Design patterns & best practices

Saját tapasztalatok alapján

CONST!!!

HASZNÁLJUK!!!

Példa

A const ha teheti mindig a tőle balra álló dologra vonatkozik.

```
const Person* addName(const char * const) const
```

Példa

A const ha teheti mindig a tőle balra álló dologra vonatkozik.

```
const Person* addName(const char * const) const
```

Nem módosíthatjuk a visszaadott objektumot

Példa

A const ha teheti mindig a tőle balra álló dologra vonatkozik.

```
const Person* addName(const char * const) const
```

Nem módosíthatjuk a pointer által mutatott területet

Példa

A const ha teheti mindig a tőle balra álló dologra vonatkozik.

```
const Person* addName(const char * const)
const
```

Nem módosíthatjuk magát a pointert sem.

Példa

A const ha teheti mindig a tőle balra álló dologra vonatkozik.

```
const Person* addName(const char * const)
const
```

Nem módosíthat az eljárás az objektumon (pont rossz a példa)

RAII

RAII

Érintőleg volt róla már szó

Resource **A**llocation **I**s **I**nitialization

C++ specifikus más nyelvben nem igazán van

RAII - mutex példa

```
class CriticalSection {
   Mutex& mutex;
public:
   CriticalSection(Mutex& mutex);
   virtual ~CriticalSection();
};
```

RAII - mutex példa

```
CriticalSection:: CriticalSection(
Mutex& mutex) : mutex(mutex)
{
  mutex.lock();
}
CriticalSection::~CriticalSection()
{
  mutex.unlock();
}
```

RAII - vector példa

```
class Vector {
   int* data;
public:
   Vector(size_t size);
   virtual ~Vector();
};
```

A dinamikus foglalás ellenőrzött módon történik. (Nincs * a kódban.)

RAII - vector példa

```
Vector::Vector(size_t size)
{
  data = new int[size];
}
Vector::~Vector()
{
  delete[] data;
}
```

Multiplatform megoldások

1. #define makrók

```
#ifdef __WIN32
  //...
#else
  //...
#endif
```

Egyszerű esetekben használható.

Bonyolultabb esetekben átláthatatlan lesz a kód.

Valahol nem is lehet így megoldani, mert a 2 rendszer logikája annyira különbözik.

Könnyű elrontani, pl az egyik ágban definiálunk egy változót, a másikban nem

2. Implementációs fájlok

Közös header: benne a class deklaráció közös.

Az implementáció már platform specifikus.

A project az egyik platformon az egyik fájlt fordítja le, a másikon a másikat.

Felesleges a külön interface. Elég a közös class deklaráció. Nagy egységek is szét lehetnek szedve így.

PI. SecureMem

```
//SecureMem.h

class SecureMem {
public:
   void* allocate(size_t size);
   void release(void* memory);
}
```

Pl. SecureMem

```
//win32/SecureMemWin32.cpp - Nem teljes példa!
#include <SecureMem.h>
void* SecureMem::allocate(size_t size) {
   PVOID memory = HeapAlloc(
        heap_handle,
        HEAP_ZERO_MEMORY, size);
   VirtualLock(memory, size);
   return memory;
}
void SecureMem::release(void* memory) {
   // ...
}
```

PI. SecureMem

```
//unix/SecureMemUnix.cpp
#include <SecureMem.h>

void* SecureMem::allocate(size_t size) {
   void* memory = malloc(size);
   mlock(memory, size);
   return memory;
}
void SecureMem::release(void* memory) {
   //...
}
```

Konstansok kezelése

Beégetett konstansok

- Egy helyen tárolva, pl. Constants.h
  - Ha bármit keresünk, tudjuk hol kell keresni
  - Egy szint felett átláthatatlan lesz
  - Nehéz kiszűrni, ha valamit már nem használunk
- Mindig a 'tulajdonos' osztályban
  - Sok konstans esetén is átláthatóbb
  - Ha nem egyértelmű, hogy mihez tartozik nehéz megtalálni

Konstansok konfig fájlból

- Lehet hozzá UI, hogy változtassuk
- Olcsóbb a program update-je, ha csak a konfig fájlt kell frissíteni
- Mi van, ha nincs ott a konfig fájl?

Konstansok szervertől

- Nagyon flexibilis.
- Általában szerverrel / hálózattal kapcsolatos paramétereket érdemes így tárolni
- Kellenek defaultok arra az esetre, ha nincs hálózat.

Pl. Reconnect timeout

Null object

Vagy default object

Létrehozunk egy statikus objektumot és ezt használjuk null objektumnak, vagy valamilyen defaultként.

Ha referenciával használjuk figyelni kell, hogy hogy nehogy módosítsuk, mert akkor minden példánya elromlik.

Példa

```
class Person {
public:
   static const Person* UNKNOWN;
   // ...
};

const Person::UNKNOWN = new
Person();
```

DRY - Don't Repeat Yourself

Minden rendszerelem rendelkezzen egy egyértelmű, megbízható forrással.

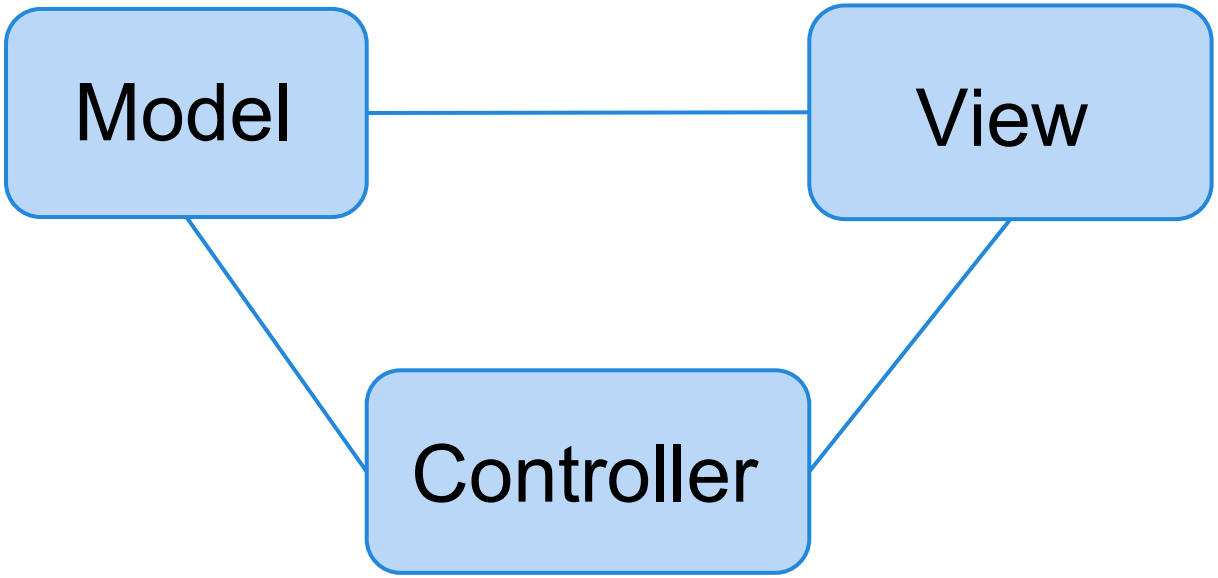
Nem csak a kódra vonatkozik.

Amit csak lehet emeljünk ki. (absztrakció, tervezési minták)

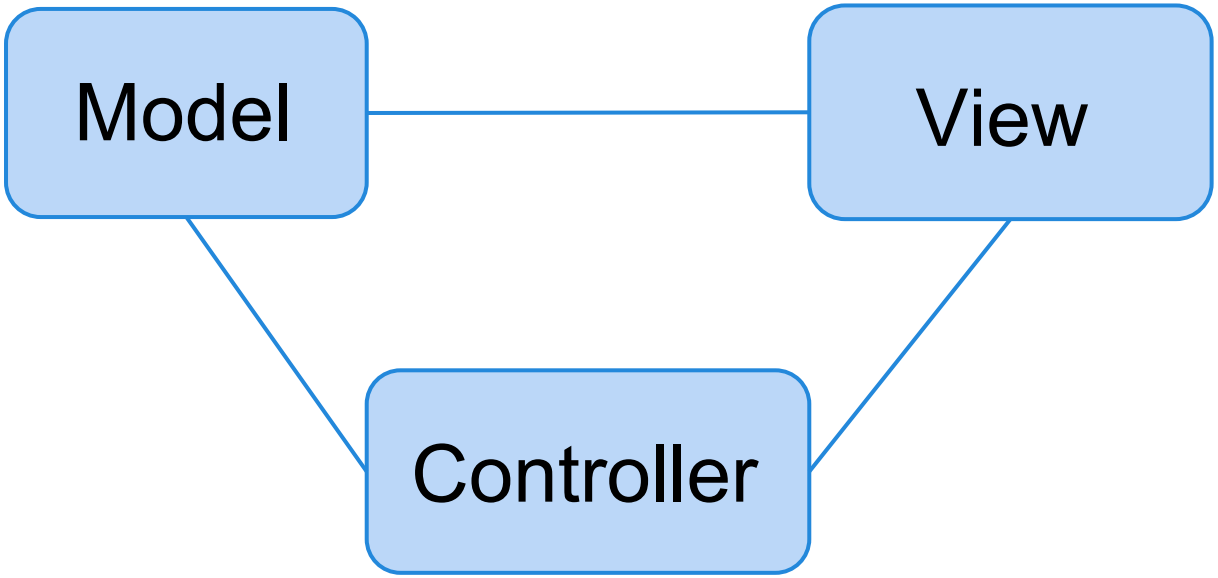
Amit csak lehet automatizáljunk (tesztelés, kód generálás)

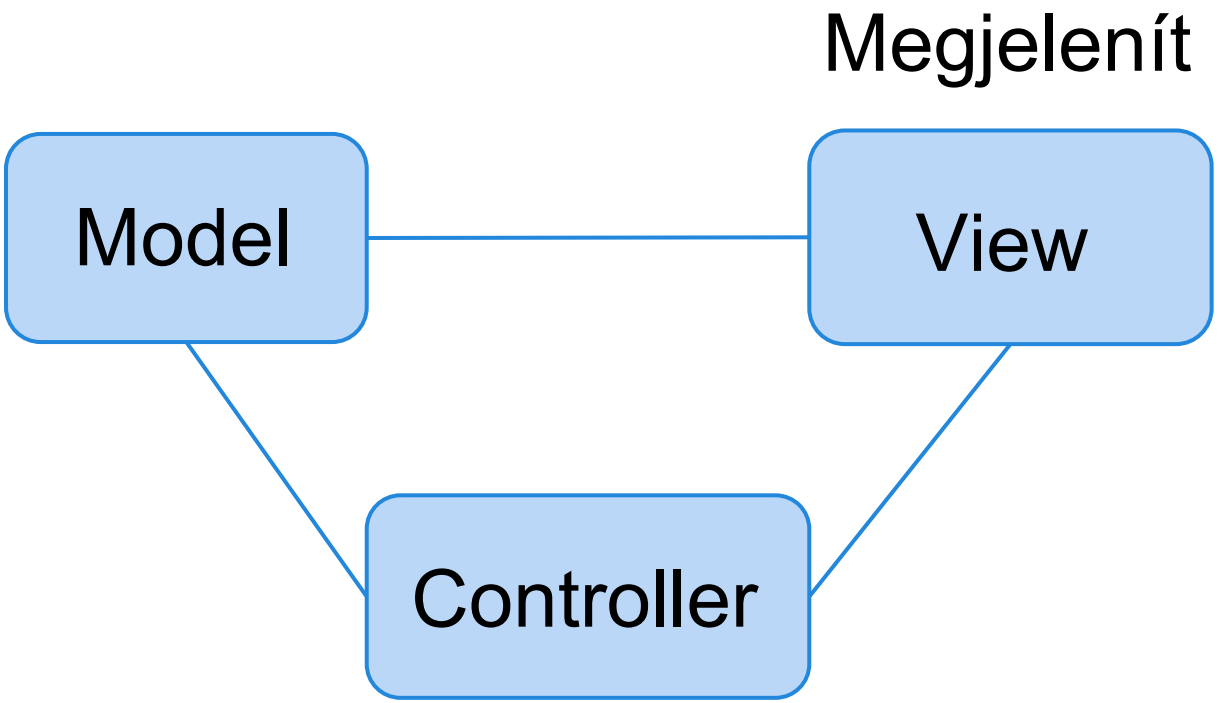
MVC

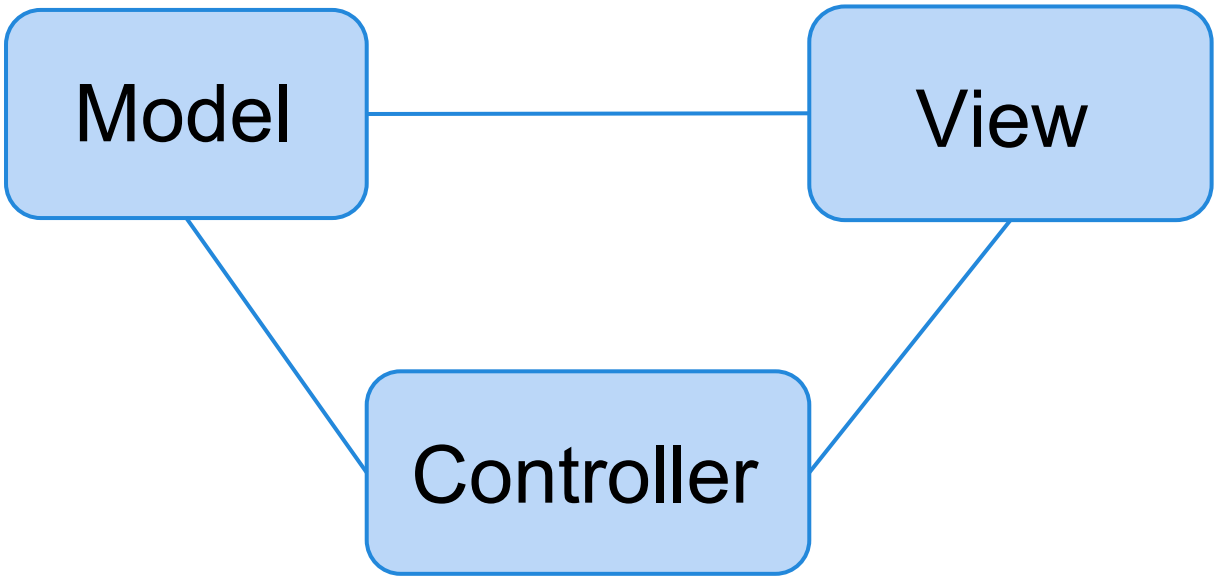
Mindenki tudja, hogy mi, mégis mindenki elrontja


Model
View
Controller

Adatokat tárol.
Műveletet végezhet rajtuk




Megjelenít
Model
View
Controller



Üzleti logika.

Amiket el lehet (és szokunk) rontani

- View-ba nem teszünk logikát. Nem hívunk be csak a saját controllerbe
- Egy másik controller függvényét csak a saját view-jából vagy egy másik controllerből hívjuk. (Esetleg modelből callbackkel)
- Nem csinálunk a controllerbe getView() függvényt, hiába nagy a kísértés. (Még akkor is, hogy ha a legtöbb fv. sima áthívás a view-ba)

Project layout - MVC

src/

- model/
- view/
- contoller/

Volt ilyen projectünk. Nem az igazi.

Project layout - logika szerint

src/

- network/
- stuff/
- ui/

Mindenhol csak a kontroller a 'publikus interface'

Tervezési minták

Tervezési minták

Csoportok: létrehozási, szerkezeti, viselkedési

Általában bonyolítják a programot (sok új osztály, interfész).

Mit jelent, h bonyolultabb? (sok if vs. sok osztály)

Mindig a megoldandó feladathoz kell méretezni.

Létrehozási minták

Létrehozási minták

Mikor az alkalmazásban ugyanazt a szerepet több osztály is betöltheti szükségünk lehet egy megoldásra, amivel egyszerűen tudunk váltani.

Általános megoldás:

1. Közös interfész a létrehozandó osztályoknak
2. A példányosítás "gyártó" osztályokba szervezése
3. A létrehozandó osztály példányosítása helyén a "gyártó" osztály használata

Factory method / Gyártófüggvény

Nem **`new`**-val hozunk létre egy objektumot, hanem valamilyen virtuális függvényen keresztül

Jó mert:

- Visszaadhat nullptr-t
- Elvégezhet plusz feladatokat a példányosítás előtt vagy után
- Választhat megfelelő alosztályt

Példa

```
class Application {
    CreateMainWindow() {
        AddButton(CreateButton());
    }
    IButton CreateButton() {
        return new Button();
    }
};
class ShinyApplication : Application {
    IButton CreateButton() {
        return new ShinyButton();
    }
}
```

Factory / Gyár

Az objektumok létrehozását függetleníthetjük a felhasználástól.

Az alkalmazáshoz szükséges objektumok létrehozását kiszervezzük egy külön osztályba.

Példa

```
class Application {
    CreateMainWindow(UIFactory& factory) {
        AddButton(factory.CreateButton());
    }
};
```

Példa

```
class UIFactory
{
    IButton CreateButton() {
        return new Button();
    }
};

class ShinyUIFactory : UIFactory {
    IButton CreateButton() {
        return new ShinyButton();
    }
};
```

Abstract Factory / Elvont gyár

Egy újabb réteg a gyárak felett.

Lehetővé teszi, hogy válasszunk a különböző objektum csoportok létrehozása között:

- platform (ShinyButtonForMac, ShinyButtonForWin)
- teszt (MockButton)

Singleton

Nagyon gyakran használt.

Egy statikus példány van az objektumból. El lehet kérni.

Lazy(késleltetett) inicializáció is szokott lenni -> Gyorsabb program indulás

Singleton

```
private:
   static ControllerFactory instance;
...

public:
   static ControllerFactory getInstance() {
      if (instance == NULL) {
         instance = new ControllerFactory();
      }
      return instance;
   }
```

Dependancy injection

Nem design pattern

El lehet vele kerülni a singletonokat.

Minden objektumnak átadjuk azokat az objektumokat, amikre szüksége lehet. (Kontrollert, modellt, mindent)

Dependancy injection

+: Flexibilis, nagy refaktorok is könnyebben végrehajthatók.

+: Elkerüli azokat a globális objektumok csapdáit

+: Könnyebben tesztelhető

-: RENGETEG függőség az objektumok között, akár. Nagyon kell figyelni a rendszer felállásakor és lebontásakor, hogy mindig minden létezzen.

-: Kis refaktornál macerás a sok paraméter

Builder / Építő

Még egy fokkal elvontabb szint

Használja az eddigieket.

Pl. egy ablakot kell csinálni

A builder ráteszi a bezáró gombot, ikont, keretet, fejlécet, címet ...

Példa

```
class Application {
    CreateDialog(const char* msg, UIFactory& factory)
    {
        AddLabel(factory.CreateLabel(msg))
        AddButton(factory.CreateButton("Yes"));
        AddButton(factory.CreateButton("No"));
    }
};
CreateDialog("Continue?");
CreateDialog("Abort?");
```

A CreateDialog függvény építi fel a factory segítségével az összetett objektumot.

Prototípus

Egyszer legyártjuk az objektumot, és utána ezt másolgatjuk.

Hasznos ha:

- Kevés dologban térnek el a példányok, viszont a semmiből létrehozni drága
- Nem akarunk mindenből más fajtát példányosítani (másik gyár), csak néhányból (prototípus gyár)

Szerkezeti minták

Szerkezeti minták

Az alkalmazásunk belső szerkezetének kialakítását segítik:

- biztosítják és/vagy egyszerűsítik a hozzáférést
- egységesítenek
- egyszerűsítik a bővítést

Adapter

Pl Van két különböző libünk, az egyik által előállított valamit kell a másikkal használni.

```
XLIB::Image image = XLIB::readImage
(fileName);

analyzeXLIBImage(image);
```

Adapter

```
void analyzeXLIBImage(image) {
   YLIB::image yimage = createYLIBImage(image);
   YLIB::analyze(yimage);
}

YLIB::image createYLIBImage(XLIB::image) {...}
```

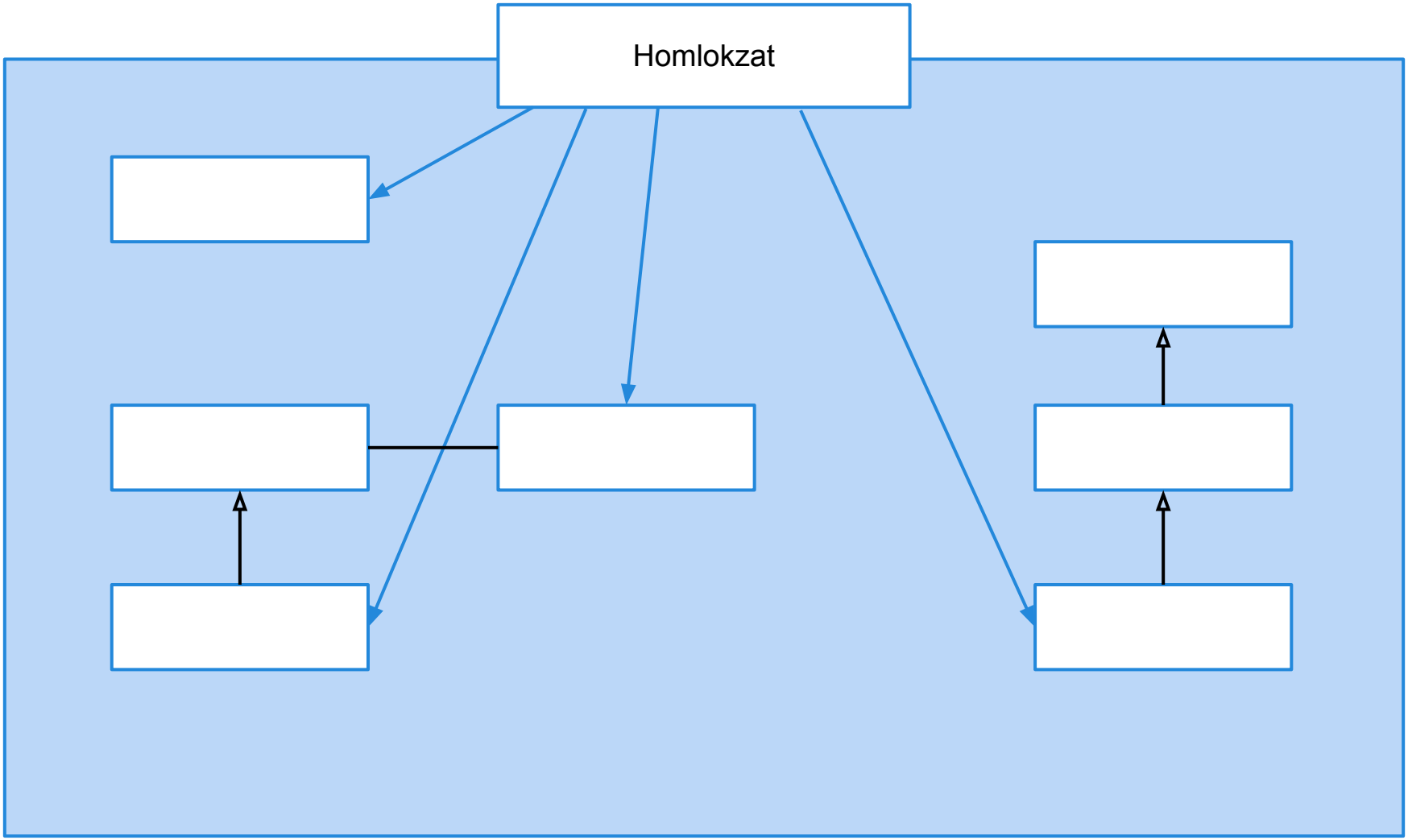
Facade / Homlokzat

Valamilyen nagy egység funkcionalitását akarjuk egy egyszerű, szűk interface-en keresztül mutatni.

Ezzel lehet elrejteni a "bonyolultságot".

A library-k előszeretettel használják.

Példa



Composite / Összetétel

Egységesen kezelhetjük az egyszerű és összetett elemeket. (közös interfész)

A GUIból ismerős lehet.

Általános működés:

- egyszerű elem végrehajt egy kérést
- összetett elem végrehajtja a kérést a gyerekein

Példa

```
class Widget {
public:
   void Add(Widget*);
   void Remove(Widget*);
   void Draw();
};
class Button : public Widget {
   void Draw() {
      // gomb kirajzolás
   }
};
```

Példa

```
class Panel : public Widget {
   void Add(Widget* w){ m_children.Add(w); }
   void Remove(Widget* w) {
      m_children.Remove(w);
   }
   void Draw() {
      for (auto w: m_children) { w->Draw(); }
   }
private:
   Container<Widget*> m_children;
};
```

Viselkedési minták

Viselkedési minták

Algoritmusoknak az egységbezárását valósítják meg.

Nem kell ismerünk a belső szerkezetet, elég ha az osztályok tudják, hogyan akarnak viselkedni.

Ez lehet:

- objektumok közötti kommunikáció
- egy objektum többfaja használata

Visitor / Látogató

Ritkán használt, a composite-tal van több értelme.

Példa:

Objektumok hierarchiáját szeretnénk több féle képpen szérializálni(xml, json, adatbázis)

Könnyen jöhet újfajta szérializálási mód később és ezzel nem szeretnénk sokat szívni.

Visitor minta

```
class SerializatorAccepter {
   virtual void acceptSerializator
(Serializator* s) = 0;
};
-------------------------------------
class Serializator {
   virtual void serializeMe(Car* car) = 0;
   virtual void serializeMe(Wheel* wheel)=0;
};
```

Példa

```
class Car : public SerializatorAccepter {
   void acceptSerializator(Serializator* s);
}

void Car::acceptSerializator(Serializator* s)
{
 s->serializeMe(this);
};
```

Példa

```
class Wheel : public SerializatorAccepter {
   void acceptSerializator(Serializator* s);
}

void Wheel::acceptSerializator(Serializator* s)
{
 s->serializeMe(this);
};
```

Observer / Megfigyelő

= Listenerek

Beregisztrálunk egy minket érdeklő eseményre (megfigyeljük - observe)

Callbacken keresztül jön az értesítés, hogy ha valami változott.

Observer / Megfigyelő

Ha nincsenek:

```
void Network::OnPeerDisconnect(Peer peer) {
    notificationController.peerDisconnected(peer);
    uiController.removePeer(peer);
    reconnectController.tryToReconnectTo(peer);
    fileTransferController.stopFileTranfer(peer);
    remoteControlController.stopRemoteControlFrom(peer);
}
```

Ha valami új komponens jön a rendszerbe még áttláthatatlanabb lesz

Observer / Megfigyelő

Observerrel:

```
void Network::OnPeerDisconnect(Peer peer) {
    for (auto listener : connectionListeners) {
        listener.onDisconnected(peer);
    }
}
```

Veszélye:

Mi van, ha valaminek előbb kell hívódnia, mint a valami másnak

Mediator / Közvetítő

Ha az alkalmazásunkban egy objektum megváltozása maga után von más objektumok megváltoztatását, ezt a viselkedést külön osztályba szervezhetjük.

Az objektumokból vezérlőbe kerül a logika:

- központosított, egyszerűbb a karbantartás.
- a vezérlő elbonyolodik

Megvalósítható a QT slot/signal párossal.

Iterator / Bejáró

Összetett objektumok elemeinek soros elérését teszi lehetővé.

Az STL-ből lehet ismerős.

```
class ListIterator {
   Element HasNext();
   Element Next();
};
```

Példa

```
class List
{
   ListIterator Begin();
   ListIterator End();
};

class ListIterator {
   ListIterator Next(); // operator++
   Element Value();     // operator*
};
```

Command / Parancs

Valamilyen utasítás reprezentálása objektumként.

Lehet user utasítása a UI-ról. (undo-redo -hoz)

Gyakoribb: Hálózaton keresztül bejövő hívás.

Általában érdemes eltenni egy sorba, hogy ne blokkoljuk azt a szálat, ami a hálózatot kezeli.

Egy feldolgozó szálban végrehajtjuk.

Chain of Resp. / Felelősséglánc

Egy kérelmet több objektum is kezelhet, de a kezelők között felállítható egy sorrend.

Ha egy kezelő nem tud mit kezdeni egy kéréssel továbbítja a következőnek.

Egyáltalán nem biztos, hogy történik bármi is!

Példa

```
class ClickHandler {
public:
   void HandleClick() {
      if (m_nextHandler == NULL) {
         m_nextHandler->HandleClick();
      }
   }
private:
   ClickHandler* m_nextHandler;
};
```

Példa

```
class Button : public ClickHandler {
   void HandleClick() {
      DoClickAction();
   }
};

class Label : public ClickHandler {
   void HandleClick() {
      ClickHandler::HandleClick();
   }
};
```

Példa

```
class Handler {
public:
   void HandleRequest(Request& request) {
      switch (request.GetType())
      {
         case Click:
            HandleClick(request);
            break;
      }
   }
};
```

Memento / Emlékeztető

Objektumok állapotát reprezentáljuk valamilyen könnyen tárolható módon.

Van egy 'mediátor' osztály, amit ezt kezeli

- Undo/Redo
- Újraindulás után vissza tudjuk állítani a program állapotát

Példa

```
class User {
public:
   JSONObject toJSON();
   void fromJSON(const JSONObject& state);
}
```

A JSON-t el lehet menteni adatbázisba, fájlba, vagy akár memóriába. A JSON-ből újra tudja magát építeni.

Template method / Sablonfüggvény

Az ősosztályban leírunk egy eljárást, de az eljárás által használt függvényeket (egy részét) már a leszármazottakban valósítjuk meg.

Pl. Egy ablak kirajzolása

Template method pl.

```
void UI::Window::draw() {
   drawBorder();
   createCloseButton();
   drawTitle();
   setupListeners();
}
```

Proxy / Helyettes

Egy objektum objektum által hivatkozott másik objektumot lehet egy proxyval helyettesíteni.

Ugyanabból az osztályból származik, mint az, amit helyettesít

Hívás esetén áthív a helyettesítettbe.

Proxy / Helyettes

- Késleltetett inicializáció
- Valamilyen műveletet még el tud végezni a helyettesítetten
- Szálak szinkronizálása a helyettesítetten

Forrás

- http://en.wikipedia.org/wiki/Don't_repeat_yourself
- Programtervezési minták (E. Gamma, R. Helm, R. Johnson, J. Vlissides)