545 Final Exam

1. Project objective

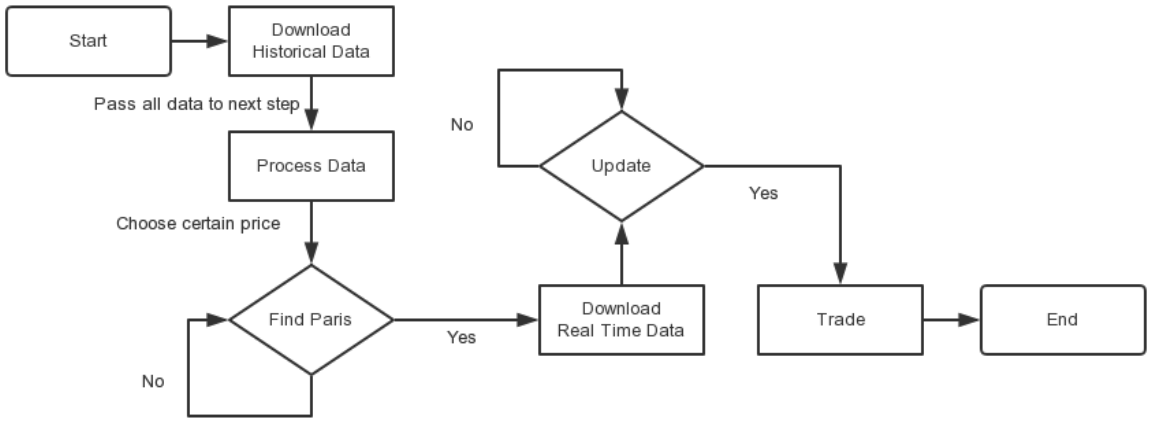
   In this project, We will use pair trading strategy to trade on real time stock. This project have two parts. First of all, we will using historical stock data to find out two stocks that are highly correlated. Then, we use the trading strategy to trade on real time stocks.

2. Trading strategy

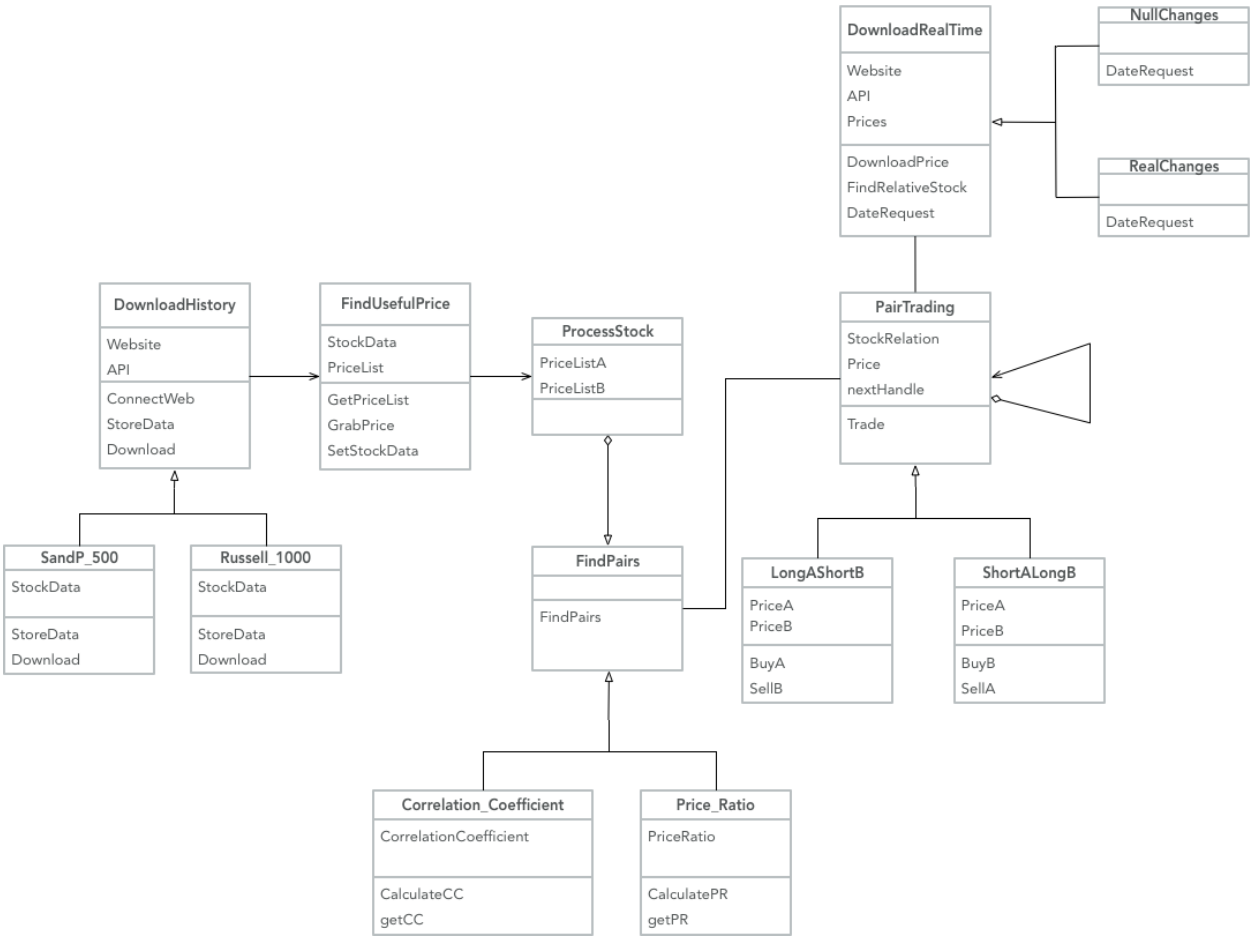
   Pairs trading is a trading strategy that matches a long position with a short position in a pair of highly correlated instruments such as two stocks, exchange-traded funds (ETFs), currencies, commodities or options. In our project, we only focus on the stocks as our underlying. Pairs traders wait for weakness in the correlation, and then go long on the under-performer while simultaneously going short on the over-performer, closing the positions as the relationship returns to its statistical norm.

   For example, stock A and stock B are highly correlated. If the correlation weakens temporarily – stock A moves up and stock B moves down – a pairs trader might exploit this divergence by shorting stock A (the over-performing issue) and going long on stock B (the under-performing issue). If the stocks revert to the statistical mean, the trader can profit.

3. Work process



4. Conceptual design diagram of the project



5. Design patterns

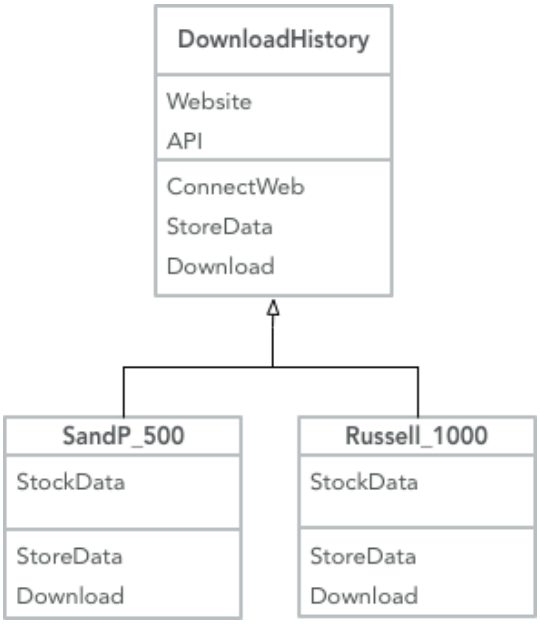
5.1. Template Method

```
class DownloadHistory
{
protected:
    std::string Link;
    //transfer the link to the standard form which could be used for different API
    virtual std::string API(std::string link_);
public:
    //By using link to connect to website
    void ConnectWeb();
    //store data
    virtual void StoreData();
    //download data from web
    virtual void Download();
};

class SandP_500: DownloadHistory
{
private:
    std::vector<std::string> StockData;
```

```
public:
    virtual void StoreData();
    virtual void Download();
    //get the data
    std::vector<std::string> getStockData();
};

class Russell_1000: DownloadHistory
{
private:
    std::vector<std::string> StockData;
public:
    virtual void StoreData();
    virtual void Download();
    //get the data
    std::vector<std::string> getStockData();
};
```

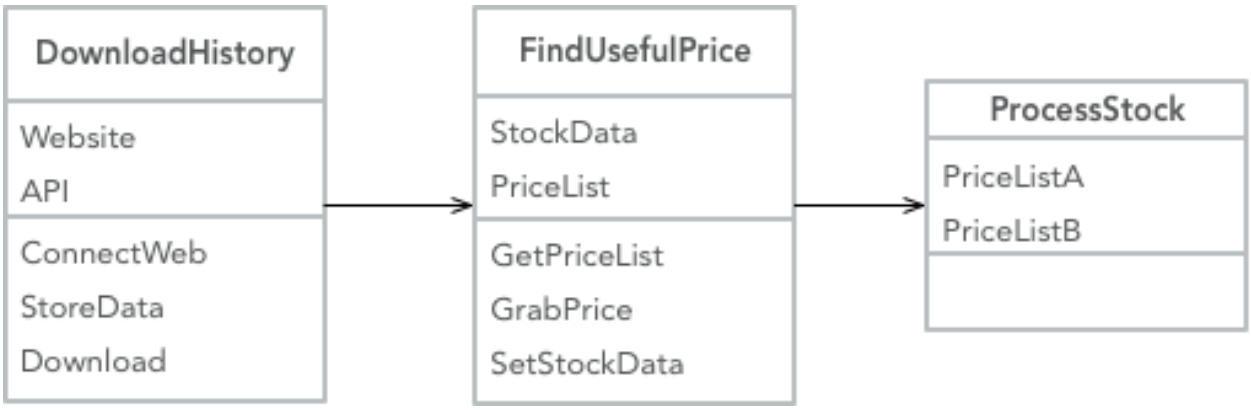


Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. Since we are going to use different datasets from S&P 500 and Russell 1000, the way to access these two dataset must be different. When we using the template method, we want to change some part of algorithm but not all. In this case, we want to change the download parts, and, probably the function to store the data. But we don't need to change the way to access certain website.

5.2. Adaptor

```
class FindUsefulPrice
{
private:
    std::vector<std::string> StockData;
```

```
    std::vector<double> PriceList;
public:
    std::vector<double> getPriceList;
    //find the certain price
    void GrabPrice(std::string name);
    void setStockData(const std::vector<std::string>& StockData_);
};
```



When we have the information downloaded from website, we can't use them directly. As a result, we need to use adaptor design pattern to convert information to numbers that we could use. In our case, we just process the information by choosing certain price of everyday. Then, we turn them from "string" to "double" and store them in vector. That is the whole process of the adaptor. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

5.3. Strategy

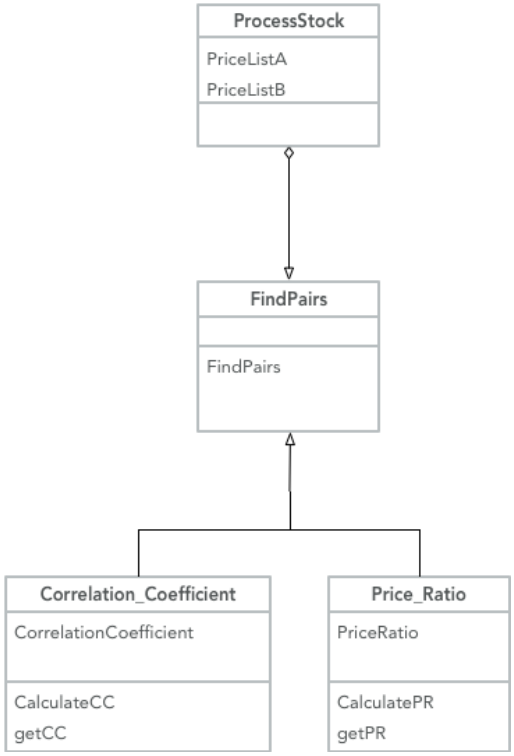
```
class ProcessStock
{
protected:
    std::vector<double> PriceListA;
    std::vector<double> PriceListB;
};

class FindParis: ProcessStock
{
public:
    virtual void Find();
};

class Correlation_Coefficient: FindParis
{
private:
    double CC;
public:
    //find the relationship between two stocks
    virtual void Find();
    //calculate the Correlation coefficient factor of stocks
```

```
    void CalculateCC();
    //return the details
    double getCC();
};

class Price_Ratio: FindParis
{
private:
    double PR;
public:
    //find the relationship between two stocks
    virtual void Find();
    //calculate the Price ratio factor of stocks
    void CalculatePR();
    //return the details
    double getCC();
};
```



In order to process the stock, we use two different ways to estimate the relationship of two stocks. We define a family of algorithms, encapsulate each one, and make them interchangeable. So we could call different functions. Two different price is the context. The FindPairs works as the interface and there are two different strategy work as implements.

5.4. Null Object

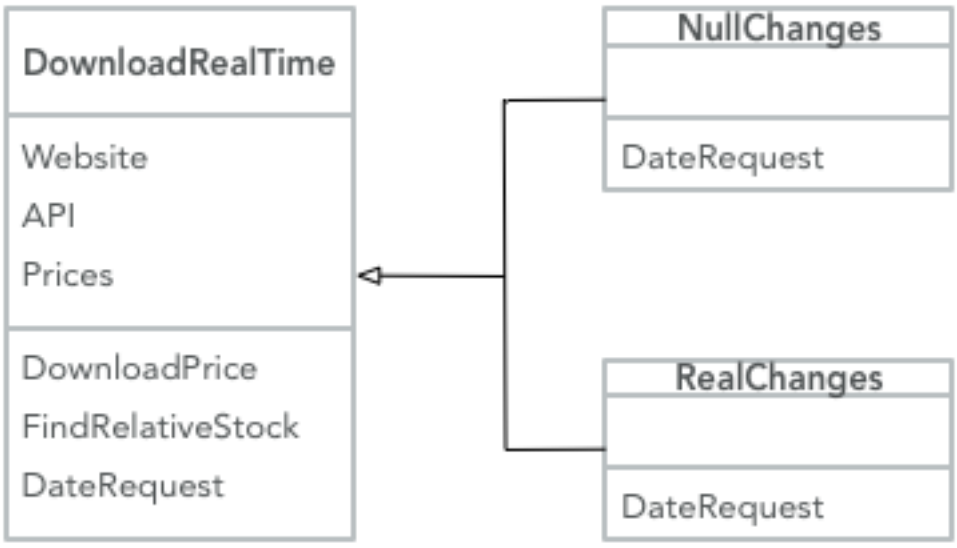
```
class DownloadrealTime
{
```

```
protected:
    std::string Wbesite;
    std::string API(std::string Website);
    double PriceA;
    double PriceB;

public:
    void DownloadPrice();
    bool FindRelativestock();
    //Check if there is any change in the price
    virtual void DataRequest();
};

class NullChanges: DownloadrealTime
{
public:
    //there is nothing happened. Set everything none.
    virtual void DataRequest();
};

class RealChanges: DownloadrealTime
{
public:
    //there are some changes of the price.
    virtual void DataRequest();
};
```



The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default do nothing behavior. In short, a design where "nothing will come of nothing". In our case, when we download the data, we always have to check whether there is anything changed from the last update. The absence of changes will call the default behavior Nullchanges which means do nothing

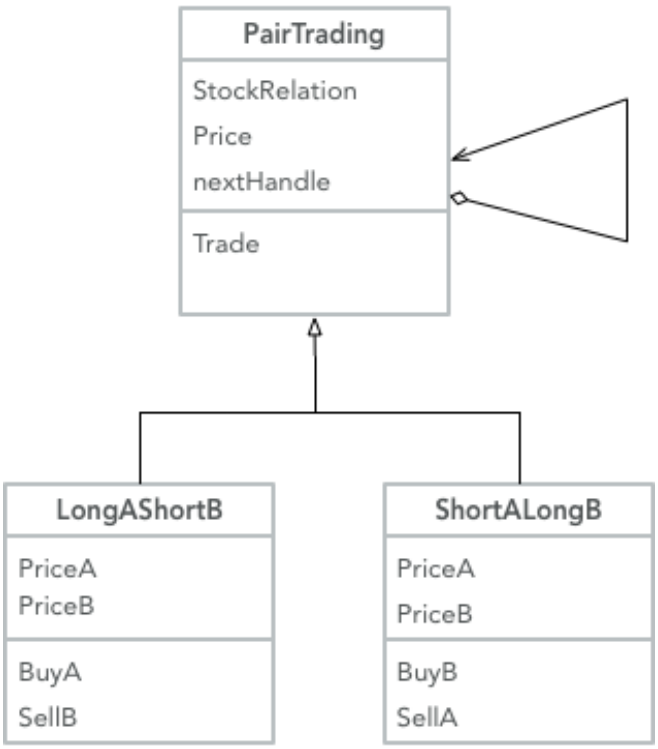
in this special situation. Otherwise, the RealChanges will help to pass the changes to next step.

5.5. Chain of Responsibility

```
class PairTrading
{
protected:
    double StockRelation;
    double PriceA;
    double PriceB;
public:
    PairTrading* super();
    PairTrading* nextHandle;
    virtual void Trade();
};

class LongAShortB: PairTrading
{
public:
    virtual void trade();
    void BuyA();
    void SellB();
};

class ShortALongB: PairTrading
{
public:
    virtual void trade();
    void BuyB();
    void SellA();
};
```



Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it. In our case, the derived classes know how to satisfy trade base on the situation given by the real time data. If the "current" object is not available or sufficient, then it delegates to the base class, which delegates to the "next" object, and the circle of life continues. When the two stock prices are given, the first step is to ask LongAShortB to process. If it doesn't work for this situation, the give it back to PairTrading, and the PairTrading give it to ShortALongB. Otherwise, the LongAShortB could handle the situation which means buy and sell the stocks.

Interview Question

1. Give an pattern example in c++ where you prefer abstract class over interface? and why?

   In general, an abstract class is a class that cannot be instantiated and is usually implemented as a class that has one or more pure virtual functions. However, an abstract classes allow you to provide default functionality for the subclasses. Compared with abstract class, an interface class is a class that specifies the polymorphic interface. The programmer using a class hierarchy can then do so via a base class that communicates only the interface of classes in the hierarchy. In my opinion, abstract class have more close relationship with inheritance.

   The example are given by following code.

```
class Door {
public:
   virtual void open() const {
      std::cout << "open horizontally" << std::endl;
   }
   virtual void close() const {
      std::cout << "close horizontally" << std::endl;
   }
};

class HorizontalDoor : public Door {
};

class VerticalDoor : public Door {
public:
   void open() const {
      std::cout << "open vertically" << std::endl;
   }
   void close() const {
      std::cout << "close vertically" << std::endl;
   }
};

class IAlarm {
public:
   virtual void alert() const = 0;
};

class Alarm : public IAlarm {
public:
   void alert() const {
      std::cout << "ring,ring,ring" << std::endl;
   }
```

```
};

class AlarmDoor : public Door {
protected:
    IAlarm* _alarm;
public:
    AlarmDoor() {
        _alarm = new Alarm;
    }
    ~AlarmDoor() {
        delete _alarm;
    }
public:
    void alert() {
        _alarm->alert();
    }
};
```

In this code, we define an abstract class door and have three derived class such as HorizontalDoor, VerticalDoor and AlarmDoor. They are closely related. All three doors show similar properties. However, the properties of alarm is not a common character of doors. As a result, we can't add the alarm to the base class. Then, we use interface to combine alarm and door. In other words, an abstract class should be used only for objects that are closely related like the classes we discussed above. Finally, when facing objects closely related, we should use an abstract class.

2. What is your favorite design style? Object-oriented design or Structure-oriented design? explain briefly your design process.

   To be honest, I prefer structure-oriented design. The structured-oriented design could be defined as a programming technique that follows a top down design approach with block oriented structures. First of all, when facing a problem, I always break them into different pieces. In other words, during my design process, I could easily divide my program source code into logically structured blocks which normally consist of conditional statements, loops and logic blocks. Then I could finish the task step by step. In most cases, straightforward pieces of code would get the job done easily.

3. Give me two design patterns you will use in your future project? explain the reasons.

   In my opinion, singleton and template method are design patterns I will definitely use in the future. First, ehe singleton pattern is a design pattern that restricts the instantiation of a class to one object. This could be extremely useful when exactly one object is needed to coordinate actions across the system. What's more, this design pattern could help us to make sure that there is only one object, and it could also provide global access to that instance. Second, in the template method, one or more algorithm steps can be overridden by subclasses to allow differing behaviors while ensuring that the overarching algorithm is still followed. When facing a problem with many different solutions, we could easily use this

design pattern. The base class could handle the general algorithm, when the subclass concerns to the steps that may be changed. This could contribute to simplify our code and reuse of code. In other words, it will save my time in the future.

4. What design pattern can standardize similar processes ? and write your own code to implement the pattern in paper, not project.

   In my opinion, the template method is the design pattern that can standardize similar process. Because the template method define the skeleton of an algorithm and allow subclasses to redefine certain steps of an algorithm without changing the algorithm's structure. When we face similar process, there must be some different steps during those process. By using template method, we could easily generated subclasses with some different steps but without changing the whole structure.

```
class Door {
public:
    virtual void open() const {
        std::cout << "open horizontally" << std::endl;
    }
    virtual void close() const {
        std::cout << "close horizontally" << std::endl;
    }
};

class HorizontalDoor : public Door {
};

class VerticalDoor : public Door {
public:
    void open() const {
        std::cout << "open vertically" << std::endl;
    }
    void close() const {
        std::cout << "close vertically" << std::endl;
    }
};

class IAlarm {
public:
    virtual void alert() const = 0;
};

class Alarm : public IAlarm {
public:
    void alert() const {
        std::cout << "ring,ring,ring" << std::endl;
    }
};
```

```
class AlarmDoor : public Door {
protected:
    IAlarm* _alarm;
public:
    AlarmDoor() {
        _alarm = new Alarm;
    }
    ~AlarmDoor() {
        delete _alarm;
    }
public:
    void alert() {
        _alarm->alert();
    }
};
```

In this case, we define a class door which have two operations including open and close. The horizontal door, vertical door and alarm door all share these properties. But each of them has different way to achieve operations. The base class just give the standard of each subclasses. And they could give details based on their own demand. They are similar process.