Kode Kelompok : KYS
Nama Kelompok : Suicide Squad
1. 13521096 / Noel Christoffel Simbolon
2. 13521114 / Farhan Nabil Suryono
3. 13521134 / Rinaldy Adin
4. 13521142 / Enrique Alifio Ditya
5. 13521148 / Johanes Lee
Asisten Pembimbing : Fabian Savero Diaz Pranoto

1. Diagram Kelas

Setelah mendekomposisi permasalahan yang tertera pada spesifikasi tugas, kami memutuskan untuk mengimplementasikan program yang terdiri dari kelas-kelas yang dibagi-bagi menjadi beberapa *group* besar. Berikut merupakan simplifikasi dari diagram kelas serta hierarki *inheritance*-nya. Kami sengaja menghilangkan detail-detail implementasi terlebih dahulu agar pembaca diagram dapat mengerti *high-level structure* dari program kami terlebih dahulu.

Kami mendesain kelas-kelas yang dapat merepresentasikan aksi pada permainan. Dasar dari kelas kelas ini adalah kelas `BaseCommand`. `BaseCommand` dibuat sebagai *generic class* untuk mempermudah membuat permainan lain dengan tipe-tipe aksi (*commands)* yang berbeda. Berikut adalah diagram untuk kelas-kelas tersebut yang kami implementasikan dalam program kami.

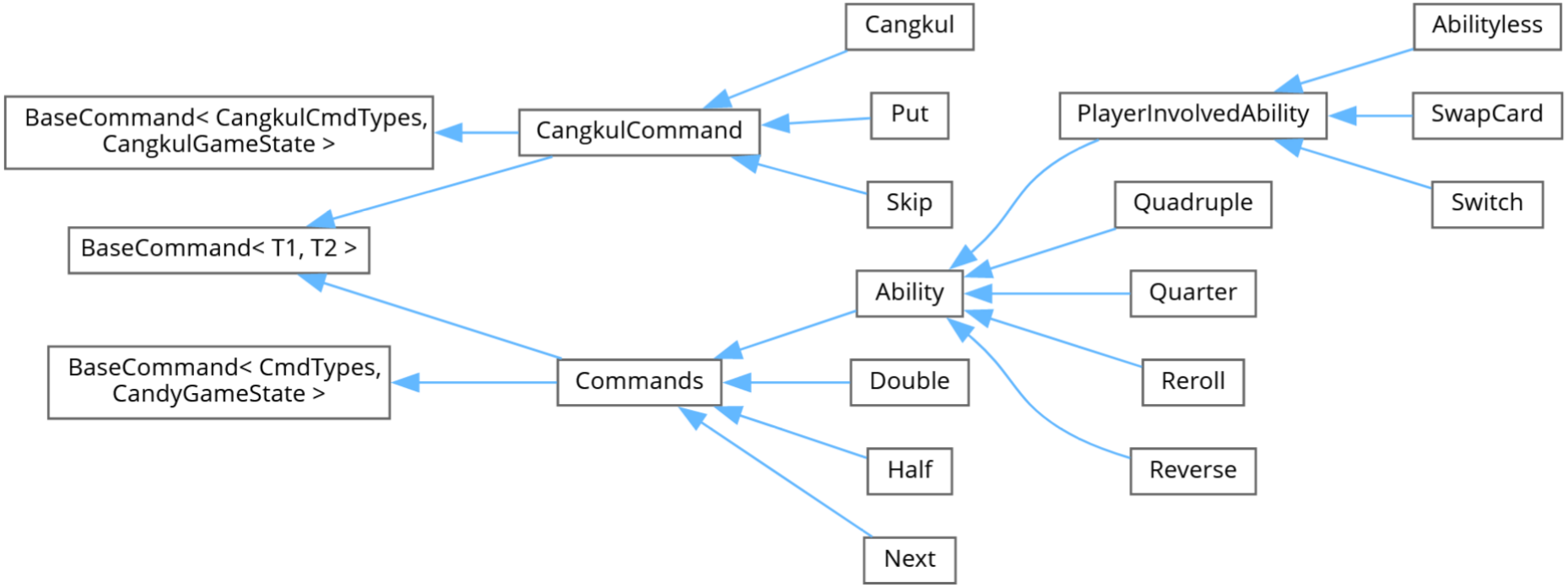
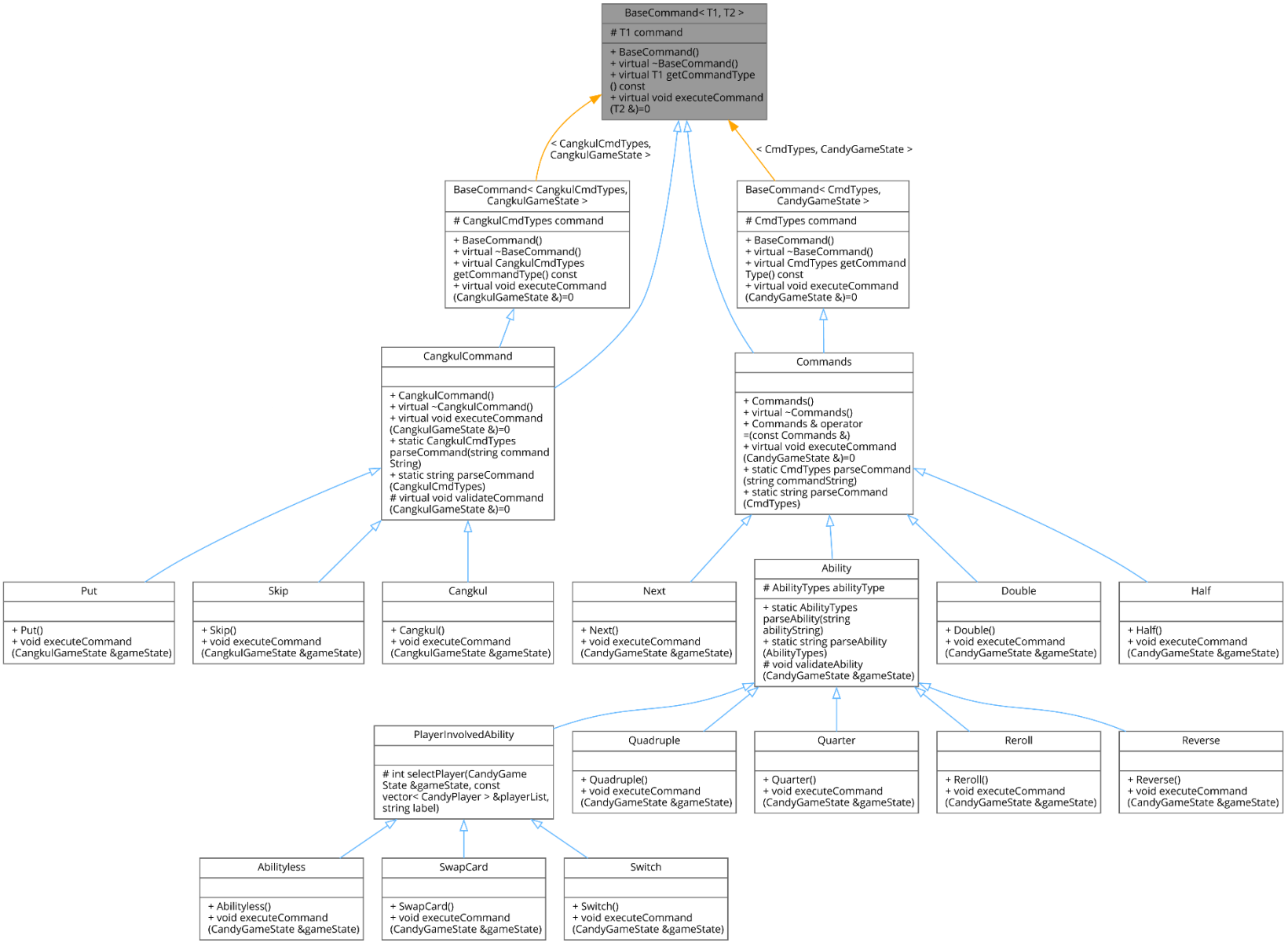


Diagram berikut merupakan versi lebih lengkapnya dan mengikuti standar UML sehingga lebih detail dalam menspesifikasikan atribut-atribut dan *method-method* pada tiap kelasnya.


BaseCommand< T1, T2 >
T1 command
+ BaseCommand()
+ virtual ~BaseCommand()
+ virtual T1 getCommandType() const
+ virtual void executeCommand(T2 &)=0
< CangkulCmdTypes, CangkulGameState >
< CmdTypes, CandyGameState >
BaseCommand< CangkulCmdTypes, CangkulGameState >
CangkulCmdTypes command
+ BaseCommand()
+ virtual ~BaseCommand()
+ virtual CangkulCmdTypes getCommandType() const
+ virtual void executeCommand(CangkulGameState &)=0
BaseCommand< CmdTypes, CandyGameState >
CmdTypes command
+ BaseCommand()
+ virtual ~BaseCommand()
+ virtual CmdTypes getCommandType() const
+ virtual void executeCommand(CandyGameState &)=0
CangkulCommand
+ CangkulCommand()
+ virtual ~CangkulCommand()
+ virtual void executeCommand(CangkulGameState &)=0
+ static CangkulCmdTypes parseCommand(string commandString)
+ static string parseCommand(CangkulCmdTypes)
# virtual void validateCommand(CangkulGameState &)=0
Commands
+ Commands()
+ virtual ~Commands()
+ Commands & operator=(const Commands &)
+ virtual void executeCommand(CandyGameState &)=0
+ static CmdTypes parseCommand(string commandString)
+ static string parseCommand(CmdTypes)
Put
+ Put()
+ void executeCommand(CangkulGameState &gameState)
Skip
+ Skip()
+ void executeCommand(CangkulGameState &gameState)
Cangkul
+ Cangkul()
+ void executeCommand(CangkulGameState &gameState)
Next
+ Next()
+ void executeCommand(CandyGameState &gameState)
Ability
AbilityTypes abilityType
+ static AbilityTypes parseAbility(string abilityString)
+ static string parseAbility(AbilityTypes)
# void validateAbility(CandyGameState &gameState)
Double
+ Double()
+ void executeCommand(CandyGameState &gameState)
Half
+ Half()
+ void executeCommand(CandyGameState &gameState)
PlayerInvolvedAbility
int selectPlayer(CandyGameState &gameState, const vector< CandyPlayer > &playerList, string label)
Quadruple
+ Quadruple()
+ void executeCommand(CandyGameState &gameState)
Quarter
+ Quarter()
+ void executeCommand(CandyGameState &gameState)
Reroll
+ Reroll()
+ void executeCommand(CandyGameState &gameState)
Reverse
+ Reverse()
+ void executeCommand(CandyGameState &gameState)
Abilityless
+ Abilityless()
+ void executeCommand(CandyGameState &gameState)
SwapCard
+ SwapCard()
+ void executeCommand(CandyGameState &gameState)
Switch
+ Switch()
+ void executeCommand(CandyGameState &gameState)

Selanjutnya kami mendesain kelas-kelas yang dapat merepresentasikan kartu dan kombinasi kartu pada permainan. Kelas abstrak CardInterface digunakan sebagai standar pengaksesan nilai kartu dan kombinasi yang akan digunakan dalam menghitung nilai dan menentukan pemenang pada suatu sub-permainan. Berikut adalah diagram untuk kelas-kelas tersebut yang kami implementasikan dalam program kami.

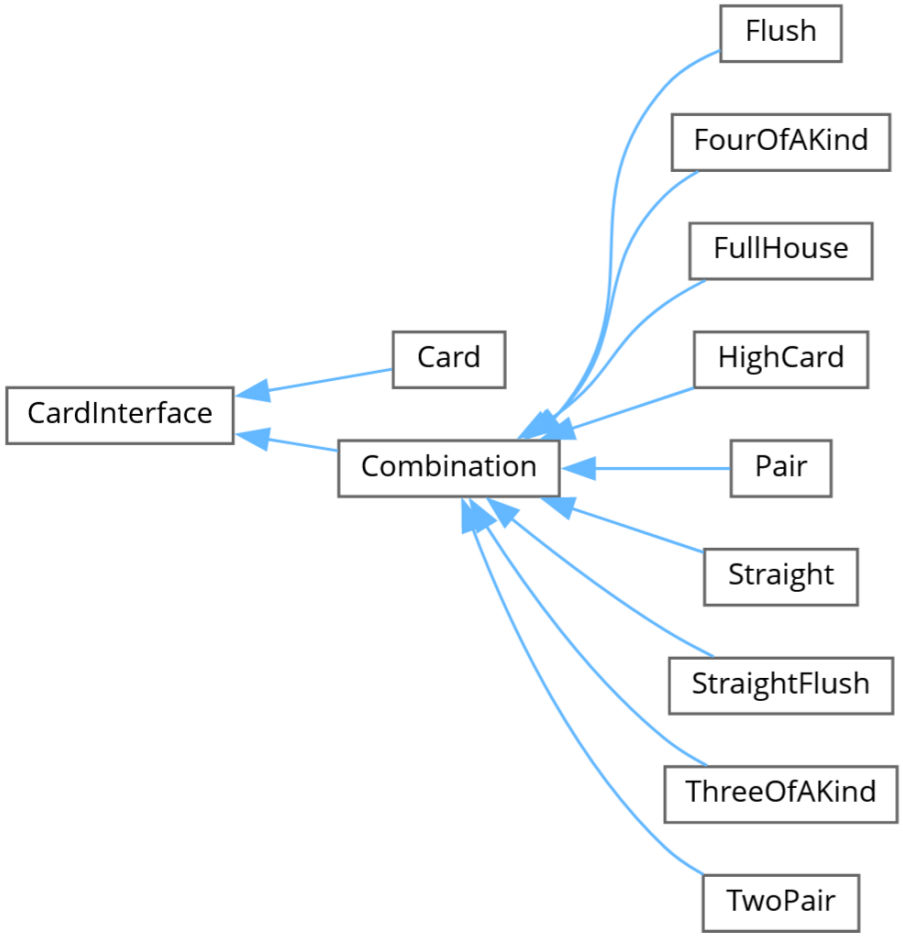
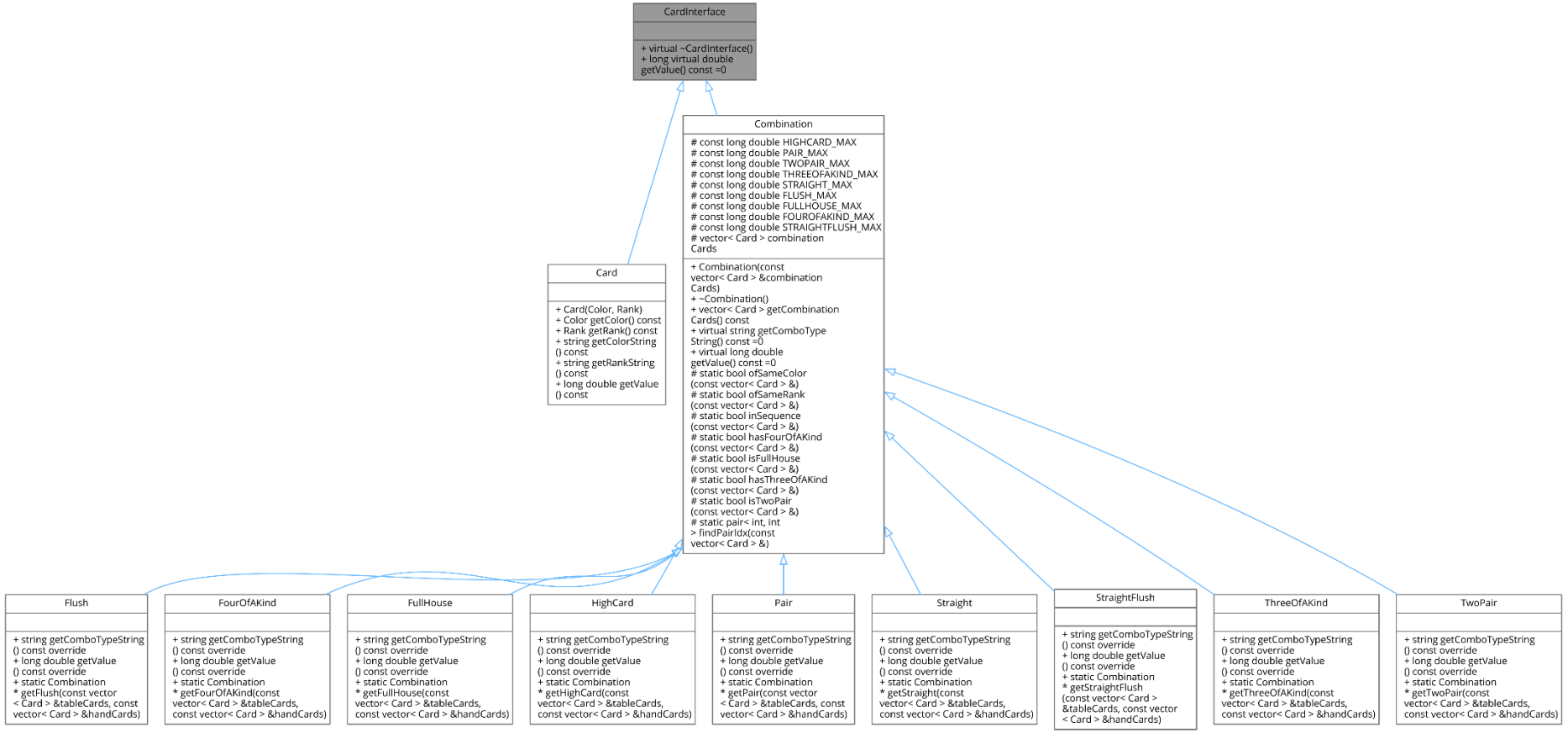


Diagram berikut merupakan versi lebih lengkapnya dan mengikuti standar UML sehingga lebih detail dalam menspesifikasikan atribut-atribut dan *method-method* pada tiap kelasnya.



Selanjutnya, kami mendesain kelas-kelas yang dapat mengatur *control flow* dan jalannya permainan. Kelas ini merupakan kelas abstrak yang kelas turunannya dapat mengimplementasikan alur permainan yang berbeda-beda. Berikut adalah diagram untuk kelas-kelas tersebut yang kami implementasikan dalam program kami.

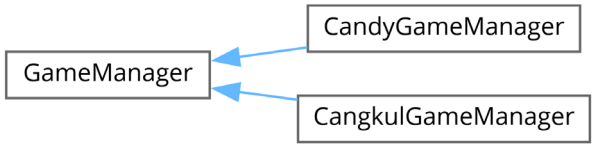
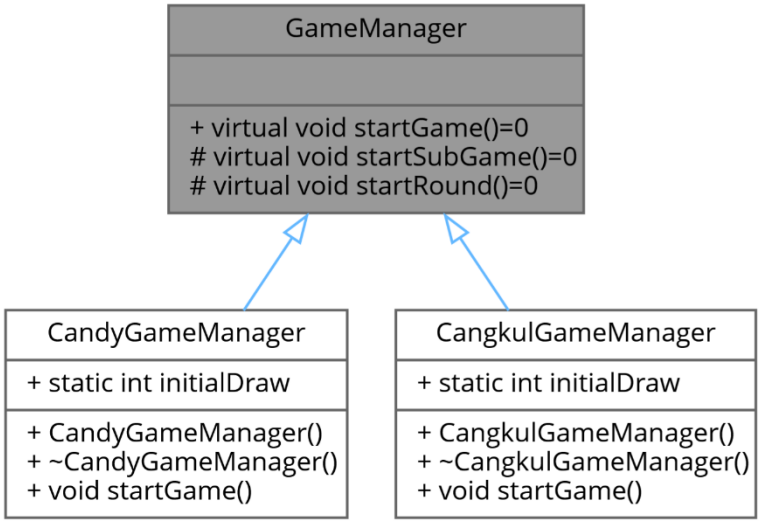


Diagram berikut merupakan versi lebih lengkapnya dan mengikuti standar UML sehingga lebih detail dalam menspesifikasikan atribut-atribut dan *method-method* pada tiap kelasnya.



Selanjutnya, kami mendesain kelas-kelas yang dapat merepresentasikan suatu instansi keadaan permainan pada suatu waktu. Kelas ini dibuat agar kelas aksi dapat melakukan perubahan terhadap status permainan serta komponen-komponen permainan terkumpul ke dalam suatu objek. Berikut adalah diagram untuk kelas-kelas tersebut yang kami implementasikan dalam program kami.

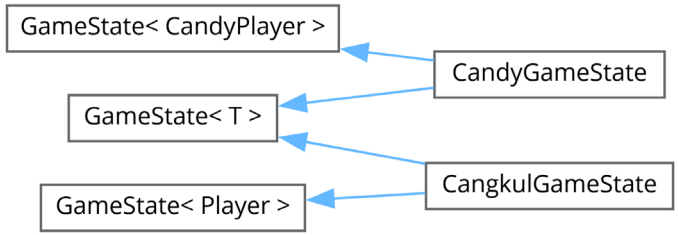
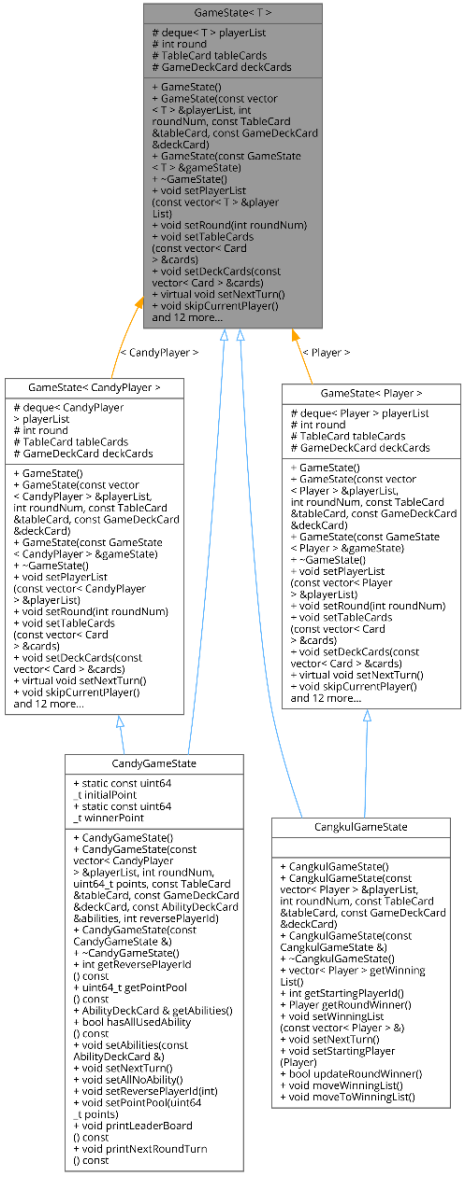


Diagram berikut merupakan versi lebih lengkapnya dan mengikuti standar UML sehingga lebih detail dalam menspesifikasikan atribut-atribut dan *method-method* pada tiap kelasnya.


GameState< T >
deque< T > playerList
int round
TableCard tableCards
GameDeckCard deckCards
+ GameState()
+ GameState(const vector < T > &playerList, int roundNum, const TableCard &tableCard, const GameDeckCard &deckCard)
+ GameState(const GameState < T > &gameState)
+ ~GameState()
+ void setPlayerList (const vector< T > &player List)
+ void setRound(int roundNum)
+ void setTableCards (const vector< Card > &cards)
+ void setDeckCards(const vector< Card > &cards)
+ virtual void setNextTurn()
+ void skipCurrentPlayer()
and 12 more...
< CandyPlayer >
< Player >
GameState< CandyPlayer >
deque< CandyPlayer > playerList
int round
TableCard tableCards
GameDeckCard deckCards
+ GameState()
+ GameState(const vector < CandyPlayer > &playerList, int roundNum, const TableCard &tableCard, const GameDeckCard &deckCard)
+ GameState(const GameState < CandyPlayer > &gameState)
+ ~GameState()
+ void setPlayerList (const vector< CandyPlayer > &playerList)
+ void setRound(int roundNum)
+ void setTableCards (const vector< Card > &cards)
+ void setDeckCards(const vector< Card > &cards)
+ virtual void setNextTurn()
+ void skipCurrentPlayer()
and 12 more...
GameState< Player >
deque< Player > playerList
int round
TableCard tableCards
GameDeckCard deckCards
+ GameState()
+ GameState(const vector < Player > &playerList, int roundNum, const TableCard &tableCard, const GameDeckCard &deckCard)
+ GameState(const GameState < Player > &gameState)
+ ~GameState()
+ void setPlayerList (const vector< Player > &playerList)
+ void setRound(int roundNum)
+ void setTableCards (const vector< Card > &cards)
+ void setDeckCards(const vector< Card > &cards)
+ virtual void setNextTurn()
+ void skipCurrentPlayer()
and 12 more...
CandyGameState
+ static const uint64 _t initialPoint
+ static const uint64 _t winnerPoint
+ CandyGameState()
+ CandyGameState(const vector< CandyPlayer > &playerList, int roundNum, uint64_t points, const TableCard &tableCard, const GameDeckCard &deckCard, const AbilityDeckCard &abilities, int reversePlayerId)
+ CandyGameState(const CandyGameState &)
+ ~CandyGameState()
+ int getReversePlayerId () const
+ uint64_t getPointPool () const
+ AbilityDeckCard & getAbilities()
+ bool hasAllUsedAbility () const
+ void setAbilities(const AbilityDeckCard &)
+ void setNextTurn()
+ void setAllNoAbility()
+ void setReversePlayerId(int)
+ void setPointPool(uint64 _t points)
+ void printLeaderBoard () const
+ void printNextRoundTurn () const
CangkulGameState
+ CangkulGameState()
+ CangkulGameState(const vector< Player > &playerList, int roundNum, const TableCard &tableCard, const GameDeckCard &deckCard)
+ CangkulGameState(const CangkulGameState &)
+ ~CangkulGameState()
+ vector< Player > getWinning List()
+ int getStartingPlayerId()
+ Player getRoundWinner()
+ void setWinningList (const vector< Player > &)
+ void setNextTurn()
+ void setStartingPlayer (Player)
+ bool updateRoundWinner()
+ void moveWinningList()
+ void moveToWinningList()

Selanjutnya, kami mendesain kelas-kelas yang dapat merepresentasikan *deck* dari kartu serta pemain dalam permainan. Kelas abstrak *InventoryHolder* juga dibuat sebagai *generic class* untuk tujuan *reusability,* terutama pada kelas *deckCard* yang dapat menyimpan jenis-jenis kartu yang berbeda. Berikut adalah diagram untuk kelas-kelas tersebut yang kami implementasikan dalam program kami.

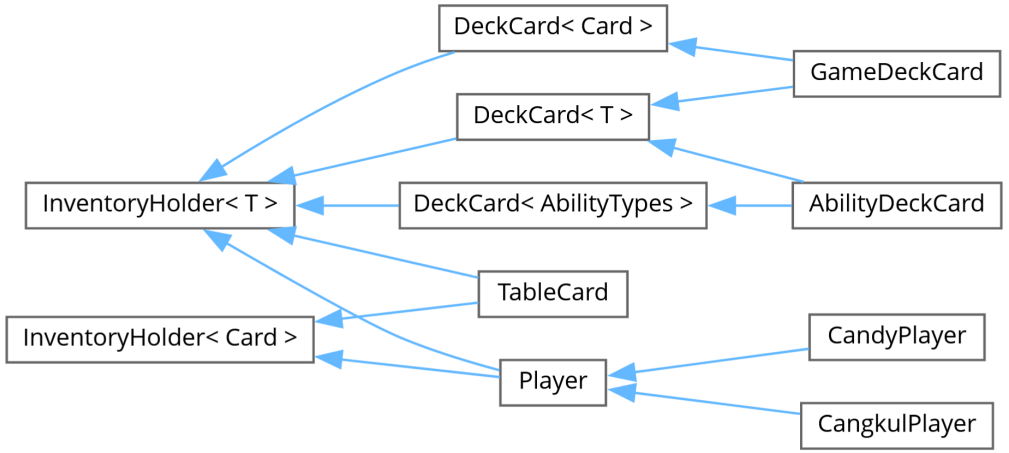
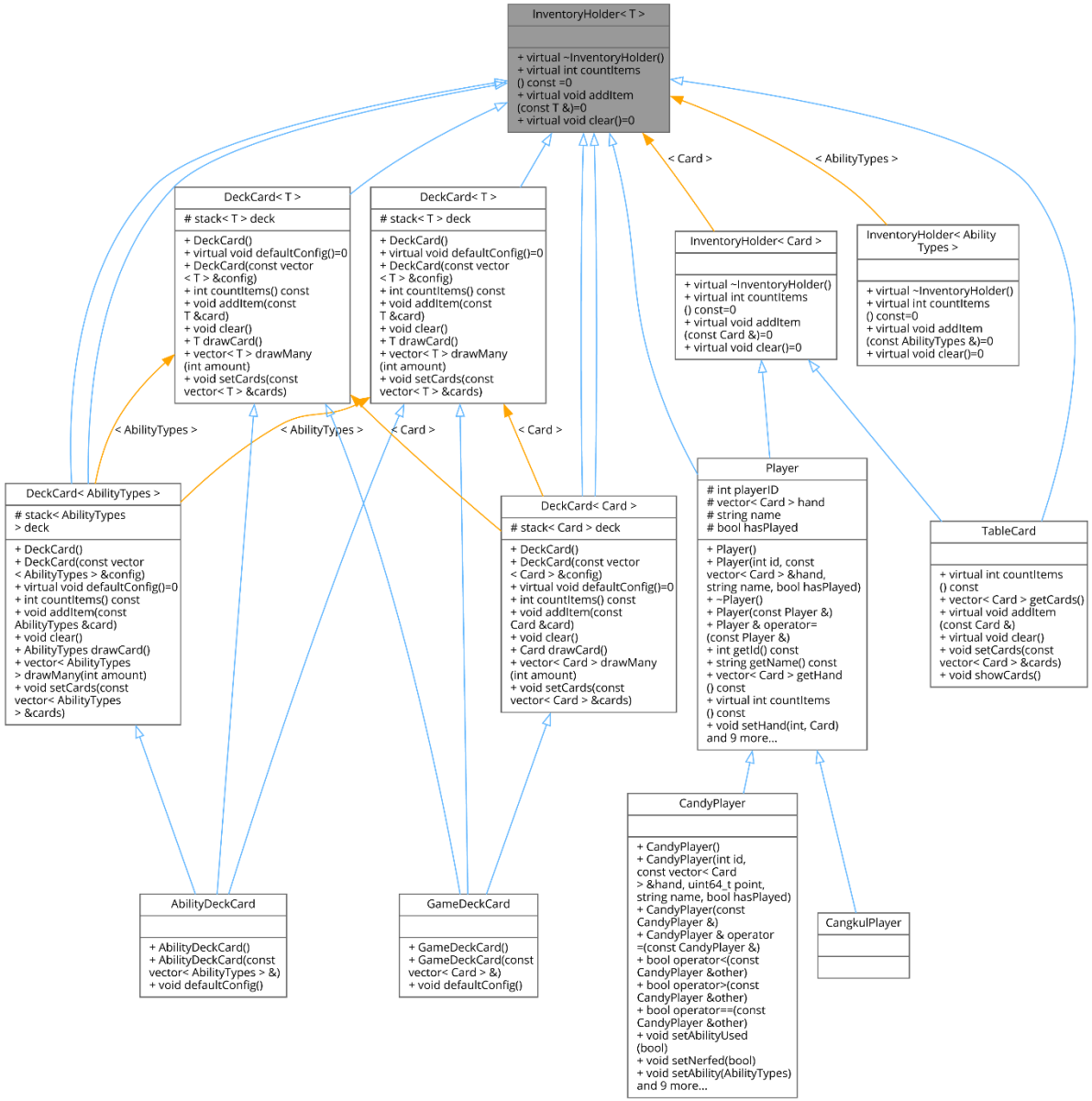
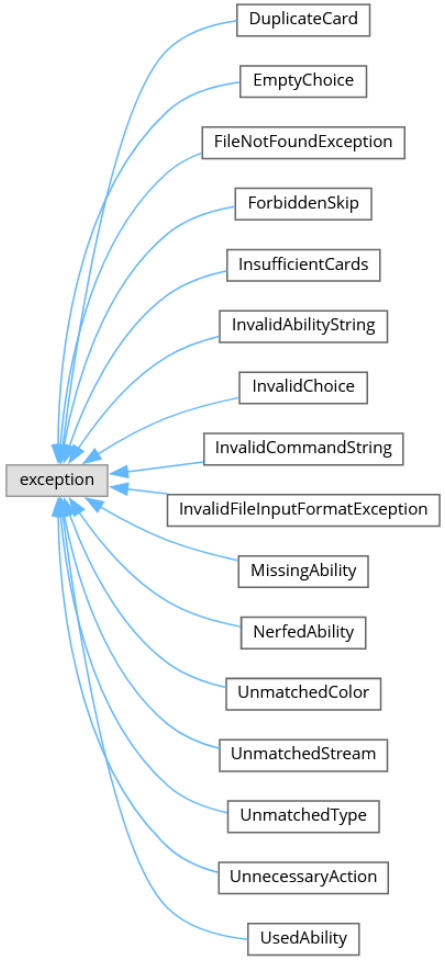


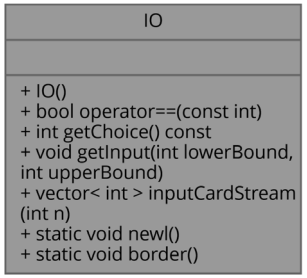
Diagram berikut merupakan versi lebih lengkapnya dan mengikuti standar UML sehingga lebih detail dalam menspesifikasikan atribut-atribut dan *method-method* pada tiap kelasnya.


InventoryHolder< T >
+ virtual ~InventoryHolder()
+ virtual int countItems
() const =0
+ virtual void addItem
(const T &)=0
+ virtual void clear()=0
< Card >
< AbilityTypes >
DeckCard< T >
stack< T > deck
+ DeckCard()
+ virtual void defaultConfig()=0
+ DeckCard(const vector
< T > &config)
+ int countItems() const
+ void addItem(const
T &card)
+ void clear()
+ T drawCard()
+ vector< T > drawMany
(int amount)
+ void setCards(const
vector< T > &cards)
DeckCard< T >
stack< T > deck
+ DeckCard()
+ virtual void defaultConfig()=0
+ DeckCard(const vector
< T > &config)
+ int countItems() const
+ void addItem(const
T &card)
+ void clear()
+ T drawCard()
+ vector< T > drawMany
(int amount)
+ void setCards(const
vector< T > &cards)
InventoryHolder< Card >
+ virtual ~InventoryHolder()
+ virtual int countItems
() const=0
+ virtual void addItem
(const Card &)=0
+ virtual void clear()=0
InventoryHolder< Ability
Types >
+ virtual ~InventoryHolder()
+ virtual int countItems
() const=0
+ virtual void addItem
(const AbilityTypes &)=0
+ virtual void clear()=0
< AbilityTypes >
< AbilityTypes >
< Card >
< Card >
DeckCard< AbilityTypes >
stack< AbilityTypes
> deck
+ DeckCard()
+ DeckCard(const vector
< AbilityTypes > &config)
+ virtual void defaultConfig()=0
+ int countItems() const
+ void addItem(const
AbilityTypes &card)
+ void clear()
+ AbilityTypes drawCard()
+ vector< AbilityTypes
> drawMany(int amount)
+ void setCards(const
vector< AbilityTypes
> &cards)
DeckCard< Card >
stack< Card > deck
+ DeckCard()
+ DeckCard(const vector
< Card > &config)
+ virtual void defaultConfig()=0
+ int countItems() const
+ void addItem(const
Card &card)
+ void clear()
+ Card drawCard()
+ vector< Card > drawMany
(int amount)
+ void setCards(const
vector< Card > &cards)
Player
int playerID
vector< Card > hand
string name
bool hasPlayed
+ Player()
+ Player(int id, const
vector< Card > &hand,
string name, bool hasPlayed)
+ ~Player()
+ Player(const Player &)
+ Player & operator=
(const Player &)
+ int getId() const
+ string getName() const
+ vector< Card > getHand
() const
+ virtual int countItems
() const
+ void setHand(int, Card)
and 9 more...
TableCard
+ virtual int countItems
() const
+ vector< Card > getCards()
+ virtual void addItem
(const Card &)
+ virtual void clear()
+ void setCards(const
vector< Card > &cards)
+ void showCards()
AbilityDeckCard
+ AbilityDeckCard()
+ AbilityDeckCard(const
vector< AbilityTypes > &)
+ void defaultConfig()
GameDeckCard
+ GameDeckCard()
+ GameDeckCard(const
vector< Card > &)
+ void defaultConfig()
CandyPlayer
+ CandyPlayer()
+ CandyPlayer(int id,
const vector< Card
> &hand, uint64_t point,
string name, bool hasPlayed)
+ CandyPlayer(const
CandyPlayer &)
+ CandyPlayer & operator
=(const CandyPlayer &)
+ bool operator<(const
CandyPlayer &other)
+ bool operator>(const
CandyPlayer &other)
+ bool operator==(const
CandyPlayer &other)
+ void setAbilityUsed
(bool)
+ void setNerfed(bool)
+ void setAbility(AbilityTypes)
and 9 more...
CangkulPlayer

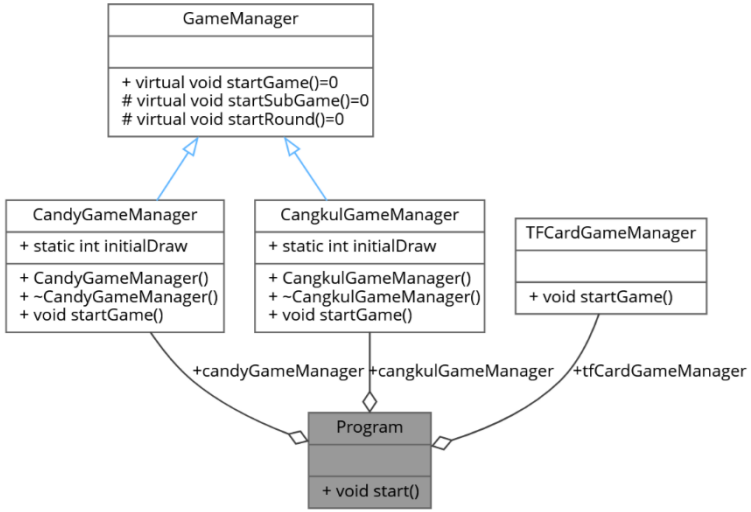
Untuk dapat mengatasi *error* dan situati yang tidak terduga, kami mengimplementasikan berbagai *exception* yang diturunkan dari *base class* `std::exception` bawaan dari bahasa pemrograman C++. *Exception* memungkinkan kami mengatasi berbagai skenario dan situasi yang tidak valid pada bagian yang terpisah dari alur utama program. Berikut adalah diagram untuk kelas-kelas tersebut yang kami implementasikan dalam program kami.

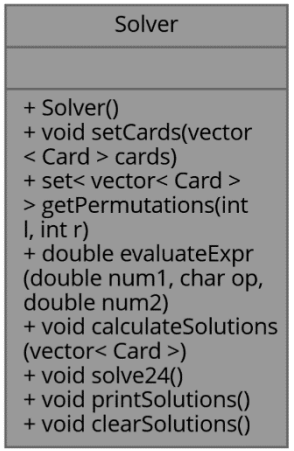


Terlepas dari itu, kami mengimplementasikan beberapa kelas yang tidak terkait dengan kelas-kelas lain. Salah satunya adalah kelas IO. Kelas IO berguna untuk meng-*handle* masukan dan luaran program. Berikut adalah diagram untuk kelas tersebut yang kami implementasikan dalam program kami.



Untuk dapat merepresentasikan program secara keseluruhan, kami mengimplementasikan kelas `Program`. Kelas ini berisi permainan kartu ala kerajaan permen, permainan cangkul, dan permainan *easter egg* (hayoo bagaimana cara memulai permainan *easter egg*?). Permainan *easter egg* memiliki *game manager*-nya sendiri yang diimplementasikan pada kelas `TFGameCardManager`. Selain itu, kami juga mengimplementasikan kelas `Solver` yang berguna pada permainan *easter egg*. Berikut adalah diagram-diagram kelas yang mengilustrasikan kelas-kelas tersebut.





2. Penerapan Konsep OOP

2.1. Inheritance & Polymorphism

Dalam bahasa pemrograman C++, jenis *polymorphism* dibagi menjadi dua: *static polymorphism* dan *dynamic polymorphism*. *Static polymorphism* merupakan jenis *polymorphism* yang ditentukan pada *compile-time*. *Static polymorphism* dicapai dengan menggunakan *function* dan *operator overloading* serta *template* pada program. Di sisi lain, *dynamic polymorphism* merupakan jenis *polymorphism* yang ditentukan pada *run-time*. *Dynamic polymorphism* dicapai dengan menggunakan *virtual function* dan *inheritance*. Pada subbab ini, yang akan dibahas hanyalah mengenai *dynamic polymorphism* sebab bagian *static polymorphism* akan dibahas pada subbab 2.2 dan 2.3.

Pada program kami, *inheritance* dan *virtual function* dapat ditemukan pada implementasi *class-class* yang memiliki atribut dan *method* yang diperlukan pada beberapa *class* berbeda. Sebagai contoh, *class* `Commands` berperan sebagai *parent class* terhadap beberapa *class* lain yang merepresentasikan operasi-operasi yang ada pada permainan kartu ala Kerajaan Permen (Candy Game). Keuntungan implementasi *inheritance* pada *class* `Commands` ini adalah *code reuse* dan *extensibility* sebab implementasi ini cocok dengan *nature* dari *command* yang memiliki banyak jenis dan operasinya masing-masing. Selain itu, *class* `Commands` memiliki *pure virtual method* `executeCommand`. *Method* ini merepresentasikan aksi yang dilakukan ketika suatu *command* dijalankan. Setiap *subclass* yang meng-*inherit class* `Commands` dan merupakan representasi sebuah *command* pada permainan, meng-*override method* `executeCommand` ini. Berikut adalah *header* dari *class* Commands yang menunjukkan *pure virtual function* executeCommand dan beberapa *method* lain.

Commands.hpp

```
class Commands : public BaseCommand<CmdTypes, CandyGameState> {
public:
    Commands();

    virtual ~Commands();

    Commands& operator=(const Commands&);

    virtual void executeCommand(CandyGameState&) = 0;

    static CmdTypes parseCommand(string commandString);

    static string parseCommand(CmdTypes);
```

```
};
```

Selain dari *class* `Commands`, *class* yang juga mengimplementasikan *inheritance* dan *polymorphism* adalah *class* `Ability`. *Class* `Ability` merupakan salah satu *class* yang meng-*inherit class* `Commands`. Kami merasa *distinction* dari *command* yang termasuk ke dalam *ability* dengan *command* yang tidak termasuk *ability* perlu karena diperlukan validasi apakah pemain memiliki kartu *ability* untuk menjalankan *ability* tersebut. Jadi *subclass-subclass* dari *class* `Ability` merepresentasikan *command-command* yang hanya dapat dijalankan jika pemain memiliki kartu *ability* yang berkorespondensi. Berikut adalah *header* dari *class* `Ability` yang menunjukkan *method* untuk memvalidasi apakah sebuah *ability* dapat dijalankan atau tidak (`validateAbility`) dan beberapa *method* lain.

Ability.hpp

```
class Ability : public Commands {
protected:
    AbilityTypes abilityType;

    void validateAbility(CandyGameState& gameState);

public:
    static AbilityTypes parseAbility(string abilityString);

    static string parseAbility(AbilityTypes);
};
```

Dari *subclass-subclass* yang meng-*inherit class* `Ability`, kami memberi *distinction* lagi untuk *ability-ability* yang melibatkan pemain dalam eksekusinya. Kami merasa *distinction* ini perlu karena diperlukan *method* untuk menyeleksi pemain yang terlibat dalam eksekusi *ability* ini. Oleh karena itu, kami mengimplementasikan *class* `PlayerInvolvedAbility` yang *subclass-subclass* nya merepresentasikan *ability-ability* yang melibatkan pemain dalam eksekusinya. *Method* untuk menyeleksi pemain yang saya sebutkan sebelumnya adalah `selectPlayer`. Berikut adalah *header* dari *class* `PlayerInvolvedAbility` yang menunjukkan *method* `selectPlayer`.

PlayerInvolvedAbility.hpp

```
class PlayerInvolvedAbility : public Ability {
protected:
```

```
    int selectPlayer(CandyGameState& gameState, const vector<CandyPlayer>& playerList, string label);
};
```

Selanjutnya, *class* lain yang mengimplementasikan konsep *inheritance* dan *polymorphism* adalah *class* `CardInterface`. *Class* `CardInterface` merupakan sebuah *abstract class* dan berperan sebagai *parent* dari *class* `Card` dan `Combination` yang merepresentasikan kartu dan kombinasi kartu secara berturut-turut. *Inheritance* pada *class* `CardInterface` perlu dilakukan karena dengan meng-*inherit class* ini, *subclass-subclass*-nya dapat memanfaatkan *method-method* telah terimplementasi pada *class* `CardInterface`, khususnya *method-method* untuk *operator overloading*. Dengan meng-*inherit class* `CardInterface`, operasi antarkartu dapat dilakukan tanpa mengimplementasikan *method* untuk operasi lagi. Berikut adalah *header* dari *class* `CardInterface`.

CardInterface.hpp

```
class CardInterface {
   public:
    virtual ~CardInterface();

    // ========== Operators ==========

    bool operator==(const CardInterface&);

    bool operator!=(const CardInterface&);

    bool operator>(const CardInterface&);

    bool operator<(const CardInterface&);

    bool operator>=(const CardInterface&);

    bool operator<=(const CardInterface&);

    // ========== Getters ==========

    long double virtual getValue() const = 0;
};
```

Class Combination merupakan salah satu *subclass* dari *class* CardInterface. *Class* ini merepresentasikan kombinasi dari kartu-kartu yang terdiri dari kartu-kartu di meja (*table cards*) dan kartu-kartu di *hand* pemain. Dalam *class* Combination, diimplementasikan *method-method* untuk menentukan kombinasi kartu terbaik yang dapat di-*construct* dari kartu-kartu yang tersedia. *Class* Combination juga berperan sebagai *superclass* dari *class-class* yang merepresentasikan jenis jenis kombinasi yang tersedia pada permainan, yaitu *high card*, *pair*, *two pair*, *three of a kind*, *straight*, *flush*, *full house*, *four of a kind*, dan *straight flush*. *Inheritance* pada *class* Combination dilakukan karena pada *subclass-subclass*-nya, banyak digunakan *method-method* yang berguna untuk menentukan kombinasi kartu terbaik. Berikut adalah *header* dari *class* Combination.

Combination.hpp

```
/**
 * @class Combination
 * @brief A class representing a combination of cards.
 * @extends CardInterface
 *
 * This class represents a combination of cards, consisting of both table cards and
 * player's hand cards, and provides methods for determining the best possible
 * combination from the available cards.
 */
class Combination : public CardInterface {
   protected:
    const long double HIGHCARD_MAX = 100;
    const long double PAIR_MAX = 100 + HIGHCARD_MAX;
    const long double TWOPAIR_MAX = 100 + PAIR_MAX;
    const long double THREEOFAKIND_MAX = 100 + TWOPAIR_MAX;
    const long double STRAIGHT_MAX = 100 + THREEOFAKIND_MAX;
    const long double FLUSH_MAX = 300 + STRAIGHT_MAX;
    const long double FULLHOUSE_MAX = 100 + FLUSH_MAX;
    const long double FOUROFAKIND_MAX = 100 + FULLHOUSE_MAX;
    const long double STRAIGHTFLUSH_MAX = 100 + FOUROFAKIND_MAX;

    static bool ofSameColor(const vector<Card>& cards);
    static bool ofSameRank(const vector<Card>& cards);
    static bool inSequence(const vector<Card>& cards);
    static bool hasFourOfAKind(const vector<Card>& cards);
    static bool isFullHouse(const vector<Card>& cards);
    static bool hasThreeOfAKind(const vector<Card>& cards);
    static bool isTwoPair(const vector<Card>& cards);
```

```
    static pair<int, int> findPairIdx(const vector<Card>& cards);

  public:
    /**
     * @brief Construct a new Combination object
     * @param combinationCards List of cards in combination
     */
    Combination(const vector<Card>& combinationCards);

    /**
     * @brief Destroy the Combination object
     *
     */
    ~Combination();

    /**
     * @brief Get the Best Combination of cards
     *
     * @return Best card combination
     */
    vector<Card> getCombinationCards() const;

    /**
     * @brief Get string representation of combination type
     *
     * @return Best card combination
     */
    virtual string getComboTypeString() const = 0;

    /**
     * @brief Get the Value of cards
     *
     * @return Card value
     */
    virtual long double getValue() const = 0;
};
```

Berikut merupakan contoh contoh *dynamic polymorphism* yang memanfaatkan *class* `Commands` di atas.

CandyGameManager.cpp

```
CandyGameManager::CandyGameManager() {
    actions = map<CmdTypes, Commands*>{
        {CmdTypes::Double, new class Double()},
        {CmdTypes::Half, new class Half()},
        {CmdTypes::Next, new class Next()},
    };

    abilities = map<AbilityTypes, class Ability*>{
        {AbilityTypes::Abilityless, new class Abilityless()},
        {AbilityTypes::Quadruple, new class Quadruple()},
        {AbilityTypes::Quadruple, new class Quadruple()},
        {AbilityTypes::Quarter, new class Quarter()},
        {AbilityTypes::Reroll, new class Reroll()},
        {AbilityTypes::Reverse, new class Reverse()},
        {AbilityTypes::SwapCard, new class SwapCard()},
        {AbilityTypes::Switch, new class Switch()},
    };
}

Commands* CandyGameManager::getPlayerCommand() {
    // Menerima input aksi pemain saat ini
    IO choiceIO;
    string commandString;
    Commands* command = nullptr;

    do {
        try {
            cout << "Pilihanmu (Contoh: DOUBLE) : ";
            cin >> commandString;
            IO::newl();

            CmdTypes commandType = Commands::parseCommand(commandString);

            if (commandType == CmdTypes::Ability) {
```

```
                command = abilities[Ability::parseAbility(commandString)];
            } else {
                command = actions[commandType];
            }
        } catch (const exception& err) {
            cout << err.what() << endl;
        }
    } while (!command);

    return command;
}
```

Selanjutnya, *class* yang menerapkan konsep *inheritance* dan *polymorphism* adalah *class* `GameManager`. *Class* `GameManager` berperan sebagai pengatur *control flow* dari permainan yang sedang dijalankan. Oleh karena itu, *class* ini memiliki *method-method* yang memulai permainan, *subgame*, dan *round*. Dalam program yang kami buat dan dokumen ini, istilah "permainan" atau "*game*" mengacu kepada satu kali eksekusi permainan sampai ada pemenang dari permainan tersebut. Istilah "*subgame*" mengacu kepada enam *round* pada permainan. Lalu, istilah "*round*" mengacu kepada satu putaran di mana setiap pemain sudah melakukan *turn*-nya masing-masing. *Class* `GameManager` diturunkan ke *class* `CandyGameManager` yang berisi *method-method* tambahan yang berguna dalam me-*manage* permainan. Berikut adalah *header* dari *class* `GameManager`.

GameManager.hpp

```
class GameManager {
protected:
    virtual void startSubGame() = 0;

    virtual void startRound() = 0;

public:
    virtual void startGame() = 0;
};
```

Selain dari *class* `GameManager`, *class* `GameState` juga menerapkan konsep *interitance*. *Class* `GameState` diturunkan menjadi *class* `CandyGameState`. Objek dari kelas ini merepresentasikan suatu instansi keadaan game pada suatu waktu. Berikut adalah *header* dari *class* `GameState`.

GameState.hpp

```
template<class T>
class GameState {
protected:
    deque<T> playerList;

    int round;

    TableCard tableCards;

    GameDeckCard deckCards;

public:
    GameState() {
        // Initialize default values
        playerList = deque<T>();

        // player giliran saat ini selalu indeks 0
        // currentTurnIdx = 0;

        round = 1;
        tableCards = TableCard();
        deckCards = GameDeckCard();
    };

    GameState(const vector<T>& playerList, int roundNum, const TableCard& tableCard, const GameDeckCard& deckCard) {
        this->playerList = deque<T>(playerList.begin(), playerList.end());

        // player giliran saat ini selalu indeks 0
        // currentTurnIdx = currentTurn;
        this->round = roundNum;
        this->tableCards = tableCard;
```

```
        this->deckCards = deckCard;
    };

    GameState(const GameState<T>& gameState) {
        playerList = gameState.playerList;
        round = gameState.round;
        tableCards = gameState.tableCards;
        deckCards = gameState.deckCards;
    };

    ~GameState() {};

    void setPlayerList(const vector<T>& playerList) {
        this->playerList = deque<T>(playerList.begin(), playerList.end());
    };

    void setRound(int roundNum) {
        round = roundNum;
    };

    void setTableCards(const vector<Card>& cards) {
        tableCards.setCards(cards);
    }

    void setDeckCards(const vector<Card>& cards) {
        deckCards.setCards(cards);
    }

    virtual void setNextTurn() {
        T& currentPlayer = getCurrentTurnPlayer();
        currentPlayer.setHasPlayed(true);

        if (hasAllPlayed()) {
            skipCurrentPlayer();
        } else {
            while (playerList[0].hasPlayedThisRound()) {
                skipCurrentPlayer();
            }
```

```
        }
    }

    // sent front player to the back of deque
    void skipCurrentPlayer() {
        T player = playerList[0];
        playerList.pop_front();
        playerList.push_back(player);
    }

    // set all player to has not played
    void setAllNotPlayed() {
        for (long unsigned int i = 0; i < getPlayerList().size(); i++) {
            T& player = getPlayerRefAt(i);
            player.setHasPlayed(false);
        }
    };

    vector<T> getPlayerList() const {
        return vector<T>(playerList.begin(), playerList.end());
    };

    T& getCurrentTurnPlayer() {
        return playerList.front();
    };

    T& getPlayerRefAt(int idx) {
        return playerList[idx];
    }

    int getRound() const {
        return round;
    }

    TableCard& getTableCards() {
        return tableCards;
    }
```

```
    GameDeckCard& getDeckCards() {
        return deckCards;
    }

    bool hasAllPlayed() const {
        for (auto player: playerList) {
            if (!player.hasPlayedThisRound())
                return false;
        }

        return true;
    }

    int getPlayerIdx(int id) const {
        for (long unsigned int i = 0; i < playerList.size(); i++) {
            if (playerList[i].getId() == id) {
                return i;
            }
        }

        return -1;
    }

    void printRemainingTurn() const {

        auto i = playerList.begin() + 1;

        while (i != playerList.end() && i->hasPlayedThisRound()) i++;

        if (i == playerList.end()) {
            cout << "(Tidak ada giliran player lain pada round ini)";
        } else {
            while (i != playerList.end() && !i->hasPlayedThisRound()) {
                cout << "<" << i->getName() << "> ";
                i++;
            }
        }
```

```
        cout << endl;
    }

    void printPlayerList() const {
        for (long unsigned int i = 0; i < playerList.size(); i++) {
            cout << i + 1 << ". " << playerList[i].getName() << endl;
        }
    }

    void printPlayerList(const vector<T>& playerVec) const {
        for (long unsigned int i = 0; i < playerVec.size(); i++) {
            cout << i + 1 << ". " << playerVec[i].getName() << endl;
        }
    }
};
```

2.2. Method/Operator Overloading

Method dan *operator overloading* merupakan bagian dari *static polymorphism*. Dalam program yang kami buat, *operator overloading* dapat ditemukan pada *class* `CandyPlayer`. Implementasi *operator overloading* pada *class* `CandyPlayer` berguna untuk membandingkan nilai poin dari pemain-pemain dalam permainan. Pada kelas tersebut, didefinisikan fungsi operator =, >, dan < sebagai fungsi anggota kelas `CandyPlayer`. Selain pada kelas tersebut, terdapat juga *operator overloading* untuk fungsi operator ==, !=, >, <, >=, dan <= pada kelas `CardInterface` yang memungkinkan perbandingan nilai kartu dan kombinasi dalam permainan. Kelas `Card` juga melakukan *operator overloading* pada operator << yang digunakan dalam mencetak kartu ke konsol. Berikut merupakan implementasi operator yang yang telah disebutkan.

CandyPlayer.cpp

```
CandyPlayer& CandyPlayer::operator=(const CandyPlayer& other) {
    if (this != &other) {
        Player::operator=(other);
        ability = other.ability;
        point = other.point;
        usedAbility = other.usedAbility;
        nerfed = other.nerfed;
```

```
    }

    return *this;
}

bool CandyPlayer::operator<(const CandyPlayer& other) {
    return point < other.point;
}

bool CandyPlayer::operator>(const CandyPlayer& other) {
    return point > other.point;
}
```

CardInterface.cpp

```
bool operator==(const CardInterface& lhs, const CardInterface& rhs) {
    return lhs.getValue() == rhs.getValue();
}

bool operator!=(const CardInterface& lhs, const CardInterface& rhs) {
    return !(lhs == rhs);
}

bool operator>(const CardInterface& lhs, const CardInterface& rhs) {
    return lhs.getValue() > rhs.getValue();
}

bool operator<(const CardInterface& lhs, const CardInterface& rhs) {
    return rhs > lhs;
}

bool operator>=(const CardInterface& lhs, const CardInterface& rhs) {
    return !(lhs < rhs);
}

bool operator<=(const CardInterface& lhs, const CardInterface& rhs) {
    return !(lhs > rhs);
```

```
}
```

Card.hpp

```
/**
     * @brief Operator << overload
     *
     * @return Reference to ostream object
     */
    friend ostream& operator<<(ostream&, const Card&);
```

Method overloading dapat terlihat pada kelas `Command` dan `CangkulCommand`, yaitu pada method `parseCommand`. Method ini dapat digunakan untuk mengubah tipe `enum Command` (ataupun `CangkulCommand`) menjadi *string* ataupun sebaliknya. Selain itu, khusus untuk kelas `Ability` didefinisikan fungsi `parseAbility` dengan tujuan yang sama. Selain yang telah disebutkan, terdapat juga *method overloading* untuk fungsi `switchCards` pada kelas `CandyPlayer`, dengan salah satu fungsi digunakan untuk menukar seluruh kartu dengan player tersebut serta fungsi lainnya digunakan untuk hanya menukar salah satu kartu untuk kedua pemain. Implementasi *method overloading* yang disebutkan dapat dilihat pada *code* berikut.

Command.hpp

```
/**
     * @brief Converts the commands from string to commands type
     *
     * @param abilityString String of ability
     * @return CangkulCmdTypes object
     */
    static CmdTypes parseCommand(string commandString);

    /**
     * @brief Converts commands object to string
     *
     * @return string representation of a commands object
     */
    static string parseCommand(CmdTypes);
```

CangkulCommand.hpp

```
/**
     * @brief Converts the commands from string to commands type
     *
     * @param abilityString String of ability
     * @return CangkulCmdTypes object
     */
    static CangkulCmdTypes parseCommand(string commandString);

    /**
     * @brief Converts commands object to string
     *
     * @return string representation of a commands object
     */
    static string parseCommand(CangkulCmdTypes);
```

Ability.hpp

```
/**
     * @brief Converts the ability from string to AbilityTypes
     *
     * @param abilityString String of ability
     * @return AbilityTypes object
     */
    static AbilityTypes parseAbility(string abilityString);

    /**
     * @brief Converts AbilityTypes object to string
     *
     * @return string representation of an AbilityTypes object
     */
    static string parseAbility(AbilityTypes);
```

2.3. Template & Generic Classes

Template dan *generic classes* merupakan bagian dari *static polymorphism*. Dalam program kami, kami menerapkan konsep ini salah satunya pada *class* `InventoryHolder` beserta *child class*-nya. Keuntungan diimplementasikannya konsep *generic class* pada *class* ini adalah *code reusability* karena `InventoryHolder` beserta *child class*-nya dapat menyimpan atribut berupa kartu biasa ataupun kartu *ability*. Berikut adalah *header* dari *class* `InventoryHolder` dan *child class*-nya yaitu `DeckCard`.

InventoryHolder.hpp

```
template<class T>
class InventoryHolder {
public:
    virtual ~InventoryHolder() {}

    virtual int countItems() const = 0;

    virtual void addItem(const T&) = 0;

    virtual void clear() = 0;
};
```

DeckCard.hpp

```
template<class T>
class DeckCard : public InventoryHolder<T> {
protected:
    stack<T> deck;

public:
    DeckCard() {}

    virtual void defaultConfig() = 0;

    DeckCard(const vector<T>& config) {
        for (int i = config.size() - 1; i >= 0; i--) {
```

```
            deck.push(config[i]);
        }
    }

    int countItems() const {
        return deck.size();
    }

    void addItem(const T& card) {
        deck.push(card);
    }

    void clear() {
        while (!deck.empty()) deck.pop();
    }

    T drawCard() {
        if (deck.empty()) {
            throw InsufficientCards();
        }

        T topCard = deck.top();
        deck.pop();

        return topCard;
    }

    vector<T> drawMany(int amount) {
        vector<Card> drawCards;

        if (this->countItems() < amount) {
            throw InsufficientCards();
        }

        for (int i = 0; i < amount; i++) {
            drawCards.push_back(deck.top());
            deck.pop();
        }
```

```
        return drawCards;
    }

    void setCards(const vector<T>& cards) {
        deck = stack<T>(deque<T>(cards.rbegin(), cards.rend()));
    }
};
```

Lalu, berikut adalah contoh penggunaan dari *generic class* `DeckCard` yaitu ketika menurunkan kelas `AbilityDeckCard` dan `GameDeckCard`. Digunakan *inheritance* pada kedua kelas ini karena implementasi *method* defaultConfig dapat berbeda-beda.

AbilityDeckCard.hpp

```
/**
*   @class AbilityDeckCard
*   @brief Class representing a deck card of ability types.
*   @extends DeckCard
**/
class AbilityDeckCard : public DeckCard<AbilityTypes> {
...
};
```

GameDeckCard.hpp

```
/**
 * @class GameDeckCard
 * @brief Class for the Deck Card of a game
 * @extends DeckCard
 *
 */
class GameDeckCard : public DeckCard<Card> {
...
};
```

Selain *generic class*, kami juga menggunakan konsep *generic function* yang digunakan pada implementasi *method* getMax pada *class* `CandyGameManager`. Keuntungan dari penggunaan konsep ini adalah *code reusability* untuk mendapatkan elemen (pointer ke Card, Combination, dan CandyPlayer) maksimum pada suatu *vector*. Berikut adalah implementasi dari *method* tersebut.

CandyGameManager.hpp

```
/**
 *   @class CandyGameManager
 *   @brief Class representing the game manager for the Candy game.
 *   @extends GameManager
 **/
class CandyGameManager : public GameManager {
   private:
    /**
        @brief Map of available commands for the Candy game.
    **/
    map<CmdTypes, Commands*> actions;

    /**
        @brief Map of available abilities for the Candy game.
    **/
    map<AbilityTypes, class Ability*> abilities;

    /**
        @brief Representing the game state for the Candy game.
    **/
    CandyGameState gameState;

    /**
     *   @brief Method to get the initial player list for the game.
     *   @param[in] playerNum the number of players for the game.
     *   @return vector of CandyPlayer objects representing the initial player list.
     **/
    vector<CandyPlayer> getInitialPlayerList(int playerNum) const;

    /**
     *`   @brief Method to get the player command from the user.
```

```
     *`   @return pointer to a Commands object representing the user's command.
     **/
    Commands* getPlayerCommand();

    /**
     *   @brief Method to get the best possible combination of cards from given lists
     *   @param tableCards list of 5 table cards
     *   @param tableCards list of 2 hand cards
     *   @return Polymorphic Combination pointer of combination type
     **/
    Combination* findComboType(const vector<Card> tableCards, const vector<Card> handCards);

    /**
     *   @brief Method to initialize the deck card for the game.
     **/
    void initiateDeck();

    /**
     *   @brief Method start round of the game.
     **/
    void startRound();

    /**
     *   @brief Method to start a sub-game of the Candy game.
     **/
    void startSubGame();

    /**
     *   @brief Template method to get the object with the highest value from a vector of objects
     *with getValue() method member.
     *   @tparam T the type of objects in the vector.
     *   @param[in] list vector of objects to search for the maximum value.
     *   @return the object with the maximum value.
     **/
    template <class T>
    T getMax(vector<T>& list) {
        T maxElmt = list[0];
```

```
        for (auto i = list.begin() + 1; i != list.end(); i++) {
            if (i->getValue() > maxElmt.getValue())
                maxElmt = *i;
        }

        return maxElmt;
    }

   public:
    /**
     *   @brief Default constructor for CandyGameManager class.
     **/
    CandyGameManager();

    /**
     *   @brief Destructor for CandyGameManager class.
     **/
    ~CandyGameManager();

    // ========== Methods ==========

    /**
     *   @brief Method to start the Candy game.
     **/
    void startGame();

    /**
     *   @brief Integer value representing the number of initial draws for the game.
     **/
    static int initialDraw;
};
```

2.4. Exception

Pada program kami, *exception* diimplementasikan dengan membuat *class-class exception* yang diturunkan dari *class exception* pada STL. CommandException dibuat untuk mengatasi berbagai skenario pengecalian dalam eksekusi aksi pemain yang dimasukkan, terutama aksi *ability.* DeckCardException digunakan untuk mengangani kasus aksi yang tidak valid (mengambil kartu dari deck kosong) dan konfigurasi kartu

yang tidak valid (duplikasi kartu dengan angka dan warna yang sama). Terakhir, IOException digunakan untuk mengatasi kasus yang tidak diinginkan untuk eksekusi program yang berhubungan dengan IO, terutama dalam validasi masukan pemain (konsol maupun *file*). Berikut adalah *exception-exception* yang kami buat beserta penjelasannya.

CommandException.hpp

```
/**
 * @exception Thrown when the player tries to use an ability that has already been used in the current round.
 *
 */
class UsedAbility : public exception {
private:
    AbilityTypes targetType;
    string msg;

public:
    UsedAbility(AbilityTypes targetType) {
        this->targetType = targetType;
        msg = "Maaf, kamu sudah pernah menggunakan ability " + Ability::parseAbility(targetType) + ".";
    }

    const char *what() const throw() {
        return msg.c_str();
    }
};

/**
 * @exception Thrown when current player doesn't have the target ability.
 *
 */
class MissingAbility : public exception {
private:
    AbilityTypes targetType;
    string msg;

public:
    MissingAbility(AbilityTypes targetType) {
```

```
        this->targetType = targetType;
        msg = "Ets, tidak bisa. Kamu tidak punya kartu Ability " + Ability::parseAbility(targetType) + ".";
    }

    const char *what() const throw() {
        // PERLU MEKANISME PRINT ENUM
        return msg.c_str();
    }
};

/**
 * @exception Thrown when a player's ability is nerfed by abilityless.
 *
 */
class NerfedAbility : public exception {
private:
    AbilityTypes targetType;
    string msg;

public:
    NerfedAbility(AbilityTypes targetType) {
        this->targetType = targetType;
        msg = "Oops, kartu ability " + Ability::parseAbility(targetType) + "-mu telah dimatikan sebelumnya:(.";
    }

    const char *what() const throw() {
        return msg.c_str();
    }
};

/**
 * @exception Thrown when an unnecessary action is done
 *
 */
class UnnecessaryAction : public exception {
private:
    CangkulCmdTypes targetType;
    string msg;
```

```
public:
    UnnecessaryAction(CangkulCmdTypes targetType) {
        this->targetType = targetType;
        msg = "Aksi " + CangkulCommand::parseCommand(targetType) +
              " tidak diperlukan! Kamu dapat meletakkan salah satu kartu yang dimiliki.";
    }

    const char *what() const throw() {
        return msg.c_str();
    }
};

/**
 * @exception Thrown when skip is forbidden.
 *
 */
class ForbiddenSkip : public exception {
public:
    const char *what() const throw() {
        return "Aksi SKIP tidak dapat dilakukan karena masih ada kartu yang dapat di-CANGKUL!";
    }
};

/**
 * @exception Thrown when a card color does not match.
 *
 */
class UnmatchedColor : public exception {
public:
    const char *what() const throw() {
        return "Warna kartu yang dipilih tidak cocok dengan kartu table!";
    }
};
```

DeckCardException.hpp

```
/**
```

```
 * @exception Thrown when there are insufficient cards to be drawn.
 *
 */
class InsufficientCards : public exception {
public:
    const char *what() const throw() {
        return "Kartu pada deck tidak cukup!";
    }
};

/**
 * @exception Thrown when there are duplicate cards while reading the configuration file.
 *
 */
class DuplicateCard: public exception {
public:
    const char *what() const throw() {
        return "Ada kartu duplikat dalam deck!";
    }
};
```

IOException.hpp

```
/**
 * @exception Thrown when command string is invalid.
 *
 */
class InvalidCommandString : public exception {
public:
    const char *what() const throw() {
        return "String input bukan command yang valid!";
    }
};

/**
 * @exception Thrown when ability string is invalid.
 *
```

```
 */
class InvalidAbilityString : public exception {
public:
    const char *what() const throw() {
        return "String input bukan ability yang valid!";
    }
};

/**
 * @exception Thrown when input type is invalid.
 *
 */
class UnmatchedType : public exception {
public:
    const char *what() const throw() {
        return "Tipe masukan tidak valid!";
    }
};

/**
 * @exception Thrown when the input is out of range.
 *
 */
class InvalidChoice : public exception {
public:
    const char *what() const throw() {
        return "Nilai pilihan tidak valid!";
    }
};

/**
 * @exception Thrown when input is empty.
 *
 */
class EmptyChoice : public exception {
public:
    const char *what() const throw() {
        return "Tidak ada pilihan!";
```

```
    }
};

class UnmatchedStream : public exception {
public:
    const char *what() const throw() {
        return "Jumlah masukan tidak valid!";
    }
};

/**
 * @exception Thrown when file is not found.
 *
 */
class FileNotFoundException : public exception {
public:
    const char *what() const throw() {
        return "File tidak ditemukan!";
    }
};

/**
 * @exception Thrown when given file format is invalid.
 *
 */
class InvalidFileInputFormatException : public exception {
private:
    string msg;
public:

    InvalidFileInputFormatException(string msg) {
        this->msg = msg;
    }
    const char *what() const throw() {
        return msg.c_str();
    }
};
```

2.5. C++ Standard Template Library

C++ Standard Template Library adalah sebuah *library* kode berisi banyak *class* dan *function* yang telah terdefinisi dan dapat digunakan untuk memudahkan pembuatan suatu program dalam bahasa C++. Dalam program yang telah kami buat, kami cukup banyak menggunakan STL seperti kelas *string* dan *exception* ataupun *container* seperti *map*, *pair*, *vector, deque*, dan *stack*. Selain itu, digunakan juga algoritma yang disediakan STL, salah satunya adalah *random_shuffle.* Berikut adalah beberapa contoh pemakaian STL dalam program yang kami buat.

Combination.cpp (std::pair)

```
pair<int, int> Combination::findPairIdx(vector<Card>& cards) {
    pair<int, int> cardPairIdx(-1, -1);
    for (long unsigned int i = 0; i < cards.size() - 1; i++) {
        if (cards[i].getRank() == cards[i + 1].getRank()) {
            cardPairIdx.first = i - 1;
            cardPairIdx.second = i;
            return cardPairIdx;
        }
    }
    return cardPairIdx;
}
```

Dalam potongan kode di atas, struktur data *pair* dari STL digunakan sebagai *container* data *index* dari 2 card yang merupakan *pair*. Keuntungan dari penggunaan struktur data *pair* ini adalah bersifat lebih natural dan lebih aman karena fungsi memang hanya mencari kombinasi 2 kartu, tidak lebih dan tidak kurang.

Card.cpp (std::map)

```
string Card::getRankString() const {
    map<Rank, string> rankMap = {
        {Rank::One, "As"},        {Rank::Two, "Two"},     {Rank::Three, "Three"},
        {Rank::Four, "Four"},     {Rank::Five, "Five"},   {Rank::Six, "Six"},
        {Rank::Seven, "Seven"},   {Rank::Eight, "Eight"}, {Rank::Nine, "Nine"},
        {Rank::Ten, "Ten"},       {Rank::Eleven, "Jack"}, {Rank::Twelve, "Queen"},
        {Rank::Thirteen, "King"},
    };
```

```
    return rankMap[rank];
}
```

Dalam potongan kode di atas, struktur data *map* dari STL digunakan untuk memetakan data *rank* suatu *card* ke *string* yang merepresentasikan *rank card* tersebut. Keuntungan dari penggunaan struktur data *map* dalam kasus ini adalah membuat implementasi lebih simpel dan elegan. Penggunaan *map* disini juga membuat implementasi lebih *object-oriented* karena di prosedural hal ini umumnya diimplementasikan menggunakan banyak *if-else*.

TableCard.hpp (std::vector)

```
/**
 * @class TableCard
 * @brief Class that defines the cards on the table
 * @extends InventoryHolder
 *
 */
class TableCard : public InventoryHolder<Card> {
private:
    /**
     * @brief Vector of cards on the table
     *
     */
    std::vector<Card> cards;

public:
    /**
     * @brief Returns the number of items in the inventory.
     *
     * @return the Number of items in the inventory
     */
    virtual int countItems() const;

    /**
     * @brief Get all of the cards on the table that is relevant to the current game state
     *
```

```
     * @return a vector of cards on the table
     */
    vector<Card> getCards();

    /**
     * @brief Adds an item to the inventory
     *
     */
    virtual void addItem(const Card&);

    /**
     * @brief Removes all of the items in the inventory
     *
     */
    virtual void clear();

    /**
     * @brief Set the table cards
     *
     * @param cards List of cards representing table cards to be set
     */
    void setCards(const vector<Card>& cards);

    /**
     * @brief Shows all of the table cards
     *
     */
    void showCards();
};
```

Dalam potongan kode di atas, struktur data *vector* dari STL digunakan sebagai *container table cards*. Keuntungan dari penggunaan *vector* pada kasus ini adalah *vector* memiliki sifat seperti *array* namun dinamis. Hal ini sangat cocok dengan kebutuhan untuk *table card* karena *table card* awalnya kosong lalu diisi satu persatu seiring pergantian *round* serta setiap pemain dapat melihat seluruh kartu yang ada pada *table card* (membutuhkan akses berdasarkan indeks).

Gamestate.hpp (std::deque)

```
/**
 * @class GameState
 * @brief Class that holds the state of a game.
 *
 * @tparam T Element type of the players in the game.
 */
template<class T>
class GameState {
protected:
    /**
     * @brief Deque that contains the list of players
     *
     */
    deque<T> playerList;
}
```

Dalam potongan *code* di atas, kelas GameState memiliki atribut *playerList* menggunakan *deque*. Penggunaan *container* ini mempermudah proses pergantian giliran saat permainan berjalan, yaitu dengan melakukan *pop* pada elemen terdepan dan memasukkannya kembali ke belakang *deque* (sistem *round robin*). Alasan tidak digunakan *queue* sebagai pengganti *deque* adalah program masih perlu mengakses pemain di dalamnya menggunakan indeks untuk keperluan tertentu, salah satunya adalah melakukan pencarian pemain dengan poin tertinggi.

DeckCard.hpp (std::stack)

```
template<class T>
class DeckCard : public InventoryHolder<T> {
protected:
    /**
     * @brief Stack that holds the deck
     *
     */
    stack<T> deck;
}
```

Stack digunakan pada kelas ini agar kartu pada *deck* tidak dapat diakses kecuali dengan melakukan *draw* pada kartu teratas. Hal ini sesuai dengan kondisi objek *deck card* pada permainan dunia nyata.

GameDeckCard.cpp (std::random_shuffle)

```
void GameDeckCard::defaultConfig() {
    vector<Card> tempCards;

    for (int i = 0; i < 4; i++) {
        for (int j = 1; j <= 13; j++) {
            tempCards.push_back(Card((Color) i, (Rank) j));
        }
    }

    srand(time(NULL));

    random_shuffle(tempCards.begin(), tempCards.end());

    temp = vector<Card> (tempCards.rbegin(), tempCards.rend()); // testing
    clear();
    // for (auto card: tempCards) std::cout << card << std::endl;
    for (Card card: tempCards) {
        deck.push(card);
    }
}
```

Fungsi *random_shuffle* digunakan dalam mempermudah melakukan pengacakan kartu pada *deck*.

2.6. Konsep OOP Lain

Konsep OOP lain yang kami terapkan pada program kami adalah *Abstract Base Class* dan *Composition*. Konsep *Abstract Base Class* kami terapkan salah satunya pada *class* `Commands`. Berikut adalah *header* dari kelas tersebut.

```
/**
 * @class Commands
 * @brief Class that defines the behaviour of in-game commands
 * @extends BaseCommand
 *
 */
class Commands : public BaseCommand<CmdTypes, CandyGameState> {
public:
    /**
     * @brief Construct a new Commands object
     *
     */
    Commands();

    /**
     * @brief Destroy the Commands object
     *
     */
    virtual ~Commands();

    /**
     * @brief Assignment operator for commands object
     *
     * @return Reference to a command to be assigned to
     */
    Commands& operator=(const Commands&);

    /**
     * @brief Executes command
     *
     * @param gameState State of the game to be applied to
     */
    virtual void executeCommand(CandyGameState&) = 0;

    /**
     * @brief Converts the commands from string to commands type
     *
     * @param abilityString String of ability
```

```
     * @return CangkulCmdTypes object
     */
    static CmdTypes parseCommand(string commandString);

    /**
     * @brief Converts commands object to string
     *
     * @return string representation of a commands object
     */
    static string parseCommand(CmdTypes);
};
```

Selanjutnya, konsep *Composition* kami terapkan salah satunya pada *class* `CandyGameManager` yang memiliki atribut *gameManager* yang merupakan instansiasi dari *class* `CandyGameState`. Berikut adalah *header* dari kelas tersebut.

CandyGameManager.hpp

```
/**
 *   @class CandyGameManager
 *   @brief Class representing the game manager for the Candy game.
 *   @extends GameManager
 **/
class CandyGameManager : public GameManager {
   private:
    /**
        @brief Map of available commands for the Candy game.
    **/
    map<CmdTypes, Commands*> actions;

    /**
        @brief Map of available abilities for the Candy game.
    **/
    map<AbilityTypes, class Ability*> abilities;

    /**
        @brief Representing the game state for the Candy game.
```

```
    **/
    CandyGameState gameState;
}
```

3. Bonus Yang Dikerjakan

3.1. Bonus yang Diusulkan oleh Spek

3.1.1. Generic Class

Sesuai yang disampaikan di spesifikasi, bonus ini adalah membuat *generic class* dan *generic function*, bukan salah satu. Dalam program kami, kami telah menggunakan kedua konsep tersebut sebagaimana yang telah disampaikan di laporan ini bagian 2.3 yaitu implementasi *generic class* pada *class* `InventoryHolder` beserta *child*-nya dan *generic function* pada fungsi getMax dalam *class* `CandyGameManager`.

3.1.2. Game Kartu Lain

Untuk bonus membuat game kartu lain, kami memutuskan untuk membuat game cangkul. Game cangkul sendiri adalah permainan kartu yang dilakukan oleh dua hingga empat orang dengan pemenangnya adalah pemain yang berhasil menghabiskan kartunya. Secara singkat, mekanisme permainan ini adalah dilakukan berkali kali putaran dimana setiap putaran pemain harus mengeluarkan kartu dengan simbol (pada program kami menggunakan warna) yang sama dengan kartu yang sudah berada di *table*. Kartu pertama di *table* diambil dari kartu teratas pada *deck* untuk urutan pertama dan kartu yang dikeluarkan pemenang putaran sebelumnya untuk urutan selanjutnya. Untuk penjelasan lebih detailnya bisa dilihat di *website* ini:

https://gamerhandal.wordpress.com/2018/11/27/bermain-kartu-101-apa-itu-cangkulan-dan-bagaimana-cara-memainkannya/

Dalam pembuatan permainan ini, kami membuat beberapa *class* tambahan yaitu `CangkulGameManager`, `CangkulGameState`, `CangkulCommand`, `Put`, `Skip`, dan `Cangkul`. Semua kelas ini diturunkan dari kelas yang sudah ada. Berikut adalah beberapa contoh tampilan jalannya permainan cangkul.

```
INISIASI PERMAINAN CANGKUL


Pilihan Jumlah Pemain :
1. Dua Pemain
2. Tiga Pemain
3. Empat Pemain
Pilihan : 2

Masukkan nama pemain 1 : Far
Masukkan nama pemain 2 : Jo
Masukkan nama pemain 3 : Didi

PERMAINAN BARU DIMULAI!

ROUND 1

Satu kartu diletakkan ke meja : 3 (Kuning)
```

```
Giliran pemain 1 : Far

Berikut kartu pada table :
[3 (Kuning)]

Berikut kartu yang kamu miliki :
[7 (Merah), 10 (Biru), 1 (Biru), 12 (Hijau), 11 (Biru), 13 (Biru), 13 (Hijau)]

Pilihan perintah:
1. PUT
2. CANGKUL
3. SKIP

Pilihanmu (Contoh: PUT) : CANGKUL

Pemain melakukan cangkul!
Banyak kartu deck awal :30
Banyak kartu deck setelah cangkul :29
Pemain mendapatkan kartu 6 (Merah)
==============================

Giliran pemain 1 : Far

Berikut kartu pada table :
[3 (Kuning)]

Berikut kartu yang kamu miliki :
[7 (Merah), 10 (Biru), 1 (Biru), 12 (Hijau), 11 (Biru), 13 (Biru), 13 (Hijau), 6 (Merah)]
```

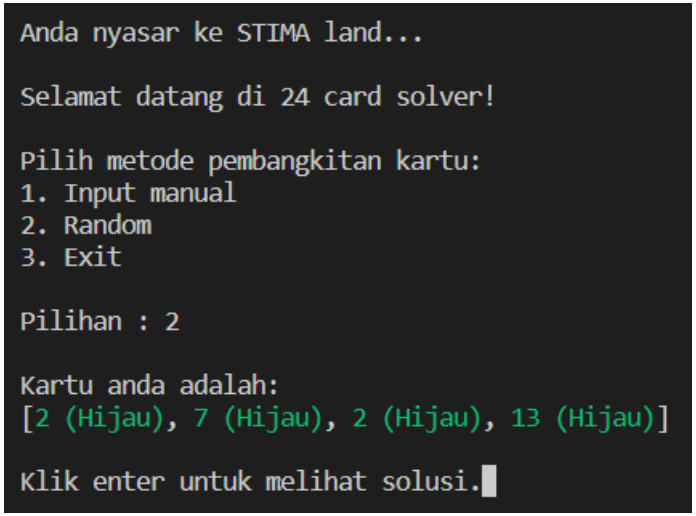
```
Giliran pemain 2 : Jo

Berikut kartu pada table :
[3 (Kuning), 5 (Kuning)]

Berikut kartu yang kamu miliki :
[5 (Merah), 2 (Hijau), 10 (Kuning), 8 (Hijau), 6 (Hijau), 10 (Hijau), 4 (Kuning)]

Pilihan perintah:
1. PUT
2. CANGKUL
3. SKIP

Pilihanmu (Contoh: PUT) : PUT

Pilih kartu yang ingin diletakkan :
1. 5 (Merah)
2. 2 (Hijau)
3. 10 (Kuning)
4. 8 (Hijau)
5. 6 (Hijau)
6. 10 (Hijau)
7. 4 (Kuning)
Pilihan : 3
Kartu berhasil diletakkan! Giliran pemain dilanjutkan.
```

```
Game Selesai.
Berikut pemenang game ini:
1. Far
2. Jo
```

3.2. Bonus Kreasi Mandiri

a. Easter Egg Solver Game 24

Kami membuat easter egg berupa 24 game solver yang hanya bisa diakses dengan input khusus pada waktu tertentu. 24 Game Solver ini bisa menerima masukan 4 kartu dari *user* maupun men-*generate* sendiri keempat kartu lalu program akan mencari operasi aritmetika yang dapat membuat nilai 4 kartu tersebut berjumlah 24. Silakan cari cara akses easter egg kami 😊.



```
70. 13 + (2 + (7 + 2))
71. ((13 + 7) + 2) + 2
72. (13 + (2 + 2)) + 2
73. (13 + 7) + (2 + 2)
74. 13 + ((7 + 2) + 2)
75. 13 + (7 + (2 + 2))
76. (13 + 7) + (2 * 2)
77. 13 + (7 + (2 * 2))
78. (13 - 7) * (2 + 2)
79. ((13 - 7) * 2) * 2
80. (13 - 7) * (2 * 2)

80 solusi unik ditemukan.
```

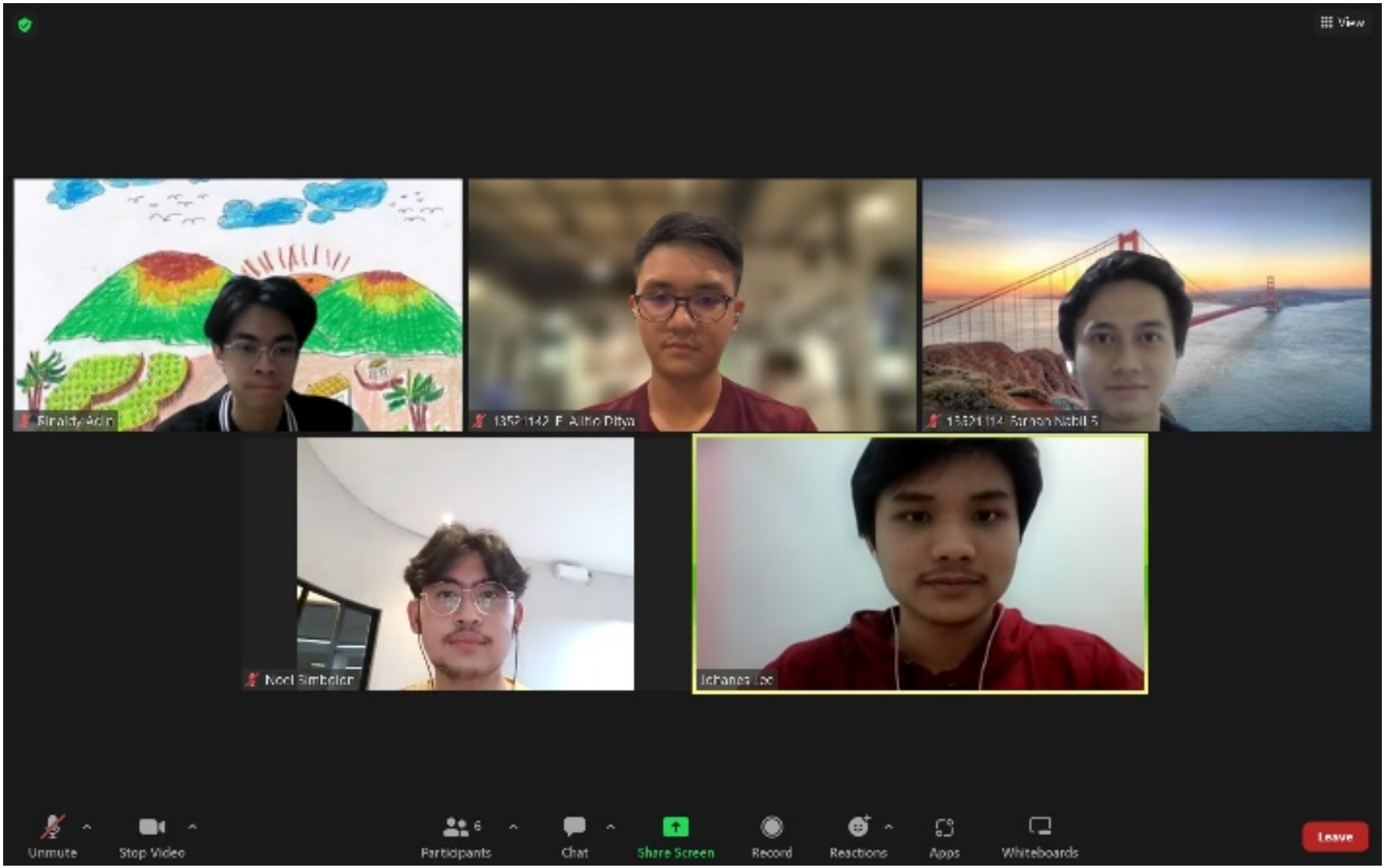
```
Pilih metode pembangkitan kartu:
1. Input manual
2. Random
3. Exit

Pilihan : 1

Masukkan empat muka kartu terpisah oleh spasi.

Input: A 10 J 6

Kartu anda adalah:
[1 (Hijau), 10 (Hijau), 11 (Hijau), 6 (Hijau)]

Klik enter untuk melihat solusi.

Tidak ada solusi yang ditemukan.
```

b. Warna Kartu pada Terminal

Kami membuat output kartu dengan warna yang sesuai dengan warna dari kartu tersebut.

4. Pembagian Tugas

Modul (dalam poin spek)	**Implementer**	**Tester**
Card (berisi kelas abstrak *CardInterface* dengan kelas turunan kartu dan berbagai kombinasi)	13521134	13521134
GameEnvironment (InventoryHolder [diturunkan menjadi Player, DeckCard, dan TableCard], GameState, GameManager, serta turunan-turunannya)	13521148, 13521142, 13521114	13521096, 13521148, 13521142, 13521114
Commands (*BaseCommand* dan berbagai turunannya)	13521096, 13521142¸13521148	13521142, 13521096, 13521148,
EasterEgg	13521142	13521142, 13521114
Program (kelas IO dan Program)	13521142, 13521148	13521096, 13521148, 13521142, 13521114


View
Unmute
Stop Video
Participants
6
Chat
Share Screen
Record
Reactions
Apps
Whiteboards
Leave

LAMPIRAN

Formulir Asistensi

Kode Kelompok : KYS
Nama Kelompok : Suicide Squad
1. 13521096 / Noel Christoffel Simbolon
2. 13521114 / Farhan Nabil Suryono
3. 13521134 / Rinaldy Adin
4. 13521142 / Enrique Alifio Ditya
5. 13521148 / Johanes Lee

Asisten Pembimbing : Fabian Savero Diaz Pranoto

1. Konten Diskusi

- **Pertanyaan**: Mengapa perhitungan *string length* di Windows dan Linux ketika membaca suatu string pada file berbeda?
  Jawaban: Kemungkinan disebabkan oleh perbedaan cara handle *newline characters* ketika membaca file pada masing-masing OS yang berbeda.

- **Pertanyaan**: Apakah program juga perlu dapat membaca urutan *ability* yang akan dibagikan?
  Jawaban: Tidak perlu. Program cukup dapat membaca urutan *deck* kartu dari *file*.

- **Pertanyaan**: Apakah implementasi *method member* dari *generic class* boleh diletakkan di *header file* untuk menghindari *linking error*?
  Jawaban: Boleh.

- **Pertanyaan**: Apakah implementasi kelas kombinasi yang mengihitung value di awal sudah benar?
  Jawaban: Sebaiknya setiap jenis kombinasi dijadikan kelas sendiri dengan setiap kelas menghitung value-nya secara masing-masing karena cara sekarang yang menggunakan if else yang banyak masih kurang sesuai dengan OOP. Mungkin bisa dilakukan dengan memindahkan algoritma evaluasi jenis kartu menjadi di luar kelas.

- **Pertanyaan**: Apakah di permainan ini bisa terjadi split pot? karena di peraturan texas holdem bisa terjadi seri sehingga ada split pot.
  Jawaban: Tidak ada split pot/seri sehingga setiap permainan pasti dapat ditemukan pemenangnya. Untuk aturan kombinasinya boleh ditentukan sendiri untuk kombinasi yang seri jika dalam permainan holdem.

2. Tindak Lanjut

- Melakukan perubahan struktur objek untuk kombinasi kartu agar setiap jenis kombinasi dijadikan kelasnya sendiri.
- Melanjutkan *debugging* dan *finishing* keseluruhan program.

Tautan *Remote Repository*

Gitub : https://github.com/AlifioDitya/TubesOOP_KYS