1. Objektumelvű tervezés

Objektumelvű rendszer tervezése nehéz és bonyolult feladat.

Különösen újrafelhasználható terv esetén.

Újrafelhasználható terv:

- meg kell felelnie a konkrét probléma sajátosságainak;
- elég általánosnak is kell lennie ahhoz, hogy más, később felmerülő feladatok megoldása során is alkalmazható legyen;
- elkerülve, vagy minimalizálva, az esetleges újratervezést.

Jó tervek létrehozásához nagy gyakorlatra van szükség. Miért van az, hogy kezdő szakemberek rendszerint nem tudnak olyan jó, újrafelhasználható terveket készíteni, mint a nagy gyakorlattal rendelkezők?

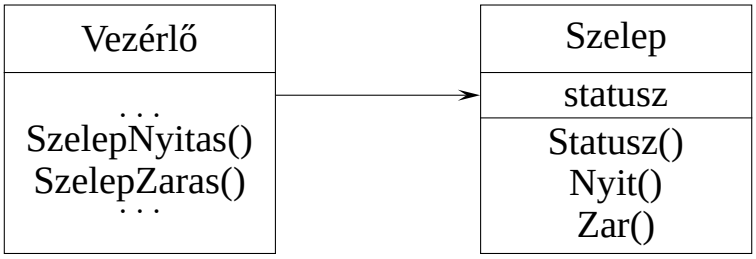
Ennek egyik legfontosabb oka, hogy a tapasztalt tervezők rendszerint bevált tervek alapján hozzák létre egy új rendszer tervét. Egy-egy jó tervet használnak újra meg újra, ezek ismerete és használata teszi őket jó szakemberekké.

Ezeket a terveket, tervrészeket nevezzük *tervmintáknak* (design patterns). Hasonló módon használhatóak a tervezésben, mint ahogy a programozás során a kód egyes részeit kódújrafelhasználás segítségével állítjuk elő.

Rossz tervek:

- „Mindenható osztály”: az osztály hajtja végre majdnem az összes teendőt, a többi osztálynak legfeljebb támogató műveleteket hagy. Az osztály lényegében egy bonyolult és összetett vezérlő osztály (gyakran ez is a neve), amelyet egyszerű osztályok vesznek körbe. Ezen osztályoknak csak adattárolási funkciójuk van (csak `get` és `set` műveletek).
- „Mindentudó osztály”: tulajdonképpen az előző osztály egy változata, csak ez nem az összes tevékenységet hajtja végre, hanem az összes adatot tartalmazza. Ekkor a többi osztály innen nyeri ki (ezen osztály `get` és `set` műveleteivel) a műveletekhez szükséges adatokat.

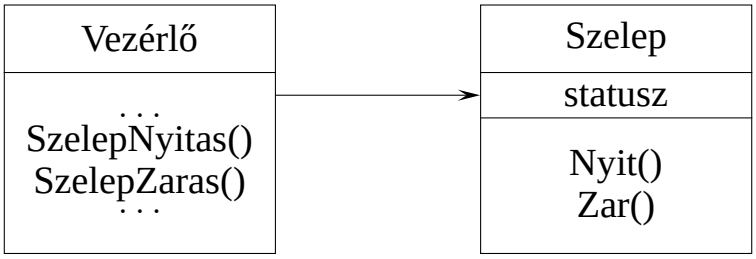
Példa „mindenható osztályra” az alábbi vezérlő:



Ekkor a szelep nyitása művelet:

```
void SzelepNyitas()
{
    int    allapot;
    allapot = sz->Statusz();
    if ( allapot != nyitva )   sz->Nyit();
}
```

Helyes megoldás:




```
void SzelepNyitas()
{
    sz->Nyit();
}

void Nyit()
{
    if ( statusz != nyitva )   statusz = nyitva;
    ...
}
```

2. Tervminta

Általában egy tervminta a következő négy alapvető elemből épül fel:

Név Ezzel hivatkozunk a tervezési feladatra és annak megoldására. Ez rendszerint egy-két szó, amely utal a funkcióra. Lehetővé teszi, hogy a tervezést magasabb absztrakciós szinten végezzük.

Feladat Itt le kell írni, hogy milyen esetekben alkalmazható a minta. A leírás tartalmazza a problémát, annak környezetét és az esetleges peremfeltételeket.

Megoldás A minta alkotóelemeinek, azok kapcsolatainak, együttműködésüknek a leírása. Ez egy absztrakt leírás, amely az általánosság érdekében nem tartalmazza az implementációt.

Következmények A minta eredményeinek és a meghozott kompromisszumoknak (futási idő, tárkapacitás) a leírása.

2.1. Tervminták osztályozása

Létrehozási: Az osztályok, objektumok (példányok) létrehozására, előállítására vonatkozó minták. Az osztályokra vonatkozó esetben az öröklődés felhasználásával érjük el, hogy különböző osztályt példányosítsunk. Az objektumok esetében a példány létrehozását egy másik objektumra delegáljuk.

Szerkezeti: Ezek a minták arra szolgálnak, hogy osztályokból vagy objektumokból nagyobb szerkezeteket hozzunk létre. A létrejött elemek rendszerint új funkcionalitással is rendelkeznek.

Viselkedési: Az objektumok közötti kapcsolatokkal, vezérlési folyamatokkal kapcsolatos minták. Használatukkal az objektumok kapcsolatára kell figyelni, a vezérlés folyama a mintában van. Az öröklődés felhasználásával érik el, hogy a viselkedést szétosszák különböző osztályok között.

2.2. Tervminták megadása

A következőkben összefoglaljuk, miként adhatunk meg egy-egy tervmintát. Az UML diagram ugyan fontos eleme ennek, de korántsem elégséges önmagában. A mintákat a megoldandó feladat alapján osztályokba soroljuk (pl.: létrehozási, szerkezeti, viselkedési), és ezt szintén fel kell tüntetnünk a meghatározás során.

1. *A minta neve és osztálya*: egy jó névnek utalnia kell a felhasználás területére, a megoldott feladatra.
2. *Cél*: rövid leírása annak, hogy mi a minta által megoldott feladat vagy feladatosztály.
3. *Más nevek*: további, mások által használt nevei a mintának, ha van ilyen.
4. *Motiváció*: egy esettanulmány, amely bemutatja a tervezési feladatot, és hogy miként oldja meg azt a minta a benne szereplő osztályok és objektumok segítségével.
5. *Felhasználhatóság*: annak leírása, hogy milyen esetekben lehet a mintát alkalmazni.
6. *Szerkezet*: itt kell megadni a megfelelő UML diagramot vagy diagramokat.
7. *Elemek*: a mintában előforduló osztályok, objektumok és szerepeik felsorolása.

8. *Együttműködés*: annak bemutatása, hogy a minta elemei miként működnek együtt a szerepük megvalósítása érdekében.
9. *Következmények*: itt kell választ adni azokra a kérdésekre, hogy miként éri el a minta a célját; milyen kompromisszumok árán; a rendszer szerkezetének milyen összetevőit lehet függetlenül változtatni.
10. *Implementáció*: itt kell megadni az implementációval kapcsolatos észrevételeket, megjegyzéseket, ajánlásokat, megszorításokat.
11. *Példa kód*: kódrészletek, amelyek bemutatják, miként lehet a minta egyes elemeit egy adott nyelven megvalósítani.
12. *Ismert használat*: példák a minta előfordulásaira működő rendszerekben.
13. *Rokon minták*: a hasonló minták nevei, a fontosabb különbségek felsorolása, illetve annak a megadása, hogy mely más mintákkal célszerű együtt használni.

A tervminta megadásának elemei közül néhány (10-13) értelemszerűen elmaradhat.

Az, hogy mely terveket tekintünk tervmintáknak, relatív fogalom. Az egyetlen követelmény, hogy az többször felhasználható legyen. Ezen belül egy adott fejlesztői közösség dönthet. Ugyanakkor léteznek jól bevált tervminták.

Figyelő

Név, osztály: figyelő (observer); viselkedési (behavioral).

Cél: Olyan egy-több függőség megadása objektumok között, amelyben ha egy objektum állapotot vált, akkor az összes tőle függő objektumot értesíteni és módosítani kell.

Más nevek: dependents, publish-subscribe.

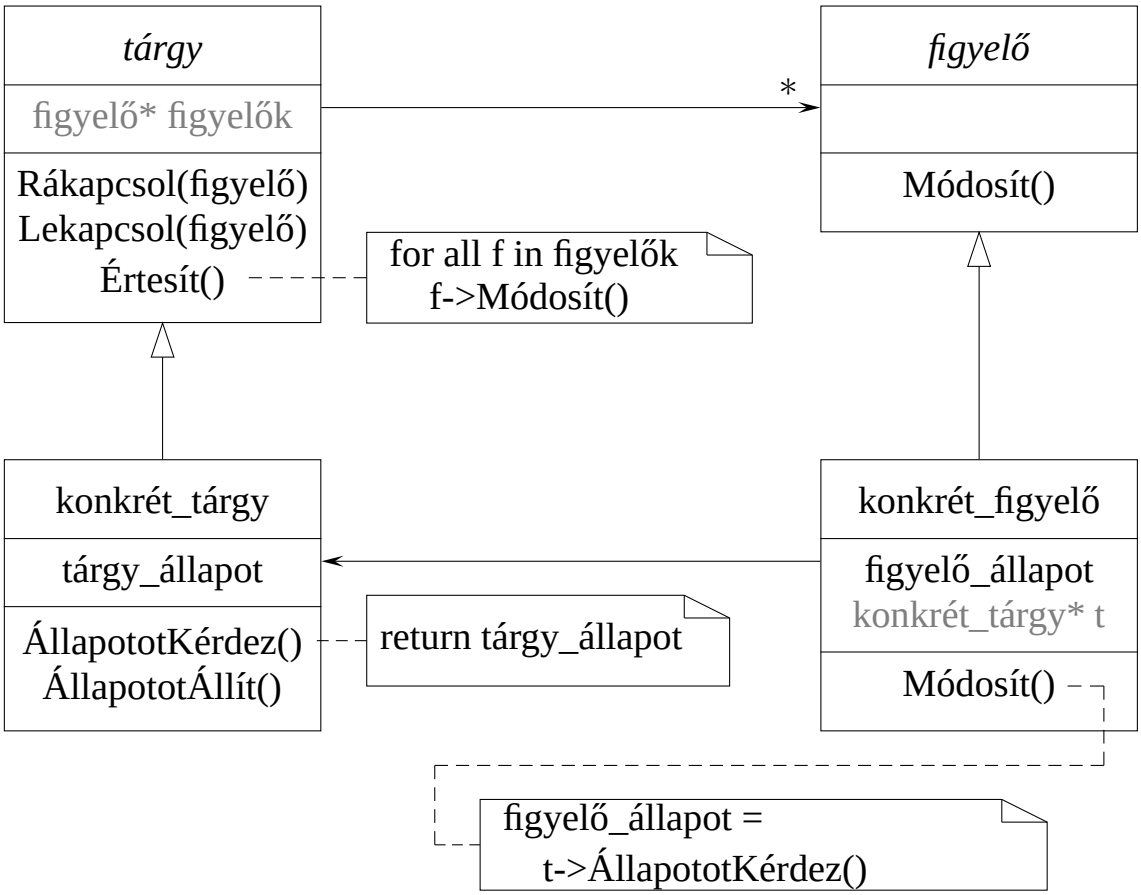
Motiváció: Egy gyakori mellékhatás, ami egy rendszer együttműködő osztályokra bontása során felmerül, a kapcsolatban álló objektumok közötti konzisztencia fenntartásának szükségessége.

Erre egy példa, amikor egy statisztika százalékos adatait akarjuk megjeleníteni táblázat, oszlopdiagram vagy kördiagram formájában. Egyidejűleg több formában is láthatók az adatok. Ekkor a statisztika a megjelenítés tárgya, a diagramoknak pedig mindig annak aktuális állapotát kell mutatniuk, anélkül hogy a diagramok egymásról tudnának. Ha bármelyiken változtatás történik, akkor az megjelenik a statisztikában, és az értesíti az összes diagramot. Ebben a példában a statisztika a *tárgy* (subject), a diagramok pedig a *figyelők* (observers).

Felhasználhatóság: A figyelő tervminta használható az alábbi esetekben:

- Ha egy absztrakcióban két olyan tényező szerepel, amelyek közül az egyik függ a másiktól. Ha ezeket külön objektumoknak tekintjük, akkor lehetséges egymástól független változtatásuk és újrafelhasználásuk.
- Amikor egy objektum állapotának megváltoztatása más objektumok változtatását teszi szükségessé, és nem ismert ezen objektumok száma.
- Amikor egy objektumnak üzenetet kell küldenie más objektumoknak, amelyekről nem tehetünk fel semmit; azaz nem akarjuk szorosan összekapcsolni ezeket az objektumokat.

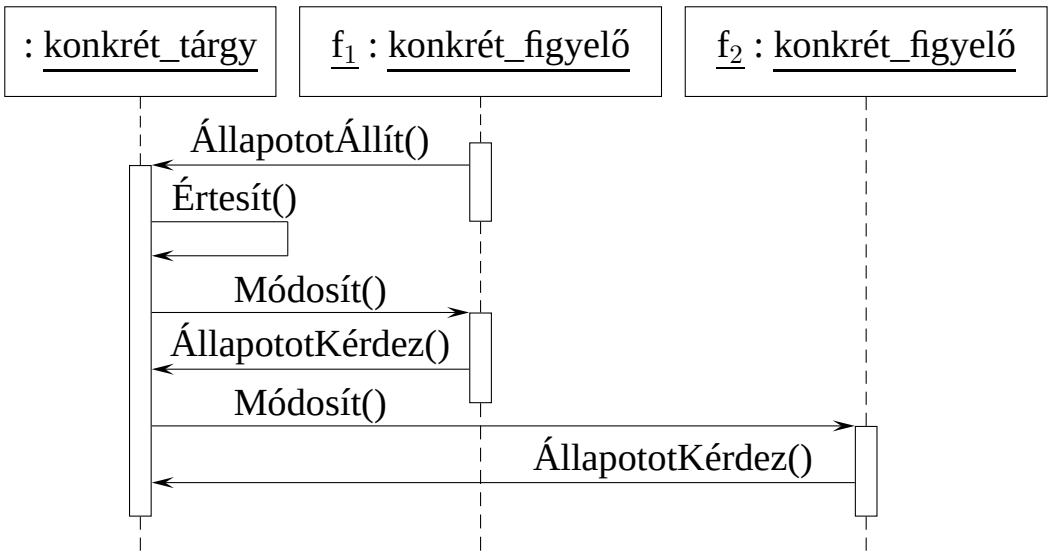
Szerkezet:


tárgy
figyelő* figyelők
Rákapcsol(figyelő)
Lekapcsol(figyelő)
Értesít()
*
figyelő
Módosít()
for all f in figyelők
f->Módosít()
konkrét_tárgy
tárgy_állapot
ÁllapototKérdez()
ÁllapototÁllít()
return tárgy_állapot
konkrét_figyelő
figyelő_állapot
konkrét_tárgy* t
Módosít()
figyelő_állapot =
t->ÁllapototKérdez()

Elemek:

- tárgy: Ismeri a figyelőit, amelyek száma tetszőleges. Lehetőséget biztosít figyelő objektumok hozzávételére és eltávolítására.
- figyelő: Meghatározza a módosítási felületét azoknak az objektumoknak, amelyeket értesíteni kell.
- konkrét_tárgy: Tartalmazza a konkrét_figyelők számára érdekes állapotot. Állapotváltozás esetén értesíti a figyelőit.
- konkrét_figyelő: Hivatkozik a konkrét_tárgy objektumra. Tartalmazza az állapotot, amelynek konzisztensnek kell lennie a tárgyéval. Implementálja a figyelő módosítási felületét, ezzel biztosítva a konzisztenciát.

Együttműködés: A konkrét_tárgy értesíti a figyelőit olyan változás esetén, amely inkonzisztenciát okozhatna az aktuális állapot és a figyelői állapota között. Az értesítés után a konkrét_figyelő lekérdezheti a tárgy állapotát. Ennek segítségével állítja helyre a konzisztenciát. Ezt szemlélteti a következő szekvenciadiagram két figyelő esetén.



Példa kód:

```
class Targy;

class Figyelo
{
public:
    virtual ~Figyelo();
    virtual void Modosit(Targy *valtott_targy) = 0;
protected:
    Figyelo();
};
```

```
class Targy
{
public:
    virtual ~Targy();
    virtual void Rakapcsol(Figyelo *f);
    virtual void Lekapcsol(Figyelo *f);
    virtual void Ertesit();
protected:
    Targy();
private:
    List<Figyelo *> *figyelok;
};
```

```
void Targy::Rakapcsol(Figyelo *f)
{
    figyelok->Append(f);
}

void Targy::Lekapcsol(Figyelo *f)
{
    figyelok->Remove(f);
}

void Targy::Ertesit()
{
    ListIterator<Figyelo *> it(figyelok);
    for ( it.First(); !it.Done(); it.Next() )
    {
        it.CurrentItem()->Modosit(this);
    }
}
```

Egy lehetséges alkalmazás az idő megjelenítése a számítógépen. Ekkor az időzítő a megfelelő műveletekkel kiegészítve a tárgy objektumból örököltethető, figyelő pedig lehet ennek digitális vagy analóg megjelenítése.

```
class Idozito : public Targy
{
public:
    Idozito();
    virtual int Ora();
    virtual int Perc();
    virtual int Masodperc();
    void Tikk();
};

void Idozito::Tikk()
{
    // idő állítása
    Ertesit();
}
```

```
class DigitalisOra : public Figyelo
{
public:
    DigitalisOra(Idozito *i);
    virtual ~DigitalisOra();
    virtual void Modosit(Targy *t);
    virtual void Rajzol();
private:
    Idozito  *targy;
};

DigitalisOra::DigitalisOra(Idozito *i)
{
    targy = i;    targy->Rakapcsol(this);
}

DigitalisOra::~DigitalisOra()
{
    targy->Lekapcsol(this);
}
```

```
void DigitalisOra::Modosit(Targy *t)
{
    if ( t == targy )    Rajzol();
}

void DigitalisOra::Rajzol()
{
    int ora = targy->Ora();
    int perc = targy->Perc();
    int mp = targy->Masodperc();
    // a digitális óra kirajzolása
}
```

```
class AnalogOra : public Figyelo
{
    AnalogOra(Idozito *i);
    virtual ~AnalogOra();
    virtual void Modosit(Targy *t);
    virtual void Rajzol();
private:
    Idozito  *targy;
};
```

Ezek után egy digitális és egy analóg óra, amelyek ugyanazt az időt mutatják:

```
Idozito *idozito = new Idozito;
AnalogOra *a_ora = new AnalogOra(idozito);
DigitalisOra *d_ora = new DigitalisOra(idozito);
```

Iterátor

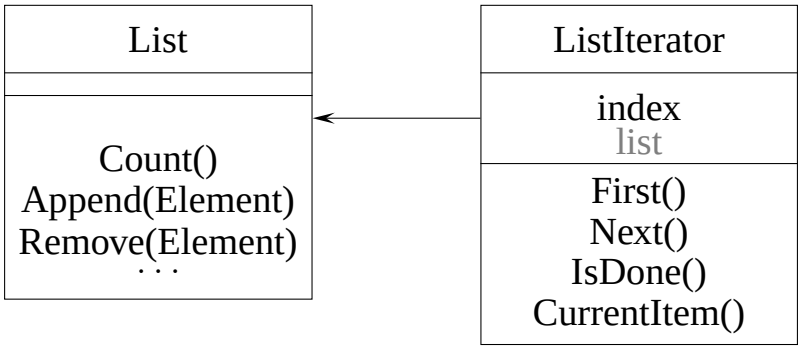
Név, osztály: iterátor (iterator); viselkedési (behavioral).

Cél: Egy aggregátum objektum elemeinek az elérését biztosítani, anélkül, hogy a reprezentációt ismernénk.

Más nevek: cursor.

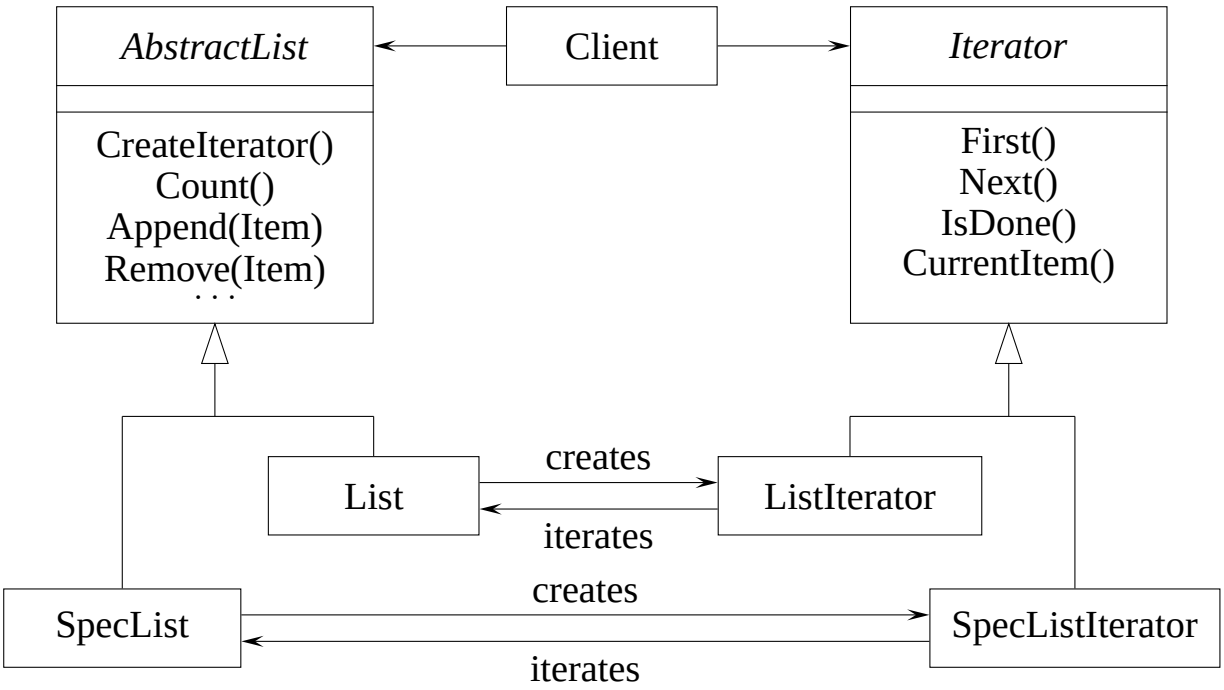
Motiváció: Egy aggregátumnak (például lista) biztosítania kell az alkotóelemeinek elérhetőségét, a belső szerkezet felfedése nélkül. Továbbá lehetséges, hogy különböző módokon akarjuk bejárni az elemeket (normál, fordított). Ugyanakkor nem szerencsés ennek érdekében túl sok művelettel ellátnunk a lista felületét, még akkor sem, ha esetleg minden bejárási módot előre jelezni tudunk. Ugyanazon a listán egyszerre több bejárás is „futhat”.

Erre egy lehetséges megoldás, ha például egy `List` (lista) osztály a `ListIterator` osztályt használja a következő ábra szerint.



A `ListIterator` használata előtt meg kell adni a bejárandó listát. A bejárás elválasztása a listától lehetővé teszi, hogy különböző bejárásokat definiáljunk, anélkül, hogy a `List` osztály interfészét nagyon megnövelnénk.

Most a bejáró művelet és a lista szoros kapcsolatban áll. A felhasználónak tudnia kell, hogy egy listát jár be, más szerkezetre a megoldás nem alkalmazható. Ezt öröklődés segítségével oldhatjuk meg, amelyben a bejárást általánosítjuk.



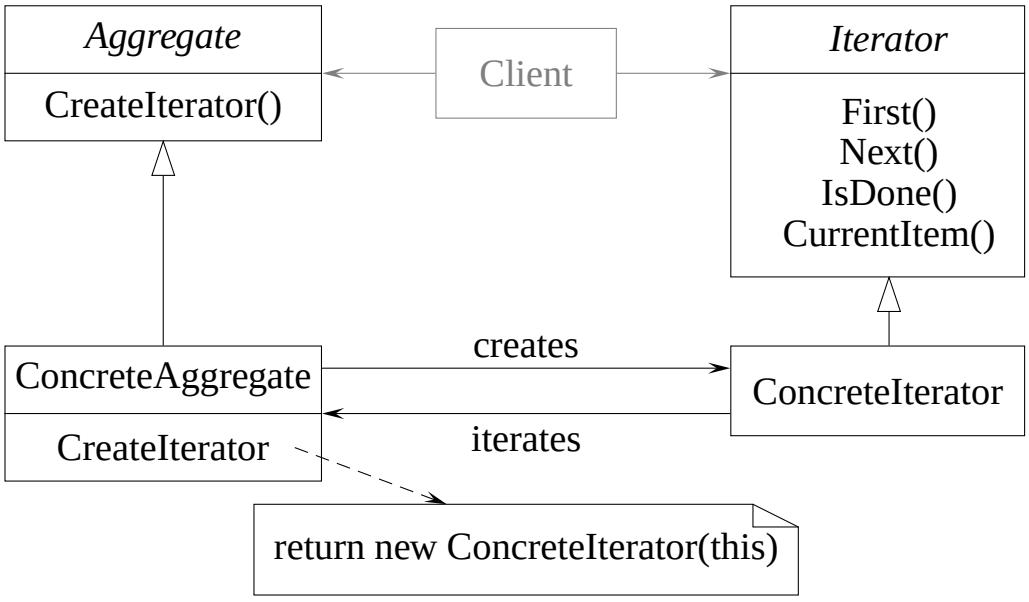
Az iterátort az aktuális lista osztálytól függetlenül kell létrehozni, ezért minden lista objektumnak létre kell hoznia a megfelelő iterátort.

Erre való a `CreateIterator` művelet. (Ez a *Gyár* minta egy esete.)

Felhasználhatóság: Az iterátor minta használható az alábbi esetekben:

- Egy aggregátum elemeinek elérésére úgy, hogy nem fedjük fel a belső reprezentációt.
- Aggregátumok többféle és többszörös bejárásának támogatására.
- Különböző aggregációs szerkezetek bejárásához egy egységes felületet biztosít.

Szerkezet:


Aggregate
CreateIterator()
Client
Iterator
First()
Next()
IsDone()
CurrentItem()
creates
ConcreteAggregate
CreateIterator
ConcreteIterator
iterates
return new ConcreteIterator(this)

Elemek:

- Iterator: megadja a bejárás felületét (elemek elérése, továbblépés).
- ConcreteIterator: megvalósítja az Iteratort, és nyilvántartja az aktuális elemet.
- Aggregate: megadja az Iterator objektum létrehozásának a felületét.
- ConcreteAggregate: megvalósítja az Iterator létrehozását egy megfelelő példány megadásával.

Együttműködés: A `ConcreteIterator` példánya nyilvántartja az aggregátum aktuális elemét, és meg tudja határozni a bejárás következő elemét.

Példa kód:

```
template <class Item> class List
{
public:
    List(long size = DEF_CAPACITY);
    long Count() const;
    Item& Get(long index) const;
    // ...
};
template <class Item> class Iterator
{
public:
    virtual void First() = 0;
    virtual void Next() = 0;
    virtual bool IsDone() const = 0;
    virtual Item CurrentItem() const = 0;
protected:
    Iterator();
};
```

```
template <class Item> class ListIterator : public Iterator<Ite
{
public:
    ListIterator(const List<Item> *li);
    virtual void First();
    virtual void Next();
    virtual bool IsDone() const;
    virtual Item CurrentItem() const;
private:
    const List<Item> *_list;
    long _current;
};

template <class Item> ListIterator<Item>::
    ListIterator(const List<Item> *li) : _list(li), _current(0
{
}
```

```
template <class Item> void ListIterator<Item>::First()
{
    _current = 0;
}

template <class Item> void ListIterator<Item>::Next()
{
    _current++;
}

template <class Item> bool ListIterator<Item>::IsDone() const
{
    return _current >= _list->Count();
}

template <class Item> Item ListIterator<Item>::CurrentItem() c
{
    if ( IsDone() )    throw IteratorOutOufBounds;
    return _list->Get(_current);
}
```

Egy lehetséges alkalmazás egy vállalat alkalmazottait kinyomtató eljárás. Tegyük fel, hogy rendelkezésünkre áll az alkalmazottak (`Employee`) listája (`List`), és az alkalmazottak osztály rendelkezik egy alkalmazott adatait kinyomtató `Print` eljárással. Ekkor a megfelelő eljárás a következő.

```
void PrintEmployees(Iterator<Employee*> &it)
{
    for ( it.First(); !it.IsDone(); it.Next() )
        it.CurrentItem()->Print();
}
```

Az eljárás használatához létre kell hozni a megfelelő iterátorokat.

```
List<Employee*> *employees;
//  ...
ListIterator<Employee*> forward(employees);
ReverseListIterator<Employee*> backward(employees);

PrintEmployees(forward);
PrintEmployees(backward);
```

Állapot

Név, osztály: állapot (state); viselkedési (behavioral).

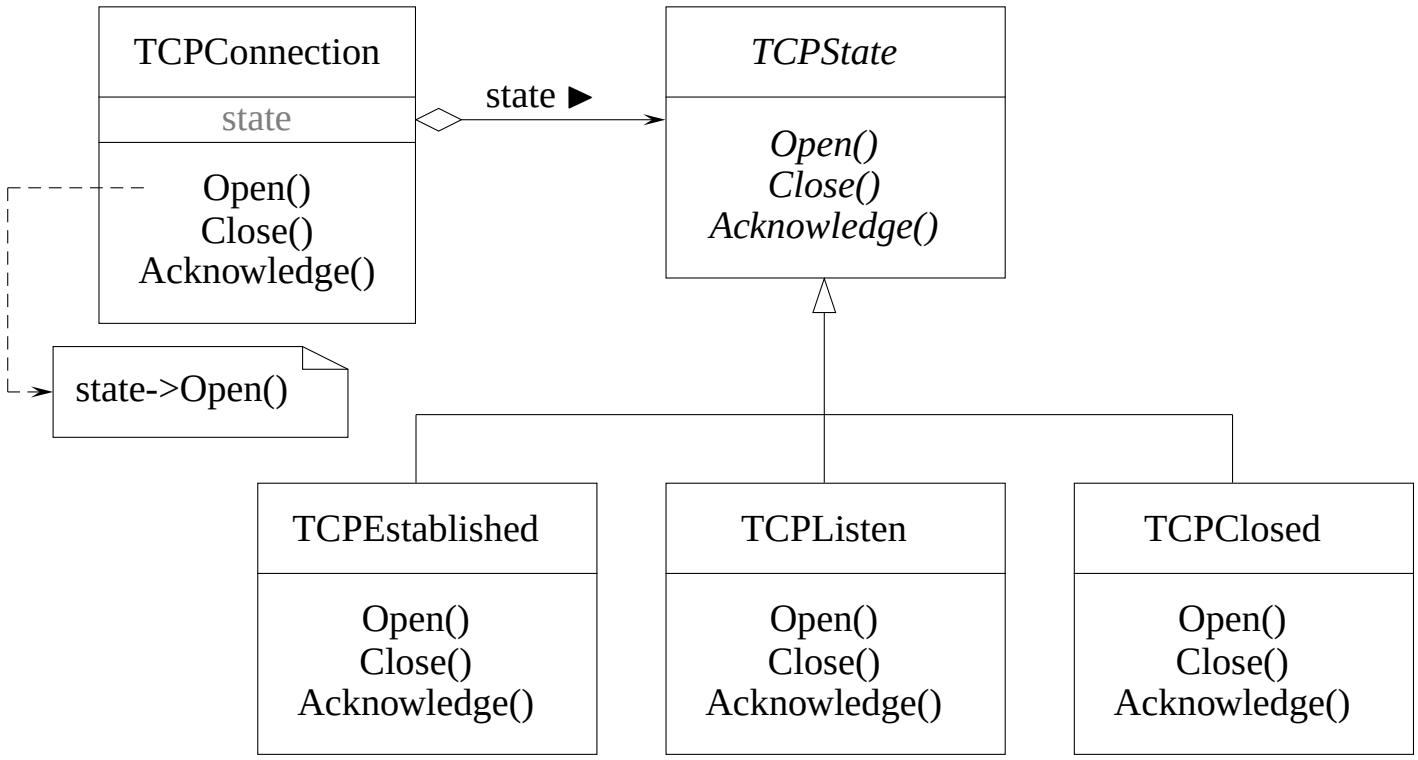
Cél: Lehetővé tenni, hogy egy objektum megváltoztassa a viselkedését, ha az állapota változik. A hatása olyan, mintha az objektum megváltoztatná az osztályát.

Más nevek: objects for states.

Motiváció: Tegyük fel, hogy adott egy `TCPConnection` osztály, amely hálózati kapcsolatot reprezentál. Ennek egy objektuma különböző állapotokat vehet fel: létrejött, figyelő, lezárt. Amikor egy ilyen objektum üzenetet kap más objektumoktól, az aktuális állapot függvényében eltérően viselkedik. Például egy megnyitási igény hatása más ha az állapot lezárt, illetve létrejött.

A minta alapötlete egy absztrakt osztály bevezetése, esetünkben ez legyen a `TCPState`. Ez reprezentálja a különböző (hálózati kapcsolati) állapotokat. Az osztályban deklarálunk egy olyan felületet, amely közös lenne az összes olyan osztályban, amelyek a különböző állapotokhoz tartoznának. Ezen absztrakt osztályból származtatott osztályok valósítják meg az állapotokra jellemző viselkedéseket.

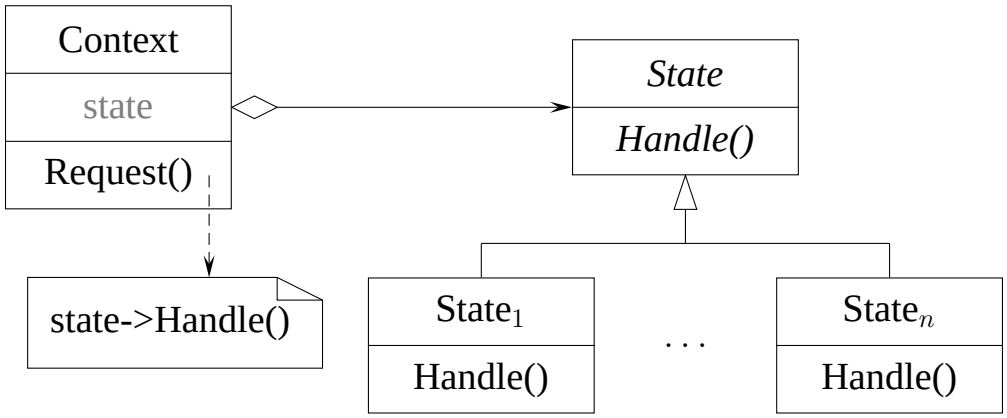
Ha az objektum állapota megváltozik, akkor a változásnak megfelelően cseréli az aggregációs állapot-objektumot.


TCPConnection
state
Open()
Close()
Acknowledge()
state ▶
TCPState
Open()
Close()
Acknowledge()
state->Open()
TCPEstablished
Open()
Close()
Acknowledge()
TCPListen
Open()
Close()
Acknowledge()
TCPClosed
Open()
Close()
Acknowledge()

Felhasználhatóság: Az állapot minta használható az alábbi esetekben:

- Egy objektum viselkedését az állapota határozza meg, és a viselkedést futási időben kell megváltoztatni az állapot függvényében,
- A műveletekben nagy méretű, több részes feltételes utasítások szerepelnek, ahol a feltétel az objektum állapotától függ. Az állapotot rendszerint egy felsorolási típussal adjuk meg. Gyakran több művelet is ugyanezt a feltételes szerkezetet tartalmazza.

Szerkezet:


Context
state
Request()
state->Handle()
State
Handle()
State1
Handle()
. . .
Staten
Handle()

Elemek:

- Context: Az ügyfeleknek (kliens objektumok) megadja a felületet. Az aktuális állapot meghatározásával együtt karbantartja a `state` változót úgy, hogy az egy megfelelő $\texttt{State}_i$ osztály példánya legyen.
- State: Meghatározza a speciális állapotok közös felületét.
- State_i: Minden osztály a Context egy állapotához tartozó speciális viselkedést implementál.

Együttműködés:

- A Context osztály objektuma továbbítja az állapotspecifikus igényeket az aktuális State$_i$ osztály objektumának.
- Egy Context objektum átadhatja önmagát, mint paramétert az igényt kezelő State objektumnak. Így szükség esetén a State objektum elérheti azt.
- Context az elsődleges felület a kliensek számára. A kliensek egy ilyen objektumot a State objektumok segítségével konfigurálhatnak. Miután egy Context objektum konfigurált, a klienseknek nem kell direkt módon kezelniük a State objektumokat.
- Vagy a Context vagy a State$_i$ osztályok határozzák meg a következő állapotot, az állapotátmenet feltételével együtt.

Példa kód:

```
class TCPOctetStream;
class TSPState;
class TCPConnection
{
public:
    TCPConnection();
    void ActiveOpen();
    void PassiveOpen();
    void Close();
    void Send();
    void Acknowledge();
    void Synchronize()
    void ProcessOctet(TCPOctetStream*);
private:
    friend class TCPState;
    void ChangeState(TCPState*);
    TCPState *_state;
};
```

A `_state` adattagban tároljuk a megfelelő viselkedést leíró objektumot.

A `TCPState` osztály szintén tartalmazza az állapotváltoztató műveletet az együttműködés miatt, és minden ilyen objektum műveletének paramétere egy hivatkozás a `TCPConnection` példányára, így biztosítva az adatok elérhetőségét, és az állapot megváltoztathatóságát.

```
class TCPState
{
public:
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    virtual void Close(TCPConnection*);
    virtual void Synchronize(TCPConnection*);
    virtual void Acknowledge(TCPConnection*);
    virtual void Send(TCPConnection*);
protected:
    void ChangeState(TCPConnection*, TCPState*);
};
```

```
TCPConnection::TCPConnection()
{
    _state = TCPClosed::Instance();
}
void TCPConnection::ChangeState(TCPState *s)
{
    _state = s;
}
void TCPConnection::ActiveOpen()
{
    _state->ActiveOpen(this);
}
void TCPConnection::PassiveOpen()
{
    _state->PassiveOpen(this);
}
void TCPConnection::Close()
{
    _state->Close(this);
}
```

```
void TCPConnection::Acknowledge()
{
    _state->Acknowledge(this);
}
void TCPConnection::Synchronize()
{
    _state->Synchronize(this);
}
```

`TCPState` megvalósítja az összes igény alapértelmezett viselkedését (esetünkben ez az üres tevékenység) és az állapotváltoztató műveletet.

```
void TCPState::Transmit(TCPConnection *t, TCPOctetStream *os)
void TCPState::ActiveOpen(TCPConnection *t) {}
void TCPState::PassiveOpen(TCPConnection *t) {}
void TCPState::Close(TCPConnection *t) {}
void TCPState::Synchronize(TCPConnection *t) {}
void TCPState::Acknowledge(TCPConnection *t) {}
void TCPState::Send(TCPConnection *t) {}

void TCPState::ChangeState(TCPConnection *t, TCPState *s)
{
    t->ChangeState(s);
}
```

A `TCPState` osztályból származtatott osztályok valósítják meg az állapotoknak megfelelő viselkedést.

```
class TCPEstablished : public TCPState {
public:
    static TCPState *Instance();
    virtual void Transmit(TCPConnection*, TCPOctetStream*);
    virtual void Close(TCPConnection*);
};
class TCPListen : public TCPState {
public:
    static TCPState *Instance();
    virtual void Send(TCPConnection*);
    // ...
};
class TCPClosed : public TCPState {
public:
    static TCPState *Instance();
    virtual void ActiveOpen(TCPConnection*);
    virtual void PassiveOpen(TCPConnection*);
    // ...
};
```

A származtatott osztályokban nincs lokális állapot, ezért ezek megoszthatók, így egyetlen példányra van csak belőlük szükség. Ezt adja meg az `Instance` művelet. (Ezért ezek az osztályok *Egykék*.)

Az implementáció:

```
void TCPEstablished::Transmit(TCPConnection *t, TCPOctetStream
{
    t->ProcessOctet(os);
}
void TCPEstablished::Close(TCPConnection *t)
{
    // tevékenységek
    ChangeState(t, TCPListen::Instance());;
}
void TCPListen::Send(TCPConnection *t)
{
    // tevékenységek
    ChangeState(t, TCPEstablished::Instance());;
}
```

```
void TCPClosed::ActiveOpen(TCPConnection *t)
{
    // tevékenységek
    ChangeState(t, TCPEstablished::Instance());;
}
void TCPClosed::PassiveOpen(TCPConnection *t)
{
    ChangeState(t, TCPListen::Instance());;
}
```

Egyke

Név, osztály: egyke (singleton); objektum létrehozási (object creational).

Cél: Biztosítani, hogy egy osztályhoz csak egy példány tartozhat, és megteremteni a példány globális elérhetőségét.

Más nevek: —.

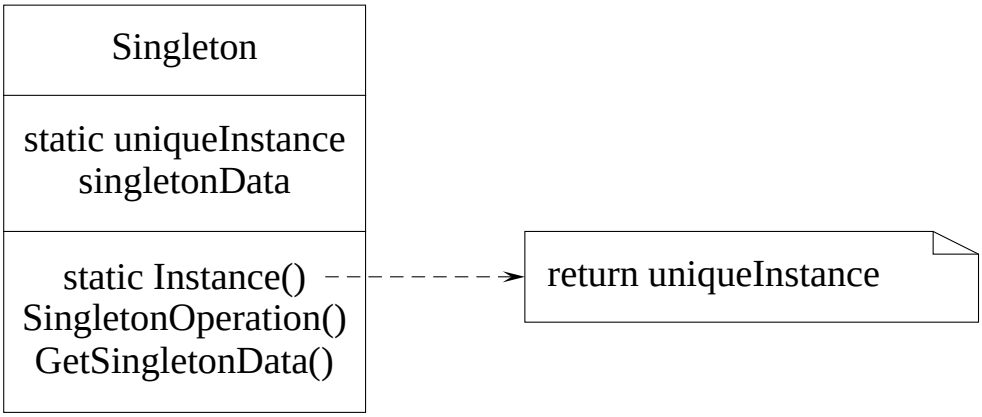
Motiváció: Bizonyos osztályok esetén fontos, hogy pontosan egy példány objektum létezzen. Például egy fájlrendszer illetve egy ablakkezelő lehet csak egy rendszer esetén. Ezt kell biztosítanunk, és a megfelelő elérhetőséget. Például egy globális változó esetén az elérhetőség biztosított, de a többszöri példányosítás lehetősége megmarad.

Jobb megoldás, ha maga az osztály felel az egyetlen példány létrehozásáért és kezeléséért. Az osztály azt is biztosítja, hogy csak egyetlen példányt lehet létrehozni, többet nem.

Felhasználhatóság: Az egyke minta használható az alábbi esetekben:

- Egy osztálynak csak egyetlen példánya lehet, és azt a klienseknek egy előre ismert módon kell elérniük, manipulálniuk.
- Amikor az egyetlen példány kiterjeszthető származtatással, és a klienseknek egy ilyen kiterjesztett példányt kell használniuk anélkül, hogy a kódot megváltoztatnánk.

Szerkezet:



Elemek:

- Singleton: Megadja az `Instance` műveletet, amelynek segítségével a kliensek elérhetik az egyetlen példányt. Ez egy osztályművelet. Felelős az egyetlen példány létrehozásáért.

Együttműködés: A kliensek csak az `Instance` műveleten keresztül férhetnek hozzá az osztály egyetlen példányához.

Példa kód:

```
class Singleton
{
public:
    static Singleton* Instance();
protected:
    Singleton();
private:
    static Singleton *_instance;
};
```

```
Singleton* Singleton::_instance = 0;

Singleton* Singleton::Instance()
{
    if ( _instance == 0 )  _instance = new Singleton;
    return _instance;
}
```

Miért nem lehet az egykét globális vagy statikus objektumként definiálni és az automatikus inicializálásra hagyatkozni?

1. Nem tudjuk biztosítani, hogy csak egy példányt deklarálnak a statikus objektumból.
2. Lehet, hogy áll rendelkezésre elegendő információ ahhoz, hogy minden egykét inicializáljunk a statikus inicializációs időben. Lehet, hogy olyan értékekre van szükségünk, amelyeket a program végrehajtása során később ismerünk meg.
3. A C++ nyelv nem definiálja a különböző egységekben található globális objektumok konstruktorainak meghívási sorrendjét. Ezért, ha az egykék között bármilyen függőség áll fenn, akkor hibák léphetnek fel.

Közvetítő

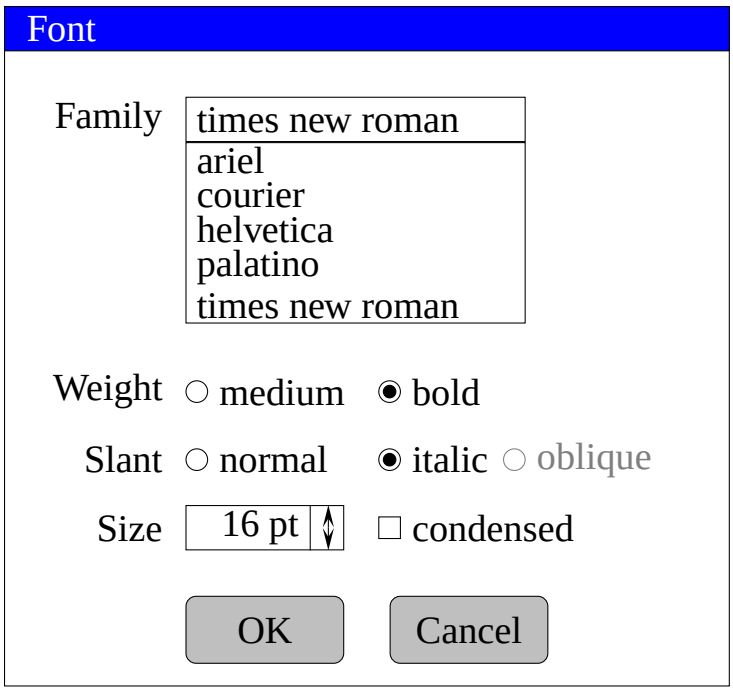
Név, osztály: közvetítő (mediator), viselkedési (behavioral).

Cél: Olyan objektum megadása, amely tartalmazza, hogy objektumok egy csoportja miként működik együtt.

Más nevek: —.

Motiváció: Az objektumelvű tervezés során a rendszer viselkedését az objektumok között osztjuk szét. Ennek eredménye lehet egy olyan szerkezet, amelyben sok kapcsolat jön létre az objektumok között. A legrosszabb esetben minden objektum ismeri az összes többit. Ez a nagymértékű összekapcsolódás gátja lehet az újrafelhasználhatóságnak, a rendszer jellege monolitikussá válik. A viselkedés módosítása is nehézkes, mert az túl sok objektum között oszlik el.

Példaként tekintsünk egy fontválasztó dialógusablakot, amelyben több elem (gomb, menü, lista) szerepel.

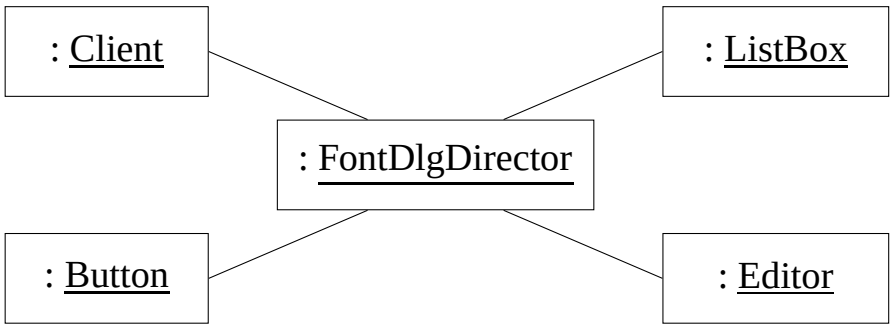


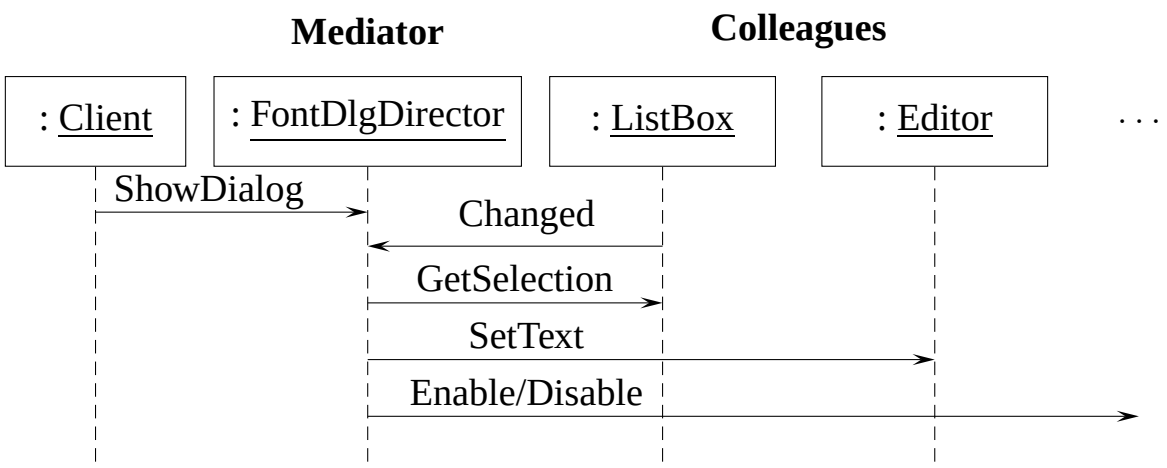
Az elemek között összefüggés van. Ennek megfelelően nem lehet minden esetben aktivizálni egy elemet, illetve nem lehet akármit választani benne.

Különböző dialógusablakok esetén az elemek közötti összefüggés eltérő, ezért az elemeket minden esetben az adott alkalmazás szerint kell specializálni, hogy megfelelően működjenek együtt. Ha ezt származtatással oldanánk meg, akkor túl sok osztály jönne létre.

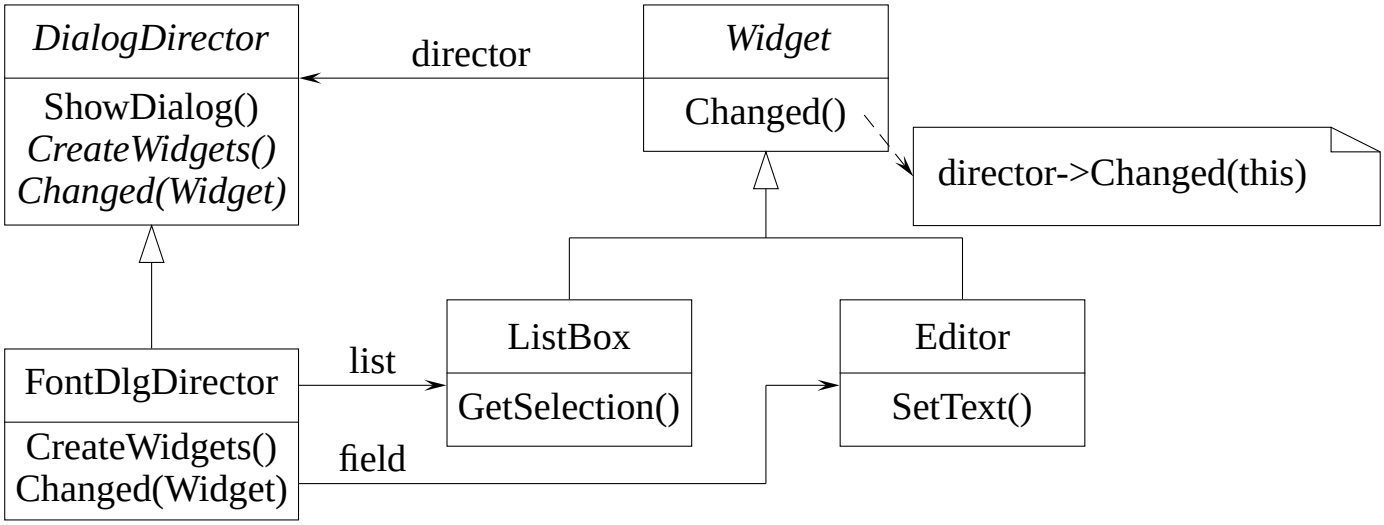
Ezt megoldhatjuk egy közvetítő objektum bevezetésével, amely tartalmazza az elemek együttes viselkedését, egymásra hatását. Egy közvetítő felelős egy csoportba tartozó objektumok kölcsönhatásainak vezérléséért és irányításáért. Ez egy közbeeső elem, amely lehetővé teszi, hogy a csoport tagjai direkt módon ne hivatkozzanak egymásra. Az objektumok (kollégák) csak a közvetítőt ismerik, így csökken a kapcsolatok száma.

Esetünkben egy `FontDlgDirector` objektumot vezethetünk be, mint közvetítőt, amely ismeri az elemeket.


: Client
: ListBox
: FontDlgDirector
: Button
: Editor


Mediator
Colleagues
: Client
: FontDlgDirector
: ListBox
: Editor
. . .
ShowDialog
Changed
GetSelection
SetText
Enable/Disable

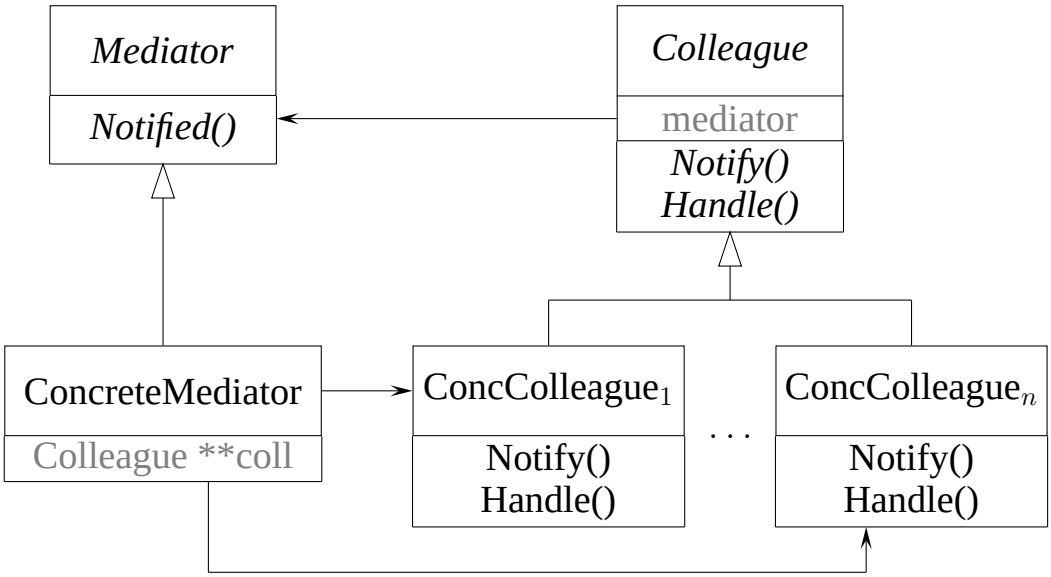
Az osztálydiagram, felhasználva az absztrakt `DialogDirector` és `Widget` osztályt:



Felhasználhatóság: A közvetítő minta használható az alábbi esetekben:

- Objektumok halmaza jól definiált, de összetett módon kommunikál; a kölcsönös függések szerkezete áttekinthetetlen, nehezen érthető.
- Egy objektum újrafelhasználása nehéz, mert sok objektumra hivatkozik illetve sok objektummal kommunikál.
- Több osztályra szétosztott viselkedést kell megvalósítanunk túl sok származtatás nélkül.

Szerkezet:


Mediator
Notified()
Colleague
mediator
Notify()
Handle()
ConcreteMediator
Colleague **coll
ConcColleague1
Notify()
Handle()
. . .
ConcColleaguen
Notify()
Handle()

Elemek:

- Mediator: A csoportba tartozó objektumok kommunikációs felületét definiálja. (Erre nincs szükség, amikor csak egyetlen közvetítőt használnak az objektumok.)
- ConcreteMediator: Az együttes viselkedést implementálja az objektumok koordinálásával. Ismeri és kezeli a csoport objektumait.
- Colleague: Absztrakt osztály a csoportba tartozó objektumokhoz.
- ConcColleague$_i$: Az csoportba tartozó objektumok konkrét osztályai. Minden ilyen osztály ismeri a közvetítő objektumát. Minden objektum a közvetítőn keresztül kommunikál a csoport többi objektumával a direkt kommunikáció helyett.

Együttműködés: A kollégák a közvetítő objektumnak küldenek és attól fogadnak üzeneteket. A közvetítő az együttes viselkedést úgy valósítja meg, hogy az egyes igényeket a megfelelő kollégáknak továbbítja. A kommunikáció megvalósítható a figyelő minta segítségével is. Ekkor a közvetítő a figyelő, a kollégák pedig a tárgyak.

Példa kód:

```
class Colleague;
class Coll_1;
// ...
class Coll_n;

class Mediator
{
public:
    virtual ~Mediator();
    virtual void Notified(Colleague*) = 0;
protected:
    Mediator();
    virtual void CreateColleagues() = 0;
};
```

```
class Colleague
{
public:
    virtual ~Colleague();
    virtual void Notify();
    virtual void Handle() = 0;
protected:
    Colleague(Mediator*);
    Mediator *mediator;
};
void Colleague::Notify()
{
    mediator->Notified(this);
}
class Coll_i
{
public:
    Coll_i(Mediator *m);        // mediator = m
    void Handle();
};
```

```
class ConcreteMediator : public Mediator
{
public:
    ConcreteMediator();
    virtual ~ConcreteMediator();
    void Notified(Colleague*);
protected:
    void CreateColleagues();
private:
    Colleague **coll;
};
```

```
void ConcreteMediator::CreateColleagues()
{
    coll = new Colleague*[k];
    // ...
    coll[j] = new Coll_i(this);
    // ...
    // coll elemeinek inicializálása
}

void ConcreteMediator::Notified(Colleague *c)
{
    for ( int i = 0; i < k; i++ )
        if ( coll[i] != c )
            // ha kell az objektum értesítése: coll[i]->Handle
}
```

Egy lehetséges alkalmazás a már bemutatott font meghatározás.

```
class DialogDirector
{
public:
    virtual ~DialogDirector();
    virtual void ShowDialog();
    virtual void Changed(Widget*) = 0;
protected:
    DialogDirector();
    virtual void CreateWidgets() = 0;
};
```

```
class Widget
{
public:
    Widget(DialogDirector*);
    virtual void Changed();
    virtual void HandleMouse(MouseEvent &e);
    // ...
private:
    DialogDirector *_director;
};

void Widget::Changed()
{
    _director->Changed(this);
}
```

```
class ListBox : public Widget
{
public:
    ListBox(DialogDirector*);
    virtual const char *GetSelection();
    virtual void SetList(List<char*>* listItems);
    virtual void HandleMouse(MouseEvent &e);
    // ...
};
class Editor : public Widget
{
public:
    Editor(DialogDirector*);
    virtual const char *GetText();
    virtual void SetText(const char *text);
    virtual void HandleMouse(MouseEvent &e);
    // ...
};
```

```
class Button : public Widget
{
public:
    Button(DialogDirector*);
    virtual void SetText(const char *text);
    virtual void HandleMouse(MouseEvent &e);
    // ...
};

void Button::HandleMouse(MouseEvent &e)
{
    // ...
    Changed();
};
```

```
class FontDlgDirector : public DialogDirector
{
public:
    FontDlgDirector();
    virtual ~FontDlgDirector();
    virtual void Changed(Widget*);
protected:
    virtual void CreateWidgets();
private:
    Button *ok;
    Button *cancel;
    ListBox *fontlist;
    Editor *fontname;
};
```

```
void FontDlgDirector::CreateWidgets()
{
    ok = new Button(this);         cancel = new Button(this);
    fontlist = new ListBox(this); fontname = new Editor(this);
    // lista feltöltése, elemek elhelyezése az ablakban
}

void FontDlgDirector::Changed(Widget *w)
{
    if ( w == fontlist )
        fontname->SetText(fontlist->GetSelection());
    else if ( w == ok )
        // a font kiválasztása, és a dialógus bezárása
    else if ( w == cancel )
        // dialógus bezárása
}
```

Feljegyzés

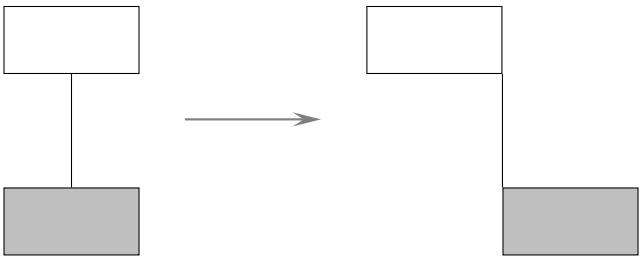
Név, osztály: feljegyzés (memento), viselkedési (behavioral).

Cél: Egy objektum belső állapotának felfedése és feljegyzése anélkül, hogy az információ elrejtést megsértenénk. A feljegyzett állapot alapján az objektum állapota a későbbiekben visszaállítható.

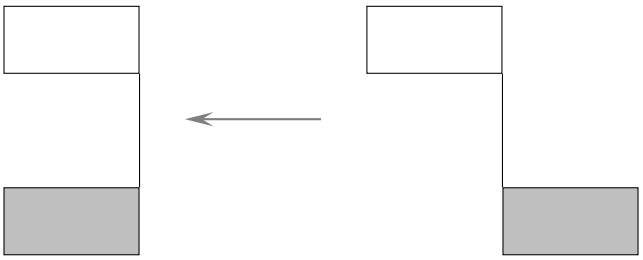
Más nevek: token.

Motiváció: Vannak olyan esetek, amikor szükséges egy objektum belső állapotának megjegyzése. Ezt kell tennünk, ha a rendszerben támogatni akarjuk a visszavonás (undo) műveletet. Ehhez el kell mentenünk egy megelőző állapotot, amit vissza lehet majd állítani. Ugyanakkor az objektumok elrejtik az adattagjaikat, azaz az állapotukat, ezért ezt kívülről nem lehet menteni az elrejtés megsértése nélkül.

Tekintsünk például egy grafikus szerkesztőt, amely támogatja az objektumok közötti kapcsolatokat. Két téglalapot például összeköthetünk vonallal, majd az egyik téglalapot eltolhatjuk. Az összekötő vonal megfelelően változik (a program gondoskodik erről).



Az eltolás visszavonásánál ügyelni kell arra, hogy nem elegendő csak a téglalapot mozgatni.



Ezt a problémát oldhatjuk meg a feljegyzés minta használatával. A feljegyzés egy objektum, amely egy másik objektum, a feljegyzés eredete, belső állapotának egy pillanatra vonatkozó képét tárolja. A visszavonás az eredet objektumtól igényli a feljegyzés objektumot. Az eredet inicializálja a feljegyzést az aktuális állapotnak megfelelő értékekkel. Csak az eredet képes kommunikálni a feljegyzéssel, a feljegyzés mások számára átlátszatlan.

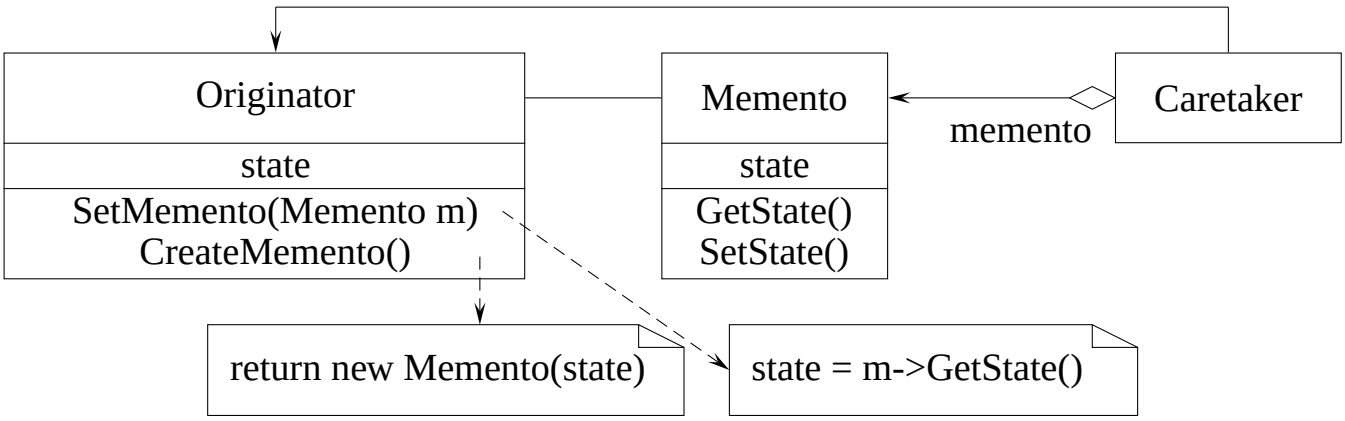
A példánkban az alakzatok közötti kapcsolatokat kezelő objektum az eredet. A visszavonás menete:

1. A program a kapcsolatkezelőtől egy feljegyzést igényel a mozgatás művelet mellékhatásairól, amit az létrehoz.
2. A visszavonás kiadásakor a program ezt a feljegyzést adja meg a kapcsolatkezelőnek.
3. A feljegyzés alapján a kapcsolatkezelő visszaállítja a megelőző állapotot.

Felhasználhatóság: A feljegyzés minta használható az alábbi esetben:

- Egy objektum pillanatnyi állapotát (vagy egy részét) el kell mentenünk, és visszaállítanunk később, és nem akarjuk felfedni az objektum reprezentációját.

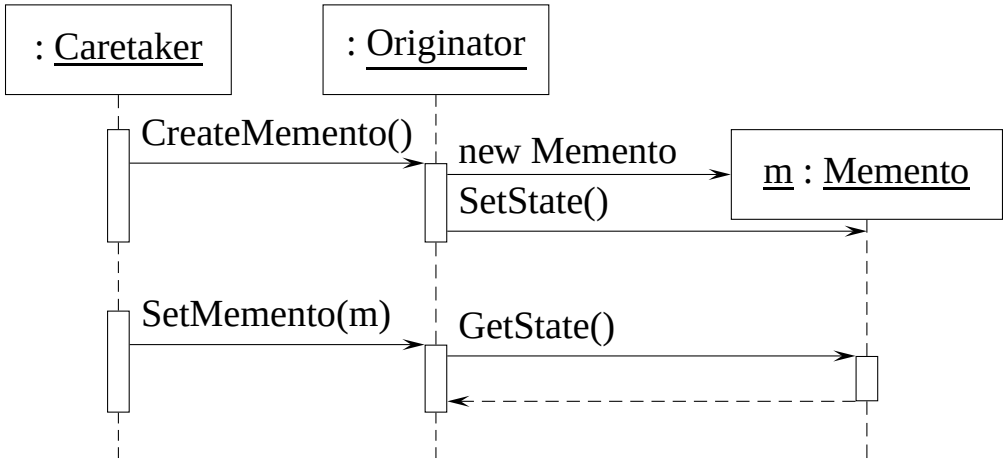
Szerkezet:



Elemek:

- Memento: Az eredet objektum belső állapotát vagy annak egy részét tárolja. Csak az eredet férhet hozzá a tárolt állapothoz. Gyakorlatilag két felülettel rendelkeznek.
- Originator: Létrehoz egy feljegyzést, amelyben a belső állapotát tárolja, és felhasználja a feljegyzést az állapot visszaállításához.
- Caretaker: Felel a feljegyzések tárolásáért, de nem hajt végre műveletet (állapot vizsgálat) a feljegyzésen. Ez a tulajdonképpeni visszavonó művelet.

Együttműködés:



Példa kód:

```
class State;

class Memento;

class Originator
{
public:
    Memento* CreateMemento();
    void SetMemento(const Memento*);
    // ...
private:
    State *_state;
    // ...
};
```

```
class Memento
{
public:
    // szűk felület, bármely objektum számára
    virtual ~Memento();
private:     // ezeket csak az Originator éri el
    friend class Originator;
    Memento();
    void SetState(State*);
    State *GetState();
    // ...
private:
    State *_state;
    // ...
};
```

Egy lehetséges alkalmazás a grafikus szerkesztő program. A kapcsolatkezelő objektum a `ConstraintSolver` osztály egyetlen példánya (egyke). Most csak a mozgatást vizsgáljuk, ami egy parancs mintának felel meg.

```
class Graphic;   // grafikai objektumok a szerkesztőben

class ConstraintSolverMemento;

class MoveCommand
{
public:
    MoveCommand(Graphic *target, const Point &delta);
    void Execute();
    void Unexecute();
private:
    ConstraintSolverMemento *_state;
    Point _delta;
    Graphic *_target;
};
```

```
class ContraintSolver
{
public:
    static ConstraintSolver *Instance();
    void Solve();
    void AddConstraint(Graphic *start, Graphic *end);
    void RemoveConstraint(Graphic *start, Graphic *end);
    ConstraintSolverMemento *CreateMemento();
    void SetMemento(ConstraintSolverMemento*);
private:
    // ...
};
class ConstraintSolverMemento
{
public:
    virtual ~ConstraintSolverMemento();
private:
    friend class ConstraintSolver;
    ConstraintSolverMemento();    // és még az állapot
};
```

Az eddigiek alapján a mozgatás parancs és annak visszavonása:

```
void MoveCommand::Execute()
{
    ConstraintSolver *solver = ConstraintSolver::Instance();
    _state = solver->CreateMemento();
    _target->Move(delta);
    solver->Solve();
}

void MoveCommand::Unexecute()
{
    ConstraintSolver *solver = ConstraintSolver::Instance();
    _target->Move(-delta);
    solver->SetMemento(_state);
    // állapot helyreállítása _state alapján
    solver->Solve();
}
```

Parancs

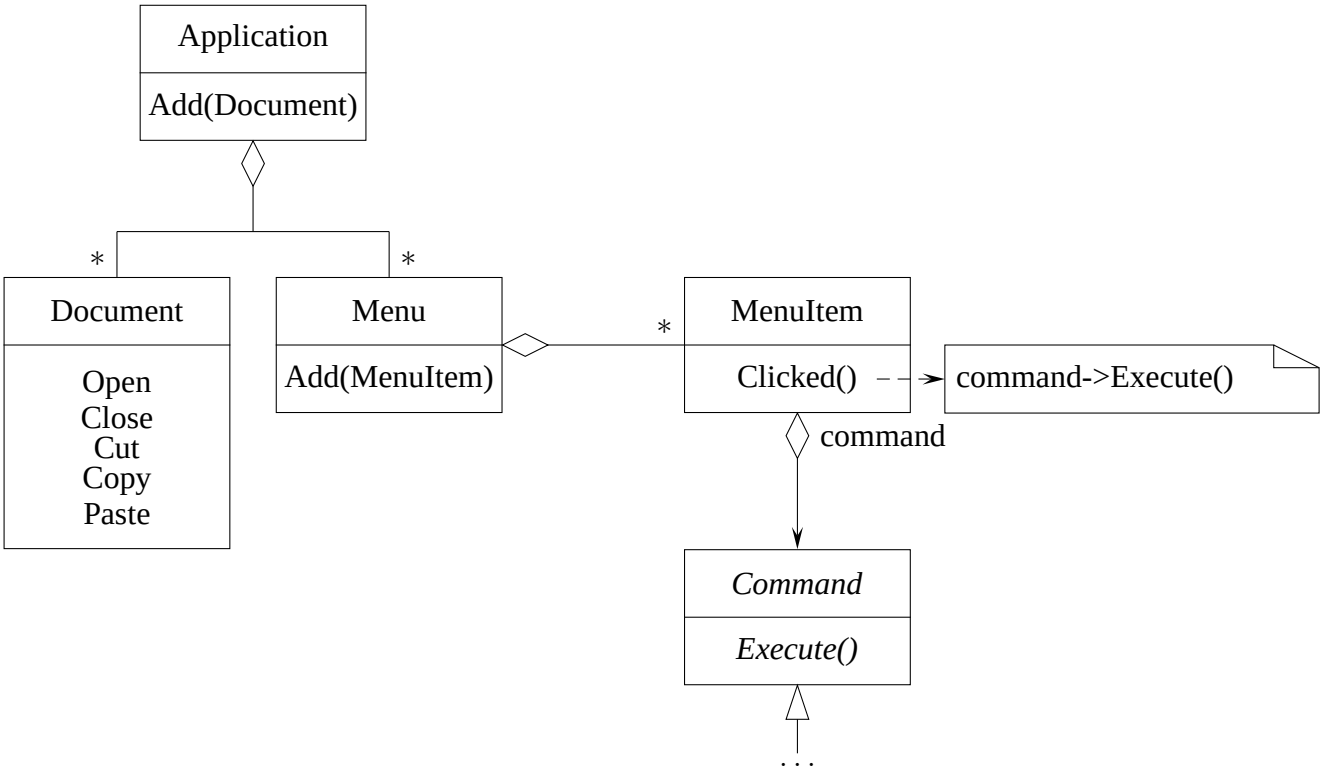
Név, osztály: parancs (command), viselkedési (behavioral).

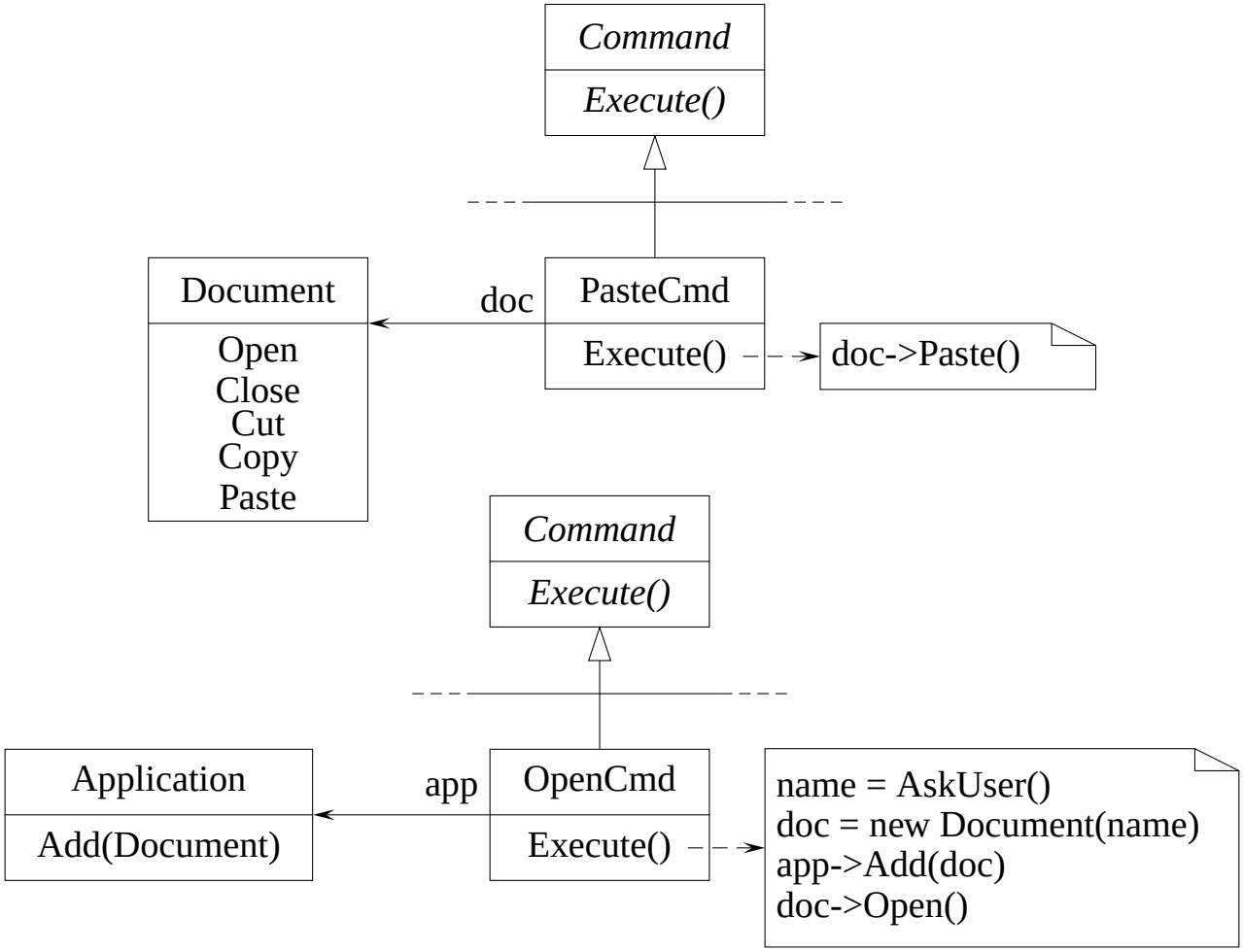
Cél: Egy igény objektumba ágyazása, így lehetővé téve, hogy a kliensek különböző igényeket adjanak ki, és ezeket tároljuk, visszavonjuk.

Más nevek: action, transaction.

Motiváció: Van amikor úgy kell egy igényt (parancsot) feldolgoznunk, hogy nem tudunk semmit magáról a műveletről vagy a művelet fogadójáról. Ilyen esetre példa, ha felhasználói felület készítő eszközt szeretnénk létrehozni. Ebben szerepelnek gombok, menük, amelyekhez akciók tartoznak. Ugyanakkor az eszköz nem valósíthatja meg az akciót, mert csak az eszközt használó alkalmazás ismeri azt.

A parancs minta segítségével az igényeket objektumokká alakítjuk, amelyeket tárolhatunk és továbbíthatunk igény szerint. Az alapötlet az absztrakt `Command` osztály, amely egy felületet ad a végrehajtandó műveletekhez. A legegyszerűbb esetben tartalmaz egy absztrakt `Execute` műveletet. Az ebből származtatott konkrét osztályok határozzák meg a fogadó–akció párosokat úgy, hogy a fogadót példányváltozóként tárolják, és az `Execute` műveletet megvalósítják.

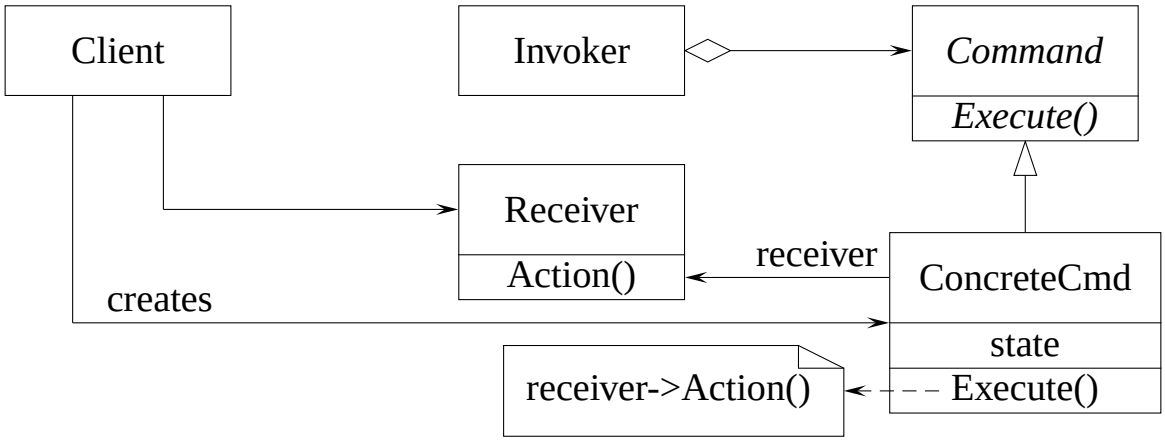

Application
Add(Document)
*
*
Document
Open
Close
Cut
Copy
Paste
Menu
Add(MenuItem)
*
MenuItem
Clicked()
command->Execute()
command
Command
Execute()
. . .


Command
Execute()
Document
Open
Close
Cut
Copy
Paste
doc
PasteCmd
Execute()
doc->Paste()
Command
Execute()
Application
Add(Document)
app
OpenCmd
Execute()
name = AskUser()
doc = new Document(name)
app->Add(doc)
doc->Open()

Felhasználhatóság: A parancs minta használható az alábbi esetekben:

- Ha objektumokat végrehajtandó akciókkal kell paraméterezni (callback függvények).
- Ha igényeket kell meghatároznunk, sorba tennünk, végrehajtanunk különböző időpontokban. A parancs objektum élettartama független az eredeti igénytől.
- A visszavonási művelet támogatására. Ebben az esetben a végrehajtás során tárolnunk kell a megfelelő állapotot, és rendelkezni kell egy `Unexecute` művelettel is.
- Változások naplózására. A napló alapján a tevékenységek újra végrehajthatók.
- Összetett tevékenységek (tranzakciók) megvalósítására.

Szerkezet:

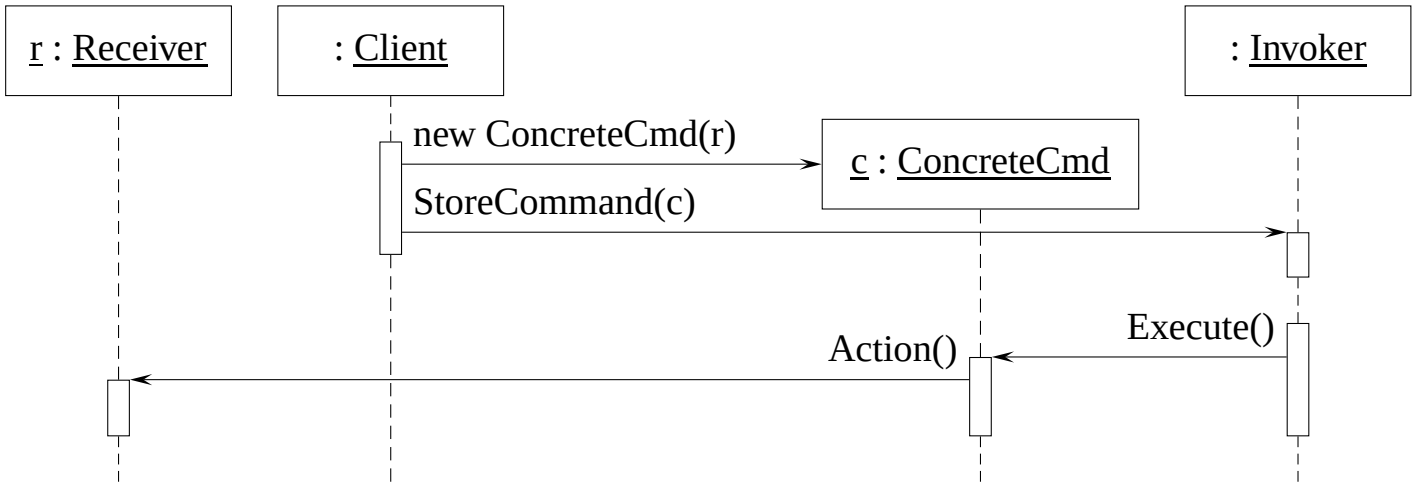

Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCmd
state
Execute()
creates
receiver->Action()

Elemek:

- Command: A művelet végrehajtási felületének deklarálása.
- ConcreteCmd: A fogadó (Receiver) objektum és az akció közötti összekapcsolás definíciója. (PasteCmd, OpenCmd)
- Client: Létrehozza a konkrét parancs objektumot és beállítja a fogadóját. (Application)
- Invoker: A parancs végrehajtását kezdeményezi. (MenuItem)
- Receiver: Ismeri az igényhez tartozó művelet végrehajtásának módját. Bármelyik osztály betöltheti ezt a szerepet. (Document, Application)

Együttműködés:

- A kliens létrehozza a konkrét parancsot és meghatározza a fogadóját.
- Egy kibocsátó (Invoker) objektum tárolja a konkrét parancsot.
- A kibocsátó kiad egy igényt az `Execute` művelet meghívásával. (Visszavonási lehetőség esetén a konkrét parancs tárolja a szükséges állapotokat.)
- A konkrét parancs meghívja a fogadó műveletét az igény kielégítése céljából.



Példa kód:

```
class Command
{
public:
    virtual ~Command();
    virtual void Execute() = 0;
protected:
    Command();
};

class Receiver
{
public:
    Receiver();
    virtual ~Receiver();
    void Action()
    // ...
};
```

```
class ConcreteCmd : public Command
{
public:
    ConcreteCmd(Receiver *r)    { rec = r; }
    virtual void Execute();
private:
    Receiver *rec;
    // ...
};

void ConcreteCmd::Execute()
{
    // visszavonási feljegyzések
    rec->Action();
};
```

(Egyszerű, argumentum nélküli és visszavonást nem igénylő parancsok esetén sablonként is elkészíthető a parancs osztály. A sablon paramétere a fogadó osztály.)

Egy lehetséges alkalmazás a már bemutatott felhasználói felület készítés. Csak az `OpenCmd` és a `PasteCmd` parancsokat vázoljuk. Felhasználjuk a már ismert `Command` osztályt.

```
class OpenCmd : public Command
{
public:
    OpendCmd(Application *a)    { app = a; }
    virtual void Execute();
protected:
    virtual const char *AskUser();
private:
    Application *app;
};
```

```
void OpenCmd::Execute()
{
    char *name = AskUser();
    if ( name != 0 )
    {
        Document *doc = new Document(this);
        app->Add(doc);
        doc->Open();
     }
}
```

```
class PasteCmd : public Command
{
public:
    PasteCmd(Document *d)       { doc = d; }
    virtual void Execute();
private:
    Document *doc;
};

void PasteCmd::Execute()
{
    doc->Paste();
}
```