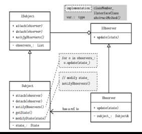
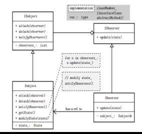




OBJECT-ORIENTED PROGRAMMING

MY PERSONAL NOTES ON

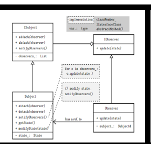
UML MODELLING DESIGN PATTERNS

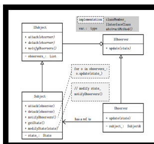
BY

leonmavr (github.com/leonmavr)

DECEMBER 16, 2023

DRAFT X.YY





Contents

1 UML class diagrams

1.1 Class representation

In Object-Oriented Programming (OOP), Unified Modeling Language (UML) class diagram is a notation to visualise classes, their data, operations and the relationships between them. A class is visualised with a rectangle while relationships are visualised by arrows.

Classes are visualised as boxes, with their public data and methods preceeded by + and their private ones by −. In Fig. 1, the ATM class has the `deposit(int amount)`, `withdraw(int amount)` as public methods and `cash` as private data.

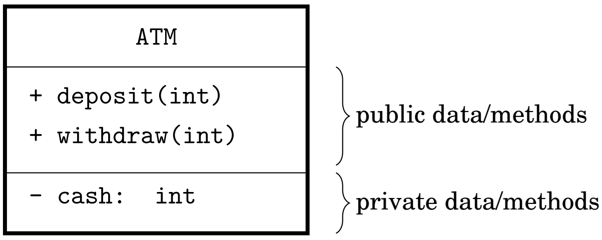


Fig. 1. UML block of a simplified ATM class.

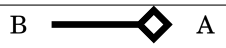
1.2 UML arrows

A and B denote two class instances. So normally they would be represented by boxes but for brevity they're represented by letters.

<table>
<tr><th>Arrow</th><th>Name and explanation</th><th>C++ example</th></tr>
<tr><td>B ⟶ A</td><td>Inheritance
B inherits from A, i.e. it inherits its public/private methods and data including their implementation. B is free to overwrite their implementation.</td><td><pre><code>class A {
public:
  // will be inherited
  int foo() { return 42; }
protected:
  // will be inherited
  unsigned bar() { return 1337; }
private:
  // will not be inherited
};

class B: public A {
public:
  unsigned bar() { return 0xdeadbeef; }
};

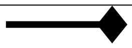
int main()
{
  B b;
  b.foo(); // 42
  b.bar(); // 0xdeadbeef
}</code></pre></td></tr>
</table>



(Weak) aggregation
B is associated with A but B's lifetime does not necessarily depend on A's – if B is destroyed, A may still live.
Summary: B has but shares an object B.

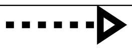
```cpp
#include <iostream>

class B
{
public:
  ~B() { std::cout << "B is destroyed\n"; }
};

class A
{
public:
  A(): obj(nullptr) {};
  ~A() { std::cout << "A is destroyed\n"; }
  void SetB(const B& b) { *obj = b; }
private:
  B* obj;
};
```

```cpp
#include <iostream>
class A {
public:
  ~A() { std::cout << "A is destroyed\n"; }
};

class B {
public:
  B(): obj(nullptr) {};
  ~B() { std::cout << "B is destroyed\n"; }
void SetA(const A& a) { *obj = a; }
private:
  A* obj;
};

int main() {
  A a;
  bool do_something = true;
  if (do_something) {
    B b;
    b.SetA(a);
  }
  // a is still alive
}
```

B  A

Strong aggregation aka *composition*. B fully contains A. Composition occurs when a class contains another one as part and lifetime of contained object (A) is tightly bound to the lifetime of the container (B). Summary: B has and owns an object A.

```
#include <iostream>

class A {
public:
  A() { std::cout << "A is created\n"; }
  ~A() { std::cout << "A is destroyed\n"; }
  void foo() { std::cout << "A is calling foo\n"; }
};

class B {
public:
  B() { std::cout << "B is created\n"; }
  ~B() { std::cout << "B is destroyed\n"; }
  A a;
private:
};

int main()
{
  B b;
  b.a.foo(); // a exists only within b
}
```

B  A

Realisation. B realises A. In this case, A is an interface; it defines but does not implement its methods. A's methods are called abstract. B inherits from it and implements its methods.

```
#include <iostream>

// interface class
class A {
public:
  // abstract (aka virtual) unimplemented methods
  virtual int foo() = 0;
  virtual int bar() = 0;
};


// inherit A and then implement all its methods
class B: public A {
public:
  // implement abstract methods (virtual->override)
  int foo() override { return 42; }
  int bar() override { return 1337; }
};

int main() {
  B b;
  std::cout << b.foo() << std::endl;
  std::cout << b.bar() << std::endl;
}
```

<table>
<tr>
<td>B ⟶ A</td>
<td>Association
Class B has a connection to class A. Association is a broad term to represent the “has-a” relationship between two classes. It means that an object of one class somehow communicates to an object of another. Summary: B has an object A.</td>
<td><pre><code>class A {
public:
  int foo() { return 420; }
};

class B {
public:
  B(A& a): a_(a) {};
  int bar() { return a_.foo(); }
private:
    A& a_; // has-a reference
};

int main() {
  A a;
  B b(a);
  b.bar(); // a.foo()
}</code></pre></td>
</tr>
</table>

Table 1: UML class diagram arrow meanings.

For example, the relationship of a shirt having a pocket is composition since a pocket only exists in a shirt but the relationship of a car having a wheel is aggregation as a wheel can be removed and used by another car.

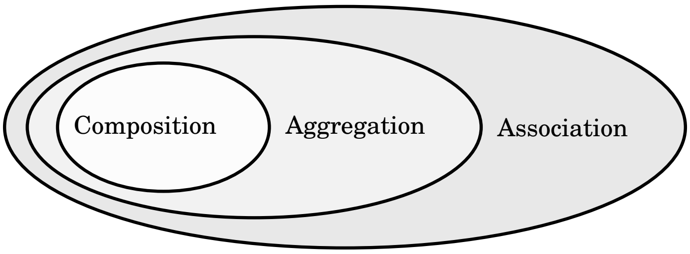


Fig. 2. Superset view of the “has-a” OOP relationships.

1.3 Abstract Class vs Interface

Interface and *abstract class* are two terms to define classes that have at least one *pure virtual method* (aka *abstract method*), i.e. one unimplemented method meant to be implemented by a subclass. In C++, pure (virtual) methods are denoted by the following syntax:

```
virtual void MyAbstractMethod(int i) = 0;
```

The difference is that an interface ONLY contains abstract methods while an abstract class is simply one with at least one abstract method and is allowed to contain its own implemented (non-virtual) methods as well.

Interface	Abstract class

`class Interface {`
`public:`
`  // virtual destructor to ensure`
`  // proper deletion`
`  virtual ~Interface() {};`
`  virtual void AbstractMethod1() = 0;`
`  virtual void AbstractMethod2() = 0;`
`};`
	`class AbstractClass {`
	`public:`
	`  // virtual destructor`
	`  virtual ~Interface() {};`
	`  virtual void AbstractMethod1() = 0;`
	`  virtual void AbstractMethod2() = 0;`
	`  int Method1(int i);`
	`};`

Table 2: The difference between interface and abstract class in C++.

2 Software Design Patterns

A design pattern is a tried and true solution to a common problem. Particularly, they provide standard OOP solutions to common problems while making components of the system reusable.

2.1 The 23 Gang of Four Design Patterns

Creational patterns deal with object creation mechanisms, *structural patterns* ease the design by identifying a simple way to realise relationships between entities and *behavioural patterns* are concerned with communication between objects. There exist 23 established design patterns listed in Table 3.

Creational	Structural	Behavioural
Factory	Adapter	Interpreter
Abstract factory	Bridge	Template method
Builder	Composite	Chain of responsibility
Prototype	Decorator	Command
Singleton	Facade	Iterator
	Flyweight	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

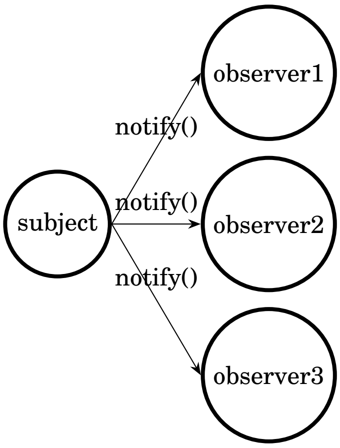
Table 3: The "Gang of 4" 23 fundamental design patterns.

2.1.1 Observer

The aim of the observer pattern is to propagate state changes of the object to be observed (subject) into multiple observer objects. It does this by calling each update method of its observers.

For example in a program to monitor stocks where the subject stores a list of stock prices. Implementing the graph view and the tabular view inside the subject itself would make it cluttered and hard to maintain. Therefore it's better to assign the responsibility of observing the price to some UI and tabular view classes and whenever the price changes they're updated.

More formally, we say that the observer pattern defines a one-to-many relationship so that when an object changes all its dependents are notified automatically when the state of the subject changes.



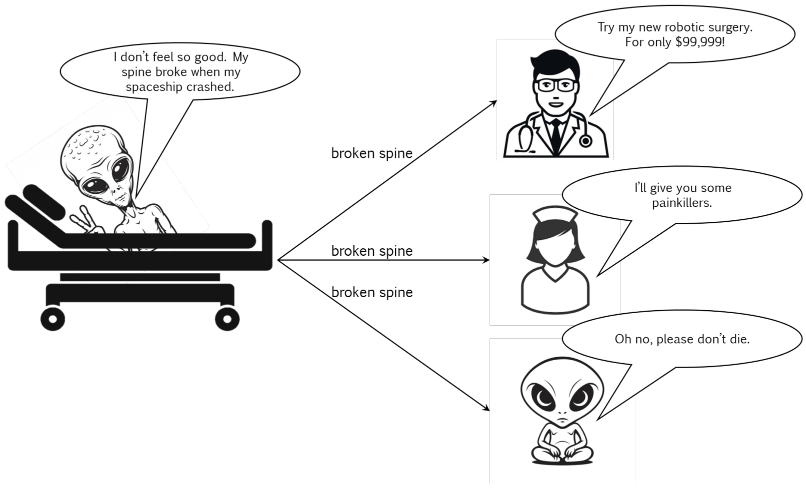


Fig. 3. The observer pattern describes and one-to-many relationship.

Fig. 4. The alien subject reports his health to the observers on the right.

Let's define the classes this pattern uses:

- `ISubject` – the abstract subject (aka subject interface). Defines the abstract attach, detach and notify methods.
- `Subject` – The object of interest whose internal state changes we want to observe. It maintains a list of observers and it is able to *attach* an observer to it, *detach* it, or *notify* all observers. [1] It's able to modify and return the state.
- `IObserver` – the observer interface. Defines the abstract *update* method.
- `ConcreteObserverA`, `ConcreteObserverB`, ... These subclasses of `IObserver` inherit from it and implement the update method. It's also convenient for them to store a reference to `Subject` in order to query its data if necessary.

Now whenever the state to be observed changed in the `Subject`, it is responsible to call the `notifyObservers` method inside the method itself. For example if `Subject` has a method `motifyState`, it is responsible to call `notifyObserver` in it in order to broadcast the new state to the observers. Expressing these ideas in UML, the following diagram is constructed. In the diagram, `Subject` and `Observer` are concrete classes.

[1]Strictly speaking, it maintains a list of `IObservers` and concreted observers, which inherit from `IObserver` are appended to it via the attach method. Due to polymorphism the list can accommodate all subclasses of it. Hence `IObserver` is downcast to the class of the concrete observer.

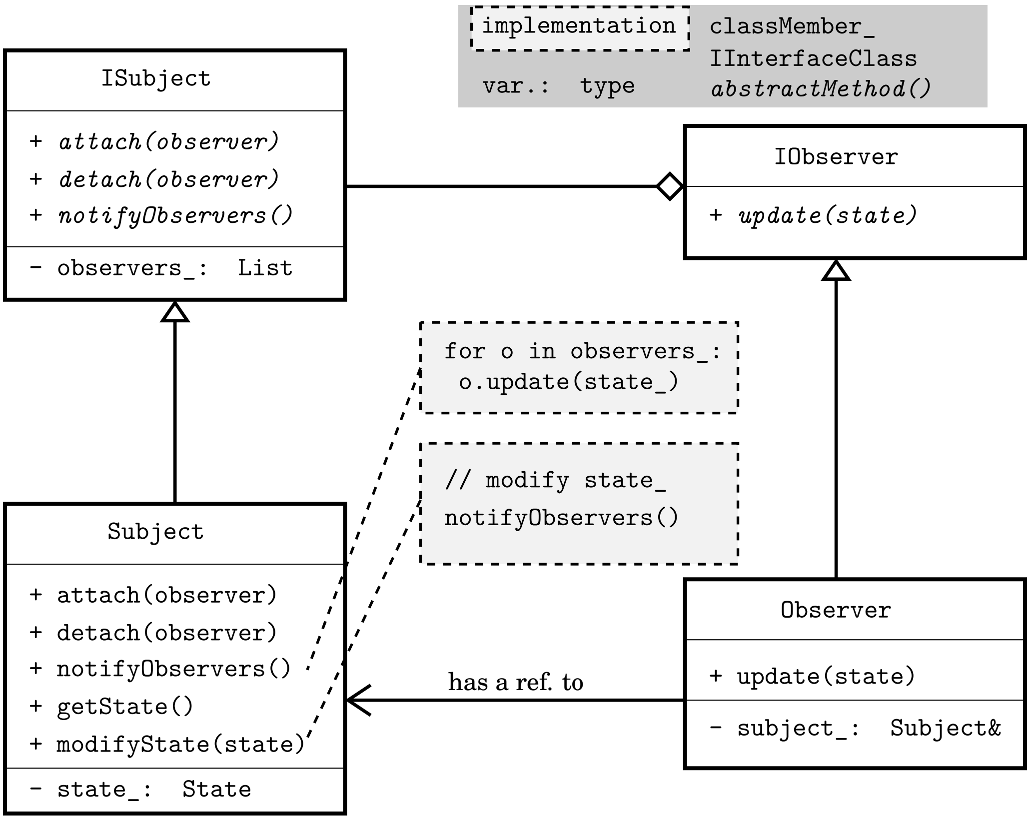


Fig. 5. UML diagram of the observer design pattern.

A practical example that demonstrates the observer design pattern is a model of the stock market found in A.1.

A `StockMarket` subject calls `UpdateState()` to update its ticker (fancy word for stock symbol) pairs modelled by the state variable `pairs_`. The latter stores the price for each stock marker symbol in a dictionary, e.g. `{{GOOG: 152.99}, {NVDA: 461.72}}`. `UpdateState()` furthermore calls `NotifyObservers()`, which in turns goes through all observers and for each pair in `pairs_` it calls `Update(std::string ticker, double price)`. `ticker` is the first field of each pair and `price` its second. `Investor` is a dummy concrete observer but bot keeps a history of each pair in its internal variables, e.g. `GOOG: [148, 152, 149]` hence can perform a simulation of an analysis.

Table 4 shows how the diagram in Fig. 5 corresponds to the stock market code in A.1.

Diagram	**Code**	**Diagram**	**Code**
`Subject`	`StockMarket`	`attach`	`AttachObserver`
`detach`	`DetachObserver`	`update(state)`	`Update(std::string, int)`
`state_`	`pairs_`	`modifyState`	`UpdatePrices`
`getState()`	`pairs()`	`ConcreteObserver`	`Bot, Investor`

Table 4: How the naming in Fig. 5 corresponds to that in A.1's code.

In the end, the two observers report their updates and the bot additionally makes its super sophisticated and advanced analysis.

```
-------- day 15 --------
  Investor Alice received update: AAPL price is 183.4
  Investor Alice received update: NVDA price is 477.4
  Investor Alice received update: GOOG price is 154.3
  Bot received an update of 183.4 on AAPL ticker
  Bot received an update of 477.4 on NVDA ticker
```

```
Bot received an update of 154.3 on GOOG ticker
Bot says: AAPL's tomorrow price will be 189.28 with RSI = 72 --> SELL
Bot says: NVDA's tomorrow price will be 476.77 with RSI = 68 --> HOLD
Bot says: GOOG's tomorrow price will be 163.90 with RSI = 24 --> BUY
```

2.2 Other Design Patterns

A Appendices

A.1 Observer: stock market model

Listing 1: A stock market system modelled by the observer design apttern (src/observer.cpp).

```cpp
#include <iostream>
#include <iomanip>
#include <vector>
#include <deque>
#include <ctime>
#include <cstdlib>
#include <algorithm>
#include <unordered_map>
#include <cmath>
#include <memory>

// forward declaration of subject as it's required by a concrete observer
class StockMarket;

// observer interface
class IObserver {
public:
    virtual void Update(const std::string& stockSymbol, double price) = 0;
    virtual ~IObserver() = default;
};


// Concrete observer A
class Bot: public IObserver {
public:
    Bot(StockMarket& stock_market);
    void Update(const std::string& stockSymbol, double price) override;
    void Predict(const std::string& ticker);
private:
    StockMarket& stock_market_;
    std::unordered_map<std::string, std::deque<double>> price_history_;
    unsigned hist_length_;
};


// subject interface
class ISubject {
public:
    virtual void AttachObserver(std::shared_ptr<IObserver> investor) = 0;
    virtual void DetachObserver(std::shared_ptr<IObserver> investor) = 0;
    virtual void NotifyObservers() = 0;
    virtual ~ISubject() = default;
protected:
    std::vector<std::shared_ptr<IObserver>> observers_;
};


// Concrete subject
class StockMarket : public ISubject {
public:
    StockMarket() = delete;
    StockMarket(std::unordered_map<std::string, double> prices) : pairs_(prices
    ) {}

    void AttachObserver(std::shared_ptr<IObserver> observer) override {
        observers_.push_back(observer);
    }

    void DetachObserver(std::shared_ptr<IObserver> observer) override {
        auto it = std::find(observers_.begin(), observers_.end(), observer);
        if (it != observers_.end())
```

```
            observers_.erase(it);
    }

    void NotifyObservers() override {
        for (auto observer : observers_) {
            for (const auto& pair: pairs_) {
                observer->Update(pair.first, pair.second);
            }
        }
    }

    // Simulate a change in the state variable and notify observers
    void UpdatePrices() {
        for (auto& pair: pairs_) {
            auto price = pair.second;
            pair.second += 0.03*price * (rand()%100 - 40)/100;
        }
        NotifyObservers(); // Notify all registered observers
    }

    std::unordered_map<std::string, double> pairs() const {
        return pairs_;
    }

private:
    // state variable of subject - observers are interested in it
    std::unordered_map<std::string, double> pairs_;
};


// Concrete observer B
class Investor: public IObserver {
public:
    Investor(const std::string& name, StockMarket& stock_market) :
        name_(name),
        stock_market_(stock_market) {}
    void Update(const std::string& stockSymbol, double price) override {
        std::cout << "\tInvestor " << name_ << " received update: "
            << stockSymbol << " price is " << std::fixed
            << std::setprecision(1) << price << std::endl;
    }
private:
    std::string name_;
    StockMarket& stock_market_;
};


// Concrete observer B
Bot::Bot(StockMarket& stock_market) :
    stock_market_(stock_market),
    hist_length_(7) {
    for (auto& pair: stock_market_.pairs()) {
        const auto symbol = pair.first;
        const auto price = pair.second;
        std::deque<double> price_copies;
        // push N copies of the current price to each ticker to initialise it
        for (int i = 0; i < hist_length_; ++i)
            price_copies.push_back(price);
        price_history_[symbol] = price_copies;
    }
};

void Bot::Update(const std::string& ticker, double price) {
    std::cout << "\tBot received an update of " << price << " on " <<
```

```cpp
    ticker << " ticker" << std::endl;
    auto it = price_history_.find(ticker);
    if (it != price_history_.end()) {
        it->second.pop_front();
        it->second.push_back(price);
    }
}

// predict next price, estimate a technical indicator, suggest buy/sell/hold
void Bot::Predict(const std::string& ticker) {
    std::cout << "\tBot says: " << ticker << "'s tomorrow price will be ";
    auto it = price_history_.find(ticker);
    if (it != price_history_.end()) {
        const auto prices = it->second;
        // "predict" it as the moving average with some
        // positively biased randomness
        double prediction = 0.0;
        for (auto p: prices)
            prediction += p;
        prediction /= prices.size();
        prediction += rand() % 20 - 5;
        // model the RSI by my arbitrary definition
        std::cout << std::fixed << std::setprecision(2)
                  << prediction << " with RSI = ";
        auto it = std::max_element(prices.begin(), prices.end());
        double max = *it;
        it = std::min_element(prices.begin(), prices.end());
        double min = *it;
        double curr = prices[prices.size() - 1];
        int rsi_perc = static_cast<int>(std::round((curr - min)/(max - min + 0.0001) * 100));
        // simulate an analysis (buy/hold/sell)
        std::string suggestion = "HOLD";
        if (rsi_perc > 70)
            suggestion = "SELL";
        else if (rsi_perc < 30)
            suggestion = "BUY";
        std::cout << rsi_perc << " --> " << suggestion << std::endl;
    }
}


int main() {
    // Create an instance of the subject (stock market)
    std::unordered_map<std::string, double> trading_pairs =
        {{"GOOG", 150}, {"NVDA", 470}, {"AAPL", 180}};
    auto stock_market = StockMarket(trading_pairs);
    // Create instances of observers (investors/bots)
    auto investor = std::make_shared<Investor>("Alice", stock_market);
    auto bot = std::make_shared<Bot>(stock_market);
    // Attach observers to the subject
    stock_market.AttachObserver(investor);
    stock_market.AttachObserver(bot);

    // Simulate changes in stock prices
    srand(static_cast<unsigned>(time(nullptr)));
    constexpr int ndays = 20;
    for (int i = 0; i < ndays; ++i) {
    // wait for some samples to collect some more meaningful data
    if (i > 5) {
            std::cout << "-------- day " << i << " --------" << std::endl;
            stock_market.UpdatePrices();
            for (auto& pair: stock_market.pairs())
                bot->Predict(pair.first);
```

```
        }
    }
    // Detach all observers
    stock_market.DetachObserver(investor);
    stock_market.DetachObserver(bot);
    return 0;
}
```