Kelebogile Masemola – 16189541
Avkar Gopal – 19149159
Mompati Keetile – 24034313
Batsirai Malcolm Dzimati - 20456078

10 June 2024

Assignment 2

Introduction

For the purpose of this assignment, we decided to implement a generic pattern for a behavioural design pattern namely the Observer pattern and our language of choice is C++. This design pattern is used to create a one-to-many dependency between objects, so that when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically, which is useful in scenarios where an object should broadcast changes to other objects without knowing who those objects are. The observer pattern usually has the following use cases:

- Event handling systems: GUI frameworks, where user interactions including clicks and key presses need to be handled by multiple components.
- Model-View-Controller (MVC) architecture: Ensures that views update automatically when the data model changes.
- Real-time data streaming: Applications including stock tickers where multiple displays need to be updated in real time.
- Notification systems: Email, messaging or logging systems where various parts of the system need to be notified of certain events.

The observer pattern offers flexibility to manage dependencies and ensures consistency for applications where state changes may need to be propagated to multiple dependent objects. We made use of a stock trading notification scenario in this assignment.

Design Choice

The Observer design pattern makes use of 2 class templates namely, observer and subject, which enables us to implement the Observer pattern in a type-independent manner. The observer class template defines an interface where concrete observers must implement, it uses a template parameter 'T' to represent the type of data that the observers will receive. The Subject class template manages the list of observers and provides methods to either add, remove and/or notify them. To use the observer and subject class templates a concrete implementation of both classes for a specific type of data will need to be created.

- Observer Template: Defines a generic interface with an update method that concrete observers must implement.
- Subject Template: Manages a list of observers and provides methods to add, remove, and notify them.
- Concrete Implementations: Specific types like int are used to create concrete observers and subjects, demonstrating how the pattern can be applied to real-world scenarios.
- Main Function: Illustrates adding, notifying, and removing observers, showcasing the dynamic nature and flexibility of the Observer pattern.

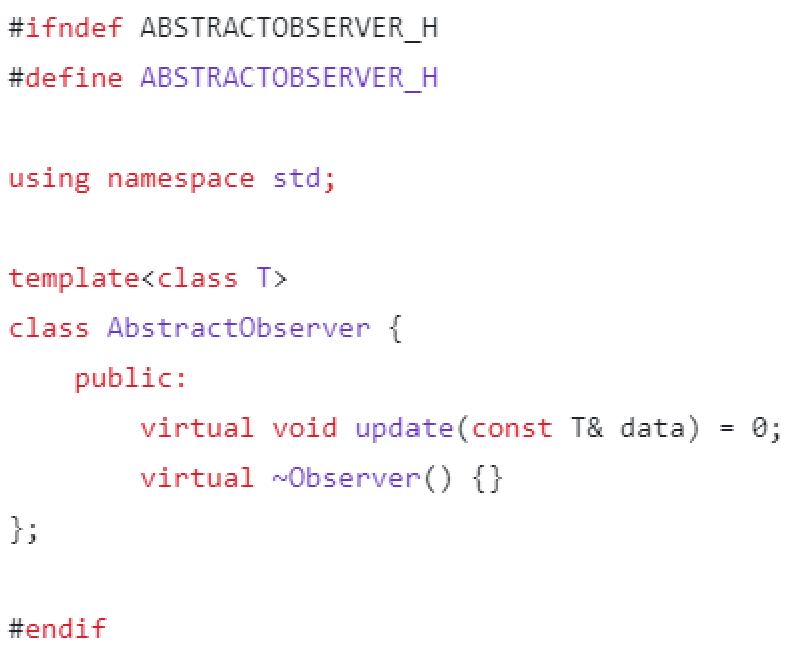
Some of the benefits of using templates to create a generic implementation of the Observer pattern are as follows:

- Independent types – through the use of templates the observer and subject classes can handle any data type which allows for reuse of the code without any duplications or rewrites
- Reuse of code – the same classes can be used in various parts of an application without redundancy
- Maintainability – any and all changes or bug fixes within the template are automatically applied to all instances of the template
- Compile-Time type checking – the use of templates ensures that type mismatches are caught at compile time instead of at runtime which reduces the chances of errors

- Performance – as templates are resolved at compile time there are no runtime overheads associated with dynamic type checking or type casting
- Content interface – there's a uniformed interface regardless of the data type used
- Simplified code – the use of templates decreases the need for repetitive boilerplate code which results in comprehensive implementations

Implementation

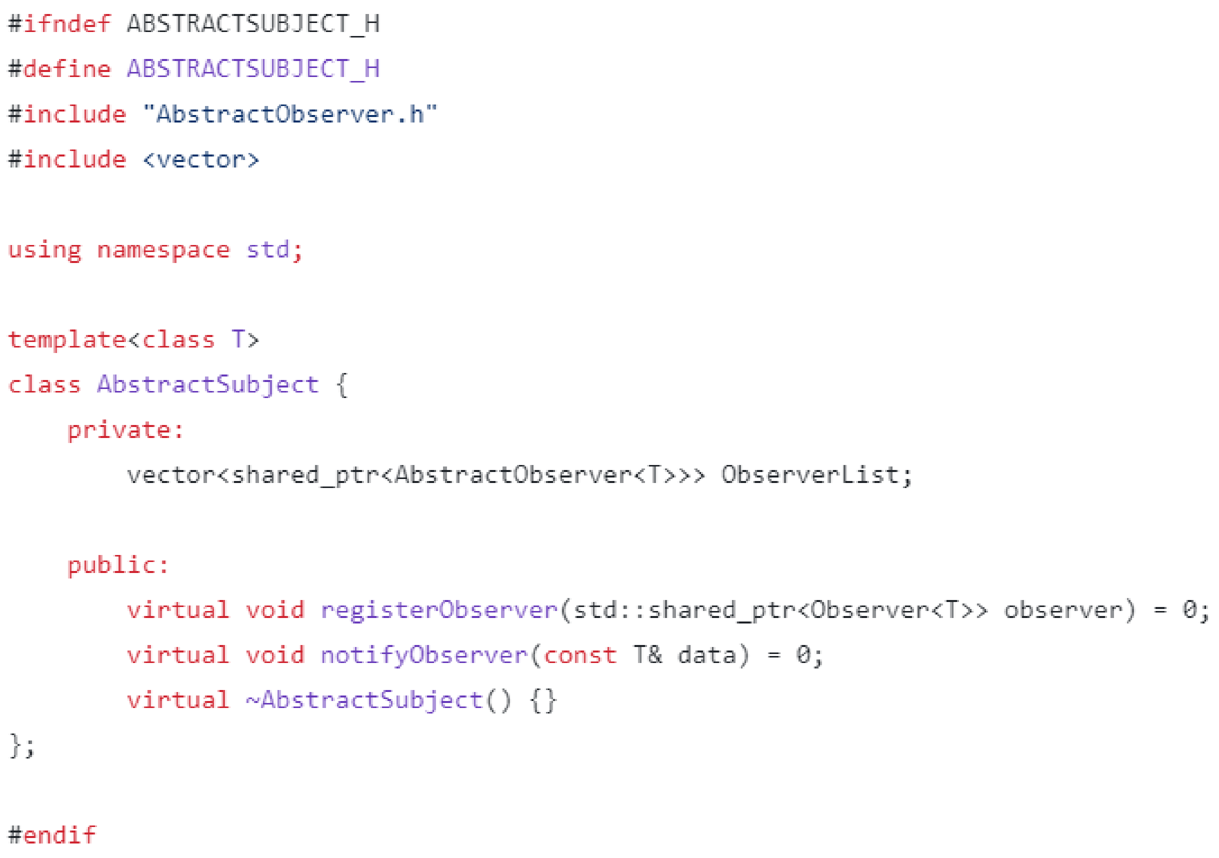
Abstract Observer defines the interface or abstract base class that a concrete observer class should implement. This class declares the methods that will be called by the subject to notify the observer of any changes



```
#ifndef ABSTRACTOBSERVER_H
#define ABSTRACTOBSERVER_H

using namespace std;

template<class T>
class AbstractObserver {
    public:
        virtual void update(const T& data) = 0;
        virtual ~Observer() {}
};

#endif
```

- Template parameter – T represents the type of data that will be passed to the observer once the update has been completed
- Method – virtual update method is implemented by concrete observer classes as it defines how the observer reacts to updates
- Destructor – a virtual destructor makes sure that derived classes are updated accordingly once an observer object is deleted or removed

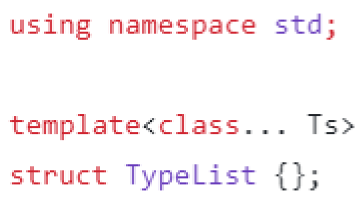
Abstract Subject is the base class that defines a unified interface and behaviour for concrete subjects, this helps various types of subjects to inherit from the abstract subject which helps with consistency and reusability.



```
#ifndef ABSTRACTSUBJECT_H
#define ABSTRACTSUBJECT_H
#include "AbstractObserver.h"
#include <vector>

using namespace std;

template<class T>
class AbstractSubject {
    private:
        vector<shared_ptr<AbstractObserver<T>>> ObserverList;

    public:
        virtual void registerObserver(std::shared_ptr<Observer<T>> observer) = 0;
        virtual void notifyObserver(const T& data) = 0;
        virtual ~AbstractSubject() {}
};

#endif
```

Template parameter – ‘T’ represents the type of data that will be passed to the observers during notification
Smart pointer – std::shared_ptr is used to retain shared ownership of an object through a pointer as it manages observers
Method – notifyObserver method calls all registered observers and passes them the notification, while registerObserver method registers all
Destructor – virtual destructor AbstractSubject is used to ensure that the destuctor of the derived class is called when an object is deleted through a pointer to the base class

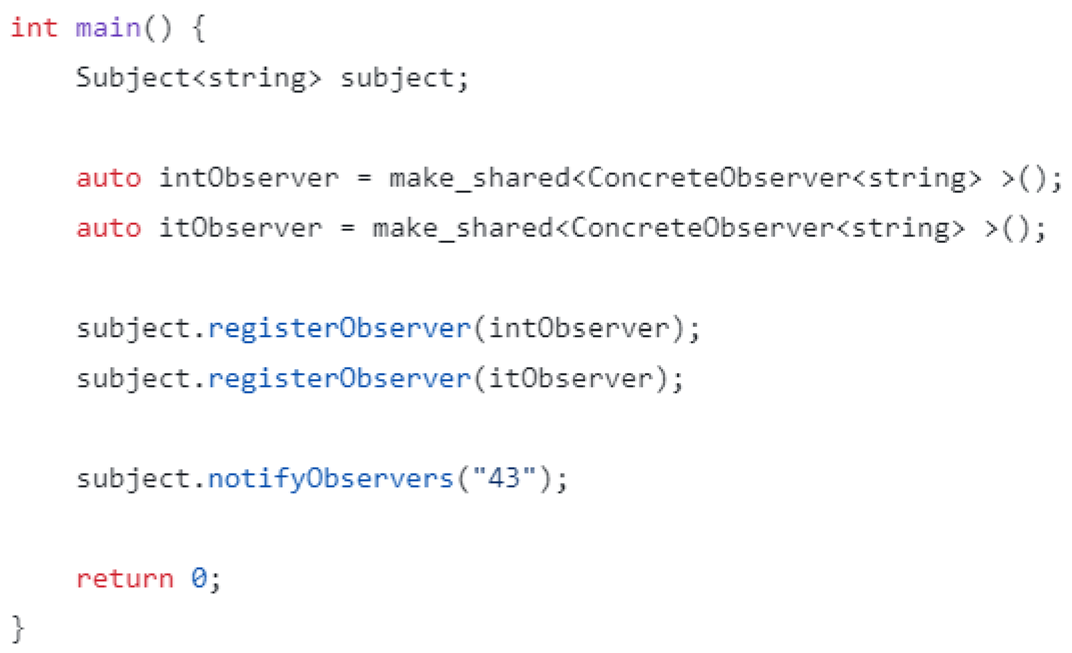
Typelist represents a list of types at compile time and is usually used to create flexible and reusable code that can operate different types



```
using namespace std;

template<class... Ts>
struct TypeList {};
```

The typelist structure is defined, the parameter ‘Ts’ is used to represent the typelist for the template

Main function is used to refer to the point of entry for the program. This method is used to execute the program.



```
int main() {
    Subject<string> subject;

    auto intObserver = make_shared<ConcreteObserver<string> >();
    auto itObserver = make_shared<ConcreteObserver<string> >();

    subject.registerObserver(intObserver);
    subject.registerObserver(itObserver);

    subject.notifyObservers("43");

    return 0;
}
```

Scenario: Stock Market Notification

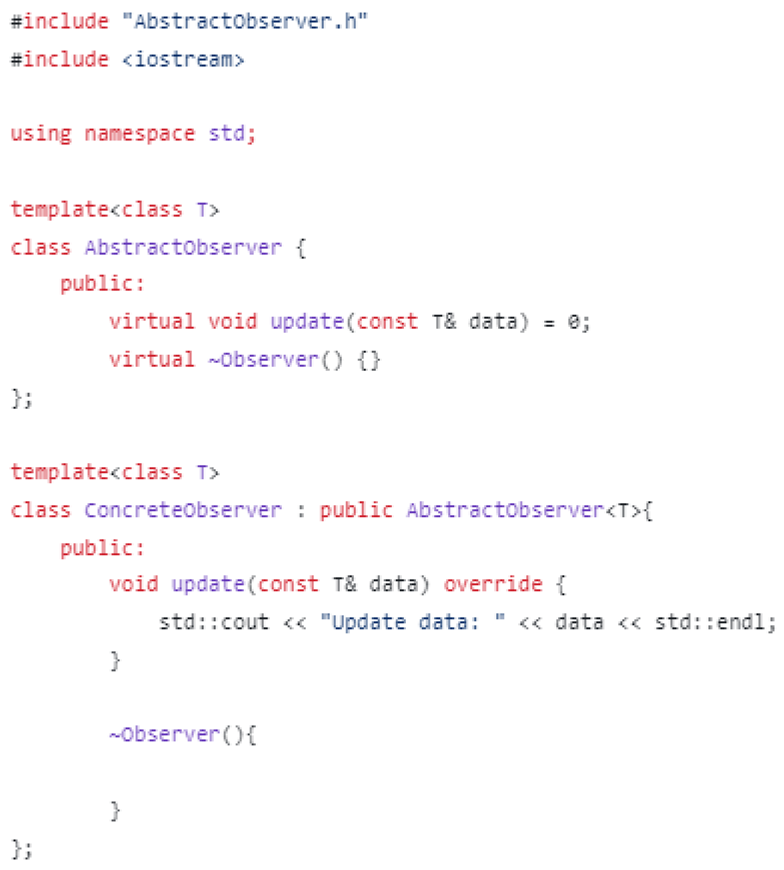
Main

```cpp
#include <tuple>
#include <memory>
#include <vector>
#include <iostream>
#include <algorithm>

using namespace std;

// Define a typelist to store multiple types
template<typename... Types>
struct TypeList {};

// Base Observer Interface for handling updates of different types
class BaseObserver {
public:
    virtual void updateName(const string& name) = 0;
    virtual ~BaseObserver() {}
};

// Templated Observer Interface
template<typename DataType>
class ObserverInterface : public BaseObserver {
public:
    virtual void update(const DataType& data) = 0;
};

// Concrete Observer that can observe multiple data types
template<typename TypeList>
class MultiTypeObserver;

// Specialization for TypeList
template<typename... Types>
class MultiTypeObserver<TypeList<Types...> > : public ObserverInterface<Types>... {
public:
    using ObserverInterface<Types>::update...;
};

// Stocks
// Stock update classes
class NaspersStock {
public:
    float price;
    explicit NaspersStock(float p) : price(p) {}
};
```

```
class SasolStock {
public:
    float price;
    explicit SasolStock(float p) : price(p) {}
};

// Observer that handles integer and string updates
class ConcreteObserver : public MultiTypeObserver<TypeList<NaspersStock, SasolStock>> {
private:
    string observerName;

public:
    ConcreteObserver(const string& name) : observerName(name) {}

    void update(const NaspersStock& data) override {
        cout << observerName << " has been notified that Naspers Stock price has changed to: " << data.price << endl;
    }

    void update(const SasolStock& data) override {
        cout << observerName << " has been notified that Sasol Stock price has changed to: " << data.price << endl;
    }

    void updateName(const string& name) override {
        observerName = name;
        cout << "Observer name updated to: " << observerName << endl;
    }
};

// Subject interface
template<typename DataType>
class SubjectInterface {
public:
    virtual void addObserver(std::shared_ptr<ObserverInterface<DataType> > observer) = 0;
    virtual void removeObserver(std::shared_ptr<ObserverInterface<DataType>> observer) = 0;
    virtual void notifyAll(const DataType& data) = 0;
    virtual ~SubjectInterface() {}
};

// Concrete subject implementation
template<typename DataType>
class SubjectImplementation : public SubjectInterface<DataType> {
    vector<shared_ptr<ObserverInterface<DataType> > > observers;
```

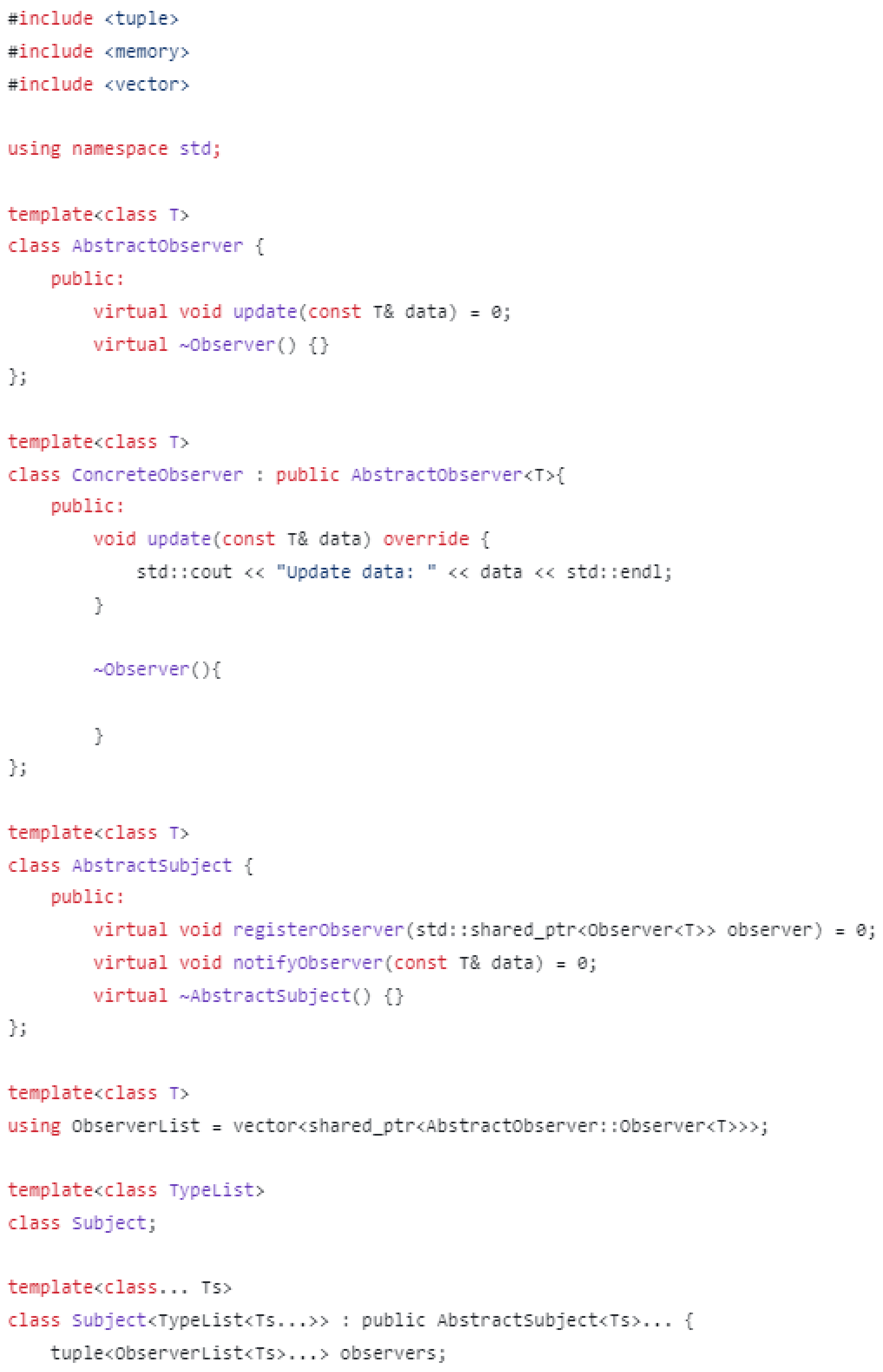
```cpp
public:
    void addObserver(shared_ptr<ObserverInterface<DataType> > observer) override {
        observers.push_back(observer);
    }

    void removeObserver(shared_ptr<ObserverInterface<DataType>> observer) override {
        observers.erase(
        remove_if(observers.begin(), observers.end(),
                    [&observer](const shared_ptr<ObserverInterface<DataType>>& element) {
                        return element == observer;
                    }),
        observers.end());
    }

    void notifyAll(const DataType& data) override {
        for (auto& observer : observers) {
            observer->update(data);
        }
    }
};

int main() {
    // Stock Exchange Example
    SubjectImplementation<NaspersStock> naspersSubject;
    SubjectImplementation<SasolStock> sasolSubject;

    auto John = make_shared<ConcreteObserver>("John");
    auto Steve = make_shared<ConcreteObserver>("Steve");

    cout << endl << "Adding Steve to Observe Sasol";
    cout << endl << "Adding John to Observe Sasol";
    cout << endl << "Adding John to Observe naspers" << endl << endl;

    naspersSubject.addObserver(John);
    sasolSubject.addObserver(Steve);
    sasolSubject.addObserver(John);

    cout << "Stocks changing price" << endl << endl;
    naspersSubject.notifyAll(NaspersStock(2500.50));
    sasolSubject.notifyAll(SasolStock(320.75));

    cout << endl << "Removing Steve from Observing Sasol" << endl << endl;


     cout << "Stocks changing price" << endl << endl;
     naspersSubject.notifyAll(NaspersStock(2570.50));
     sasolSubject.notifyAll(SasolStock(302.75));
     return 0;
 }
```

Observer



```
#include "AbstractObserver.h"
#include <iostream>

using namespace std;

template<class T>
class AbstractObserver {
    public:
        virtual void update(const T& data) = 0;
        virtual ~Observer() {}
};

template<class T>
class ConcreteObserver : public AbstractObserver<T>{
    public:
        void update(const T& data) override {
            std::cout << "Update data: " << data << std::endl;
        }

        ~Observer(){

        }
};
```

Subject

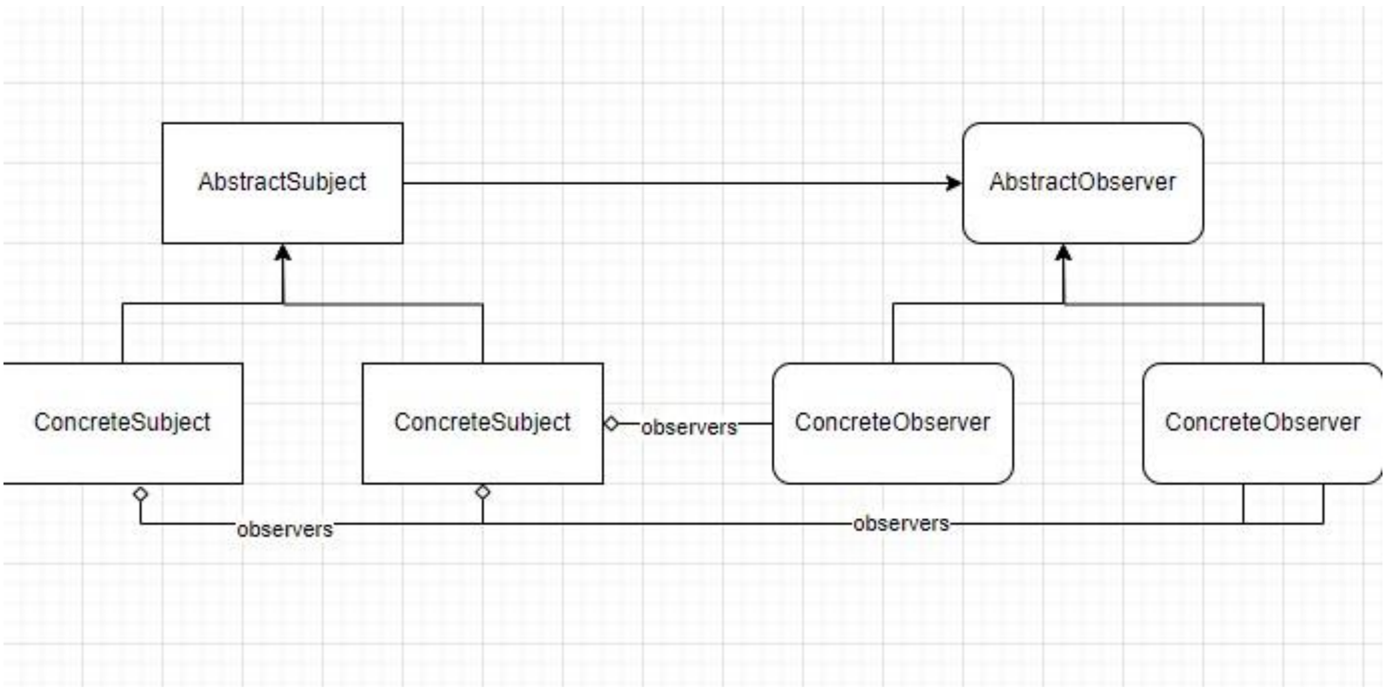


```cpp
#include <tuple>
#include <memory>
#include <vector>

using namespace std;

template<class T>
class AbstractObserver {
    public:
        virtual void update(const T& data) = 0;
        virtual ~Observer() {}
};

template<class T>
class ConcreteObserver : public AbstractObserver<T>{
    public:
        void update(const T& data) override {
            std::cout << "Update data: " << data << std::endl;
        }

        ~Observer(){

        }
};

template<class T>
class AbstractSubject {
    public:
        virtual void registerObserver(std::shared_ptr<Observer<T>> observer) = 0;
        virtual void notifyObserver(const T& data) = 0;
        virtual ~AbstractSubject() {}
};

template<class T>
using ObserverList = vector<shared_ptr<AbstractObserver::Observer<T>>>;

template<class TypeList>
class Subject;

template<class... Ts>
class Subject<TypeList<Ts...>> : public AbstractSubject<Ts>... {
    tuple<ObserverList<Ts>...> observers;
    public:
        template<class T>
        void registerObserver(shared_ptr<Observer<T>> observer) override {
            auto& observerList = get<ObserverList<T>>(observers);
            observerList.push_back(observer);
        }

        template<class T>
        void notifyObservers(const T& data) override {
            auto& observerList = get<ObserverList<T>>(observers);
            for (auto& observer : observerList){
                observer->update(data);
            }
        }

        ~AbstractSubject() override{

        }
};
```

Makefile

```
make:
        g++ main.cpp -o main.exe

run:
        ./main.exe
```

Diagram showcasing the structure of the program



This implementation is generic due to several key features that allow it to handle multiple types and operations dynamically:

TypeList and Variadic Templates: The TypeList struct combined with variadic templates (template<typename... Types>) allows the program to accept and work with an arbitrary number of types. This flexibility lets you define MultiTypeObserver for any combination of data types without modifying the underlying class structure.

Template Specialization: The specialization of MultiTypeObserver for TypeList<Types...> enables the observer to inherit from multiple ObserverInterface<Types> instances. This means a single observer can listen to updates for multiple data types, adhering to each type's specific update method.

Observer and Subject Interfaces: Both observer and subject are defined using templates, which makes them adaptable to any data type. This abstraction not only ensures type safety but also enhances reusability and scalability of the code.

Use of Shared Pointers: The use of std::shared_ptr for managing observers in SubjectImplementation ensures that memory management is handled automatically, and it's safe to use with different data types without worrying about memory leaks or dangling pointers.

Method Overloading and Overrides: The observer methods update(const DataType& data) are overridden for specific data types in ConcreteObserver, demonstrating how the observer can handle different types of data updates specifically.

These features collectively make the implementation highly flexible and adaptable to various data types and scenarios, which is a hallmark of generic programming.

observer -> subject In the Observer Pattern implementation, the interaction between subjects and observers is designed for flexibility and modularity. Observers, such as instances of ConcreteObserver, register with subjects like SubjectImplementation<NaspersStock> to be notified about specific events, such as changes in stock prices. The subject maintains a dynamically managed list of registered observers and, upon an event (e.g., a price update), iterates over this list to notify each observer by invoking their update method with the new data. This method is implemented by each observer to handle the received data appropriately, allowing for customized responses to the same event. Observers have the option to unregister from the subject if they no longer wish to receive updates, which is handled through the removeObserver method, ensuring that the observer is cleanly removed from the subject's notification list. This pattern effectively decouples the state management of the subject from the response behavior of the observers, enabling a system where changes in the subject's state are efficiently propagated to interested parties without requiring tight integration, thereby promoting scalability and maintainability.