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

Regular Game++

Lucas Tanaka, Yudi Gunzi
lucasscussel@alunos.utfpr.edu.br, yudigunzi@alunos.utfpr.edu.br

Disciplina: **Técnicas de Programação – CSE20** / S173 – 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 apresenta um modelo para o texto do trabalho de Técnicas de Programação, além de instruções/especificações para o trabalho ele mesmo e detalhes sobre sua avaliação. 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.* ***ISTO DITO USE O RESUMO Q SEGUE APENAS MUDANDO AS PARTES EM VERMELHO SE FOR O CASO***: 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 **RegularGame**, no qual o jogador enfrenta inimigos em um **dado cenário**. O jogo tem **duas 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) via Diagrama de Classes em Linguagem de Modelagem Unificada (*Unified Modeling Language - UML*) usando como base um diagrama assaz genérico e 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 ditos 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 elaborada. Por fim, salienta-se que o desenvolvimento em questão permitiu cumprir o objetivo de aprendizado visado.

Palavras-chave ou Expressões-chave (máximo quatro itens, não excedendo três linhas)**:** Artigo-Relatório para o Trabalho em Técnicas de Programação, Trabalho Acadêmico Voltado a Implementação em C++.

INTRODUÇÃO

Este trabalho foi desenvolvido no contexto da disciplina de Técnicas de Programação, com o objetivo de aplicar os conceitos de Programação Orientada a Objetos (POO) e gerenciamento de memória em C++, consolidando os conhecimentos teóricos por meio de uma implementação prática. A proposta consistiu na criação de um jogo de plataforma, seguindo as diretrizes estabelecidas pelo professor, de modo a integrar os princípios de modelagem, arquitetura de software e boas práticas de codificação.

O objeto de estudo e implementação deste projeto é um jogo de plataforma 2D, desenvolvido em C++ utilizando o padrão de 2003, com o auxílio da biblioteca SFML (Simple and Fast Multimedia Library) para tratamento de gráficos e interações do usuário. O jogo foi projetado para demonstrar a aplicação de herança, polimorfismo, encapsulamento e outros pilares da POO, além de técnicas de otimização de recursos e tratamento de colisões.

O método adotado seguiu um ciclo simplificado de Engenharia de Software, partindo da

compreensão dos requisitos e da modelagem do sistema por meio de diagramas de classes UML, derivados do modelo fornecido. Em seguida, realizou-se a implementação em C++, com testes iterativos para garantir o funcionamento correto do software. A abordagem permitiu alinhar a solução desenvolvida com as expectativas da disciplina, garantindo robustez e organização no código.

Nas próximas seções, serão detalhados os aspectos técnicos do projeto, incluindo a arquitetura adotada, as decisões de design, os desafios enfrentados e os resultados obtidos. Além disso, serão apresentadas as considerações finais, destacando os aprendizados e possíveis melhorias para versões futuras do trabalho.

EXPLICAÇÃO DO JOGO EM SI

Conceito Geral e Jogabilidade

RegularGame++ é um jogo 2D de ação e plataforma com geração procedural de fases, garantindo que cada partida seja única. O jogo se destaca por sua jogabilidade ágil e desafiadora, combinando movimentação precisa, combate dinâmico e obstáculos imprevisíveis. O jogador controla um personagem que pode andar, pular e atacar, enfrentando inimigos variados e navegando por plataformas que mudam de posição a cada nova execução. A alta rejogabilidade é um dos principais atrativos, pois nenhuma partida é igual à anterior, exigindo adaptação constante do jogador.

Objetivos e Condições de Vitória

A vitória no jogo só é alcançada quando o jogador chega ao final da fase, superando todos os obstáculos e inimigos gerados proceduralmente. A pontuação, obtida ao derrotar inimigos, serve como um bônus para competição ou auto-desafio, mas não influencia diretamente na conclusão do jogo. Se o jogador morrer, a partida acaba em derrota, e uma nova fase procedural é gerada ao reiniciar.

Regras Fundamentais e Mecânicas

<u>Controles básicos:</u>

Movimentação (Player1: teclas direcionais; Player2: WASD).

Ataque Giratório (Player1: espaço; Player2: Left Shift).

<u>Sistema de vida:</u>

O jogador tem 30 corações. Se perder todos eles, a fase termina e o jogo acaba em derrota.

Tocar em inimigos, estacas, serras ou ser atingido por projéteis causa dano.

Cada tipo de entidade possui atributos únicos.

Estaca: Obstáculo estático que causa dano ao jogador.

Serra: Obstáculo que se movimenta de um lado ao outro e causa dano ao jogador.

Youkai (inimigo fácil): Inimigo com movimento simples que causa dano ao contato.

Cannonhead (inimigo médio): Inimigo com movimento simples que atira projéteis na direção em que está olhando.

Ghost (inimigo difícil/chefão): Inimigo com movimento complexo, envolvendo pulos e perseguição ao jogador.

Geração procedural:

Plataformas e obstáculos são gerados aleatoriamente em posições pré-definidas, com mínimo de 3 e máximo dependendo do mapa da fase.

Inimigos são gerados aleatoriamente em posições pré-definidas com mínimo de 3 e máximo dependendo do mapa da fase. Ao serem criados, são atribuídos aleatóriamente um valor de maldade de 1 a 5, que definirá a magnitude dos atributos de tal inimigo, como velocidade e dano.

DESENVOLVIMENTO DO JOGO NA VERSÃO ORIENTADA A OBJETOS

O começo do processo de criação do jogo foi muito baseado no diagrama de classes disponibilizado pelo Professor, com a adaptação dele ao nosso jogo foi possível ter uma noção do que seria o jogo de uma forma mais primitiva, porém creio que este início foi um pouco prejudicado por uma procura por elementos gráficos como sprites, que logo foi percebido e saiu da rota de prioridades.

A construção do jogo foi sendo feita classe a classe, com primeiramente a idealização das funcionalidades que ela teria e após isso a implementação seguida de vários testes e ajustes necessários. Ao passo que mais classes foram sendo criadas, a necessidade de novos métodos em classes já implementadas surgia, e para isso uma análise clara das relações firmadas entre elas era necessária assim aprimorando o código como um todo aos poucos.

Essa abordagem e a utilização de ferramentas como git e github foi também essencial para o desenvolvimento do jogo, ajudando muito no versionamento e trabalho em equipe, assim como os diagramas uml que eram necessários para as reuniões previstas com o Professor, ajudando na visualização do projeto e a percepção da evolução do software.

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, escolher ver colocação (*ranking*) de jogadores e escolher demais opções pertinentes (previstas nos demais requisitos).	**REALIZADO**	Cf. classe MenuState, com suporte da SFML.
2	Permitir um ou dois jogadores com representação gráfica aos usuários do Jogo, sendo que no último caso é para que os dois joguem de maneira concomitante.	**REALIZADO**	Cf. classe Player cujos objetos estão agregados em Stage. A quantidade de jogadores pode ser escolhida vida menu. Os dois jogadores possuem diferenças visuais.
3	Disponibilizar ao menos duas fases distintas que podem ser jogadas sequencialmente ou selecionadas, via menu, nas quais jogadores tentam neutralizar inimigos por meio de algum artifício e vice-versa.	**REALIZADO**	Cf. classes DayMountainStage e NightMountainStage derivadas de Stage. Jogadores podem neutralizar inimigos via ataque.
4	Ter pelo menos três tipos distintos de inimigos, cada qual com sua representação gráfica, sendo que ao menos um deles deve poder lançar projetil contra o(s) jogador(es) e um dos inimigos dever ser um 'chefão'.	**REALIZADO**	Cf. classes Youkai, Cannonhead (lança projétil) e Ghost ('chefão').
5	Ter a cada fase ao menos dois tipos de inimigos (um deles exclusivo nela) com número aleatório de instâncias, podendo ser várias instâncias (definindo um máximo) e sendo pelo menos 3 instâncias para cada tipo que estiver na fase.	**REALIZADO**	Cf. Youkai (presente em ambas as fases), Cannonhead (exclusivo de DayMountainStage), Ghost (exclusivo de NightMountainStage).
6	Ter três tipos de obstáculos, cada qual com sua representação gráfica, sendo que ao menos um causa dano em jogador se colidirem.	**REALIZADO**	Cf. classes Platform, Spike e Saw. Spike e Saw causam dano.
7	Ter em cada fase ao menos dois tipos de obstáculos (um deles exclusivo nela) com número aleatório (definindo um máximo) de instâncias (*i.e.*, objetos), sendo pelo menos 3	**REALIZADO**	Cf. Platform (presente em ambas as fases), Spike (exclusivo de DayMountainStage),

	instâncias por tipo.		Saw (exclusivo de NightMountainStage).
8	Ter em cada fase um cenário de jogo constituído por obstáculos, sendo que parte deles **devem ser** plataformas ou similares, sobre as quais pode haver inimigos e podem subir jogadores. Em cada fase, só poder ter um tipo coincidente de inimigo e um tipo coincidente de obstáculo (que é a	**REALIZADO**	Cf. classes Stage, Platform (obstáculo coincidente), Youkai(inimigo coincidente).

	plataforma) em relação as demais fases.		
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 de alguma ´gravidade´ no âmbito deste jogo de plataforma vertical e 2D.	**REALIZADO**	Cf. classe CollisionManager, pacote Entities.
10	Permitir: (1) salvar nome do usuário, manter/salvar pontuação (incrementada via neutralização de inimigos) do jogador controlado pelo usuário e gerar lista de pontuação (*ranking*). **E** (2) Pausar e **Salvar/Recuperar** Jogada.	**NÃO** realizado.	Cf. classes DayMountainStage, NightMountainStage, StateStack.
Total de requisitos funcionais apropriadamente realizados.			**100%** (cem por cento).
Os requisitos dependem em algo uns dos outros, na chamada interdependência de requisitos.			

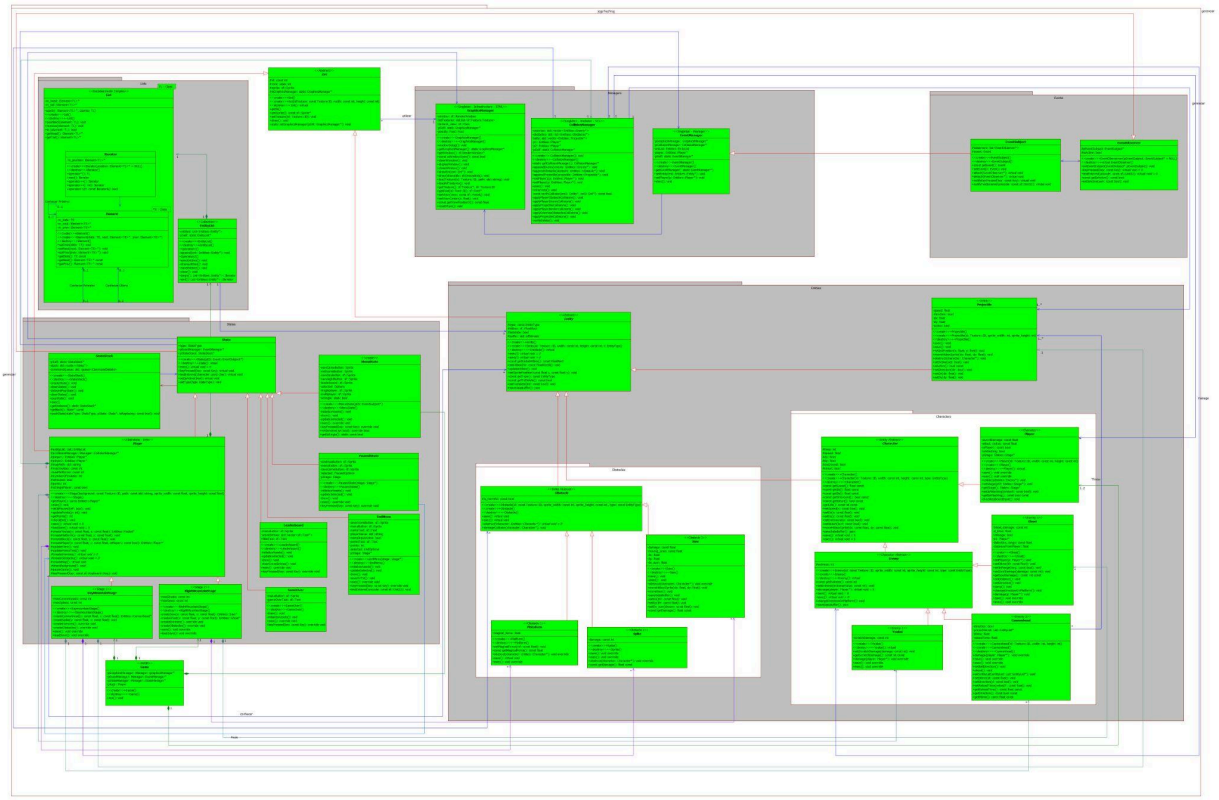

Figura 1-Diagrama de Classes UML final

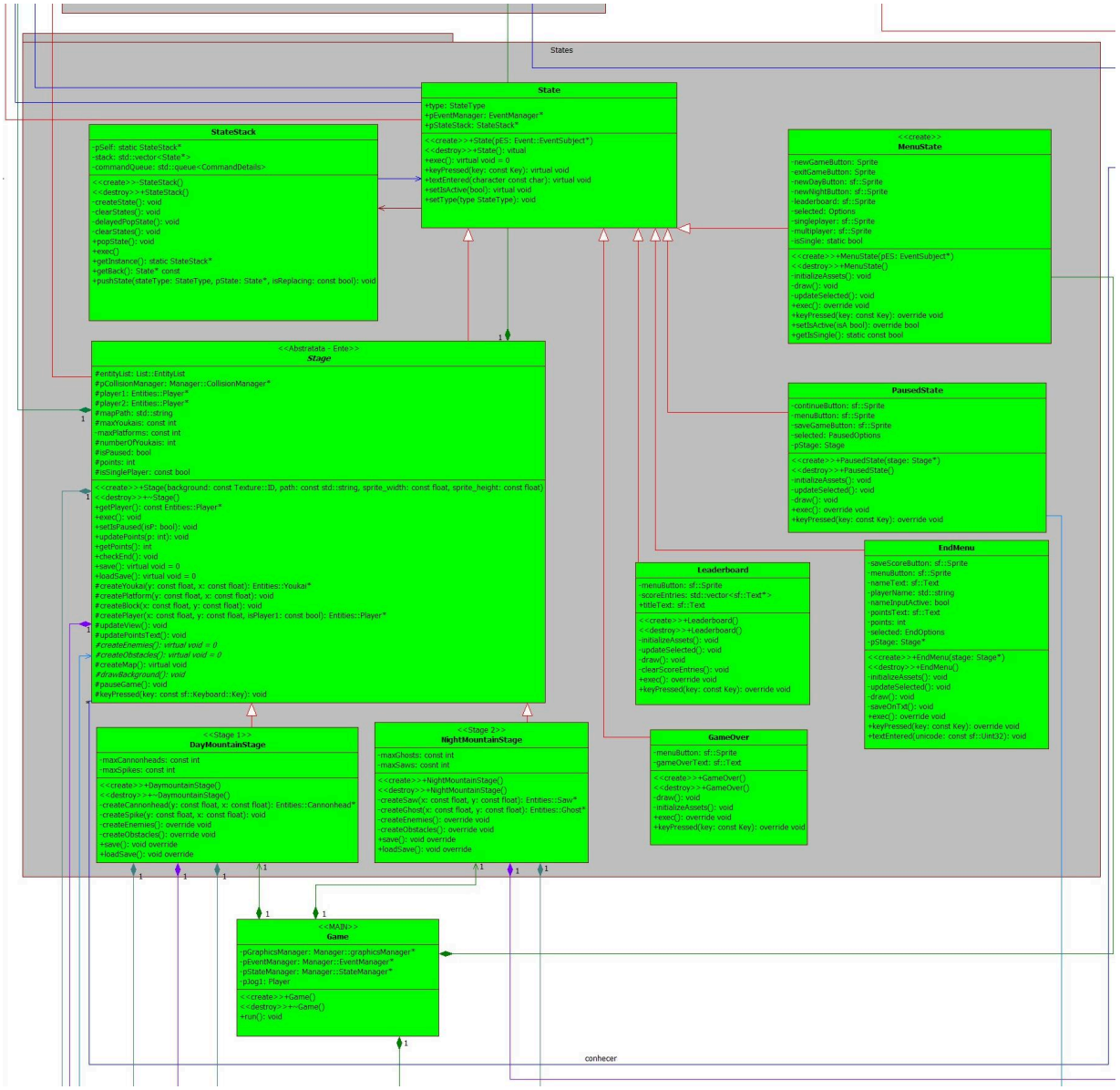

Figura 2-Namespace States

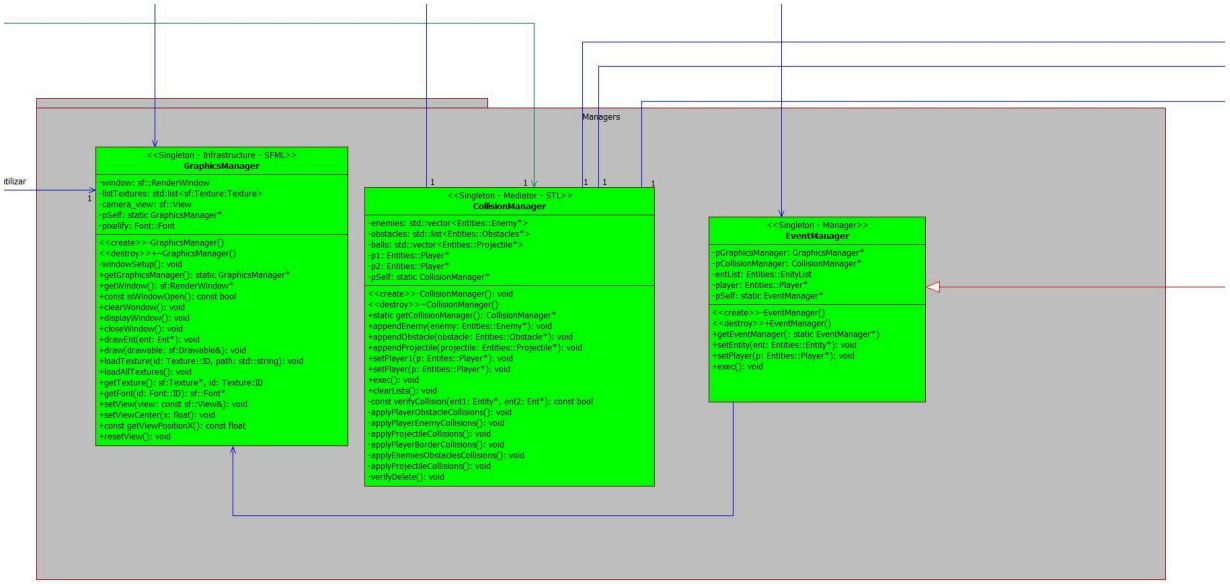

Figura 3-Namespace Managers

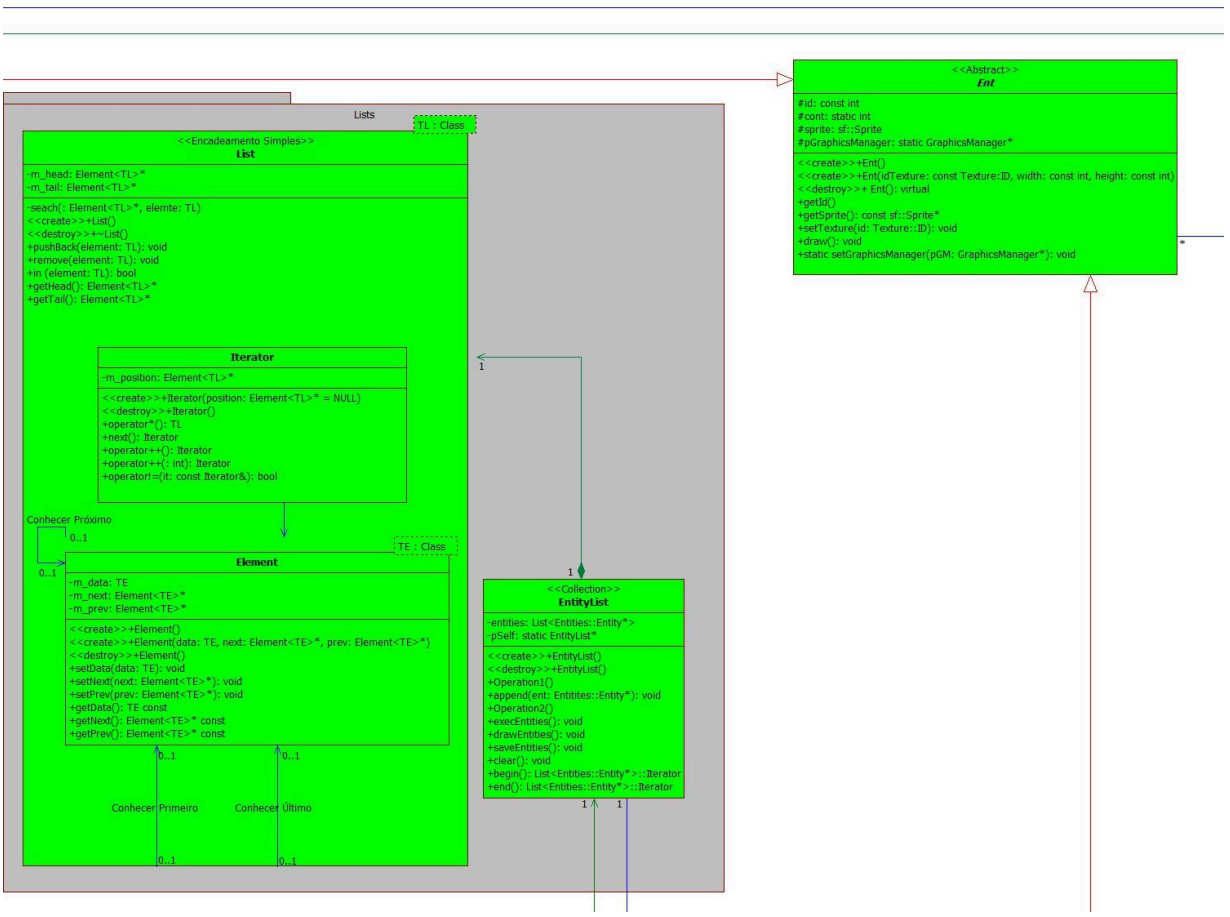

Figura 4-Namesspace Lists e Classe Ent

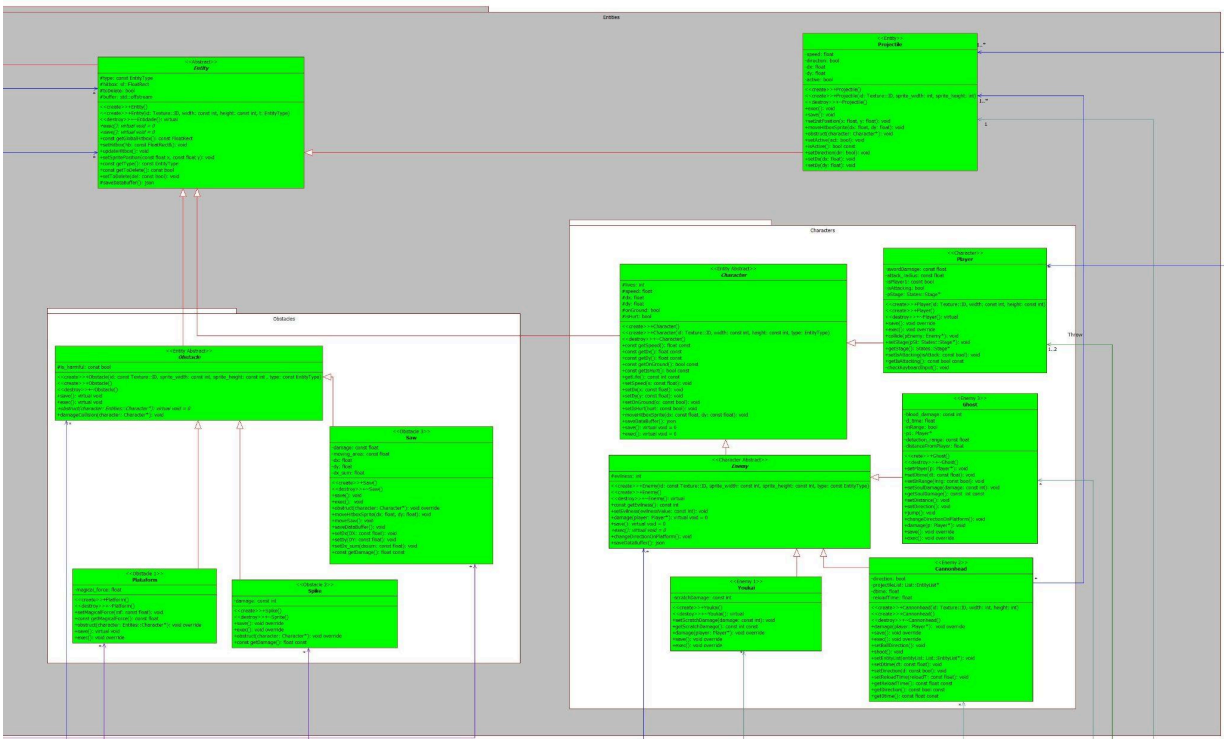

Figura 5-Namespace Entities

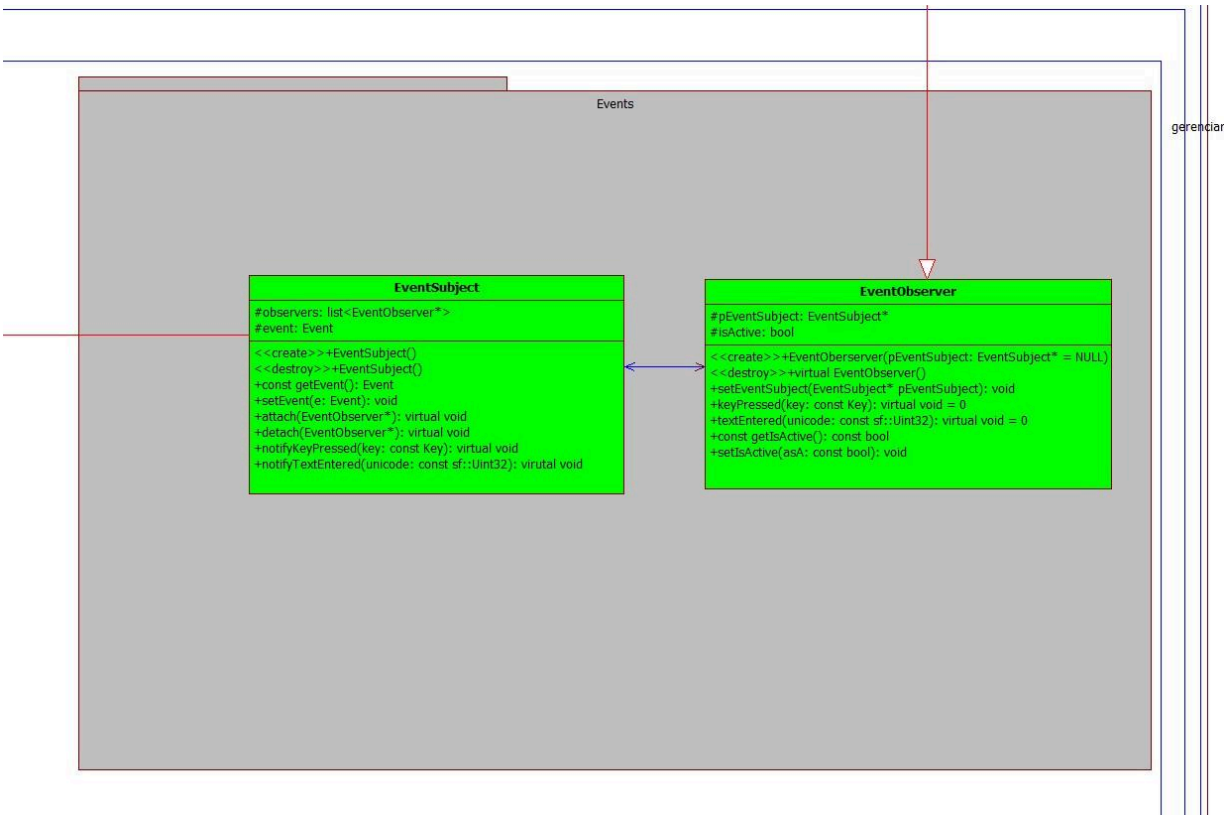

Figura 6-Namespace Events

A maioria das classes presentes no projeto herdam de Ent, destaca-se o atributo sprite e o método draw(), portanto toda classe que possui um sprite desenhável na tela herda de Ent. Entrando no namespace Entities, temos todas as entidades do jogo, separadas principalmente entre Obstacle, Character e Projectile, sendo todas elas filhas de Entity. A classe Entity possui tudo que se vê necessária em uma entidade, como coordenadas, hitbox, velocidade, movimento, e também uma interessante implementação de polimorfismo para salvar todas as informações das entidades em arquivos. A classe Character serve como classe abstrata para outras duas, sendo elas Enemy, que vai englobar todos os inimigos pertinentes no jogo, cada um com comportamentos diferentes, únicos e inovadores, e a classe player. Outro namespace importante é o Manager, que contém todos os gerenciadores do projeto. Existe também o namespace List, com todas as classes de apoio que vão ajudar a criar listas para, principalmente, guardar as entidades. A classe EntityList, que utiliza List como template, possui métodos para facilitar o percorrimento de todas as entidades existentes no jogo e chamar métodos como exec(), que mostram o grande poder do polimorfismo. O namespace Event contém as classes EventSubject e EventObserver, que tem como função aplicar o padrão de projeto Observer no projeto. O único EventSubject no projecto é o EventManager, que possui um vetor de EventObservers para percorrer, sendo a classe State também a única classe no projeto a herdar dessa vez de EventObservers. Essa classe está dentro do namespace States, o qual possui classes fundamentais para state como State, e StackStack que é usada para criar um pilha e gerenciar os diferentes states criados durante a duração do jogo. Este namespace também contém os diferentes tipos de States, como Pause, Menu, EndGame e Stage. A classe Stage além de ser um State é também quem gerencia a criação de entidades, execução do jogo e qualquer outro aspecto em relação com o jogo em si. Duas classes herdam de Stage, sendo elas os diferentes mapas disponíveis para jogar, que podem ser escolhidos no Menu.

TABELA DE CONCEITOS UTILIZADOS E NÃO UTILIZADOS

O propósito da tabela a seguir é apresentar quais dos conceitos aprendidos na disciplina de Técnicas de Programação foram usados para a construção do projeto, justificando as escolhas feitas em cada situação, para apresentar domínio do conteúdo da disciplina.

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

N.	Conceitos	Uso	O quê / Onde & Justificativa em uma frase

1	**Elementares:**		
1.1 **&**	- Classes, objetos. & -Atributos(privado), variáveis e constantes. - Métodos (com e sem retorno).	Sim	- Todos .h e .cpp, como nas classes nos *namespaces* States e Entities. - Classes, Objetos, Atributos e Métodos foram utilizados porque são conceitos elementares na orientação a objetos. Exemplo de '&' em getSprite() na classe Ent, usado para retornar endereço de sprite.
1.2 **&**	- Métodos (com retorno *const* e parâmetro *const*). - Construtores (sem/com parâmetros) e destrutores	Sim	- Na maioria dos .h e .cpp, como nas classes nos *namespaces* Entities e States. - A constância pertinente evita mudanças equivocadas, construtores são mandatórios para inicializar atributos e destrutores pertinentes para finalizações como desalocações.
1.3	- Classe Principal.	Sim	- main.cpp
1.4	- Divisão em .h e .cpp.	Sim	- No desenvolvimento como um todo, como nas classes nos *namespaces* Entities e States. - Permite organizar as classes e afins que compõem o sistema.
2	**Relações de:**		
2.1	- Associação direcional. & - Associação bidirecional.	Sim	- Há associações direcionais evidentes, como a classe game, possuindo ponteiros para `Graphics Manager`, Event Manager. Associações bidirecionais são observadas entre Player e Stage.
2.2 **&**	- Agregação via associação. - Agregação propriamente dita.	Sim	- Stage agrega Obstáculos e Inimigos. As duas classes agregadas existem apenas dentro de uma fase.
2.3 **&**	- Herança elementar. - Herança em vários níveis.	Sim	- Conceito fundamental em POO. Observado nas heranças de Ent, Entity, Character, Player, por exemplo.
2.4	- Herança múltipla.	Sim	- A classe Stage herda de Ent e de State.
3	**Ponteiros, generalizações e exceções**		
3.1	- Operador *this* para fins de relacionamento bidirecional	Sim	- A classe State usa *this* para se adicionar ao EventManager. - Utilizado para adicionar Observer à lista de observers do Subject.
3.2	- Alocação de memória (*new* & *delete*).	Sim	- *new* usado nos métodos create dos stages, para criar objetos que são adicionados a EntityList. - Na destrutora de EntityList, esses objetos são deslocados com *delete.*
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores para Listas.	Sim	- Criação de List.h, que por conseguinte é usada para criar EntityList.

3.4	- Uso de Tratamento de Exceções (*try catch*).	Sim	- Usado em Stage na função createYoukai(), para tratar possíveis erros na alocação de Youkai.
4	**Sobrecarga de:**		
4.1	- Construtoras e Métodos.	Sim	- Character, Enemy, Entity, Ent usam mais de uma construtora. - changeDirectionOnPlatform() em Ghost é uma sobrecarga da mesma função em Enemy.
4.2	- Operadores (2 tipos de operadores pelo menos).	Sim	Em Iterator do namespace List, utilizado para melhor navegação dos elementos da lista.
---	**Persistência de Objetos (via arquivo de texto ou binário)**		
4.3	- Persistência de Objetos.	Sim	-saveOnTxt de EndMenu salva informações em leaderboard.txt -saveDataBuffer() e save() salvam dados em save.json
4.4	- Persistência de Relacionamento de Objetos.	Não	
5	**Virtualidade:**		
5.1	- Métodos Virtuais Usuais.	Sim	-Há diversos métodos virtuais, por exemplo save() e exec() em Entity, que se tornam usuais em classes derivadas como Player, Youkai e Ghost, que são utilizados para aplicação de polimorfismo.
5.2	- Polimorfismo.	Sim	-execEntities() em EntityList chama o método virtual exec() de Entity dos objetos da lista, que por sua vez chama a implementação de exec() das classes derivadas.
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Sim	-exec() e save() em Entity são utilizados para aplicação de polimorfismo. O mesmo acontece em obstruct() de Obstacle.
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto (mais de 5 padrões).	Sim	-Singleton usado em todos as classes no namespace Manager, pois são usados apenas um Manager de cada tipo em todo o jogo. -Iterator utilizado em List para melhor navegação da lista.
6	**Organizadores e Estáticos**		

6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Sim	-Namespace State, Entities, Manager, Event, por exemplo, usados para melhor organização do código.
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Sim	-As classes Element e Iterator de List

6.3	- Atributos estáticos e métodos estáticos.	Sim	-Na aplicação do Singleton para os managers -Em *cont* em Ent para incremento de *id.*
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Sim	-Em métodos set e get o const é constantemente usado em parâmetros e retorno. -Em parâmetros de construtoras const é constantemente usado.
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	-Std string usada em EndMenu para receber e mudar o nome inserido. -São usadas *List* e *Vectors* para controle de entidades em CollisionManager.
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Sim	-O controle dos States e feitos por StateStack, que possui um stack de States*.
---	**Programação concorrente**		
7.3	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run Time **OU** Win32API ou afins.	*Nao*	
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	Nao	
8	**Biblioteca Gráfica / Visual**		
8.1 **&**	- Funcionalidades Elementares. - Funcionalidades Avançadas como: tratamento de colisões, duplo *buffer* ou outros.	Sim	-Foram usados elementos da biblioteca gráfica como sprites, window, textures e respectivos métodos presentes nessas classes, pois esses elementos são essenciais para o jogo. - Demasiada utilização de FloatRects e elementos parecidos presentes na biblioteca para navegação de coordenadas. -Tratamento de colisões feito com elementos de coordenadas da biblioteca e seus respectivos métodos como *Intersect()*.

8.2 **OU**	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive, via padrão de projeto *Observer*) em algum ambiente gráfico. - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	Não	
---	**Interdisciplinaridades via utilização de Conceitos de Matemática Contínua e/ou Física.**		
8.3	- Ensino Médio Efetivamente.	Sim	-Movimento retilíneo uniforme na movimentação de personagens e gravidade. Gravidade é implementada com aceleração.
8.4	- Ensino Superior Efetivamente.	Sim	- Velocidade terminal implementada em Player.
9	**Engenharia de Software**		

9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos.	Sim	- Foi seguido o diagrama base disponibilizado pelo professor vigente. - Foram seguidos os requisitos presentes neste documento na aba Requisitos.
9.2	- Diagrama de Classes em *UML*.	Sim	- Diagrama realizado utilizando o software StarUML e com os conceitos vistos em sala de aula.
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, mais de 5 padrões.	Não	Foram implementados 2
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Sim	- Jogo foi realizado seguindo a tabela de requisitos e o diagrama de classes base como referência.
10	**Execução de Projeto**		
10.1 **&**	- Controle de versão de modelos e códigos automatizado (via github). - Uso de alguma forma de cópia de segurança (*i.e.*, *backup*).	Sim	Foi utilizado git e github https://github.com/Chico7854/RegularGame

10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	**Sim**	Foram feitas duas reuniões com o professor nos dias 11/06/2025 e 18/06/2025.
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO PARA A ENTREGA DO TRABALHO]**	**Sim**	Foram cumpridas 2 horas de reuniões com o PETECO, por ambos os autores, por meio das oficinas: -Gerenciador de Colisões e Gerenciador Gráfico; - Fase, TileMap e Padrão de Projeto;
10.4 &	- Escrita do trabalho e feitura da apresentação - Revisão do trabalho escrito de outra equipe e vice-versa.	Sim	Revisão do trabalho escrito realizada por Victor Hugo e pela dupla formada por Nicolas Krause e Gustavo Moretto.
Total de conceitos apropriadamente utilizados.			**87%** (oitenta e sete por cento).

DISCUSSÃO E CONCLUSÕES

Ao longo da realização deste trabalho foi possível ver uma notável evolução da equipe na construção de códigos em C++ orientada a objetos. Conceitos importantes como herança, templates, agregação e polimorfismo foram aplicados e compreendidos de forma eficaz.

Além dos aprendizados no C++, o uso de ferramentas como a biblioteca gráfica SFML, e o GitHub que é amplamente utilizado no mercado de trabalho são conhecimentos valiosos para possuir. Este projeto proporcionou também uma boa introdução a conceitos usados na indústria como o seguimento de requisitos sólidos e elaboração de relatórios robustos.

O resultado do projeto foi positivo, com o cumprimento de todos os requisitos solicitados de forma a criar um jogo coeso e alinhado aos objetivos propostos.

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. A tabela 4 pode e mesmo deveria ser melhorada à luz das tabelas de requisitos e conceitos.

Tabela 4. Lista de Atividades e Responsáveis.

N.	Atividades	Responsáveis
1	**Elementares: AMBOS**	
1.1	- Classes, objetos. & - Atributos (privados), variáveis e constantes. & - Métodos (com e sem retorno).	Yudi e Lucas
1.2	- Métodos (com retorno *const* e parâmetro *const*). & - Construtores (sem/com parâmetros) e destrutores	Yudi e Lucas
1.3	- Classe Principal.	Yudi e Lucas
1.4	- Divisão em .h e .cpp.	Yudi e Lucas
2	**Relações de: AMBOS**	
2.1	- Associação direcional. & - Associação bidirecional.	Yudi e Lucas
2.2	- Agregação via associação. & - Agregação propriamente dita.	Yudi e Lucas
2.3	- Herança elementar. & - Herança em vários níveis.	Yudi e Lucas
2.4	- Herança múltipla.	Yudi e Lucas
3	**Ponteiros, generalizações e exceções: Mais Lucas**	
3.1	- Operador *this* para fins de relacionamento bidirecional.	Yudi e Lucas
3.2	- Alocação de memória (*new* & *delete*).	Yudi e Lucas
3.3	- Gabaritos/*Templates* criada/adaptados pelos autores para Listas.	Lucas
3.4	- Uso de Tratamento de Exceções (*try catch*).	Lucas
4	**Sobrecarga de: Mais Lucas**	
4.1	- Construtoras e Métodos.	Yudi e Lucas
4.2	- Operadores (2 tipos de operadores pelo menos)	Lucas
---	**Persistência de Objetos (via arquivo de texto ou binário): Mais Lucas**	

4.3	- Persistência de Objetos.	Yudi e Lucas

4.4	- Persistência de Relacionamento de Objetos.	
5	**Virtualidade: Ambos**	
5.1	- Métodos Virtuais Usuais.	Yudi e Lucas
5.2	- Polimorfismo.	Yudi e Lucas
5.3	- Métodos Virtuais Puros / Classes Abstratas.	Lucas
5.4	- Coesão/Desacoplamento efetiva e intensa com o apoio de padrões de projeto (mais de 5 padrões).	Yudi e Lucas
6	**Organizadores e Estáticos:Mais Lucas**	
6.1	- Espaço de Nomes (*Namespace*) criada pelos autores.	Yudi e Lucas
6.2	- Classes aninhadas (*Nested*) criada pelos autores.	Lucas
6.3	- Atributos estáticos e métodos estáticos.	Yudi e Lucas
6.4	- Uso extensivo de constante (*const*) parâmetro, retorno, método...	Lucas
7	**Standard Template Library *(STL)* e String OO: Ambos**	
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)	Yudi e Lucas
7.2	- Pilha, Fila, Bifila, Fila de Prioridade, Conjunto, Multi-Conjunto, Mapa **OU** Multi-Mapa.	Yudi e Lucas
---	**Programação concorrente: …**	
7.3	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos, utilizando Posix, C-Run-Time **OU** Win32API ou afins.	. . .
7.4	- *Threads* (Linhas de Execução) no âmbito da Orientação a Objetos com uso de Mutex, Semáforos, **OU** Troca de mensagens.	. . .
8	**Biblioteca Gráfica / Visual: Ambos**	
8.	- Funcionalidades Elementares. **&**	Yudi e Lucas

1	- Funcionalidades Avançadas como: tratamento de colisões e duplo *buffer*	
8.2	- Programação orientada e evento efetiva (com gerenciador apropriado de eventos inclusive, via padrão de projeto *Observer*) em algum ambiente gráfico. **OU** - *RAD – Rapid Application Development* (Objetos gráficos como formulários, botões etc).	
---	**Interdisciplinaridades via uso de Conceitos de Matemática Contínua e/ou Física: Ambos**	
8.3	- Ensino Médio Efetivamente.	Yudi e Lucas
8.4	- Ensino Superior Efetivamente.	Yudi e Lucas
9	**Engenharia de Software: Ambos**	
9.1	- Compreensão, melhoria e rastreabilidade de cumprimento de requisitos.	Yudi e Lucas
9.2	- Diagrama de Classes em *UML*.	Yudi e Lucas
9.3	- Uso efetivo e intensivo de padrões de projeto *GOF*, *i.e.*, + de 5 padrões.	Yudi e Lucas
9.4	- Testes à luz da Tabela de Requisitos e do Diagrama de Classes.	Yudi e Lucas
10	**Execução de Projeto:Ambos**	
10.1	- Controle de versão de modelos e códigos automatizado (via github). *& - Uso de alguma forma de cópia de segurança (i.e., backup).*	Yudi e Lucas
10.2	- Reuniões com o professor para acompanhamento do andamento do projeto. **[ITEM OBRIGATÓRIO A ENTREGA DO TRABALHO]**	Yudi e Lucas
10.3	- Reuniões com monitor da disciplina para acompanhamento do andamento do projeto. ***[ITEM OBRIGATÓRIO PARA A ENTREGA]***	Yudi e Lucas
10.4	- Escrita do trabalho e feitura da apresentação **&** - Revisão do trabalho escrito de outra equipe e vice-versa.	Yudi e Lucas

- Lucas trabalhou em 50% das atividades as realizando ou colaborando nelas efetivamente.
-Yudi trabalhou em 50% das atividades as realizando ou colaborando nelas efetivamente.

AGRADECIMENTOS PROFISSIONAIS

Agradecemos ao Victor Hugo pela revisão criteriosa deste trabalho, contribuindo para a qualidade e clareza do documento.

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 20/06/2021, às 20:32 - 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.