

Kode Kelompok : LLH

Nama Kelompok : LeLaH

1. 13521062 / Go Dillon Audris

2. 13521084 / Austin Gabriel Pardosi

3. 13521108 / Michael Leon Putra Widhi

4. 13521160 / M. Dimas Sakti Widyatmaja

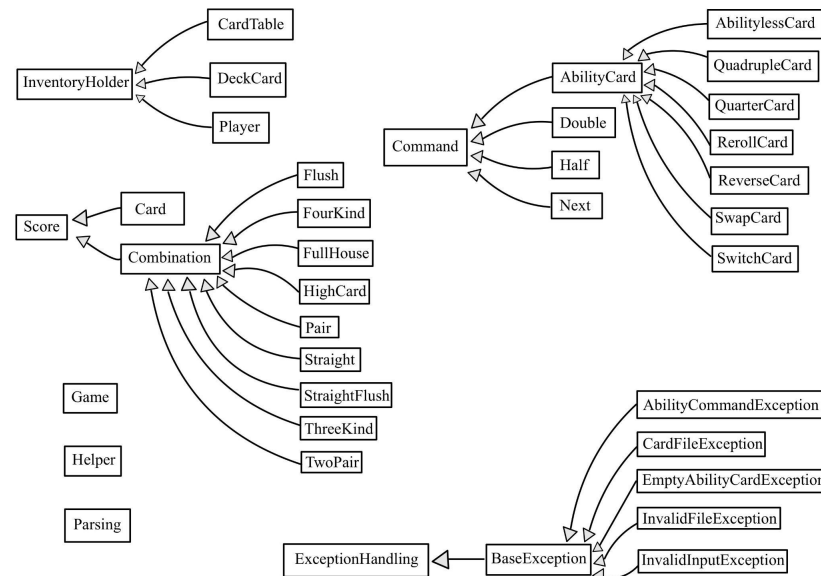
5. 13521172 / Nathan Tenka

Asisten Pembimbing : Widya Anugrah Putra

Lampiran : <https://github.com/AustinPardosi/tubes-oop1>

1. Diagram Kelas

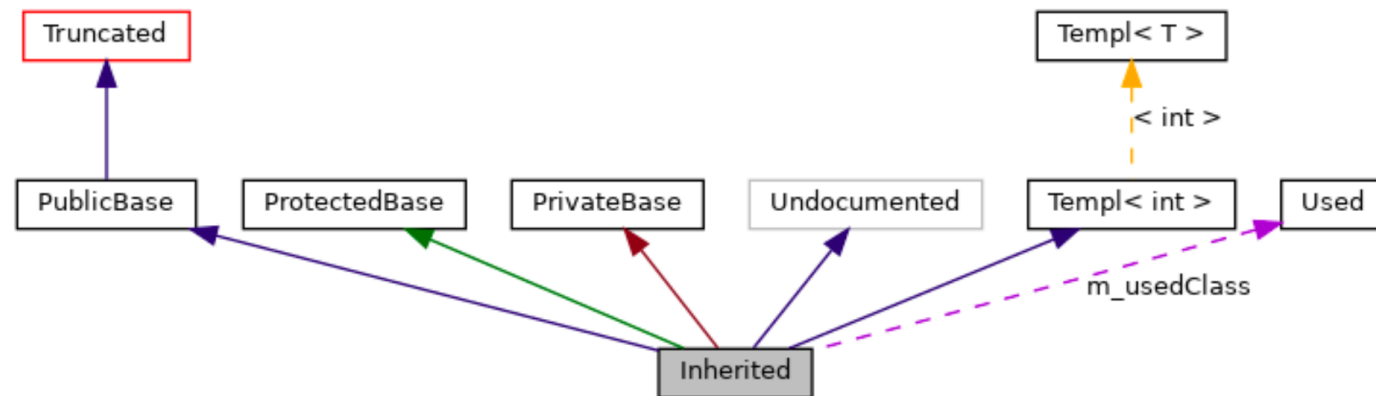
1.1. Class Hierarchy



Gambar 1.1. Skema Class Hierarchy

1.2. Class Diagram

1.2.1. Legenda Class Diagram



Gambar 1.2.1. Legenda Class Diagram

Kotak-kotak pada grafik di atas memiliki arti sebagai berikut:

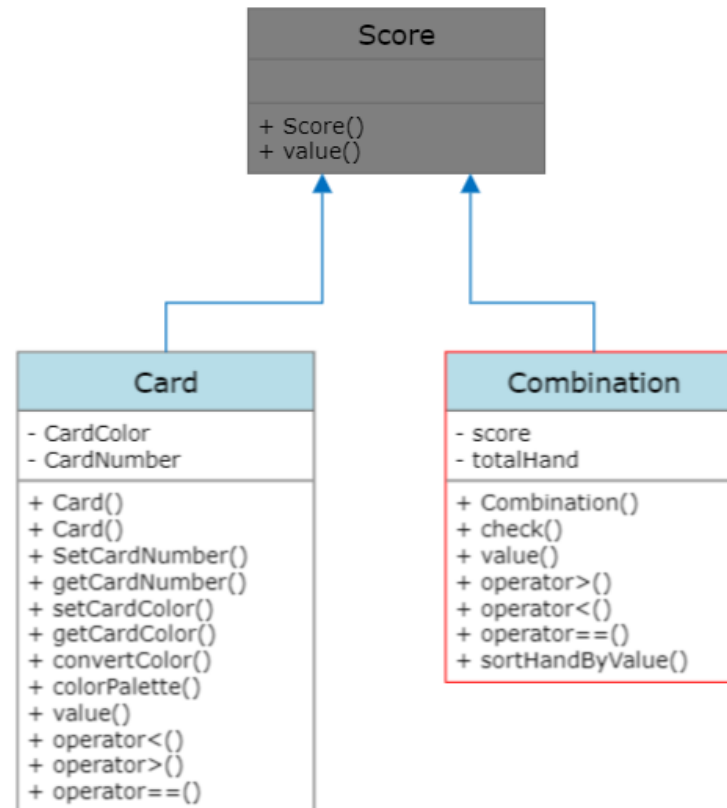
- Kotak abu-abu yang diisi (gelap) mewakili kelas tempat *Class Diagram* dibuat.
- Kotak dengan batas tepi berwarna hitam menunjukkan kelas yang terdokumentasi.
- Kotak dengan batas tepi berwarna abu-abu menunjukkan kelas yang tidak berdokumentasi.
- Kotak dengan batas tepi berwarna merah menunjukkan kelas terdokumentasi yang tidak semua hubungan *inheritance* ditampilkan. Diagram dipotong jika tidak sesuai dengan batas yang ditentukan.

Panah memiliki arti sebagai berikut:

- Panah biru tua digunakan untuk memvisualisasikan hubungan *inheritance public* antara dua kelas.
- Panah hijau tua digunakan untuk hubungan *inheritance protected*.

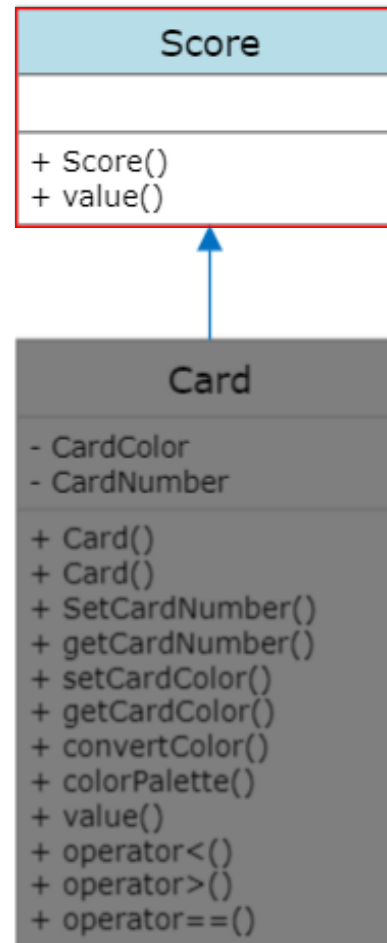
- c. Panah merah tua digunakan untuk hubungan *inheritance private*.
- d. Panah putus-putus berwarna ungu digunakan jika sebuah kelas digunakan oleh kelas lain. Panah diberi label dengan variabel yang melaluinya kelas yang dituju dapat diakses.
- e. Panah putus-putus berwarna kuning menunjukkan hubungan antara *instance template* dan kelas *template* tempat *instance* dibuat. Panah diberi label dengan parameter *template* dari *instance*.

1.2.2. Class Diagram Score



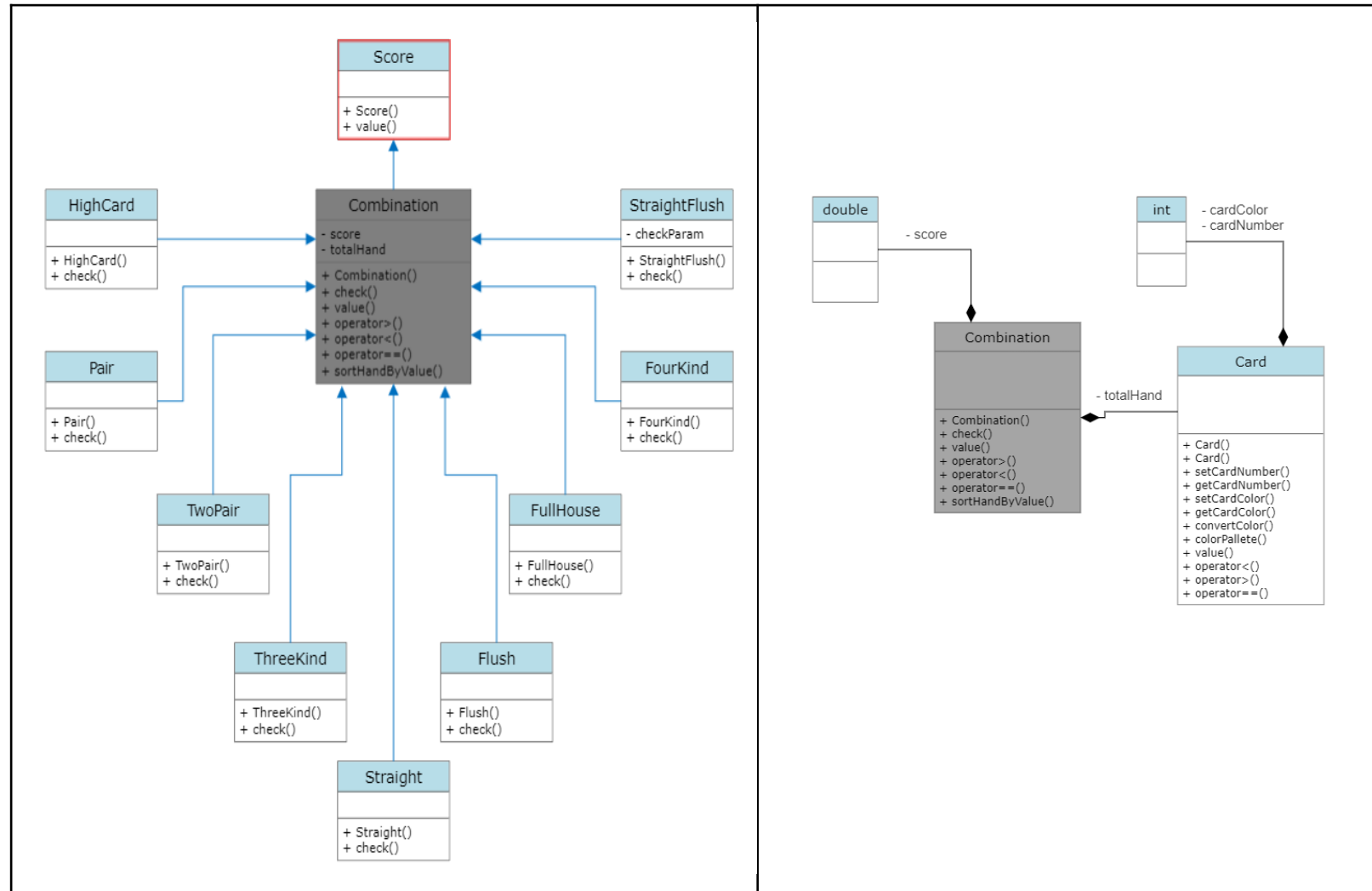
Gambar 1.2.2. Inheritance Diagram Kelas Score

1.2.3. Class Diagram Card



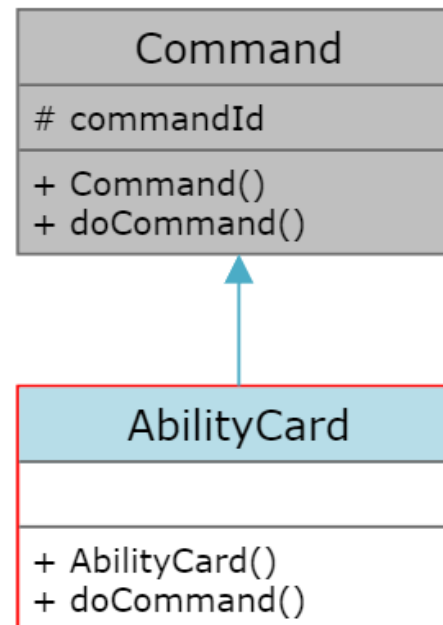
Gambar 1.2.3. *Inheritance Diagram* Kelas Card

1.2.4. Class Diagram Combination



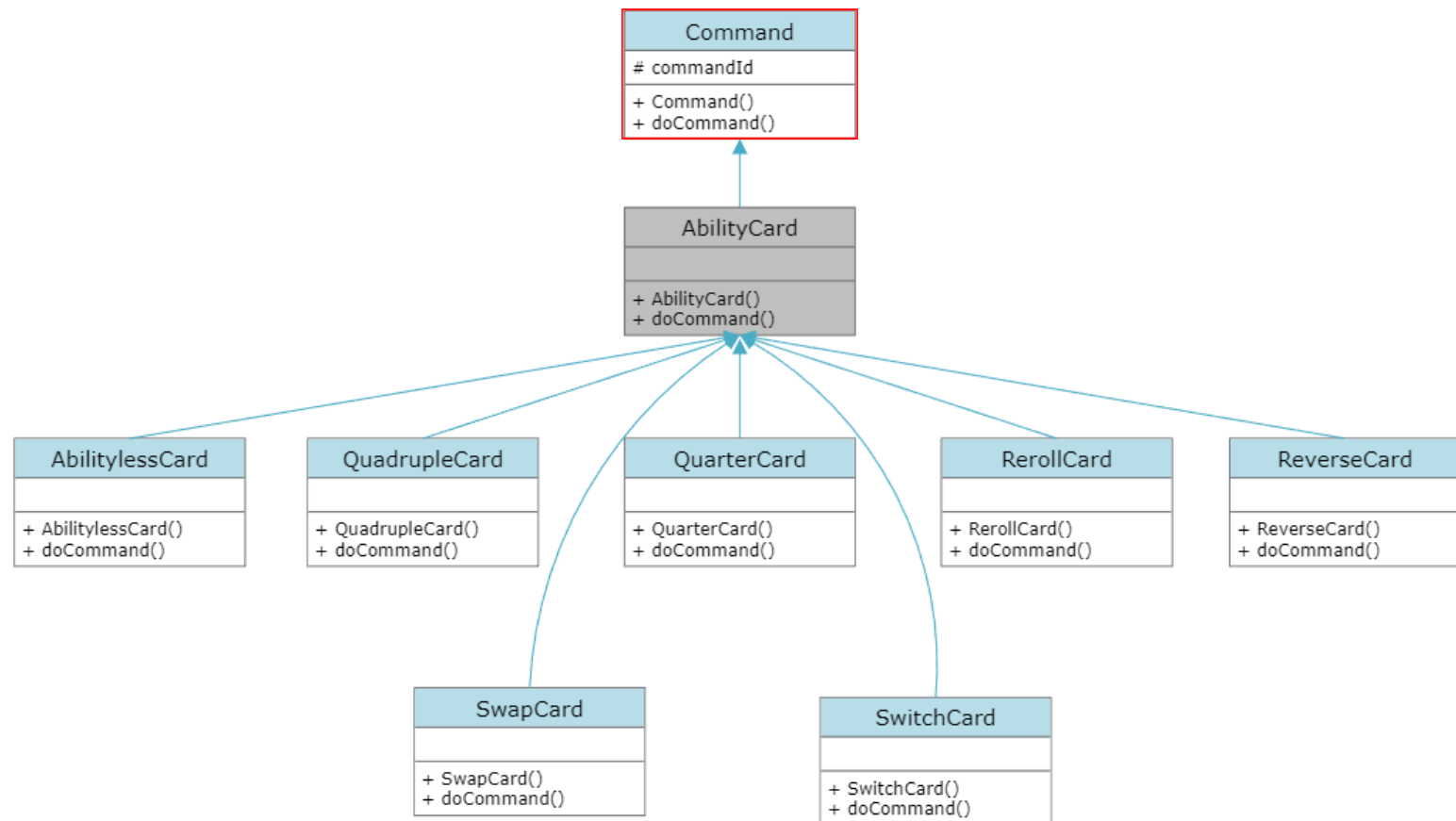
Gambar 1.2.4. Inheritance Diagram (kiri) dan Collaboration Diagram (kanan) Kelas Combination

1.2.5. Class Diagram Command



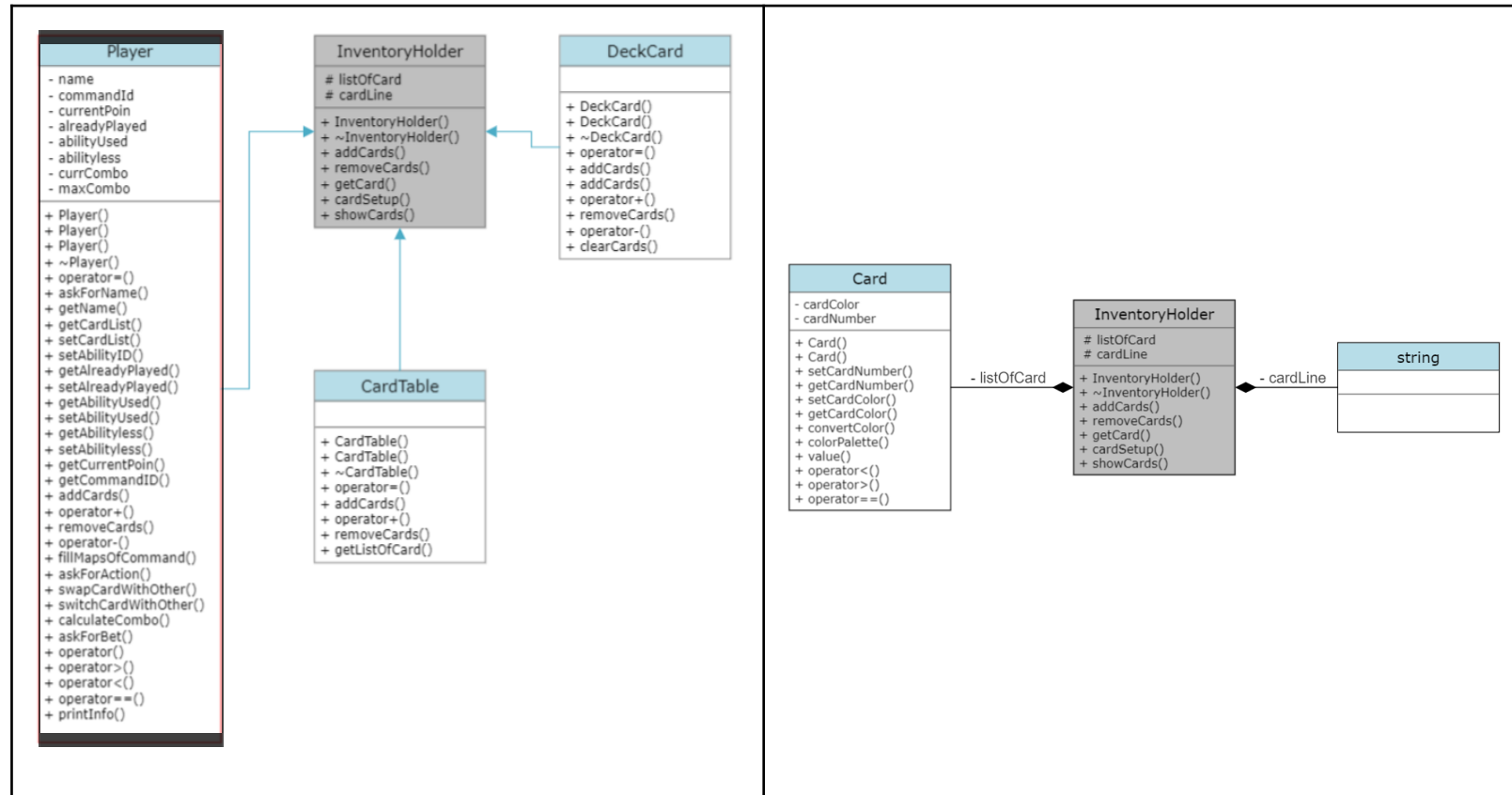
Gambar 1.2.5. *Inheritance Diagram* Kelas Command

1.2.6. Class Diagram AbilityCard



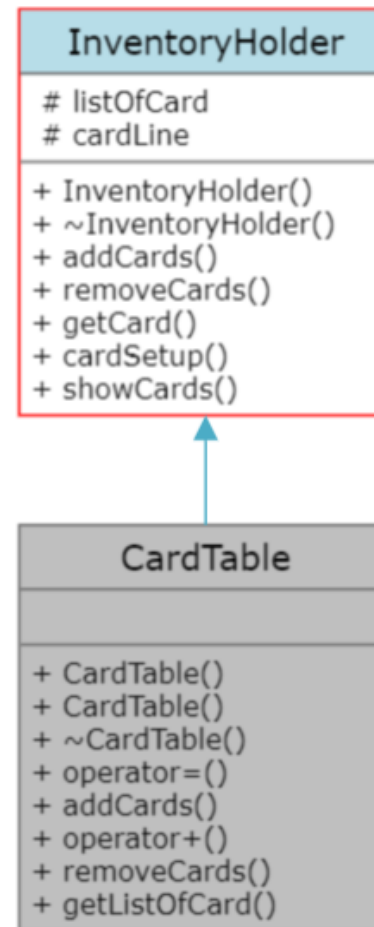
Gambar 1.2.6. Inheritance Diagram Kelas AbilityCard

1.2.7. Class Diagram InventoryHolder



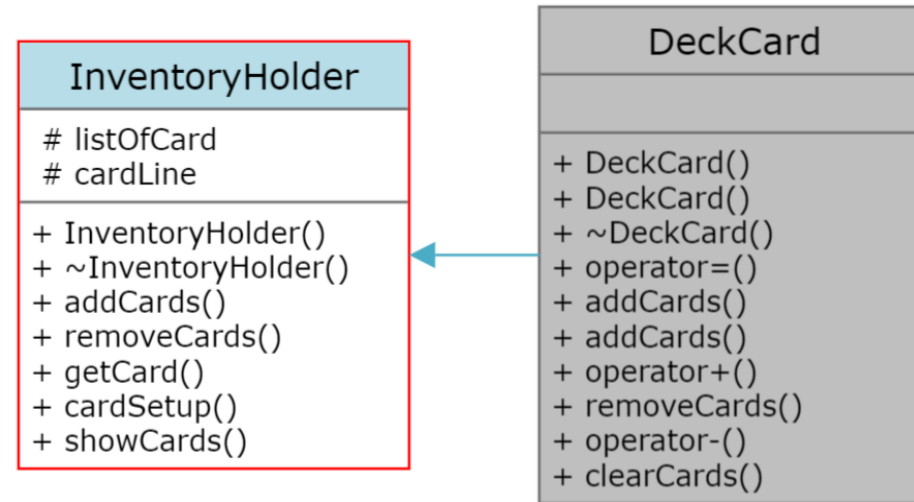
Gambar 1.2.7. Inheritance Diagram (kiri) dan Collaboration Diagram (kanan) Kelas InventoryHolder

1.2.8. Class Diagram CardTable



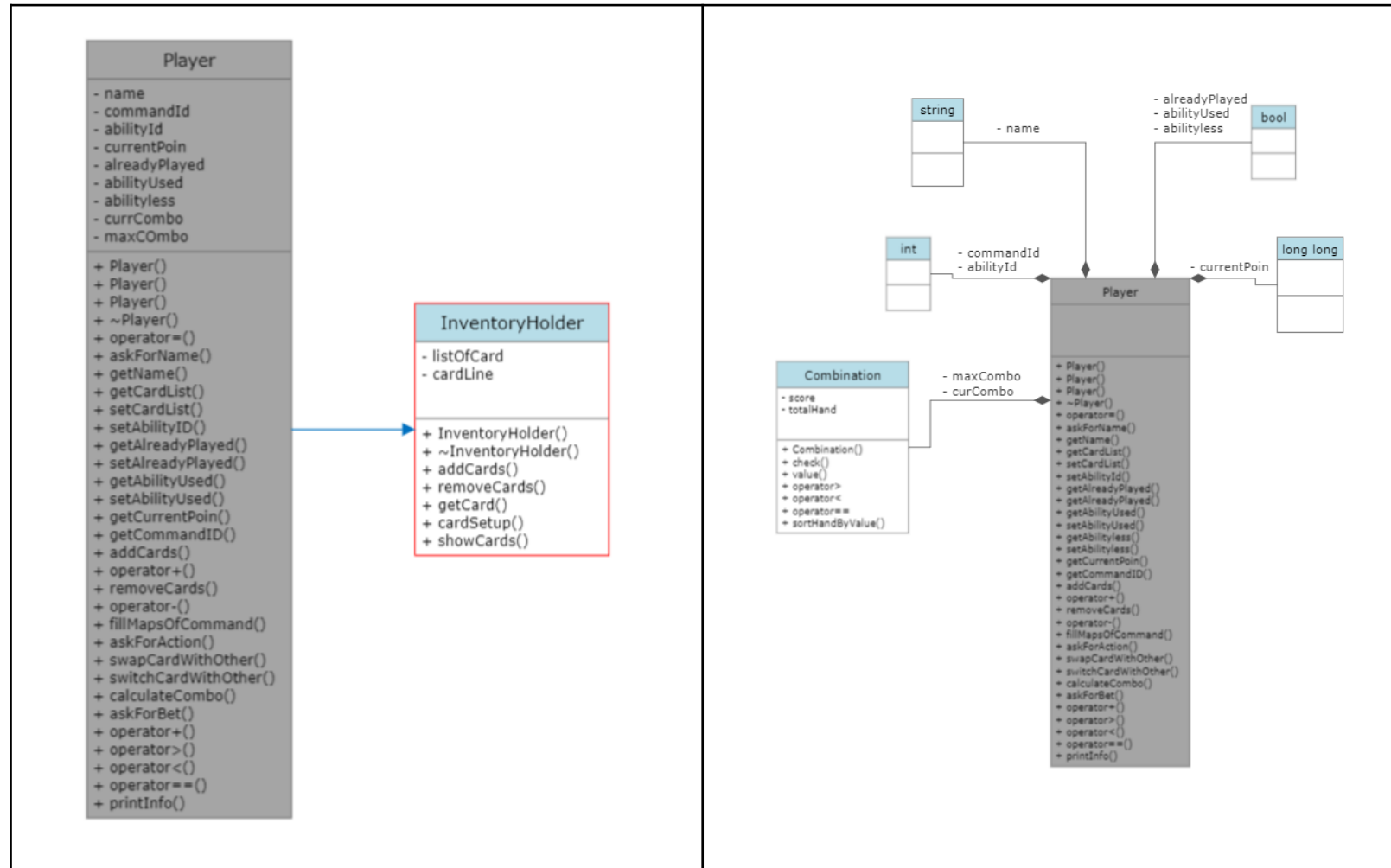
Gambar 1.2.8. *Inheritance Diagram Kelas CardTable*

1.2.9. Class Diagram DeckCard



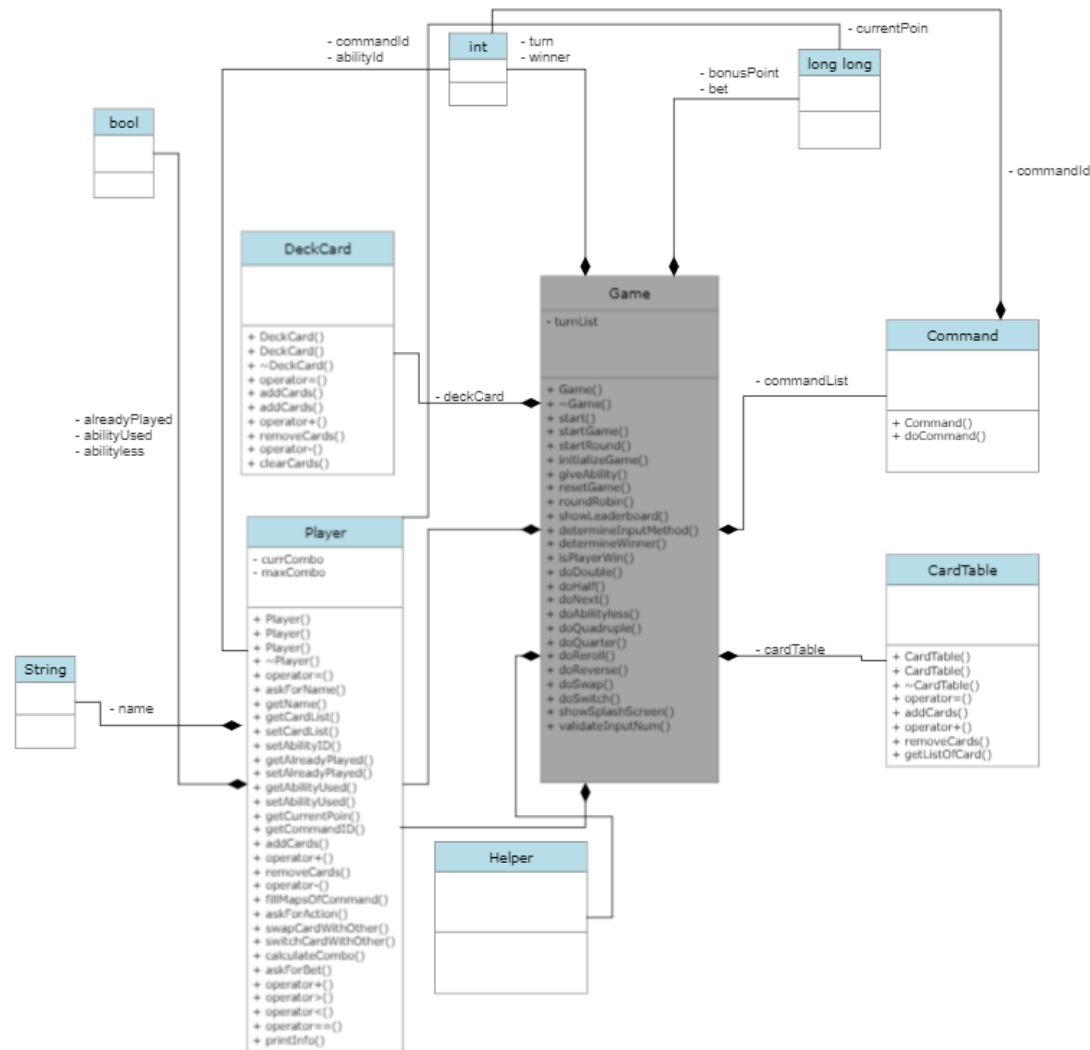
Gambar 1.2.4. *Inheritance Diagram Kelas DeckCard*

1.2.10. Class Diagram Player



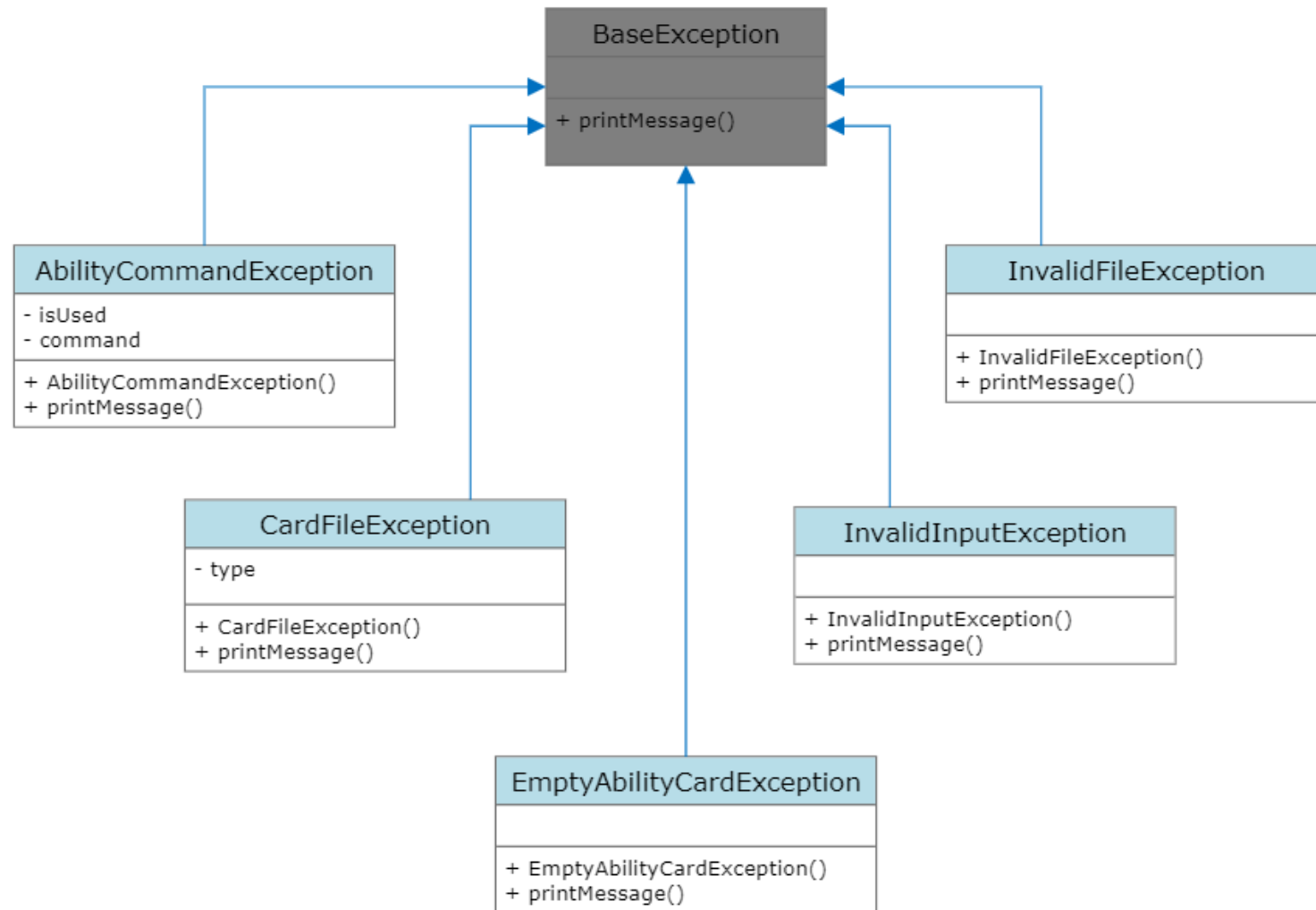
Gambar 1.2.10. Inheritance Diagram (kiri) dan Collaboration Diagram (kanan) Kelas Player

1.2.11. Class Diagram Game



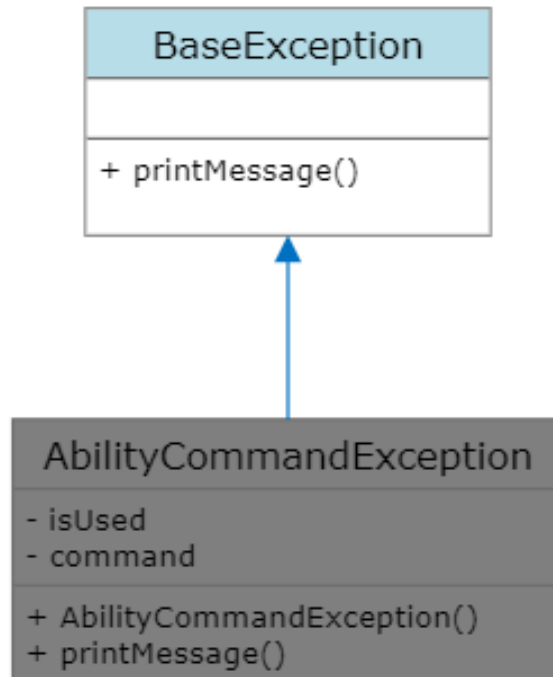
Gambar 1.2.11. Collaboration Diagram Kelas BaseException

1.2.12. Class Diagram BaseException



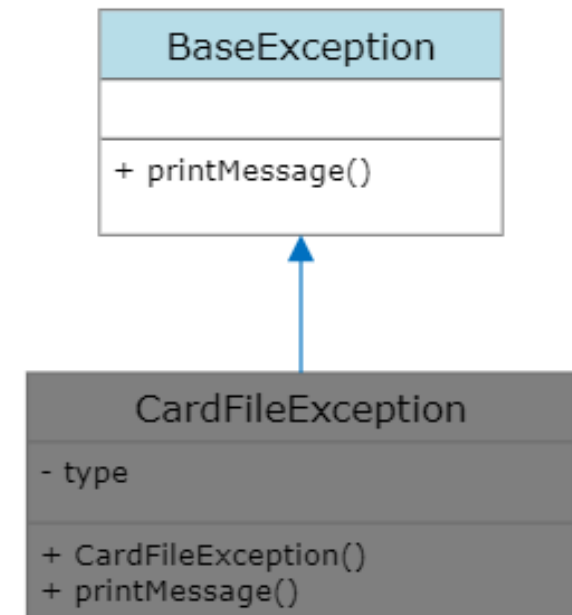
Gambar 1.2.12. Inheritance Diagram Kelas BaseException

1.2.13. Class Diagram AbilityCommandException



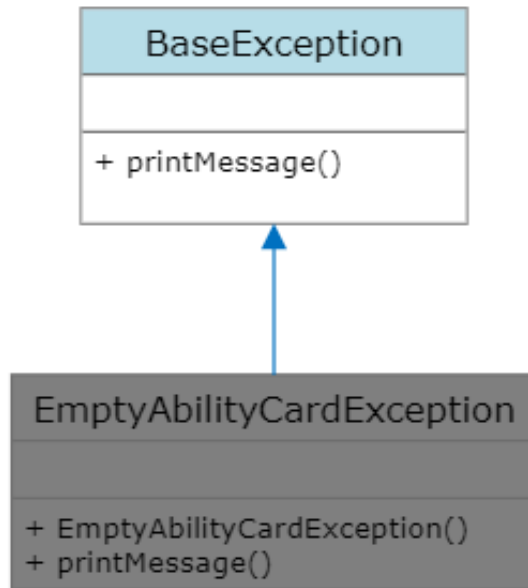
Gambar 1.2.13. *Inheritance Diagram* Kelas
AbilityCommandException

1.2.14. Class Diagram CardFileException



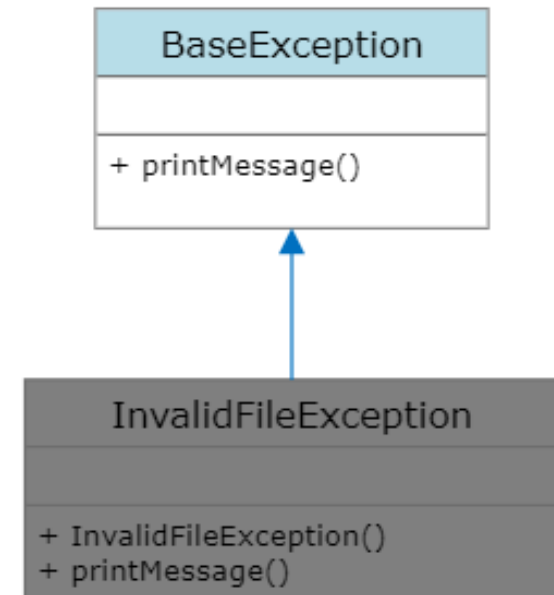
Gambar 1.2.14. *Inheritance Diagram* Kelas
CardFileException

1.2.15. Class Diagram EmptyAbilityCardException



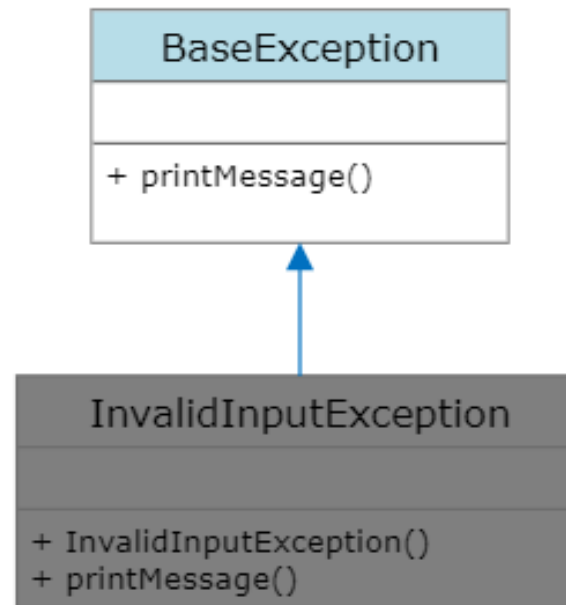
Gambar 1.2.15. *Inheritance Diagram* Kelas
EmptyAbilityCardException

1.2.16. Class Diagram InvalidFileException



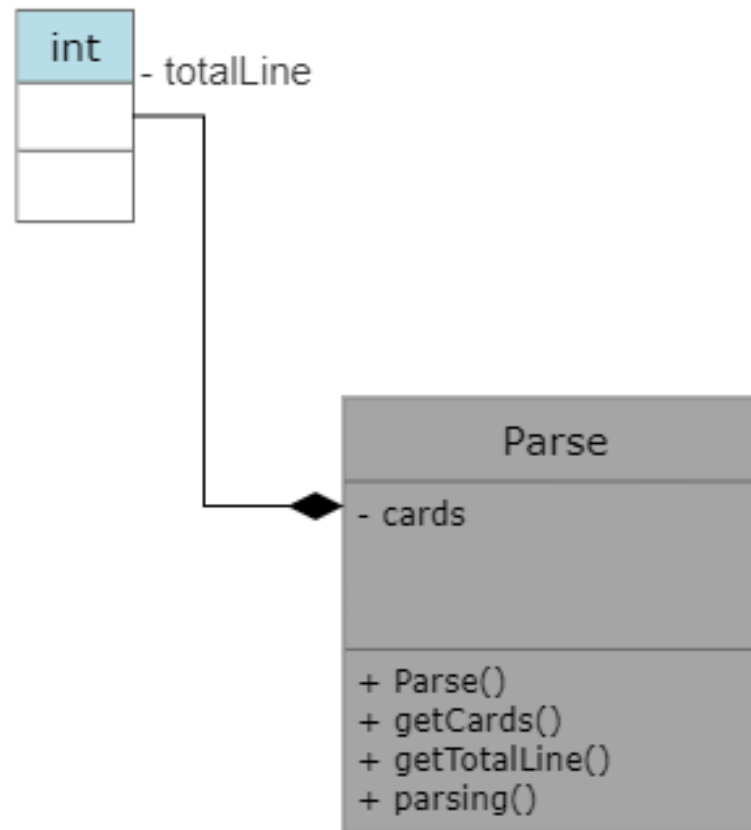
Gambar 1.2.16. *Inheritance Diagram* Kelas
InvalidFileException

1.2.17. Class Diagram InvalidInputException



Gambar 1.2.17. *Inheritance Diagram* Kelas InvalidInputException

1.2.18. Class Diagram Parse



Gambar 1.2.18. Collaboration Diagram Kelas Parse

1.3. Class Design

1.3.1. Command

Kelas Command banyak menggunakan konsep *inheritance* dan *polymorphism* dalam proses implementasinya. *Inheritance* digunakan untuk melakukan pemilihan *command* secara spesifik, baik itu daftar perintah yang bisa dijalankan oleh masing-masing pemain, maupun AbilityCard yang bisa dimiliki oleh setiap pemain. Adapun konsep dari *polymorphism* digunakan sebagai generalisasi tipe objek pada kelas Command, khususnya untuk *method* doCommand sehingga *method* ini dapat berperilaku baik pada masing-masing *command* maupun AbilityCard yang melakukan *inherit* pada kelas ini.

Kelebihan dari desain kelas ini adalah kelas ini mudah digunakan dimanapun karena memiliki struktur yang jelas. Tidak perlu dilakukan banyak instansiasi objek untuk setiap kelas yang melakukan *inherit* pada kelas Command. Kekurangan dari desain kelas ini yaitu pengimplementasian *method* yang bergantung pada kelas Game sehingga realisasi *method* dari kelas ini adalah dengan melakukan *invocation method* yang sesuai kepada kelas Game.

1.3.2. InventoryHolder

Kelas InventoryHolder menggunakan konsep *inheritance* dan C++ STL dalam implementasinya. Kelas ini memiliki tanggung jawab untuk mengatur dan mengelola kumpulan objek Card. Konsep *inheritance* digunakan karena kelas ini masih terlalu abstrak untuk menjalankan metode memasukkan dan mengeluarkan objek Card. Tugas memasukkan dan mengeluarkan objek Card harus dilakukan oleh kelas *child* dari InventoryHolder sesuai dengan aturan pemasukan dan pengeluaran masing-masing kelas *child*. Kelas *child* dari InventoryHolder yang juga harus mengelola kumpulan objek Card juga menjadi alasan lain digunakannya konsep *inheritance*. Sementara itu, C++ STL digunakan dengan menggunakan *vector of Card* sebagai atribut dari kelas InventoryHolder. Kelebihan dari desain kelas ini adalah *re-usability*, dimana kelas yang sama dapat diwariskan kepada kelas-kelas lain yang memiliki karakteristik yang serupa. Implementasi *method* pada kelas ini (selain *method* abstrak) juga mengurangi *code redundancy* dan meningkatkan *readability* kode. Kekurangan dari desain

kelas ini adalah penggunaan *operator overloading* berupa + dan - yang mengembalikan kelas InventoryHolder menjadi tidak mungkin dilakukan karena kelas InventoryHolder tidak dapat diinstansiasi.

Kelas DeckCard dan CardTable, yang menjadi *child* dari kelas InventoryHolder, memanfaatkan konsep *operator overloading*, *method overloading*, dan C++ STL dalam implementasinya. Masing-masing bertanggung jawab untuk mengelola kumpulan objek *Card* di tumpukan dan di meja. Konsep *operator overloading* berupa + dan - digunakan di kedua kelas untuk menambah *Card* atau mengurangi jumlah *Card* yang disimpan. Konsep *method overloading* dimanfaatkan oleh kelas DeckCard untuk memungkinkan pembangkitan objek *Card* secara acak atau melalui pembacaan file namun dengan nama *method* yang sama. C++ STL berupa *map* dan *pair* juga digunakan oleh kelas DeckCard untuk memetakan pasangan *int*, *Card* dimana *int* merujuk pada kode angka yang khusus untuk suatu *Card*. Desain kedua kelas ini memberikan perbedaan objek yang baik mengingat keduanya merupakan objek yang akan berada di dalam permainan. Bisa dibayangkan jika dua objek yang berbeda dalam permainan ini hanya dibuat dengan menggunakan kelas InventoryHolder yang sama.

Kelas Player juga merupakan salah satu kelas *child* dari kelas InventoryHolder. Kelas ini memiliki tanggung jawab yang lebih kompleks dari dua kelas sebelumnya dan merepresentasikan pemain dalam permainan. Kelas ini dapat memberikan *command* untuk dijalankan, mendapatkan poin, mendapatkan objek Card, dan lain-lain. Kelas ini memanfaatkan konsep *polymorphism*, *operator overloading*, C++ STL, dan *template function* dalam implementasinya. Konsep *polymorphism* dimanfaatkan untuk menghitung nilai dari setiap kombinasi yang mungkin dimiliki oleh pemain. Mengingat banyaknya tipe kombinasi yang dapat dilakukan (dari *high card* hingga *straight flush*), akan lebih mudah untuk menyimpan *base class* dari kombinasi saja. *Operator overloading* juga dimanfaatkan pada kelas ini dan terdiri atas + (menambah Card atau poin), operator - (mengurangi jumlah Card), serta >, <, dan == (untuk membandingkan kombinasi tertinggi antara pemain). Kelas Player menggunakan C++ STL berupa *map*, *pair*, dan *vector*. *Map* dan *pair* digunakan untuk memetakan perintah *command* pemain yang berupa *string* menjadi suatu *integer* kode *command*. Terakhir, *template function* digunakan untuk mencari kombinasi dengan nilai tertinggi dari kumpulan kombinasi yang dapat dilakukan pemain. Salah satu kelebihan dari desain kelas ini adalah memungkinkan terpisahnya tanggung jawab antara *game* dan *player*, dimana objek *game* tidak langsung bertanya kepada *user* mengenai *command* yang harus dilakukan, namun meminta objek *player* untuk menanyakan hal tersebut. Kekurangan desain dari ini adalah

dibutuhkan pengecekan nilai kombinasi satu per satu untuk tiap tipe, meskipun hal ini dapat dilakukan dengan menimpa nilai kombinasi sebelumnya, jika terdapat kombinasi yang mungkin pada tipe yang lebih tinggi. Kekurangan lainnya adalah implementasi dari kelas ini berkembang dan menjadi semakin kompleks seiring dengan bertambahnya fitur pada kelas Game. Hal ini terjadi karena kelas Player menjadi komposisi dari kelas Game.

1.3.3. Score

Kelas Score menggunakan konsep *inheritance* dalam implementasinya. *Inheritance* digunakan untuk mendapat nilai dari sebuah kartu atau kombinasi. Kelebihan dari desain kelas ini adalah tinggal digunakan *method value()* untuk mendapat nilai suatu kartu maupun kombinasi.

Kelas Combination, sebagai *child* dari kelas Score, juga banyak menggunakan konsep *inheritance*, *polymorphism*, dan *operator overloading* dalam implementasinya. *Inheritance* digunakan untuk memeriksa ada/tidaknya masing-masing jenis kombinasi kartu serta mendapat nilainya, sedangkan *polymorphism* digunakan untuk generalisasi tipe objek kelas Combination. Hal ini terlihat terutama pada kelas Player, tepatnya di *method calculateCombo* dan atribut *currCombo* serta *maxCombo*. Dengan demikian, tidak perlu dibuat *vector* maupun *maxCombo* yang berbeda-beda untuk setiap jenis kombinasi kartu. *Operator overloading* dilakukan untuk mempermudah perbandingan antar kombinasi. Terdapat operator *>*, *<*, dan *==*. Perbandingan dilakukan dengan membandingkan value masing-masing kombinasi. Kelebihan dari desain kelas Combination ini adalah mudah digunakan karena strukturnya sudah jelas. Kekurangannya adalah untuk memeriksa tiap jenis kombinasi perlu dilakukan instansiasi objek untuk masing-masing jenis kombinasi.

Kelas Card, yang juga merupakan *child* dari kelas Score hanya menggunakan konsep *operator overloading* untuk mempermudah perbandingan nilai antar kartu. Terdapat operator *<*, *>*, dan *==*, sama seperti kelas Combination. Kelebihan dari desain kelas Card ini adalah membuat implementasi kelas seperti DeckCard dan TableCard menjadi lebih mudah karena hanya diperlukan *container* (seperti *vector*) yang berisi objek Card. Kekurangannya adalah atribut warna kartu yang berupa kode *integer* menyebabkan konversinya menjadi kata

warna (e.g. 3 adalah “Merah”) atau warna asli (untuk melakukan *color print*) menjadi lebih kompleks dan membutuhkan *method* tambahan.

1.3.4. Game

Kelas Game memiliki peran sebagai penyatu dari objek-objek game yang telah dibuat sebelumnya dan mengorkestra komunikasi antar seluruh objek. Kelas ini tidak mewarisi kelas manapun, namun memanfaatkan modul-modul lain yang telah dibuat. Kelas ini menerapkan konsep *polymorphism*, *exception*, dan C++ STL. *Polymorphism* dimanfaatkan dalam membuat *vector* berisi *Command** untuk menampung berbagai *command* yang dapat diperintahkan oleh *player*. Penggunaan *Command** memberikan kelebihan dimana seluruh *child* dari *command* tidak perlu dijelaskan atau di-*include* satu per satu di dalam header kelas Game. *Exception* juga dimanfaatkan pada implementasi kelas untuk melakukan validasi terhadap input angka dari *user*. Kelas ini juga memanfaatkan C++ STL berupa *vector* untuk menyimpan *turnList*, *playerList*, dan *commandList*, serta algoritma STL seperti *for_each*, *shuffle*, dan *sort*. Logika dari *game* akan dijalankan pada kelas ini. Kelas ini juga bertanggung jawab untuk mengubah objek-objek yang terdapat dalam *game* ketika *action* tertentu dijalankan seperti mengubah *bonus point*, meminta objek *player* menukar kartu, ataupun membalik urutan. Kelebihan dari adanya kelas ini adalah kemampuan orkestra dan mengorganisir objek-objek lain dengan baik, sehingga cara kerja setiap objek tidak ditunjukkan dan tidak dijadikan satu kode program yang sangat panjang. Kekurangan dari desain kelas Game adalah sulitnya mengetahui letak permasalahan ketika terjadi error saat *runtime*, karena kelas ini hanya berkomunikasi dengan objek lain, dan objek tersebut bisa saja berkomunikasi dengan objek lain lagi.

1.3.5. ExceptionHandling

Implementasi dari kelas *exception* dinyatakan dalam kelas *BaseException*. Kelas ini menggunakan konsep *inheritance* sehingga setiap kelas yang menjadi turunannya dapat diakses dari kelas *BaseException*. Adapun *child* dari kelas ini adalah kelas *AbilityCommandException*, kelas *CardFileException*, kelas *EmptyAbilityException*, kelas *InvalidFileException*, dan kelas *InvalidInputException*. Setiap kelas memiliki

sebuah *method* yang sama yaitu *printMessage* untuk melakukan pencetakan penyebab ditimbulkannya *exception* pada eksekusi program. Implementasi *method* ini menggunakan konsep *polymorphism*.

Keuntungan dari desain kelas *exception* seperti skema yang dijelaskan diatas adalah dapat menangani berbagai macam *exception* dengan menggunakan konsep *polymorphism* dari kelas induk bagi setiap kasus *exception* yang timbul. Adapun kekurangannya adalah perlu melakukan pendefinisian kelas *exception* dalam jumlah yang relatif banyak untuk setiap kasus *exception* yang mungkin. Selain itu, akan relatif sulit untuk melakukan pelacakan dimana *exception* dilempar.

1.3.6. Parsing

Kebutuhan akan menerima masukan dari *file* membuat kelas ini perlu diimplementasikan. Konsep Pemrograman Berbasis Objek yang digunakan pada kelas ini adalah STL, khususnya *vector*. Kelas ini melakukan pembacaan dari *file* yang berisi konfigurasi kartu untuk kemudian disimpan pada kelas DeckCard. Proses *parsing* dilakukan dengan menggunakan beberapa *library* seperti *fstream* dan *sstream*. Kelebihan dari adanya kelas ini adalah proses *parsing* tidak perlu dilakukan pada kelas Game sehingga kelas Game tidak perlu mengetahui bagaimana proses *parsing* dilakukan. Implementasi ini mendukung adanya konsep *information hiding* pada Pemrograman Berorientasi Objek. Selain itu, implementasi kelas ini didukung dengan adanya *method overloading* yang dimiliki oleh kelas DeckCard yang memungkinkan menerima konfigurasi kartu berdasarkan *file* masukan.

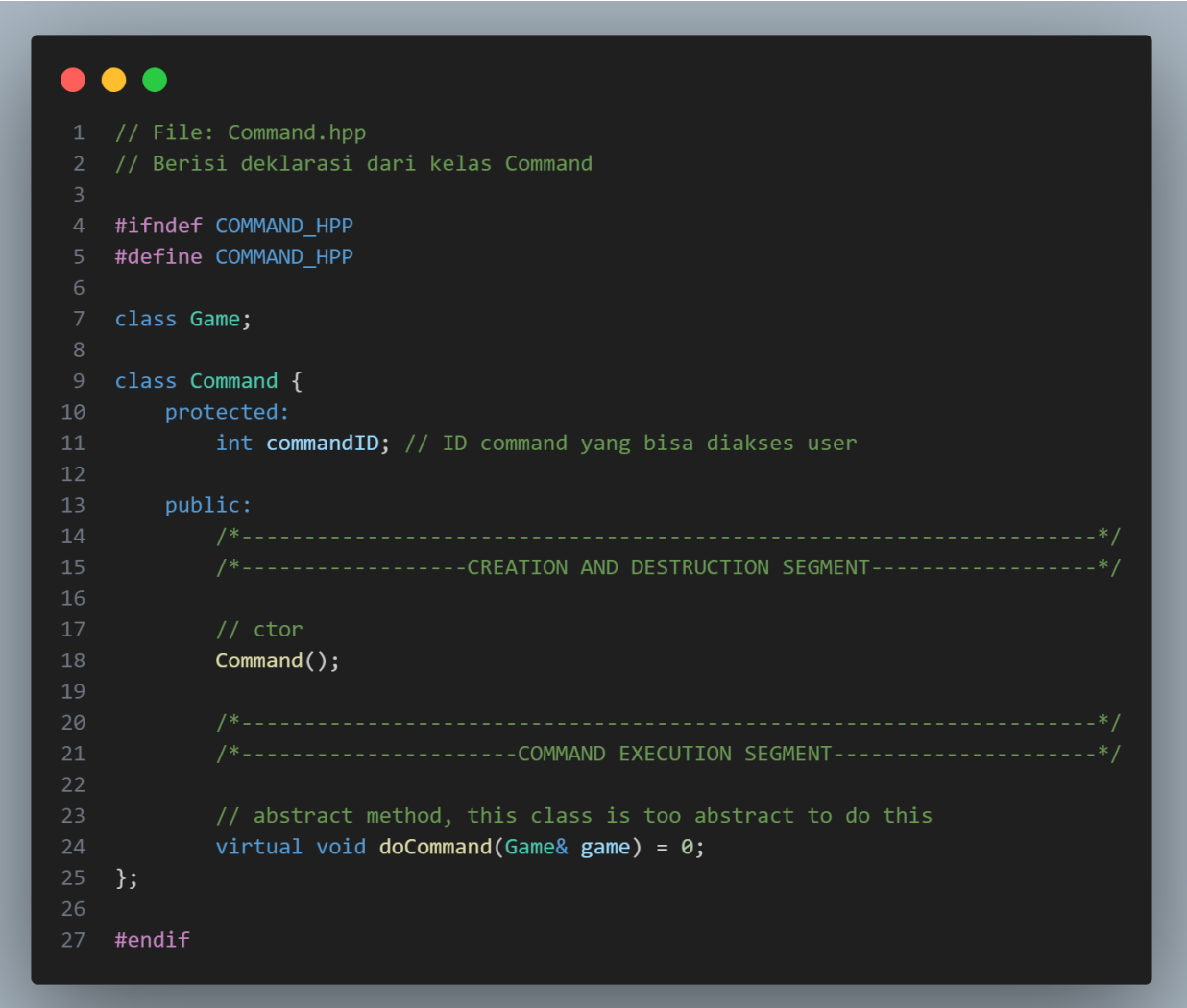
2. Penerapan Konsep OOP

2.1. Inheritance & Polymorphism

Konsep *inheritance* dan *polymorphism* digunakan oleh banyak kelas pada program. Daftar kelas yang menggunakan konsep *inheritance* dan *polymorphism* adalah :

1. *Parent* : Command
Child : Double, Half, Next, AbilityCard
2. *Parent* : AbilityCard
Child : AbilitylessCard, QuadrupleCard, QuarterCard, RerollCard, ReverseCard, SwapCard, SwitchCard
3. *Parent* : InventoryHolder
Child : Player, CardTable, DeckCard
4. *Parent* : Score
Child : Combination, Card
5. *Parent* : Combination
Child : HighCard, Pair, TwoPair, ThreeKind, Straight, Flush, FullHouse, FourKind, StraightFlush
6. *Parent* : BaseException
Child : InvalidFileException, CardFileException, InvalidInputException, AbilityCommandException, EmptyAbilityCardException

Implementasi dari *inheritance* digunakan untuk memberikan abstraksi terhadap kelas yang terbentuk untuk melakukan *polymorphism*. Implementasi *polymorphism* sendiri dilakukan untuk memudahkan penggunaan dari suatu objek atau ketika banyak objek dari kelas yang berbeda harus diletakkan pada satu *container* yang sama, yaitu dengan menggunakan *pointer* ke objek dari kelas dasarnya saja. Kedua konsep ini memiliki peran yang sangat signifikan dan memberikan kemudahan dalam beberapa kelas yang disebutkan diatas. Berikut adalah contoh implementasinya pada program.



```
1 // File: Command.hpp
2 // Berisi deklarasi dari kelas Command
3
4 #ifndef COMMAND_HPP
5 #define COMMAND_HPP
6
7 class Game;
8
9 class Command {
10     protected:
11         int commandID; // ID command yang bisa diakses user
12
13     public:
14         /*-----*/
15         /*-----CREATION AND DESTRUCTION SEGMENT-----*/
16
17         // ctor
18         Command();
19
20         /*-----*/
21         /*-----COMMAND EXECUTION SEGMENT-----*/
22
23         // abstract method, this class is too abstract to do this
24         virtual void doCommand(Game& game) = 0;
25 };
26
27 #endif
```

Gambar 2.1.1. Kelas Command sebagai kelas *parent*


```

1 // File: Double.hpp
2 // Berisi deklarasi dari kelas Double
3
4 #ifndef COMMAND_DOUBLE_HPP
5 #define COMMAND_DOUBLE_HPP
6
7 #include "Command.hpp"
8
9 class Double : public Command {
10 public:
11     /*-----*/
12     /*-----COMMAND EXECU*/
13
14     // ctor
15     Double();
16
17     /*-----*/
18     /*-----COMMAND EXECU*/
19
20     // tell the game object to do double
21     void doCommand(Game& game);
22 };
23
24 #endif

```

Gambar 2.1.2. Kelas Double sebagai kelas *child*

```

1 // File: AbilityCard.hpp
2 // Berisi deklarasi dari kelas AbilityCard
3
4 #ifndef ABILITY_CARD_HPP
5 #define ABILITY_CARD_HPP
6
7 #include "../Command/Command.hpp"
8
9 class AbilityCard : public Command {
10 public:
11     /*-----*/
12     /*-----CREATION AND DESTRU*/
13
14     // ctor
15     AbilityCard();
16
17     /*-----*/
18     /*-----COMMAND EXECUTI*/
19
20     // abstract method to execute ability, */
21     virtual void doCommand(Game& game) = 0;
22 };
23
24 #endif

```

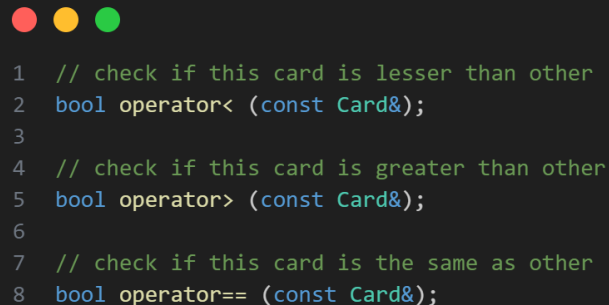
Gambar 2.1.3. Kelas AbilityCard sebagai kelas *child* sekaligus kelas *parent* untuk AbilityCard yang lain

2.2. Method/Operator Overloading

Konsep *Method Overloading* digunakan di kelas DeckCard, sedangkan konsep *operator overloading* digunakan di kelas Card, CardTable, Combination, DeckCard, dan Player. *Method Overloading* dilakukan dengan mendeklarasikan

fungsi dengan nama yang sama namun *signature* yang berbeda (biasanya parameternya yang berbeda). Hal ini dilakukan ketika ada beberapa operasi yang memiliki logika yang sama. Contohnya pada *method* `AddCards` di kelas `DeckCard`, ada *method* yang menerima parameter `InventoryHolder` dan ada yang menerima `string`. Keduanya sama-sama memasukkan kartu ke *deck*, tapi yang satu menerima object `InventoryHolder` sedangkan yang satunya menerima masukan dari file dengan nama yang diterima di parameter.

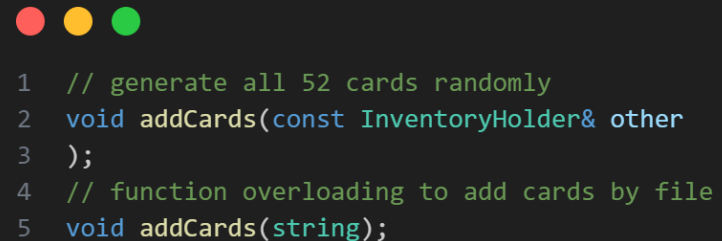
Operator Overloading dilakukan dengan mendeklarasikan *method* dengan nama operator<simbol operasi>. *Operator Overloading* dilakukan untuk menyederhanakan operasi-operasi pada suatu objek. Contohnya pada kelas `CardTable`, untuk menambahkan satu kartu ke tumpukan kartu meja, tinggal menggunakan operasi `+` dengan objek `Card`.



```

1 // check if this card is lesser than other
2 bool operator< (const Card&);
3
4 // check if this card is greater than other
5 bool operator> (const Card&);
6
7 // check if this card is the same as other
8 bool operator== (const Card&);
  
```

Gambar 2.2.1. Definisi *operator overloading* untuk operasi `<`, `>`, dan `==` pada kelas `Card`



```

1 // generate all 52 cards randomly
2 void addCards(const InventoryHolder& other
3 );
4 // function overloading to add cards by file
5 void addCards(string);
  
```

Gambar 2.2.2. Definisi *method overloading* untuk *method* `addCards` pada kelas `DeckCard`

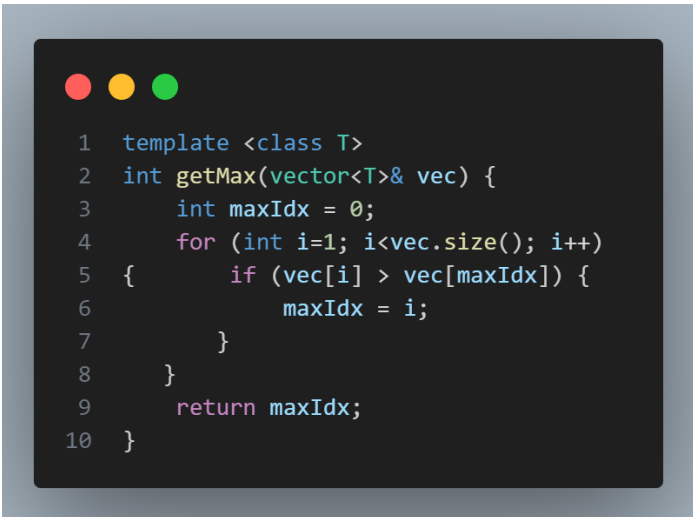
2.3. Template & Generic Classes

Konsep *Template* dan *Generic Classes* digunakan untuk memudahkan penggunaan fungsi dan kelas untuk tipe data yang berbeda-beda. Dengan adanya *template* dan *generic classes*, tipe data yang berbeda dapat dioperasikan

dengan operasi yang sama untuk menghasilkan hasil yang serupa pula. Pada program, diimplementasikan suatu *function template* bernama *getMax*. *Function* ini menerima suatu *vector* berisi tipe *class template* dan mengembalikan indeks elemen terbesar pada *vector* tersebut. *Function* ini digunakan pada beberapa tipe *class* yaitu:

1. Player:
Akan mengembalikan indeks *player* yang memiliki nilai kombinasi kartu terbesar.
2. Card:
Akan mengembalikan indeks *card* dengan *value* terbesar.

Function template memanfaatkan operator *>* yang telah di-*overload* untuk menentukan nilai antara 2 objek. Khusus untuk *vector* berisi pointer ke *Combination*, terdapat *function template specialization* yang digunakan agar elemen *vector* menunjuk kepada objek *Combination*.

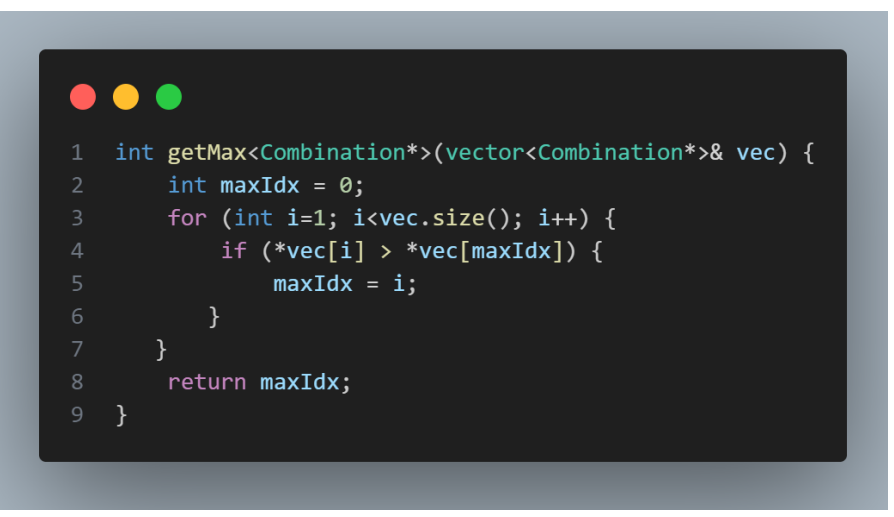


```

1  template <class T>
2  int getMax(vector<T>& vec) {
3      int maxIdx = 0;
4      for (int i=1; i<vec.size(); i++)
5      {          if (vec[i] > vec[maxIdx]) {
6                  maxIdx = i;
7              }
8      }
9      return maxIdx;
10 }

```

Gambar 2.3.1. Definisi dan implementasi dari *template function getMax*



```

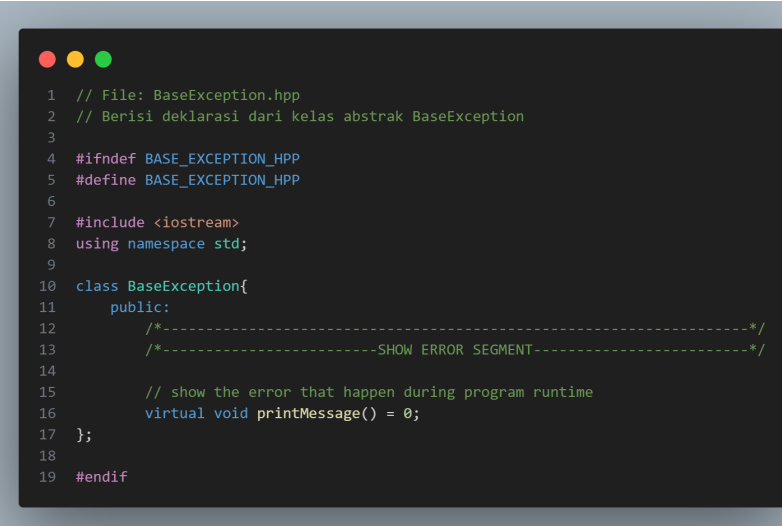
1  int getMax<Combination*>(vector<Combination*>& vec) {
2      int maxIdx = 0;
3      for (int i=1; i<vec.size(); i++) {
4          if (*vec[i] > *vec[maxIdx]) {
5              maxIdx = i;
6          }
7      }
8      return maxIdx;
9  }

```

Gambar 2.3.2. Definisi dan implementasi dari *template specialization function getMax* bagi tipe *Combination**

2.4. Exception

Penanganan terhadap kesalahan yang mungkin muncul pada program menjadi sebuah aspek yang penting. Akan tetapi, dengan banyaknya aspek yang perlu ditangani, penanganan sederhana menjadi tidak efektif. Instruksi tertentu justru membuat program menjadi lebih rumit, sehingga rawan menimbulkan konflik pada proses implementasi. Bahasa C++ menyediakan konsep implementasi *exception* yang menangani kasus kesalahan pada saat *run time* dengan menggunakan sintaks *throw*, *try*, dan *catch*. Sintaks *throw* membuat hasil eksekusi program berakhir secara abnormal. Untuk mengakses eksekusi yang di-*throw*, maka blok *method* yang memungkinkan adanya *exception* diletakkan pada blok *try* dan menggunakan *catch* untuk menangkap objek yang di-*throw*. Berikut adalah implementasi dari kelas *exception*.



```
1 // File: BaseException.hpp
2 // Berisi deklarasi dari kelas abstrak BaseException
3
4 #ifndef BASE_EXCEPTION_HPP
5 #define BASE_EXCEPTION_HPP
6
7 #include <iostream>
8 using namespace std;
9
10 class BaseException{
11 public:
12     /*-----*/
13     /*-----SHOW ERROR SEGMENT-----*/
14
15     // show the error that happen during program runtime
16     virtual void printMessage() = 0;
17 };
18
19 #endif
```

Gambar 2.4.1. Definisi kelas *exception* dasar dalam kelas BaseException

```

1 // File: AbilityCommandException.hpp
2 // Berisi deklarasi dari kelas AbilityCommandException
3
4 #ifndef ABILITY_COMMAND_EXCEPTION_HPP
5 #define ABILITY_COMMAND_EXCEPTION_HPP
6
7 #include "BaseException.hpp"
8
9 class AbilityCommandException : public BaseException {
10 private:
11     bool isUsed;
12     string command;
13
14 public:
15     /*-----*/
16     /*-----CREATION AND DESTRUCTION SEGMENT-----*/
17
18     // user defined ctor
19     AbilityCommandException(bool _isUsed, string _command);
20
21     /*-----*/
22     /*-----SHOW ERROR SEGMENT-----*/
23
24     // show an error when player try to use their ability card
25     void printMessage();
26 };
27
28 #endif

```

Gambar 2.4.2. Definisi kelas *exception* turunan dalam kelas AbilityCommandException

```

1 // File: InvalidFileException.hpp
2 // Berisi deklarasi dari kelas InvalidFileException
3
4 #ifndef INVALID_FILE_EXCEPTION_HPP
5 #define INVALID_FILE_EXCEPTION_HPP
6
7 #include "BaseException.hpp"
8
9 class InvalidFileException : public BaseException {
10 public:
11     /*-----*/
12     /*-----CREATION AND DESTRUCTION SEGMENT-----*/
13
14     // ctor
15     InvalidFileException();
16
17     /*-----*/
18     /*-----SHOW ERROR SEGMENT-----*/
19
20     // show an error when file is not found
21     void printMessage();
22 };
23
24 #endif

```

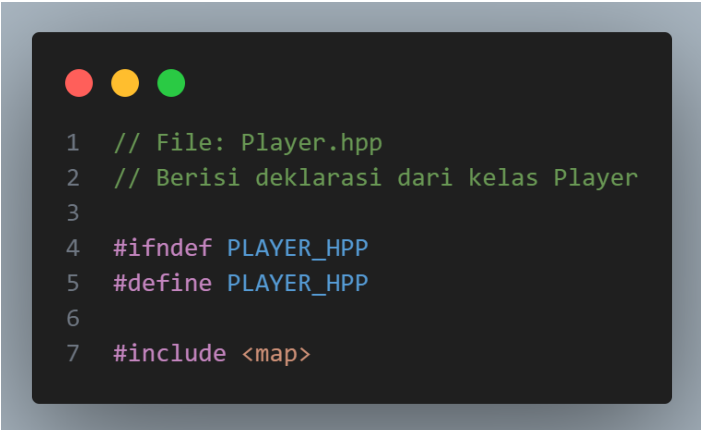
Gambar 2.4.3. Definisi kelas *exception* turunan dalam kelas InvalidFileException

2.5. C++ Standard Template Library

Konsep C++ STL berhasil diimplementasikan pada berbagai kelas pada seperti DeckCard, Player, Game, dan masih banyak lagi. STL yang digunakan antara lain *vector*, *map*, *pair*, dan algoritma STL seperti *find_if*, *for_each*, *transform*, dan *sort*. Pemanfaatan STL memungkinkan adanya penggunaan berbagai jenis *container* yang disediakan untuk menampung objek dan memudahkan perhitungan dan implementasi berbagai *method*. *Map* digunakan untuk

menyimpan variabel *collection* yang tipe datanya berbentuk *pair* seperti *listCommands* dan kode *Cards*. *Vector* digunakan dalam berbagai *collection* yang bersifat dinamis. Berikut adalah daftar implementasi dari penggunaan STL :

1. *Vector* : CardTable, Combination dan kelas *child*-nya, DeckCard
2. *Map* : Player, DeckCard
3. *Pair* : Parse, DeckCard




```

1 // File: Player.hpp
2 // Berisi deklarasi dari kelas Player
3
4 #ifndef PLAYER_HPP
5 #define PLAYER_HPP
6
7 #include <map>

```

Gambar 2.5.1. Proses *include library* STL *map* pada Kelas Player



```

1 void DeckCard::addCards(string inputfile) {
2     Parse p1;
3     ifstream inputFile(inputfile);
4     p1.parsing(inputfile);
5     vector<pair<int,int>> result = p1.getCards
6 ();
7     // Masukin nilai
8     vector<Card> cards;
9     for (auto i : result) {
10         Card card(i.second, i.first);
11         cards.push_back(card);
12     }
13     for (auto i : cards) {
14         *this = *this + i;
15     }
16 }

```

Gambar 2.5.3. Implementasi *method addCards* menggunakan STL *vector* pada Kelas DeckCard



```

1 // fill list of commands with pair<string, int>
2 void fillMapsOfCommand(map<string, int>& listCommands);
3
4 // calculate the player highest combo with the cards on the table
5 void calculateCombo(vector<Card>,int);

```

Gambar 2.5.2. Definisi *method fillMapsOfCommand* dan *calculateCombo* yang menggunakan STL *vector* dan *map* pada Kelas Player



```

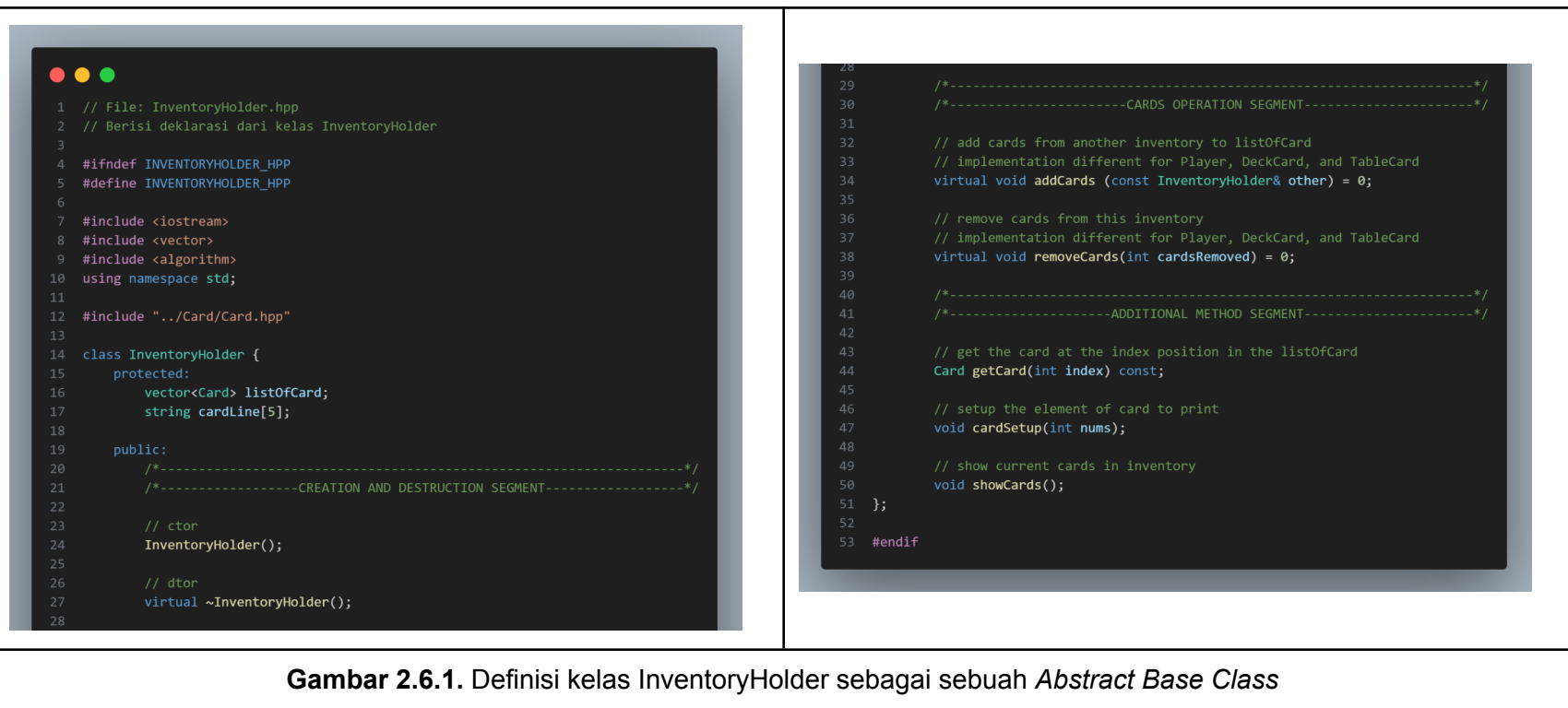
1 vector<Player> playerList;
2 vector<int> turnList;
3 int turn;
4
5 vector<Command*> commandList;

```

Gambar 2.5.4. Definisi berbagai atribut yang menggunakan STL *vector* dengan delemen yang berbeda-beda pada Kelas Game

2.6. Konsep OOP lain

Konsep Pemrograman Berbasis Objek lain yang diimplementasikan adalah *Abstract Base Class* dan *Composition*. Konsep *Abstract Base Class* banyak diimplementasikan pada realisasi program, salah satunya digunakan pada *base class* InventoryHolder yang di-*inherit* oleh kelas Player, kelas CardTable, dan kelas DeckCard, serta *base class* Score yang di-*inherit* oleh kelas Card dan kelas Combination. Sedangkan konsep *Composition* digunakan pada kelas Game yang pendefinisianya menggunakan atribut kelas DeckCard, kelas CardTable, kelas Command, dan kelas Player.





```
1  class Game {  
2      private:  
3          DeckCard deckCard;  
4          CardTable cardTable;  
5  
6          vector<Player> playerList;  
7          vector<int> turnList;  
8          int turn;  
9  
10         vector<Command*> commandList;  
11         long long bonusPoint;  
12         long long bet;  
13         int winner;
```

Gambar 2.6.2. Konsep *composition* pada kelas Game yang disusun atas elemen dari kelas DeckCard, kelas CardTable, kelas Command, dan kelas Player

3. Justifikasi *Library* Umum dan STL Opsional yang Digunakan

a. *Library cmath* :

Library ini digunakan untuk melakukan operasi matematika dasar seperti perpangkatan (pow) yang dibutuhkan untuk menentukan nilai kombinasi. Kebutuhan muncul karena terjadinya perpangkatan 10 dengan nilai yang lumayan besar.

b. *Library random*:

Library ini digunakan untuk melakukan pembangkitan kartu pada objek DeckCard dan membagikan objek AbilityCard kepada pemain secara acak. Di dalam *library*, digunakan beberapa tipe data seperti mt19937 (*mersenne twister engine*), result_type, dan random_device. Penggunaan tipe data ini dibutuhkan untuk menghasilkan hasil randomisasi yang lebih optimal dan dapat diandalkan dibandingkan dengan menggunakan fungsi srand dan rand.

c. *Library chrono*:

Library ini digunakan secara bersamaan dengan *library random* untuk mengambil nilai waktu saat ini. Nilai waktu tersebut akan digunakan sebagai seed randomisasi pada *mersenne twister engine*.

d. *Library fstream*:

Library ini digunakan untuk membuka *file* konfigurasi Card dan mengubahnya menjadi *stream* untuk keperluan *parsing*.

e. *Library sstream*:

Library ini digunakan mengubah setiap *line* dari *filestream* menjadi *stringstream*. Hal ini dibutuhkan untuk keperluan *parsing*.

f. *Library algorithm*:

Library algorithm digunakan untuk melakukan modifikasi tipe data di dalam *container* berupa *vector* ataupun *string*. Algoritma digunakan untuk memudahkan pemrosesan di dalam *container* dan memberikan kompleksitas komputasi yang lebih efisien. Algoritma juga dipakai untuk meminimalisir penggunaan pengulangan (terutama *for loop*) yang tidak terlalu cocok untuk paradigma berorientasi objek. Beberapa algoritma yang digunakan adalah:

1. find_if:

Algoritma `find_if` digunakan di dalam kelas `Player` untuk mencari nilai *key* yang *valuenya* sudah diketahui. Kebutuhan tersebut muncul karena objek *player* menyimpan suatu kode *ability* yang dimilikinya namun tidak memiliki nama *ability* tersebut, padahal *map* yang tersedia menyimpan nama *ability* sebagai *key* dan kode *ability* sebagai *value*. Akibatnya, *method member map* yaitu *find* tidak dapat digunakan. Penggunaan algoritma `find_if` mengurangi keharusan program untuk memakai *control flow* berupa *if statement* dan *switch* yang dekat dengan paradigma prosedural.

2. `for_each`:

Algoritma `for_each` banyak digunakan di dalam program. Algoritma ini digunakan ketika terjadi keharusan untuk memproses elemen *vector* satu per satu. Sebagai contoh, pada kelas `Game`, algoritma `for_each` digunakan untuk menambahkan 2 kartu dari objek *deckCard* kepada objek *player* di awal ronde 1 permainan. Penggunaan algoritma `for_each` mengurangi penggunaan *for loop* yang dekat dengan paradigma prosedural.

3. `transform`:

Algoritma `transform` digunakan di dalam kelas `Player`. Algoritma ini digunakan untuk mengubah string *command* yang berasal dari input *player* menjadi string dengan huruf kecil. Hal ini dibutuhkan untuk mengakomodasi tipe *user* yang mungkin saja memasukkan input dengan huruf besar dan kecil bergantian, namun memiliki maksud untuk menjalankan *command* yang sama.

4. `shuffle`:

Algoritma `shuffle` digunakan di dalam kelas `DeckCard` dan `Game`. Algoritma ini digunakan oleh kelas `DeckCard` untuk mengacak *vector of integer* yang kemudian akan dipetakan menjadi *vector of Card*. Sementara itu, kelas `Game` mengacak *vector of integer* yang kemudian dipetakan menjadi *AbilityCard* yang akan diberikan kepada *player*. Algoritma `shuffle` ini digunakan bersama-sama dengan *library random*.

5. `sort`:

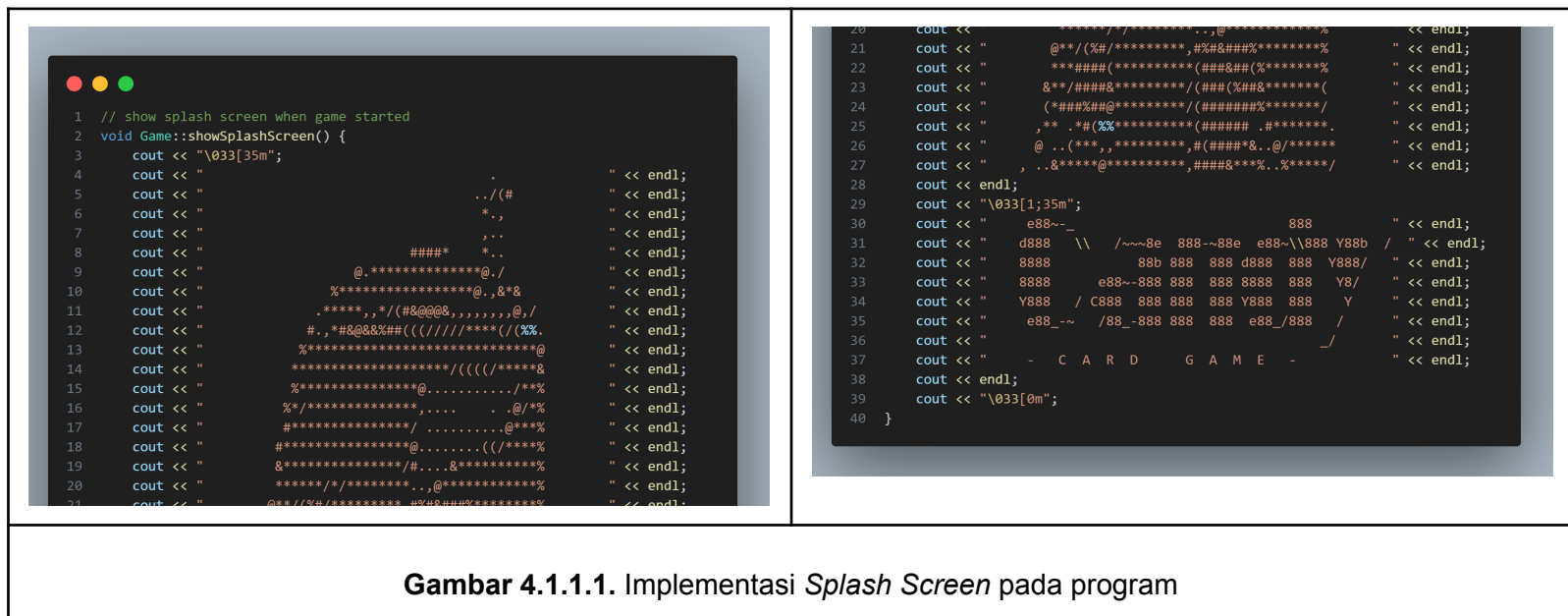
Algoritma `sort` digunakan terutama oleh kelas `Game`. Algoritma ini akan mengurutkan elemen-elemen *vector* berdasarkan kriteria tertentu. Sebagai contoh, algoritma ini digunakan untuk menampilkan *leaderboard* di akhir permainan. Karena *leaderboard* yang ditampilkan terurut mengecil berdasarkan nilai poin *player*, maka harus dilakukan *sorting* terlebih dahulu sebelum menampilkannya. Akhirnya, digunakanlah algoritma `sort` dengan menggunakan *lambda expression* sebagai kriteria pengurutan.

4. Bonus yang Dikerjakan

4.1. Bonus Kreasi Mandiri

4.1.1. Pembuatan *Splash Screen*

Program yang penulis buat mengimplementasikan *splash screen* sebagai tampilan awal sebelum program memasuki menu utama. Yang ditampilkan pada bagian ini adalah ASCII *art* dari Princess Bubblegum dan teks bertuliskan “*Candy Card Game*” berwarna merah muda.



```

      .
      . / (#
      * ,
      , .
      * .
      #####*
      @.*****@./
      %*****@.,&*%
      .***** ,*/ (#&@@@&, , , , , , @, /
      #. ,*#&@&%##(((// //****(/(%%.
      %*****@
      *****/(((/****&
      %*****@...../**%
      %*/***** , . . . . . @/*%
      #*****/ .....@**%
      #*****@.....(/****%
      &*****/#...&*****%
      *****/*/*.....@*****%
      @**/(%#/****** ,#%#&###%*****%
      ***####(***** (###&##(%*****%
      &*/*###&*****/(###(%##&***** (
      (*###%##@*****/(#####%*****/
      ,** .*(%***** (##### .#*****.
      @ .. (** , , ***** ,#(#####&. @/*****
      , ..&*****@***** ,####&***%..%*****/

      e88~-_                               888
      d888 \ /~~~8e 888~88e e88~\888 Y88b /
      8888      88b 888 888 d888 888 Y888/
      8888      e88~-888 888 888 8888 888 Y8/
      Y888 / C888 888 888 888 Y888 888 Y
      e88_~ /88_-888 888 888 e88_/888 /
      _/
      - C A R D      G A M E -

```

Gambar 4.1.1.2. Hasil implementasi *Splash Screen* pada terminal

4.1.2. Pewarnaan Terminal

Program yang penulis buat mengimplementasikan pewarnaan terminal untuk membuat tampilan yang lebih menarik. Pewarnaan dilakukan pada saat menerima masukan, pilihan menu, kartu, dan detail menu setelah *command* dipilih.

```

===== ROUND 1 =====
The cards on the table are:
=====
| 4 |
|   |
| - LLH - |
|   |
| 4 |
=====

This is a's turn!
Your cards:
=====
| 1 | | 8 |
|   | |   |
| - LLH - | | - LLH - |
|   | |   |
| 1 | | 8 |
=====


Your current point: 1000
You currently do not have any ability card
Insert your command!
>> DOUBLE
a do DOUBLE!
The prize points goes up from 64 to 128!

```

Gambar 4.1.2. Hasil implementasi pewarnaan terminal

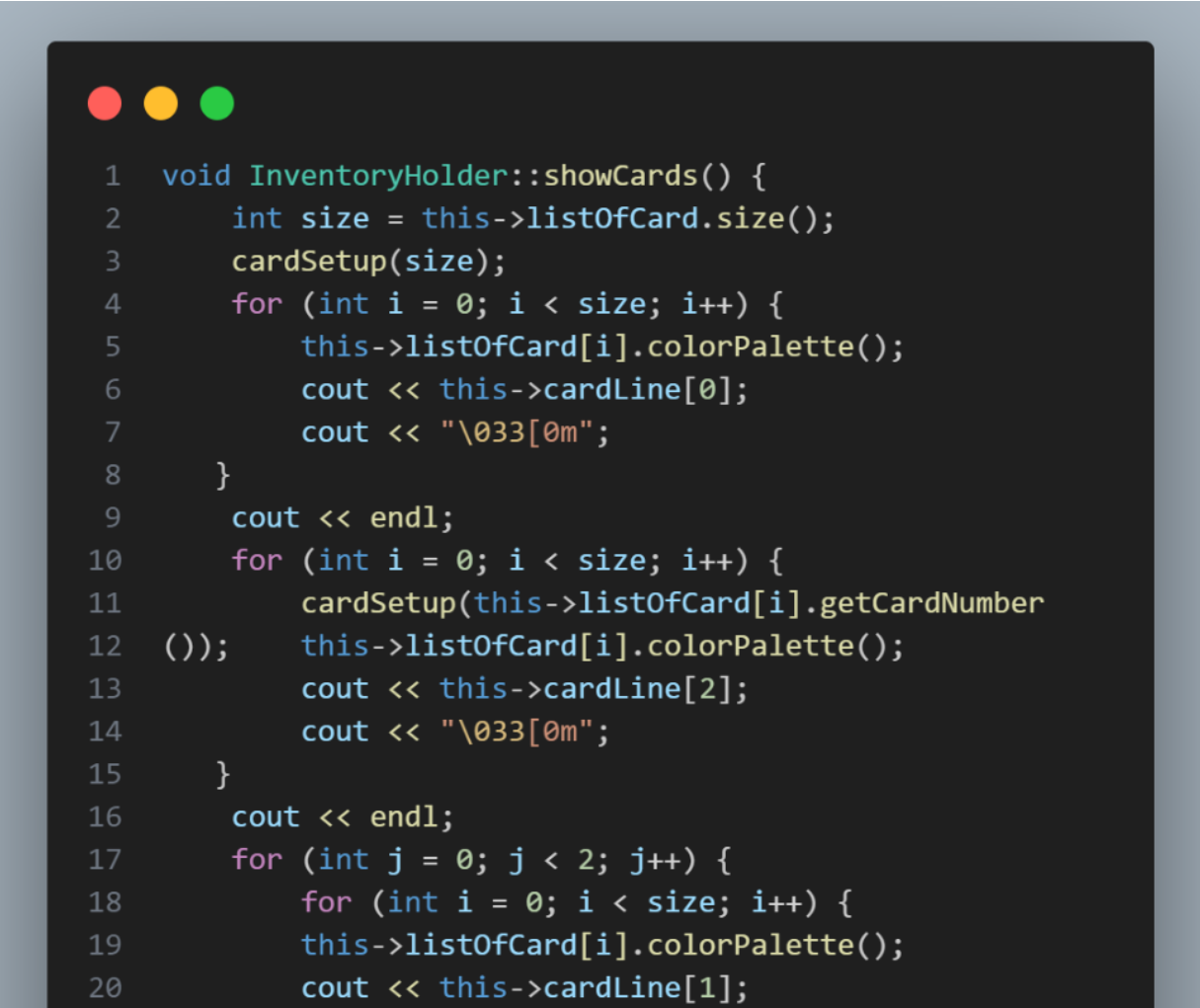
4.1.3. Perwujudan Fisik Kartu Permainan

Pada program juga dilakukan implementasi bentuk fisik kartu berdasarkan nomor lengkap dengan warnanya. Implementasinya dilakukan dengan melakukan pencetakan baris per baris sejumlah kartu yang akan ditampilkan berdasarkan elemen penyusun kartu terkait hingga semua kartu tercetak dan semua baris tercetak dengan baik.



```
1 void InventoryHolder::cardSetup(int nums) {
2     this->cardLine[0] = "===== ";
3     this->cardLine[1] = "|           | ";
4     if (nums < 10) {
5         this->cardLine[2] = "| " + to_string(nums) + "           | ";
6         this->cardLine[3] = "|           " + to_string(nums) + " | ";
7     } else {
8         this->cardLine[2] = "| " + to_string(nums) + "           | ";
9         this->cardLine[3] = "|           " + to_string(nums) + " | ";
10    }
11    this->cardLine[4] = "|   - LLH -   | ";
12 }
```

Gambar 4.1.3.1. Implementasi *setup* elemen kartu yang akan dicetak



```
1 void InventoryHolder::showCards() {
2     int size = this->listOfCard.size();
3     cardSetup(size);
4     for (int i = 0; i < size; i++) {
5         this->listOfCard[i].colorPalette();
6         cout << this->cardLine[0];
7         cout << "\033[0m";
8     }
9     cout << endl;
10    for (int i = 0; i < size; i++) {
11        cardSetup(this->listOfCard[i].getCardNumber
12    ());    this->listOfCard[i].colorPalette();
13        cout << this->cardLine[2];
14        cout << "\033[0m";
15    }
16    cout << endl;
17    for (int j = 0; j < 2; j++) {
18        for (int i = 0; i < size; i++) {
19            this->listOfCard[i].colorPalette();
20            cout << this->cardLine[1];
```

Gambar 4.1.3.2. Implementasi *method showCards* untuk menampilkan kartu berdasar elemen *setup* (1)


```

21     cout << "\033[0m";
22     }
23     cout << endl;
24 }
25 for (int i = 0; i < size; i++) {
26     this->listOfCard[i].colorPalette();
27     cout << this->cardLine[4];
28     cout << "\033[0m";
29 }
30 cout << endl;
31 for (int j = 0; j < 2; j++) {
32     for (int i = 0; i < size; i++) {
33         this->listOfCard[i].colorPalette();
34         cout << this->cardLine[1];
35         cout << "\033[0m";
36     }
37     cout << endl;
38 }
39 for (int i = 0; i < size; i++) {
40     cardSetup(this->listOfCard[i].getCardNumber
41 ());    this->listOfCard[i].colorPalette();
42     cout << this->cardLine[3];
43     cout << "\033[0m";
44 }
45 cout << endl;
46 for (int i = 0; i < size; i++) {
47     this->listOfCard[i].colorPalette();
48     cout << this->cardLine[0];
49     cout << "\033[0m";
50 }
51 cout << endl;
52 }

```

Gambar 4.1.3.3. Implementasi *method showCards* untuk menampilkan kartu berdasar elemen *setup* (2)



4.1.4. Mekanisme Taruhan (*Bet*)

Tanpa mengubah skema permainan, implementasi program menambahkan mekanisme *bet*. Secara garis besar, konsep ini mirip seperti *bet* pada umumnya. Pada awal sebuah permainan besar, setiap pemain akan diberikan poin hadiah sebesar 1000 dan pada awal setiap *game* (termasuk jika permainan tidak selesai dalam 1 *game* akibat belum ada yang mencapai jumlah poin 2^{32}), setiap pemain akan diminta untuk memasukkan nominal *bet* yang ingin dipertaruhkan oleh setiap pemain. Nominal tersebut akan “dipotong” dari poin hadiah setiap pemain untuk kemudian disimpan dalam penampung sementara yang berisi akumulasi nilai tersebut. Pada akhir dari setiap *game*, pemain yang menang akan mendapatkan poin hadiah yang nilainya telah diubah sepanjang permainan ditambah dengan akumulasi hasil *bet* dari setiap pemain yang bertanding. Implementasi ini membawa tantangan tersendiri bagi setiap pemain yang bermain karena bisa saja pada akhir permainan besar, ada pemain yang sudah tidak memiliki poin hadiah.



```
1  long long Player::askForBet() {
2      bool valid;
3      string temp;
4
5      cout << endl << "This is " << "\033[92m" << this->name << "\033[0m" << "'s turn!" << endl
6  ; cout << "Your cards: " << endl;
7      showCards();
8      cout << "Your current point: " << "\033[35m" << this->currentPoin << "\033[0m" << endl;
9      while (!valid) {
10         try {
11             cout << "Please choose how many points you want to bet!" << endl;
12             cout << ">> " << "\033[34m";
13             cin >> temp;
14             cout << "\033[0m";
15
16             if (!temp.empty() && all_of(temp.begin(), temp.end(), ::isdigit))
17                 || (stoll(temp) < 0 || stoll(temp) > this->currentPoin)) {
18                 InvalidInputException e;
19                 throw e;
20             }
21             valid = true;
22         }
23         catch (InvalidInputException e) {
24             e.printMessage();
25         }
26     }
27
28     this->currentPoin -= stoll(temp);
29     return stoll(temp);
30 }
```

Gambar 4.1.4.1. Implementasi *method askForBet* untuk menangani mekanisme *bet*

```

===== BETTING TIME =====
This is player1's turn!
Your cards:
=====
| 5 | | 3 |
|   | |   |
| - LLH - | | - LLH - |
|   | |   |
| 5 | | 3 |
=====
Your current point: 1000
Please choose how many points you want to bet!
>> 150

This is player2's turn!
Your cards:
=====
| 11 | | 13 |
|   | |   |
| - LLH - | | - LLH - |
|   | |   |
| 11 | | 13 |
=====
Your current point: 1000
Please choose how many points you want to bet!
>> 200

```

Gambar 4.1.4.2. Pada awal setiap *game* akan ada *section betting time* yang menanyakan nominal *bet* setiap pemain

Pada awal permainan besar, setiap pemain diberi poin hadiah sebesar 1000 poin.

```

===== FINAL RESULT =====
The game ends!
Leaderboard:
1. player1: 4294970996
2. player2: 800
3. player3: 700
4. player4: 600
5. player5: 500
6. player6: 400
7. player7: 300

player1 won the game! Congratulations!

```

Gambar 4.1.4.3. Hasil *leaderboard* pada saat permainan besar berakhir

Sebagai detail, berikut adalah nominal *bet* setiap pemain pada awal *game* :

- Player1 : 150
- Player2 : 200
- Player3 : 300
- Player4 : 400
- Player5 : 500
- Player6 : 600
- Player7 : 700

Player1 menang permainan dengan total poin 2^{32} atau 4294967296. Selain poin bonus, Player1 juga akan mendapatkan poin tambahan sebesar total *bet* atau 2850, sehingga total poinnya menjadi $4294967296 + (1000 - 150) + 2850 = 4294970996$ dan pengurangan poin hadiah setiap pemain yang kalah sebesar *bet*.

5. Lampiran

5.1. Pembagian Tugas

Modul (dalam poin spek)	Implementer	Tester
Command	13521062, 13521108, 13521160	13521062, 13521084, 13521108, 13521160, 13521172
InventoryHolder	13521062	13521062, 13521084, 13521108, 13521160, 13521172
Score	13521084, 13521172	13521062, 13521084, 13521108, 13521160, 13521172
Game	13521062, 13521108, 13521160	13521062, 13521084, 13521108, 13521160, 13521172
ExceptionHandling	13521084	13521062, 13521084, 13521108, 13521160, 13521172
Parsing	13521084, 13521108	13521062, 13521084, 13521108, 13521160, 13521172

5.2. Foto Kelompok



Gambar 5.2. Foto Kelompok kami yang sudah **LeLaH**

FORM ASISTENSI

Kode Kelompok : LLH

Nama Kelompok : LeLaH

1. 13521062 / Go Dillon Audris
2. 13521084 / Austin Gabriel Pardosi
3. 13521108 / Michael Leon Putra Widhi
4. 13521160 / M. Dimas Sakti Widyatmaja
5. 13521172 / Nathan Tenka

Asisten Pembimbing : Widya Anugrah Putra

1. Konten Diskusi

1.1. Teknis *Combo*

- Untuk setiap pemain di dalam sebuah permainan (dari ronde 1 sampai 6) hanya memiliki 2 kartu, ditambah dengan kartu yang di meja.
- Pada akhir ronde ke-6, semua pemain menunjukkan kartunya untuk kemudian dilakukan kalkulasi kombinasi yang paling tinggi.
- Kalau kartu yang di meja sudah yang kombinasi (*straight flush*), berarti ditentukan oleh perbandingan besar nominal dari kartu yang dimiliki setiap pemain.
- Hanya memperhitungkan kombinasi tertinggi untuk kemudian dipertandingkan dengan nilai kombinasi tertinggi pemain lain.

1.2. *Functional Overloading*

- Yang dibandingkan kombinasi kartu.

1.3. *Inheritance* dan *Polymorphism*

- Mempertegas makna kata “*Action*” (apakah termasuk dalam *ability* dan *command* atau bukan).

1.4. Validasi Input dengan File

- Formatnya ada dan akan dibagikan kemudian.

1.5. *AbilityCard*

- Hanya dibagi di *round* kedua saja, jadi ga akan mungkin ada player yang punya 2 ability card, dan jika sudah dipakai maka tidak dapat dipakai lagi.
- Untuk *Ability Reverse Card*, pada round robin selanjutnya, urutannya ga akan balik normal.

1.6. Desain Kelas

- Dibebaskan yang penting sesuai spek saja, jangan sampai melanggar spek.

2. Tindak Lanjut

2.1. Teknis *Combo*

- Skema pembagian kartu pada setiap *game* diberikan dua buah kartu.
- Menangani kasus saat *Card Table* sudah memenuhi kondisi *straight flush* dengan menentukan nilai kombinasi yang satu tingkat lebih rendah dari yang ada di meja.

2.2. *Functional Overloading*

- Digunakan dua buah *operator overloading* yaitu membandingkan poin kombinasi kartu antar pemain dan membandingkan kombinasi kartu yang terjadi dalam 1 orang pemain yang sama.

2.3. *Inheritance* dan *Polymorphism*

- Diimplementasikan dalam sebuah kelas *Command* yang meliputi *command* itu sendiri dan kelas *AbilityCard* sebagai *child* yang meng-*inherit* kelas *Command*.

2.4. Validasi Input dengan File

- Diinformasikan di akhir bahwa format masukan bebas, maka implementasi dilakukan dengan menggunakan *file* yang berisi 52 buah pasang kartu dengan nilai pertama menyatakan nilai kartu (1 - 13) dengan nilai warna (dinyatakan dalam angka 0 - 3) dengan setiap kartu memiliki nilai yang unik

2.5. AbilityCard

- Mengimplementasikan skema *round robin* sesuai dengan yang disampaikan pada spesifikasi tugas besar.