Diagonal.h

```cpp
#pragma once
#include <vector>

class DifferentSize : public std::exception{};
class InvalidIndex : public std::exception{};
class ReferenceToNullPart : public std::exception{};

class Diagonal{
    private:
        std::vector<int> vec;
    public:
        Diagonal(int n) : vec(n,0)
        {

        }

        Diagonal(const std::vector<int> &v) : vec(v)
        {

        }

        friend Diagonal operator+(const Diagonal &a, const Diagonal &b)
        {
            if(a.vec.size() != b.vec.size())
            {
                throw DifferentSize();
            }

            Diagonal x(a.vec.size());

            for(int i = 0; i < c.vec.size(); i++)
            {
                c.vec[i] = a.vec[i] + b.vec[i];
            }

            return x;
        }

        friend Diagonal operator*(const Diagonal &a, const Diagonal &b)
        {
            if(a.vec.size() != b.vec.size())
            {
                throw DifferentSize();
            }

            Diagonal x(a.vec.size());

            for(int i = 0; i < c.vec.size(); i++)
            {
                c.vec[i] = a.vec[i] * b.vec[i];
            }

            return x;
        }


        int get(int x, int y) const
        {
            if(x >= vec.size() || y >= vec.size() || x < 0 || y < 0)
            {
                throw InvalidIndex();
            }
            else if(x == y)
            {
                return vec[x];
            }
            else
            {
                return 0;
            }
        }
```

```
        void set(int x, int y, int n)
        {
            if(x >= vec.size() || y >= vec.size() || x < 0 || y < 0)
            {
                throw InvalidIndex();
            }
            else if(x == y)
            {
                vec[x] = n;
            }
            else
            {
                throw ReferenceToNullPart();
            }
        }

};
```

Cactus.cpp

```
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>

using namespace std;

struct Cactus{
    std::string name, country, color;
    int height;
};

enum Status{
    abnorm, norm
};

bool read(std::ifstream &f, Cactus &c, Status &st)
{
    std::string line;
    if(getline(f,line))
    {
        std::istringstream sstream(line);
        sstream >> c.name >> c.country >> c.color >> c.height
        st = norm;
    }
    else
    {
        st = abnorm;
    }

    return st==norm;
}

int main(int argc, char* argv[])
{
    std::ifstream in("in.txt");
    //std::ifstream in(argv[1])
    if(in.fail())
    {
        std::cout << "File open error" << std::endl;
    }

    Cactus c;
    Status st;

    std::ofstream out1("out1.txt");
    std::ofstream out2("out2.txt");

    while(read(in, c, st))
    {
        if(c.height > 10)
        {
            out1 << c.name << " " << c.country << std::endl;
        }
    }

    return 0;
}
```

SeqInFile.hpp

```cpp
#include <string>
#include <fstream>
#include <sstream>

class SeqInFile
{
private:
    enum Status{
        norm, abnorm
    };

    Status st;

    std::ifstream inputStream;

public:
    int num;
    SeqInFile(std::string f)
    {
        inputStream.open(f);
        if(inputStream.fail())
        {
            std::cout << "File open error!" << std::endl;
        }
    }

    bool read()
    {
        if(inputStream >> num)
        {
            st = norm;
        }
        else
        {
            st = abnorm;
        }

        return st == norm;
    }

};

```

Vasarlas.hpp + main.cpp

```cpp
#include <string>
#include <fstream>
#include <sstream>

class SeqInFile
{
private:
    enum Status{
        norm, abnorm
    };

    Status st;

    std::ifstream inputStream;

public:

    struct Szamla{
        std::string name;
        int amount;
    };
    Szamla szamla;
```

```
    SeqInFile(std::string f)
    {
        inputStream.open(f);
        if(inputStream.fail())
        {
            std::cout << "File open error!" << std::endl;
        }
    }

    bool read()
    {
        std::string line;

        if(std::getline(inputStream,line);)
        {
            st = norm;
            std::istringstream lineStream(line;)
            szamla.amount = 0;

            lineStream >> szamla.name;

            std::string termek;
            int ar;
            while(lineStream >> termek >> ar)
            {
                szamla.amount += ar;
            }
        }
        else
        {
            st = abnorm;
        }

        return st == norm;
    }

};
```

```
#include <iostream>
#include "SeqInFile.hpp"

using namespace std;

int main()
{
    SeqInFile f("input2.txt");

    f.read();

    int bevetel = 0;
    int maxSpending = f.szamla.amount;
    std::string topSpender = f.szamla.name;

    while(f.read())
    {
        bevetel += f.szamla.amount;

        if(f.szamla.amount > maxSpending)
        {
            maxSpending = f.szamla.amount;
            topSpender = f.szamla.name;
        }
    }

    std::cout << bevetel << std::endl;

    return 0;
}
```

mintaZH.hpp + main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <sstream>
#include <string>

class InvalidFileException : public std::exception{};

class SeqInFile
{
private:
    enum Status{
        norm, abnorm;
    };

    Status st;

    std::ifstream beolvasas;

    struct Korhaz{
        std::string nev;
        int betegDB;
        int gepDB;
    };

    Korhaz korhaz;

public:
    SeqInFile(std::string fileNev)
    {
        beolvasas.open(fileNev);
        if(beolvasas.fail())
        {
            throw InvalidFileException();
        }
    }

    struct NapiAdat{
        std::string datum;
        int fertozottSzam;
        int korhazbanOSSZ;
    };

    NapiAdat napiadat;

    bool Read()
    {
        std::string sor;

        if(std::getline(beolvasas, sor))
        {
            st = norm;
            std::istringstream adatok(sor);

            adatok >> napiadat.datum >> napiadat.fertozottSzam;

            while(adatok >> korhaz.nev >> korhaz.betegDB >> korhaz.gepDB)
            {
                napiadat.korhazbanOSSZ += korhaz.betegDB;
            }

        }
        else
        {
            st = abnorm;
        }

        return st == norm;
    }
};
```

```cpp
#include <iostream>
#include "SeqInFile.hpp"

using namespace std;

int main()
{
    SeqInFile f("input.txt");

    bool tobbM5k = false;
    int maxKorhazban = 0;
    std::string maxDatum;

    while(f.Read())
    {
        tobbM5k = tobbM5k || f.napiadat.fertozottSzam > 5000;

        if(f.korhaz.betegDB > maxKorhazban)
        {
            maxKorhazban = f.korhaz.betegDB;
            maxDatum = f.napiadat.datum;
        }
    }

    std::cout << tobbM5k << std::endl << maxDatum << std::endl;

    SeqInFile f2("input.txt")

    while(f2.Read() && f2.napiadat.korhazbanOSSZ <=100)
    {

    }

    maxKorhazban = 0;
    maxDatum = "";

    int fertozottDB = 0;

    while(f2.Read())
    {
        fertozottDB += f2.napiadat.fertozottSzam;
        if(f2.napiadat.korhazbanOSSZ > maxKorhazban)
        {
            maxKorhazban = f2.napiadat.korhazbanOSSZ;
            maxDatum = f2.napiadat.datum;
        }
    }

    std::cout << fertozottDB << std::endl << maxDatum << " " << maxKorhazban;

    return 0;
}
```

Product.hpp

```cpp
#pragma once

#include <string>

class Product
{
private:
    std::string name;
    int price;
public:
    Product(std::string n, int p): name(n), price(p) {}

    std::string getName(){
        return name;
    }

    int getPrice(){
        return price;
    }
};
```

Department.hpp

```cpp
#pragma once

#include "Product.hpp"
#include <vector>
#include <fstream>

class Department
{
public:
    std::vector<Product*> stock;

    Department(std::string fileName)
    {
        std::ifstream f(filename);
        std::string name;
        int price;

        while(f>>name>>price)
        {
            stock.push_back(new Product(name,price));
        }
    }

    void takeOutFromStock(Product *product)
    {
        bool productFound = false;
        int index = 0;

        while(index < stock.size() && !productFound)
        {
            if(stock[index] == product)
            {
                productFound = true;
            }
            index++;
        }
        index--;

        if(productFound)
        {
            stock.erase(stock.begin()+index);
        }
    }
}
```

Store.hpp

```cpp
#pragma once

#include "Department.hpp"

class Store
{
public:
    Department *foods;
    Department *technical;

    Store(std::string foodsFile, std::string technicalFile)
    {
        foods = new Department(foodsFile);
        technical = new Department(technicalFile);
    }

    ~Store()
    {
        delete foods;
        delete technical;
    }
};
```

Costumer.hpp

```cpp
#pragma once

include "Store.hpp"
include <fstream>
include <iostream>
include <string>

class Costumer{

private:
    std::vector<std::string> list;

    void buy(Product *p, Department *d)
    {
        d->takeOutFromStock(p);
        std::cout << p->getName() << " " << p->getPrice() << std::endl;
    }

public:
    Costumer(std::string fileName)
    {
        std::ifstream f(fileName);

        std:: string s;
        while(f >> s){
            list.push_back(s);
        }
    }

    void goShopping(Store &st)
    {
        for(std::string productName : list)
        {
            Product *product;
            if(search(productName, s.foods, product))
            {
                buy(product, s.foods);
            }
            if(minsearch(productName, s.technical, product))
            {
                buy(product, s.technical);
            }
        }
    }

    bool search(std::string name, Department *d, Product* &p)
    {
        bool l = false;

        for(Product *p : d->stock)
        {
            if(name == p->getName())
            {
                l = true;
                product = p;
                break;
            }
        }

        return l;
    }
```

```
    bool minsearch(const std::string &name, Department* d, Product* &Product) const
    {
        bool l = false;
        int MIN;

        for(Product *p : d->stock)
        {
            if(p->getName() != name) continue;
            if(l)
            {
                if(MIN > p->getPrice())
                {
                    MIN = p->getPrice();
                    product = p;
                }
            }
            else{
                l = true;
                MIN = p->getPrice();
                product = p;
            }
        }

        return l;
    }
};
```

main.cpp

```
#include "Costumer.hpp"
#include "Store.hpp"

using namespace std;

int main()
{

    Costumer c("costumer.txt");
    Store s("food.txt", "technical.txt");

    c.goShopping(s);

    return 0;
}
```

Plants.hpp

```
#pragma once

class Plant{
protected:
    int ripeningTime;
    Plant(int r) : ripeningTime(r) {}

public:
    virtual bool isVegetable() const {return false;}
    virtual bool isFlower() const {return false;}
    int getRipeningTime() const {return ripeningTime;}

    virtual ~Plant() {}

};
```

```
class Vegetable : public Plant{
protected:
    Vegetable(int r) : Plant(r) {}

public:
    bool isVegetable() const override {return false;}
};

class Flower : public Plant{
protected:
    Flower(int r) : Plant(r) {}

public:
    bool isFlower() const override {return false;}
};

class Potato : public Vegetable{
private:
    static Potato* _inst;
    Potato() : Vegetable(5) {}

public:
    static Potato* inst();
    static void destroy();
};

class Pea : public Vegetable{
private:
    static Pea* _inst;
    Pea() : Vegetable(3) {}

public:
    static Pea* inst();
    static void destroy();
};

class Onion : public Vegetable{
private:
    static Onion* _inst;
    Onion() : Vegetable(4) {}

public:
    static Onion* inst();
    static void destroy();
};

class Rose : public Flower{
private:
    static Rose* _inst;
    Rose() : Vegetable(8) {}

public:
    static Rose* inst();
    static void destroy();
};

class Carnation : public Flower{
private:
    static Carnation* _inst;
    Carnation() : Vegetable(10) {}

public:
    static Carnation* inst();
    static void destroy();
};

class Tulip : public Flower{
private:
    static Tulip* _inst;
    Tulip() : Vegetable(7) {}

public:
    static Tulip* inst();
    static void destroy();
};
```

Plants.cpp

```cpp
#include "plants.hpp"

Potato* Potato::_inst = nullptr;
Potato* Potato::inst(){
    if(_inst == nullptr){
        _inst = new Potato();
    }
    return _inst;
}

void Potato::destroy()
{
    if(_inst != nullptr){
        delete _inst;
        _inst = nullptr;
    }
}

Pea* Pea::_inst = nullptr;
Pea* Pea::inst(){
    if(_inst == nullptr){
        _inst = new Pea();
    }
    return _inst;
}

void Pea::destroy()
{
    if(_inst != nullptr){
        delete _inst;
        _inst = nullptr;
    }
}

Onion* Onion::_inst = nullptr;
Onion* Onion::inst(){
    if(_inst == nullptr){
        _inst = new Onion();
    }
    return _inst;
}

void Onion::destroy()
{
    if(_inst != nullptr){
        delete _inst;
        _inst = nullptr;
    }
}

Rose* Rose::_inst = nullptr;
Rose* Rose::inst(){
    if(_inst == nullptr){
        _inst = new Rose();
    }
    return _inst;
}

void Rose::destroy()
{
    if(_inst != nullptr){
        delete _inst;
        _inst = nullptr;
    }
}

Carnation* Carnation::_inst = nullptr;
Carnation* Carnation::inst(){
    if(_inst == nullptr){
        _inst = new Carnation();
    }
    return _inst;
}

```

```cpp
void Carnation::destroy()
{
    if(_inst != nullptr){
        delete _inst;
        _inst = nullptr;
    }
}

Tulip* Tulip::_inst = nullptr;
Tulip* Tulip::inst(){
    if(_inst == nullptr){
        _inst = new Tulip();
    }
    return _inst;
}

void Tulip::destroy()
{
    if(_inst != nullptr){
        delete _inst;
        _inst = nullptr;
    }
}
```

Parcel.hpp

```cpp
#pragma once

#include "plants.hpp"
#include <iostream>

class Parcel{
private:
    Plant* _p;
    int plantingDate;

public:
    Parcel() : _p(nullptr), plantingDate(0) {}

    void plant(Plant *p, int date){
        if(_p == nullptr){
            _p = p;
            plantingDate = date;
        } else{
            std::cout << "Plant already planted here!" << std::endl;
        }
    }

    void harvest() {_p = nullptr;}

    bool isRipe(int month){
        if(_p != nullptr && (month - plantingDate) == _p->getRipeningTime()){
            return true;
        }
        return false;
    }
};
```

Garden.hpp

```cpp
# pragma once

#include <vector>
#include "parcel.hpp"

class Garden{
private:
    std::vector<Parcel*> parcels;

public:
    Garden(int n){
        if(n>=1){
            parcels.resize(n);

            for(int i=0; i<n; i++){
                parcels[i] = new Parcel();
            }
        }
    }

    std::vector<int> canHarvest(int date){
        std::vector<int> result;

        for(int i=0; i<parcels.size(); i++){
            if(parcels[i]->isRipe(date)){
                result.push_back(i);
            }
        }
        return result;
    }

    Parcel* getParcel(int i) const{
        return parcels[i];
    }

    ~Garden(){
        for(Parcel* p: parcels){
            delete p;
        }
    }
};

```

Gardner.hpp

```cpp
#pragma once

#include "garden.hpp"

class Gardner{
public:
    Garden* _g;

    Gardner(Garden g) : _g(g) {}

    void harvest(int i){
        _g->getParcel(i)->harvest();
    }

    void plant(int i, Plant* p, int date){
        _g->getParcel(i)->plant(p, date);
    }
};

```

main.cpp

```cpp
#include <iostream>
#include "gardner.hpp"
#include "plants.hpp"

using namespace std;

int main()
{
    Garden* garden = new Garden(5);
    Gardner* gardner = new Gardner(garden);

    gardner->plant(1, Potato::inst(), 3);
    gardner->plant(2, Pea::inst(), 3);
    gardner->plant(4, Rose::inst(), 3);

    std::cout << "A betakarítható parcellák azonosítói: ";
    for(int i=0 : gardner->_g->canHarvest(6)){
        std::cout << i << " ";
    }
    std::cout << std::endl;

    delete gardner;
    delete garden;

    Potato::destroy;
    Pea::destroy;
    Rose::destroy;

    return 0;
}
```

Animal.hpp

```cpp
#pragma once

class Animal{
public:
    Animal(double w, bool l) : _weight(w), _male(l) {}

    virtual bool isLion() const {return false;}
    virtual bool isRhino() const {return false;}
    virtual bool isElephant() const {return false;}
    virtual ~Animal(){}

    double _weight;
    bool _male;
};

class Lion : public Animal{
public:
    Lion(double w, bool l) : Animal(w,l) {}
    bool isLion const override {return true;}
};

class Rhino : public Animal{
public:
    Rhino(double w, bool l, double h) : Animal(w,l), _horn(h) {}
    bool isRhino const override {return true;}

    double _horn;
};

class Elephant : public Animal{
public:
    Elephant(double w, bool l, double lt, double rt) : Animal(w,l), _leftTusk(lt), _rightTusk(rt) {}
    bool isElephant const override {return true;}

    double _leftTusk, _rightTusk;
};
```

Trophy.hpp

```cpp
#pragma once
#include <string>
#include "Animal.hpp"

class Trophy{
public:
    std::string place;
    std::string date;
    Animal* animal;

    Trophy(Animal* a, std::string p, std::string d) : animal(a), place(p), date(d) {}

    ~Trophy(){
        if(animal != nullptr){
            delete animal;
        }
    }
};

```

Hunter.hpp

```cpp
#pragma once

#include "Trophy.hpp"
#include <vector>

class Hunter{
public:
    std::string name;
    int age;
    std::vector<Trophy*> trophies;

    Hunter(std::string n, int a): name(n), age(a) {}

    void capture(Animal* a, std::string p, std::string d);
    int countMaleLions() const;
    float maxHornWeightRatio() const;
    bool searchEqualTusks() const;

    ~Hunter();
};

```

Hunter.cpp

```cpp
#include "Hunter.hpp"

void Hunter::capture(Animal* a, std::string p, std::string d){
    trophies.push_back(new Trophy(a,p,d))
}

int Hunter::countMaleLions() const{
    int count = 0;
    for(Trophy* t : trophies){
        if(t->animal->isLion() && t->animal->_male){
            count++;
        }
    }
    return count;
}
```

```cpp
float Hunter::maxHornWeightRatio() const{
    float maxRate = -1;
    float rate;

    for(Trophy* t: trophies){
        if(t->animal->isRhino){
            Rhino* rhino = (Rhino*)t->animal;
            rate = rhino->_horn/rhino->_weight;
            if(rate > maxRate){
                maxRate = rate;
            }
        }
    }


    return maxRate;
}

bool Hunter::searchEqualTusks() const{
    bool l = false;
    for(Trophy* t: trophies){
            if(t->animal->isElephant()){
                Elephant* e = (Elephant*)t->animal;
                l = e->_leftTusk == e->_rightTusk;
            }

        if(l){
            break;
        }
    }
    return l;
}

~Hunter(){
    for(Trophy* t: trophies){
        delete t;
    }
}
```

main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <sstream>

#include "Hunter.hpp"

void read(Hunter &h, std::string fileName)
{
    std::ifstream f(fileName);
    std::string line, place, date, species, gender;
    double weight, lTusk, rTusk, horn;
    Animal* animal;

    while(getline(f,line)){
        std::istringstream stringStream(line);

        stringStream >> place >> date >> species >> weight >> gender;

        if(species == "lion"){
            animal = new Lion(weight, gender == "male");
        } else if(species == "rhino"){
            stringStream >> horn;
            animal = new Rhino(weight,gender =="male", horn)
        } else if(species == "elephant"){
            stringStream >> lTusk >> rTusk;
            animal = new Elephant(weight, gender == "male", lTusk, rTusk);
        }

        h.capture(animal,place,date);
    }
}

int main()
{

    Hunter hunter("Bill", 55);
    read(hunter, "input.txt");

    std::cout << hunter.countMaleLions() << std::endl;
    std::cout << hunter.maxHornWeightRatio() << std::endl;
    std::cout << hunter.searchEqualTusks() << std::endl;

    return 0;
}
```

SolarSystem.hpp

```hpp
#pragma once

#include <vector>

class Planet;
class StarShip;

class SolarSystem{
private:
    SolarSystem(){}
    static SolarSystem* _instance;

public:
    std::vector<Planet*> planets;

    static SolarSystem* Intsance(){
        if(_instance == nullptr){
            _instance = new SolarSystem();
        }
        return _instance;
    }

    static void destroy(){
        if(_instance != nullptr){
            delete _instance;
        }
    }

    ~SolarSystem();

    bool MaxFirepower(StarShip* &bestShip) const;
    Planet* Defenseless() const;
};
```

SolarSystem.cpp

```cpp
#include "SolarSystem.hpp"
#include "Planet.hpp"

SolarSystem* SolarSystem::_instance = nullptr;

bool SolarSystem::MaxFirepower(StarShip* &bestShip) const{
    bool l = false;
    int maxFp;

    for(Planet* p:planets){
        int power;
        StarShip* ship;

        bool ll = p->MaxFirepower(power,ship);

        if(!ll){
            continue;
        } else if(!l){
            l = true;
            maxFp = power;
            bestShip = ship;
        } else if(power > maxFp){
            maxFp = power;
            bestShip = ship;
        }
        return l;
    }
}

Planet* SolarSystem::Defenseless() const{
    for(Planet* p:planets){
        if(p->ships.size() == 0){
            return p;
        }
    }

    return nullptr;
}

SolarSystem::~SolarSystem(){
    for(Planet* p : planets){
        delete p;
    }
}
```

Planet.hpp

```cpp
#pragma once

#include <string>
#include <vector>

class Starship;

class Planet{
public:
    std::string name;
    std::vector<StarShip*> ships;

    Planet(std::string n) : name(n){}
    bool MaxFirepower(int &maxFp, StarShip* &ship) const;
    int TotalShields() const;

    ~Planet();
};
```

Planet.cpp

```cpp
#include "Planet.hpp"
#include "StarShip.hpp"

bool Planet::MaxFirepower(int &maxFp, StarShip* &ship) const{
    bool l = false;

    for(StarShip* ss : ships){
        int fp = ss->Firepower();
        if(!l){
            l = true;
            maxFp = fp;
            ship = ss;
        } else if(fp > maxFp){
            maxFp = fp;
            ship = ss;
        }
    }

    return l;
}

int Planet::TotalShields() const{
    int sum = 0;
    for(StarShip* ss : ships){
        sum += ss->_shield;
    }
    return sum;
}

Planet::~Planet(){
    for(StarShip *ss : ships){
        delete s;
    }
}
```

Starship.hpp

```cpp
#pragma once

#include "Planet.hpp"

class StarShip{
public:
    std::string _name;
    int _shield;
    int _armor;
    int _spaceguard;
    Planet* _planet;

    StarShip(std::string name, int shield, int armor, int guard) : _name(name), _shield(shield), _armor(armor), _spaceguard(guard) {}
    virtual ~StarShip(){}

    void Protect(Planet* p){
        _planet = p;
        p->ships.push_back(this);
    }

    virtual int Firepower() const = 0;
};

class Breaker : public StarShip
{
public:
    Breaker(std::string name, int shield, int armor, int guard) : StarShip(name, shield, armor, guard) {}
    int Firepower() const override {return _armor/2;}
};

class Lander : public StarShip
{
public:
    Breaker(std::string name, int shield, int armor, int guard) : StarShip(name, shield, armor, guard) {}
    int Firepower() const override {return _spaceguard;}
};

```

```cpp
class Laser : public StarShip
{
public:
    Breaker(std::string name, int shield, int armor, int guard) : StarShip(name, shield, armor, guard) {}
    int Firepower() const override {return _shield;}
};

```

main.cpp

```cpp
#include <iostream>
#include <fstream>

#include "SolarSystem.hpp"
#include "Planet.hpp"
#include "StarShip.hpp"

using namespace std;

int main()
{
    SolarSystem* ss = SolarSystem::Instance();

    std::ifstream f("input.txt");

    std::string pName, sName, sType;
    int shield, armor, guard;
    int n, m;

    f >> n;

    for(int i=0; i<n; i++){
        f >> pName >> m;

        Planet* p = new Planet(pName);

        for(int j = 0; j<m; j++){
            f >> sName >> sType >> shield >> armor >> guard;
            StarShip* sh;
```

```
30
31              if(sType == "Breaker"){
32                  sh = new Breaker(sName, shield,armor,guard);
33              } else if(sType == "Lander"){
34                  sh = new Lander(sName, shield,armor,guard);
35              } else if(sType == "Laser"){
36                  sh = new Laser(sName, shield,armor,guard);
37              }
38
39              sh->Protect(p);
40          }
41
42          ss->planets.push_back(p);
43      }
44
45      StarShip* ship;
46      if(ss->MaxFirepower(ship)){
47          std::cout << ship->_name << std::endl;
48      }
49
50      for(Planet* p:ss->planets){
51          if(p == "Earth"){
52              std::cout << p->TotalShields() << std::endl;
53          }
54      }
55
56      std::cout << ss->Defenseless()->name << " is defenseless!" << std::endl;
57  }
58
```

Signal.hpp

```
1  #pragma once
2
3  enum Signal{open, close, start, stop, press};
4
```

Magnetron.hpp

```
1  #pragma once
2
3  #include "Signal.hpp"
4
5  class Microwave;
6
7  enum MagnetronState {on, off};
8
9  class Magnetron{
10
11      Microwave* micro;
12
13      MagnetronState = off;
14
15 public:
16      Magnetron(Microwave* m) : micro(m) {}
17      void send(Signal sig);
18
19 }
20
```

Magnetron.cpp

```cpp
#include "Magnetron.hpp"
#include "Signal.hpp"
#include "Microwave.hpp"

void Magnetron::send(Signal sig){
    if(state == on){
        switch(sig){
        case press:
            state = off;
            micro->lamp->send(Signal stop);
            break;
        case open:
            state = off;
            break;
        default:
            break;
        }
    } else if(state == off){
        switch(sig){
        case press:
            state = on;
            micro->lamp->send(Signal start);
            break;
        default:
            break;
        }
    }
}
```

Lamp.hpp

```cpp
#pragma once

#include "Signal.hpp"

class Microwave;

enum LampState {lit, notLit};

class Lamp{

    Microwave* micro;

    LampState state = notLit;

public:
    Lamp(Microwave* m) : micro(m) {}
    void send(Signal sig);

};
```

Lamp.cpp

```cpp
#include "Lamp.hpp"
#include "Signal.hpp"
#include "Microwave.hpp"

void Lamp::send(Signal sig){
    switch(sig){
    case open:
        state = lit;
        break;
    case close:
        state = notLit;
        break;
    case start:
        state = lit;
        break;
    case stop:
        state = notLit;
        break;
    default:
        break;
    }
}
```

Door.hpp

```cpp
#pragma once

#include "Signal.hpp"

class Microwave;

enum DoorState {opened, closed};

class Door{

    Microwave* micro;

    DoorState state = closed;

public:
    Door(Microwave* m) : micro(m) {}
    void open();
    void close();
}
```

Door.cpp

```cpp
#include "Button.hpp"
#include "Signal.hpp"
#include "Microwave.hpp"

void Button::press(){
    micro->magnetron->send(Signal::press)
}
```

Button.hpp

```hpp
#pragma once

#include "Signal.hpp"

class Microwave;

class Button{

    Microwave* micro;

public:
    Button(Microwave* m) : micro(m) {}

    void press();

}
```

Button.cpp

```cpp
#include "Button.hpp"
#include "Signal.hpp"
#include "Microwave.hpp"

void Button::press(){
    micro->magnetron->send(Signal::press)
}
```

Microwave.hpp

```hpp
#pragma once

#include "Magnetron.hpp"
#include "Lamp.hpp"
#include "Door.hpp"
#include "Button.hpp"

class Microwave{
public:
    Magnetron* magnetron;
    Lamp* lamp;
    Door* door;
    Button* button;

    Microwave(){
        magnetron = new Magnetron(this);
        lamp = new Lamp(this);
        door = new Door(this);
        button = new Button(this);
    }

    ~Microwave(){
        delete magnetron;
        delete lamp;
        delete door;
        delete button;
    }


};
```

Vaccine.hpp

```cpp
#pragma once

#include <string>

class Vaccine
{
private:
    /* data */
public:
    int expirationDate;
    Vaccine(int d): expirationDate(d) {}

    virtual std::string Name() = 0;
    virtual int Repetition() = 0;

    virtual ~Vaccine(){}
};

class Pfizer : public Vaccine
{
    public:

    std::string Name() override{
        return "Pfizer";
    }

    int Repetition() override{
        return 28;
    }

    Pfizer(int d): Vaccine(d) {}
};

class Astra : public Vaccine
{
    public:

    std::string Name() override{
        return "Astra";
    }

    int Repetition() override{
        return 21;
    }

    Astra(int d): Vaccine(d) {}
};

class Moderna : public Vaccine
{
    public:

    std::string Name() override{
        return "Moderna";
    }

    int Repetition() override{
        return 84;
    }

    Moderna(int d): Vaccine(d) {}
};
```

Vaccination.hpp

```cpp
#pragma once

#include "Vaccine.hpp"

class Vaccination
{
private:
    /* data */
public:
    int date;
    Vaccine* vaccine;
};
```

Patient.hpp

```cpp
#pragma once

#include<vector>
#include "Vaccination.hpp"

class Patient
{
private:
    /* data */
public:
    std::vector<Vaccination*> vaccinations;
    std::string TAJ;

    int NumOfVacc()
    {
        return vaccinations.size();
    }
    Patient(std::string t): TAJ(t) {}
    ~Patient();
};
```

Hospital.hpp

```cpp
#pragma once

#include <iostream>

#include "Vaccine.hpp"
#include "Patient.hpp"

class Hospital
{
public:
    std::string name;
    std::vector<Vaccine*> vaccines;
    std::vector<Patient*> registered;
    Hospital();

    void Procure2(Vaccine* v)
    {
        vaccines.push_back(v);
    }
```

```
20
21      void Procure(std::string vName)
22      {
23          if(vName == "Pfizer")
24          {
25              std::cout << "Vaccine Procured" << std::endl;
26              vaccines.push_back(new Pfizer(10));
27          }
28          else if(vName == "Astra")
29          {
30              std::cout << "Vaccine Procured" << std::endl;
31              vaccines.push_back(new Astra(10));
32          }
33          else if(vName == "Moderna")
34          {
35              std::cout << "Vaccine Procured" << std::endl;
36              vaccines.push_back(new Moderna(10));
37          }
38          else
39          {
40              std::cout << "Vaccine cannot be procured" << std::endl;
41          }
42      }
43      void Register(Patient* p)
44      {
45          bool l = false;
46          for(Patient* e : registered)
47          {
48              if(e->TAJ == p->TAJ)
49              {
50                  l = true;
51                  std::cout << "Patient already registered" << std::endl;
52                  break;
53              }
54          }
55          if(!l)
56          {
57              std::cout << "Patient registered" << std::endl;
58              registered.push_back(p);
59          }
60      }
61      void Vaccinate(Patient* p, std::string vName/*, int date*/)
62      {
63          bool l1 = false;
64          bool l2 = false;
65
66          Vaccine* vaccine;
67
68          for(Vaccine* e : vaccines)
69          {
70              if(e->Name() == vName /*&& e->expirationDate >= date*/)
71              {
72                  l1 = true;
73                  vaccine = e;
74                  break;
75              }
76          }
77
78          for(Patient* e : registered)
79          {
80              if(e->TAJ == p->TAJ)
81              {
82                  l2 = true;
83                  break;
84              }
85          }
```

```
        if(l1 && l2)
        {
            //vaccines.erase(std::remove(vaccines.begin(), vaccines.end(), vaccine), vaccines.end());
            for(int i = 0; i<vaccines.size();i++)
            {
                if(vaccines[i] == vaccine)
                {
                    vaccines[i] = vaccines.back();
                    vaccines.pop_back();
                    //break;
                }
            }

            Vaccination* v = new Vaccination();
            v->vaccine = vaccine;
            p->vaccinations.push_back(v);
            std::cout << "Patient vaccinated" << std::endl;
        }
        else
        {
            std::cout << "Vaccine not found or Patient not registered" << std::endl;
        }
    }

    int NumOfMultiple()
    {
        int count = 0;
        for(Patient* e : registered)
        {
            if(e->NumOfVacc() >= 2)
            {
                count++;
            }
        }

        return count;
    }
    ~Hospital();
};
```

main.cpp

```cpp
#include<iostream>
#include "Hospital.hpp"

int main()
{
    Hospital* h = new Hospital();
    Patient* p1 = new Patient("1");
    Patient* p2 = new Patient("2");
    Patient* p3 = new Patient("3");

    h->Procure("Pfizer");
    h->Procure("Pfizer");
    h->Procure("Pfizer");
    h->Procure("Pfizersd");

    h->Register(p1);
    h->Register(p2);
    h->Register(p2);

    h->Vaccinate(p1,"Pfizer");
    h->Vaccinate(p2,"Pfizer");
    h->Vaccinate(p2,"Pfizer");
    h->Vaccinate(p2,"Pfizer");
    h->Vaccinate(p1,"Astra");

    std::cout << h->NumOfMultiple() << std::endl;
    return 0;
}
```

Ground.hpp

```hpp
#pragma once

#include <string>

class Greenfinch;
class DuneBeetle;
class Squelchy;

// class of abstract grounds
class Ground{
public:
    virtual Ground* change(Greenfinch *p) = 0;
    virtual Ground* change(DuneBeetle *p) = 0;
    virtual Ground* change(Squelchy   *p) = 0;
    virtual ~Ground() {}
    static Ground* create(int k);
};

// class of sand
class Sand : public Ground
{
public:
    static Sand* instance();
    Ground* change(Greenfinch *p) override;
    Ground* change(DuneBeetle *p) override;
    Ground* change(Squelchy   *p) override;
    void static destroy() { if ( nullptr!=_instance ) delete _instance; _instance = nullptr; }
private:
    Sand(){}
    static Sand* _instance;
};
```

```hpp
// class of grass
class Grass : public Ground
{
public:
    static Grass* instance();
    Ground* change(Greenfinch *p) override;
    Ground* change(DuneBeetle *p) override;
    Ground* change(Squelchy   *p) override;
    static void destroy() { if ( nullptr!=_instance ) delete _instance; _instance = nullptr; }
private:
    Grass(){}
    static Grass* _instance;
};

// class of marsh
class Marsh : public Ground
{
public:
    static Marsh* instance();
    Ground* change(Greenfinch *p) override;
    Ground* change(DuneBeetle *p) override;
    Ground* change(Squelchy   *p) override;
    void static destroy() { if ( nullptr!=_instance ) delete _instance; _instance = nullptr; }
private:
    Marsh(){}
    static Marsh* _instance;
};
```

Ground.cpp

```cpp
#include "ground.h"
#include "creature.h"

using namespace std;

Ground* Ground::create(int k)
{
    switch (k)
    {
        case 0: return Sand::instance();
        case 1: return Grass::instance();
        case 2: return Marsh::instance();
    }
    return nullptr;
}
```

```cpp
// implementation of class Sand
Sand* Sand::_instance = nullptr;
Sand* Sand::instance()
{
    if(_instance == nullptr) {
        _instance = new Sand();
    }
    return _instance;
}

Ground* Sand::change(Greenfinch *p)
{
    p->changePower(-2);
    return this;
}

Ground* Sand::change(DuneBeetle *p)
{
    p->changePower(3);
    return this;
}

Ground* Sand::change(Squelchy *p)
{
    p->changePower(-5);
    return this;
}

// implementation of class Grass
Grass* Grass::_instance = nullptr;
Grass* Grass::instance()
{
    if(_instance == nullptr) {
        _instance = new Grass();
    }
    return _instance;
}

Ground* Grass::change(Greenfinch *p)
{
    p->changePower(1);
    return this;
}

Ground* Grass::change(DuneBeetle *p)
{
    p->changePower(-2);
    return Sand::instance();
}

Ground* Grass::change(Squelchy *p)
{
    p->changePower(-2);
    return Marsh::instance();
}
```

```cpp
// implementation of class Marsh
Marsh* Marsh::_instance = nullptr;
Marsh* Marsh::instance()
{
    if(_instance == nullptr) {
        _instance = new Marsh();
    }
    return _instance;
}

Ground* Marsh::change(Greenfinch *p)
{
    p->changePower(-1);
    return Grass::instance();
}

Ground* Marsh::change(DuneBeetle *p)
{
    p->changePower(-4);
    return Grass::instance();
}

Ground* Marsh::change(Squelchy *p)
{
    p->changePower(6);
    return this;
}

```

Creature.hpp

```cpp
#pragma once

#include <fstream>
#include <string>
#include "ground.h"

// class of abstract creatures
class Creature{
protected:
    std::string _name;
    int _power;
    Creature (const std::string &str, int e = 0) :_name(str), _power(e) {}
public:
    std::string name() const { return _name; }
    bool alive() const { return _power > 0; }
    void changePower(int e) { _power += e; }
    virtual void transmute(Ground* &court) = 0;
    virtual ~Creature () {}
    static Creature* create(char ch, const std::string name, int p);
};

// class of green finches
class Greenfinch : public Creature {
public:
    Greenfinch(const std::string &str, int e = 0) : Creature(str, e){}
    void transmute(Ground* &court) override {
        court = court->change(this);
    }
};
```

```
35  // class of dune beetles
36  class DuneBeetle : public Creature {
37  public:
38      DuneBeetle(const std::string &str, int e = 0) : Creature(str, e){}
39      void transmute(Ground* &court) override {
40          court = court->change(this);
41      }
42  };
43
44  // class of squelchies
45  class Squelchy : public Creature {
46  public:
47      Squelchy(const std::string &str, int e = 0) : Creature(str, e){}
48      void transmute(Ground* &court) override{
49          court = court->change(this);
50      }
51  };
52
```

Creature.cpp

```
5   #include "creature.h"
6
7   Creature* Creature::create(char ch, const std::string name, int p)
8   {
9       switch (ch)
10      {
11          case 'G': return new Greenfinch(name, p);
12          case 'D': return new DuneBeetle(name, p);
13          case 'S': return new Squelchy(name, p);
14      }
15      return nullptr;
16  }
17
```

beadandó.cpp

```
int createPlanet(const std::string fileName, std::vector<Plant*> &plants){
    ifstream f(fileName);
    if(f.fail()){
        std::cout << "Sowwy but I can't find the file o.o" << std::endl;
        exit(1);
    }

    int plantDB, dayDB, nut;
    std::string name;
    char type;

    f >> plantDB;
    plants.resize(plantDB);

    for(int i=0; i<plantDB; ++i){
        f >> name >> type >> nut;
        plants[i] = Plant::create(name, type, nut);
    }

    f >> dayDB;
    return dayDB;
}

void destroyRad(Radiant* &rad){
    if(rad->isAlpha()){
        Alpha::destroy();
    } else if(rad->isDelta()){
        Delta::destroy();
    } else if (rad->isNoRad()){
        NoRad::destroy();
    }
}
```