenciais de engenharia de software e os principais conceitos de programação orientada a objetos em C++. O projeto, denominado Gangster, foi concebido para oferecer ao jogador uma experiência desafiadora e envolvente, com fases que apresentam níveis de dificuldade variados e inimigos estrategicamente posicionados. Utilizando um diagrama de classes em UML como base, a estrutura do jogo foi cuidadosamente modelada para incorporar conceitos fundamentais como herança, encapsulamento e polimorfismo, além de explorar aspectos avançados, como a implementação de classes abstratas, a sobrecarga de operadores e o uso da biblioteca gráfica SFML. Após a conclusão da implementação, o jogo foi submetido a rigorosos testes para garantir que os requisitos funcionais e de desempenho fossem plenamente atendidos. Em suma, o projeto permitiu a consolidação do aprendizado em programação orientada a objetos e a aplicação de práticas de desenvolvimento de software de forma eficaz.*

**Palavras-chave:** Desenvolvimento de Jogo de Plataforma em C++; Técnicas de Programação Avançadas em C++; Estruturação e Normas de Trabalho Acadêmico Técnico; Implementação de Projetos Acadêmicos em C++.

## **INTRODUÇÃO**

Este trabalho foi desenvolvido no contexto da disciplina de Técnicas de Programação, lecionada pelo Prof. J. M. Simão do DAINF/UTFPR. O principal objetivo deste projeto é aplicar os conceitos de programação orientada a objetos, abordados durante o curso, por meio do desenvolvimento de um jogo de plataforma. Através deste processo, busca-se aprofundar o entendimento prático sobre princípios como coesão, desacoplamento e reutilização de código, essenciais para a construção de software de qualidade.

O foco do projeto é a criação de um jogo de plataforma, previamente acordado com o professor. A escolha do jogo foi baseada em sua capacidade de explorar múltiplos aspectos da programação orientada a objetos, incluindo herança, polimorfismo e encapsulamento. Este trabalho visa não apenas consolidar os conhecimentos adquiridos em sala, mas também desafiar o aluno a aplicar novas habilidades e conceitos de forma prática e eficiente.

O desenvolvimento do jogo seguiu um ciclo simplificado de Engenharia de Software, que inclui a compreensão dos requisitos, a modelagem do software através de diagramas de classes em UML, a implementação em C++ e a realização de testes. O diagrama de classes utilizado foi derivado de um modelo exemplar fornecido previamente pelo professor, servindo como base para a estrutura do jogo e garantindo a aderência aos princípios orientados a objetos ensinados na disciplina.

Nas seções subsequentes deste documento, serão detalhados os requisitos do jogo, o processo de modelagem, a implementação, os testes realizados e os resultados obtidos. Além

disso, serão apresentadas as principais dificuldades encontradas durante o desenvolvimento e as soluções adotadas para superá-las, bem como uma avaliação crítica sobre o aprendizado proporcionado por este projeto.

## EXPLICAÇÃO DO JOGO EM SI

O jogo narra a trajetória de Carlo, um gângster determinado a libertar as cidades italianas de Veneza e Florença do controle da máfia. Ele é frequentemente acompanhado por seu irmão gêmeo, Vito, que se destaca por sua gravata azul. A eficácia de Carlo na missão é avaliada pela pontuação, onde uma maior pontuação indica um sucesso mais significativo no combate aos mafiosos. No início do jogo, o jogador tem a opção de escolher entre Veneza ou Florença como o cenário de ação e decidir se deseja contar com a ajuda de Vito nessa empreitada, conforme ilustrado na Figura 1.

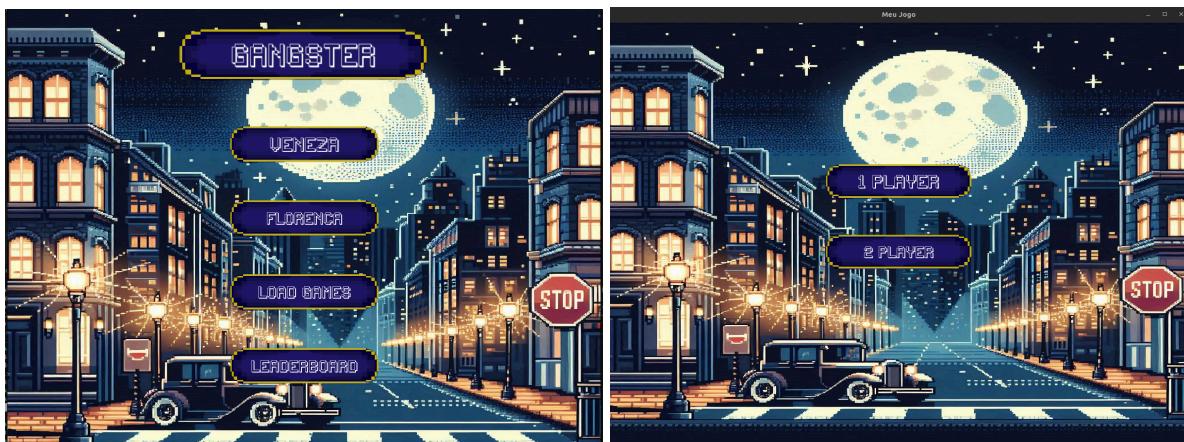


Figura 1. Tela de Início (esquerda) e Tela de Escolha dos Jogadores (direita)

Ambas as fases se inspiram em um combate realizado nos telhados de cidades italianas, como pode ser visto na Figura 2. Além disso, a fase mostrará a quantidade de vida dos jogadores, bem como a sua pontuação.



Figura 2. Cidade [Italiana](#) (esquerda) e visão da Fase (direita)

Durante sua jornada, o jogador encontrará três tipos principais de obstáculos, conforme ilustrado na Figura 3: espinhos, que causam dano ao contato; lixo, que obstrui o caminho e reduz a velocidade do jogador; e minas, que arremessam o jogador para trás ao serem acionadas. Além desses obstáculos, o jogador enfrentará três tipos distintos de

mafiosos. O Lutador, que persegue o jogador quando o avista, atacando-o com seus punhos; o Atirador, que dispara sempre que o jogador está em sua mira; e o Soldado Chefe, um mafioso profissional que, além de disparos poderosos, utiliza granadas que podem lançar o jogador para fora do telhado. Todos esses inimigos estão representados na Figura 4.

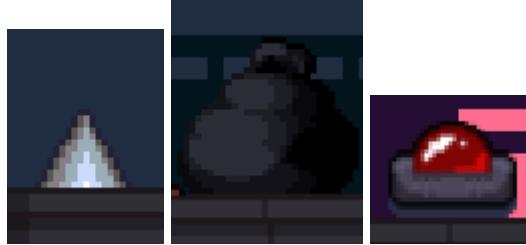


Figura 3. Espinho, Lixo e Mina, respectivamente.



Figura 4. Lutador, Atirador e Soldado Chefe, respectivamente.

Veneza e Florença são as duas cidades principais do jogo, cada uma oferecendo um desafio único e ajustado ao nível de dificuldade. Veneza, sendo a fase mais fácil, apresenta um ambiente inicial onde a máfia ainda está começando a se estabelecer. Nesse cenário, o jogador não encontrará minas nem Soldados Chefes, permitindo uma experiência mais tranquila para quem está começando.

Por outro lado, Florença é uma fase mais difícil, refletindo a influência mais forte da máfia. Nesta cidade, a máfia utiliza Soldados Chefes e protege seu território com minas, além de espinhos e lixos, aumentando significativamente o nível de dificuldade.

O jogo também oferece uma experiência completa de gerenciamento e progressão, com um menu de pause que permite ao jogador interromper a partida a qualquer momento. O sistema de salvamento e carregamento de jogos salvos permite que o jogador salve seu progresso e retome o jogo posteriormente. Além disso, a pontuação é salva ao final de cada partida ou após a morte, e é possível visualizar e comparar suas melhores pontuações em um leaderboard.

Os jogadores têm a opção de jogar as fases sequencialmente, passando da Fase 1 para a Fase 2 e vice-versa, ou podem selecionar as fases diretamente pelo menu principal, permitindo flexibilidade para aumentar suas pontuações e explorar diferentes desafios conforme desejarem.



Figura 5. Menu de Game Over (esquerda) e Leaderboard (direita)



Figura 6. Menu de Pause (esquerda) e Menu de Carregamento de Jogos (Direita)

## DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

Nesta seção, são apresentados os requisitos funcionais estabelecidos para o desenvolvimento do jogo, bem como as situações práticas que ilustram a implementação desses requisitos. A Tabela 1 resume os requisitos definidos, destacando se foram cumpridos integralmente ou se houve alguma limitação na implementação. Além disso, são fornecidos exemplos de situações que mostram como os requisitos foram aplicados no contexto do desenvolvimento do jogo. A tabela também reflete o percentual de requisitos funcionais realizados com sucesso, indicando o progresso e as áreas que ainda requerem aprimoramento.

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

| N. | Requisitos Funcionais   | Situação                                     | Implementação  |
|----|---|--|--|
| 1  | Apresentar graficamente menu de opções aos usuários do Jogo, no qual pode se escolher fases, ver colocação ( <i>ranking</i> ) de jogadores e demais opções pertinentes (previstas nos demais requisitos). | Requisito previsto inicialmente e realizado. | Requisito cumprido via classe Menu e suas classes derivadas.             |
| 2  | Permitir um ou dois jogadores com representação gráfica aos usuários do Jogo, sendo que no  | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via classe Jogador cuja construtora permite |

|   |   |  |   |
|---|---|--|---|
|   | último caso seria para que os dois joguem de maneira concomitante.  |  | diferenciar o jogador 1 do jogador 2, bem como a classe Fase funcionar com 1 e/ou 2 jogadores.  |
| 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. | O requisito foi realizado completamente porque existem a classe Veneza e a classe Florença, que podem ser jogadas sequencialmente, ou selecionadas via menu, no qual o jogador tenta neutralizar os inimigos e vice-versa.            |
| 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 projéteis contra o(s) jogador(es) e um dos inimigos deve ser um ‘Chefão’. | Requisito previsto inicialmente e realizado. | O requisito foi realizado completamente, como se vê no pacote Personagens, em que se encontra a classe Inimigo o qual possui 3 Inimigos Distintos, Lutador, Atirador e SoldadoChefe.  |
| 5 | Ter a cada fase ao menos dois tipos de inimigos com número aleatório de instâncias, podendo ser várias instâncias (definindo um máximo) e sendo pelo menos 3 instâncias por tipo.   | Requisito previsto inicialmente e realizado. | Requisito cumprido via pacote Fase, Veneza e Florença, a qual gera a criação de inimigos via Métodos - cria(NOME), tendo um mínimo e máximo anteriormente já definidos e gerando um número aleatório de inimigos entre esses valores. |
| 6 | Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa dano em jogador se colidirem.  | Requisito previsto inicialmente e realizado. | Pode ser encontrado via pacote Entidades a qual a classe Obstáculos, agrupa 3 tipos, Espinho, Lixo e Mina.  |
| 7 | Ter em cada fase ao menos dois tipos de obstáculos com número aleatório (definindo um máximo) de instâncias ( <i>i.e.</i> , objetos), sendo pelo menos 3 instâncias por tipo.   | Requisito previsto inicialmente e realizado. | Requisito cumprido via pacote Fase, a qual gera a criação de obstáculos via Métodos - cria(NOME), tendo um mínimo e máximo anteriormente já definidos e gerando um número aleatório de obstáculos entre esses valores.                |
| 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 inclusive via pacote Fase, a qual gera os obstáculos e os personagens (jogador e inimigos), os quais podem subir nas plataformas.  |

|   |   |  |   |
|---|---|--|---|
| 9   | Gerenciar colisões entre jogador para com inimigos e seus projéteis, bem como entre jogador para com obstáculos. Ainda, todos eles devem sofrer o efeito de alguma 'gravidade' no âmbito deste jogo de plataforma vertical e 2D.            | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via pacote gerenciadores, a qual possui a classe Gerenciador de Colisões, no qual como o próprio nome diz, gerencia a colisão entre os objetos, a aplicação da gravidade não está diretamente no gerenciador, mas sim em objetos considerados móveis |
| 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 ( <i>ranking</i> ). E (2) Pausar e <u>Salvar/Recuperar</u> Jogada. | Requisito previsto inicialmente e realizado. | Requisito cumprido inclusive via menus que permitem ver o leaderboard com os pontos obtidos, e o menu de pause (salvar) e o menu de carregamento.   |
| <b>Total de requisitos funcionais apropriadamente realizados.</b><br><i>(Cada tópico vale 10%, sendo que para ser contabilizado deve estar realizado efetivamente e não parcialmente)</i> |   |  | <b>100% (cem por cento).</b>  |

## Gerenciadores

Primeiramente, implementamos o Gerenciador\_Grafico como uma classe Singleton, acessada por meio de um ponteiro estático na classe base Ente, sendo responsável por todas as ações relacionadas à tela que o usuário vê ao jogar o jogo. Esta classe não apenas desenha sprites e textos, mas também lida com o redimensionamento da tela. Além disso, o Gerenciador\_Grafico agrega todas as fontes e imagens usadas em um mapa da STL. Quando uma classe solicita um elemento gráfico, o Gerenciador\_Grafico verifica se o elemento está presente no mapa; se não estiver, ele aloca um novo Elemento Gráfico e o adiciona ao mapa. O Gerenciador\_Colisoes, por sua vez, tem a função de verificar quais objetos estão colidindo em cada momento e notificar os objetos pertinentes, permitindo uma resposta adequada às colisões.

O Gerenciador\_Input utiliza o padrão de projeto Observer para notificar a classe Jogador e os Menus sempre que o usuário envia um evento para o jogo, como uma tecla pressionada ou solta, ou um fechamento de janela. Este gerenciador mantém uma lista de observadores contínuos, que precisam de notificações contínuas, como no caso de um usuário pressionando "A" para andar para a esquerda. Ele também gerencia observadores normais, que são interessados apenas em eventos específicos, como cliques em menus.

Além disso, o Gerenciador\_Estado, outra classe Singleton, auxilia no controle do estado atual do jogo, gerenciando a transição entre diferentes estados, como o menu inicial, as fases do jogo e o estado de game over.

Finalmente, a classe Configurações, também implementada como Singleton, armazena e gerencia as configurações do jogo, como a escolha entre um ou dois jogadores. Essas informações podem ser alteradas através do Menu, e a classe Configurações garante que essas alterações sejam aplicadas e mantidas durante o jogo.

## Entidades e Lista de Entidades

As Entidades representam todos os elementos do jogo que interagem entre si durante as fases. Dentro dessa categoria, temos os Obstáculos, que são representados por imagens estáticas na classe ObjetoEstático. Esses obstáculos incluem plataformas, que servem de superfície para os Personagens, e barreiras que dificultam o progresso do Jogador, como Espinhos, Lixo e Minas. Cada um desses obstáculos possui uma implementação específica, gerando efeitos distintos ao serem encontrados pelo Jogador. Além disso, há o Projétil, utilizado pelos Personagens para neutralizar uns aos outros.

Os Personagens compartilham várias características em comum, como a lógica de animação, a aplicação de gravidade e o gerenciamento da caixa de colisão, que garante que a área de colisão permaneça constante, independentemente da animação em curso. Eles também possuem um atributo “morto”, que impede qualquer interação de colisão com outros personagens após sua morte.

Como essas Entidades compartilham um conjunto de funções e atributos, elas são organizadas em uma Lista duplamente encadeada, implementada por meio de classes Aninhadas e Templates. A ListaDeEntidades é responsável por iterar sobre essas Entidades na fase, executando as ações de cada uma, garantindo a coesão e a fluidez da jogabilidade.

## Elementos Gráficos

Os Elementos Gráficos contém o ObjetoEstático que é uma abstração de uma imagem estática e os métodos auxiliares dessa classe. O atributo mais interessante são as classes Animação e TrilhaAnimação. Todo objeto animado agrega uma Animação e com ela, pode selecionar, facilmente, qual das animações deve ser apresentada. Isso é feito através de um mapa que associa strings que identificam a trilha de animação e a classe de TrilhaAnimação, que caso não seja encontrada, ela é alocada. Com isso, apenas as trilhas de animação que são utilizadas são carregadas, e uma vez carregadas, elas persistem para fácil e rápida troca entre trilhas.

## Menus

Os menus, todos similares entre si, permitem ao usuário realizar diversas ações pelo jogo. Para isso, a classe Base menu implementa toda a lógica de alternar entre botões, e elementos necessários para o menu, como os botões, textos e fundos. Com isso, foi fácil criar os diferentes menus de início de jogo, leaderboard, fim de jogo, pause, escolha de jogadores e fim de jogo.

## Fases

As Fases geram, primeiramente, as plataformas, variando, aleatoriamente, suas alturas, e deixando espaços entre elas. Em seguida, são gerados os inimigos e obstáculos, que segundo uma distribuição normal, cuja média é a quantidade de plataformas e o desvio padrão foi escolhido com base em experimentos, escolhe aleatoriamente as plataformas para posicioná-los. A fase acaba quando o jogador ou jogadores chegam ao fim da fase, demarcado por uma caixa de correio, ou quando ambos os jogadores morrem, ou o jogador morre, no modo single-player. Além disso, as Fases têm a capacidade de salvar/carregar em um arquivo json todas as entidades presentes na fase usando o método polimórfico gravar da classe

Entidade, bem como salvar/carregar a pontuação daquele momento. Elas se diferenciam tanto na quantidade de plataformas, quanto nos obstáculos e inimigos propostos em cada uma das fases denominadas como Florença e Veneza.

## Jogo

A função da classe Jogo tem como utilidade usar o GerenciadorEstado para saber qual o estado atual e executar o objeto pertinente, sendo responsável também por alocar e desalocar as diferentes partes do jogo, conforme são necessários.

## Diagrama em UML

Todas as relações mencionadas, métodos e atributos de cada classe foram agrupadas em um diagrama de classes através do *Unified Modeling Language* (UML), mostrado na figura 7, que ajuda na visualização das relações entre as classes, bem como de sua caracterização propriamente dita.

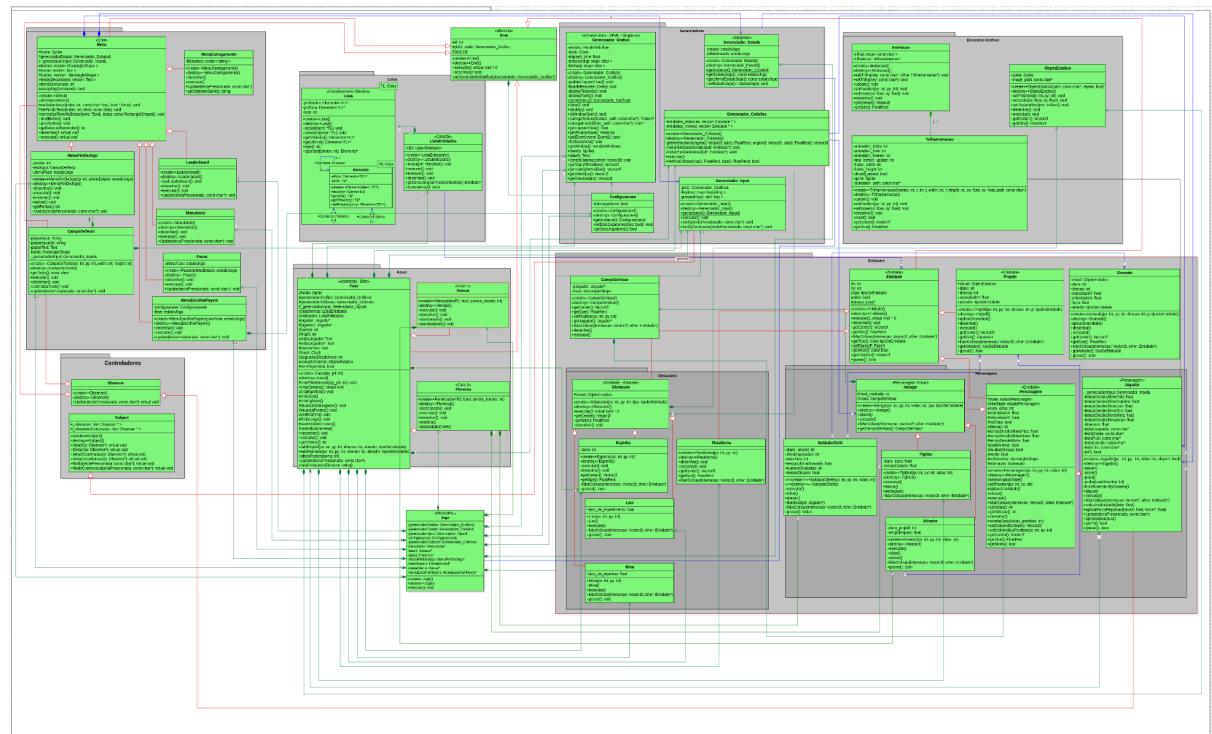


Figura 7. Diagrama de Classes em UML.

## TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

A seguinte tabela apresenta os conceitos aprendidos em Técnicas de Programação que foram utilizados ou não neste trabalho, bem como o modo no qual foram utilizados.

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

| N. | Conceitos    | Uso | Onde / O quê / Justificativa em uma linha |
|----|--------------|-----|---|
| 1  | Elementares: |     |   |

|          |  |     |   |
|----------|--|-----|---|
| 1.1      | - Classes, objetos. &<br>- Atributos (privados), variáveis e constantes. &<br>- Métodos (com e sem retorno).           | Sim | Ao longo do .h e .cpp, todos esses conceitos podem ser encontrados.   |
| 1.2      | - Métodos (com retorno <i>const</i> e parâmetro <i>const</i> ). &<br>- Construtores (sem/com parâmetros) e destrutores | Sim | Ao longo do .h e .cpp, todos esses conceitos podem ser encontrados.   |
| 1.3      | - Classe Principal.  | Sim | Main.cpp & Principal.h/.cpp   |
| 1.4      | - Divisão em .h e .cpp.  | Sim | No desenvolvimento como um todo, toda classe que não era template (como Lista e Elemento).  |
| <b>2</b> | <b>Relações de:</b>  |     |   |
| 2.1      | - Associação direcional. &<br>- Associação bidirecional.   | Sim | Exemplo: Todo Ente tem uma associação direcional ao gerenciador gráfico.<br>Todos que usam o Gerenciador_Input, usam associações bidirecionais. |
| 2.2      | - Agregação via associação. &<br>- Agregação propriamente dita.  | Sim | Toda Personagem agrupa um objeto de Animação.<br>O Jogo agrupa as diferentes fases e menus via associação.                                      |
| 2.3      | - Herança elementar. &<br>- Herança em diversos níveis.  | Sim | O Jogador herda de Personagem que herda de Entidade que herda de Ente.  |
| 2.4      | - Herança múltipla.  | Sim | O Jogador herda de Personagem e também de Observer  |
| <b>3</b> | <b>Ponteiros, generalizações e exceções</b>  |     |   |
| 3.1      | - Operador <i>this</i> para fins de relacionamento bidirecional.   | Sim | Todos que herdam de Observer, como o Jogador, usam o <i>this</i> para se inscreverem no Gerenciador Input                                       |
| 3.2      | - Alociação de memória ( <i>new</i> & <i>delete</i> ).   | Sim | Todas as Entidades são alocadas usando <i>new</i> e desalocadas usando <i>delete</i> .  |
| 3.3      | - Gabaritos/ <i>Templates</i> criada/adaptados pelos autores (e.g., Listas Encadeadas via <i>Templates</i> ).          | Sim | Lista e Elemento são Gabaritos/ <i>Templates</i> usados para a implementação de listas encadeadas. Baseado no UML disponibilizado no moodle.    |
| 3.4      | - Uso de Tratamento de Exceções ( <i>try/catch</i> ).  | Sim | A Fase usa <i>try/catch</i> no salvamento/carregamento dos arquivos .json   |
| <b>4</b> | <b>Sobrecarga de:</b>  |     |   |
| 4.1      | - Construtoras e Métodos.  | Sim | O GerenciadorGrafico sobrecarrega a função draw para atender diferentes tipos de parâmetros (Texto, Sprite) para serem desenhados na tela.      |
| 4.2      | - Operadores (2 tipos de operadores pelo menos – Quais? ).   | Sim | Foi usado o operador ++ e – no Jogador e o operador [] na ListaEntidades  |
| ---      | <b>Persistência de Objetos (via arquivo de texto ou binário)</b>   |     |   |
| 4.3      | - Persistência de Objetos.   | Sim | Após o fim do jogo, a pontuação e o nome do jogador são salvos em um .txt   |
| 4.4      | - Persistência de Relacionamento de Objetos.   | Sim | O salvamento da fase em .json também salva quem atirou o projétil.  |
| <b>5</b> | <b>Virtualidade:</b>   |     |   |
| 5.1      | - Métodos Virtuais Usuais.   | Sim | Ao longo dos .cpp e .h que definem classes abstratas  |

|   |  |     |  |
|---|--|-----|--|
| 5.2   | - Polimorfismo.  | Sim | Diversas classes implementam as funções, derivadas da classe Ente, executar e desenhar, específicos para seu caso.           |
| 5.3   | - Métodos Virtuais Puros / Classes Abstratas.  | Sim | A class Ente, contém as funções virtuais puras executar e desenhar   |
| 5.4   | - Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto.   | Não | Usamos apenas os padrões de projeto Singleton e Observer.  |
| <b>6 Organizadores e Estáticos</b>  |  |     |  |
| 6.1   | - Espaço de Nomes ( <i>Namespace</i> ) criada pelos autores.   | Sim | No Campo de Visão que posteriormente foi colocado dentro de Entidades  |
| 6.2   | - Classes aninhadas ( <i>Nested</i> ) criada pelos autores.  | Sim | O Elemento é uma classe Aninhada de Lista  |
| 6.3   | - Atributos estáticos e métodos estáticos.   | Sim | O Ente tem um ponteiro estático para o Gerenciador_Grafico e uma função estática para definir essa ponteiro                  |
| 6.4   | - Uso extensivo de constante ( <i>const</i> ) parâmetro, retorno, método...  | Sim | Ao longo dos .h e .cpp desenvolvidos no projeto.   |
| <b>7 Standard Template Library (STL) e String OO</b>  |  |     |  |
| 7.1   | - A classe Pré-definida <i>String</i> ou equivalente. &<br>- <i>Vector</i> e/ou <i>List</i> da <i>STL</i> (p/ objetos ou ponteiros de objetos de classes definidos pelos autores)  | Sim | <i>String</i> no CampoDeTexto. <i>Vector</i> no GerenciadorColisoes.   |
| 7.2   | - Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa OU Multi-Mapa.   | Sim | Map usado nos elementos gráficos para carregar apenas uma vez as animações, imagens e fontes. (Lembrar onde está o set)      |
| <b>--- Programação concorrente</b>  |  |     |  |
| 7.3   | - <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time OU Win32API ou afins.  | Não | Não implementado por falta de tempo.   |
| 7.4   | - <i>Threads</i> (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, OU Troca de mensagens.  | Não | Não implementado por falta de tempo.   |
| <b>8 Biblioteca Gráfica / Visual</b>  |  |     |  |
| 8.1   | - Funcionalidades Elementares. &<br>- Funcionalidades Avançadas como: <ul style="list-style-type: none"><li>• tratamento de colisões</li><li>• duplo buffer</li></ul>  | Sim | O Tratamento de colisões é realizado pelo GerenciadorColisões. O Jogo é atualizado baseado no tempo passado entre os frames. |
| 8.2   | - Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive) em algum ambiente gráfico.<br><b>OU</b><br>- <i>RAD</i> – <i>Rapid Application Development</i> (Objetos gráficos como formulários, botões etc). | Sim | O GerenciadorInput recebe os eventos realizados no jogo pelo usuário, como cliques, e notifica os objetos pertinentes.       |
| <b>--- Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.</b> |  |     |  |
| 8.3   | - Ensino Médio Efetivamente.   | Sim | Gravidade em todos os corpos, quantidade de movimento e perda de energia da granada ao bater no chão.                        |

|   |   |                                 |  |
|---|---|---------------------------------|--|
| 8.4   | - Ensino Superior Efetivamente.   | Sim                             | Distribuição Normal no posicionamento dos Inimigos e Obstáculos, além do Cálculo de fator de impulsão da mina.   |
| <b>9</b>  | <b>Engenharia de Software</b>   |                                 |  |
| 9.1   | - Compreensão, melhoria e rastreabilidade de cumprimento de requisitos. &   | Sim                             | As tabelas de requisito e de conceitos foram utilizadas para o desenvolvimento do projeto.   |
| 9.2   | - Diagrama de Classes em <i>UML</i> .   | Sim                             | O Diagrama de Classe em UML foi realizado.   |
| 9.3   | - Uso efetivo e intensivo de padrões de projeto <i>GOF</i> , i.e., mais de 5 padrões.   | Não                             | Usamos apenas os padrões de projeto Singleton e Observer.  |
| 9.4   | - Testes à luz da Tabela de Requisitos e do Diagrama de Classes.  | Sim                             | A tabela e o diagrama foram usados.  |
| <b>10</b>   | <b>Execução de Projeto</b>  |                                 |  |
| 10.1  | - Controle de versão de modelos e códigos automatizados (via github e/ou afins). &<br>- Uso de alguma forma de cópia de segurança (i.e., <i>backup</i> ). | Sim                             | Github: <a href="https://github.com/gefgu/jogo-tecprog">https://github.com/gefgu/jogo-tecprog</a>  |
| 10.2  | - Reuniões com o professor para acompanhamento do andamento do projeto.   | Sim                             | Reuniões realizadas:<br>20/08 - 11:00-11:30h<br>27/08 - 11:00-11:30h   |
| 10.3  | - Reuniões com monitor da disciplina para acompanhamento do andamento do projeto.   | Sim                             | Reuniões Realizadas:<br>15/08 - 20:30-21:00 - Monitor Giovane<br>21/08 - 9:30-10:10 - Monitor Nicky<br>22/08 - 9:30-9:50 - Monitor Nicky<br>26/08 - 9:50-12:20 - Monitor Nicky |
| 10.4  | - Revisão do trabalho escrito de outra equipe e vice-versa.   | Sim                             | Gustavo Chemin e Gabriel Picinato  |
| <b>Total de conceitos apropriadamente utilizados.</b> |   | <b>90% (noventa por cento).</b> |  |

## DISCUSSÃO E CONCLUSÕES

O desenvolvimento do jogo de plataforma "Gangster" permitiu a aplicação prática dos conceitos de programação orientada a objetos ensinados na disciplina de Técnicas de Programação. Durante o projeto, foram implementados princípios essenciais, como herança, encapsulamento e polimorfismo, utilizando C++ e a biblioteca gráfica SFML. A estrutura do jogo foi cuidadosamente modelada com diagramas de classes UML, explorando técnicas avançadas, como classes abstratas e sobrecarga de operadores, proporcionando uma base sólida para o design do software.

O projeto foi bem-sucedido em criar um jogo desafiador e envolvente, com fases que apresentam níveis variados de dificuldade, inimigos com comportamentos complexos, e mecânicas que incentivam a estratégia e a habilidade do jogador. Os testes realizados demonstraram que os requisitos funcionais e de desempenho foram atendidos de maneira satisfatória. Assim, "Gangster" cumpriu seu objetivo de consolidar o aprendizado dos principais conceitos de programação orientada a objetos, fortalecendo as habilidades dos alunos no desenvolvimento de forma prática e eficaz.

## DIVISÃO DO TRABALHO

Essa seção detalha as atividades realizadas durante o trabalho do jogo, bem como quem foi responsável por quais atividades.

Tabela 4. Lista de Atividades e Responsáveis.

| Atividades.                               | Responsáveis              |
|---|---------------------------|
| Compreensão de Requisitos                 | Santos e Ferreira         |
| Diagramas de Classes                      | Santos e Ferreira         |
| Programação em C++                        | Santos e Ferreira         |
| Engenharia de Software                    | Santos e Ferreira         |
| Implementação de <i>Template</i>          | Santos                    |
| Implementação da Persistência dos Objetos | Santos e Ferreira (menos) |
| Implementação dos Gerenciadores           | Santos e Ferreira         |
| Implementação dos Obstáculos              | Santos e Ferreira         |
| Implementação dos Personagens             | Santos e Ferreira         |
| Implementação dos Menus                   | Santos e Ferreira (menos) |
| Implementação das Fases                   | Santos e Ferreira         |
| Implementação dos Objetos Graficos        | Santos e Ferreira         |
| Preparativo Apresentação                  | Santos e Ferreira         |
| Escrita do Trabalho                       | Santos e Ferreira         |
| Revisão do Trabalho                       | Santos (menos) e Ferreira |

Os dois trabalharam juntos na maior parte de todo o projeto, revisaram 100% do código e testaram diversas vezes cada modificação que foi implementada, ficando Gustavo Santos com 100% da realização do trabalho e Gustavo Ferreira com aproximadamente 93%.

## AGRADECIMENTOS PROFISSIONAIS

Agradecemos ao professor J. M. Simão, ao monitor Nicky e também ao Matheus Burda, o qual trabalha junto com o aluno Ferreira, que deram ótimas dicas na construção do projeto. Ademais, também agradecemos toda equipe do PETECO que participou na realização do “Project Simas”, disponibilizando seus tempos e conhecimentos. Por fim, aos amigos Gustavo Chemin e Gabriel Picinato por ajudarem a revisar o nosso trabalho.

## REFERÊNCIAS CITADAS NO TEXTO

[1] Endereço para o Repositório Online no GitHub - <https://github.com/gefgu/jogo-tecprog>

## REFERÊNCIAS UTILIZADAS NO DESENVOLVIMENTO

[A] SIMÃO, J. M. Site das Disciplina de Fundamentos de Programação 2, Curitiba – PR, Brasil, Acessado em 20/06/2021, às 20:32 -

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

[B] O Catálogo dos Padões de Projeto - <https://refactoring.guru/pt-br/design-patterns/catalog>

[C] SFML 2.4 for Beginners -

[https://www.youtube.com/watch?v=axIgxBQVBg0&list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY\\_SLRW9](https://www.youtube.com/watch?v=axIgxBQVBg0&list=PL21OsoBLPpMOO6zyVlxZ4S4hwkY_SLRW9)