Kode Kelompok : **JKW**
Nama Kelompok : **Nasi Cumi Hitam Madura Pak Bagas**

1. Adam Dharma Sakti/10023359
2. Farhan Nafis Rayhan/13522037
3. Kharris Khisunica/13522051
4. Bagas Sambega Rosyada/13522071
5. Raden Francisco Trianto B./13522091
6. Fabian Radenta Bangun/13522105

Asisten Pembimbing : Marcellus Michael Herman Kahari/13520057

1. Diagram Kelas

Diagram kelas adalah diagram yang menggambarkan deskripsi mengenai kelas dan hubungan antarkelas pada suatu program. Diagram kelas yang digunakan menjelaskan mengenai hubungan antarkelas pada Tugas Besar ini. Dalam Tugas Besar ini ada total 17 kelas yang terdiri dari kelas utama berupa kelas Game yang menjadi *controller* dan penghubung antara IO pengguna dengan fungsi-fungsi yang ada, lalu kelas Player yang menjadi cetakbiru pemain-pemain yang ada di permainan, kelas Storage yang menjadi penyimpanan bagi Player dan tambahan penyimpanan yang dibutuhkan *role*, kelas Plant atau kelas tanaman, kelas Animal atau kelas hewan, kelas Product atau kelas produk, kelas Shop atau kelas toko, dan kelas Building atau kelas bangunan.

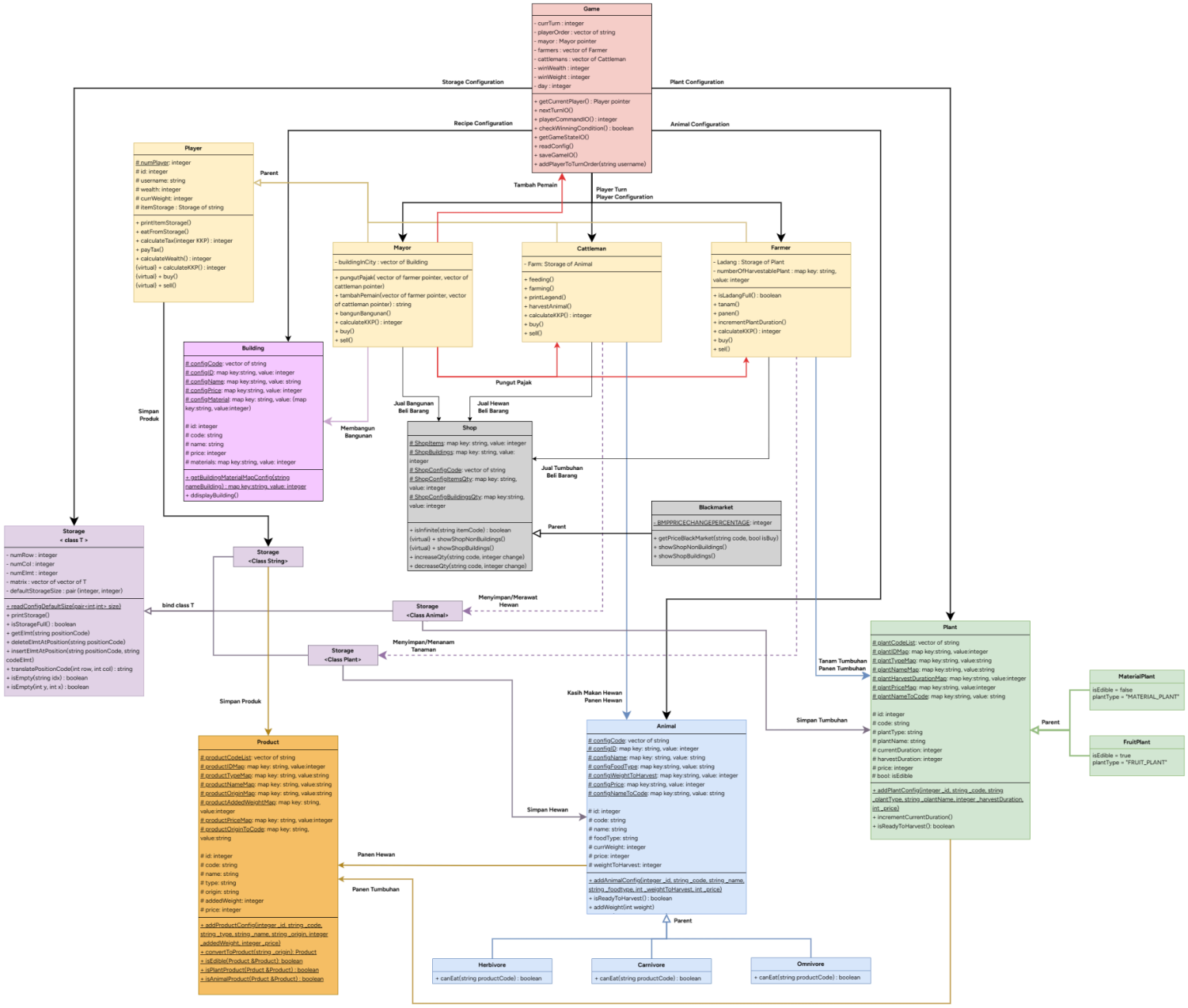
Kelas Player adalah kelas yang merepresentasikan pemain di dalam permainan. Kelas Player merupakan sebuah *abstract base class* karena memiliki *pure virtual method* pada fungsi beli, jual dan hitung KKP yang akan dijelaskan di bagian Inheritance dan Abstract Base Class. Kelas Player memiliki hubungan *inheritance* dengan kelas Mayor, Farmer dan Cattleman, karena setiap peran tersebut merupakan Player juga. Setiap Player memiliki sebuah atribut Storage sebagai penyimpanan, sehingga memiliki hubungan agregasi. Storage tersebut dapat menyimpan berbagai tipe barang, yaitu Plant, Animal, Product, dan Building.

Setiap turunan kelas Player memiliki spesialisasi dan perannya masing-masing. Mayor memiliki kemampuan untuk membangun sebuah bangunan, atau melakukan instansiasi terhadap kelas Building. Farmer memiliki atribut khusus berupa Ladang yang merupakan tempat menanam tanaman atau Storage of Plant, dan Cattleman memiliki atribut khusus berupa Farm yang menjadi tempat beternak atau Storage of Animal. Kedua kelas Farmer dan Cattleman memiliki tambahan atribut agregasi khusus dengan kelas Storage dan hubungan asosiasi dengan Plant dan Animal.

Setiap Player memiliki kemampuan untuk membeli dan menjual barang, sehingga mereka memiliki akses ke kelas Shop dan BlackMarket. Kelas BlackMarket sendiri merupakan turunan kelas Shop dan menjadi tipe khusus Shop (dijelaskan di bagian Bonus Kreasi Mandiri). Oleh karenanya kelas Player memiliki hubungan asosiasi dengan kelas Shop, dengan catatan bahwa kelas Mayor tidak dapat membeli Building dari Shop dan Farmer juga Cattleman tidak dapat menjual Building ke Shop.

Hubungan antara Game dengan kelas Mayor, Cattleman, Farmer, Shop, Plant, Animal, Product, dan Building adalah hubungan agregasi dan asosiasi. Terdapat beberapa fungsi dan atribut statik pada kelas-kelas tersebut berupa data yang disimpan dari file konfigurasi sehingga tidak memerlukan instansiasi untuk mengaksesnya. Sementara itu kelas Game memiliki atribut-atribut berupa objek Mayor, objek Shop dan BlackMarket, *vector of* Farmer dan Cattleman. Oleh karenanya kelas Game juga memiliki hubungan agregasi.

Diagram UML program dapat dilihat pada gambar di halaman berikut ini, atau untuk mengakses diagram secara penuh dapat melalui pranala berikut Diagram UML (DrawIO).


Game
- currTurn : integer
- playerOrder : vector of string
- mayor : Mayor pointer
- farmers : vector of Farmer
- cattlemans : vector of Cattleman
- winWealth : integer
- winWeight : integer
- day : integer
+ getCurrentPlayer() : Player pointer
+ nextTurnIO()
+ playerCommandIO() : integer
+ checkWinningCondition() : boolean
+ getGameStateIO()
+ readConfig()
+ saveGameIO()
+ addPlayerToTurnOrder(string username)
Storage Configuration
Plant Configuration
Recipe Configuration
Animal Configuration
Player
numPlayer: integer
id: integer
username: string
wealth: integer
currWeight: integer
itemStorage : Storage of string
+ printItemStorage()
+ eatFromStorage()
+ calculateTax(integer KKP) : integer
+ payTax()
+ calculateWealth() : integer
(virtual) + calculateKKP() : integer
(virtual) + buy()
(virtual) + sell()
Parent
Tambah Pemain
Player Turn
Player Configuration
Mayor
- buildingInCity : vector of Building
+ pungutPajak(vector of farmer pointer, vector of cattleman pointer)
+ tambahPemain(vector of farmer pointer, vector of cattleman pointer) : string
+ bangunBangunan()
+ calculateKKP() : integer
+ buy()
+ sell()
Cattleman
- Farm: Storage of Animal
+ feeding()
+ farming()
+ printLegend()
+ harvestAnimal()
+ calculateKKP() : integer
+ buy()
+ sell()
Farmer
- Ladang : Storage of Plant
- numberOfHarvestablePlant : map key: string, value: integer
+ isLadangFull() : boolean
+ tanam()
+ panen()
+ incrementPlantDuration()
+ calculateKKP() : integer
+ buy()
+ sell()
Pungut Pajak
Building
configCode: vector of string
configID: map key:string, value: integer
configName: map key:string, value: string
configPrice: map key:string, value: integer
configMaterial: map key: string, value: (map key:string, value:integer)
id: integer
code: string
name: string
price: integer
materials: map key:string, value: integer
+ getBuildingMaterialMapConfig(string nameBuilding) : map key:string, value: integer
+ ddisplayBuilding()
Simpan Produk
Membangun Bangunan
Jual Bangunan Beli Barang
Jual Hewan Beli Barang
Shop
ShopItems: map key: string, value: integer
ShopBuildings: map key: string, value: integer
ShopConfigCode: vector of string
ShopConfigItemsQty: map key: string, value: integer
ShopConfigBuildingsQty: map key:string, value: integer
+ isInfinite(string itemCode) : boolean
(virtual) + showShopNonBuildings()
(virtual) + showShopBuildings()
+ increaseQty(string code, integer change)
+ decreaseQty(string code, integer change)
Jual Tumbuhan Beli Barang
Blackmarket
- BMPPRICECHANGEPERCENTAGE: integer
+ getPriceBlackMarket(string code, bool isBuy)
+ showShopNonBuildings()
+ showShopBuildings()
Parent
Storage
< class T >
- numRow : integer
- numCol : integer
- numElmt : integer
- matrix : vector of vector of T
- defaultStorageSize : pair (integer, integer)
+ readConfigDefaultSize(pair<int,int> size)
+ printStorage()
+ isStorageFull() : boolean
+ getElmt(string positionCode)
+ deleteElmtAtPosition(string positionCode)
+ insertElmtAtPosition(string positionCode, string codeElmt)
+ translatePositionCode(int row, int col) : string
+ isEmpty(string idx) : boolean
+ isEmpty(int y, int x) : boolean
bind class T
Storage
<Class String>
Storage
<Class Animal>
Storage
<Class Plant>
Menyimpan/Merawat Hewan
Menyimpan/Menanam Tanaman
Simpan Produk
Kasih Makan Hewan
Panen Hewan
Tanam Tumbuhan
Panen Tumbuhan
Simpan Tumbuhan
Plant
plantCodeList: vector of string
plantIDMap: map key:string, value:integer
plantTypeMap: map key:string, value:string
plantNameMap: map key:string, value:string
plantHarvestDurationMap: map key:string, value:integer
plantPriceMap: map key:string, value:integer
plantNameToCode: map key:string, value: string
id: integer
code: string
plantType: string
plantName: string
currentDuration: integer
harvestDuration: integer
price: integer
bool: isEdible
+ addPlantConfig(integer _id, string _code, string _plantType, string _plantName, integer _harvestDuration, int _price)
+ incrementCurrentDuration()
+ isReadyToHarvest(): boolean
Parent
MaterialPlant
isEdible = false
plantType = "MATERIAL_PLANT"
FruitPlant
isEdible = true
plantType = "FRUIT_PLANT"
Product
productCodeList: vector of string
productIDMap: map key: string, value:integer
productTypeMap: map key: string, value:string
productNameMap: map key: string, value:string
productOriginMap: map key: string, value:string
productAddedWeightMap: map key: string, value:integer
productPriceMap: map key: string, value:integer
productOriginToCode: map key: string, value:string
id: integer
code: string
name: string
type: string
origin: string
addedWeight: integer
price: integer
+ addProductConfig(integer _id, string _code, string _type, string _name, string _origin, integer _addedWeight, integer _price)
+ convertToProduct(string _origin): Product
+ isEdible(Product &Product): boolean
+ isPlantProduct(Product &Product) : boolean
+ isAnimalProduct(Product &Product) : boolean
Animal
configCode: vector of string
configID: map key: string, value: integer
configName: map key: string, value: string
configFoodType: map key:string, value: string
configWeightToHarvest: map key:string, value: integer
configPrice: map key:string, value: integer
configNameToCode: map key:string, value: string
id: integer
code: string
name: string
foodType: string
currWeight: integer
price: integer
weightToHarvest: integer
+ addAnimalConfig(integer _id, string _code, string _name, string _foodtype, int _weightToHarvest, int _price)
+ isReadyToHarvest() : boolean
+ addWeight(int weight)
Simpan Hewan
Simpan Tumbuhan
Parent
Herbivore
+ canEat(string productCode) : boolean
Carnivore
+ canEat(string productCode) : boolean
Omnivore
+ canEat(string productCode) : boolean

2. Penerapan Konsep OOP

2.1. Inheritance & Polymorphism

Inheritance merupakan salah satu konsep dalam pemrograman berbasis objek yang memungkinkan suatu objek atau kelas memiliki kelas dan/atau objek turunan yang memiliki ciri/sifat yang diturunkan dari kelas/objek lainnya. Saat suatu kelas diturunkan dari kelas lain, kelas yang menjadi *child* (yang diturunkan) akan membawa *atribut*, *method*, dan *properties* lainnya yang dimiliki oleh kelas *parent* (kelas yang menurunkan).

Pada tugas ini, terdapat beberapa kelas yang menggunakan konsep *inheritance* yaitu kelas Player yang merupakan representasi kelas pemain sebagai *parent class,* memiliki beberapa kelas turunan yaitu kelas Farmer (petani), Cattleman (peternak), dan kelas Mayor (walikota). Selain itu beberapa kelas seperti kelas Animal (hewan) memiliki kelas turunan yang mendefinisikan jenis spesifik untuk hewan, Plant (tanaman) memiliki kelas turunan yang mendefinisikan jenis spesifik untuk tumbuhan.

Polymorphism berasal dari dua suku kata “poly” yang artinya banyak dan “morph” yang artinya bentuk. *Polymorphism* (banyak bentuk) adalah salah satu konsep pada pemrograman berorientasi objek yang memungkinkan suatu kelas untuk memiliki lebih dari satu kelakuan atau *behavior* yang berbeda pada suatu *method*-nya.

2.1.1 Kelas Player

Player merupakan kelas yang menyatakan cetak biru pemain yang bermain dalam permainan. Player sendiri merupakan sebuah *abstract class* karena memiliki *method* virtual pada fungsi buy (membeli), sell (menjual), dan calculateKKP (hitung besar KKP), yang akan dijelaskan lebih lanjut pada subbab 2.6 Abstract Class. Player merupakan Parent Class yang memiliki atribut dasar yang meliputi seluruh atribut yang diperlukan baik oleh Farmer (Petani), Cattleman (Peternak), maupun Mayor (Walikota). Kelas Player dibuat untuk menghindari repetisi atribut dan fungsi pada setiap Child Class (DRY : Don’t Repeat Yourself) Farmer, Cattleman, dan Mayor, misalnya atribut seperti ID, username, wealth (kekayaan), penyimpanan, dan currWeight (berat Player saat ini).

```
class Player{
   protected:
       /* Attributes */
       static int numPlayer;
       const int id;
       const string username;
```

```
    int wealth;
    int currWeight;
    Storage<string> ItemStorage;

    /* Default Variables */
    static const int DefaultPlayerStartingWealth = 50;
    static const int DefaultPlayerStartingWeight = 40;

public:
    Player(string _userName);

    Player(string _username, int _wealth, int _currWeight);

    ~Player();

    int getId() const;

    string getUsername() const;

    int getWealth() const;

    int getCurrWeight() const;

    void setWealth(int _wealth);

    void setCurrWeight(int _currWeight);

    void printItemStorage();

    void eatFromStorage();

    virtual void buy() = 0;

    virtual void sell() = 0;

    string itemType(string positionCode);

    string itemType(int y, int x);
```

```
        int calculateTax(int KKP);

        void payTax();

        int calculateWealth();

        virtual int calculateKKP() = 0;
};
```

Sehingga dengan menggunakan konsep Inheritance, kelas Farmer, Cattleman dan Mayor yang menjadi Child Class dari Player, tidak perlu menulis kembali atribut-atribut yang sudah ada dari kelas Player. Kelas-kelas tersebut cukup menambahkan atribut tambahan yang memang diperlukan pada setiap perannya, misalnya kelas Farmer yang memerlukan tambahan atribut Ladang dan Cattleman dengan Peternakan yang berasal dari Storage. Hal ini juga menjadikan kelas-kelas Child dapat memakai kelas Player dan substitusi atribut dan fungsi ke kelas Player (Liskov Substitution Principle).

```
class Farmer : public Player {
private:
    Storage<Plant> Ladang;

public:
    Farmer(string _username);

    Farmer(string _username, int _wealth, int _currWeight);

    ~Farmer();

    bool isLadangFull();

    Storage<Plant> getLadang();

    void tanam();
```

```
    void panen();

    bool isLadangSlotEmpty(string idx);

    Plant getItem(string idx);

    void buy() override;

    void sell() override;

    void buyBlackMarket();

    void sellBlackMarket();

    void incrementPlantDuration();

    int calculateKKP() override;
};
```

Misalnya dapat dilihat pada kelas Farmer bahwa terdapat atribut tambahan berupa Ladang yang bertipe data Storage, dengan fungsi tambahan seperti tanam dan panen yang hanya dapat dilakukan oleh Farmer saja, namun untuk fungsi seperti payTax (bayar pajak) tidak diimplementasikan ulang karena akan digunakan untuk semua child class dari Player lainnya. Hal tersebut juga berlaku untuk Cattleman yang dapat memanen hewan dan memberi makan hewan, dan untuk Mayor yang dapat membangun bangunan namun tidak dapat membeli bangunan.

2.1.2 Kelas Plant

Kelas Plant merupakan kelas yang merepresentasikan tanaman. Kelas Plant merupakan sebuah Parent Class yang memiliki 2 child class, yaitu kelas Material Plant dan Fruit Plant. Kelas Material Plant adalah kelas yang nantinya jika dipanen akan digunakan sebagai bahan membangun bangunan, dan kelas Fruit Plant adalah kelas yang produknya dapat dimakan. Perbedaan utama dari kedua kelas ini adalah hasil produk yang nantinya yang dapat dimakan dan juga tipe tanaman, sehingga *default value* untuk atribut isEdible dan plantType berbeda untuk kedua child class. Oleh karenanya kelas ini juga menerapkan prinsip Liskov Substitution Principle dan juga hanya terdapat beberapa tambahan nilai atribut saja pada child class, sehingga menghindari pengulangan kode (konsep DRY).

```
class Plant {
protected:
      // Global variable, diambil dari file config
      static vector <string> plantCodeList; // Digunakan code 3 char karena parameter unik
      static map <string, int> plantIDMap; // <code, map>
      static map <string, string> plantTypeMap; // <code, plantType>
      static map <string, string> plantNameMap; // <code, plantName>
      static map <string, int> plantHarvestDurationMap; // <code, harvestDuration>
      static map <string, int> plantPriceMap; // <code, price>
      static map <string, string> plantNameToCode; // <plantName, code>

      int id;                          // plant ID
      string code;                     // 3 characters plant code
      const string plantType;          // Plant type (Material Plant, Food Plant)
      string plantName;                // Plant name
      int currentDuration = 0;         // Durasi saat ini sebelum bisa di-harvest, default 0
      const int harvestDuration;       // Durasi yang dibutuhkan untuk bisa di-harvest
      const int price;                 // Harga jual tanaman
      bool isEdible;
public:
      Plant();

      Plant(string _code);

      Plant(int _id, string _code, string _plantType, string _plantName, int _harvestDuration, int
_price);

      Plant(Plant &other);

      ~Plant();

      int getID();

      string getCode();
```

```
    string getPlantType();

    string getPlantName();

    int getCurrentDuration();

    int getHarvestDuration();

    bool getEdible();

    int getPrice();

    void setID(int _id);

    void setCode(string _code);

    void setPlantName(string _plantName);

    void setCurrentDuration(int _currentDuration);

    void printInfo();

    void static addPlantConfig(int _id, string _code, string _plantType, string _plantName, int
_harvestDuration, int _price);

    bool isReadyToHarvest();

    bool empty();

    static vector<string> getPlantCodeListConfig();

    static map<string, int> getPlantIDMapConfig();

    static map<string, string> getPlantTypeMapConfig();
```

```
     static map<string, string> getPlantNameMapConfig();

     static map<string, int> getPlantHarvestDurationMapConfig();

     static map<string, int> getPlantPriceMapConfig();

     static map<string, string> getPlantNameToCodeConfig();

   static void printParsedConfig();

   void incrementCurrentDuration();
};

class MaterialPlant : public Plant {
public:

     MaterialPlant(int _id, string _code, string _plantName, int _harvestDuration, int _price);


     ~MaterialPlant();
};


class FoodPlant : public Plant {
public:

     FoodPlant(int _id, string _code, string _plantName, int _harvestDuration, int _price);

     ~FoodPlant();
};
```

2.1.3 Kelas Animal

Kelas Animal adalah kelas yang merepresentasikan hewan. Kelas Animal merupakan Parent Class yang memiliki 3 kelas turunan, yaitu Omnivore, Herbivore, dan Carnivore. Pada atribut dasarnya, kelas Animal memiliki semua atribut yang sama dengan child class-nya, namun terdapat beberapa atribut yang nilainya dijadikan *default*, seperti atribut foodType. Kelas Animal sendiri merupakan sebuah *abstract class* karena terdapat fungsi virtual yaitu printInfo() dan canEat(). Fungsi printInfo() adalah fungsi untuk mencetak seluruh informasi mengenai hewan, dan fungsi ini dibuat virtual karena terdapat informasi mengenai canEat() yaitu tipe makanan apa yang bisa dimakan oleh hewan.

```
class Animal{
    protected:
        const int id;               // Animal id
        const string code;          // Animal code
        const string name;          // Animal name
        const int weightToHarvest;  // Animal minimum weight for animal to be harvested
        const int price;            // Animal price
        const string foodType;      // Animal Food type
        int currWeight;             // Animal current weight
virtual bool canEat(string productType);
virtual void printInfo();
```

Fungsi canEat() merupakan fungsi untuk mengecek apakah suatu makanan dapat dimakan oleh tipe Animal tersebut. Fungsi ini di-*override* di *child class*-nya masing-masing karena memiliki nilai pengecekan yang berbeda-beda setiap tipe hewan, yaitu jika Herbivore jika makanannya bertipe product fruit plant, Carnivore jika makanannya bertipe product animal, dan Omnivore jika keduanya atau salah satu di antara keduanya.

```
bool Herbivore::canEat(string productType){
    return productType == "PRODUCT_FRUIT_PLANT";
}
```

```
bool Omnivore::canEat(string productType){
    return productType == "PRODUCT_ANIMAL" || productType == "PRODUCT_FRUIT_PLANT";
}
```


```
bool Omnivore::canEat(string productType){
    return productType == "PRODUCT_ANIMAL" || productType == "PRODUCT_FRUIT_PLANT";
}
```


2.2. Method/Operator Overloading

Method Overloading adalah suatu konsep OOP pada bahasa pemrograman C++ yang memungkinkan suatu fungsi memiliki nama sama namun memiliki parameter atau *return type* yang berbeda. Pada Tugas Besar ini, digunakan beberapa *method overloading* pada fungsi:

a. getElmt(string positionCode) dan getElmt(int y, int x) pada kelas Storage. Keduanya merupakan fungsi yang digunakan untuk mendapatkan elemen pada vektor yang menjadi struktur penyimpanan pada kelas Storage, namun memiliki 2 jenis parameter masukan, yaitu fungsi pertama menggunakan string dan yang kedua memasukkan nilai integer y dan x. Kedua fungsi dibuat saling *overload* untuk mengatasi kasus saat mengakses elemen dari masukan pengguna berupa string posisi (misal A09, G11), dan fungsi yang menggunakan parameter integer digunakan saat terdapat fungsi lain yang memerlukan elemen dari Storage dengan masukan berupa titik y dan x, misalnya jika berasal dari *double for loop*, sehingga tidak diperlukan konversi ulang dari integer ke string.

```
template<class T>
T Storage<T>::getElmt(string positionCode){
    try{
        pair<int, int> position = translatePositionCode(positionCode);
        return *matrix[position.first][position.second];
    } catch (PositionCodeInvalidException e){
        startTextRed();
        cout << e.what() << endl;
        resetTextColor();
    }
}
```

```
template<class T>
T Storage<T>::getElmt(int y, int x){
   T *elmt = matrix[y][x];
   return *elmt;
}
```

b. isEmpty(string idx) dan isEmpty(int y, int x) pada kelas Storage. Fungsi ini digunakan untuk mengecek apakah suatu slot pada Storage kosong dan memiliki return boolean. Seperti penggunaan getElmt, fungsi isEmpty juga memiliki 2 macam method dengan parameter berbeda, yaitu string position untuk mengecek slot kosong dengan masukan dari pengguna berupa string, dan juga method yang memiliki parameter int y dan int x untuk mengambil nilai x dan y dari looping.

```
template <class T>
bool Storage<T>::isEmpty(string idx) {
   try {
       pair<int, int> coor = translatePositionCode(idx);
       return matrix[coor.first][coor.second] == nullptr;
   } catch (PositionCodeInvalidException e) {
       startTextRed();
       cout << e.what() << endl;
       resetTextColor();
   }
}

template <class T>
bool Storage<T>::isEmpty(int y, int x) {
   if (y >= 0 && y < numRow && x >= 0 && x < numCol){
       return matrix[y][x] == nullptr;
   } else {
       throw PositionCodeInvalidException("Indeks diluar ukuran storage");
   }
}
```

c. translatePositionCode(string idx) yang *return pair of integers* dan translatePositionCode(int row, int col) yang *return string of* position. Penggunaan dua fungsi ini untuk mengubah input pengguna dari positionCode yang bernilai string agar menjadi integer row dan col sehingga bisa diakses melalui indeks vector, dan juga untuk mengubah nilai row dan col pada fungsi yang berparameter integer untuk diubah menjadi string position untuk utilisasi fungsi yang sudah ada sehingga tidak perlu membuat banyak *overloading function*.

```
template<class T>
pair<int, int> Storage<T>::translatePositionCode(string positionCode){
    int row = (stoi(positionCode.substr(1)) - 1);
    int col = (positionCode[0] - 'A');
    if (row >= 0 && row < numRow && col >= 0 && col <numCol){
        return make_pair(row, col);
    } else {
        throw PositionCodeInvalidException("Kode posisi melampaui ukuran penyimpanan");
    }
}

template<class T>
string Storage<T>::translatePositionCode(int row, int col){
    char firstChar =  col + 'A';
    row++;
    string nextChars;
    if (row < 10){
        nextChars = '0' + to_string(row);
    } else {
        nextChars = to_string(row);
    }
    return firstChar + nextChars;
}
```

d. *Method* operator==(Building a, Building b) merupakan operator *overloading* pada kelas Building yang mengembalikan nilai boolean. *Method* ini berfungsi untuk membandingkan apakah Building a dan Building b serupa. Building a dan b dikatakan serupa apabila keduanya memiliki nama dan tipe yang sama. *Method* akan mengembalikan nilai true jika a dan b serupa, dan false jika a dan b tidak serupa.

```
bool operator==(Building a, Building b){
   return a.code == b.code && a.name == b.name;
}
```

e. *Method* operator+(T item) pada kelas storage merupakan operator *overloading* pada kelas storage yang memiliki fungsi untuk menambahkan item kepada storage. *Method* ini digunakan saat meng-*handle* item dengan berbagai tipe yang ingin ditambahkan ke storage.

```
template<class T>
void Storage<T>::operator+(T item){
   insertElmtAtEmptySlot(&item);
}
```

f. *Method* operator+(string item) pada kelas storage merupakan operator *overloading* pada kelas storage yang memiliki fungsi untuk menambahkan item kepada storage. *Method* ini digunakan saat meng-*handle* item bertipe string yang ingin ditambahkan ke storage.

```
template<>
void Storage<string>::operator+(string item){
   insertElmtAtEmptySlot(item);
}
```

g. *Method* operator+(Animal item) pada kelas storage merupakan operator *overloading* pada kelas storage yang memiliki fungsi untuk menambahkan item kepada storage. *Method* ini digunakan saat meng-*handle* item bertipe Animal yang ingin ditambahkan ke storage.

```
template<>
void Storage<Animal>::operator+(Animal item){
   insertElmtAtEmptySlot(item.getCode());
}
```

h. *Method* operator+(Plant item) pada kelas storage merupakan operator *overloading* pada kelas storage yang memiliki fungsi untuk menambahkan item kepada storage. *Method* ini digunakan saat meng-*handle* item bertipe Plant yang ingin ditambahkan ke storage

```
template<>
void Storage<Plant>::operator+(Plant item){
   insertElmtAtEmptySlot(item.getCode());
}
```

2.3. Template & Generic Classes

Generic Class adalah suatu fitur yang memungkinkan berbagai jenis tipe data menjadi parameter untuk suatu kelas ataupun fungsi. Penggunaan Generic Class dapat meningkatkan efisiensi kode dan mengurangi repetisi kode untuk tipe data yang berbeda. Implementasi Generic Class pada C++ adalah dengan menggunakan Template Class. Pada Tugas Besar ini, Template Class diimplementasikan untuk kelas Storage (penyimpanan) yang menjadi struktur data penyimpanan bagi setiap Player. Penggunaan Template Class pada kelas Storage dikarenakan Storage dapat digunakan untuk menyimpan berbagai struktur data, misalnya Ladang sebagai Storage of Plant (tanaman), Peternakan sebagai Storage of Animal, dan Storage of String untuk menyimpan kode item.

```
template <class T>
class Storage{
   private:
       /* Attributes */
       const int numRow;
```

```
        const int numCol;
        int numElmt;
        vector<vector<T*>> matrix;
… // Implementasi lainnya
```

Header kelas Storage

```
template<class T>
Storage<T>::Storage() : Storage(defaultStorageSize.first, defaultStorageSize.second) {}

template<class T>
Storage<T>::Storage(int _numRow, int _numCol) : numRow(_numRow), numCol(_numCol), numElmt(0){
   matrix.resize(numRow, vector<T*>(numCol, nullptr));
}

template<class T>
Storage<T>::~Storage(){}

template<class T>
T Storage<T>::getElmt(string positionCode){
   try{
       pair<int, int> position = translatePositionCode(positionCode);
       return *matrix[position.first][position.second];
   } catch (PositionCodeInvalidException e){
       startTextRed();
       cout << e.what() << endl;
       resetTextColor();
   }
}
```

Implementasi kelas Storage

Meskipun Storage menggunakan Template Class, namun beberapa fungsi untuk item seperti Ladang dan Peternakan memerlukan fungsi khusus karena format Cetak Ladang dan Cetak Peternakan berbeda dengan cetak penyimpanan biasa, yaitu penggunaan warna untuk mengindikasikan tanaman/hewan sudah siap dipanen. Oleh karenanya dilakukan *override* khusus untuk Cetak Ladang dan Cetak Peternakan.

```
template<class T>
void Storage<T>::printStorage(){
```

```
    resetTextColor();
    cout << "     ";
    for (int i=0; i<max(0,(numCol*3)-7); i++){
        cout << "=";
    }
    cout << "[ Penyimpanan ]";
    for (int i=0; i<max(0,(numCol*3)-7); i++){
        cout << "=";
    }
    cout << endl;
    cout << endl;
    cout << "      ";
… // Lanjutan kode untuk printStorage generic class

template<>
void Storage<Animal>::printStorage(){
    resetTextColor();
    cout << "     ";
    for (int i=0; i<max(0,(numCol*3)-6); i++){
        cout << "=";
    }
    cout << "[ Peternakan ]";
    for (int i=0; i<max(0,(numCol*3)-6); i++){
        cout << "=";
    }
    cout << endl;
    cout << endl;
    cout << "      ";
… // Lanjutan code untuk printStorage Animal


template<>
void Storage<Plant>::printStorage(){
    resetTextColor();
    cout << "     ";
    for (int i=0; i<max(0,(numCol*3)-5); i++){
        cout << "=";
    }
```

```
    cout << "[ Ladang ]";
    for (int i=0; i<max(0,(numCol*3)-4); i++){
        cout << "=";
    }
    cout << endl;
    cout << endl;
    cout << "      ";

… // Lanjutan kode printStorage untuk Plant
```

2.4. Exception

Exception adalah tipe data khusus yang mengembalikan suatu nilai saat terjadi kesalahan dalam proses *run* program. Exception akan dikembalikan dengan suatu *keyword* “throw” berdasarkan suatu alur yang sudah didefinisikan, dan saat terjadi suatu kesalahan yang sesuai dengan definisi Exception tersebut, suatu objek Exception yang dilempar tersebut harus ditangkap (*catch*) agar proses *run* program tidak diberhentikan karena *error*. Hal ini disebut sebagai *error handling*.

Pada Tugas Besar ini, Exception yang dibuat adalah Exception untuk Storage, Player, dan Game.

1. StorageException

Exception yang dibuat untuk membuat *error handling* pada penggunaan kelas Storage.

a. PositionCodeInvalidException yang digunakan saat masukan pengguna untuk mengakses Storage tidak valid, baik saat konversi dari string ke nilai posisi x dan y, maupun pengecekan apakah nilai posisi yang diberikan ada pada Storage.

```
class PositionCodeInvalidException : public exception {
  private:
      string message;

  public:
      PositionCodeInvalidException() : message("Position Code is not valid for this storage"){}
      PositionCodeInvalidException(string msg) : message(msg) {}
```

```
        string what (){
            return message;
        }
};
```

b. StorageFullException, yang digunakan saat method yang menambahkan *item* ke dalam Storage namun Storage sudah penuh dan tidak dapat ditambah lagi item ke dalamnya.

```
class StorageFullException : public exception {
   private:
        string message;

   public:
        StorageFullException() : message("Storage is already full"){}
        StorageFullException(string msg) : message(msg) {}
        string what (){
            return message;
        }
};
```

c. StorageSlotException, *exception* yang digunakan saat suatu slot pada Storage tidak dapat digunakan, misalnya jika akan *insert* elemen ke suatu slot yang penuh, atau mengambil elemen dari slot yang kosong pada Storage.

```
class StorageSlotException : public exception {
private:
   string message;

public:
   StorageSlotException() : message("Storage slot is not available"){}
   StorageSlotException(string msg) : message(msg) {}
   string what (){
       return message;
   }
```

```
};
```

2. PlayerException

Exception yang dibuat untuk melakukan *error handling* pada penggunaan kelas Player.

a. InedibleProductException, exception yang digunakan saat Player ingin memakan produk yang bukan merupakan produk yang bisa dimakan (*edible product*).

```
class InedibleProductException : public exception {
    private:
        string message;

    public:
        InedibleProductException() : message("Selected product is Inedible"){}
        InedibleProductException(string msg) : message(msg) {}
        string what (){
            return message;
        }
};
```

b. NotProductException, exception yang digunakan saat Player ingin menjual atau mengutilisasi item yang bukan produk.

```
class NotProductException : public exception {
    private:
        string message;
```

```
    public:
        NotProductException() : message("Selected item is Inedible"){}
        NotProductException(string msg) : message(msg) {}
        string what (){
            return message;
        }
};
```

3. GameException

Exception pada kelas Game untuk melakukan *error handling* pada fungsi-fungsi Game dan juga input pengguna saat Game berjalan.

a. FileNotFoundException, exception yang digunakan untuk *handling* jika proses baca config tidak berhasil karena file config tidak ditemukan.

```
class FileNotFoundException : public exception {
   private:
        string message;

   public:
        FileNotFoundException() : message("Failed to open File") {}
        FileNotFoundException(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

b. FileBadPathException, exception yang digunakan untuk *handling* jika path file tidak valid.

```
class FileBadPathException : public exception {
    private:
        string message;

    public:
        FileBadPathException() : message("File path is not valid") {}
        FileBadPathException(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

c. FileFormatException, exception yang muncul untuk *handle* kondisi saat format file yang dibaca tidak valid atau tidak sesuai.

```
class FileFormatException : public exception {
    private:
        string message;

    public:
        FileFormatException() : message("File format is not valid") {}
        FileFormatException(string msg) : message(msg) {}
        string what () {
            return message;
        }
```

```
};
```

d. FileReadingFailedException, exception yang muncul untuk *handle* saat proses baca dan *parsing* file tidak berhasil atau menemui kendala.

```
class FileReadingFailedException : public exception {
    private:
        string message;

    public:
        FileReadingFailedException() : message("File reading failed") {}
        FileReadingFailedException(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

e. DirectoryNotFoundException, exception yang dibuat untuk mengatasi *error* saat mencari sebuah direktori yang tidak ditemukan pada struktur direktori yang ada.

```
class DirectoryNotFoundException : public exception {
    private:
        string message;

    public:
```

```
        DirectoryNotFoundException() : message("Directory not found") {}
        DirectoryNotFoundException(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

f. PlayerNameIsTakken, exception yang muncul saat membuat sebuah Player baru yang menggunakan username yang sama dengan Player yang sudah ada dalam Game sehingga tidak dapat dibuat Player dengan nama yang sudah ada.

```
class PlayerNameIsTakken : public exception{
    private:
        string message;

    public:
        PlayerNameIsTakken() : message("Player Name already exist") {}
        PlayerNameIsTakken(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

g. DirectoryCreationFailedException, exception yang muncul untuk *handling* jika saat proses pembuatan sebuah direktori baru gagal.

```
class DirectoryCreationFailedException : public exception {
```

```
    private:
        string message;

    public:
        DirectoryCreationFailedException() : message("Creating directory failed, path is not correct")
{}
        DirectoryCreationFailedException(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

h. PlayerNotFound, exception yang muncul saat Game tidak menemukan Player yang dicari sehingga melempar *error message* saat kondisi ini terjadi.

```
class PlayerNotFound : public exception{
    private:
        string message;

    public:
        PlayerNotFound() : message("Player Not Found") {}
        PlayerNotFound(string msg) : message(msg) {}
        string what () {
            return message;
        }
};
```

2.5. C++ Standard Template Library

Tugas Besar ini juga menggunakan beberapa Standard Template Library (STL) lain untuk menggunakan tipe data khusus yang memudahkan operasi fungsi dan penyimpanan. Beberapa STL yang digunakan pada Tugas Besar ini adalah:

1. **Vector**

   Vector merupakan tipe STL yang merepresentasikan array dinamis pada C++. Seperti pada array biasa, pengaksesan elemen pada vektor juga menggunakan indeks. Vektor memungkinkan berbagai tipe data untuk digunakan sebagai array, sehingga penggunaan vektor sebagai tipe data untuk penyimpanan pada Storage, menyimpan data config dari file, menyimpan kumpulan Player pada Game (karena banyak Player yang bermain dapat berubah jika dilakukan fungsi Tambah Pemain). Penggunaan vektor sebagai array dinamis diperlukan pada kasus-kasus di atas karena jumlah elemen yang ada pada kumpulan data tersebut dapat berubah-ubah (dinamis) seiring keberjalanan program.

```
// Animal.cpp
vector<string> Animal::configCode;

static vector <string> plantCodeList;
// Plant.hpp

static vector <string> productCodeList;
// Product.hpp
```

Untuk menyimpan keseluruhan kode item yang ada baik pada Product, Plant, maupun Animal, digunakan vector karena seluruh code yang akan ada di permainan berasal dari file configuration yang belum diketahui banyak code yang didefinisikan, sehingga menggunakan vector sebagai array dinamis akan memudahkan pengerjaan.

```
// Game.cpp, initialization of game
farmers = vector<Farmer>();
farmers.push_back(Farmer("Petani1", 50, 40));

cattlemans = vector<Cattleman>();
```

```
cattlemans.push_back(Cattleman("Peternak1", 50, 40));
```

Selain itu, pada kelas Game yang mengatur keberjalanan permainan juga digunakan vector untuk menyimpan data seluruh player/pemain yang ada. Hal ini karena jumlah pemain dapat berubah-ubah seiring keberjalanan permainan dikarenakan Mayor dapat menambah jumlah Petani/Peternak.

```
class Storage{
   private:
       /* Attributes */
       const int numRow;
       const int numCol;
       int numElmt;
       vector<vector<T*>> matrix;
```

Vector juga digunakan pada kelas Storage karena memungkinkan penyimpanan pada kelas Storage untuk diubah ukurannya secara mudah setelah membaca file configuration selesai. Atribut matrix pada kelas Storage yang berperan sebagai tempat penyimpanan yang merepresentasikan array of array (matriks) yang dinamis. Hal ini karena saat pembacaan file configuration untuk mendapatkan ukuran penyimpanan Storage, ukuran pasti dari matrix belum dapat ditentukan. Selain itu ukuran Ladang dan Peternakan yang dapat berbeda ukurannya juga dapat di-*handle* dengan mudah jika menggunakan vector.

2. Map

Map adalah tipe data yang menunjukkan pasangan elemen berbentuk key dan value. Untuk mengakses suatu value, diperlukan sebuah key khusus yang unik. Penggunaan map pada Tugas Besar ini adalah pada atribut-atribut kelas yang nilainya berasal dari file configuration dan bersifat terikat pada identifier unik. Misalnya pada kelas Animal, Plant, dan Product, harga untuk setiap jenis item berbeda-beda satu sama lain berdasarkan jenisnya. Oleh karenanya, digunakan map dengan kode item sebagai key (karena sudah dipastikan kode item unik) dan harga sebagai value. Penggunaan map ini juga berlaku untuk atribut config lain yang bersifat statik pada kelas lainnya, seperti pasangan map kode dengan nama item, kode dengan jenis item, kode item asal dengan kode produk yang dihasilkan.

```
// Atribut statik pada Plant
vector <string> Plant::plantCodeList; // Digunakan code 3 char karena parameter unik
map <string, int> Plant::plantIDMap; // <code, map>
map <string, string> Plant::plantTypeMap; // <code, plantType>
```

```
map <string, string> Plant::plantNameMap; // <code, plantName>
map <string, int> Plant::plantHarvestDurationMap; // <code, harvestDuration>
map <string, int> Plant::plantPriceMap; // <code, price>
```

Atribut statik tempat menyimpan data konfigurasi pada implementasi kelas Plant

```
static map <string, int> productIDMap; // <code, map>
static map <string, string> productTypeMap; // <code, productType>
static map <string, string> productNameMap; // <code, productName>
static map <string, string> productOriginMap; // <code, productOrigin>
static map <string, int> productAddedWeightMap; // <code, addedWeight>
static map <string, int> productPriceMap; // <code, price>
static map <string, string> productOriginToCode; // <origin code, product code>
```

Atribut statik menyimpan data konfigurasi pada header kelas Product

```
/* Configuration Variables */
static vector<string> configCode;               // Animal Configuration Codes
static map<string, int> configID;               // Animal Configuration Key: Code, Value: Animal Id, Id starts
from 1.
static map<string, string> configName;          // Animal Configuration Names
static map<string, string> configFoodType;      // Animal Configuration Food Types
static map<string, int> configWeightToHarvest;  // Animal Configuration Weights To Harvest
static map<string, int> configPrice;
```

Atribut statik untuk menyimpan data konfigurasi pada header kelas Animal

Shop juga menggunakan pasangan map untuk menyimpan pasangan item yang dijual yang disimpan dalam bentuk string kode barang dengan jumlah barang yang tersedia di Shop. Selain itu, map juga digunakan sebagai variabel antara yang membutuhkan pasangan item dengan suatu value yang bergantung pada item tersebut, seperti pada kelas Farmer dibuat sebuah variabel sementara untuk menyimpan pasangan tanaman yang sudah bisa di-harvest dan jumlahnya.

3. Pair

Pair adalah tipe data yang menunjukkan pasangan elemen, namun tidak seperti Map, Pair tidak memerlukan suatu key untuk mengakses value-nya, melainkan kedua pasang nilai pada Pair adalah suatu nilai itu sendiri. Penggunaan Pair pada Tugas Besar ini

adalah pada pengaksesan Storage menggunakan koordinat untuk menghindari penggunaan variabel yang berlebihan. Pair pada kelas Storage tersusun atas nilai koordinat y dan nilai koordinat x yang disimpan sepasang untuk menyimpan data posisi item tersebut, sehingga dapat menghindari kebingungan kombinasi titik x dan y mana yang akan digunakan untuk mengakses lokasi suatu item. Pair juga digunakan sebagai variabel antara setelah konversi dari posisi yang berbentuk string yang berasal dari masukan pengguna lalu diubah menjadi pasangan koordinat y, x juga.

```
template<class T>
pair<int, int> Storage<T>::translatePositionCode(string positionCode){
   int row = (stoi(positionCode.substr(1)) - 1);
   int col = (positionCode[0] - 'A');
   if (row >= 0 && row < numRow && col >= 0 && col <numCol){
       return make_pair(row, col);
   } else {
       throw PositionCodeInvalidException("Kode posisi melampaui ukuran penyimpanan");
   }
}
```

Selain digunakan untuk menyimpan koordinat, Pair juga digunakan untuk menjadi atribut statik yang menyimpan definisi ukuran penyimpanan Storage yang diambil dari file configuration. File configuration menyimpan definisi ukuran Storage berukuran n x m, sehingga data tersebut akan lebih mudah jika disimpan dalam satu variabel Pair, yang nantinya diakses dengan Pair.first dan Pair.second.

```
/* Configuration Variable */
static pair<int, int> defaultStorageSize;

template<class T>
Storage<T>::Storage() : Storage(defaultStorageSize.first, defaultStorageSize.second) {}
```

4. Iterator

Iterator adalah tipe data yang digunakan untuk menunjuk *memory address* untuk STL lainnya. Iterator pada Tugas Besar ini digunakan dalam proses iterasi saat menggunakan STL seperti vector dan map. Iterator digunakan sehingga tidak perlu dilakukan iterasi manual menggunakan indeks, terutama pada tipe data seperti map yang tidak memiliki indeks tetap sebagai key-nya. Pada cuplikan kode di bawah, iterator digunakan pada tipe data map untuk

```
map<string, pair<int,int>>::iterator it = numMap.begin();
int idx = 1;
vector<string> harvestableCodes;
harvestableCodes.push_back("");
while (it != numMap.end()){
   if (it->second.second > 0){
       harvestableCodes.push_back(it->first);
       cout << "    " << idx << ". " << it->first << " (" << it->second.second << " petak siap panen)" << endl;
       idx++;
   }
   ++it;
}
```

Penggunaan iterator untuk melakukan iterasi pada seluruh item di map

```
for (map<string, int>::iterator it = this->numberOfHarvestablePlant.begin(); it !=
this->numberOfHarvestablePlant.end(); ++it) {
   if (it->second > 0) {
       penomoran[num] = it->first;
       cout << num << ". " << it->first << " (" << it->second << " petak siap panen)" << endl;
       num++;
   }
}
```

Penggunaan iterator untuk melakukan iterasi pada tipe data map dan manipulasi data

```
vector <string>::iterator it;

// Find in Plant
it = find(Plant::getPlantCodeListConfig().begin(), Plant::Plant::getPlantCodeListConfig().end(), itemCode);
```

Penggunaan iterator untuk melakukan proses pencarian item dengan menggunakan algoritma find

2.6. Konsep OOP lain

1. Abstract Base Class (Player)

Player merupakan sebuah Abstract Base Class karena memiliki method pure virtual yaitu pada fungsi buy(), sell(), dan calculateKKP() sehingga perlu diimplementasikan pada Child Class-nya yaitu Farmer, Cattleman, dan Mayor. Fungsi-fungsi di atas dijadikan virtual karena setiap child class memiliki aturannya masing-masing dalam melakukan jual-beli dan juga dalam aturan penghitungan kekayaan yang dikenakan KKP.

```
// Player.cpp
virtual int calculateKKP() = 0;

virtual void sell() = 0;

virtual void buy() = 0;
```

Pada fungsi buy(), terdapat aturan bahwa Mayor tidak dapat membeli bangunan sehingga jika fungsi buy tidak diimplementasikan secara virtual pada kelas Player, maka tidak dapat diimplementasikan aturan tersebut khusus kelas Mayor. Selain itu pada fungsi sell(), khusus untuk Farmer dan Cattleman tidak dapat menjual bangunan yang dimilikinya, oleh karenanya fungsi sell() tidak tepat jika diimplementasikan pada kelas Player, namun juga akan lebih terabstraksi bahwa setiap Player dapat melakukan buy dan sell sehingga fungsi-fungsi tersebut dijadikan virtual pada kelas Player untuk memberikan pemahaman bahwa Player dapat melakukan jual-beli meskipun implementasinya berbeda.

Hal yang sama juga terdapat pada fungsi calculateKKP() yaitu fungsi untuk menghitung besaran KKP yang akan dikenakan pajak. Hal ini karena baik Cattleman dan Farmer memiliki besaran KTKP yang berbeda-beda, bahkan Mayor tidak dikenakan pajak sama sekali. Hal ini menyebabkan fungsi calculateKKP juga dijadikan virtual, karena ada KKP yang diimplementasikan berbeda pada setiap kelas, namun khusus untuk Mayor nilainya adalah 0.

2. Aggregation

Aggregation adalah konsep OOP yang menjelaskan bahwa suatu “bagian” dari kelas merupakan kelas yang lain. Pada Tugas Besar ini, Player memiliki Storage yang menjelaskan konsep agregasi. Hal ini dikarenakan suatu objek dari kelas Storage menjadi salah satu atribut dari kelas Player, yaitu ItemStorage atau penyimpanan untuk Player.

```
class Player{
   protected:
       /* Attributes */
       static int numPlayer;
       const int id;
       const string username;
       int wealth;
       int currWeight;
       Storage<string> ItemStorage;
```

3. Algorithm

Algorithm merupakan *library* bawaan dari C++ yang digunakan untuk melakukan operasi terhadap STL pada C++.

a. Find

Penggunaan *find* pada Tugas Besar ini adalah pada proses iterasi pencarian item pada ItemStorage. ItemStorage merupakan penyimpanan pada Player yang menyimpan kode dari setiap item yang ada pada Player. Fungsi itemType berfungsi untuk return tipe data dari item pada Storage jika diberikan koordinat titik penyimpanan item pada ItemStorage tersebut. Untuk melakukan pencarian jenis item, dilakukan pencocokan kode item dengan setiap kode dari konfigurasi, dan jika ditemukan kode tersebut pada konfigurasi suatu kelas, maka kelas itu adalah tipe data item yang ditunjuk. Penggunaan fungsi *find* dari algorithm digunakan untuk mempermudah pencarian dan menghindari proses looping yang panjang.

```
string Player::itemType(int y, int x) {
   string itemCode = this->ItemStorage.getElmt(y, x);
```

```
    vector <string>::iterator it;

    // Find in Plant
    it = find(Plant::getPlantCodeListConfig().begin(), Plant::Plant::getPlantCodeListConfig().end(), itemCode);
    if (it != Plant::Plant::getPlantCodeListConfig().end()) {
        return "Plant";
    } else {
        // Find in Animal
        it = find(Animal::getAnimalCodeConfig().begin(), Animal::Animal::getAnimalCodeConfig().end(), itemCode);
        if (it != Animal::Animal::getAnimalCodeConfig().end()) {
            return "Animal";
        } else {
            it = find(Product::getProductCodeListConfig().begin(),
Product::Product::getProductCodeListConfig().end(), itemCode);
            if (it != Product::Product::getProductCodeListConfig().end()) {
                return "Product";
            } else {
                return "Building";
            }
        }
    }
}
```

3. Bonus Yang dikerjakan

3.1. Bonus yang diusulkan oleh spek

3.1.1. Generic Class

Penggunaan Generic Class pada kelas Storage merupakan penerapan konsep OOP yang memungkinkan penggunaan kelas ataupun fungsi dengan tipe data yang akan ditentukan kemudian. Penggunaan Generic Class digunakan untuk menghindari dan meminimalisir penggunaan method overloading yang menyebabkan repetisi kode berulang kali dengan logika yang sama namun hanya berbeda tipe data saja. Seperti sudah dijelaskan di subbab 2.3, penggunaan Generic Class terdapat pada kelas Storage karena tipe data yang disimpan dalam struktur data Storage dapat berbeda-beda, misalnya penyimpanan Player berbentuk string, Ladang yang menyimpan Storage of Plant, dan Peternakan yang menyimpan Storage of Animal.

```
template <class T>
class Storage{
    private:
        /* Attributes */
        const int numRow;
        const int numCol;
        int numElmt;
        vector<vector<T*>> matrix;

        /* Configuration Variable */
        static pair<int, int> defaultStorageSize;
```

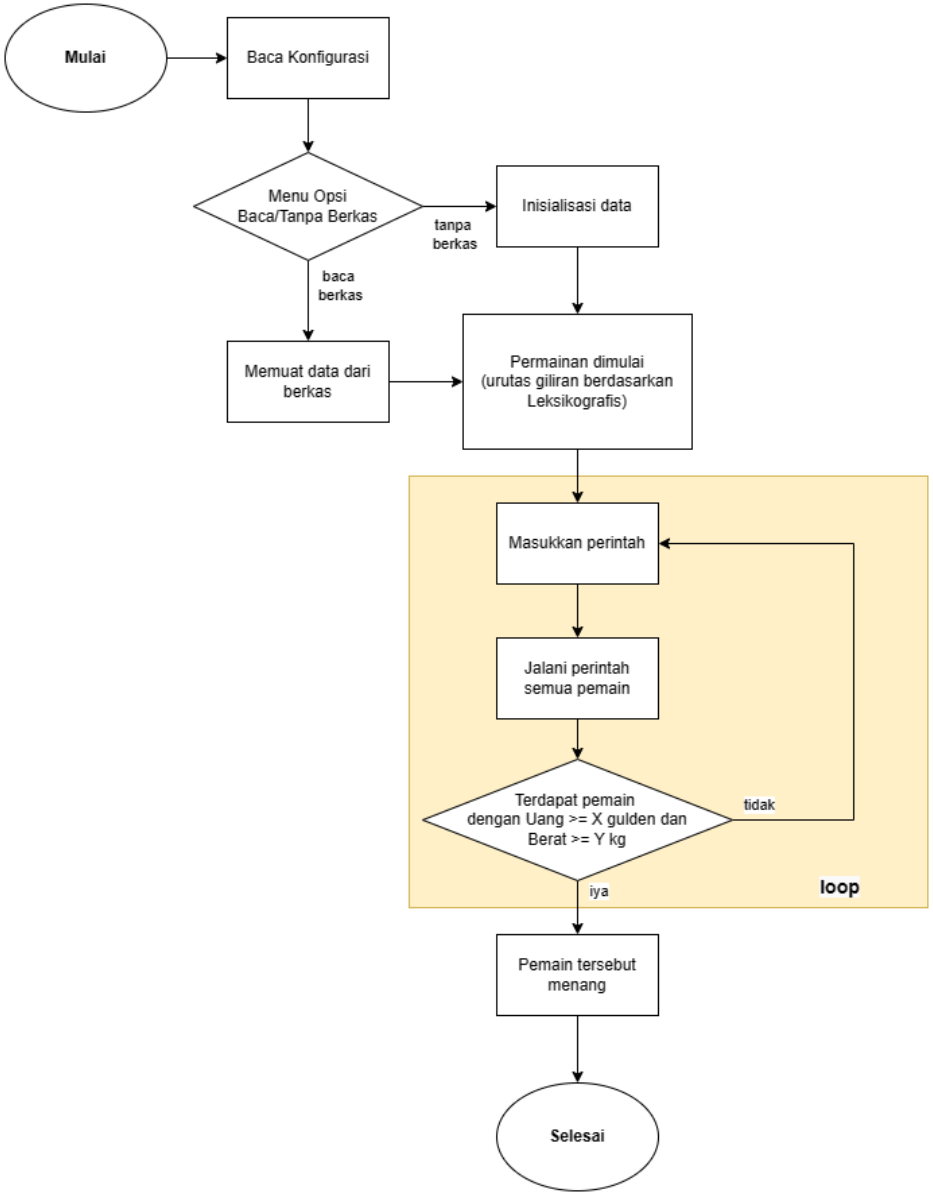
Dengan menggunakan Generic Class, tidak diperlukan pengulangan kode yang berbeda untuk logika yang sama untuk membentuk Ladang, Peternakan, dan Inventory Player.

```
class Farmer : public Player {
private:
    Storage<Plant> Ladang;
```

```
class Cattleman : public Player{
    private:
        Storage<Animal> Farm;
```

```
class Player{
    protected:
        /* Attributes */
        static int numPlayer;
        const int id;
        const string username;
        int wealth;
        int currWeight;
        Storage<string> ItemStorage;
```

3.1.2. Diagram Sistem



3.1.3.

3.2. Bonus Kreasi Mandiri

3.2.1 Black Market

Black Market merupakan tempat untuk melakukan transaksi ilegal. Tempat ini tidak diketahui oleh walikota sehingga yang dapat mengunjungi Black Market hanyalah *role* selain walikota. Black Market serupa dengan *shop* yang menjadi pusat untuk jual beli barang, hanya saja bedanya di Black Market pemain bisa membeli barang dengan harga yang lebih murah daripada harga aslinya dan menjual barang dengan harga yang lebih mahal daripada harga aslinya. Black Market jarang muncul dalam permainan, ia hanya muncul setiap 4 kali giliran bermain. Pada implementasinya di C++, Black Market adalah kelas BlackMarket dan merupakan *child* dari kelas Shop. Kelas BlackMarket memiliki 1 atribut yang bersifat *protected*, yaitu e. Apabila pemain menjual barangnya di Black Market, atribut wealth yang dimiliki pemain tersebut akan bertambah sejumlah (1 + e) kali harga barang yang ia jual. Apabila pemain membeli barang di Black Market, pemain tersebut akan mendapatkan barang yang diinginkan dengan harga (1 - e) kali barang yang ia beli. Berikut adalah *header* dari kelas BlackMarket.

```
class BlackMarket: public Shop{
   protected:
       static constexpr float e = 0.3;

   public:
       /**
        * @brief ctor user defined
        */
       BlackMarket();

       /**
        * @brief dtor
        *
        */
       ~BlackMarket();
```

```
        //**************** GETTER SETTER *****************//

        /**
         * @brief Get the Black Market Qty of item with Code = code
         *
         * @param string code
         * @param isBuy is it for buy case
         *
         * @return int Qty
         */
        int getPriceBlackMarket(string code, bool isBuy);

        //*************** Show Black Market Inventory ******************//

        void showShopNonBuildings() override;

        void showShopBuildings(int prevNumber) override;

        /**
         * @brief Show Black Market Inventory title
         */
        void showShopTitle(bool isBuy) override;
};
```

4. Pembagian Tugas

Modul (dalam poin spek)	Implementer	Tester
Next	13522091	13522037, 13522091
Cetak Penyimpanan	13522037	13522037, 13522071, 13522091
Pungut Pajak	13522105, 13522071	13522071, 13522091
Cetak Ladang dan Cetak Peternakan	13522037	13522037, 13522071, 13522091
Tanam	13522071, 10023359	13522037, 13522071
Ternak	13522037, 13522091	13522037, 13522091
Bangun Bangunan	13522105, 13522071	13522071
Makan	13522037	13522037
Memberi Pangan	13522037, 13522091	13522037, 13522091
Membeli	13522037, 13522071, 13522051	13522037, 13522071
Menjual	13522037, 13522071, 13522051	13522037, 13522091
Memanen	13522037, 13522071, 13522091	13522037, 13522071, 13522091
Muat	13522091, 13522037, 13522071, 13522105	13522037, 13522091, 13522071
Simpan	13522091	13522091
Tambah Pemain	13522091, 13522105	13522091
Penyimpanan	13522037, 13522071, 13522091	13522037, 13522071, 13522091

Pajak	13522037, 13522071, 13522105	13522037, 13522071, 13522091
[BONUS] Black Market	13522037	13522037, 13522091
Diagram UML	13522091, 13522071, 13522105	13522091, 13522071
Diagram Sistem	13522091	13522091, 13522105