МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе № 5

по дисциплине «Объектно-ориентированное программирование»

Тема: Построение логистики.

Студент гр.7304 _____ Давыдов А.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2019

1. Постановка задачи

1.1. Цель работы

Реализовать программу для симуляции товаров и пассажиров между городами.

1.2. Формулировка задачи

Набор городов задается в виде графа. Вершина описывает один город. Каждый город характеризуется набором складов для определенного типа товаров и вместимостью склада, а также наличием стоянки под определенный тип транспорта. Ребра графа задают тип связи между городами (*наземный, водный и воздушный*) и расстояние.

Транспорт может быть трех типов (наземный, водный, воздушный). В рамках одного типа транспорт различается по следующим параметрам:

- Какой типы товары может перевозить и в каком кол-ве. Один транспорт может перевозить разные товары одновременно.
- Возможностью перевозить пассажиров
- Стоимость перевозки на единицу расстояния

Товары характеризуются иерархией, например, 1 уровень - жидкие, твердые, газообразные. Твердыми товарами (2 уровень) могут быть металлы, стройматериалы, и.т.д. Металлы (3 уровень) алюминий, медь и.т.д.

Программа должна позволять пользователю создать карту городов, вывести текущее состояние (какие товары и транспорт находятся в городах), а также проводить транспортировку транспорта с товаром и без (если это возможно). Создавать и удалять транспорт.

Минимальные требования к заданию:

1. Иерархия товаров глубинной 3, на каждом уровне по 2 товара
2. По 2 транспорта каждого типа (*например, грузовик и поезд*)
3. Добавление новых городов в ходе работы программы

Дополнительное задание (выбрать минимум одно):

1. Решение задачи по оптимальной транспортировке набора товаров и/или пассажиров из города в город. То есть минимизировать стоимость транспортировки
2. Возможность выполнения цепочки команд транспортировки. Если цепочку выполнить нельзя, то команды не должны исполняться и программы должна остаться в том же состоянии, то есть обеспечить транзакционность.

3. Добавить баланс пользователя, возможность покупать и продавать товары, добавить сущность банк. Пользователь может брать кредит. Также при транспортировке учитывать баланс.

2. Ход работы

Описание структуры используемых классов

2.1. В качестве дополнительного задания был реализован GUI с использованием средств QT и был минимизирован путь транспортировки транспорта с пассажирами использованием жадного алгоритма поиска кратчайшего пути в графе.

2.2. Была реализована иерархия продукции с помещением интерфейса в базовый класс Production и последующей его реализацией в классах наследниках и конкретных продуктах

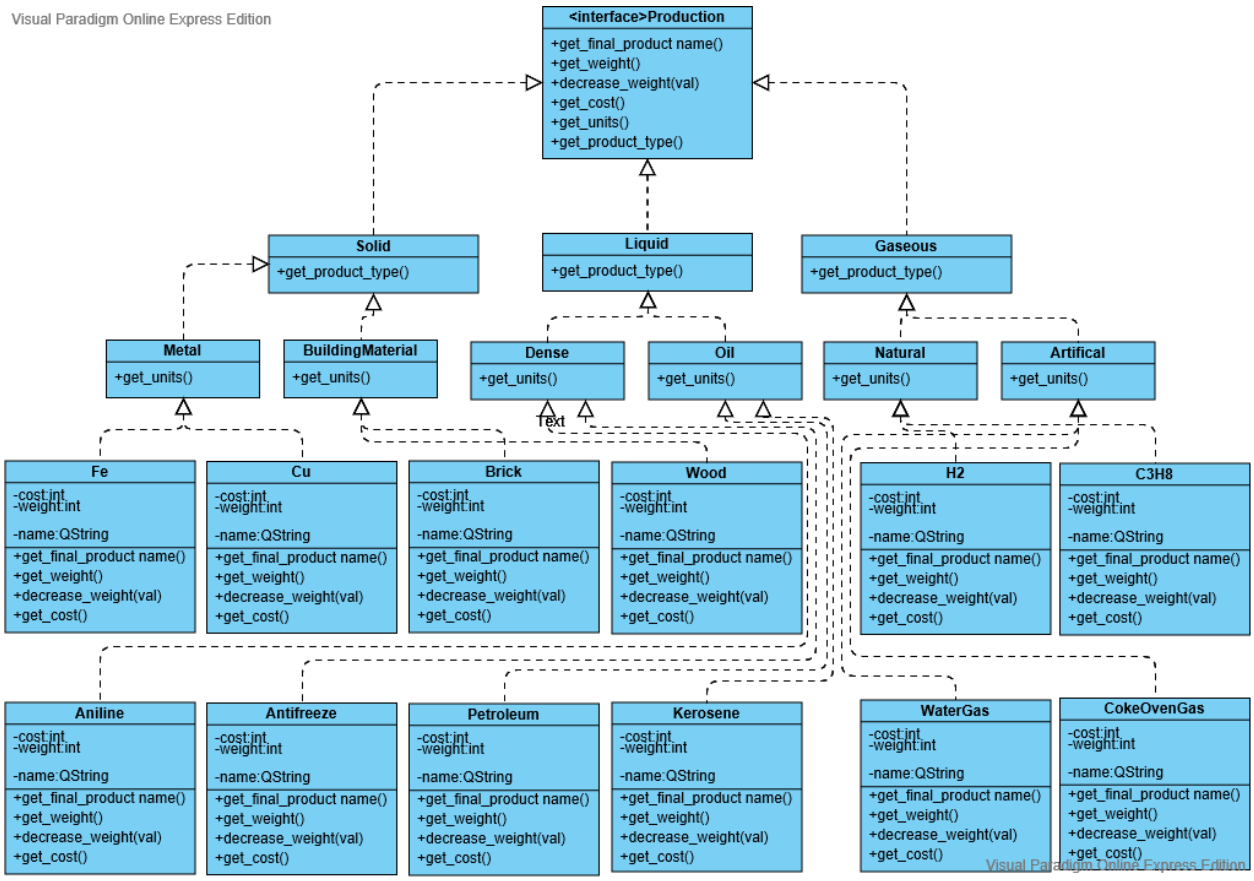


Рис. 1

2.3. Был реализован класс Stock для хранения продукции определенного типа. При создании данного класса был использован порождающий паттерн «Фабричный метод» - Stock является Creater'ом, содержит метод

add_product() и использует базовый класс Production для хранения в векторе - указателей на конкретные продукты, которые можно видеть в пункте выше.

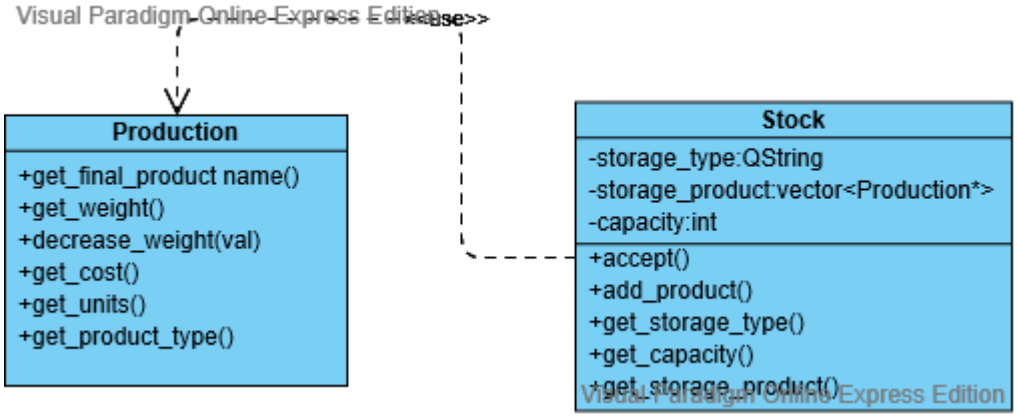


Рис. 2

2.4. Была реализована иерархия транспортных средств для перемещения 1 или нескольких типов продукции и пассажиров, если это возможно. Интерфейс был помещен в базовый класс Transport.

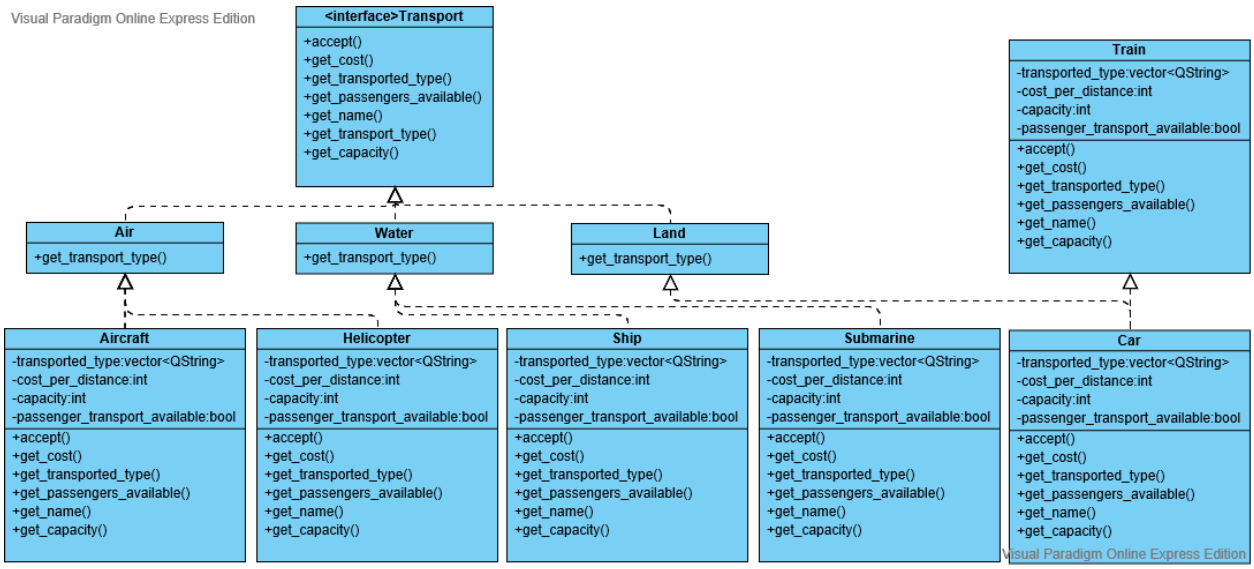


Рис. 3

2.5. Был реализован класс города – City, содержащий в себе вектор складов и транспорта. Здесь так же как и для склада реализован порождающий паттерн «Фабричный метод». Класс City является Creater'ом, использует базовый класс Transport для добавления методом содержит add_transport() нового конкретного транспорта, описание которого можно видеть в пункте выше.

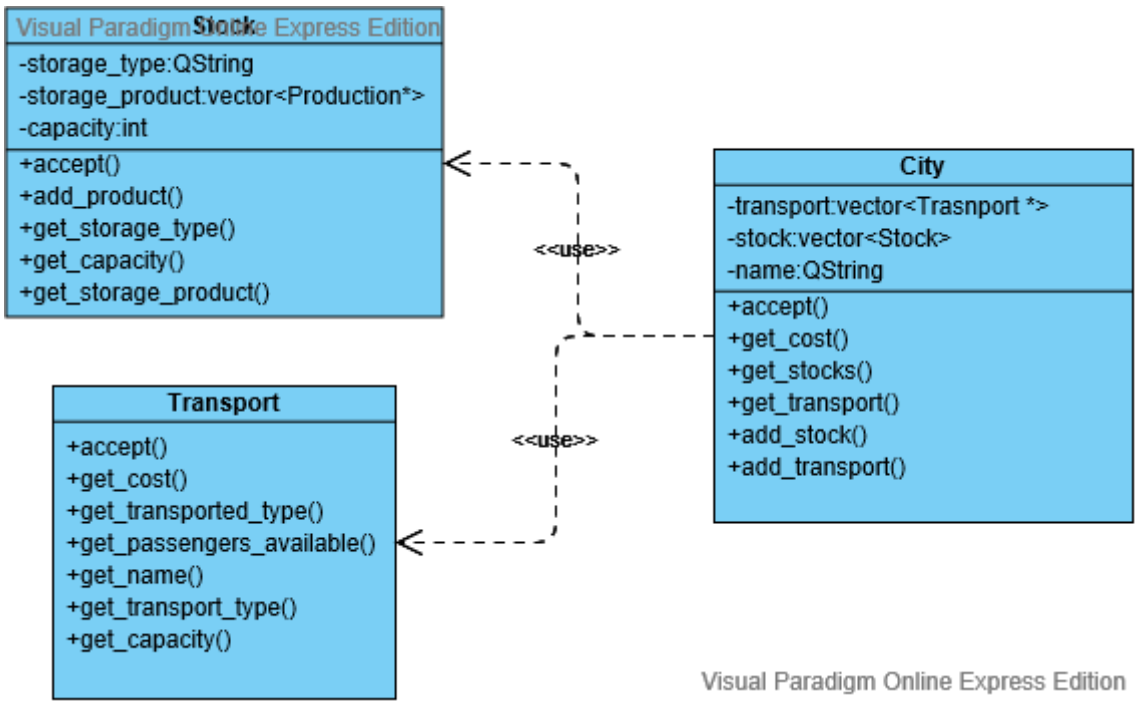


Рис. 4

2.6. Был реализован собственный класс обработки ошибок – MyException, Событие, в котором произошла ошибка и данные, на которых она произошла выводятся в виде всплывающего окошка – QMessageBox

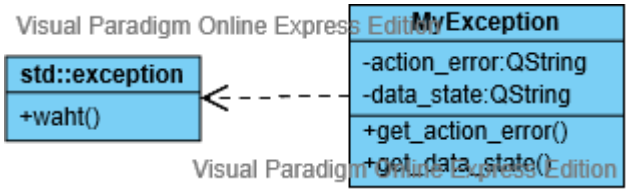


Рис. 5

2.7. Был реализован механизм логирования с использованием структурного паттерна «Proxy» – сперва логи сохраняются в кэш и отображаются. При переполнении кэша идет запись в файл, а после – очищение кэша.

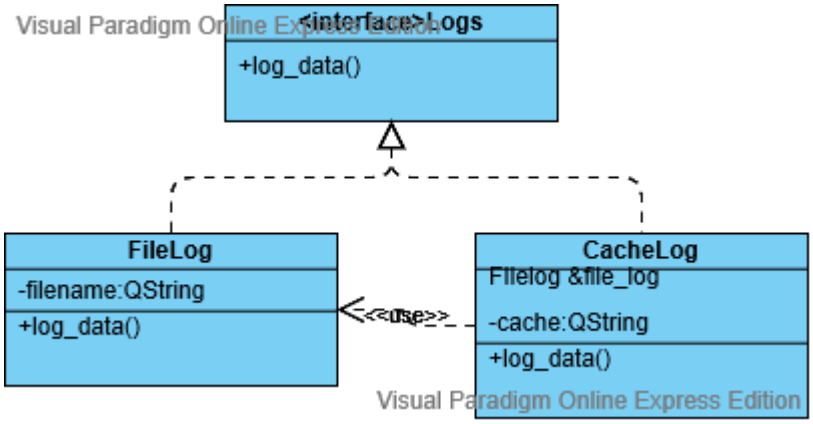


Рис. 6

2.8. Был реализован класс графа городов – graph_cities, содержащий список дорог между городами – их длину и тип.

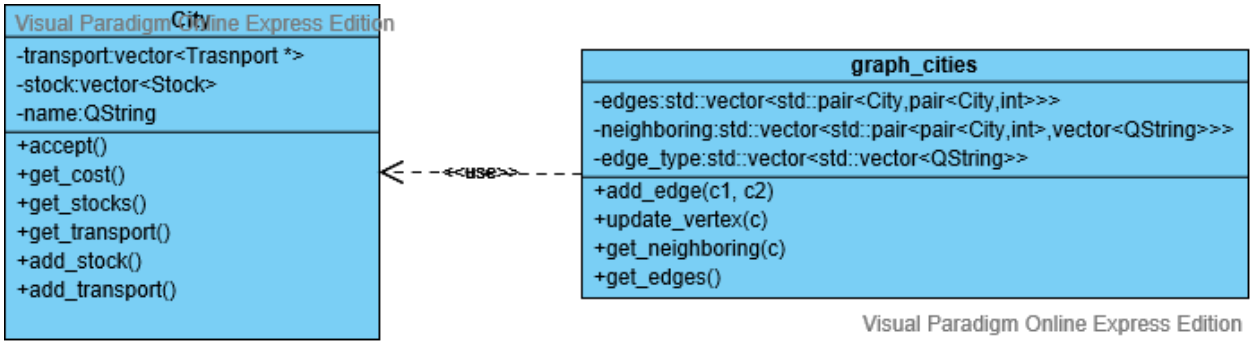


Рис. 7

2.9. Был реализован алгоритм поиска кратчайшего пути транспортировки между городами, если это возможно, с использованием жадного алгоритма(данный алгоритм был вынесен в отдельную функцию см. приложение А).

2.10. Был реализован поведенческий паттерн «Посетитель» для того, чтобы посещать город, в котором произошло удаление или добавление чего-то нового и собирать актуальную информацию о том, что есть в городе. Затем эта информация обновляется в окошке Cities Status.

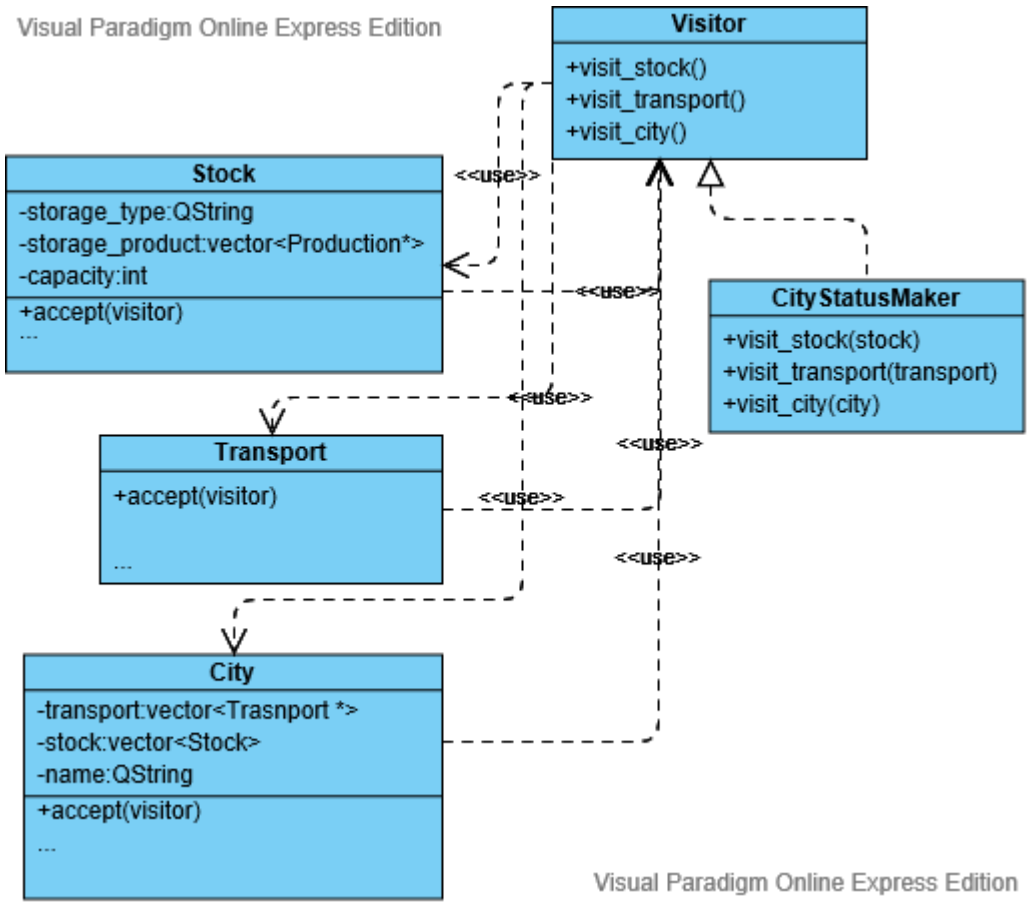
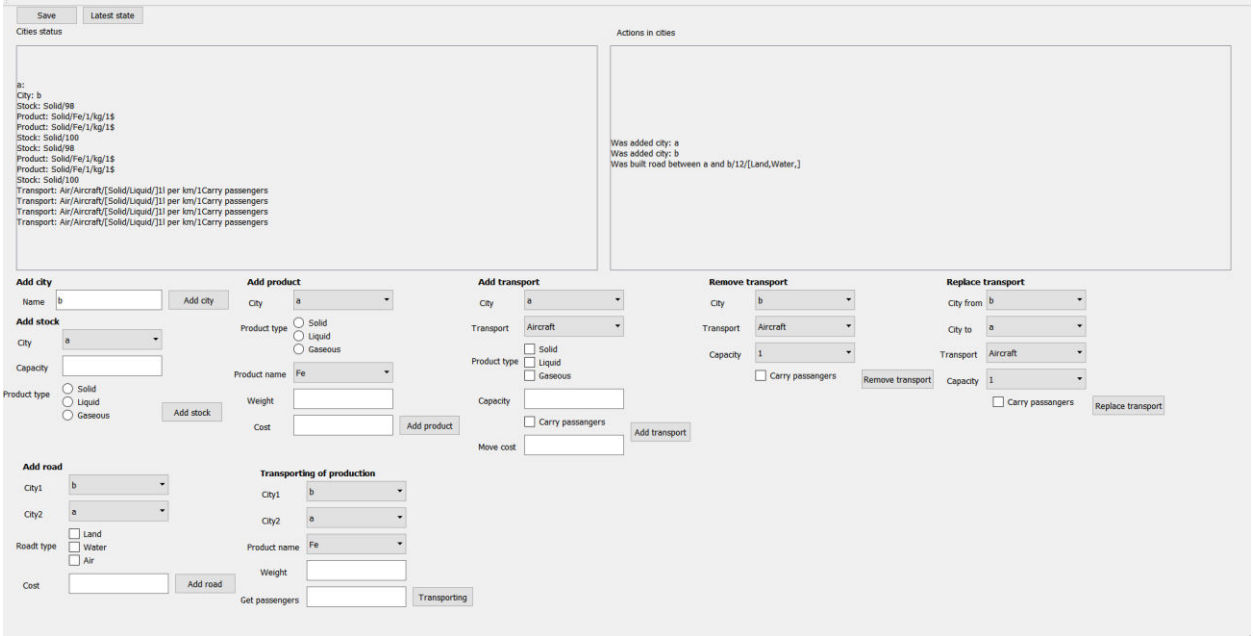


Рис. 8

Описание интерфейса

2.11. Был реализован графический интерфейс с использованием инструментов Qt(исходный код см. в приложении А). Данный интерфейс позволяет добавлять новые города. В город добавлять склады с определенным типом хранимой продукции и транспорты, способные перевозить 1 или несколько типов продукции, а также пассажиров. Выбрать транспорт можно в выпадающем списке. Транспорт из города можно удалять и перемещать из одного города в другой. На склады в городе можно добавлять продукцию – выбрать ее можно из выпадающего списка. Есть возможность строить дороги между городами с выбором тип дорог, после чего по этим дорогам можно перевозить продукцию из одного города в другой. В качестве дополнительного задания были добавлены кнопки Save – сохранение текущего состояния и кнопка –

Latest state, используя которую после перезапуска программы, можно восстановить последнее сохраненное состояние.



3. Вывод

В ходе выполнения лабораторной работы были реализованы иерархии классов для транспорта и продуктов, а также был применен порождающий паттерн «Фабричный метод» для того, чтобы хранить объекты этих классов на складе и в городе. Также был реализован механизм логирования с использованием структурного паттерна «Proxy» - логи записываются в файл только в том случае, если переполнен кэш заместителя, во всех случаях логи отображаются на экране – окошко Cities Actions. Был реализован механизм обновления текстовой информации о статусах городов, в которых что-то поменялось с использованием поведенческого паттерна «Visitor», который содержит методы для посещение склада, транспорта и города. В итоге была построена рабочая логистика, где можно добавлять города, транспорт, склады, продукцию, перемещать и удалять транспорт, а также перевозить товары и пассажиров из одного в другой город по кратчайшему пути, если это возможно.

Приложение А.

Исходный код

Файл city.h

```
#include "stock.h"
#include "transport.h"

using namespace std;

class City
{
public:
   City(QString name);

   QString accept(Visitor * v);
   QString get_name() const;
   vector<Stock>& get_stocks();
   vector<Transport *> & get_transports();
   Stock& add_stock(QString storage_type, int capacity);
   Transport * add_transport(QString name, vector<QString> &transported_type, int cost, int capacity,
                             bool passenger_available);

   bool operator < (City const & other) const
   {
      return name < other.get_name() ? true : false;
   }

   bool operator == (City const & other) const
   {
      return name == other.name;
   }
private:
   QString name;
   vector<Stock> stock;
   vector<Transport *> transport;
};

#endif // CITY_H
```

Файл graph_city.h

```
#ifndef GRAPH_CITIES_H
#define GRAPH_CITIES_H

#include "city.h"
#include <vector>
#include <utility>

class graph_cities
{
public:
   void add_edge(City &v1, City & v2, int cost, vector<QString> road_type);
   //void add_vertex(City &c);
   void update_vertex(City &c);
   std::vector<std::pair<pair<City, int>, vector<QString>>> & get_neighboring(City &c);
   std::vector<std::pair<City, pair<City, int>>> & get_edges();
private:
   std::vector<std::pair<City, pair<City, int>>> edges;
   std::vector<std::pair<pair<City, int>, vector<QString>>> neighboring;
   std::vector<std::vector<QString>> edge_type;
};

/*
 * greedy algoright for searching transporting path
 * return -> int : summary cost of passed way
*/
int greedy_algorith(City &from, City &to, graph_cities & graph, Production *p, int weight,int passengers);

#endif // GRAPH_CITIES_H
```

Файл logs.h

```
#define LOGS_H

#include <QString>
#include <fstream>

extern int SIZE;
```

```
class Logs
{
public:
    virtual ~Logs();
    //interface
    virtual void log_data(QString data) = 0;
};

class FileLog : public Logs
{
public:
    FileLog(QString filename);
    void log_data(QString data) override;
private:
    QString filename;
};

class CacheLog : public Logs
{
public:
    CacheLog(FileLog &file_log);
    void log_data(QString data) override;
private:
    FileLog &file_log;
    QString cache;
};

#endif // LOGS_H
```

Файл myexception.h

```
#ifndef MYEXCEPTION_H
#define MYEXCEPTION_H

#include <exception>
#include <QString>
using namespace std;

class MyException : public exception
{
public:
    MyException(QString action_error, QString data_state);
    QString get_action_error() const;
    QString get_data_state() const ;
private:
    QString action_error;
    QString data_state;
};

#endif // MYEXCEPTION_H
```

Файл production.h

```
#ifndef PRODUCTION_H
#define PRODUCTION_H

#include <QString>

extern std::vector<QString> p_types;

/*
 *
 * Product ---> Solid ---> Metall ---> Fe
 *
*/

//base abstract class of production
class Production
{
public:
    virtual ~Production();

    //interface
    virtual QString get_final_product_name() const = 0;
    virtual int get_weight() const = 0;
    virtual void decrease_weight(int val) = 0;
    virtual int get_cost() const = 0;
    virtual  QString get_units() const = 0;
    virtual QString get_product_type() const = 0;
```

```
};

class Solid : public Production
{
public:
    QString get_product_type() const override;
};

class Metall : public Solid
{
public:
    QString get_units() const override;
};

class Fe : public Metall
{
public:
    Fe(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class Cu : public Metall
{
public:
    Cu(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class BuildingMaterial : public Solid
{
public:
    QString get_units() const override;
};

class Brick : public BuildingMaterial
{
public:
    Brick(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class Wood : public BuildingMaterial
{
public:
    Wood(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
```

```
};


class Liquid : public Production
{
public:
    QString get_product_type() const override;
};

class Dense : public Liquid
{
public:
    QString get_units() const override;
};

class Aniline : public Dense
{
public:
    Aniline(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class Antifreeze : public Dense
{
public:
    Antifreeze(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class Oil : public  Liquid
{
public:
    QString get_units() const override;
};

class Petroleum : public Oil
{
public:
    Petroleum(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class Kerosene : public Oil
{
public:
    Kerosene(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
```

```
    QString name;
};


class Gaseous : public Production
{
public:
    QString get_product_type() const override;
};

class Natural : public Gaseous
{
public:
    QString get_units() const override;
};

class H2 : public Natural
{
public:
    H2(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class C3H8 : public Natural
{
public:
    C3H8(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class Artifical : public Gaseous
{
public:
    QString get_units() const override;
};

class WaterGas : public Artifical
{
public:
    WaterGas(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
    int weight;
    QString name;
};

class CokeOvenGas : public Artifical
{
public:
    CokeOvenGas(int cost, int weight, QString name);

    QString get_final_product_name() const override;
    int get_weight() const override;
    int get_cost() const override;
    void decrease_weight(int val) override;
private:
    int cost;
```

```
    int weight;
    QString name;
};

#endif // PRODUCTION_H
```

Файл stockh

```
#ifndef STOCK_H
#define STOCK_H

#include "production.h"
#include <vector>

class Visitor;

class Stock
{
public:
    Stock(QString storage_type, int capacity);

    QString accept(Visitor *v);
    bool add_product(QString p_type, QString name, int weight, int cost);
    QString get_storage_type() const;
    int get_capacity() const;
    std::vector<Production *> &get_storage_product();
    //bool ship_product(QString p_name, int weight);
private:
    QString storage_type;
    std::vector<Production *> storage_product;
    int capacity;
};

#endif // STOCK_H
```

Файл transport.h

```
#ifndef TRANSPORT_H
#define TRANSPORT_H

#include "production.h"
#include <vector>
#include <QString>

class Visitor;

class Transport
{
public:
    virtual ~Transport();

    //bool load_product(QString product_name, int weight, int cost);
    //std::vector<Production *> & get_products();

    //interface
    virtual QString accept(Visitor *v) = 0;
    virtual int get_cost() const = 0;
    virtual std::vector<QString> & get_transported_type() = 0;
    virtual bool get_passenger_type_available() const = 0;
    virtual QString get_name() const = 0;
    virtual QString get_transport_type() const = 0;
    virtual int get_capacity() const = 0;
private:
    //std::vector<Production *> products;
};

class Air : public Transport
{
public:
    QString get_transport_type() const override;
};

class Aircraft : public Air
{
public:
    Aircraft(std::vector<QString> transported_type, int cost_per_distance, int capacity,
             bool passenger_transport_available);
    int get_cost() const override;
    std::vector<QString> & get_transported_type() override;
```

```
    bool get_passenger_type_available() const override;
    QString get_name() const override;
    int get_capacity() const override;
    QString accept(Visitor *v) override;
private:
   std::vector<QString> transported_type;
   int cost_per_distance;
   int capacity;
   bool passenger_transport_available;
};

class Helicopter : public Air
{
public:
    Helicopter(std::vector<QString> transported_type, int cost_per_distance, int capacity,
            bool passenger_transport_available);
    int get_cost() const override;
    std::vector<QString> & get_transported_type() override;
    bool get_passenger_type_available() const override;
    QString get_name() const override;
    int get_capacity() const override;
    QString accept(Visitor *v) override;
private:
   std::vector<QString> transported_type;
   int cost_per_distance;
   int capacity;
   bool passenger_transport_available;
};

class Water : public Transport
{
public:
    QString get_transport_type() const override;
};

class Ship : public Water
{
public:
    Ship(std::vector<QString> transported_type, int cost_per_distance, int capacity,
            bool passenger_transport_available);
    int get_cost() const override;
    std::vector<QString> & get_transported_type() override;
    bool get_passenger_type_available() const override;
    QString get_name() const override;
    int get_capacity() const override;
    QString accept(Visitor *v) override;
private:
   std::vector<QString> transported_type;
   int cost_per_distance;
   int capacity;
   bool passenger_transport_available;
};

class Submarine : public Water
{
public:
    Submarine(std::vector<QString> transported_type, int cost_per_distance, int capacity,
            bool passenger_transport_available);
    int get_cost() const override;
    std::vector<QString> & get_transported_type() override;
    bool get_passenger_type_available() const override;
    QString get_name() const override;
    int get_capacity() const override;
    QString accept(Visitor *v) override;
private:
   std::vector<QString> transported_type;
   int cost_per_distance;
   int capacity;
   bool passenger_transport_available;
};


class Land : public Transport
{
public:
    QString get_transport_type() const override;
};
```

```
class Car : public Land
{
public:
    Car(std::vector<QString> transported_type, int cost_per_distance, int capacity,
            bool passenger_transport_available);
    int get_cost() const override;
    std::vector<QString> & get_transported_type() override;
    bool get_passenger_type_available() const override;
    QString get_name() const override;
    int get_capacity() const override;
    QString accept(Visitor *v) override;
private:
   std::vector<QString> transported_type;
   int cost_per_distance;
   int capacity;
   bool passenger_transport_available;
};

class Train : public Land
{
public:
    Train(std::vector<QString> transported_type, int cost_per_distance, int capacity,
            bool passenger_transport_available);
    int get_cost() const override;
    std::vector<QString> & get_transported_type() override;
    bool get_passenger_type_available() const override;
    QString get_name() const override;
    int get_capacity() const override;
    QString accept(Visitor *v) override;
private:
   std::vector<QString> transported_type;
   int cost_per_distance;
   int capacity;
   bool passenger_transport_available;
};

#endif // TRANSPORT_H
```

Файл visitor.h

```
#ifndef VISITOR_H
#define VISITOR_H

#include <QString>

class City;
class Stock;
class Transport;

class Visitor
{
public:
    virtual ~Visitor();

    //interface
    virtual QString visit_stock(Stock & s) = 0;
    virtual QString visit_transport(Transport * t) = 0;
    virtual QString visit_city(City &c) = 0;
};

class CityStatusMaker : public Visitor
{
public:
    QString visit_stock(Stock & s) override;
    QString visit_transport(Transport * t) override;
    QString visit_city(City &c) override;
};

#endif // VISITOR_H
```

Файл city.cpp

```
#include "city.h"
#include "myexception.h"
#include "visitor.h"

City::City(QString name) : name(name)
{}

QString City::get_name() const
```

```
{
    return name;
}

Stock& City::add_stock(QString storage_type, int capacity)
{
    stock.push_back(Stock(storage_type, capacity));

    if(stock.back().get_storage_type().toLower() == "unknown")
        throw MyException("Add stock error", "Incorrect storage type: " + storage_type);

    return stock.back();
}

vector<Stock>& City::get_stocks()
{
    return stock;
}

vector<Transport *> & City::get_transports()
{
    return transport;
}

Transport * City::add_transport(QString name, vector<QString> &transported_type, int cost, int capacity,
                                bool passenger_available)
{
    /* add Aircraft */
    if(name.toLower() == QString("aircraft"))
        transport.push_back(new Aircraft(transported_type, cost, capacity, passenger_available));
    /* add Helicopter */
    else if(name.toLower() == QString("helicopter"))
        transport.push_back(new Helicopter(transported_type, cost, capacity, passenger_available));
    /* add Ship */
    else if(name.toLower() == QString("ship"))
        transport.push_back(new Ship(transported_type, cost, capacity, passenger_available));
    /* add Submarine */
    else if(name.toLower() == QString("submarine"))
        transport.push_back(new Submarine(transported_type, cost, capacity, passenger_available));
    /* add Car */
    else if(name.toLower() == QString("car"))
        transport.push_back(new Car(transported_type, cost, capacity, passenger_available));
    /* add Train */
    else if(name.toLower() == QString("train"))
        transport.push_back(new Train(transported_type, cost, capacity, passenger_available));
    else
        throw MyException("Add transport error", "Incorrect transport name: " + name);

    return transport.back();
}

QString City::accept(Visitor * v) { v->visit_city(*this); }
```

Файл graph_cities.cpp

```
#include "graph_cities.h"
#include "myexception.h"
#include <QDebug>
#include <algorithm>


void graph_cities::add_edge(City &v1, City & v2, int cost, vector<QString> road_type)
{
    for(auto & e : edges)
        if(e.first == v1 && e.second.first == v2)
            throw MyException("Add road error", "Road from " + v1.get_name() + " to " + v2.get_name() + " already exists");

    qDebug() << "Add edge";
    edges.push_back(pair<City, pair<City, int>>(v1, pair<City, int>(v2, cost)));
    edges.push_back(pair<City, pair<City, int>>(v2, pair<City, int>(v1, cost)));
    edge_type.push_back(road_type);
    edge_type.push_back(road_type);
}

/*
void graph_cities::add_vertex(City &c)
{
    qDebug() << "Add vertex";
    edges.push_back(pair<City, pair<City, int>>(c, pair<City, int>(c, 0)));
```

```
    edge_type.push_back({"Land", "Water", "Air"});
}
*/

void graph_cities::update_vertex(City &c)
{
    qDebug() << "Update vertex";

    for(auto & edge : edges)
        if(edge.first.get_name().toLower() == c.get_name().toLower())
            edge.first = c;
}

std::vector<std::pair<City, pair<City, int>>> & graph_cities::get_edges() { return edges;}

std::vector<std::pair<pair<City, int>, vector<QString>>> & graph_cities::get_neighboring(City &c)
{
    neighboring.clear();
    int idx = 0;

    for(auto & edge : edges)
    {
        if(edge.first == c)
            neighboring.push_back(std::pair<pair<City, int>, vector<QString>>(edge.second, edge_type[idx]));

        idx += 1;
    }

    sort(neighboring.begin(), neighboring.end(), [](const std::pair<pair<City, int>, vector<QString>> & a, const std::pair<pair<City, int>,
vector<QString>> & b) -> bool
    {
        return a.first.second < b.first.second;
    });

    return neighboring;
}


bool can_pass(Transport *t, std::vector<QString> & road_type)
{
    for(auto & type : road_type)
        if(type.toLower() == t->get_transport_type().toLower())
            return true;

    return false;
}


int greedy_algorith(City &from, City &to, graph_cities & graph, Production *p, int weight,int passengers)
{
    bool has_from = false, has_to = false;

    for(auto & e : graph.get_edges())
        if(e.first.get_name().toLower() == from.get_name())
            has_from = true;
        else if(e.first.get_name().toLower() == to.get_name())
            has_to = true;

    if(!has_to || !has_from)
        return 0;

    map<QString, bool> visitedVertex;
    std::vector<std::pair<City, pair<City, int>>>::iterator it_for_graph;
    vector<City> cur_path;
    vector<int> cost;
    bool no_path = false;

    for(it_for_graph = graph.get_edges().begin(); it_for_graph!= graph.get_edges().end(); it_for_graph++)
        visitedVertex[it_for_graph->first.get_name()] = false;

    cur_path.push_back(from);

    for(int i=0; i < graph.get_edges().size(); ++i)
    {
            if(cur_path.size() == 0)
            {
                no_path = true;
                break;
```

```
        }

        if(cur_path[cur_path.size()-1] == to)
            break;

        qDebug() << "Cur: " << cur_path[cur_path.size() - 1].get_name();
        visitedVertex[cur_path[cur_path.size() - 1].get_name()] = true; //mark last added vertex as visited
        vector<std::pair<pair<City, int>, vector<QString>>> incident_vertexs;
        std::vector<std::pair<pair<City, int>, vector<QString>>> neighboring = graph.get_neighboring(cur_path[cur_path.size() - 1]);

        //simplify cycle with iterator
        for(auto &incident_vertex: neighboring)
        {
            if(!visitedVertex[incident_vertex.first.first.get_name()])
            {
                qDebug() << "Incident: " +  incident_vertex.first.first.get_name();
                incident_vertexs.push_back(incident_vertex);
            }
        }

        //back at 1 step
        if(incident_vertexs.size()==0)
        {
            cur_path.pop_back();
            cost.pop_back();
            continue;
        }

        City min("");

        for(auto & edge : incident_vertexs)
        {
            for(auto & t : cur_path[cur_path.size() - 1].get_transports())
            {
                for(auto & type : t->get_transported_type())
                    if(type.toLower() == p->get_product_type().toLower())
                        if(t->get_capacity() >= weight && can_pass(t, edge.second))
                        {
                            min = edge.first.first;
                            cost.push_back(t->get_cost() * edge.first.second);
                            break;
                        }

                if(min.get_name()!= "")
                    break;
            }

            if(min.get_name()!= "")
                break;
        }

        if(!min.get_name().length())
            throw MyException("Transporting production error", "City " + cur_path.back().get_name() +
                              " hasn't transport for going by any roads");
        cur_path.push_back(min);
    }

    if(no_path)
        throw MyException("Transporting production error", "Path from " + from.get_name() + " to " + to.get_name() +
                          " doesn't exist");

    int summary_cost = 0;

    for(auto c : cost)
        summary_cost += c;

    return summary_cost;
}
```

Файл logs.cpp

```
#include "logs.h"

int SIZE = 1000;

Logs::~Logs() {}

FileLog::FileLog(QString filename) : filename(filename){}
```

```
void FileLog::log_data(QString data)
{
    std::ofstream f(filename.toStdString(), std::ios::binary|std::ios::app);
    f << data.toStdString();
    f.close();
}

CacheLog::CacheLog(FileLog &file_log) : file_log(file_log) {}

void CacheLog::log_data(QString data)
{
    if(cache.length() + data.length() < SIZE)
        file_log.log_data(cache + data);
    else
        cache += data;
}
```

Файл myexception.cpp

```
#include "myexception.h"

MyException::MyException(QString action_error, QString data_state) : action_error(action_error),
    data_state(data_state) {}

QString MyException::get_action_error() const { return action_error; }

QString MyException::get_data_state() const { return data_state; }
```

Файл production.cpp

```
#include "production.h"

std::vector<QString> p_types = {"Solid", "Liquid", "Gaseous"};

Production::~Production(){}

QString Solid::get_product_type() const
{
    return "Solid";
}

QString Metall::get_units() const
{
    return "kg";
}

Fe::Fe(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Fe::get_final_product_name() const
{
    return name;
}

int Fe::get_weight() const
{
    return weight;
}

int Fe::get_cost() const
{
    return cost;
}

void Fe::decrease_weight(int val)
{
    weight -= val;
}

Cu::Cu(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Cu::get_final_product_name() const
{
    return name;
}

int Cu::get_weight() const
{
    return weight;
}

int Cu::get_cost() const
```

```
{
    return cost;
}

void Cu::decrease_weight(int val)
{
    weight -= val;
}

QString BuildingMaterial::get_units() const
{
    return "pieces";
}

Brick::Brick(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Brick::get_final_product_name() const
{
    return name;
}

int Brick::get_weight() const
{
    return weight;
}

int Brick::get_cost() const
{
    return cost;
}

void Brick::decrease_weight(int val)
{
    weight -= val;
}

Wood::Wood(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Wood::get_final_product_name() const
{
    return name;
}

int Wood::get_weight() const
{
    return weight;
}

int Wood::get_cost() const
{
    return cost;
}

void Wood::decrease_weight(int val)
{
    weight -= val;
}

QString Liquid::get_product_type() const
{
    return "Luquid";
}

QString Dense::get_units() const
{
    return "liters";
}

Aniline::Aniline(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Aniline::get_final_product_name() const
{
    return name;
}

int Aniline::get_weight() const
{
    return weight;
```

```
}

int Aniline::get_cost() const
{
    return cost;
}

void Aniline::decrease_weight(int val)
{
    weight -= val;
}

Antifreeze::Antifreeze(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Antifreeze::get_final_product_name() const
{
    return name;
}

int Antifreeze::get_weight() const
{
    return weight;
}

int Antifreeze::get_cost() const
{
    return cost;
}

void Antifreeze::decrease_weight(int val)
{
    weight -= val;
}

QString Oil::get_units() const
{
    return "barrel";
}

Petroleum::Petroleum(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Petroleum::get_final_product_name() const
{
    return name;
}

int Petroleum::get_weight() const
{
    return weight;
}

int Petroleum::get_cost() const
{
    return cost;
}

void Petroleum::decrease_weight(int val)
{
    weight -= val;
}

Kerosene::Kerosene(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString Kerosene::get_final_product_name() const
{
    return name;
}

int Kerosene::get_weight() const
{
    return weight;
}

int Kerosene::get_cost() const
{
    return cost;
}
```

```
void Kerosene::decrease_weight(int val)
{
    weight -= val;
}


QString Gaseous::get_product_type() const
{
    return "Gaseous";
}

QString Natural::get_units() const
{
    return "m^3";
}

H2::H2(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString H2::get_final_product_name() const
{
    return name;
}

int H2::get_weight() const
{
    return weight;
}

int H2::get_cost() const
{
    return cost;
}

void H2::decrease_weight(int val)
{
    weight -= val;
}

C3H8::C3H8(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString C3H8::get_final_product_name() const
{
    return name;
}

int C3H8::get_weight() const
{
    return weight;
}

int C3H8::get_cost() const
{
    return cost;
}

void C3H8::decrease_weight(int val)
{
    weight -= val;
}

QString Artifical::get_units() const
{
    return "m^3";
}

WaterGas::WaterGas(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString WaterGas::get_final_product_name() const
{
    return name;
}

int WaterGas::get_weight() const
{
    return weight;
}

int WaterGas::get_cost() const
```

```
{
    return cost;
}

void WaterGas::decrease_weight(int val)
{
    weight -= val;
}

CokeOvenGas::CokeOvenGas(int cost, int weight, QString name) : cost(cost), weight(weight), name(name) {}

QString CokeOvenGas::get_final_product_name() const
{
    return name;
}

int CokeOvenGas::get_weight() const
{
    return weight;
}

int CokeOvenGas::get_cost() const
{
    return cost;
}

void CokeOvenGas::decrease_weight(int val)
{
    weight -= val;
}
```

Файл stock.cpp

```
#include "stock.h"
#include "visitor.h"

Stock::Stock(QString p_type, int capacity) : capacity(capacity)
{
    storage_type = "Unknown";

    for(QString type : p_types)
        if(p_type.toLower() == type.toLower())
            storage_type = type;
}

bool Stock::add_product(QString p_type, QString name, int weight, int cost)
{
    if(p_type.toLower() == storage_type.toLower() && weight <= capacity)
    {
        /* add Fe */
        if(name.toLower() == QString("fe"))
        {
            capacity -= weight;
            storage_product.push_back(new Fe(cost, weight, name));
            return true;
        }

        /* add Cu */
        if(name.toLower() == QString("cu"))
        {
            capacity -= weight;
            storage_product.push_back(new Cu(cost, weight, name));
            return true;
        }

        /* add Brick */
        if(name.toLower() == QString("brick"))
        {
            capacity -= weight;
            storage_product.push_back(new Brick(cost, weight, name));
            return true;
        }

        /* add Wood */
        if(name.toLower() == QString("wood"))
        {
            capacity -= weight;
```

```
            storage_product.push_back(new Wood(cost, weight, name));
            return true;
        }

        /* add Aniline */
        if(name.toLower() == QString("aniline"))
        {
            capacity -= weight;
            storage_product.push_back(new Aniline(cost, weight, name));
            return true;
        }

        /* add Antifreeze */
        if(name.toLower() == QString("antifreeze"))
        {
            capacity -= weight;
            storage_product.push_back(new Antifreeze(cost, weight, name));
            return true;
        }

        /* add Petroleum */
        if(name.toLower() == QString("petroleum"))
        {
            capacity -= weight;
            storage_product.push_back(new Petroleum(cost, weight, name));
            return true;
        }

        /* add Kerosene */
        if(name.toLower() == QString("kerosene"))
        {
            capacity -= weight;
            storage_product.push_back(new Kerosene(cost, weight, name));
            return true;
        }

        /* add H2 */
        if(name.toLower() == QString("h2"))
        {
            capacity -= weight;
            storage_product.push_back(new H2(cost, weight, name));
            return true;
        }

        /* add C3H8 */
        if(name.toLower() == QString("c3h8"))
        {
            capacity -= weight;
            storage_product.push_back(new C3H8(cost, weight, name));
            return true;
        }

        /* add WaterGas */
        if(name.toLower() == QString("water-gas"))
        {
            capacity -= weight;
            storage_product.push_back(new WaterGas(cost, weight, name));
            return true;
        }

        /* add CokeOvenGas */
        if(name.toLower() == QString("coke-oven-gas"))
        {
            capacity -= weight;
            storage_product.push_back(new CokeOvenGas(cost, weight, name));
            return true;
        }
    }

    return false;
}

QString Stock::get_storage_type() const {return storage_type;}

int Stock::get_capacity() const {return capacity;}

std::vector<Production *> & Stock::get_storage_product() {return storage_product;}
```

```
QString Stock::accept(Visitor *v)
{
   return v->visit_stock(*this);
}
```

Файл transport.cpp

```
#include "transport.h"
#include "visitor.h"

Transport::~Transport() {}

QString Air::get_transport_type() const { return "Air";}

Aircraft::Aircraft(std::vector<QString> transported_type, int cost_per_distance, int capacity,
                   bool passenger_transport_available) : transported_type(transported_type),
                   cost_per_distance(cost_per_distance), capacity(capacity),
                   passenger_transport_available(passenger_transport_available) {}

int Aircraft::get_cost() const { return cost_per_distance;}

std::vector<QString> & Aircraft::get_transported_type() { return transported_type;}

bool Aircraft::get_passenger_type_available() const { return passenger_transport_available;}

QString Aircraft::get_name() const { return "Aircraft";}

int Aircraft::get_capacity() const { return capacity;}

QString Aircraft::accept(Visitor *v) { return v->visit_transport(this); }


Helicopter::Helicopter(std::vector<QString> transported_type, int cost_per_distance, int capacity,
                   bool passenger_transport_available) : transported_type(transported_type),
                   cost_per_distance(cost_per_distance), capacity(capacity),
                   passenger_transport_available(passenger_transport_available) {}

int Helicopter::get_cost() const { return cost_per_distance;}

std::vector<QString> & Helicopter::get_transported_type() { return transported_type;}

bool Helicopter::get_passenger_type_available() const { return passenger_transport_available;}

QString Helicopter::get_name() const { return "Helicopter";}

int Helicopter::get_capacity() const { return capacity;}

QString Helicopter::accept(Visitor *v) { return v->visit_transport(this); }


QString Water::get_transport_type() const { return "Water";}

Ship::Ship(std::vector<QString> transported_type, int cost_per_distance, int capacity,
                   bool passenger_transport_available) : transported_type(transported_type),
                   cost_per_distance(cost_per_distance), capacity(capacity),
                   passenger_transport_available(passenger_transport_available) {}

int Ship::get_cost() const { return cost_per_distance;}

std::vector<QString> & Ship::get_transported_type() { return transported_type;}

bool Ship::get_passenger_type_available() const { return passenger_transport_available;}

QString Ship::get_name() const { return "Ship";}

int Ship::get_capacity() const { return capacity;}

QString Ship::accept(Visitor *v) { return v->visit_transport(this); }

Submarine::Submarine(std::vector<QString> transported_type, int cost_per_distance, int capacity,
                   bool passenger_transport_available) : transported_type(transported_type),
                   cost_per_distance(cost_per_distance), capacity(capacity),
                   passenger_transport_available(passenger_transport_available) {}

int Submarine::get_cost() const { return cost_per_distance;}

std::vector<QString> & Submarine::get_transported_type() { return transported_type;}

bool Submarine::get_passenger_type_available() const { return passenger_transport_available;}
```

```
QString Submarine::get_name() const { return "Submarine";}

int Submarine::get_capacity() const { return capacity;}

QString Submarine::accept(Visitor *v) { return v->visit_transport(this); }

QString Land::get_transport_type() const { return "Land";}

Car::Car(std::vector<QString> transported_type, int cost_per_distance, int capacity,
                bool passenger_transport_available) : transported_type(transported_type),
                cost_per_distance(cost_per_distance), capacity(capacity),
                passenger_transport_available(passenger_transport_available) {}

int Car::get_cost() const { return cost_per_distance;}

std::vector<QString> & Car::get_transported_type() { return transported_type;}

bool Car::get_passenger_type_available() const { return passenger_transport_available;}

QString Car::get_name() const { return "Car";}

int Car::get_capacity() const { return capacity;}

QString Car::accept(Visitor *v) { return v->visit_transport(this); }

Train::Train(std::vector<QString> transported_type, int cost_per_distance, int capacity,
                bool passenger_transport_available) : transported_type(transported_type),
                cost_per_distance(cost_per_distance), capacity(capacity),
                passenger_transport_available(passenger_transport_available) {}

int Train::get_cost() const { return cost_per_distance;}

std::vector<QString> & Train::get_transported_type() { return transported_type;}

bool Train::get_passenger_type_available() const { return passenger_transport_available;}

QString Train::get_name() const { return "Train";}

int Train::get_capacity() const { return capacity;}

QString Train::accept(Visitor *v) { return v->visit_transport(this); }
```

Файл visitor.cpp

```
#include "visitor.h"
#include "city.h"
#include "transport.h"
#include "stock.h"

Visitor::~Visitor() {}

QString CityStatusMaker::visit_stock(Stock & s)
{
   QString info = "Stock: " + s.get_storage_type() + "/" + QString::number(s.get_capacity()) + "\n";

   for(auto & p : s.get_storage_product())
      info += "Product: " + p->get_product_type() + "/" + p->get_final_product_name() +
            QString::number(p->get_weight()) + p->get_units() + "/" + QString::number(p->get_cost()) + "$\n";

   return info;
}

QString CityStatusMaker::visit_transport(Transport * t)
{
   QString transported_type = "[";
   QString carry_passengers = t->get_passenger_type_available() ? "Carry passengers" : "Doesn't carry passengers";

   for(auto & type : t->get_transported_type())
      transported_type += type + "/";

   transported_type += "]";

   QString info = "Transport: " + t->get_transport_type() + "/" + t->get_name() + "/" + transported_type +
         QString::number(t->get_cost()) + "l per km" + "/" + QString::number(t->get_capacity()) +
         carry_passengers + "\n";
```

```
    return info;
}

QString CityStatusMaker::visit_city(City &c)
{
    QString info = "City: " + c.get_name() + "\n";

    for(auto & s : c.get_stocks())
        info += s.accept(this);

    for(auto & t : c.get_transports())
        info += t->accept(this);

    return info;
}
```

Файл main.cpp

```
#include <QApplication>
#include "mainwindow.h"

int main(int argc, char **argv)
{
  QApplication app(argc, argv);
  MainWindow window;
  window.show();

  return app.exec();
}
```