Foi escolhida a linguagem C++ para a implementação do jogo de Pokemon, devido à minha familiaridade com a linguagem e seus processos de compilação, além de ser uma escolha fácil para ambientes Linux.

Utilizou-se a IDE Visual Studio Code para o desenvolvimento e depuração. As dependências de pacotes de software, para se executar numa distribuição Linux Ubuntu, versão 18, estão no arquivo **Ubuntu-18-Dependencies-CLANG.sh** (para o compilador C++ Clang) e **Ubuntu-18-Dependencies-CLANG.sh** (para o compilador C++ GCC). A compilação, execução da aplicação e execução dos testes unitários podem ser feitas em linha de comando, sem a necessidade de uma IDE:

mkdir build/
cd build/
cmake -DCMAKE_BUILD_TYPE=Debug ../.
cmake --build .

Para executar:
cd src/Pokemon
./Pokemon

Para executar os testes unitários:
cd -
ctest -V

A aplicação é componentizada, da seguinte forma, com os comentários abaixo como justificativas:

Pokemon
É o binário (executável) que representa a aplicação.

PokemonBLL
É a biblioteca (executável de vínculo dinâmico) que contém toda a lógica da aplicação (Business Layer Logic), de forma a desacoplar a implementação da interface do programa. É também uma estratégia vital para se implementar os testes unitários.

Em termos de criação de classes, cada uma foi criada com seu par declaração (ou cabeçalho) .hpp e implementação .cpp. As classes descritas abaixo referem-se somente ao módulo **PokemonBLL**, pois o módulo Pokemon não tem classes, apenas coleta as escolhas das cartas das rodadas e mostra o resultado.

Jogo – encapsula as rodadas de uma partida, que compõem o jogo.
Pokemon – encapsula as propriedades das cartas Pokemon.
Pokemongenerator – encapsula o mecanismo que ‘embaralha’ as cartas e cria as rodadas para alimentarem o jogo.
Pokemonrodada – encapsula as escolhas de cartas dos dois jogadores, e seu resultado.

Dadas as condições do projeto, não se viu necessidade de usar sobrecarga de método ou derivadas de classes a partir de uma superclasse. Os 16 tipos de cartas, com suas propriedade, viraram 16 estruturas (C++ strut) que alimentam a classe Pokemon. Dessa forma, um ‘baralho’ foi criado na forma de variáveis constantes estáticas, facilitando a geração de cartas aleatórias na hora de se criar um jogo.

C++, ao contrário de Java, tem na própria definição da linguagem as bibliotecas de algoritmos (STL – Standard Template Library), com facilidades como algoritmos e coleções nativas na linguagem. Tudo que é necessário fazer é apenas declarar seus cabeçalhos. Os mais relevantes são:

std::vector (coleção)
std::algorithm (vários algoritmos de busca, ordenação e outros)
std::string (manipulação de caracteres)

Entendimento e facilidade de leitura da aplicação utilizando Testes Unitários

TDD (*Test Driven Development*) é a metodologia *de facto* hoje para desenvolvimento no mercado de trabalho. Dentre as diversas vantagens da metodologia, como facilidade de prototipação e antecipação do desenvolvimento a partir das regras de negócio, se destaca também a facilidade de introduzir novos desenvolvedores às regras e conceitos da aplicação, facilitando a leitura intuitiva do código a partir da forma como ele deve funcionar.

Exemplo de clareza na descrição de um teste. Abaixo, testamos de forma individual o ciclo de efetividade, como descrito no enunciado do projeto:

```
BOOST_AUTO_TEST_CASE(test_pokemon_vantagem_electric_over_water)
{
    BOOST_TEST_MESSAGE( "TEST CENÁRIO VANTAGEM. Electric sobre Water." );
    Pokemon jogadorelectric = Pokemon(BLL::g_PikachuSt, Status::Ativo, "", Caracteristica::Agilidade);
    Pokemon jogadorwater = Pokemon(BLL::g_TotodileSt, Status::Ativo, "", Caracteristica::Agilidade);
    BOOST_TEST ( jogadorelectric.temvantagem(jogadorwater) );
}
```

Exemplo de saída dos testes unitários:

```
┌─mpeschke@Inspiron-7559 ~/src/OOP-PROJ1/build <main*>
└─$ ctest -V
UpdateCTestConfiguration  from :/home/mpeschke/src/OOP-PROJ1/build/DartConfiguration.tcl
UpdateCTestConfiguration  from :/home/mpeschke/src/OOP-PROJ1/build/DartConfiguration.tcl
Test project /home/mpeschke/src/OOP-PROJ1/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 1
    Start 1: unittests

1: Test command: /home/mpeschke/src/OOP-PROJ1/build/tests/PokemonBLL-Unit-Tests "--log_level=message"
1: Test timeout computed to be: 10000000
1: Running 4 test cases...
1: TEST CENÁRIO VANTAGEM. Electric sobre Water.
1: TEST CENÁRIO VANTAGEM. Water sobre Fire.
1: TEST CENÁRIO VANTAGEM. Fire sobre Grass.
1: TEST CENÁRIO VANTAGEM. Grass sobre Electric.
1:
1: *** No errors detected
1:
1/1 Test #1: unittests .......................   Passed    0.01 sec

100% tests passed, 0 tests failed out of 1

Total Test time (real) =   0.01 sec
```