UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO E ENGENHARIA DE COMPUTAÇÃO

ANDRÉ DEXHEIMER
GABRIEL PISCOYA
RODRIGO WIEBBELLING

Projeto Inicial do Trabalho Final para a Disciplina de MLP do Instituto de Informática da UFRGS

Relatório apresentado como requisito parcial para a obtenção de conceito na Disciplina de Modelos de Linguagens de Programação

Prof. Dr. Lucas Mello Schnorr
Orientador

Porto Alegre
2017

SUMÁRIO

1 INTRODUÇÃO

Este capítulo tem o objetivo de descrever de forma sucinta a historia das linguagens de programação e os principais tópicos envolvidos na realização deste trabalho. Logo após, serão abordados os temas diretamente relacionados ao trabalho.

Historia das Linguagens de Programação

As primeiras linguagens de programação eram simples códigos utilizados para automatizar processos nem sempre relacionadas à computação. Na década de 1940, com a criação do primeiro computador moderno, eram utilizados cartões perfurados para facilitar o processo de programação e diminuir a quantidade de erros introduzidos pelo programador. Não foi até meados de 1950 que surgiu a primeira linguagem de programação moderna: FORTRAN, criada por John Backus. Os seguintes anos foram frutíferos, vieram acompanhados de duas novas linguagens de programação: LISP - John McCarthy e COBOL - Grace Hopper.

No começo, todas as linguagens de programação somente permitiam a criação de programas monolíticos e careciam de recursos que facilitassem sua utilização. Somente no fim da década de 1970 que foram estabelecidos os principais paradigmas de programação conhecidos hoje em dia: imperativo, funcional e lógico. Durante estes anos, surgiu o termo "Programação Estruturada", que visava restringir o uso de desvios incondicionais (GoTo) (ORGANICK; FORSYTHE; PLUMMER, 2014).

Em 1980, foi criada C++, que combinava orientação a objetos e programação de sistemas, também foi introduzida uma mudança de pensamento na concepção de linguagens de programação, junto com o movimento RISC em arquitetura de computadores, despertou-se maior interesse no uso de compiladores para linguagens de alto nível.

Com a chegada da internet, sugiram as linguagens de scripting, que não são evolução direta de nenhuma linguagem já estabelecida anteriormente, e que foram concebidas com novas sintaxes e novas funções (CERUZZI, 1998).

Ambiente e Linguagem de Programação

Como o objetivo do trabalho é aproximar os alunos das linguagens de programação modernas, optamos por escolher uma linguagem que seja amplamente usada na atualidade, também sabemos que ela deve ser multi paradigma, já que devemos implementar soluções utilizando dois paradigmas diferentes. Pelos motivos citados previamente, escolhemos **C++**.

Problema Abordado

A intenção inicial foi a de resolver um problema que já fosse conhecido pelos integrantes do grupo e que despertasse o interesse de todos, portanto escolhemos **Tower Defence**.

2 A LINGUAGEM C++

A linguagem de programação C++ foi criada por Bjarne Stroustrup nos anos de 1980, vindo a ter sua padronização ISO apenas 18 anos depois em 1998. Ela é uma linguagem compilada multi-paradigma, com suporte ao modelo imperativo, ao orientado a objetos, ao genérico, entre outros. Por causa disso, é de uso amplo entre as linguagens comerciais e acadêmicas.

Algumas características do C++

Operadores: O C++ possui todo o conjunto de operadores do C, além de alguns implementados apenas no C++, que dizem respeito à conversão entre tipos, os quais que podem ser const_cast, static_cast, dynamic_cast e reinterpret_cast. Além disso, a linguagem possui sobrecarga de operadores, permitindo que um mesmo operador tenha mais do que 1 significado dependendo do contexto em que é utilizado.

Pré-Processador: antes da compilação propriamente dita, o C++ passa pelo seu pré-processador, gerando modificações léxicas que servem como entrada para a compilação.

Objetos: O C++ tem suporte aos conceitos de orientação à objetos, permitindo a criação de classes que apresentam quatro características desses conceitos: abstração, encapsulamento, herança e polimorfismo. O encapsulamento permite proteger atributos e métodos do objeto, dessa forma é possível que outros trechos do programa tenham acesso apenas aos métodos de interface com a classe. A herança de classes permite que uma classe herde atributos e métodos de outra, podendo ser relacionado com a ideia de classes mãe e filha. O polimorfismo trata da capacidade de se utilizar um operador ou método de diferentes maneiras, facilitando a estendibilidade da classe.

Tratamento de Exceções: Erros podem ser tratados pelo sistema, permitindo que a aplicação se recupere de algum erro sem travar ou ter de ser fechada.

Espaço de Nomes: Permite uma melhor organização das bibliotecas, de forma que cada uma pode criar o seu próprio espaço de nomes para que não existam conflitos.

3 TOWER DEFENCE

É um estilo de jogo de estratégia que consiste em defender uma determinada entidade de inimigos. No nosso jogo, a entidade em questão é uma torre que se encontra no centro da tela. Esta torre possui uma certa quantidade de vida, velocidade de ataque, penetração de armadura, dano e alcance de ataque. Tais características podem ser melhoradas e outras habilidades podem ser adquiridas por meio de compras com a unidade monetária do jogo, obtida matando os inimigos.

Objetivo do jogo

Defender a sua torre de ondas progressivamente maiores de inimigos progressivamente mais fortes.

Os inimigos

Eles têm como objetivo atacar a torre até que sua vida chegue a 0 pontos, surgem de pontos aleatórios nas bordas da tela e vão em direção a torre. Eles possuem atributos definidos pelo nível do jogo, como: velocidade de ataque e de movimento, poder de ataque, quantidade de vida e de defesa. Existem 3 classes de inimigos: a classe "Soldier"se trata de um soldado que anda a pé e possui apenas armas de curto alcance. Ele vai em direção ao centro da tela e somente danifica a torre ao chegar nela. A classe "Horseman"se comporta de maneira semelhante ao soldier, porém possui mais defesa, dano de ataque e velocidade de movimento. A classe "Archer"é a que mais se diferencia das outras pois consegue atacar a torre de longas distâncias, tendo em suas características algo que as outras classes não têm, a distância de ataque, que indica a distância da qual o inimigo deve estar da torre para poder atacá-la.

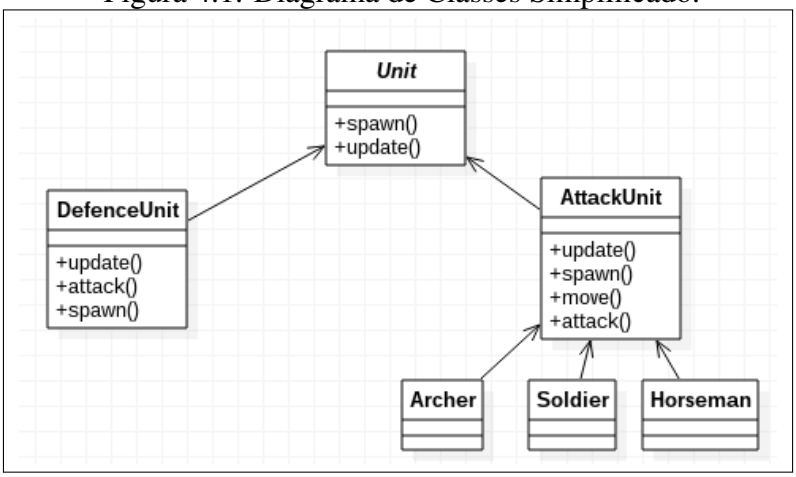
4 IMPLEMENTAÇÃO ORIENTADA A OBJETOS

Na linguagem C++, o paradigma orientado a objetos é o mais conhecido, logo é o mais utilizado entre os programadores. Serão introduzidos os conceitos principais abordados neste trabalho para depois realizar uma breve discução sobre quando e onde devem ser utilizados.

Classes

O propósito principal da programação em C++ é a acrescentar orientação a objetos à linguagem C. Classes são a característica principal da linguagem C++ que suporta orientação a objetos, também são chamadas de tipos definidos pelo usuário. A diferença das estruturas de dados presentes em C, as classes combinam a representação dos dados com métodos associados, formando assim uma unidade compacta.

Figura 4.1: Diagrama de Classes Simplificado.



Classes Abstratas

As classes abstratas são um tipo de classes que agem como expressões de conceitos gerais das quais classes mais específicas podem ser derivadas. Não está permitido instanciar objetos de classes abstratas. Uma classe derivada de uma classe abstrata deve implementar o método virtual puro ou também será considerada uma classe abstrata.
Métodos virtuais puros são declarados da seguinte forma:

- virtual void pureVirtualFunction() = 0

A classe **Unit** é uma classe abstrata já que define metodos virtuáis puros que são implementados nas classes derivadas correspondentes(AttackUnit e DefenceUnit).

Herança

Herança é fundamental na programação orientada a objetos, já que fornece meios para promover a extensibilidade do codigo, reutilização e maior coerência lógica no modelo de implementação. As classes que são usadas para derivação são chamadas de classes base de uma classe derivada específica.Na herança a classe derivada contém os membros da classe base mais os novos mebros que sejam adicionados na declaração da classe derivada. Herança é declarada da seguinte forma:

- class Derived :[virtual] [access-specifier] Base {

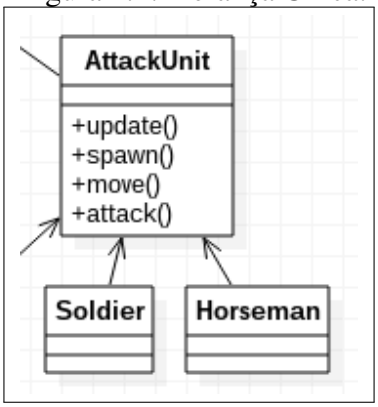
    // Member List

};

Herança Unica

As classes tem apenas uma classe base, gerando assim uma árvore de derivação.

Figura 4.2: Herança Unica.

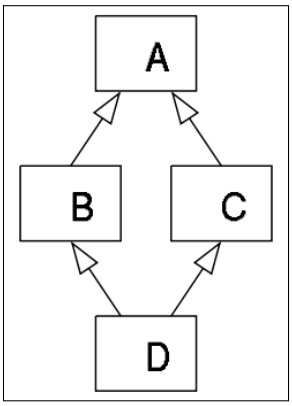


Herança Multipla

As classes podem herdar de mais de uma classe base, gerando assim um grafo de derivação. Herança multipla apresenta diversos problemas de implementação:

- Problema do Diamante: Uma classe D herda de duas classes B e C. B e C herdam de A. Os atributos de A serão duplicados em D gerando assim conflitos e ambiguidades no momento do acesso.

Figura 4.3: Problema do Diamante.



Uma solução para este problema é utilizar herança virtual nas classes B e C.

A planificação inicial foi feita de modo a evitar o uso de Herança Multipla.

Encapsulamento

O encapsulamento consiste em isolar atributos e métodos afim de progê-los de qualquer uso indevido, isto, junto com a devida documentação, favorece em muito a reusabilidade do código, uma vez que facilita sua correta instanciação.

Na nossa implementação, fizemos com que quase, senão todos, os atributos das classes fossem protegidos, isto permite que eventuais classes que herdem destas ainda consigam utilizá-los. Aliando isso aos métodos de acesso (*getters* e *setters*), temos uma camada que torna mais difícil o assinalamento de dados inválidos ou inesperados a variáveis internas, assim tornando o código mais robusto e seguro.

No nosso código, o trecho onde essa propriede se torna mais visível é na classe *Unit*, a qual possui dados sobre todas as unidades de ataque e de defesa presentes no jogo. Aqui, todas as variáveis são protegidas para que nenhum contexto de fora possa fazer alterações diretas. Isto se tornou importante não só para manter a integridade dos dados, mas também porque todos os atributos de ataque de defesa devem se manter consistentes com seus níveis já que, apenas assim, o sistema de *upgrades* pode funcionar corretamente. Portanto, todas as alterações a esses atributos só podem ser feitas por meio dos métodos que levam em consideranção os seus valores base e seus níveis atuais.

Delegates

Delegates consistem em uma maneira de generalizar tarefas que possuem diferentes rotinas e/ou entradas para um mesmo fim. No nosso código, encontramos dificuldades em implementar Delegates como normalmente são feitos em C++, com classes ou structs específicas para este fim. Portanto, tentamos utilizar caracteríscas desse modelo em algumas funções específicas. Os melhores exemplos disso a serem citados na nossa implementação desta etapa do trabalho estão na classe *Unit* e dizem respeito ao sistema de *upgrades*.

Esse sistema necessita de três informações básicas: o nível de um atributo, o seu valor base (inicial) e um coeficiente de aumento. Então juntamos estas três informações em três chamadas que realizam as seguintes operações sobre todos os atributos: consultar um valor, verificar e incrementar seu nível atual. Estas funções possuem em comum um parâmetro que consiste em uma *enum* que define o nome de um dado (vida, armadura, dano de ataque, número de alvos, velocidade de ataque e alcance de ataque). Desta forma, cada função pode realizar seis ações diferentes, sendo uma para cada atributo.

Análise Crítica

Critérios	Nota	Justificativas
Simplicidade	6	Pode se tornar bastante complexa uma vez que é necessário administrar ponteiros para listas e alocação de memória, como foi feito na classe Game para controlar todos os inimigos presentes no jogo.
Ortogonalidade	6	Não permite muita extensão das funcionalidades dos operadores e tipos básicos, assim sendo necessárias diversas funções aparentemente básicas através de includes, como a biblioteca string.h.
Expressividade	5	Por ser mais baixo nível, são necessárias mais linhas de código para fazer menos, como exemplo podemos citar as várias linhas de código utilizadas para manejar as listas de unidades mostradas na tela bem como os vários cálculos utilizados para mover unidades.
Adequabilidade	9	
Variedade de estruturas de controle	10	Tem todas as estruturas de controle necessárias presentes nas linguagens de programação atuais.
Mecanismos de definição de tipos	10	Permite a declaração de structs, enums bem como a redefinição de nomes com typedefs. Esses mecanismos foram muito utilizados na classe Unit, com a definição de enums para todos os tipos de unidades, por exemplo.
Suporte a abstração de dados e de processos	10	Permite a criação de classes, que, por si só, permitem um nível muito abrangente de abstração de dados.
Modelo de tipos	10	Possui um rígido controle de uso dos diferentes tipos disponibilizados, o que torna o desenvolvimento mais trabalhoso porém compensa uma vez que não dá muita margem de erro para o programador, tornando o código mais robusto.

Critérios	Nota	Justificativas
Portabilidade	10	É uma linguagem muito portável, possui compiladores para uma vasta quantidade de arquiteturas de processadores e sistemas operacionais.
Reusabilidade	10	É bastante reusável pois permite fácil importação das mais variadas bibliotecas.
Suporte e documentação	10	Microsoft oferece uma ampla documentação, rica em exemplos e conteúdo confiável, fora toda a documentação produzida por usuários em fórums na internet.
Tamanho de código	7	Permite o uso de templates para a criação de algoritmos e estruturas genéricas. Facilita a programação, mas o codigo é replicado quando passa pelo compilador.
Generalidade	7	a
Eficiência e custo	10	A diferença da maioria das linguaguens Orientadas a objetos, é uma linguagem compilada, o que outorga um alto desempenho, também permite utilizar vinculação tardia para permitir polimorfismo.

REFERÊNCIAS

CERUZZI, P. **A History of Modern Computing (Cambridge, MA & London**. [S.l.]: MIT Press, 1998.

ORGANICK, E. I.; FORSYTHE, A. I.; PLUMMER, R. P. **Programming language structures**. [S.l.]: Academic Press, 2014.