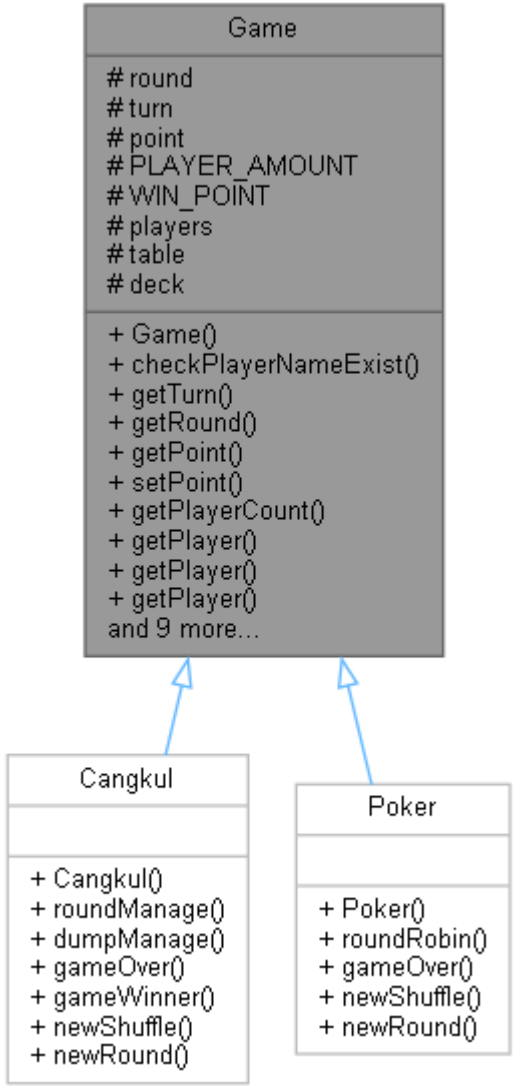
Kode Kelompok : SUS
Nama Kelompok : When The Object

1. 13521082 / Farizki Kurniawan
2. 13521092 / Frankie Huang
3. 13521138 / Johann Christian Kandani
4. 13521150 / I Putu Bakta Hari Sudewa
5. 13521170 / Haziq Abiyyu Mahdy

Asisten Pembimbing : Widya Anugrah Putra

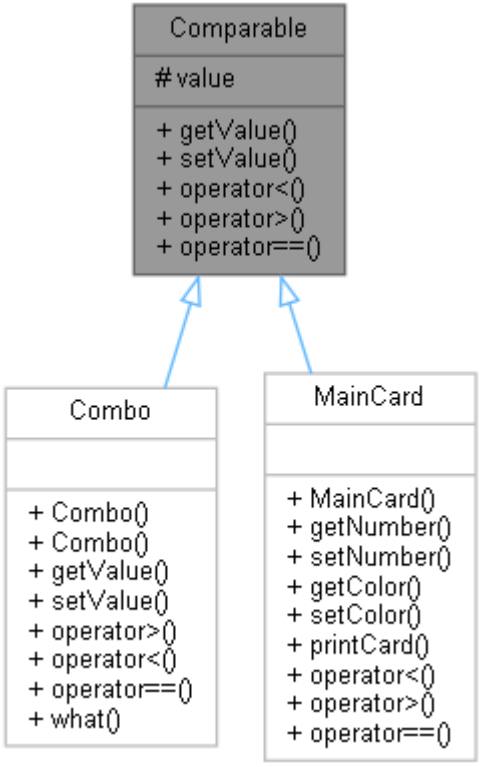
1. Diagram Kelas



a. Game, Poker, dan Cangkul

Kelas Game menerapkan konsep abstract base class, inheritance, dan polymorphism. Kelas ini merupakan abstract base class yang memiliki beberapa pure virtual function, yaitu `newShuffle()` dan `newRound()`, dimana fungsi ini akan diimplementasikan sesuai dengan kebutuhan anak-anaknya. Selain itu, inheritance digunakan untuk membuat kelas yang lebih spesifik, yaitu **kelas `Poker`** dan **kelas `Cangkul`**. Polymorphism digunakan beserta dengan konsepsi abstract base class sebelumnya, dimana digunakan konsep ini agar objek yang diturunkan dari `Game` dapat digeneralisasi dan diperlakukan secara serupa.

Kelebihan dari desain kelas seperti ini adalah kita dapat membentuk kelas untuk permainan kartu yang serupa tanpa mengimplementasi isi-isinya dari awal. Kekurangan dari desain kelas ini adalah sebagai kelas abstrak, diperlukan perhatian ekstra dalam pengimplementasian kelas atau fungsi yang berkaitan dengan kelas ini, dimana kita tidak boleh menginstansiasikan objek dari kelas `Game` secara langsung, melainkan menggunakan reference ataupun pointer dalam penggunaannya.

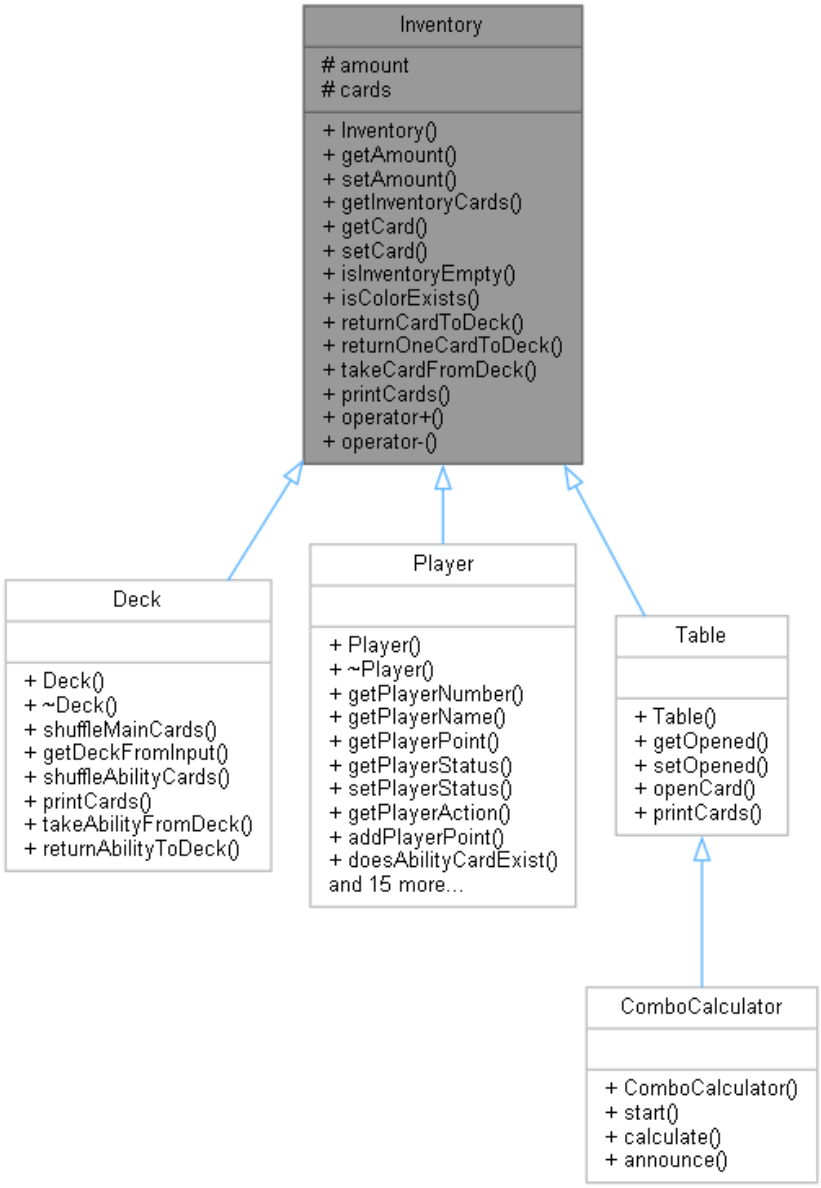


b. Comparable, Player, MainCard, dan Combo

Kelas `Comparable` menerapkan konsep abstract base class, inheritance, polymorphism, dan operator overloading. Kelas ini merupakan abstract base class dengan beberapa pure virtual function yang dapat diimplementasikan sesuai dengan kebutuhan dari anak-anaknya. Selain itu, inheritance digunakan untuk membuat kelas yang memiliki kebutuhan yang serupa secara logis, yaitu **kelas Player, kelas MainCard** dan **kelas Combo**. Polymorphism digunakan agar objek yang diturunkan dari Comparable dapat digeneralisasi dan diperlakukan dengan serupa. Terakhir, ada operator overloading berupa overloading operator <, =, dan > dimana operator ini digunakan untuk membandingkan antara dua instansi dari kelas Comparable.

Kelebihan dari desain kelas seperti ini adalah kelas yang memiliki kesamaan secara logis (memiliki value tertentu, dapat dibandingkan antara satu sama lain) dapat menurunkan secara langsung kelas ini tanpa membuat atribut atau method yang diperlukan dari awal. Kekurangan dari desain kelas seperti ini adalah sebagai kelas abstrak, diperlukan perhatian ekstra dalam

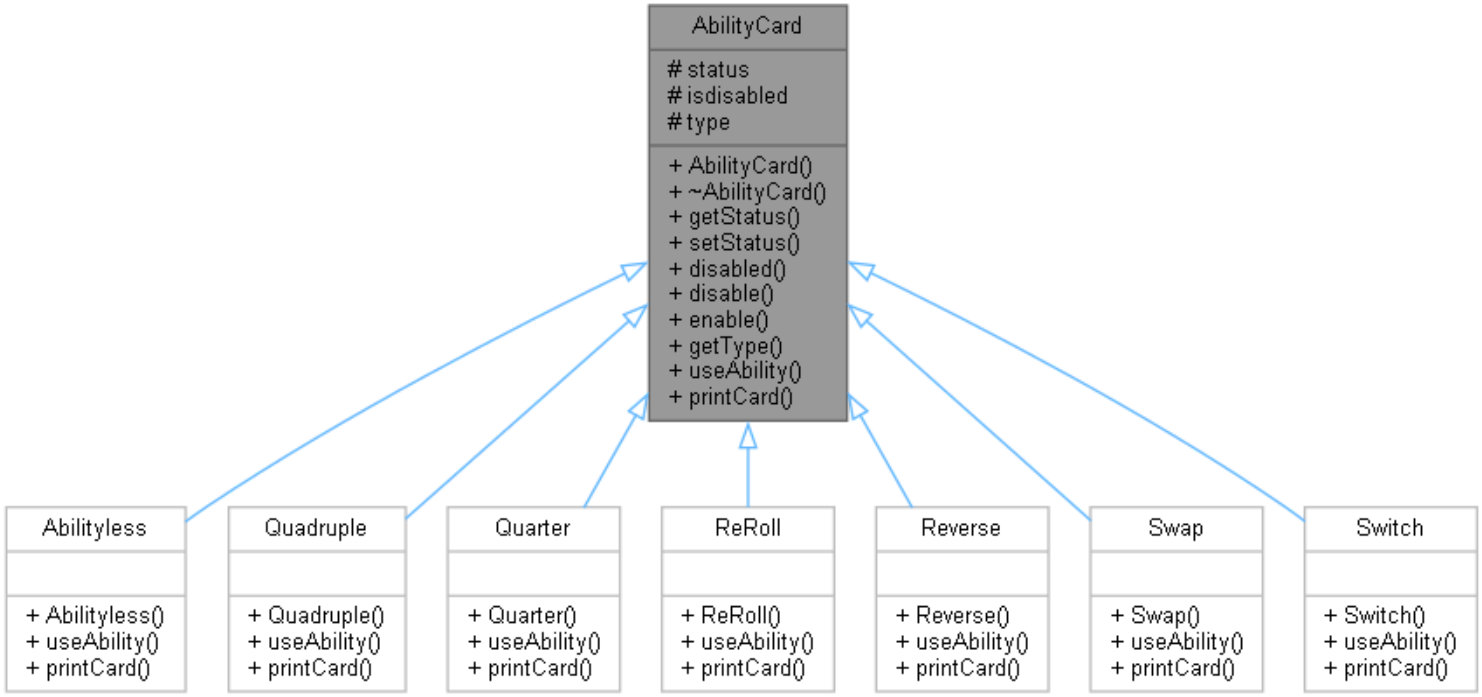
pengimplementasian kelas atau fungsi yang berkaitan dengan kelas ini, dimana kita tidak boleh menginstansiasikan objek dari kelas Comparable secara langsung, melainkan menggunakan reference ataupun pointer dalam penggunaannya.


Inventory
amount
cards
+ Inventory()
+ getAmount()
+ setAmount()
+ getInventoryCards()
+ getCard()
+ setCard()
+ isInventoryEmpty()
+ isColorExists()
+ returnCardToDeck()
+ returnOneCardToDeck()
+ takeCardFromDeck()
+ printCards()
+ operator+()
+ operator-()
Deck
+ Deck()
+ ~Deck()
+ shuffleMainCards()
+ getDeckFromInput()
+ shuffleAbilityCards()
+ printCards()
+ takeAbilityFromDeck()
+ returnAbilityToDeck()
Player
+ Player()
+ ~Player()
+ getPlayerNumber()
+ getPlayerName()
+ getPlayerPoint()
+ getPlayerStatus()
+ setPlayerStatus()
+ getPlayerAction()
+ addPlayerPoint()
+ doesAbilityCardExist()
and 15 more...
Table
+ Table()
+ getOpened()
+ setOpened()
+ openCard()
+ printCards()
ComboCalculator
+ ComboCalculator()
+ start()
+ calculate()
+ announce()

c. Inventory, Table, Deck, dan Player

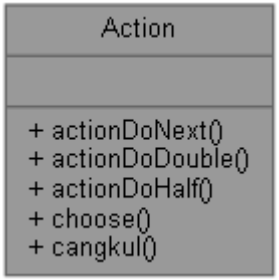
Kelas Inventory menerapkan konsep STL vector, abstract base class, inheritance, polymorphism, dan operator overloading. Kelas ini memiliki atribut cards yang merupakan vector of MainCard, yaitu kumpulan kartu-kartu yang dimiliki. Kelas ini merupakan abstract base class dengan pure virtual function printCards() yang dapat diimplementasikan oleh anak-anaknya sesuai dengan bagaimana anaknya perlu mengoutput kartu yang dimiliki. Selain itu, inheritance digunakan untuk membuat kelas yang memiliki kebutuhan penyimpanan dan pengelolaan kartu, yaitu **kelas Table, kelas Deck** dan **kelas Player**. Polymorphism digunakan agar objek yang diturunkan dari Inventory dapat digeneralisasi dan diperlakukan dengan serupa. Terakhir, ada operator overloading berupa overloading operator + dan - dimana operator ini digunakan untuk menambahkan atau mengurangi kartu pada suatu instansi dari kelas Inventory.

Kelebihan dari desain kelas seperti ini adalah kelas-kelas yang membutuhkan penyimpanan dan pengelolaan kartu dapat langsung menurunkan atribut dan metode yang diperlukan dari kelas Inventory. Selain itu, penggunaan std::vector juga mempermudah implementasi dari penambahan dan pengurangan kartu yang diperlukan.



d. Ability Card

Untuk implementasi *ability card*, kami membuat sebuah kelas abstrak bernama `AbilityCard` dan kelas-kelas ability lainnya yang berupa turunan dari `AbilityCard`. Masing-masing kelas turunan tersebut memiliki implementasi metode `useAbility()` yang berbeda-beda. Bentuk implementasi ini kami pilih karena setiap kelas mempunyai metode yang hampir serupa, yang berbeda hanyalah di metode `useAbility()` dan `printCard()` saja. Namun, implementasi ini memiliki kekurangan yaitu bisa terjadinya *recursive include* akibat Player yang memiliki variable AbilityCard dan AbilityCard yang mengubah atribut `Player` lain. Kekurangan ini kami perbaiki dengan mengubah parameter yang diberikan menjadi `Game`.



e. Action

Action diimplementasikan sebagai sebuah kelas yang berdiri sendiri. Kelas ini berfungsi untuk mengubah flow pada game dan akan dikomposisi oleh Player.

2. Penerapan Konsep OOP

2.1. Inheritance & Polymorphism

Konsep *inheritance* dan *polymorphism* digunakan pada kelas `AbilityCard` yang memiliki kelas turunan berupa `Abilityless`, `Quadruple`, `Quarter`, `ReRoll`, `Reverse`, `Swap`, dan `Switch`. Kelas-kelas tersebut dibuat menjadi kelas turunan `AbilityCard` agar kelas-kelas tersebut dapat mewarisi atribut dan *method* kelas `AbilityCard` dan tiap kelas turunan dapat memiliki implementasi masing-masing pada *method* `useAbility()` dan `printCard()`. Berikut adalah cuplikan kode dari kelas `AbilityCard` dan salah satu turunannya, yaitu `Abilityless`.

```
// FILE: AbilityCard.hpp
class AbilityCard {
protected:
    bool status;
```

```
public:
    bool getStatus() const;
    void setStatus(bool);
    virtual void useAbility(Game&);
    virtual void printCard();
};

// FILE: Abilityless.hpp
class Abilityless : public AbilityCard {
public:
    void useAbility(Game&);
    void printCard();
};
```

Dengan adanya kelas turunan dari `AbilityCard`, kita dapat memasukkan *pointer* `AbilityCard` sebagai atribut dari kelas `Player`, sehingga *pointer* tersebut dapat di-*assign* dengan alamat dari seluruh jenis turunan `AbilityCard`. Hal ini memungkinkan setiap pemain memiliki `AbilityCard` yang berbeda-beda, misalnya pemain 1 memiliki `Abilityless`, pemain 2 memiliki `Switch`, dan seterusnya.

```
//FILE: Player.hpp
class Player : public Inventory{
private:
    string name;
    int number;
    int point;

    AbilityCard* ability;
    Action playerAction;
// ...
};
```

Selain itu, konsep *inheritance* juga terdapat pada kelas `Player` yang merupakan turunan dari kelas `Inventory` dan kelas karena pada dasarnya `Player` merupakan sebuah `Inventory` (dapat menyimpan sekumpulan kartu). Kelas `Player` menurunkan atribut dan *method* dari *parent*-nya, tetapi juga memiliki atribut dan *method* yang hanya dimiliki oleh kelas `Player`, misalnya `name`, `number`, `point`, `ability`, dan `action`. Kelas `Deck` juga merupakan turunan dari kelas `Inventory`, karena sebuah `Deck` juga merupakan `Inventory`, namun memiliki *method* tersendiri, seperti `shuffleMainCards()` dan `getDeckFromInput()`. Kelas `MainCard` juga merupakan turunan dari kelas `Comparable`, karena `MainCard` adalah `Comparable`, tetapi memiliki atribut tersendiri yaitu `card` bertipe `pair`. Selain itu, kelas `Combo` juga merupakan turunan dari kelas `Comparable` dan kelas `Table` merupakan turunan dari kelas `Inventory`.

```
//FILE: Deck.hpp
class Deck : public Inventory {
public:
    Deck();
    void shuffleMainCards();
```

```
    void getDeckFromInput();
    void shuffleAbilityCards();
    void printCards();
    AbilityCard* takeAbilityFromDeck(int);
    void returnAbilityToDeck(AbilityCard*);
private:
    vector<AbilityCard> abilities;
    map<AbilityCard*, int> usedBy;
    int abilityCardTop;
};

//FILE: MainCard.hpp
class MainCard : public Comparable {
private:
    pair<int, int> card;

public:
    MainCard(int, int);
    int getNumber() const;
    // ...
    friend bool operator< (const MainCard, const MainCard);
    bool operator< (Comparable&);
    bool operator> (Comparable&);
    bool operator== (Comparable&);
};
```

Contoh lain dari penerapan *inheritance* adalah pada kelas `Poker` dan `Cangkul`, yang merupakan turunan dari kelas `Game`. Keuntungannya adalah, pada program utama, pengguna dapat memilih permainan yang akan dimainkan dan tiap permainan tersebut. Kelas `Poker` dan `Cangkul` dapat menggunakan *method* dari kelas `Game`, seperti `startGame()` atau `getPlayer()`. Selain itu, kelas `Poker` dan `Cangkul` juga dapat menggunakan *method* yang spesifik pada kelas tersebut. Kelas `Inventory`, `Comparable`, dan `Game` akan dijelaskan lebih lanjut di bagian 2.6 (*Abstract base class*).

```
//FILE: Poker.hpp
class Poker : public Game {
private:
    int shuffle;
    const int ROUND_AMOUNT;
public:
    Poker();
    void roundRobin();
    bool gameOver();
    void newShuffle();
    void newRound();
};

//FILE: Combo.cpp
class Combo : public Comparable {
public:
    Combo(const Player&, const Table&);
    float getValue() const;
    void setValue(float);
    bool operator> (Comparable&);
```

```
    bool operator< (Comparable&);
    bool operator== (Comparable&);
private:
    vector<MainCard> cards;
    // ...
};
```

2.2. Method/Operator Overloading

Method overloading berupa penimpaan dari fungsi dengan nama yang sama dengan pembeda berupa jenis dari parameter input yang diberikan dapat ditemukan pada kelas `Game`. Terdapat fungsi `getPlayerInTurn()` dan `getPlayerInTurn(int)` dimana fungsi tanpa parameter akan memberikan pemain yang sedang memiliki turn, dan fungsi dengan parameter int akan memberikan pemain pada indeks tertentu.

```
//FILE: Game.hpp
class Game {
// ...

public:
// ...
    /* Returns the player in turn */
    Player& getPlayerInTurn();
    /* Returns the player in turn index
     * @int index           Index of player */
    Player& getPlayerInTurn(int);
// ...
```

Berikut contoh penggunaan *method overloading* pada program.

```
//FILE: Ability_Reverse.cpp
void Reverse::useAbility(Game &game)
{
// ...
        string cur_name = game.getPlayerInTurn().getPlayerName();
// ...

        for (int i = cur_turn; i >= 0; i--)
        {
            current.push_back(game.getPlayerInTurn(i));
        }

        for (int i = game.getPlayerCount() - 1; i > cur_turn; i--)
        {
            current.push_back(game.getPlayerInTurn(i));
        }
// ...
}
```

Method overloading yang lain dapat dilihat pada kelas `IOHandler` dimana terdapat fungsi `getInputInAccepted()` dan `getInputInAccepted(int,int)` dimana fungsi tanpa parameter akan memastikan input berada pada vector accepted dan fungsi dengan parameter akan memastikan input berada pada range tertentu.

```
//FILE: Io_Handler.hpp
class IOHandler{
// ...
    /* Input a value of type T that exists in acceptedInput*/
    T getInputInAccepted();

    /* Input a value of type T that exists in a certain range
     * @param int l                 left of range(inclusive)
     * @param int r                 right of range(inclusive) */
    T getInputInAccepted(int,int);
// ...
};
```

Berikut contoh penggunaan *method overloading* pada program.

```
//FILE: Ability_Swap.cpp
void Swap::useAbility(Game &game) {
    // ...
    // Mendapatkan input kartu mana yang akan diswap (kiri/kanan)
    IOHandler<int> optionIO;
    int option1, option2;
    cout << "Silakan pilih kartu kiri/kanan " << player1.getPlayerName() << endl;
    cout << "1. Kanan \n2. Kiri" << endl;
```

```
    option1 = optionIO.getInputInAccepted(1, 2);
    // ...
}

//FILE: Action.cpp
void Action::cangkul(Cangkul& game, int color, Inventory& dump)
{
   IOHandler<string> stringIO;
   stringIO.addAccepted("YA");
   stringIO.addAccepted("TIDAK");
// ...
      cout<<"Ingin mencangkul? (YA/TIDAK)"<<endl;
      string input;
      input = stringIO.getInputInAccepted();
// ...
}
```

Operator overloading berupa operator perbandingan dapat ditemukan pada kelas `Comparable` dan seluruh turunannya. Overloading operator '<', '>', dan '==' digunakan untuk mempermudah perbandingan antara dua objek dari kelas `Comparable`. Tentunya, penggunaan operator untuk melakukan perbandingan akan lebih intuitif dan lebih praktis dibandingkan dengan menggunakan method seperti `isLess()`, `isGreater()`, atau `isEqual()`. Berikut adalah deklarasi operator perbandingan pada kelas `Comparable`.

```
class Comparable {
// ...
public:
```

```
// ...
    virtual bool operator< (Comparable&) = 0;
    virtual bool operator> (Comparable&) = 0;
    virtual bool operator== (Comparable&) = 0;
};
```

Berikut adalah contoh penggunaan operator perbandingan.

```
//FILE: Poker.cpp
void Poker::newShuffle()
{
    // Calculate winning combo
    for (int i = 0; i < this->PLAYER_AMOUNT; i++)
    {
        Player &player = this->players[i];
        // ...
    }

    int winnerID = maxElmtidx(this->combos); // Operator > terdapat pada fungsi ini
}

//FILE: comparable.hpp
template <class T>
int maxElmtidx(const vector<T> &container)
{
```

```
    int maxidx = 0;

    for (int i = 1; i < container.size(); i++)
    {
        T temp=container[i];
        T temp2=container[maxidx];
        if (temp > temp2)
        // Untuk contoh di atas, T = Combo, sehingga statement if tersebut menggunakan operator < pada kelas Combo
        {
            maxidx = i;
        }
    }
    return maxidx;
}
```

Selain itu, operator *left shift* ('<<') juga di-*overload* untuk melakukan output objek dari kelas `Inventory` dan `Combo`. Operator tersebut dijadikan sebagai *friend* pada kelas `Inventory` dan `Combo`, karena operan kiri operator tersebut bukan berasal dari kedua kelas tersebut, tetapi berasal dari kelas `ostream` (misalnya `cout`). Berikut adalah deklarasi operator left shift pada kelas `Inventory` dan `Combo`.

```
//FILE: Inventory.hpp
class Inventory {
protected:
    int amount;
    vector<MainCard> cards;
```

```
public:
    // ...
    friend ostream& operator<<(ostream&, const Inventory&);
};

//FILE: Combo.hpp
class Combo : public Comparable {
public:
    // ...
    friend ostream& operator<<(ostream&, const Combo&);

private:
    // ...
};
```

Berikut adalah contoh penggunaan operator *left shift*.

```
// FILE: Player.cpp
void Player::printCards()
{
    cout << (*this);
}

// FILE: Poker.cpp
void Poker::newShuffle()
```

```
{
// ...
    // Calculate winning combo
    for (int i = 0; i < this->PLAYER_AMOUNT; i++)
    {
        Player &player = this->players[i];
        this->combos[i] = Combo(player, this->table);
        // player.printInfo();
        cout << "Player " << player.getPlayerName() << " mendapatkan combo ";
        cout << "⚡" << this->combos[i].what() << "⚡" << '\n';
        cout << this->combos[i]; // Operator << untuk menampilkan combo
    }
// ...
}
```

Operator '+' dan '-' di-overload pada kelas `Inventory` untuk melakukan penambahan atau pengurangan kartu yang ada di `Inventory`.

```
// FILE: Inventory.hpp
class Inventory {
protected:
    // ...
public:
    // ...
    Inventory& operator+(vector<MainCard>);
```

```
    vector<MainCard> operator-(int);
};
```

Berikut adalah contoh penggunaan operator '+' dan '-'

```
// FILE: Inventory.cpp
void Inventory::takeCardFromDeck(Inventory& deck, int amount)
{
    vector<MainCard> temp = deck - amount;
    *this = *this + temp;
}
```

2.3. Template & Generic Classes

Penggunaan *generic class* terdapat pada kelas `IOHandler` yang berguna untuk menerima input dengan tipe T sesuai kebutuhan. *Method* `getInputInAccepted()` sudah *include* menerima input dan memvalidasi input pengguna, sehingga banyak digunakan pada *method* pada kelas lain yang membutuhkan input pengguna.

```
//FILE: IOHandler.h
template<class T>
class IOHandler{
private:
    vector<T> acceptedInput;
    vector<T> declinedInput;
```

```
public:
    IOHandler();
    IOHandler(vector<T>);
    void addAccepted(T);
    void addDeclined(T);
    T getInput();
    T getInputInAccepted();
    T getInputInAccepted(int,int);
    T getInputNotInDeclined();
};
```

Salah satu penggunaannya terdapat pada *method* useAbility() pada kelas Swap, di mana program meminta pemain untuk memilih kartu yang akan ditukar (kiri/kanan). Pada kasus ini, digunakan kelas IOHandler<int>, karena dibutuhkan input berupa *integer*.

```
//FILE: Ability_Swap.cpp
void Swap::useAbility(Game &game) {
    // ...
    // Mendapatkan input kartu mana yang akan diswap (kiri/kanan)
    IOHandler<int> optionIO;
    int option1, option2;
    cout << "Silakan pilih kartu kiri/kanan " << player1.getPlayerName() << endl;
    cout << "1. Kanan \n2. Kiri" << endl;
    option1 = optionIO.getInputInAccepted(1, 2);
    // ...
```

```
}

//FILE: Action.cpp
void Action::cangkul(Cangkul& game, int color, Inventory& dump)
{
   IOHandler<string> stringIO;
   stringIO.addAccepted("YA");
   stringIO.addAccepted("TIDAK");
// ...
      cout<<"Ingin mencangkul? (YA/TIDAK)"<<endl;
      string input;
      input = stringIO.getInputInAccepted();
// ...
}
```

Generic function yang diterapkan adalah class maxElmt dan maxElmtidx, dimana fungsi maxElmt ini menerima sebuah vector dengan elemen bertipe generik dan mengembalikan suatu elemen dengan nilai terbesar. Sedangkan, maxElmtidx menerima sebuah vector dengan elemen bertipe generik dan mengembalikan indeks dengan nilai terbesar. Cara fungsi mengetahui variabel dengan nilai terbesar adalah dengan menggunakan operator <, sehingga tipe generik yang digunakan harus memiliki implementasi operator< tersendiri.

```
//FILE: Comparable.hpp
```

```
/* Return maximum element of a vector, must contain at least one element */
template <class T>
T maxElmt(const vector<T> &container)
{
    T maxs = container[0];


    for (T element : container)
    {
        if (element > maxs)
        {
            maxs = element;
        }
    }
    return maxs;
}

/* Return index of maximum element of a vector, must contain at least one element */
template <class T>
int maxElmtidx(const vector<T> &container)
{
    int maxidx = 0;
    for (int i = 1; i < container.size(); i++)
    {
        T temp=container[i];
        T temp2=container[maxidx];
```

```
        if (temp > temp2)
        {
            maxidx = i;
        }
    }
    return maxidx;
}
```

Berikut adalah contoh pengimplementasian generic function pada program

```
//FILE: Poker.cpp
void Poker::newShuffle()
{
//...
    int winnerID = maxElmtidx(this->combos);
//...
}

void ComboCalculator::announce()
{
    int winnerID=maxElmtidx(this->combos);
    cout << "Pemain " << this->players[winnerID].getPlayerNumber() << " memenangkan permainan!" << endl;
}
```

2.4. Exception

Berikut adalah *exception* yang digunakan pada program ini. Terdapat *base class* dari seluruh *exception* yang digunakan pada program ini, yaitu `GameException`. Setiap *exception* memiliki method `printError()` yang akan mengembalikan *C-style string* berisi deskripsi dari *exception* tersebut.

```
class GameException : public exception
{
public:
    virtual ~GameException() {}
    virtual const char *printError() const throw() = 0;
};
class PlayerNotExist : public GameException
{/*...*/};
class PlayerNameInvalid : public GameException
{/*...*/};
class CreatePlayerFailed : public GameException
{/*...*/};
class NoAbilityAvailable : public GameException
{/*...*/};
class NotExpected : public GameException
{/*...*/};
class AbilityCardDisabled : public GameException
{/*...*/};
class InvalidFile : public GameException
{/*...*/};
class InvalidFileInput : public GameException
```

```
{/*...*/};
class DuplicateCardExist : public GameException
{/*...*/};
class CardsIncomplete : public GameException
{/*...*/};
```

`InvalidFileSyntax` akan dilemparkan apabila file yang digunakan untuk membaca deck tidak dapat dibuka

```
//FILE: Deck.cpp
void Deck::getDeckFromInput() {
    cout << "Input nama file: ";
    string filename;
    do {
        getline(cin, filename);
    } while (filename == "");

    FILE* fin = fopen(filename.c_str(), "r");
    if(fin == NULL){
        cout << "File not found\n";
        throw InvalidFile();
    }
    // ...
}
```

`PlayerNotExist` akan dilemparkan apabila suatu *method* yang digunakan untuk pencarian pemain tidak menemukan pemain yang diinginkan

```
// FILE: Game.cpp
Player& Game::getPlayer(int number)
{
    for (int i = 0; i < this->players.size(); i++)
        if (this->players[i].getPlayerNumber() == number)
            return this->players[i];

    throw PlayerNotExist();
}

Player& Game::getPlayer(string name)
{
    for (int i = 0; i < this->players.size(); i++)
        if (this->players[i].getPlayerName() == name)
            return this->players[i];

    throw PlayerNotExist();
}
```

`CreatePlayerFailed` digunakan apabila program gagal menambahkan pemain karena nama pemain baru yang akan ditambahkan sudah ada sebelumnya.

```
//FILE: Game.cpp
```

```
void Game::checkPlayerNameExist(string name)
{
    for (unsigned i = 0; i < this->players.size(); i++)
        if (this->players[i].getPlayerName() == name)
            throw CreatePlayerFailed();
}
```

Salah satu penggunaan `CreatePlayerFailed` adalah pada konstruktor kelas `Poker`.

```
//FILE: Poker.cpp
Poker::Poker() : Game(0, 64, 0, 7, 1 << 31), ROUND_AMOUNT(6)
{
    this->shuffle = 0;
    // Create player
    int i = 0;
    while (i < this->PLAYER_AMOUNT)
    {
        bool exception_caught = true;
        try
        {
            cout << "Masukkan nama pemain ke-" << i + 1 << ": ";
            string name;
            cin >> name;
            checkPlayerNameExist(name);
            Player p(name, i + 1);
```

```
            this->players.push_back(p);
            exception_caught = false;
        }
        catch (CreatePlayerFailed &e)
        {
            cout << e.printError() << endl;
        }
        if (!exception_caught)
            i++;
    }
}
```

`InputInvalid` digunakan apabila format penulisan deck yang ada di file salah, misalnya karakternya bukan digit.

```
//FILE: Deck.cpp
void Deck::getDeckFromInput()
{
    // ...
    string::iterator it;
    // read card number
    for(it = line.begin(); it != line.end(); ++it) {
        if(!numRead) {
            if (*it == ' ') {
                continue;
            }
```

```
            else if (isdigit(*it)) {
                int temp = *it - '0';
                if(temp == 0) {
                    continue;
                }
                num = temp;
                numRead = true;
            }
            else {
                throw InputInvalid();
            }
        }
        else if (num == 1 && (*it - '0')<= 3) {
            num *= 10;
            num += *it - '0';
        }
        else throw InputInvalid();
    }
    // ...
}
```

2.5. C++ Standard Template Library

- **Vector**
  Penyimpanan (*container*) dinamis serbaguna untuk kumpulan data yang serupa yang tidak memerlukan manajemen tambahan

```
//FILE: Combo.cpp
Combo::Combo(const Player& player, const Table& table)
{
//...
    vector<vector<MainCard>> perm;
    vector<MainCard> temp;
//...
// insert tableCard permutations
    int table_length = tableCard.size();
    for (int i = 0; i < table_length; i++){
        int n = perm.size();
        for(int j = 0; j < n; j++){
            if(perm[j].size() < 5){
                temp = perm[j];
                temp.push_back(tableCard[i]);
                perm.push_back(temp);
                temp.clear();
            }
        }
    }
//...
}
```

```
//FILE: Inventory.hpp
class Inventory {
```

```
protected:
    int amount;
    vector<MainCard> cards;
    // ...
}
```

```
//FILE: Deck.cpp
class Inventory {
protected:
    int amount;
    vector<MainCard> cards;
    // ...
}
```

`Vector` digunakan karena kemampuan alokasi memori yang dinamis, sehingga perubahan memori akibat penambahan (terutama penambahan) dan pengurangan elemen tidak perlu diperhatikan dalam implementasi. Selain itu, penggunaan vector memudahkan dalam menyalin *container* secara keseluruhan tanpa perlu mencari informasi mengenai ukuran *container* tersebut terlebih dahulu.

- **Map**

Penyimpanan dinamis yang menyediakan kemampuan untuk memetakan sebuah nilai (*key*) ke nilai lain (*value*) yang terkait ke *key* tersebut. Pencarian *key* dalam `Map` memiliki kompleksitas O(*log n*) yang lebih sederhana dibandingkan pencarian linear seperti pada *array* yang memiliki kompleksitas O(*n*)

```
//FILE: Player.hpp
class Player : public Inventory{
```

```
    //...
private:
    vector<AbilityCard*> abilities;
    map<AbilityCard*, int> usedBy;  // map kartu ability ke player yang sedang memiliki, -1 jika sedang tidak dimiliki (berada di deck)
    int abilityCardTop; // indeks top dari tumpukan ability card
};
```

`Map` digunakan untuk memetakan kartu *ability* yang terdapat pada deck kepada masing-masing pemain yang telah mengambil *ability* dari Deck tersebut.

- **Pair**

  Penyimpanan atribut/nilai dengan keterikatan satu sama lain, sehingga memudahkan dalam *passing* objek/informasi tanpa harus memproses dua informasi secara terpisah

```
//FILE: Player.hpp
class MainCard : public Comparable {
private:
    pair<int, int> card;
    //...
};
```

`Pair` digunakan untuk menyimpan pasangan nilai kartu dan warna kartu secara bersamaan, sehingga lebih teratur dalam pemrosesan informasi. Selain itu, pair juga digunakan pada `Map` untuk menyimpan pasangan *key* dan *value* seperti yang telah dijelaskan pada poin sebelumnya.

- **String**
  Penggunaan string memudahkan dalam *passing, copy*, *concat*, dan operasi-operasi kumpulan karakter lain.

```
//FILE: Exception.hpp
//...
class NoAbilityAvailable : public GameException
{
private:
    string _message;
public:
    NoAbilityAvailable()
    {
        this->_message = "Kamu belum punya ability saat ini 😔";
    }
    NoAbilityAvailable(string type)
    {
        this->_message = "Ets, tidak bisa. Kamu tidak punya kartu Ability " + type;
    }
    //...
};
```

  `String` digunakan untuk menggabungkan (konkatenasi) dua buah string sebagai pesan error dalam exception.

- **Rand() & Time()**
  `Rand()` digunakan untuk membangkitkan angka acak & `Time()` digunakan sebagai *seed* random sehingga membangkitkan angka yang relatif lebih “acak”.

```cpp
//FILE: Deck.cpp
//...
// Acak MainCard
void Deck::shuffleMainCards() {
    int size = cards.size();

    srand(time(NULL));
    for(int i = 0; i < size; i++) {
        int idx = rand() % size;  // generate random index from 0..size
        // swap
        MainCard temp = cards[i];
        cards[i] = cards[idx];
        cards[idx] = temp;
    }
}

// Acak AbilityCard
void Deck::shuffleAbilityCards() {
    int size = abilities.size();

    srand(time(NULL));
    for(int i = 0; i < size; i++) {
        int idx = rand() % size;  // generate random index from 0..size
        // swap
        AbilityCard* temp = abilities[i];
        abilities[i] = abilities[idx];
```

```
        abilities[idx] = temp;
    }
}
```

Penggunaan `Rand()` & `Time()` digunakan untuk membangkitkan angka acak sebagai metode mengocok (*shuffle*) kumpulan kartu.

- **Sort**

  `Sort()` digunakan untuk mengurutkan nilai-nilai dari sebuah kumpulan nilai (salah satunya `Vector`) dengan algoritma yang relatif cepat dibandingkan metode-metode sorting tradisional seperti *bubble sort*.

```
//FILE: Combo.cpp
Combo::Combo(const Player& player, const Table& table)
{
    vector<vector<MainCard>> perm;
    // insert permutations
    // ...

    // check for combos
    int perm_length = perm.size();
    for(int i = 0; i < perm_length; i++) {
        sort(perm[i].begin(), perm[i].end());
        // ...
    }
    // ...
}
```

`Sort()` digunakan untuk mengurutkan kumpulan kartu yang disimpan dalam `Vector` secara menaik untuk memudahkan dalam pemeriksaan kombo yang dibentuk.

- **Exception**

  Penggunaan `exception` sebagai kelas dasar untuk menggeneralisasi exception-exception yang mungkin terjadi di luar exception umum atau yang dapat diakibatkan sistem.

```
//FILE: Exception.hpp
class GameException : public exception
{
public:
    virtual ~GameException() {}
    virtual const char *printError() const throw() = 0;
};
```

2.6. Konsep OOP lain

Konsep OOP lain yang digunakan adalah *abstract base class*, yaitu kelas abstrak yang memiliki kelas turunan dan *pure virtual method* yang hanya bisa diimplementasikan pada kelas turunannya. Terdapat tiga *abstract base class* yang digunakan, yaitu kelas `Game`, `Inventory`, dan `Comparable`.

```
// FILE: Game.hpp
class Game {
protected:
    int round;
    int turn;
```

```
    long long point;
    const int PLAYER_AMOUNT;
    const int WIN_POINT;
    vector<Player> players;
    Table table;
    Deck deck;

public:
    Game(int, int, long long, int, long long);
    void checkPlayerNameExist(string);
    int getTurn() const;
    int getRound() const;
    long long getPoint() const;
    void setPoint(long long);
    int getPlayerCount() const;
    Player& getPlayer();
    Player& getPlayerInTurn();
    Player& getPlayer(string);
    Player& getPlayer(int);
    void printPlayerTurn();
    Table& getTable();
    Deck& getDeck();
    virtual bool gameOver()=0;
    void startGame();
    virtual void newShuffle() = 0;
};
```

Alasan pembuatan *abstract base class* `Game` adalah karena terdapat persamaan karakteristik antara permainan poker dan cangkul, misalnya sama-sama memiliki round, turn, players, point, deck, *method* untuk memulai dan mengakhiri permainan, dsb. Namun, implementasi `gameOver()` dan `newShuffle()` bergantung pada jenis `Game` yang digunakan. Oleh karena itu, kedua *method* ini dibuat *pure virtual*.

```
//FILE: Inventory.hpp
class Inventory {
protected:
    int amount;
    vector<MainCard> cards;

public:
    Inventory(int);
    int getAmount() const;
    void setAmount(int);
    vector<MainCard> getInventoryCards() const;
    MainCard getCard(int) const;
    void setCard(int, MainCard);
    bool isInventoryEmpty();
    bool isColorExists(int);
    void returnCardToDeck(Inventory&);
    void returnOneCardToDeck(Inventory&, int);
    void takeCardFromDeck(Inventory&, int);
    virtual void printCards() = 0;
    Inventory& operator+(vector<MainCard>);
    vector<MainCard> operator-(int);
```

```
};
```

Alasan pembuatan *abstract base class* `Inventory` adalah karena terdapat persamaan karakteristik antara kelas `Player`, `Table`, dan `Deck`, yaitu sama-sama dapat menyimpan sejumlah kartu, menampilkan kartu, menambahkan kartu, dan mengembalikan kartu. Namun, implementasi `printCard()` berbeda-beda pada kelas `Player`, `Table`, dan `Deck`. Oleh karena itu, *method* ini dibuat *pure virtual*.

```
//FILE: Comparable.hpp
class Comparable {
protected:
    float value;

public:
    float getValue() const;/
    void setValue(float);

    /***** Comparison operator *****/
    virtual bool operator< (Comparable&) = 0;
    virtual bool operator> (Comparable&) = 0;
    virtual bool operator== (Comparable&) = 0;
};
```

Alasan pembuatan *abstract base class* `Comparable` adalah karena terdapat persamaan karakteristik antara kelas `MainCard` dan `Combo`, yaitu sama-sama memiliki value yang dapat dibandingkan dengan `Comparable` lainnya. Namun, implementasi operator ‘<’, ‘>’, dan ‘==’ berbeda-beda pada kelas `Player`, `MainCard`, dan `Combo`. Oleh karena itu, operator ini dibuat *pure virtual*.

Konsep *composition* juga digunakan pada program ini, di mana objek dari suatu kelas dapat menjadi atribut dari kelas lainnya. Salah satu contohnya adalah pada kelas `Player`, yang memiliki atribut dari kelas `AbilityCard` dan `Action`.

```
//FILE: Player.hpp
class Player : public Inventory {
private:
    string name;
    int number;
    long long point;
    bool status;
    AbilityCard* ability;
    Action playerAction;

public:
// ...
};
```

3. Bonus Yang dikerjakan

3.1. Bonus yang diusulkan oleh spek

3.1.1. Generic Class

Generic class yang diterapkan adalah class IOHandler, dimana kelas ini memiliki atribut acceptedInput dan declinedInput, dimana acceptedInput merepresentasikan kumpulan input dengan tipe generik yang diterima, sedangkan

declinedInput merepresentasikan kumpulan input dengan tipe generik yang tidak diterima. Penggunaan kelas ini telah dijelaskan pada bagian 2.3.

```
template<class T>
class IOHandler{
private:
    vector<T> acceptedInput;
    vector<T> declinedInput;

public:
    /* IOHandler default constructor
     * @param vector<T> acc         Vector of accepted inputs */
    IOHandler();

    /* IOHandler constructor
     * @param vector<T> acc         Vector of accepted inputs */
    IOHandler(vector<T>);

    /* accepted adder
     * @param T acc                 new accepted inputs */
    void addAccepted(T);


    /* declined adder
     * @param T acc                 new declined inputs */
    void addDeclined(T);
```

```
    /***** Input *****/
    /* Input a value of type T */
    T getInput();


    /* Input a value of type T that exists in acceptedInput*/
    T getInputInAccepted();


    /* Input a value of type T that exists in a certain range
     * @param int l                 left of range(inclusive)
     * @param int r                 right of range(inclusive) */
    T getInputInAccepted(int,int);


    /* Input a value of type T that does not exists in declinedInput*/
    T getInputNotInDeclined();
};
```

3.1.2. Generic Function

Generic function yang diterapkan adalah class maxElmt dan maxElmtidx, dan telah dijelaskan pada bagian 2.3

```
/* Return maximum element of a vector, must contain at least one element */
template <class T>
T maxElmt(const vector<T> &container)
{
    T maxs = container[0];


    for (T element : container)
    {
        if (element > maxs)
        {
            maxs = element;
        }
    }
    return maxs;
}

/* Return index of maximum element of a vector, must contain at least one element */
template <class T>
int maxElmtidx(const vector<T> &container)
{
    int maxidx = 0;
    for (int i = 1; i < container.size(); i++)
    {
        T temp=container[i];
        T temp2=container[maxidx];
```

```
        if (temp > temp2)
        {
            maxidx = i;
        }
    }
    return maxidx;
}
```

3.1.3. Game Kartu Lain

Game kartu lain yang diimplementasikan sebagai bonus adalah permainan Cangkul. Permainan ini terdiri dari 4 orang dengan deck kartu yang serupa seperti deck permainan utama (52 kartu, dengan 13 angka dan 4 warna). Permainan ini juga memiliki sebuah tempat 'pembuangan' kartu yang disebut dump.

Pada awal permainan, sebuah kartu dari deck akan ditaruh ke atas table, kemudian pemain dapat bermain secara bergilir. Pada setiap giliran, pemain dapat menaruh kartu yang sesuai jika ia memiliki kartu dengan warna yang berada di table, atau mengambil (men*cangkul*) kartu dari deck apabila tidak ada kartu yang sesuai, hingga didapat kartu dengan warna yang sesuai, untuk kemudian diberikan ke table. Pemain yang tidak ingin mencangkul dapat keluar dari game dan kartu yang ia miliki akan diletakkan pada dump. Apabila deck telah kosong, maka deck akan diisi oleh kartu dari dump. Apabila keduanya telah kosong, maka pemain dianggap kalah dan kartunya akan diberikan pada dump.

Pada setiap giliran yang baru, urutan pemain akan ditentukan oleh nilai kartu yang diberikan oleh tiap pemain, berdasarkan nilai kartu terbesar. Permainan akan selesai apabila tinggal terdapat satu orang pemain atau terdapat pemain yang telah menghabiskan seluruh kartunya.

Implementasi dilakukan dengan membentuk kelas Cangkul yang diinherit dari kelas abstrak Game. Kemudian, dilakukan perubahan kecil dalam pengaturan alur permainan seperti pengaturan ronde, pengaturan gameOver, dan juga aksi-aksi pemain.

```
class Cangkul : public Game {
private:
    int shuffle;
    Table dump;
    const int ROUND_AMOUNT;

public:
    /***** Constructor dan destructor *****/
    /* Cangkul constructor */
    Cangkul();


    /* Determines the next round position */
    void roundManage();


    /* Manage the dump, return true if deck is cangkul-able */
    bool dumpManage();


    /* Check whether the game is over */
    bool gameOver();


    /* Announce who won the game  */
    void gameWinner();
```

```
    /* Starts a new shuffle instance */
    void newShuffle();


    /* Starts a new round instance */
    void newRound();
};
```

3.2. Bonus Kreasi Mandiri

Bonus kreasi yang dibuat adalah Combo Calculator, yaitu program untuk menentukan combo terbaik yang didapatkan dari susunan tertentu. Program ini dapat digunakan untuk mensimulasikan permainan utama dan memperlihatkan combo-combo terbaik tiap pemain dan juga pemenang dari permainan tersebut.

```
#include "Combo.hpp"

class ComboCalculator:public Table{
    private:
    vector<MainCard> stored;
    vector<Player> players;
    vector<Combo> combos;
    float value;
    string comboType;
```

```
    int playerNum;

    public:
    /* ComboCalculator constructor */
    ComboCalculator(){}

    /* Start the module */
    void start();

    /* Calculate all combos */
    void calculate();

    /* Announce winner */
    void announce();

    /* Input a certain amount of card
     * @param int amount        Amount of card to be input */
    void getCard(int);
};
```

4. Pembagian Tugas

Modul	Implementer	Tester
Combo	13521170, 13521138	13521170, 13521138, 13521082
Ability swap, switch	13521170	13521170
Ability abilityless, quadruple, quarter	13521150	13521150
Ability reroll, reverse	13521082	13521082
Action, table	13521082	13521082
Comparable	13521092	13521150
Combo Calculator	13521082	13521092
Deck	13521138	13521138
Game	13521092	13521082, 13521092
Header	13521092	13521082, 13521138, 13521150, 13521170
Main	13521092	13521092
MainCard	13521150	13521150
Inventory	13521092	13521092
IO handler	13521082	13521082, 13521170
Player	13521138	13521138
Poker	13521082, 13521092, 13521138	13521082, 13521092, 13521138

Exception	13521138, 13521092	13521138
Cangkul	13521082	13521082

LINK REPOSITORY

https://github.com/frankiehuangg/IF2210-Tubes-1

FOTO KELOMPOK

Kode Kelompok : SUS
Nama Kelompok : WhenTheObject
1. 13521082 / Farizki Kurniawan
2. 13521092 / Frankie Huang
3. 13521138 / Johann Christian Kandani
4. 13521150 / I Putu Bakta Hari Sudewa
5. 13521170 / Haziq Abiyyu Mahdy
Asisten Pembimbing : Widya Anugrah Putra

1. Konten Diskusi

1. Bagaimana detail atau garis besar implementasi operator+ & operator- pada MainCard?
2. Apakah kartu bisa dibandingkan dengan combo?
3. Apakah ada saran terkait penggunaan map yang sesuai?
4. Apakah generic function bisa diimplementasikan di luar kelas?
5. Friend class yang diimplementasikan apakah bisa terhitung kelas tersendiri?
6. Bagaimana menghadapi saling import antar kelas?

2. Tindak Lanjut

1. Implementasi dilakukan dengan operator+ memiliki parameter kartu yang ditambahkan, dan operator- dengan parameter jumlah kartu yang akan ditarik
2. Diimplementasikan kartu bisa dibandingkan dengan kartu, dan kombo hanya bisa dibandingkan dengan kombo
3. Diimplementasikan penggunaan map bisa digunakan di combo
4. Bebas dilakukan di luar atau dalam kelas
5. Akan ditanyakan pada tim asisten
6. Coba cari informasi mengenai “circular import” di Google