Design Patternek

OOP és DB - OOP 8. óra

Mi az, hogy design pattern?

Olyan absztrakció, ami mintázatot ad egy rendszer strukturálásához, felépítéséhez.

A híres “Gang of Four” nyomán alakult ki ez a kifejezés (E. Gamma, R. Helm, R. Johnson, J. Vlissides) - akik 1994-ben publikálták a “Design Patterns: Elements of Reusable Object-Oriented Software” c. könyvüket.

> Olyan nagy hatású könyv volt, hogy sokan ma is esküsznek a design patternekre.
>
> Természetesen kritikusai is vannak ennek a megközelítésnek, ld. később.
>
> Azért érdemes őket ismerni és ahol hasznos, felhasználni őket.

Design patternek fő típusai

Creational patterns (*kreációs minták*) - arra adnak választ, hogy hogyan hozzunk létre új objektumokat (akkor érdekes, ha pl. futásidőben dől el, hogy pontosan milyen típusú objektumot kell példányosítanunk, vagy ha több lépcsőben kell szabályozni, hogy hogyan jöjjön létre egy objektum, vagy ha egyéb szabályszerűségeket kell a példányosításnak kielégítenie).

Structural patterns (*strukturális minták*) - entitások közötti kapcsolatok hatékony megvalósítására, nevezetesen akkor is, ha alapesetben nem kompatibilisek egymással.

Behavioral patterns (*viselkedési minták*) - objektumok közötti kommunikáció hatékony megvalósítására, főleg akkor, amikor a kapcsolat többféleképpen létrejöhet és nem feltétlenül direkt módon.

Kreációs minta I: Builder

Probléma: összetett objektumot szeretnénk létrehozni, de nem szeretnénk, hogy ehhez összetett (sok-paraméteres) konstruktort, vagy sok különböző paraméterezésű konstruktort kelljen készíteni / használni

Megoldás: hozzunk létre egy köztes (Builder) objektumot, aminek a metódusai a létrehozandó objektum különböző részeit inicializálják.

Példa: egy pizzának többféle tésztája, feltéte lehet. Egy bonyolult Pizza konstruktor helyett használjuk a PizzaBuilder osztályt! A PizzaBuilder osztályból származtatva sokféle típusú pizza létrehozható.

Builder - példa

```
// "Product"
class Pizza
{
public:
    void setDough(const string& dough)
    {
        m_dough = dough;
    }
    void setSauce(const string& sauce)
    {
        m_sauce = sauce;
    }
    void setTopping(const string& topping)
    {
        m_topping = topping;
    }
    void open() const
    {
        cout << "Pizza with " << m_dough << " dough, " << m_sauce << " sauce and "
            << m_topping << " topping. Mmm." << endl;
    }
private:
    string m_dough;
    string m_sauce;
    string m_topping;
};
```

Tegyük fel, hogy nem akarunk bonyolult konstruktort használni, ahol a pizza minden részletét aprólékosan meg kell adni.

Builder - példa

```
// "Abstract Builder"
class PizzaBuilder
{
public:
    virtual ~PizzaBuilder() {};

    Pizza* getPizza()
    {
        return m_pizza.release();
    }
    void createNewPizzaProduct()
    {
        m_pizza = make_unique<Pizza>();
    }
    virtual void buildDough() = 0;
    virtual void buildSauce() = 0;
    virtual void buildTopping() = 0;
protected:
    unique_ptr<Pizza> m_pizza;
};
```

unique_ptr - ld. cppreference.com.

A manage-elt pointerek segítenek a pointerek helyes életciklus-menedzsmentjében

Builder - példa

```
class HawaiianPizzaBuilder : public PizzaBuilder
{
public:
    virtual ~HawaiianPizzaBuilder() {};

    virtual void buildDough()
    {
        m_pizza->setDough("cross");
    }
    virtual void buildSauce()
    {
        m_pizza->setSauce("mild");
    }
    virtual void buildTopping()
    {
        m_pizza->setTopping("ham+pineapple");
    }
};
```

```
class SpicyPizzaBuilder : public PizzaBuilder
{
public:
    virtual ~SpicyPizzaBuilder() {};

    virtual void buildDough()
    {
        m_pizza->setDough("pan baked");
    }
    virtual void buildSauce()
    {
        m_pizza->setSauce("hot");
    }
    virtual void buildTopping()
    {
        m_pizza->setTopping("pepperoni+salami");
    }
};
```

Builder - példa

```
class Cook
{
public:
    void openPizza()
    {
        m_pizzaBuilder->getPizza()->open();
    }
    void makePizza(PizzaBuilder* pb)
    {
        m_pizzaBuilder = pb;
        m_pizzaBuilder->createNewPizzaProduct();
        m_pizzaBuilder->buildDough();
        m_pizzaBuilder->buildSauce();
        m_pizzaBuilder->buildTopping();
    }
private:
    PizzaBuilder* m_pizzaBuilder;
};
```

```
int main()
{
    Cook cook;
    HawaiianPizzaBuilder hawaiianPizzaBuilder;
    SpicyPizzaBuilder    spicyPizzaBuilder;

    cook.makePizza(&hawaiianPizzaBuilder);
    cook.openPizza();

    cook.makePizza(&spicyPizzaBuilder);
    cook.openPizza();
}
```

Kreációs minta II: Factory

Probléma: Futásidőben szeretnénk eldönteni, hogy pontosan milyen objektumot hozunk létre. Fordításidőben még nem tudjuk, melyiket szeretnénk.

Megoldás: Készítsünk egy interfészt az objektum létrehozására, de intézzük úgy, hogy az objektum csak a származtatott osztályokon keresztül legyen példányosítható.

Példa: Paramétertől függően a ComputerFactory vagy desktop, vagy laptop gépet példányosít. Ettől függetlenül mindenképpen Computer* típusú változót ad vissza.

Factory - példa

```
class Computer
{
public:
    virtual void Run() = 0;
    virtual void Stop() = 0;

    virtual ~Computer() {}; /* without this, you do not call Laptop or Desktop destructor in
this example! */
};
```

Factory - példa

```
class Laptop: public Computer
{
public:
    void Run() override {mHibernating = false;};
    void Stop() override {mHibernating = true;};
    virtual ~Laptop() {}; /* because we have virtual functions, we need virtual destructor
*/
private:
    bool mHibernating; // Whether or not the machine is hibernating
};
```

```
class Desktop: public Computer
{
public:
    void Run() override {mOn = true;};
    void Stop() override {mOn = false;};
    virtual ~Desktop() {};
private:
    bool mOn; // Whether or not the machine has been turned on
};
```

Factory - példa

```
class ComputerFactory
{
public:
    static Computer *NewComputer(const std::string &description)
    {
        if(description == "laptop")
            return new Laptop;
        if(description == "desktop")
            return new Desktop;
        return nullptr;
    }
};
```

Builder és Factory összehasonlítása:

Buildernél a Builderből volt többféle típus (HawaianPizzaBuilder, SpicyPizzaBuilder), és csak egyféle Pizza osztály volt. Ezen osztály egyazon metódusait hívtuk meg más és más paraméterekkel.

Factory-nál egy factory osztály volt és az valamilyen (mindig más és más) származtatott osztályt példányosított.

Factory - példa 2

Pizzát is lehet Factory patternnel létrehozni:

```
class Pizza {
public:
    virtual int getPrice() const = 0;
    virtual ~Pizza() {};  /* without this, no destructor for derived Pizza's will be called.
*/
};

class HamAndMushroomPizza : public Pizza {
public:
    virtual int getPrice() const { return 850; };
    virtual ~HamAndMushroomPizza() {};
};

class DeluxePizza : public Pizza {
public:
    virtual int getPrice() const { return 1050; };
    virtual ~DeluxePizza() {};
};
```

Factory - példa 2

```
class HawaiianPizza : public Pizza {
public:
    virtual int getPrice() const { return 1150; };
    virtual ~HawaiianPizza() {};
};

class PizzaFactory {
public:
    enum PizzaType {
        HamMushroom,
        Deluxe,
        Hawaiian
    };

    static unique_ptr<Pizza> createPizza(PizzaType pizzaType) {
        switch (pizzaType) {
        case HamMushroom: return make_unique<HamAndMushroomPizza>();
        case Deluxe:      return make_unique<DeluxePizza>();
        case Hawaiian:    return make_unique<HawaiianPizza>();
        }
        throw "invalid pizza type.";
    }
};
```

Factory - példa 2

```
/*
* Create all available pizzas and print their prices
*/
void pizza_information(PizzaFactory::PizzaType pizzatype)
{
    unique_ptr<Pizza> pizza = PizzaFactory::createPizza(pizzatype);
    cout << "Price of " << pizzatype << " is " << pizza->getPrice() << std::endl;
}

int main()
{
    pizza_information(PizzaFactory::HamMushroom);
    pizza_information(PizzaFactory::Deluxe);
    pizza_information(PizzaFactory::Hawaiian);
}
```

Kreációs minta III: Singleton

Probléma: Vannak esetek, amikor garantálni szeretnénk, hogy egy osztálynak csak egy példánya létezzen az egész alkalmazásban. Tipikusan a menedzser osztályok ilyenek (adatbázis menedzser, hozzáférés-menedzser, stb. stb.)

Megoldás: a singleton osztályban egy privát, statikus változóban van a singleton példány. Ezt a példányt egy publikus accessor (tipikusan *instance*() vagy *getInstance*() nevű) metóduson keresztül lehet elérni.

Fontos az is, hogy a konstruktor és copy constructor, valamint a copy assignment vagy privátak, vagy letiltottak legyenek. Ezzel garantálhatjuk, hogy az osztályt példányosítani / duplikálni nem lehet.

Singleton - példa

```
class StringSingleton
{
public:
    // Some accessor functions for the class, itself
    std::string GetString() const
    {return mString;}
    void SetString(const std::string &newStr)
    {mString = newStr;}

    // The magic function, which allows access to the class from anywhere
    // To get the value of the instance of the class, call:
    //     StringSingleton::Instance().GetString();
    static StringSingleton &Instance()
    {
        // This line only runs once, thus creating the only instance in existence
        static std::auto_ptr<StringSingleton> instance( new StringSingleton );
        // dereferencing the variable here, saves the caller from having to use
        // the arrow operator, and removes temptation to try and delete the
        // returned instance.
        return *instance; // always returns the same instance
    }
```

Singleton - példa

```
 private:
     // We need to make some given functions private to finish the definition of the
singleton
     StringSingleton(){} // default constructor available only to members or friends of this
class

     // Note that the next two functions are not given bodies, thus any attempt
     // to call them implicitly will return as compiler errors. This prevents
     // accidental copying of the only instance of the class.
     StringSingleton(const StringSingleton &old); // disallow copy constructor
     const StringSingleton &operator=(const StringSingleton &old); //disallow assignment
operator

     // Note that although this should be allowed,
     // some compilers may not implement private destructors
     // This prevents others from deleting our one single instance, which was otherwise
created on the heap
     ~StringSingleton(){}
 private: // private data for an instance of this class
     std::string mString;
 };
```

Strukturális minta I: Adapter

Probléma: A szoftver egyik része egy adott interfészt biztosító objektumokkal működik, de mi egy másmilyen interfészt biztosító objektummal is használni szeretnénk.

Megoldás: hozzunk létre egy adapter osztályt, amely tartalmaz egy változót (vagy hivatkozást változóra) az eredeti típusból, és kiegészíti azt egy újfajta interfésszel.

Példa: miért ne sprintelhetne egy maraton futó?

```
class Person {
public:
    Person(const std::string nm) : name(nm) {}
    std::string getName() { return name; }
private:
    std::string name;
};
```

Adapter - példa

```
class MarathonRunner : public Person {
public:
    MarathonRunner(const std::string& n) : Person(n) {}
    void runMarathon() { std::cout << getName() << ": Running marathon!" << std::endl;
};

class Sprinter : public Person {
public:
    Sprinter(const std::string& n) : Person(n) {}
    void runSprint() { std::cout << getName() << ": Running sprint!" << std::endl; }
};
```

Adapter - példa

```
class MarathonRunnerAdapter {
private:
    MarathonRunner * mr;
public:
    MarathonRunnerAdapter(MarathonRunner* mrp) { mr = mrp; }
    void runSprint() {
        std::cout << mr->getName() << " is running a sprint for a change!" << std::endl;
    }
};

int main()
{
    MarathonRunner kipcho("E. Kipchoge");
    Sprinter bolt("U. Bolt");
    MarathonRunnerAdapter(&kipcho).runSprint();
    std::cin.get();
    return 0;
}
```

Strukturális minta II - Decorator

“Wrapper” néven is ismert. A minta lényege, hogy dinamikusan csatolunk hozzá további funkciókat / viselkedés-módosulásokat az adott osztályhoz.

Mindezt nem úgy tesszük, hogy az osztályt kiterjesztjük. Ez jó, mert így többféle osztály is egyazon dekorátorral kiterjeszthető (nem kell mindegyikből u.azt származtatni)

A példában van egy *Car* nevű absztrakt ősosztály, és egy *CarModel1* nevű konkrét leszármazott osztály. Annak érdekében, hogy a *CarModel1*-nek további funkciókat adjunk, nem a *CarModel1* osztályt terjesztjük ki, hanem készítünk a *Car* osztálynak egy további - absztrakt leszármazottat: az *OptionsDecorator* osztályt. Ebből származnak a további, konkrét decorator osztályok, amik egy-egy speciális Car objektumot tudnak különféleképpen módosítani.

Decorator - példa

```
class Car //Our Abstract base class
{
        protected:
                string _str;
        public:
                Car()
                {
                        _str = "Unknown Car";
                }

                virtual string getDescription()
                {
                        return _str;
                }

                virtual double getCost() = 0;

                virtual ~Car()
                {
                        cout << "~Car()\n";
                }
};
```

```
class CarModel1 : public Car
{
        public:
                CarModel1()
                {
                        _str = "CarModel1";
                }
                virtual double getCost()
                {
                        return 31000.23;
                }

                ~CarModel1()
                {
                        cout<<"~CarModel1()\n";
                }
};
```

Decorator - példa

```
class Car //Our Abstract base class
{
        protected:
                string _str;
        public:
                Car()
                {
                        _str = "Unknown Car";
                }

                virtual string getDescription()
                {
                        return _str;
                }

                virtual double getCost() = 0;

                virtual ~Car()
                {
                        cout << "~Car()\n";
                }
};
```

A Decorator alap osztály is Car-ból származik!

+ egy már implementált virtuális metódusból csinál pure virtual metódust

```
class OptionsDecorator : public Car //Decorator Base class
{
        public:
                virtual string getDescription() = 0;

                virtual ~OptionsDecorator()
                {
                        cout<<"~OptionsDecorator()\n";
                }
};
```

Decorator - példa

```
class Navigation: public OptionsDecorator
{
                Car *_b;
        public:
                Navigation(Car *b)
                {
                        _b = b;
                }
                string getDescription()
                {
                        return _b->getDescription() + ", Navigation";
                }

                double getCost()
                {
                        return 300.56 + _b->getCost();
                }
                ~Navigation()
                {
                        cout << "~Navigation()\n";
                        delete _b;
                }
};
```

A konkrét dekorátorok meg kell, hogy valósítsák a pure virtual metódust

```
class PremiumSoundSystem: public OptionsDecorator
{
                Car *_b;
        public:
                PremiumSoundSystem(Car *b)
                {
                        _b = b;
                }
                string getDescription()
                {
                        return _b->getDescription() + ", PremiumSoundSystem";
                }

                double getCost()
                {
                        return 0.30 + _b->getCost();
                }
                ~PremiumSoundSystem()
                {
                        cout << "~PremiumSoundSystem()\n";
                        delete _b;
                }
};
```

Decorator - példa

```
int main()
{
        //Create our Car that we want to buy
        Car *b = new CarModel1();

        cout << "Base model of " << b->getDescription() << " costs $" << b->getCost() <<
"\n";

        //Who wants base model let's add some more features

        b = new Navigation(b);
        cout << b->getDescription() << " will cost you $" << b->getCost() << "\n";
        b = new PremiumSoundSystem(b);
        b = new ManualTransmission(b);
        cout << b->getDescription() << " will cost you $" << b->getCost() << "\n";

        // WARNING! Here we leak the CarModel1, Navigation and PremiumSoundSystem objects!
        // Either we delete them explicitly or rewrite the Decorators to take
        // ownership and delete their Cars when destroyed.
        delete b;

        return 0;
}
```

Decorator - példa

Lehetett volna, hogy *NavigationCar*, *PremiumSoundSystemCar* és *ManualTransmissionCar* osztályok a *CarModel1-ből* származnak tovább.

De ez esetben ha készítünk egy másik, *CarModel2* osztályt, ugyanúgy le kellett volna ezeket klónozni

Ezért hasznos, hogy a decorator osztályok “más ágon” vannak - így bármilyen konkrét Car altípust “ki lehet velük dekorálni”.

Viselkedési minta I - Command

Az OOP-ben (és a programozásban általában) célszerű nem előre behuzalozni, hogy ki kommunikálhat kivel.

Különösen az OOP-ben előfordulhat, hogy változik, mennyire specializált osztály szeretne mennyire általános interfészhez hozzáférni.

Funkcionális programozásban a *callback* mechanizmus támogatja, hogy egy függvény lefutását követően paraméterezhető legyen, hogy mi történjen

Ezt C++-ban is lehet csinálni.

A Command pattern osztályok szintjén tesz lehetővé valami hasonlót.

Viselkedési minta I - Command

Alapötlet: reprezentáljuk a kérést egy objektummal. Az objektumban sokféle információt tárolhatunk, + könnyebb utólag is visszanézni, hogy mi történt.

```
/*the Command interface*/
class Command
{
public:
    virtual void execute()=0;
};
```

```
/*Receiver class*/
class Light {

public:
    Light() {  }

    void turnOn()
    {
        cout << "The light is on" << endl;
    }

    void turnOff()
    {
        cout << "The light is off" << endl;
    }
};
```

Viselkedési minta I - Command

Command egy absztrakt osztály, amiből a specializáltabb parancs típusok származnak.

```
/*the Command for turning on the light*/
class FlipUpCommand: public Command
{
public:

    FlipUpCommand(Light& light):theLight(light)
    {

    }

    virtual void execute()
    {
        theLight.turnOn();
    }

private:
    Light& theLight;
};
```

```
/*the Command for turning off the light*/
class FlipDownCommand: public Command
{
public:
    FlipDownCommand(Light& light) :theLight(light)
    {

    }
    virtual void execute()
    {
        theLight.turnOff();
    }
private:
    Light& theLight;
};
```

Viselkedési minta I - Command

Switch osztály egy összetettebb funkcionalitást tesz lehetővé a korábban definiált Command-altípusok segítségével:

```
class Switch {
public:
    Switch(Command& flipUpCmd, Command& flipDownCmd)
    :flipUpCommand(flipUpCmd),flipDownCommand(flipDownCmd)
    {

    }

    void flipUp()
    {
        flipUpCommand.execute();
    }

    void flipDown()
    {
        flipDownCommand.execute();
    }

private:
    Command& flipUpCommand;
    Command& flipDownCommand;
};
```

```
/*The test class or client*/
int main()
{
    Light lamp;
    FlipUpCommand switchUp(lamp);
    FlipDownCommand switchDown(lamp);

    Switch s(switchUp, switchDown);
    s.flipUp();
    s.flipDown();
}
```

Viselkedési minta II - Mediator

Az általános viselkedések rugalmasabb implementálásához néha célszerű, hogy különböző objektumok ne direktben, hanem egy mediátoron keresztül kommunikáljanak.

```
class MediatorInterface;

class ColleagueInterface {
        std::string name;
    public:
        ColleagueInterface (const std::string& newName) : name (newName) {}
        std::string getName() const {return name;}
        virtual void sendMessage (const MediatorInterface&, const std::string&) const = 0;
        virtual void receiveMessage (const ColleagueInterface*, const std::string&) const = 0;
};
```

Viselkedési minta II - Mediator

```
class Colleague : public ColleagueInterface {
    public:
        using ColleagueInterface::ColleagueInterface;
        virtual void sendMessage (const MediatorInterface&, const std::string&) const override;
    private:
        virtual void receiveMessage (const ColleagueInterface*, const std::string&) const override;
};
```

```
class MediatorInterface {
    private:
        std::list<ColleagueInterface*> colleagueList;
    public:
        const std::list<ColleagueInterface*>& getColleagueList() const {return colleagueList;}
        virtual void distributeMessage (const ColleagueInterface*, const std::string&) const = 0;
        virtual void registerColleague (ColleagueInterface* colleague) {colleagueList.emplace_back (colleague);}
};

class Mediator : public MediatorInterface {
    virtual void distributeMessage (const ColleagueInterface*, const std::string&) const override;
};
```

Viselkedési minta II - Mediator

```
void Colleague::sendMessage (const MediatorInterface& mediator, const std::string& message) const {
    mediator.distributeMessage (this, message);
}

void Colleague::receiveMessage (const ColleagueInterface* sender, const std::string& message) const {
    std::cout << getName() << " received the message from " << sender->getName() << ": " << message << std::endl;
}

void Mediator::distributeMessage (const ColleagueInterface* sender, const std::string& message) const {
    for (const ColleagueInterface* x : getColleagueList())
        if (x != sender)  // Do not send the message back to the sender
            x->receiveMessage (sender, message);
}
```

Viselkedési minta II - Mediator

```
int main() {
    Colleague *bob = new Colleague ("Bob"),  *sam = new Colleague ("Sam"),  *frank = new Colleague ("Frank"),  *tom = new Colleague ("Tom");
    Colleague* staff[] = {bob, sam, frank, tom};
    Mediator mediatorStaff, mediatorSamsBuddies;
    for (Colleague* x : staff)
        mediatorStaff.registerColleague(x);
    bob->sendMessage (mediatorStaff, "I'm quitting this job!");
    mediatorSamsBuddies.registerColleague (frank);  mediatorSamsBuddies.registerColleague (tom);  // Sam's buddies only
    sam->sendMessage (mediatorSamsBuddies, "Hooray!  He's gone!  Let's go for a drink, guys!");
    return 0;
}
```

Teljesen mindegy, hogy a kommunikációra *mediatorStaff*, vagy *mediatorSamsBuddies* objektumot használjuk - mindkettő egy mediator, ami a hozzá regisztrált kollégákhoz eljuttatja az üzenetet.

Viselkedési minta III - Observer

Alap probléma: valamilyen eseményről a program több részének értesülnie kell. Ahelyett, hogy ezt behuzaloznánk, célszerű valamilyen publish-subscribe modellt követni.

OOP-ben a lényeg, hogy az Observer és az Observable (subject) entitásokat is objektumokon keresztül reprezentáljuk.

Sok minden megérthető ezen a példán keresztül. Előbb nézzük a main függvényt!

(Itt az alapelv, hogy ha wdata változik, az observerek kapjanak róla értesítést. Ezt nyilván úgy lehet elérni, ha az observerek wdata-ba “beregisztrálásra kerülnek”. Ezt erősíti meg a wdata->removeOb… sor is)

Viselkedési minta III - Observer

```
int main(int argc, char *argv[])
{

    ParaWeatherData * wdata = new ParaWeatherData;
    CurrentConditionBoard* currentB = new
CurrentConditionBoard(*wdata);
    StatisticBoard* statisticB = new StatisticBoard(*wdata);

    wdata->SensorDataChange(10.2, 28.2, 1001);
    wdata->SensorDataChange(12, 30.12, 1003);
    wdata->SensorDataChange(10.2, 26, 806);
    wdata->SensorDataChange(10.3, 35.9, 900);

    wdata->removeOb(currentB);

    wdata->SensorDataChange(100, 40, 1900);
    delete statisticB;
    delete currentB;
    delete wdata;

    return 0;
}
```

Viselkedési minta III - Observer

```
// The Abstract Observer
class ObserverBoardInterface
{
public:
    virtual void update(float
a,float b,float c) = 0;
};
```

```
// Abstract Interface for
Displays
class DisplayBoardInterface
{
public:
    virtual void show() = 0;
};
```

```
// The Abstract Subject
class WeatherDataInterface
{
public:
    virtual void registerOb(ObserverBoardInterface* ob) = 0;
    virtual void removeOb(ObserverBoardInterface* ob) = 0;
    virtual void notifyOb() = 0;
};
```

Viselkedési minta III - Observer

```
// The Concrete Subject
class ParaWeatherData: public WeatherDataInterface
{
public:
    void SensorDataChange(float a,float b,float c)
    {
        m_humidity = a;
        m_temperature = b;
        m_pressure = c;
        notifyOb();
    }

    void registerOb(ObserverBoardInterface* ob)
    {
        m_obs.push_back(ob);
    }
    void removeOb(ObserverBoardInterface* ob)
    {
        m_obs.remove(ob);
    }
protected:
    void notifyOb()
    {
        list<ObserverBoardInterface*>::iterator pos = m_obs.begin();
        while (pos != m_obs.end())
        {
            ((ObserverBoardInterface* )(*pos))-
>update(m_humidity,m_temperature,m_pressure);
            (dynamic_cast<DisplayBoardInterface*>(*pos))->show();
            ++pos;
        }
    }
private:
    float         m_humidity;
    float         m_temperature;
    float         m_pressure;
    list<ObserverBoardInterface* > m_obs;
};
```

Viselkedési minta III - Observer

Maguk a display board-ok egyszerre observerek is (értesülnek egy subject változásairól), és a DisplayBoardInterface-t is megvalósítják

```
// A Concrete Observer
class CurrentConditionBoard : public ObserverBoardInterface, public
DisplayBoardInterface
{
public:
    CurrentConditionBoard(ParaWeatherData& a):m_data(a)
    {
        m_data.registerOb(this);
    }
    void show()
    {
        cout<<"_____CurrentConditionBoard_____"<<endl;
        cout<<"humidity: "<<m_h<<endl;
        cout<<"temperature: "<<m_t<<endl;
        cout<<"pressure: "<<m_p<<endl;
        cout<<"_____"<<endl;
    }
```

Viselkedési minta III - Observer

Maguk a display board-ok egyszerre observerek is (értesülnek egy subject változásairól), és a DisplayBoardInterface-t is megvalósítják

```
    void update(float h, float t, float p)
    {
        m_h = h;
        m_t = t;
        m_p = p;
    }

private:
    float m_h;
    float m_t;
    float m_p;
    ParaWeatherData& m_data;
};
```

```
// A Concrete Observer
class StatisticBoard : public ObserverBoardInterface, public DisplayBoardInterface
{
public:
    StatisticBoard(ParaWeatherData&
a):m_maxt(-1000),m_mint(1000),m_avet(0),m_count(0),m_data(a)
    {
        m_data.registerOb(this);
    }

    void show()
    {
        cout<<"________StatisticBoard_________"<<endl;
        cout<<"lowest  temperature: "<<m_mint<<endl;
        cout<<"highest temperature: "<<m_maxt<<endl;
        cout<<"average temperature: "<<m_avet<<endl;
        cout<<"_____________________________"<<endl;
    }
```

Viselkedési minta III - Observer

```
    void update(float h, float t, float p)
    {
        ++m_count;
        if (t>m_maxt)
        {
            m_maxt = t;
        }
        if (t<m_mint)
        {
            m_mint = t;
        }
        m_avet = (m_avet * (m_count-1) + t)/m_count;
    }

private:
    float m_maxt;
    float  m_mint;
    float m_avet;
    int m_count;
    ParaWeatherData& m_data;
};
```

Kritikák a design patternekkel szemben

Gyakran előforduló kritikák:

- A Design Patternek bekasznizzák a gondolkodásunkat
- A Design Patternek valójában nem általánosak, hiszen nyelvfüggő is, hogy mi oldható meg elegánsan és kényelmesen
- A Design Patternek erőltetettek és túl sok kódot eredményeznek
  - Való igaz, hogy nem kell minden alkalmazáshoz absztrakt osztály, absztrakt interfész stb.
  - De: ezek minták! soha senki nem gondolta, hogy egy-az-egyben át kell őket venni.

Miért érdemes mégis a Design Patterneket ismerni

Sokat lehet belőlük tanulni

> Ráébresztenek bennünket arra, hogy milyen vissza-visszatérő kihívások merülhetnek fel összetett rendszerekben.
>
> Vannak alapelvek, amiket jól meg lehet a design patterneken keresztül érteni.
>
> Nem mellesleg: láthatjuk, hogy absztrakt osztályokkal + örökléssel milyen sokféleképpen tehetjük rugalmasabbá a rendszerarchitektúráinkat.
>
> A design patterneknek van neve, ezért új nyelv megismerésekor gyorsan el tudunk igazodni.

Még ha némelyik DP erőltetettnek is tűnik, örökzöld problémákra nyújtanak mintaszerű megoldást. Érdemes ezekből kiindulva eljutni a saját megoldásainkhoz.