OOP – 8. előadás

Design Patternek C++-ban

Előadó: Dr. Csapó Ádám

Mi az, hogy design pattern?

- Olyan absztrakció, ami mintázatot ad egy rendszer struktúrájának felépítéséhez
- A híres “Gang of Four” nyomán alakult ki ez a kifejezés (Gamma, Helm, Johnson, Vlissides) – akik 1994-ben publikálták a “Design Patterns: Elements of Reusable Object-Oriented Software” c. könyvüket
  - Nagy hatású könyv volt, 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 objektumokat (akkor érdekes, ha pl. futásidőben dől el, hogy pontosan milyen típusú objektumot szeretnénk; vagy ha több lépcsőben kell szabályozni egy példány inicializációját és ezt káoszos lenne 1 konstruktorral megtenni)
- ***Structural patterns (strukturális minták)*** – entitások funkcióinak bővítése, hogy más entitásokkal kompatibilisek legyenek, vagy azokban foglalt funkciókat (is) megvalósítsák
- ***Behavioral patterns (viselkedési minták)*** – objektumok közötti kommunikáció hatékony megvalósításai, akár indirekt úton is!

Kreációs minta 1: Builder

- ***Probléma***: összetett objektumot szeretnénk létrehozni, de nem akarjuk, hogy ehhez összetett (sok-argumentumos) konstruktort, vagy több inicializáló függvényt kelljen meghívni
- ***Megoldás***: köztes (Builder) objektum létrehozása, mely kiterjeszthető interfészt ad a létrehozandó objektum különböző részeinek inicializálásához
- ***Példa***: egy pizzának többféle tésztája, feltéte lehet. Ahelyett, hogy mindezen infokat a Pizza osztály konstruktorának kellene betáplálni, hozzunk létre:
  - A Pizza osztályon belül mindenféle settert (setDough(), setTopping(), …)
  - Egy PizzaBuilder absztrakt osztályt, mely kiajánl egy megfelelő interfészt (getPizza(), buildDough(), buildSauce(), buildTopping())!
  - Ebből származtatva létre lehet hozni a HawaiianPizzaBuilder, SpicyPizzaBuilder, … stb. Osztályokat, melyek mind egy speciális Pizza objektumot hoznak létre és setterekkel beállítják a megfelelő értékeket.
  - A PizzaMaker::makePizza(PizzaBuilder*) metódusa pedig végig hívogatja a megfelelő build… metódusokat.

Kreációs minta 2: Factory

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

- ***Megoldás***: Készítsünk egy Factory osztályt, melynek New(const std::string&) statikus metódusa létrehoz egy konkrét példányt. Mivel a New metódus csak egy típusra hivatkozó pointert tud visszaadni, ez a típus legyen egy absztrakt ősosztálya a lehetséges típusoknak

- ***Példa***: Computer absztrakt ősosztály. Ebből származik a Laptop, Desktop. A ComputerFactory osztály statikus NewComputer(const std::string&) metódusa pedig egy Computer*-t ad vissza, mely cím egy new-val létrehozott Laptop vagy Desktop típusú változó.
  - Fontos, hogy mivel ezeket a pointereket new-val hozzuk létre, egyszer majd még delete-et is kell hívni rájuk. Ez a hívó fél felelőssége!

Builder és Factory összehasonlítása

- A Builder esetben a Builder osztályból volt többféle változat (HawaiianPizzaBuilder, SpicyPizzaBuilder), melyek ugyanazon Pizza osztály settereit hívogatták.

- A Factory esetben viszont egyetlen Factory osztály volt, viszont annak egy (static) metódusa az argumentuma függvényében különböző típusú objektumokat hozott létre. Annak érdekében, hogy ezek címét visszaadhassa, a típusoknak közös (absztrakt) őse kellett, hogy legyen.

- A két módszer persze részben átalakítható egymásba, pl. lehetett volna úgy is csinálni, hogy a Pizza osztályból örököl a HawaiianPizza osztály meg a SpicyPizza osztály, majd a PizzaFactory osztály NewPizza() metódusa egy Pizza* címet ad vissza. Persze ezt a címet a hívó félnek majd egyszer fel is kell szabadítania, ha már nem használja.

Kreációs minta 3: Singleton

- ***Probléma***: Garantálni szeretnénk, hogy adott osztálynak csak egyetlen példánya létezzen az egész alkalmazásban. Tipikusan a menedzser osztályok ilyenek (adatbázis menedzser, hozzáférés-menedzser, …)
- ***Megoldás***:
  - A SingletonX osztályban a konstruktor privát, a copy constructor és copy assignment pedig le vannak tiltva – ezzel garantáljuk, hogy az osztály csak belülről példányosítható
  - A SingletonX osztálynak van egy statikus SingletonX típusú, instance nevű változója
  - A szintén statikus, de publikus SingletonX::GetInstance() metódus egy SingletonX& referenciát ad vissza, mely erre a statikus változóra hivatkozik.
- Vegyük észre, hogy kívülről csak a GetInstance() lesz elérhető. Ez azért statikus, hogy úgy is meghívható legyen, hogy az osztályt nem példányosítjuk (az osztályhoz tartozik, nem a példányhoz) – hiszen példányosítani nem is tudnánk! Ugyanígy az instance változó is pont azért statikus, hogy az osztályhoz tartozzon, ne annak egy példányához.

Kreációs minta 3: Singleton

```
class IntegerSingleton {
  int value;
  IntegerSingleton() : value() {}
  IntegerSingleton(const IntegerSingleton& other) = delete;
  IntegerSingleton& operator=(const IntegerSingleton& other) = delete;
  static IntegerSingleton* instance;
public:
  void setValue(int v) { value = v; }
  static IntegerSingleton& GetInstance() {
    // static IntegerSingleton* instance = new IntegerSingleton; // ezzel 1 sor megsporolhato
    return *instance;
  }
};

IntegerSingleton* IntegerSingleton::instance = new IntegerSingleton(0);
```

Kreációs minta 3: Singleton

```
int main() {
  //IntegerSingleton is1 = IntegerSingleton::GetInstance(); // nincs copy constructor…
  IntegerSingleton::GetInstance().setValue(5);
  IntegerSingleton::GetInstance().print();
  IntegerSingleton::GetInstance().print();
  IntegerSingleton::GetInstance().setValue(6);
  IntegerSingleton::GetInstance().print();
  std::cout << "address of singleton: " << &IntegerSingleton::GetInstance() << std::endl;
  std::cout << "address of singleton: " << &IntegerSingleton::GetInstance() << std::endl;
}
```

Strukturális minta 1: Adapter

- ***Probléma***: A szoftver egyik komponense egy adott interfészt biztosító objektumokkal működik, de mi egy másmilyen intefészt biztosító objektummal is használni szeretnénk
- ***Megoldás***:
  - Ha egy function(InterfaceTypeA*) függvényt szeretnénk használni egy InterfaceTypeB interfészt megvalósító osztállyal, akkor annyi a teendőnk, hogy egy Adapter osztályt készítünk az InterfaceTypeB-t megvalósító típushoz.
  - Ennek az Adapter osztálynak a konstruktora vár egy InterfaceTypeB*-t, amit az osztály eltárol.
  - E mellett az Adapter osztály megvalósítja az InterfaceTypeA-t is (pl. származik belőle, vagy csak ugyanolyan szignatúrájú metódusokat tartalmaz). Ezen metódusok megvalósításai az eltárolt InterfaceTypeB* címen levő objektum dolgait hívogathatják…

Adapter - példa

```
class Printable {
public:
    virtual void print() = 0;
};

class Person : public Printable {
public:
    void print() { std::cout << "I am a person" << std::endl; }
};

class Animal {
public:
    void saySomething() { std::cout << "Hi I am an animal!" << std::endl; }
};

void printSomething(Printable* prp) {
    prp->print();
}
```

Adapter - példa

```
class AnimalPrintableAdapter : public Printable {
    Animal* animal;
public:
    AnimalPrintableAdapter(Animal* ap) : animal(ap) {}
    void print() { animal->saySomething(); }
};

int main()
{
    Person p;
    printSomething(&p);
    Animal a;
    // printSomething(&a); // animal nem egyfajta Printable
    AnimalPrintableAdapter apa(&a);
    printSomething(&apa);
}
```

Strukturális minta 2: Decorator

- ***Probléma***: Dinamikusan szeretnénk több osztályhoz további funkciót / viselkedést hozzácsatolni, ahelyett, hogy egyenként kiterjesztjük azokat.
  - lehet, hogy egyazon funkciót rengeteg osztályhoz hozzá akarjuk csatolni -> nyögvenyelős lenne 25 különböző osztályból származtatni csak az adott funkcióért!
- ***Megoldás***:
  - Az egész működés előfeltétele, hogy a “dekorálandó” osztályoknak legyen egy közös public őse
  - Ebből a közös ősből származik szintén egy új **Decorator** osztály, amely az ős bizonyos virtuális metódusait pure virtuálissá alakítja át (surprise: ilyet lehet csinálni)
  - A Decoratorból származó konkrét DecoratorX, DecoratorY stb. osztályok pedig ezeket a metódusokat újradefiniálhatják. A Decorator osztályok emellett tárolnak egy hivatkozást egy dekorálandó példányra is.
  - Mivel a dekorálandó és Decorator osztályoknak is van közös őse, gyakorlatilag felhasználhatóak ugyanazokon a helyeken. Egy DecoratorX típusú objektum pl. használható egy dekorált típus helyett, miközben maga is tartalmaz önmagában egy ilyen típusú objektumra hivatkozást!

Decorator - példa

```
class Car {
    std::string description;
public:
    Car(const std::string& desc) : description(desc) {}
    virtual void getDescription() { std::cout << description << std::endl; }
};

class AudiA6 : public Car {
public:
    AudiA6() : Car("Audi A6") {}
};

class RenaultClio : public Car {
public:
    RenaultClio() : Car("RenaultClio") {}
};
```

Decorator - példa

```
class AbstractCarDecorator : public Car {
protected:
    Car* mycar;
public:
    AbstractCarDecorator(Car* cp) : Car(""), mycar(cp) {}
    void getDescription() = 0;
};

class CarWithSoundSystem : public AbstractCarDecorator {
public:
    CarWithSoundSystem(Car* cp) : AbstractCarDecorator(cp) {}
    void getDescription() {
        mycar->getDescription();
        std::cout << "\tThis car also has a sound system!" << std::endl;
    }
};
```

Decorator - példa

```
class CarWithAC : public AbstractCarDecorator {
public:
    CarWithAC(Car* cp) : AbstractCarDecorator(cp) {}
    void getDescription() {
        mycar->getDescription();
        std::cout << "\tThis car also has an air conditioner!" << std::endl;
    }
};
```

Decorator - példa

```
int main()
{
    AudiA6 audi1;
    AudiA6 audi2proto0;
    CarWithAC audi2(&audi2proto0);

    RenaultClio rc1, rc2proto0;
    CarWithAC rc2proto1(&rc2proto0);
    CarWithSoundSystem rc2(&rc2proto1);

    audi1.getDescription();
    audi2.getDescription();
    rc1.getDescription();
    rc2.getDescription();

    // itt mindossze arra kell figyelni, hogy audi2proto0 meg rcproto0 es rcproto1
    // ne tunjenek el a memoriabol meg azelott, hogy audi2 es rc2 eltunnenek!
```

Decorator - példa

```
    // Pl.:
    std::cout << "Problem:" << std::endl;
    AudiA6* audi3proto0 = new AudiA6;
    CarWithAC audi3(audi3proto0);
    audi3.getDescription();
    delete audi3proto0;
    audi3.getDescription(); // jo esetben crash!

    // Erre nyujthat megoldast a smart pointerek hasznalata, mint std::unique_ptr
    // Ez tulmutat a targy keretein, de az a lenyege, hogy ha audi3proto0 egy
    // std::unique_ptr lenne, akkor az audi3 letrehozasakor "atruhazhato" lenne
    // ennek a pointernek a birtoklasa az audi3 reszere ->
    // innentol audi3 erteke nullptr es csak CarWithAC destruktora tudna felszabaditani.
https://stackoverflow.com/questions/26318506/transferring-the-ownership-of-object-from-one-unique-ptr-to-another-unique-ptr-i
}
```

Viselkedési minta 1: 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.
- ***Probléma***: bizonyos funkcionalitások lefutását követően további műveleteket szeretnénk dinamikusan végrehajtani
- ***Megoldás***: reprezentáljuk magát a kérést egy objektummal, melyet átadhatunk a különböző függvényeknek, és amely alapján el tudják dönteni, mi legyen a következő művelet

Viselkedési minta 1: Command

- Konkrétabban:
  - A Command osztályunk egy absztrakt osztály, melyből további, specializáltabb parancs-típusok származhatnak. Legegyszerűbb esetben a Command osztálynak van egy execute() pure virtual metódusa.
  - Ezen parancs-típusok példányaira a hivatkozásokat (pointereket, referenciákat) ezután szabadon lehet passzolgatni a függvényhívások között. Mindez egyfajta callback-mechanizmust tesz lehetővé, + a hívások nyomonkövetésére is alkalmas.

```
class Command {
public:
    virtual void execute() = 0;
};

class LightUpCommandFake : public Command {
public:
    void execute() { std::cout << "Lampa felkapcsol" << std::endl; }
};
```

Viselkedési minta 1: Command

```
class LightDownCommandFake : public Command {
public:
    void execute() { std::cout << "Lampa lekapcsol" << std::endl; }
};

class LightUpCommandReal : public Command {
public:
    void execute() { std::cout << "Lampa felkapcsol, nemcsak kiir" <<
std::endl; }
};

class LightDownCommandReal : public Command {
public:
    void execute() { std::cout << "Lampa lekapcsol, nemcsak kiir" << std::endl;
}
};
```

Viselkedési minta 1: Command

```
class LightSwitch {
    Command& upC;
    Command& downC;
public:
    LightSwitch(Command& up, Command& down) : upC(up), downC(down) {}
    void turnUp() { upC.execute(); }
    void turnDown() { downC.execute(); }
};

int main()
{
    LightUpCommandFake lucf; LightDownCommandFake ldcf;
    LightUpCommandReal lucr; LightDownCommandReal ldcr;

    LightSwitch fs(lucf, ldcf); LightSwitch rs(lucr, ldcr);
    fs.turnUp(); fs.turnDown(); rs.turnUp(); rs.turnDown();
}
```

Viselkedési minta 2: Mediator

- ***Probléma***: üzeneteket szeretnénk eljuttatni adott típusú objektumokhoz, de mindig azok eltérő csoportjaihoz (részhalmazokat szeretnénk képezni belőlük, és ezen részhalmazoknak elküldeni)
- ***Megoldás***: hozzunk létre egy Mediator típust, ami enkapszulálja, hogy mely egyedek tartoznak hozzá, és azoknak juttatja el az üzenetet

```
template <class T>
class Mediator {
    std::vector<T*> entities;
public:
    void addEntity(T* ep) { entities.push_back(ep); }
    void distributeMessage(T* sender, const std::string& msg) {
        for (auto elem : entities) {
            if (elem != sender) {
                elem->receiveMsg(sender, msg);
            }
        }
    }
};
```

Viselkedési minta 2: Mediator

```
class Colleague {
    std::string name;
public:
    Colleague(const std::string& nm) : name(nm) {}
    void sendMsg(Mediator<Colleague>* mp, const std::string& msg) {
        mp->distributeMessage(this, msg);
    }
    void receiveMsg(Colleague* cp, const std::string& msg) {
        std::cout << name << " received message from " << cp->name;
        std::cout << std::endl << "\t" << msg << std::endl;
    }
};
```

Viselkedési minta 2: Mediator

```
int main()
{
    Colleague adam("Szabo Adam");
    Colleague eva("Kuti Eva");
    Colleague utalatos("Hernyo Guszti");

    Mediator<Colleague> baratok;
    baratok.addEntity(&adam);
    baratok.addEntity(&eva);

    Mediator<Colleague> mindenki;
    mindenki.addEntity(&adam);
    mindenki.addEntity(&eva);
    mindenki.addEntity(&utalatos);

    adam.sendMsg(&mindenki, "Talalkozzunk holnap 3-kor");
    eva.sendMsg(&baratok, "OK, de Gusztit meg ne hivd ebedre!");
}
```

Viselkedési minta 3: Observer

- ***Probléma***: valamilyen eseményekről értesülnie kell a program más részének, de a kommunikációt nem direktben szeretnénk összehuzalozni
- ***Megoldás***: létrehozhatunk egy Observable interfészt, amin keresztül az Observerek beregisztrálhatják magukat. Ezek után az observerek mindegyike értesülni fog az adott eseményről
- Nagyon sok szempontból hasonlít a Mediator-hoz, mivel a Mediator esetében is “beregisztráltunk” valamit, ami aztán üzeneteket kapott.
- Persze a kettő nem teljesen ugyanaz, mert:
  - Mediator minta esetében maga az entitás kezdeményezte az üzenetküldést, Observer esetében az observable objektum kezdeményezi saját maga
  - Az observerek elvben egymástól nagyon különböző típusok is lehetnek, csak az a lényeg, hogy mindegyik megvalósítsa az Observer interfészt (jóóó, a Mediator esetben is lehetett volna T egy absztrakt ősosztály típus…)

Viselkedési minta 3: Observer

```
// ez az osztaly azert kell, hogy az Observable osztalyban
// el tudjuk tarolni, valamint hogy interfeszt adjon a kommunikaciohoz...
class Observer {
public:
    virtual void update(std::string&, double, int) = 0;
};

class Observable { // subject-nek is hivhato
protected:
    std::vector<Observer*> observers;
public:
    void registerObserver(Observer* ob) { observers.push_back(ob); }
    void deregisterObserver(Observer* ob) {
        observers.erase(
            std::remove(
                observers.begin(), observers.end(), ob
            ),
        observers.end());
    }
};
```

Viselkedési minta 3: Observer

```
class Environment : public Observable {
    std::string name;
public:
    Environment(const std::string& nm) : name(nm) {}
    void updateInfo(double temp, int humi) {
        for (auto elem : observers) {
            elem->update(name, temp, humi);
        }
    }
};

class WeatherDataObserverEng : public Observer {
public:
    void update(std::string& observableNm, double temp, int humi) {
        std::cout << "WeatherDataObserver for ";
        std::cout << observableNm << ": temperature is " << temp << " deg C - ";
        std::cout << humi << "% humidity" << std::endl;
    }
};
```

Viselkedési minta 3: Observer

```
class WeatherDataObserverHun : public Observer {
public:
    void update(std::string& observableNm, double temp, int humi) {
        std::cout << "WeatherDataObserver ";
        std::cout << observableNm << " kornyezetre: a homerseklet " << temp << " fok C - ";
        std::cout << humi << "% paratartalom" << std::endl;
    }
};

int main()
{
    Environment e1("nappali"); Environment e2("haloszoba"); Environment e3("jatszoter");

    Observer* wdoe = new WeatherDataObserverEng;
    Observer* wdoh = new WeatherDataObserverHun;

    e1.registerObserver(wdoe); e2.registerObserver(wdoe);
    e2.registerObserver(wdoh); e3.registerObserver(wdoh);
    e1.updateInfo(22, 50); e2.updateInfo(20.5, 48); e3.updateInfo(33, 81);

    e2.deregisterObserver(wdoe); e2.updateInfo(20.3, 49);
}
```

Kritikák a Design Patternekkel szemben

- 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 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.