Tervezési minták bemutatása

Készítette

Bartha Tibor
G-5S1L

Szoftver újra felhasználás

Fejlesztéskor célok:

- Gyorsaság
- Minőség
- Elfogadható ár

Az újrafelhasználhatóság céljai és lehetőségei:

- Gyorsítja a tervezést, a kódolást (egyszer már megcsináltuk)
- Biztonságosabb megoldásokat kínál (tesztelt kódot használunk)
- Megkönnyíti, gyorsítja a fejlesztők dolgát

Újrafelhasználás - evolúció

0. „Mindent a kályhától”
1. Copy & Paste
2. Függvény könyvtárak
3. Objektumok
4. Osztály könyvtárak (class libraries)
5. **Tervezési minták, minta nyelvek**
6. Komponens technológiák
7. Keretrendszerek (frameworks), vállalati sablonok (enterprise templates)

Tervezési minták

Mire jó?

- Gyakran ismétlődő problémákra működő válaszok ill. tervek.
- Nem kötődik programozási nyelvhez, annál általánosabb. Közös nyelv a fejlesztők között (nem kell sorról sorra értelmezni a kódot)

Tervezési minta

Általános minták:

Olyan, sokak által ismert formátumban dokumentált megoldásváz, amelynek alkalmazhatósága könnyen eldönthető egy adott probléma esetén, a végleges megoldás útmutató segítségével könnyen létrehozható.

SW minták:

Egymással kapcsolatban álló osztályok és objektumok, amelyek együtt egy adott objektum-orientált tervezési feladat vagy probléma megoldását szolgálják.

Tervezési minták - eredet

Alapötlet - Christopher Alexander:

- A Pattern Language (1977)
  - minták családi ház, iroda, lakóhely, középület tervezéséhez, emberközpontúbbá tételéhez
- The Timeless Way of Building (1979)
  - Építészet filozofikus alapokon és fordítva
  - Természet és ember kapcsolata taoista szemlélettel

Tervezési minták jelentése

- A tervezési minták olyan visszatérő megoldásokra adnak megoldást, melyekkel nap mint nap lehet találkozni az alkalmazás fejlesztések során.
- A tervezési minták tervezésről és objektumok együttműködéséről szól, valamint a kommunikációs platformok eleganciájáról, valamint újrafelhasználható megoldások az általánosan jelentkező programozói kihívásokra.

Tervezési minták jelentése

- A megoldást jellemzően egymással kommunikáló objektumok, interfészek és osztályok formájában adják meg, adott kontextuson belül. (Ugyanakkor nem objektum-orientált problémákra is lehetnek speciális minták).
- A tervezési minták szerepét a következő analógiával lehet egyszerűen megfogni: Ami a kódolásnak az osztálykönyvtár, az a tervezésnek a tervezési minta.

Hogyan segítenek a minták egy programozói feladat megoldásában?

- Megfelelő objektumok megtalálása (dekompozíció). A legtöbb tervezési minta absztrakt fogalmak osztályba, ill. objektumba zárásáról szól.
- A felbontás (granularitás) mértéke. Hogyan kezeljünk nagy számú objektumot, hogyan rejtsük el magasabb szinteken az összetett objektumokat? Hogyan állítsuk elő az objektumokat?
- Osztályok/objektumok interfészei: a minták meghatározzák, hogy mit szerepeltessünk, és mit ne jelenítsünk meg az interfészeken, ill. azt is, hogy hogyan kapcsolódhatnak az objektumok ezeken az interfészeken keresztül.

Hogyan segítenek a minták egy programozói feladat megoldásában?

- Objektumok megvalósítása: tisztább különbséget próbál tenni interfész és implementációs öröklődés között („Program to an interface, not an implementation”).
- Újrafelhasználhatóság támogatása: két megközelítés: öröklődés (statikus, fehér doboz) vagy kompozíció (dinamikus, fekete doboz). Az öröklődés felfedi az ősosztály belső szerkezetét, ugyanakkor erős megkötést ad a bővítések, újrahasznosítások szerkezetére („Favor object composition over class inheritance.”) Kompozíció esetén a feladatokat „delegálni” kell. További lehetőség (nem általános objektumorientált koncepció): paraméteres típusok.

Hogyan segítenek a minták egy programozói feladat megoldásában?

- A tervezési idejű és a futási idejű szerkezet közötti kapcsolat tisztázása. (Példa: élőlények anatómiája vs. teljes ökorendszer.)

- Későbbi módosítások, kiterjesztések: a minták önmagukban tartalmazzák a bővíthetőséget, ill. a jól definiált feladatok és hatókörök miatt várhatóan könnyebb ezen szoftverek módosítása (kevesebb, ill. tisztázottabb függőségi viszony).

Tervezési minta - felépítés

Minta nyelvek (Pattern languages)

- Eddig:
  - Szövegszerkesztő + rajzolóprogram
  - Szabad kéz a kötelező és opcionális elemek terén
- Törekvések:
  - Formátum egységesítés
  - Tool támogatás (automatikus feldolgozás)
  - Katalogizálás és visszakereshetőség elősegítése
- Egy lehetőség XML alapokon: **PCML** (Pattern & Component Markup Language)

Tervezési minták kategóriái

- Tervezési minták kategóriái
  - Előállítás: a feladat-objektumok gyártása
  - Strukturális: több objektum vagy osztály szervezése, kompozíciója
  - Viselkedési: objektumok vagy osztályok interakciója, felelősség és feladatok megosztása
  - Osztály alapú: a kapcsolatok statikusan kialakítottak, elsősorban az öröklődésen alapszanak
  - Objektum alapú: dinamikus minták (pl. mutatókkal), de természetesen ezek is használják az osztályok közötti kapcsolatokat

Tervezési minták kategóriái

		Feladat jellege (Purpose)		
		Előállítás (*Creational*)	Strukturális (*Structural*)	Viselkedési (*Behavioral*)
Hatókör (Scope)	Osztály (Class)	*Factory Method*	*Adapter*	*Interpreter* *Template Method*
	Objektum (Object)	*Abstract Factory* *Builder* *Prototype* *Singleton*	*Adapter* *Bridge* *Composite* *Decorator* *Facade* *Proxy*	*Chain of Resp.* *Command* *Iterator* *Mediator* *Memento* *Flyweight* *Observer* *State* *Strategy* *Visitor*

Tervezési minták kategóriái

- Az előállítással foglalkozó osztály alapú minták származtatott osztályokra bízzák a munka egy részét, míg az objektum alapúak egy másik objektumra.
- A strukturális osztály sablonok öröklődés alapján hoznak létre összetett osztályokat, míg az objektum sablonok adattagok (pl. mutatók) elhasználásával.
- Hasonló igaz a viselkedési mintákra.

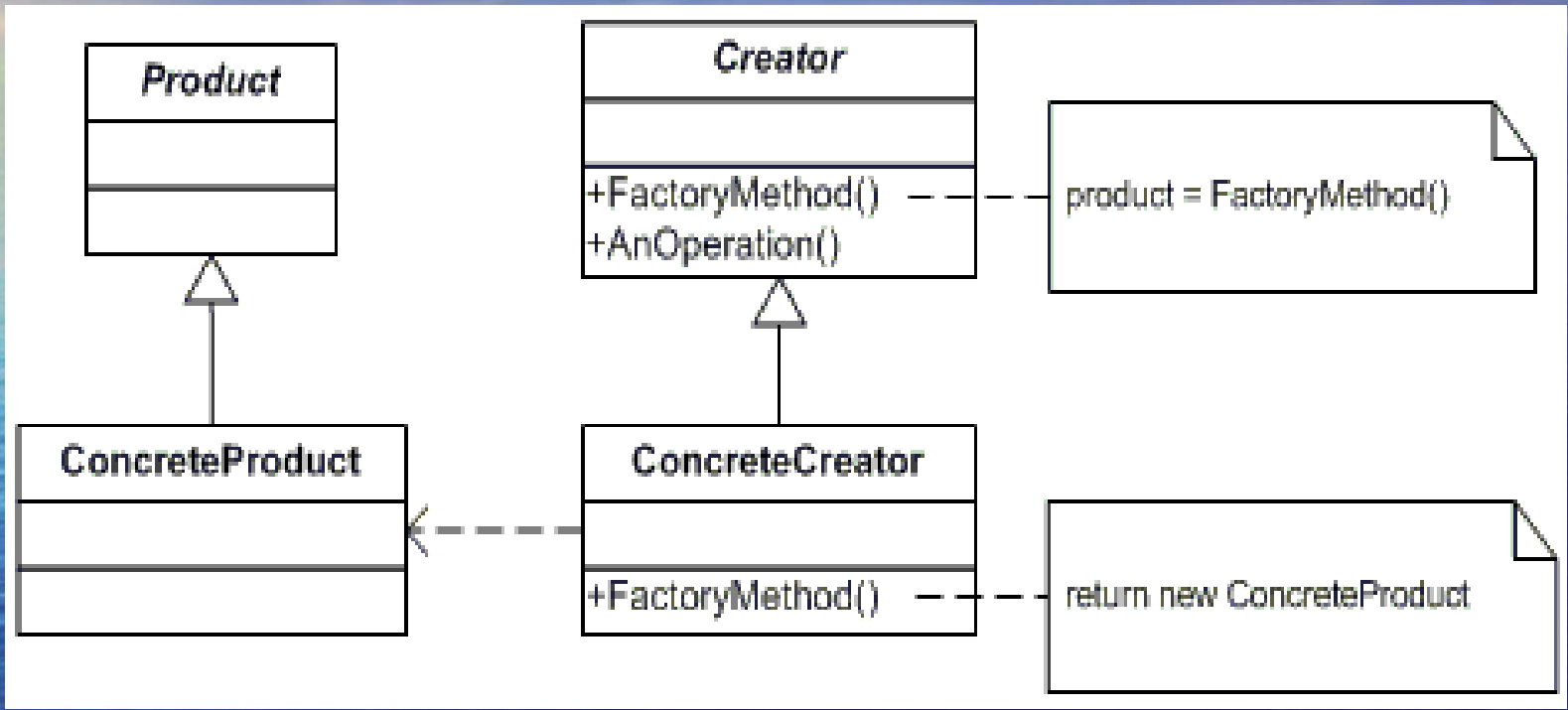
Creational Patterns
Factory Method

- Objektum létrehozására ad interfészt kiterjeszthető módon: a későbbiekben kialakított származtatott osztályok könnyen előállíthatók a meglévő infrastruktúrával.
- Cél:

  Egy objektum létrehozásához egy interfészt definiálunk, de a leszármazott osztályok eldönthetik, hogy milyen osztályt példányosítsanak valójában.
- Következmények:

  A Factory method minta segítségével elkerülhető, hogy alkalmazás specifikus osztályokat rakjunk a kódba.
- Nem bonyolult, jól tesre szabható

Creational Patterns
Factory Method



Példa: Különböző dokumentumok előállítása. Egy dokument osztályból származik a többi.

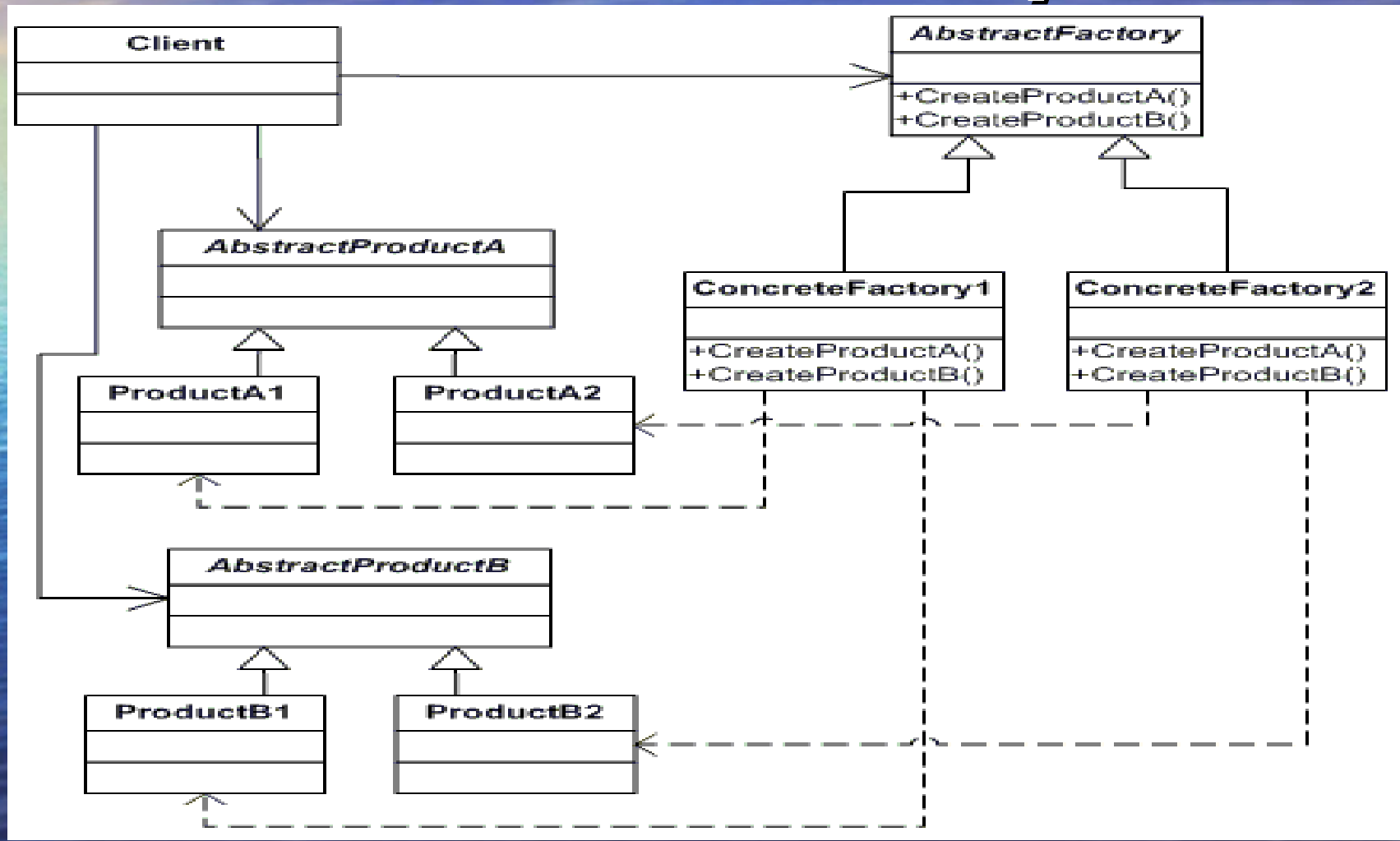
Creational Patterns
Factory Method

- **Product (Page)**
  - Interfészt definiál a metódusok létrehozásához
- **ConcreteProduct (SkillsPage, EducationPage, ExperiencePage)**
  - Implementálja a Prudoct interfészét
- **Creator (Document)**
  - Deklarálja a factory metódusait, amik egy Product típusú objektummal térnek vissza. Definiálja a factory default metódusát, mely a default ConcreteProduct objektumot adja vissza
  - Meghívható a factory method, hogy létrehozza a Product objektumot
- **ConcreteCreator (Report, Resume)**
  - Átértelmezi, felüldefiniálja a factory metódusait, így a ConcreteProduct egy példányát adja vissza

Creational Patterns
Abstract Factory

- Hasonló vagy egymástól függő objektumok készítése a konkrét osztály megnevezése nélkül.
- Létrehoz egy interfészt az egy családba tartozó, egymással kapcsolatban álló, vagy egymástól függő objektumok között anélkül hogy megadná a konkrét osztályokat.
- Abstract Factory használható platform specifikus osztályok elrejtésére
- Használható
  - amikor egy rendszert függetlenné akarunk tenni attól, ahogyan a termékeit előállítja, összekapcsolja illetve megjeleníti
  - amikor a rendszert konfigurálhatóvá akarjuk tenni, azáltal, könnyen lecserélhető legyen az előállított objektum
  - amikor egy objektumcsalád úgy lett tervezve, hogy azokban együtt kell működniük, és ezt a létrehozáskor érvényesíteni akarjuk
  - amikor egy egész osztálykönyvtárat akarunk létrehozni, de csak az interfészeket akarjuk nyilvánosságra hozni.

Creational Patterns
Abstract Factory



Creational Patterns
Abstract Factory

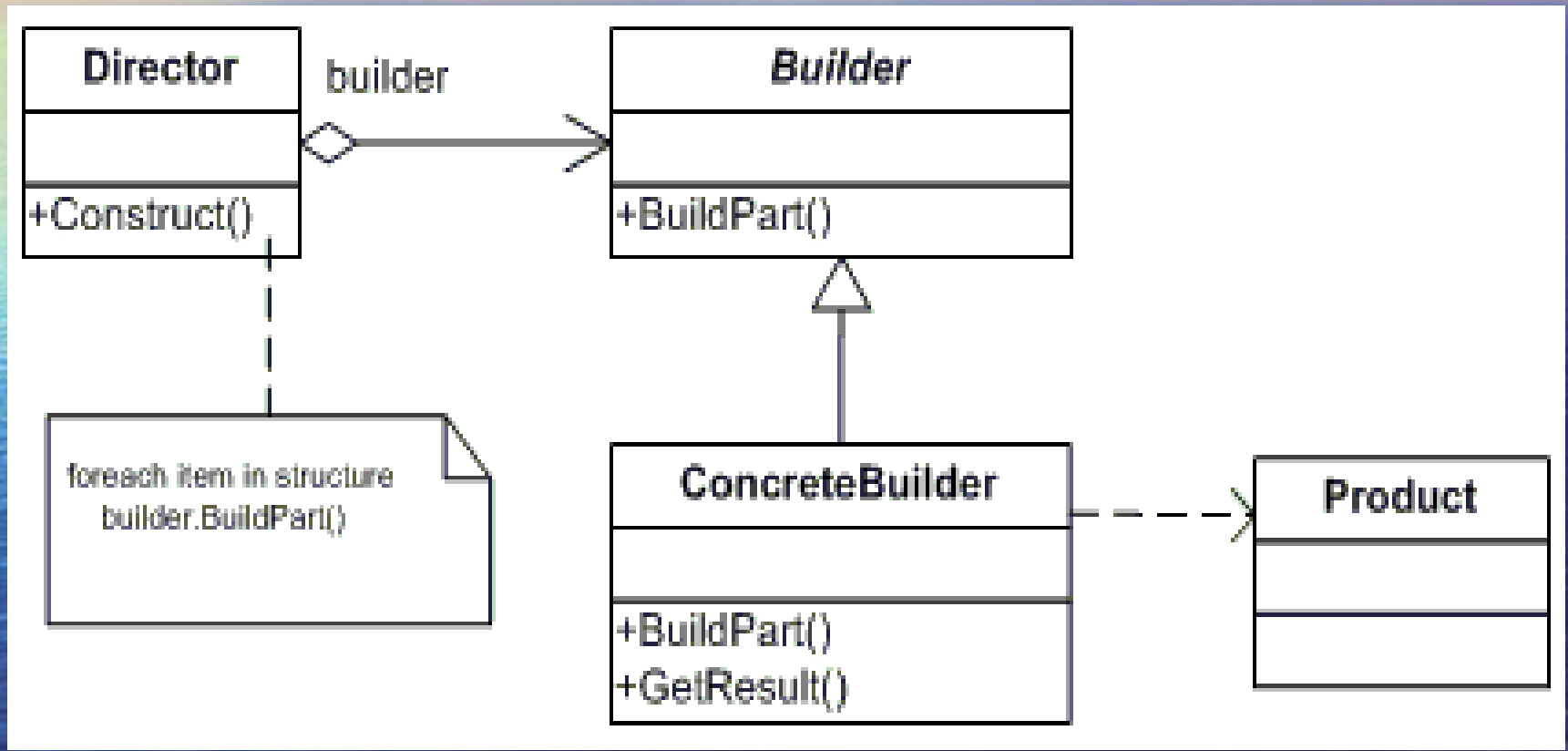
- **AbstractFactory (ContinentFactory)**
  - interfészt deklarál a műveletek számára melyeket az AbstractProduct állít elő
- **ConcreteFactory (AfricaFactory, AmericaFactory)**
  - implementálja a műveleteket egy konkrét objektum létrehozásához
- **AbstractProduct (Herbivore, Carnivore)**
  - interfészt deklarál az objektum típusai számára
- **Product (Wildebeest, Lion, Bison, Wolf)**
  - definiál egy objektumot amit a megfelelő Concrete factory hozott létre
  - implementálja az AbstractProduct interfészét
- **Client (AnimalWorld)**
  - Az abszract factory és az abstract product osztályok által deklarált interfészeket használja

Példa: állatvilág létrehozása, a létrehozott állatok különbözőek, de közüttök a kapcsolat hasonló

Creational Patterns - Builder

- Komplex objektumok létrehozását próbálja függetleníteni a konkrét adatreprezentációtól
- Elválasztja a komplex objektum felépítését, szerkezetét az ábrázolásától, így ugyanaz a konstrukció, felépítés különböző reprezentációkban jöhet létre
- Builder a komplex objektumok lépésről lépésre történő felépítésére koncentrál
- Példa: Az alkalmazásnak létre kell hoznia egy összetett gépcsoport egyes elemeit. Az egyes gépek specifikációja a második szinten van, de egyes tulajdonságaik felépítéséhez szükség van az első szint függvényeire.

Creational Patterns - Builder



Creational Patterns - Builder

- **Builder (VehicleBuilder)**
  - meghatároz egy absztrakt interfészt a Product objektum különböző részeinek létrehozásához
- **ConcreteBuilder (MotorCycleBuilder, CarBuilder, ScooterBuilder)**
  - megszerkeszti és összeállítja a termék különböző részeit a Builder interfészébe implementálva
  - meghatározza és nyomon követi a létrehozást
  - interfészt biztosít a termék visszakereséséhez, visszanyeréséhez
- **Director (Shop)**
  - összeállítja az objektumot a Builder interfészét használva
- **Product (Vehicle)**
  - az építés alatt reprezentálja az objektumot. A ConcreteBuilder létrehozza a termék belső felépítését, és definiálja azt a processzt ami összeszereli
  - tartalmazza azokat az osztályokat melyek az alkotóelemeket definiálják
  - tartalmazza azt az interfészt, ami a részek végső összeszereléséhez szükséges
- Példa: különböző járművek összeállítása lépésről lépésre

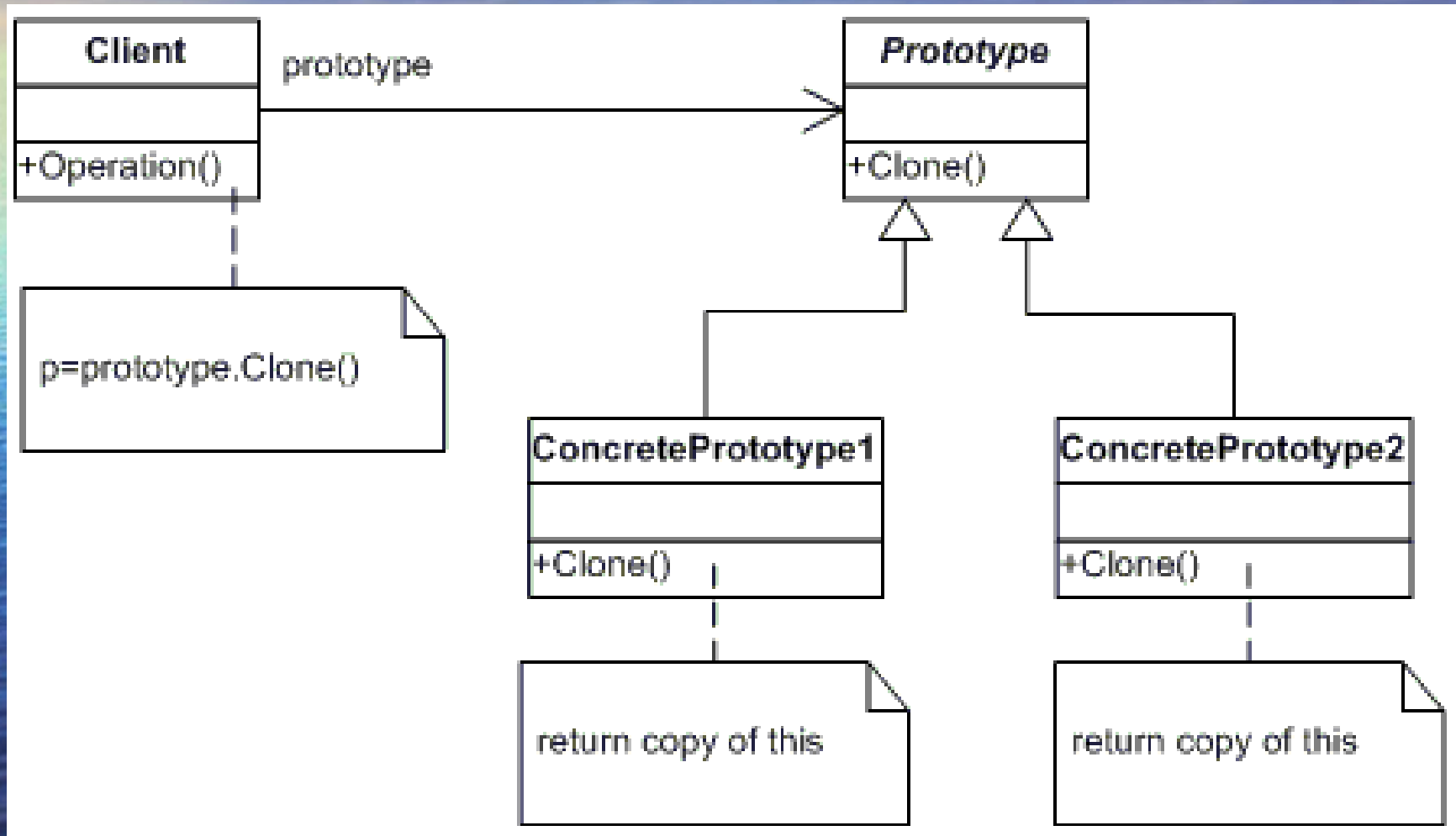
Creational Patterns – Builder Példa

- This pattern is used by fast food restaurants to construct children's meals. Children's meals typically consist of a main item, a side item, a drink, and a toy (e.g., a hamburger, fries, Coke, and toy car). Note that there can be variation in the content of the children's meal, but the construction process is the same. Whether a customer orders a hamburger, cheeseburger, or chicken, the process is the same. The employee at the counter directs the crew to assemble a main item, side item, and toy. These items are then placed in a bag. The drink is placed in a cup and remains outside of the bag. This same process is used at competing restaurants

Creational Patterns - Prototype

- Prototípus egyed definiálása és felhasználása további egyedek gyártásához
- Alkalmazás
  - Ha a rendszer független attól, hogyan jönnek létre, vannak elrendezve, ill. megjelenítve a példányai
  - Ha a példányosítandó osztályokat futási idõben specifikáljuk
  - A termékhierarchia felépítésének elkerülése az osztályhierarchia felépítésekor
  - Ha az osztály példányainak csak kevés eltérõ állapota létezik. Ekkor kényelmesebb prototípusokat bevezetni, és klónozni õket, mint mindet egyesével, a megfelelõ állapotjellemzõkkel példányosítani.

Creational Patterns - Prototype



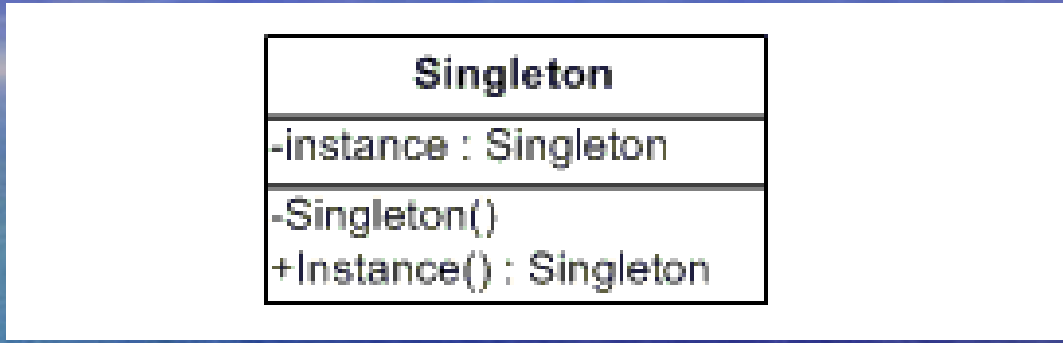
Creational Patterns - Prototype

- **Prototype (ColorPrototype)**
  - deklarál egy absztrakt osztályt saját maga klóónozására
- **ConcretePrototype (Color)**
  - műveleteket implementál saját maga klóónozására
- **Client (ColorManager)**
  -Új objektumot hoz lére megkérve a prototype-t hogy klóónozza saját magát

Creational Patterns - Singleton

- Annak biztosítása, hogy adott osztályból csak egy egyedet hozunk létre, és mindenki ezt a globális egyedet használja.
- Példa:
  - Egy nyomtatóhoz egy nyomtatási sor
  - Egy képernyőhöz egy ablak manager

Creational Patterns - Singleton



- **Singleton**
  - Definiál egy Instance műveletet. A kliens csak ehhez az egyedi Instance-hoz férhet hozzá. Ez az Instance egy osztály művelete
  - Felelős a saját egyedi Instance létrehozásáért és fenntartásáért

Creational Patterns - Singleton

Következmények:

- Ellenőrzött hozzáférés az egyetlen példányhoz
- Csökkentett namespace
- Lehetőség korlátozott számú példányosításra

Structural Pattern - Adapter

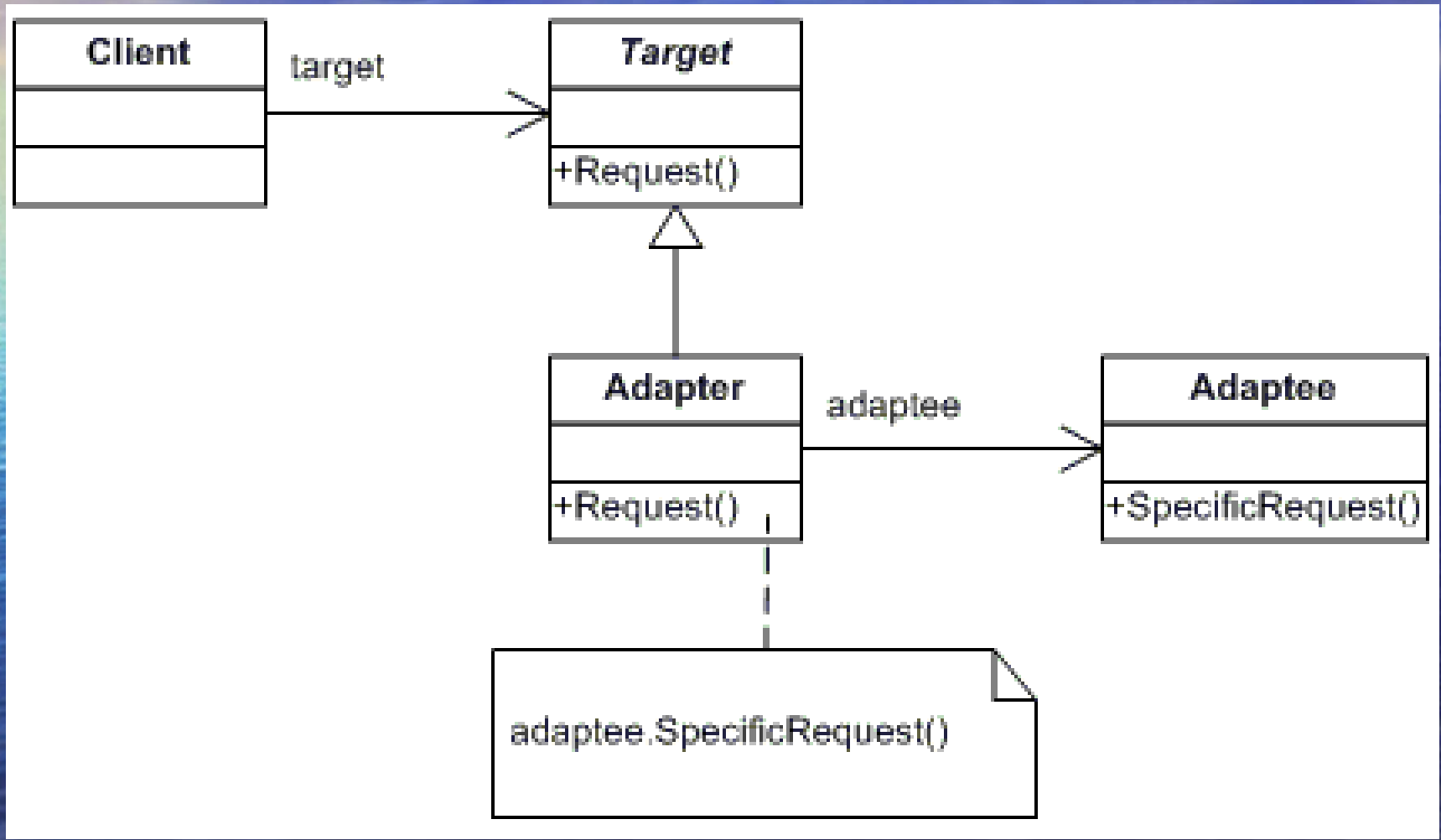
- Adott objektum interfészének konvertálása/adaptálása a befogadó környezet által igényelt interfészre.

- Cél:
  - Átalakítani az osztály interfészét olyanná, amilyet a kliens vár.
  - Az Adapter segítségével olyan osztályok is együtt tudnak működni, amelyek a különböző interfészeik miatt nem tudnának.

- Példa:
  - Távoli Web szolgáltatások elérése

- Következmények:
  - Közvetítés az osztályok között
  - Az Adapter felüldefiniálhatja az Adaptee viselkedését

Structural Pattern - Adapter



Structural Pattern - Adapter

- **Target**
  - Definiál egy domain specifikus interfészt, amit a Client tud használni
- **Adapter**
  - Az Adaptee interfészét a Target interfészhez igazítja
- **Adaptee**
  - Definiálja a létező interfészt ami a hozziellesztáshez kell
- **Client**
  - Együttműködik a Target intefész objektumaival

Példa: Kémiai adatbázis, az objektumok az kémiai adatbázishoz egy interfészen keresztül férhetnek hozzá

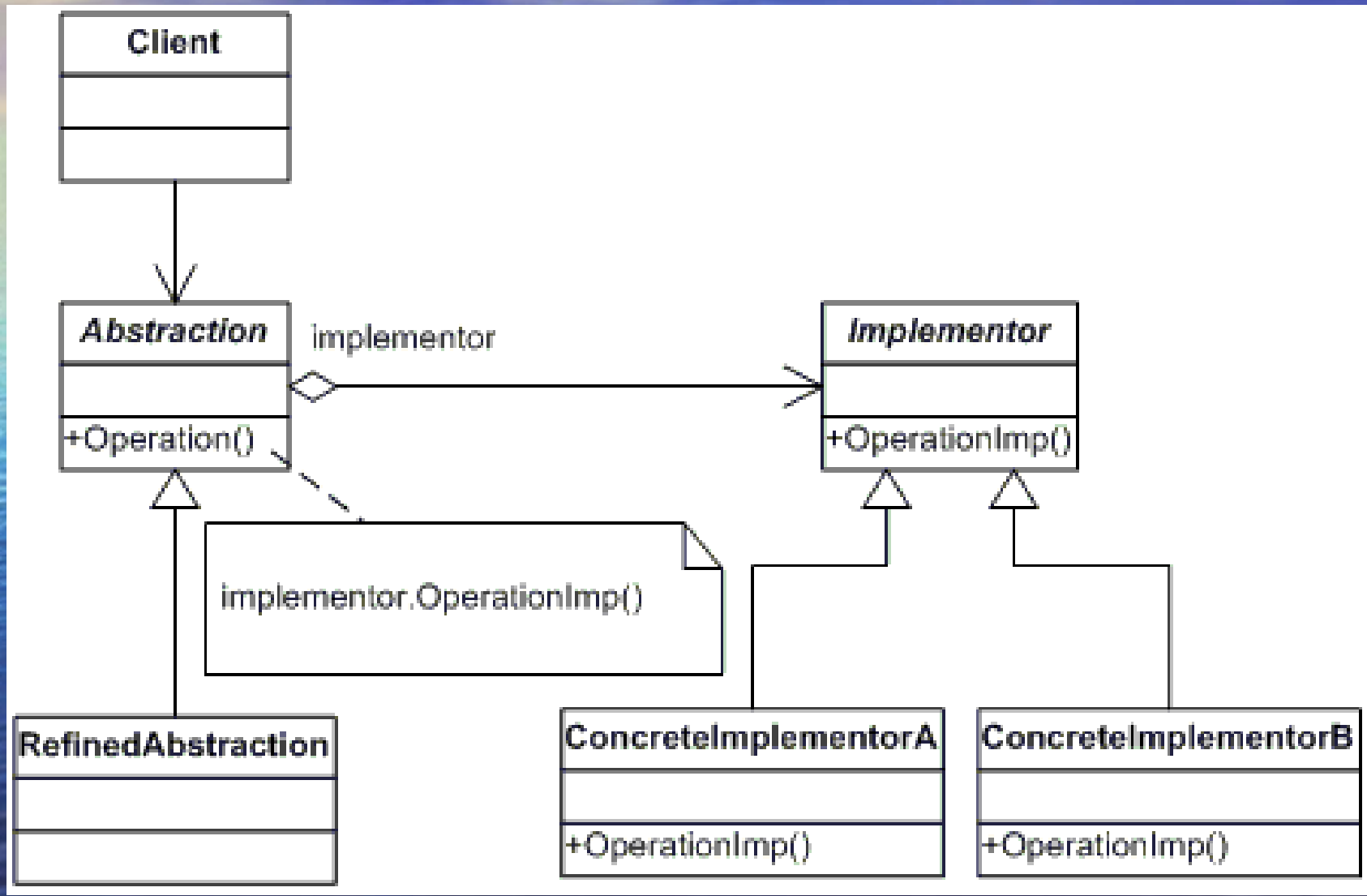
Structural Pattern – Bridge

- Az absztrakt (ős)osztály és a konkrét megvalósítások elválasztása (mivel néhány esetben az öröklődés túl erős köteléket jelenthet).
- A minta alkalmazásának célja, hogy az absztrakció és az implementáció különváljon.
- Ez a következő esetekben lehet előnyös:
  - El akarjuk kerülni a végleges kötést (futás közben cserélhető implementáció).
  - A program két részre bontható és külön fejleszthető
  - Mind az absztrakció, mind az implementáció külön örökíthető, kapcsolatuk felbomlása nélkül.

Bridge - Példa

Ha például több platformon akarunk egy ablakkezelőt megvalósítani, akkor a különböző típusú ablakokat (pl. FőAblak, PárbeszédAblak, stb.) nem kell minden platformra megírni. Elég egy implementációs interfész, melyet megvalósítunk minden platformra egy alosztályban. A különböző típusú ablakok az implementáció interfész-függvényeit használják.

Structural Pattern – Bridge



Structural Pattern – Bridge

- **Abstraction (BusinessObject)**
  - Definiálja az absztrakt interfészt
  - Implementor típusú pbjektumhoz létrehoz egy hivatkozást
- **RefinedAbstraction (CustomersBusinessObject)**
  - Az Abstraction által definiált interfészt kiterjeszti
- **Implementor (DataObject)**
  - Az implementation osztályhoz definiál egy interfészt. Ennek ez interfésznek nem kell közvetlenül együttműködnie az Abstraction interfészével, vagyis két teljesen különböző interfész lehetséges. Az implementor interfésze csak az egyszerű műveletekez adja, az Abstraction interfésze adja a magasabb szintűeket és ezek az előzökön alapszanak
- **ConcreteImplementor (CustomersDataObject)**
  - Implementája az Implementor interfészét, és definiálja ennek a valós implenetációját

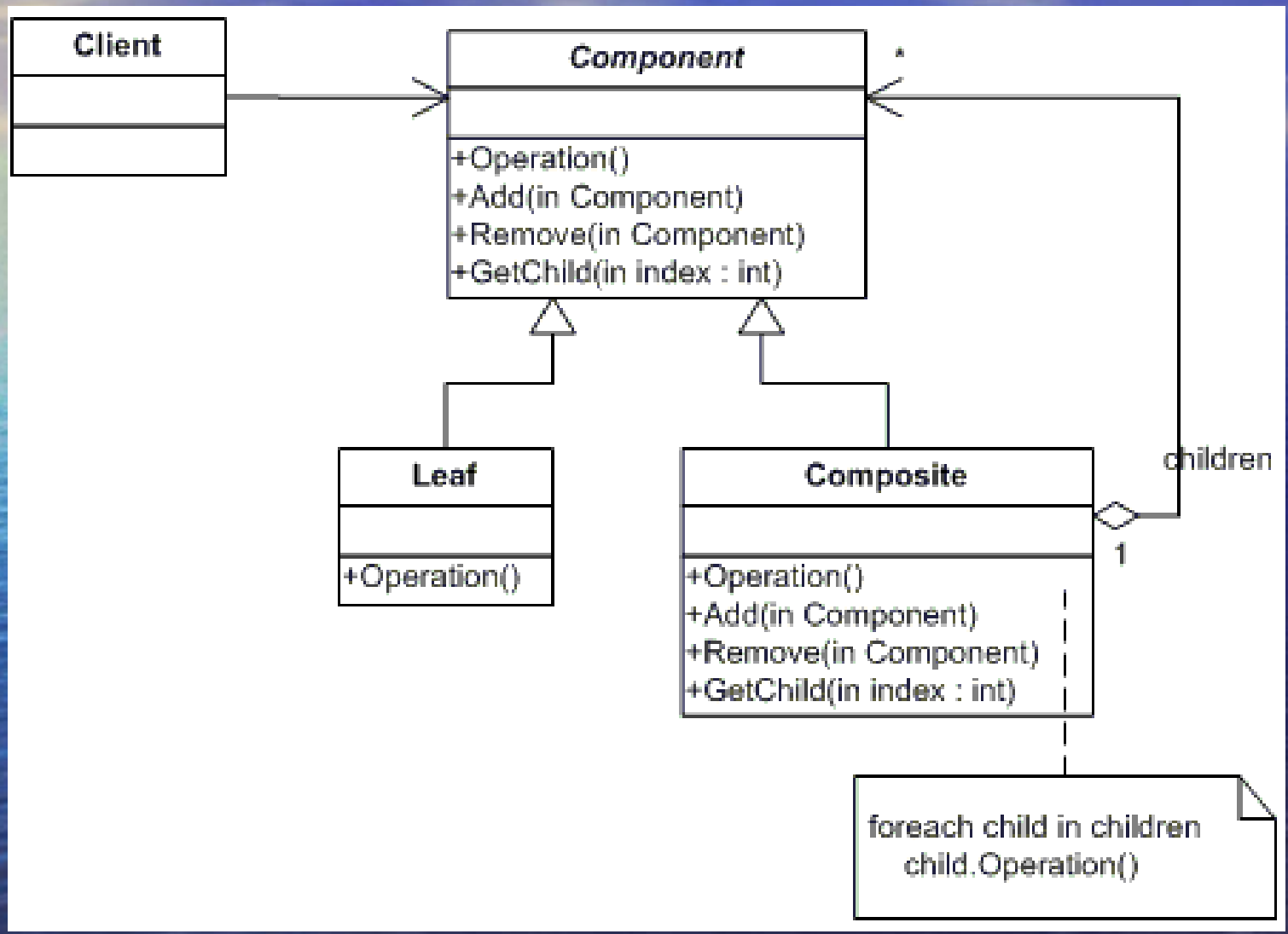
Structural Pattern - Composite

- Úgy szervezi hierarchikus struktúrába az együttműködő objektumokat, hogy azok a külvilág számára egy (virtuális) objektumként kezelhetők legyenek.
- ***Cél:***
  - Az objektumokat a rész-egész kapcsolatokat definiáló fastruktúrába szervezzük. Ebben az esetben a *Composite* tervezési minta lehetõséget ad arra, hogy a kliens mind az önálló, mind az összetett objektumokat egyformán, teljesen átlátszóan kezelje.

Structural Pattern - Composite

- ***Motiváció:***
  - A különböző grafikai alkalmazások a felhasználó számára lehetővé teszik, hogy egyszerű komponensekből bonyolultabbakat építsünk, és azokat egységesen kezeljük. Így például egy egyszerű szöveg- és vonalkomponenst csoportosíthatunk és azokat, mint egy komponenst, kezelhetjük.
    Azonban a felhasználó programnak meg kell különböztetnie őket, mint egyszerűeket illetve összetetteket, ugyanis csak összetetthez lehet új elemeket adni, vagy elvenni, egyszerűeknél e műveletek nem értelmezettek.
    Mindezek mellett vannak olyan műveletek, melyeknél ez a különbség nem játszik szerepet, és a komponensek egyformán kezelhetőek.

Structural Pattern - Composite



Structural Pattern - Composite

- *Component:*
  - Deklarálja azt az alapinterfészt, mellyel elérhetõek lesznek a minden komponensen elvégezhetõ mûveletek.
  - Implementálja ezeket a mûveleteket az alapvetõ mûködéshez
  - Interfészt ad a gyermekobjektumok eléréséhez és kezeléséhez.
  - Interfészt ad a komponens szûlõjének eléréséhez, és implementálhatja azt, ha az helyénvaló (opcionális).
- *Leaf:*
  - Az egyszerû komponenseket testesíti meg, azaz azokat, melyeknek nincsenek gyermekei.
  - A primitív komponensek alapvetõ mûködését, viselkedését definiálja.
  - Implementálja ezen mûveleteket, ha a *Component*-ben ez nem történt meg.
- *Composite:*
  - Az összetett objektumok viselkedését definiálja és implementálja.
  - Tárolja a gyermek komponenseket.
  - Implementálja a gyermekek eléréséhez és kezeléséhez a *Component*-ben definiált interfészt.
- *Client:*
  - A *Component*-ben definiált interfészen keresztül manipulál a komponensekkel.

Structural Pattern - Composite

- ***Mûködés:***
  - A kliens a *Component*-ben definiált interfészen keresztül kezeli a komponenseket. Ha a kérés egy *Leaf*-hez érkezik, akkor az közvetlenül kezeli le azt, míg egy *Composite* továbbitja azokat a gyermekeihez, elõtte és utána esetleg további mûveleteket végezve.

- ***Következtetések:***
  - A tervezési minta olyan osztályhierarchiát definiál, melyben a komponensek bonyolultabbakká tehetõk össze rekurzívan.
  - Amikor egy kliens egy egyszerû objektumot vár, összetettet is kaphat.
  - A kliens egyszerûbbé válik, mivel normális esetben nem kell megkülönböztetnie az egyszerû komponenst az öszetettõl.
  - Új komponensek hozzáadásával a klienst nem kell megváltoztatni.
  - A tervezést általánosabbá teszi.

Structural Pattern - Decorator

- Az objektum viselkedésének kiterjesztése dinamikus módon (nem osztály szinten öröklődéssel).
- A minta célja meglévő objektumokhoz új feladatok rendelése futás közben, dinamikus módon, ahelyett, hogy e feladatokat statikus módon az objektum osztályához rendelnénk.
- Ez a módszer megfelelő alternatívát nyújt az abban az esetben, ha az objektum az osztályában definiált alapvető feladatain kívük különféle kontextusokban nagy számú egyéb feladat, illetve ezen feladatok kölönféle kombinációja is hárul, és ezen feladatokat kényelmetlen, egyes esetekben áttekinthetetlen lenne az osztályhierarchia szintjén, származtatott osztályokkal megvalósítani.

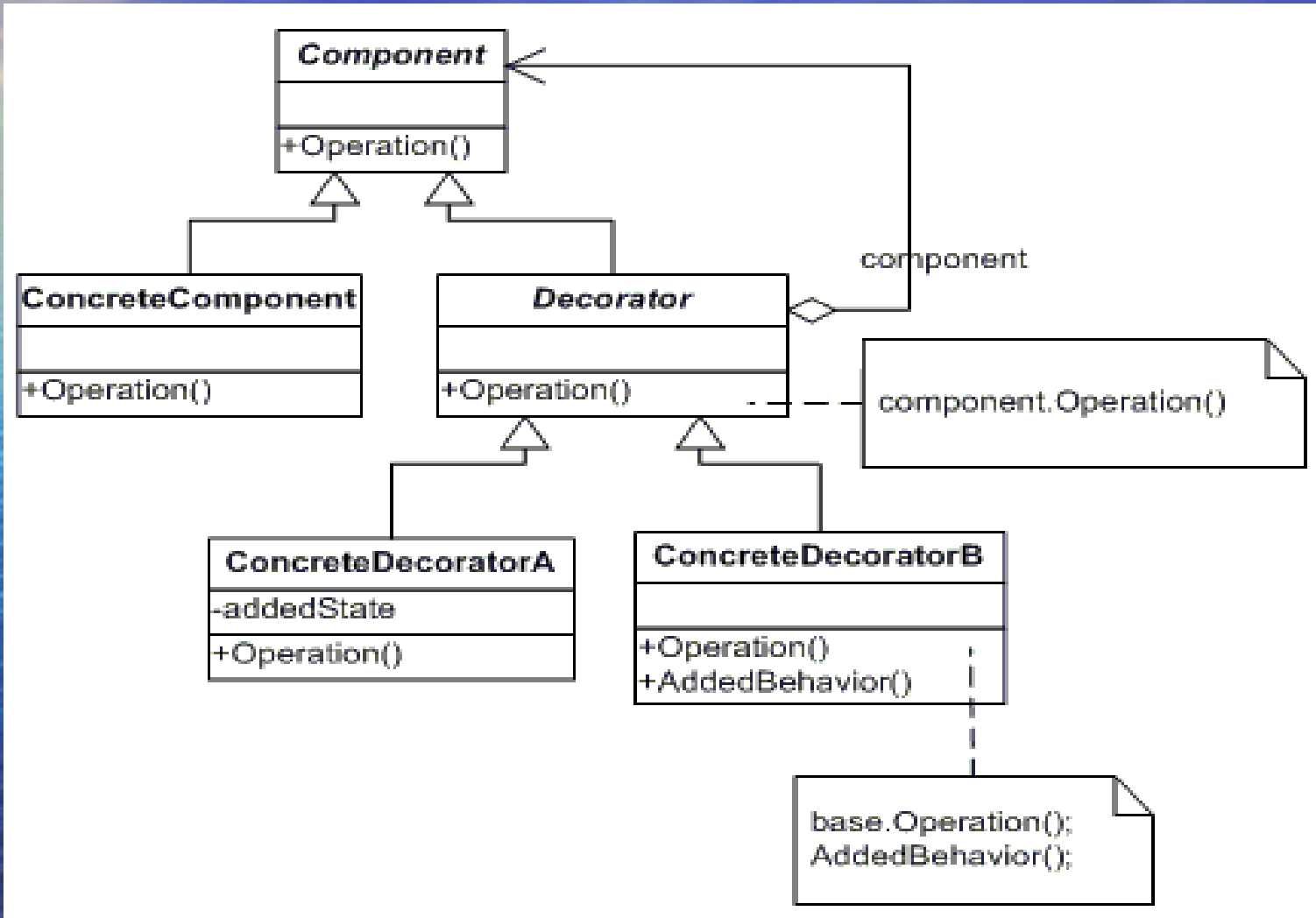
Structural Pattern - Decorator

- megvalósítás kulcsa az eredeti objektum beágyazása egy olyan objektum belsejébe, amely megvalósítja a kívánt pluszfeladatokat; ez a beágyazás tetszőleges mélységig ismételhető. Ehhez szükség van egy felület definiálására, amelyet mind az eredeti objektum, mind a pluszfunkciókat megvalósító objektumok megvalósítanak, így az osztály kívülről egységesen kezelhető.
- Ez a módszer - az Adapter mintával ellentétben - nem az objektum felületét, hanem feladatait változtatja meg. Minden objektum különféle feladatokat valósíthat meg az egyes metódusok meghívásakor, azonban maga a felület minden implementációban változatlan.
- Problémát jelenthet azonban, hogy a beágyazott objektum teljesen el van rejtve, így a direkt hozzáférés hiánya komoly gondot jelenthet egyes esetekben.

Structural Pattern - Decorator

- **Példa**
  - Tegyük fel, hogy felhasználói felület elemeinek a tervezése a feldataunk, amelyben bármely elemhez rendelhető keret, valamint vízszintes és függőleges görgetősáv, az elemek nagy száma miatt azonban nem célszerű az a módszer, hogy mindegyikhez elkészítjük a fent említett plusz szolgáltatásokat is kezelő leszármaztatott osztályt. Az alkalmazott megoldás lehet az, hogy definiálunk egy általános keret illetve görgetősáv osztályt, amelybe beágyazzuk a felhasználói felület megfelelő elemét reprezentáló objektum.

Structural Pattern - Decorator



Structural Pattern - Decorator

- **Component (LibraryItem)**
  - Az objektumok számára definiál egy interfészt, amit dinamikusan lehet a gyerek objektumokhoz hozzáadni
- **ConcreteComponent (Book, Video)**
  - Definiál egy objektumot, melyhez az előző gyerekekhez köthető
- **Decorator (Decorator)**
  - Fentart egy hivatkozást a Component objektumhoz, és definiál egy interfészt, ami alkalmazkodik, illeszkedik a Component interfészéhez
- **ConcreteDecorator (Borrowable)**
  - A Component-hez hozzáadja a gyerek egységeket

Structural Pattern - Façade

- Egységes (magasabb szintű) interfész kialakítása több interfész egységes használatára
- **A probléma:**
  - Adott egy alrendszer, sok interface-szel, és sok osztállyal. Hogyan engedjük a usert hozzáférni az alrendszerhez, anélkül, hogy ismerné az összes osztály felépítését, metódusait, attribútumait? Erre jelent megoldást a Facade.

Structural Pattern - Façade

- A megoldás:
  - Egy alrendszerhez a kliensek a Facade-on keresztül férnek hozzá, tehát a Facade tulajdonképpen egy magas szintû interface-t nyújt a kliensek részére. Így a klienseknek nem kell ismerniük az alrendszer minden egyes osztályának a belsõ felépítését, elég, ha a Facade által nyújtott szolgáltatásokat (metódusokat) ismerik. Tehát a kliensek számára az alrendszer láthatatlan. Egy összetett, bonyolult rendszerben is csökkenti a kliensek kapcsolatainak számát, így megnöveli a hordozhatóságot és a függetlenséget.

Structural Pattern - Façade

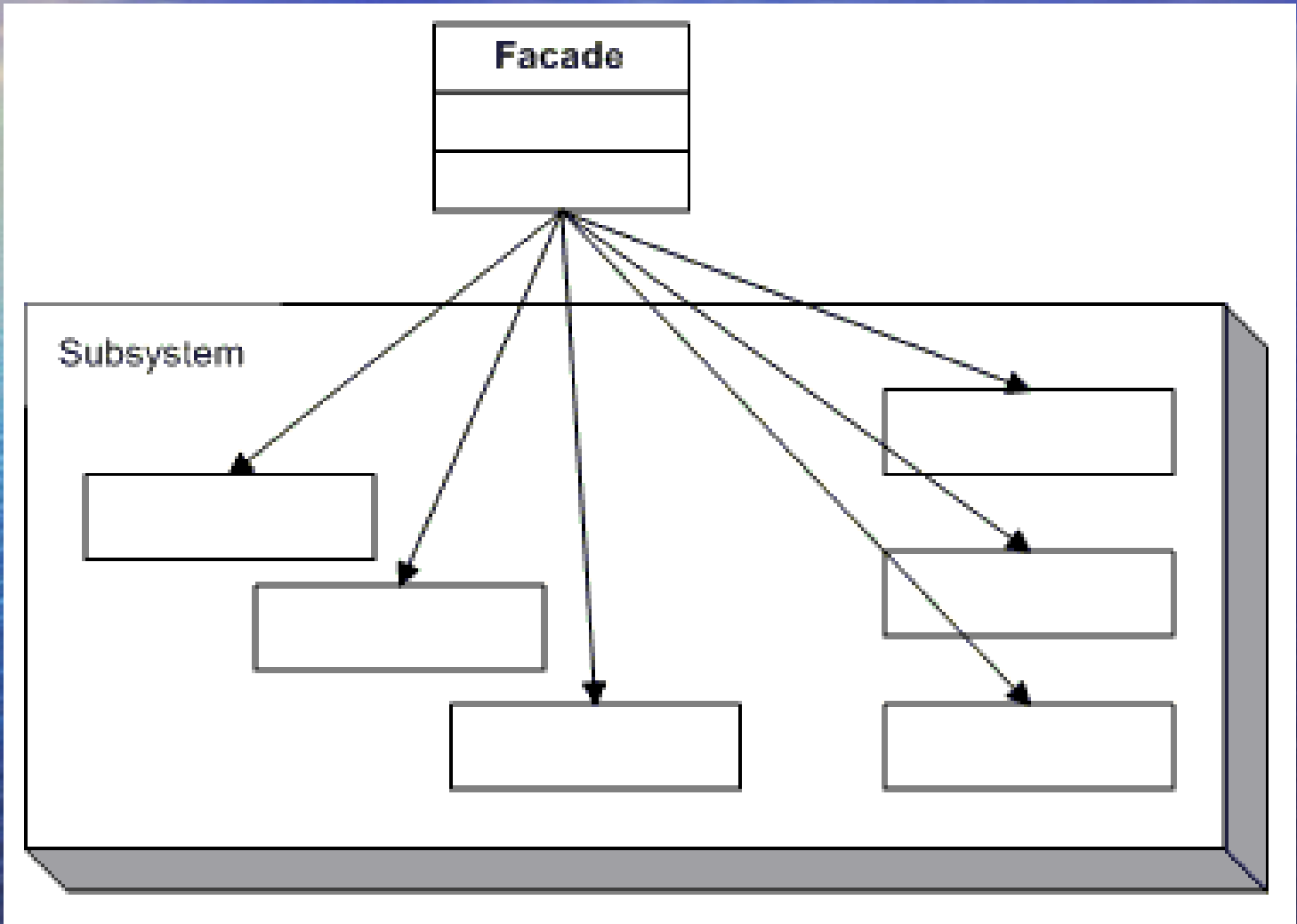
- Tehát a megvalósítás csökkentheti a kliens és az alrendszerek közötti kapcsolatokat.
- A kevesebb kapcsolat a rendszerek között nagyobb újrafelhasználhatóságot eredményez, aminek biztosítása napjainkban fontos feladat a szoftverfejlesztésben.
- Megoldható az is a Facade segítségével, hogy az alrendszerek közötti kommunikációt, - ha ez szükséges - kizárólag a Facade-on keresztül engedjük meg, ezzel erõteljesen csökkenthetõ az alrendszerek közötti függõséget.

Structural Pattern - Façade

- Példa:

  Van egy autótulajdonos, akinek van egy autója. Mivel õ nem ért az autóhoz, így ha valami probléma van az autóval, például kilyukad a gumija, akkor a szerelõjéhez fordul, hogy cserélje le a defektes gumit. Tehát az autótulajdonos nem nyúl hozzá az autójához közvetlenül, csak a szerelõjét kéri meg a javítási illetve karbantartási munkálatokra.

Structural Pattern - Façade



Structural Pattern - Façade

- A rendszer résztvevõi:
  - Kliens:

    - Meghívja a Facade metódusait, rajta keresztül fér az alrendszerekhez.
  - Facade:

    - Kezeli a kliensek kéréseit.
    - Tudja, melyik kéréssel, melyik objektumhoz kell fordulni.
  - Alrendszer osztályai:

    - A Facade számukra átlátszó, nem tudnak a létezésérõl.
    - Õk valósítják meg a Facade kliens felé nyújtott funkcionalitását

Structural Pattern - Proxy

- Objektumhoz történő hozzáférés szabályozása burkoló objektummal
- Cél:
  - Egy objektumhoz helyettesítést adni, annak érdekében, hogy a hozzáférés vezérelhető legyen.
- Példák:
  - Web szolgáltatások elérése, erőforrások helyettesítése, hozzáférési jogok szabályozása

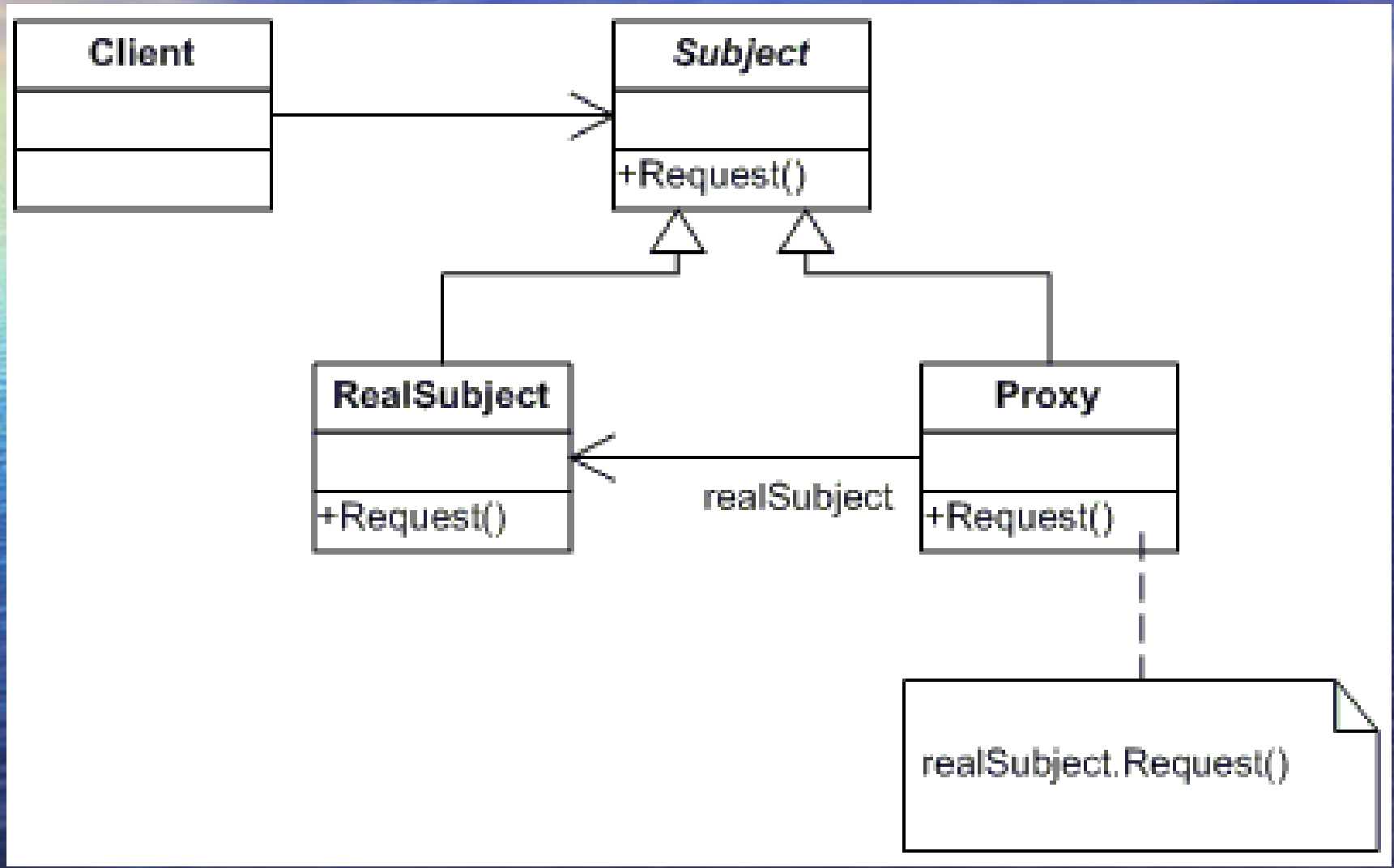
Structural Pattern - Proxy

Következmények:

Indirekció egy fajtája objektum elérés esetében. Számos felhasználási lehetőség ismert a proxy típusától függően:

1. **Távoli proxy** elrejti a tényt, hogy az objektum egy másik címtartományban található.
2. A **virtuális proxy** optimalizálási feladatokat láthat el, mint pl. létrehozás igény esetén (creating an object on demand).
3. További karbantartási feladatok elvégzésére nyílik lehetőség az objektum elérése körül.

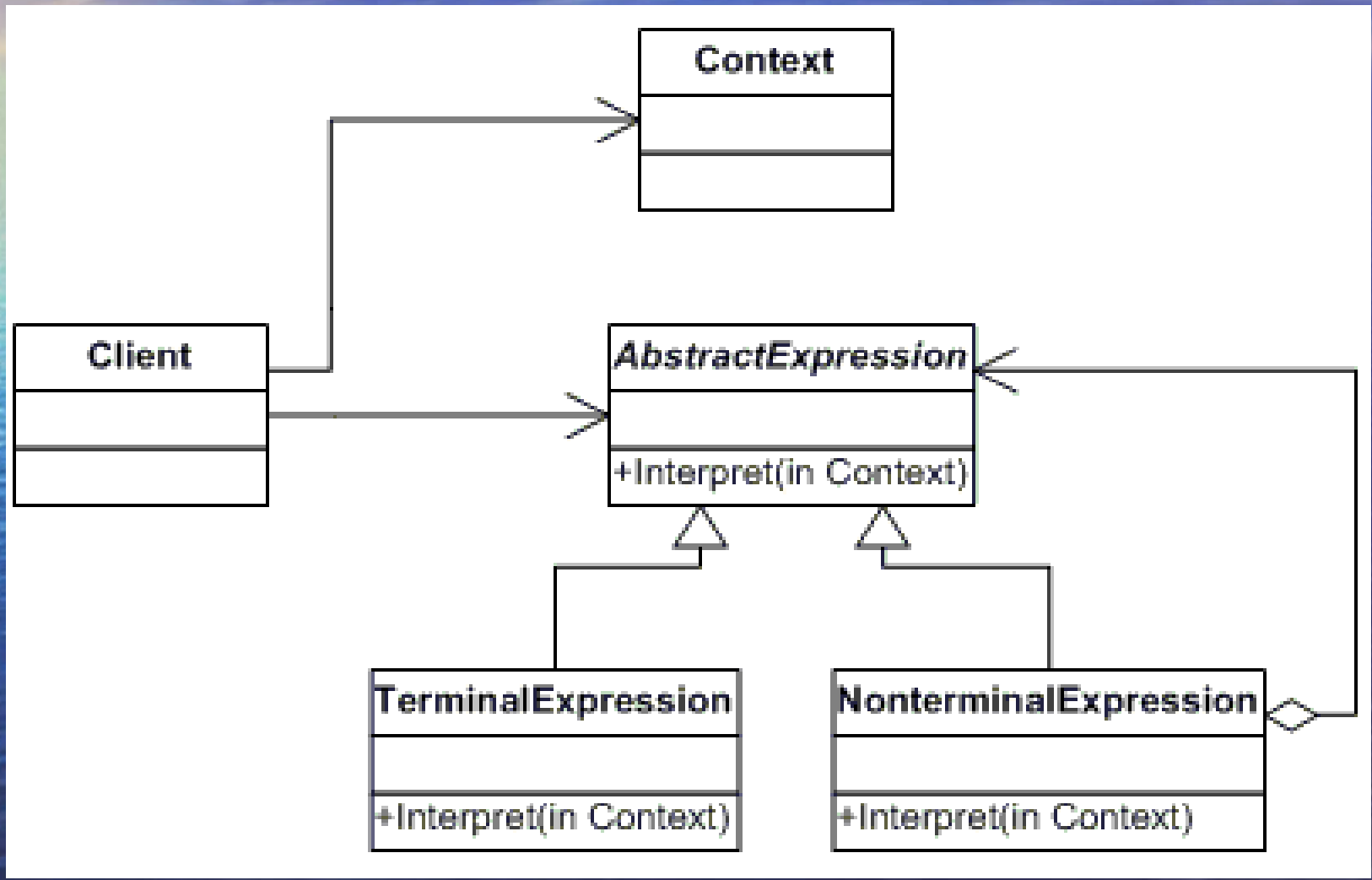
Structural Pattern - Proxy



Behavioral Patterns - Interpreter

- Adott nyelvtan reprezentációja objektumorientált eszközökkel, ill. a nyelvtanhoz kapcsolódó értelmező (interpreter) megvalósítása. Osztály szintű
- Ritkán használatos
- Péld
  - Római számról átalakító program

Behavioral Patterns - Interpreter



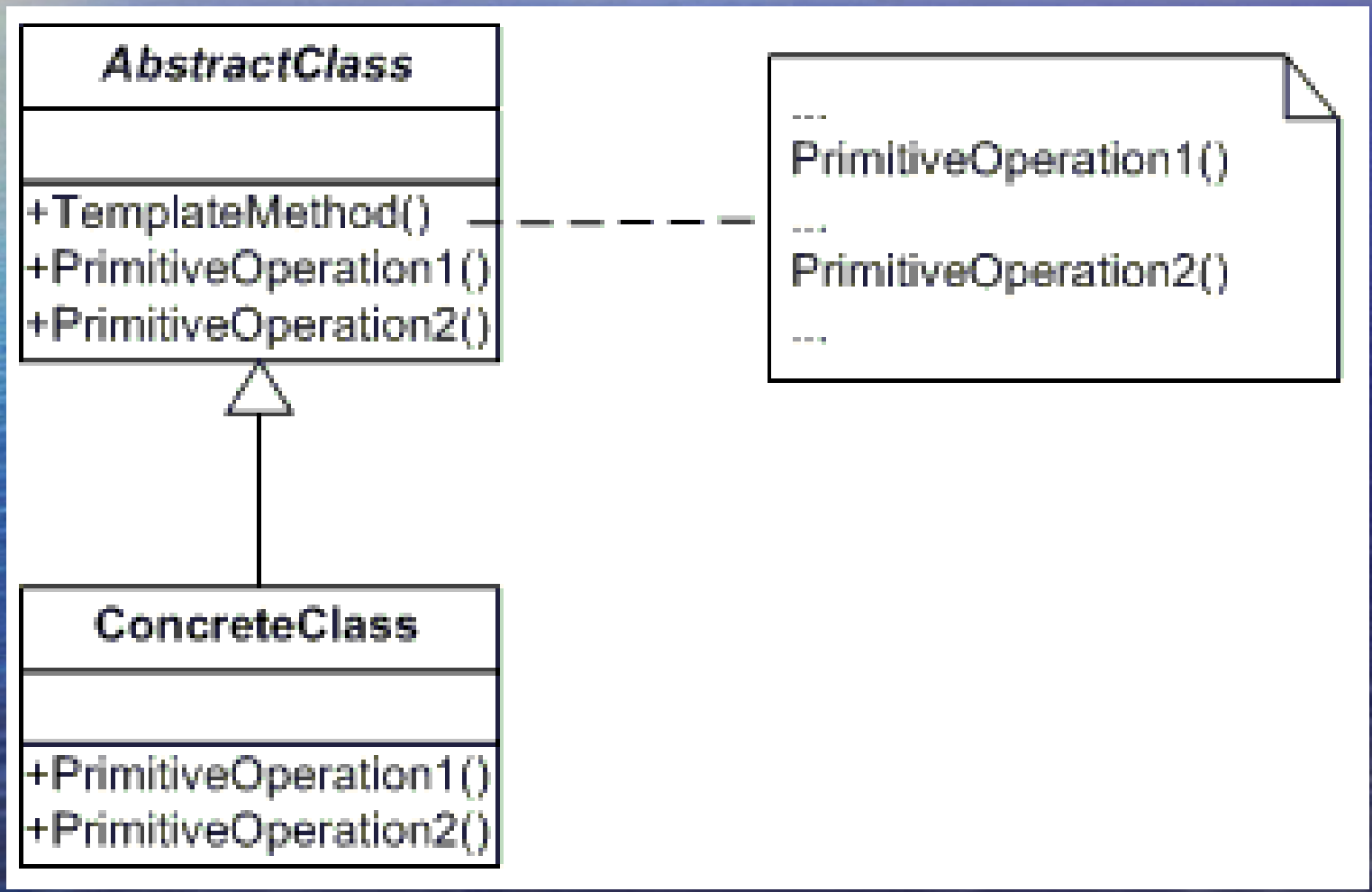
Behavioral Patterns Template Method

- Algoritmus vázának megvalósítása, objektumba zárása oly módon, hogy bizonyos közbenső végrehajtási lépések specializálhatók legyenek származtatott osztályokban.
- Cél:
  - Definiáljuk egy algoritmus vázát, de a részleteket hagyjuk, hogy az alosztályok helyettesítsék be.
  - A Template Method segítségével a leszármazott osztályok újradefiniálhatják az algoritmus bizonyos lépéseit anélkül, hogy az algoritmus struktúráját megváltoztatnák.
- Példa:
  - Különböző dokumentumfajták megnyitása

Behavioral Patterns Template Method

- Következmények:
  - Template metódus alkalmazása alapvető technika a kód újrafelhasználás terén.
  - Segítségükkel könnyen kigyűjthető a közös viselkedés osztály könyvtárakba.

Behavioral Patterns
Template Method



Behaioral Patterns - Strategy

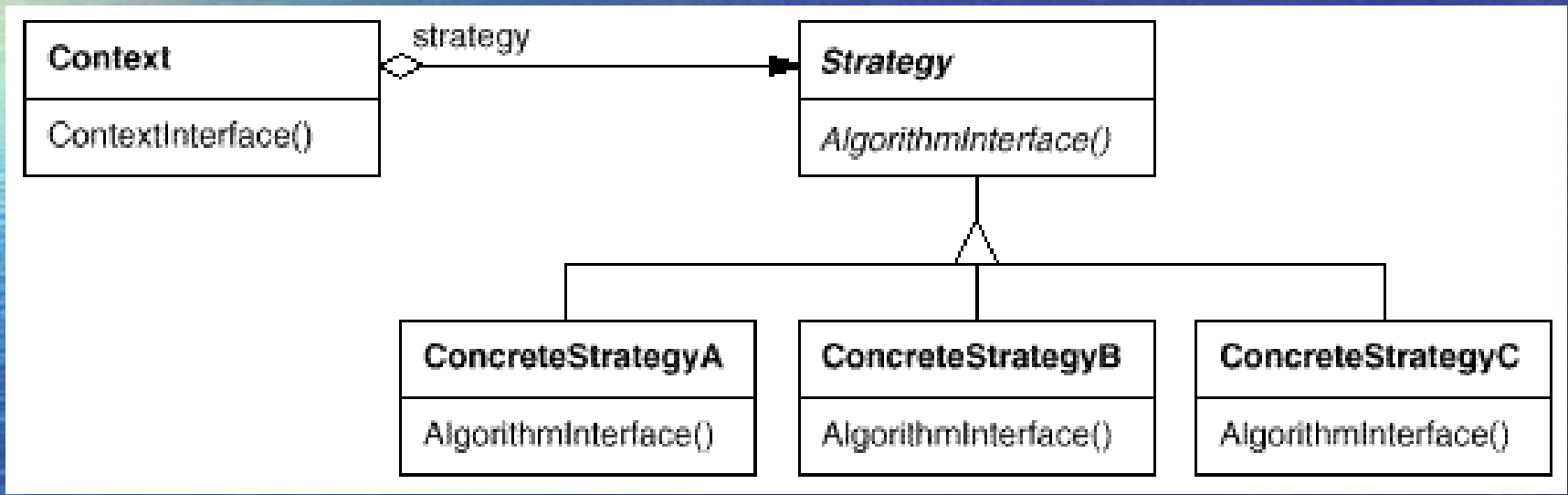
- Különböző algoritmusok objektumba zárása közös interfésszel, a megfelelő algoritmus kiválasztása polimorfizmus felhasználásával.

- Cél:
  - Definiáljuk algoritmusok egy családját, zárjuk őket egységbe és tegyük őket egymással kicserélhetővé.
  - A Strategy lehetővé teszi, hogy az algoritmus az őt használó kliensektől függetlenül változhasson.

Behaioral Patterns - Strategy

- Előnyök:
  - Feltételes utasítások elkerülése a megfelelő viselkedés kiválasztása esetében
  - Azonos viselkedés különböző implementációi
- Hátrányok:
  - A kliensnek ismernie kell a különböző stratégiákat
  - Kommunikációs overhead a Strategy és a Context között

Behaioral Patterns - Strategy



Behaioral Patterns - Strategy

- **Strategy (SortStrategy)**
  - Deklarál egy közös interfészt a összes támogatott algoritmusnak. A Context ezen interfészen keresztül tudja használni a ConcreteStrategy által definiált algoritmusokat
- **ConcreteStrategy (QuickSort, ShellSort, MergeSort)**
  - A Strategy interfészét használva implementálja az algoritmust
- **Context (SortedList)**
  - A ConcreteStrategy objektuma konfigurálja
  - Létrehoz egy hivatkozást a Strategy objektumaira
  - Interfészt definiálhat, melyen keresztül a Strategy hozzágérhet a tárolt adataihoz
- Példa: Rendezés, a kliens számára lehetűvé teszi, hogy dinamikusan válasszon a rendezési stratégiák között

Behaioral Patterns - Chain of Responsibility

- Az üzenet vagy kérés küldőjét függetleníti a kezelő objektum(ok)tól.

- Chain of Responsibility használható, ha:
  - több, mint egy objektum kezelhet le egy kérést és a kérést lekezelõ példány alapból (eleve) nem ismert, automatikusan kell megállapítani, hogy melyik objektum legyen az.
  - egy kérést objektumok egy csoportjából egy objektumnak akarjuk címezni, a fogadó objektum konkrét megnevezése nélkül.
  - egy kérést lekezelõ objektumok csoportja dinamikusan jelölhetõ ki.

Behaioral Patterns Chain of Responsibility

- **Motiváció:**
  - Tekintsük egy grafikus felhasználói felület környezet érzékeny help rendszerét. A felhasználó a felület bármely részérõl kaphat help információt, elég ha rákattint. A help, amit a rendszer szolgáltat nem csak attól függ, hogy a felhasználó mit választott ki, hanem a kiválasztott felületelem környezetétõl is. Például egy dialógusablakban elhelyezkedõ nyomógomb más help információt tartalmazhat, mint egy hasonló gomb a fõablakban. Ha nincs a kiválasztott felületelemnek specifikus help információja, akkor a help rendszer egy általánosabb help üzenetet jelenít meg a felületelem közvetlen környezetérõl. (Például a dialógusablakról.)

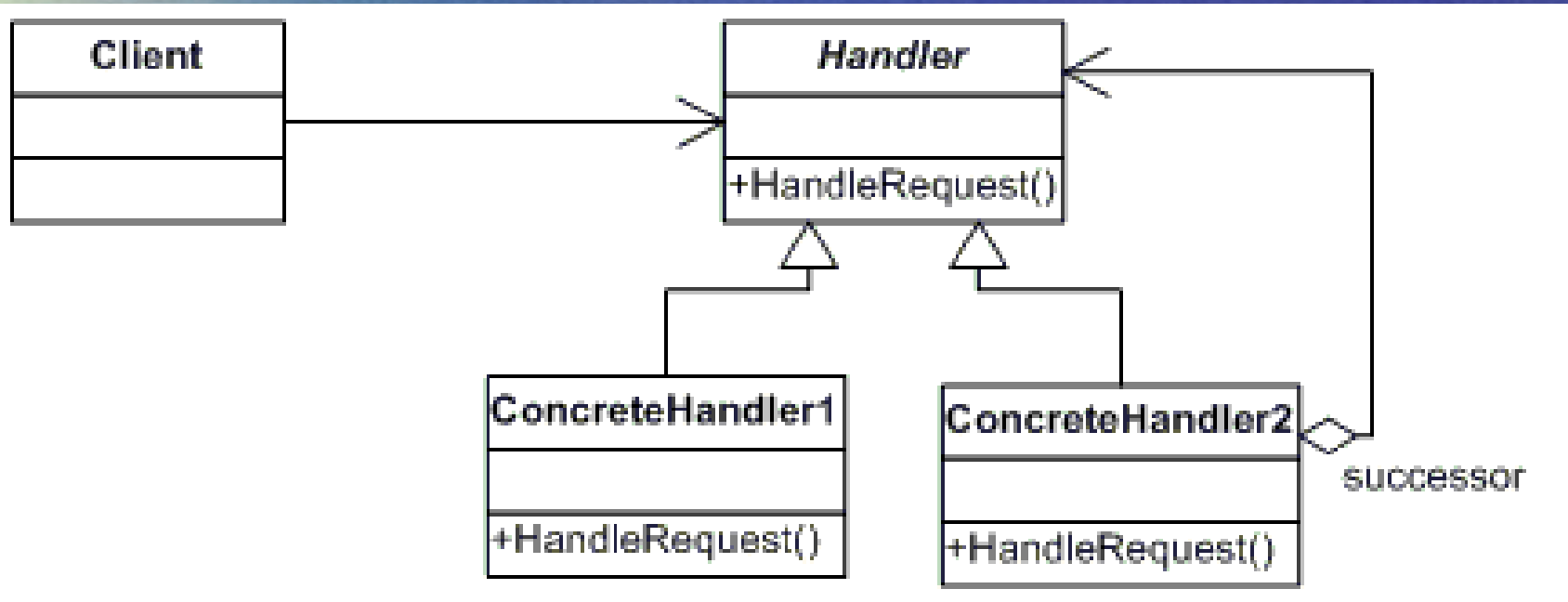
Behaioral Patterns Chain of Responsibility

- Az természetes, hogy a help információk szervezettséget tükröznek a legspecifikusabbtól a legáltalánosabbig. Továbbá az is világos, hogy a help kérést a felhasználói felület egyik objektuma kezeli le. Az hogy melyik, az a környezettõl és a rendelkezésre álló help specifikusságától függ.
  A probléma lényege, hogy az az objektum, amelyik végül kiszolgálja a help kérést nem konkrétan ismert azon objektum számára, amelyik a kérést elindította (a példában a gomb). Tehát módot kell találni arra, hogy elválasszuk a gombot, ami a kérést elindítja azoktól az objektumoktól, amik a help információt szolgáltathatják. A Chain of Responsibility tervezési minta ennek a módját írja le.

Behaioral Patterns Chain of Responsibility

- A tervezési minta alapgondolata az, hogy szétválasztja a kérést küldõket a fogadóktól azáltal, hogy több objektumnak adja meg a lehetõséget a kérés lekezelésére. A kérés egy objektumokból álló lánc szemeit járja be egészen addig, amíg valamelyik láncszem le nem kezeli.

Behaioral Patterns - Chain of Responsibility



Behaioral Patterns - Chain of Responsibility

- Handler (HelpHandler)
  - egy interfészt definiál a kérések lekezelésére opcionálisan implementálja a láncban következõ objektumhoz való kapcsolatot
- ConcreteHandler (PrintButton, PrintDialog)
  - lekezeli azokat a kéréseket, amelyek lekezeléséért felelõs
    el tudja érni a láncban következõ objektumot
    ha a ConcreteHandler le tudja kezelni a kérést, akkor le is kezeli, ellenkezõ esetben továbbadja a láncban õt követõ objektumnak
- Client
  - elindítja a kérést a láncban a ConcreteHandler objektumnál
- **Együttmûködések:**
  - Amikor a kliens elindít egy kérést, a kérés addig terjed tovább a láncon, amíg egy olyan ConcreteHandlerhez ér, amelyiknek felelõssége lekezelni azt.

Behaioral Patterns - Chain of Responsibility

- A Chain of Responsibility a következõ elõnyökkel és kötelezettségekkel/tartozásokkal bír:
  - **1. Kisebb mértékû párosítás:**
    A tervezési minta használata felszabadítja az objektumot attól, hogy ismernie kelljen a kérését lekezelõ objektumot. Egy objektumnak csak azt kell tudnia, hogy a kérést megfelelõen le fogják kezelni. Sem a kérést lekezelõ, sem a kérést küldõ nem ismeri konkrétan egymást, és a lánc egyetlen objektumának sem kell ismernie a lánc szerkezetét.
    A Chain of Responsibility tervezési minta leegyszerûsítheti az objektumok kapcsolatrendszerét. Ahelyett, hogy az egyes objektumok minden egyes lehetséges olyan objektumra rendelkeznének referenciával, ami a kéréseiket kezeli le, elég, ha egyetlen referenciával rendelkeznek a láncban õket követõrõl.

Behaioral Patterns - Chain of Responsibility

- **2. Nagyobb rugalmasság az objektumok felelõsségeinek kiosztásában:**
  Chain of Responsibility nagyobb rugalmasságot nyújt az objektumok közötti felelõsségek elosztásában. Egy esemény lekezelésére felelõsség adható az objektum láncba rakásával, illetve a lánc futási idejû megváltoztatásával. Ez alosztályokkal is kombinálható a lekezelõk statikus specializációja érdekében.

- **3. A recept nem garantált:**
  Mivel a kérésnek nincs konkrét fogadója, nincs is garancia arra, hogy le lesz kezelve, a kérés kiérhet a láncból a végén anélkül, hogy lekezelték volna. A kérés akkor is lekezeletlenül maradhat, ha a lánc nem lett jól bekonfigurálva.

Behaioral Patterns - Chain of Responsibility

- A következõ sarkallatos kérdéseket kell megfontolni az implementáláskor:
  - **1. A lánc implementálása.** Két lehetõség kínálkozik rá:
    - Új kapcsolatok definiálása. (általában a Handler-ben, de a ConcreteHandlers is definiálhatja helyette)
    - Meglévõ kapcsolatok használata
  - **2. A lánc egymást követõ elemeinek összekötése**
  - **3. A kérések megvalósítása**
    - A legegyszerûbb esetben a kérés egy paraméter nélküli függvényhívás (pl. HandleHelper-nél). Ebben az esetben csak meghatározott számú kérést továbbíthatunk, azokat, amiket a Handler osztály definiál.
    - Használhatunk egyetlen kéréslekezelõ függvényt egy bemeneti paraméterrel, amit a kérés kódjaként értelmezünk Itt az egyetlen megkötés az, hogy a küldõnek és a fogadónak a kéréskódokat egységesen kell értelmeznie.
    - A kéréseket külön kérésobjektumokba csomagolhatjuk.

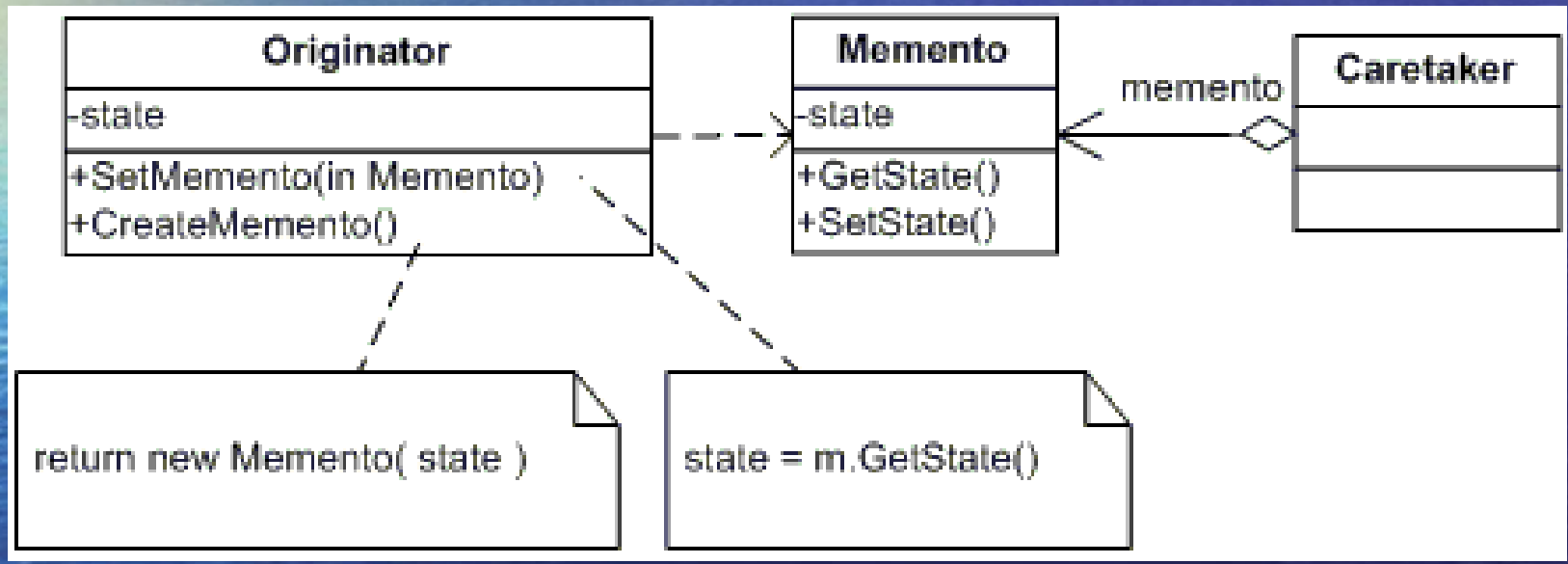
Behaioral Patterns - Memento

- Objektum állapotának tárolása (serialization) a belső tárolási szerkezet felfedése nélkül (information hiding).
- Ez a tervezési minta megoldási lehetőséget kínál az adott objektum belső állapotához való hozzáféréshez anélkül, hogy megsértenénk vele az egységbe zárás (*encapsulation*) szabályát. A belső állapothoz azonban ezután is csak olyan más objektumok férhetnek hozzá, amiknek ehhez jogosultságuk van.

Behaioral Patterns - Memento

- A tervezési mintában az **Originator** az az objektum, aminek egy régebbi belső állapotát szeretnénk visszaállítani. A régebbi állapotokat azonban nem célszerű az **Originator**on belül tárolni, ugyanis ez komplexebb struktúrájú állapotoknál nagyon megnövelné az objektum méretét.
- **Az Originator** belső állapotát nem ajánlatos más objektumnak átadni, mivel ez az egységbe zárás megsértését jelentené. A megoldást a felelősség kettéosztása jelenti: az **Originator** régebbi állapotait egy **Memento** objektum tárolja, amihez csak az **Originator** férhet hozzá, viszont ezeket a tároló objektumokat egy **CareTaker** objektum felügyeli. Ez az objektum határozza meg, hogy az **Originator** melyik régebbi állapotába térjen vissza és tartja számon azt is, hogy mely **Memento** példányokra van még szükség.

Behaioral Patterns - Memento



Behaioral Patterns - Command

- A Command minta parancsok, request-ek objektumba ágyazását, és ezen keresztül feladatok delegálását fogalmazza meg. Különféle parancsokat rendelhetünk így klienseinkhez(akár dinamikusan is), a parancsokokat akár sorba (queue) is állíthatjuk, naplózhatjuk(loging) illetve segítségével visszaavonható mûveleteket kezelhetünk (undo).

- Üzenetet, ill. kérést objektumba foglal, így megvalósulhat a kérések ideiglenes tarolása (queue) ill. az undo funkció elegáns megvalósítása.

Behaioral Patterns - Command

- Egy jellegzetes probléma:
  - Néha elkerülhetetlen, hogy kéréseket intézzünk egyes objektumokhoz anélkül, hogy bármit is tudnánk a kért mûveletrõl, vagy a kérést fogadóról. Például a felhasználói felületek programozására gyakran használt könyvtárak, ún. *toolkitek* többek között pl. menüket és gombokat tartalmazhatnak, amelyek egy esemény(pl. egérkattintás, billentyûzetleütés) hatására indítnak el valamilyen tevékenységet. Ezt azonban a menü, vagy a gomb objektumok nem tudják maguk implementálni, mert csakis a toolkitet használó applikáció tudja, hogy mit is kell ilyenkor tenni. A toolkitek tervezõi semmiképp sem tudhatják, hogy ezt a bizonyos tevékenységet végül ki(milyen objektum) és hogyan fogja végrehajtani.

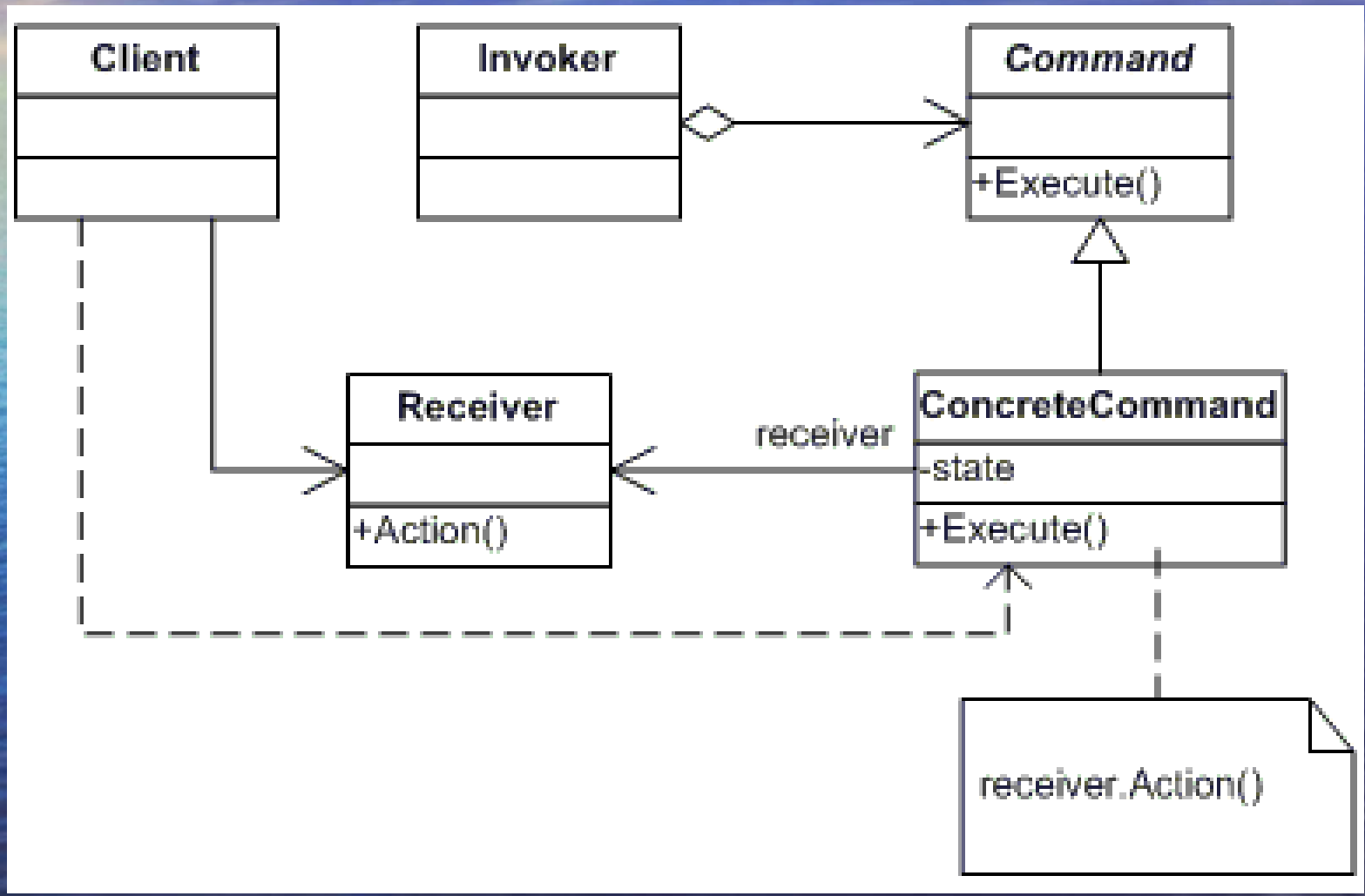
Behaioral Patterns - Command

- Egy megoldás a problémára :
  - A Command minta a kérést vagy parancsot egy objektumba ágyazza, így lehetõvé teszi, hogy például a toolkit elemei számukra ismeretlen applikációs objektumokon hajtsanak végre mûveleteket.
    Ennek kulcsa egy absztrakt Command osztály, mely mûveletek végrehajtásához definiál interface-t. Ez a legegyszerûbb esetben egy absztrakt Execute mûveletet tartalmaz. Egy konkrét Command alosztály(ConcreteCommand) egy objektumváltozó formájában tárolja a mûveletet ténylegesen végrehajtó objektumot, amely a kérést valójában fogadja. Csak ez utóbbi objektum rendelkezik a mûvelet végrehajtásához szükséges tudással.

Behaioral Patterns - Command

- *Command* :
  - interface-t deklarál mûveletek végrehajtására.
- *ConcreteCommand* :
  - Egy hozzárandelést definiál a kérést fogadó Receiver és a végrehajtandó mûvelet között.
  - A Receiver megfelelõ mûveletének meghívásával implementálja az Execute-ot.
  - Ha visszavonhatóságot is meg akarunk valósítani, akkor a *ConcreteCommand*-nak állapotot is kell tárolnia.
- *Client* :
  - Az applikációs objektum, mely létrehozza a ConcreteCommand osztályt, és hozzárendeli a megfelelõ Reveiver-t.
- *Invoker :*
  - a kérõ, egy általa ismert commanddal(egy ConcreteCommand osztály) végrehajtatja a kívánt mûveletet úgy, hogy meghívja annak Execute() metódusát
- Receiver :
  - a kérés fogadója; rendelkezik a mûvelet végrehajtásához szükséges tudással. Bármely osztály lehet fogadó.

Behaioral Patterns - Command



Behaioral Patterns - Command

- A minta további jellemzõi:
  - A Command minta alkalmazása megszünteti a csatolást a mûveletet kezdeményezõ és az azt végrehajtó objektumok között.
  - A parancs objektumok is közönséges objektumok.
  - A parancsok összetett parancsokká szervezhetõk, egy makrót képezve.

Behaioral Patterns - Iterator

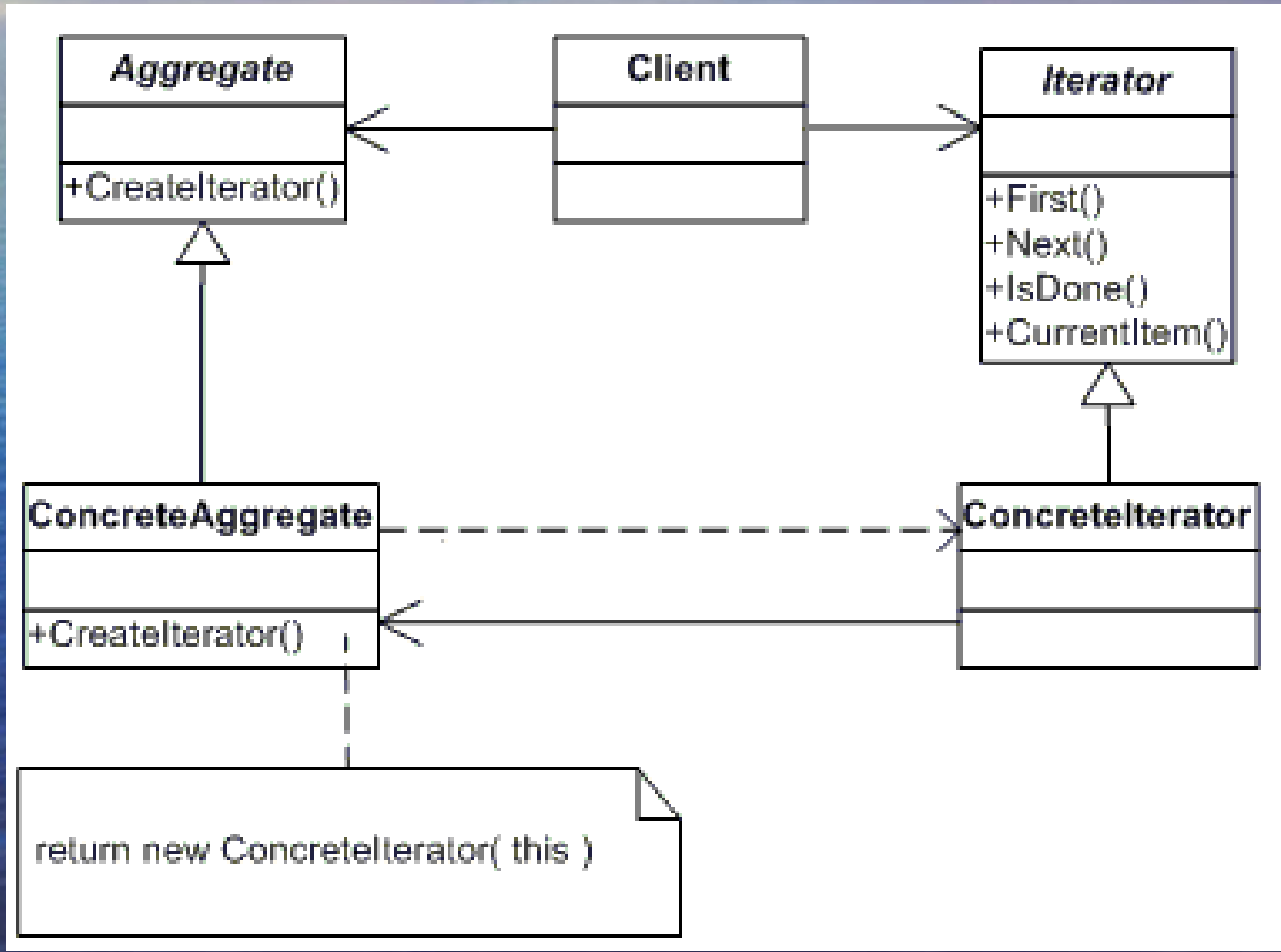
- Aggregált (gyűjtemény) objektumhalmaz elemeinek felsorolása a belső tárolási formától független módon

- Iterátor feladata:
  - egy összetett objektum elemeinek elérése a belső szerkezetének ismerete nélkül
  - többféle kiolvasási sorrend definiálása egy összetett objektumhoz
  - egységes interfész biztosítása különböző szerkezetű összetett objektumok elmeinek elérésére

Behaioral Patterns - Iterator

- Különválasztja a tárolás és a megjelenítés problémáját. Definiál egy interfészt a lista elemeinek elérésére, és minden pillanatban pontosan tudja, hogy a lista mely elemeit olvasta már ki az adott feldolgozás alatt. Azaz maga a lista objektum nem tud semmit a benne elhelyezkedő elemek sorrendjéről (esetleg van egy alapértelmezett sorrend), pusztán műveleteket biztosít az alapvető listakezelési feladatok megoldására. Úgymint új elem felvétele, törlés, az elemek megszámlálása stb. Ehhez az objektumhoz kapcsolódhatnak az Iterátorok, akik viszont a lista belső szerkezetét nem ismerik; az elemek tetszőleges szekvenciális elérését biztosíthatják.

Behaioral Patterns - Iterator



Behaioral Patterns - Mediator

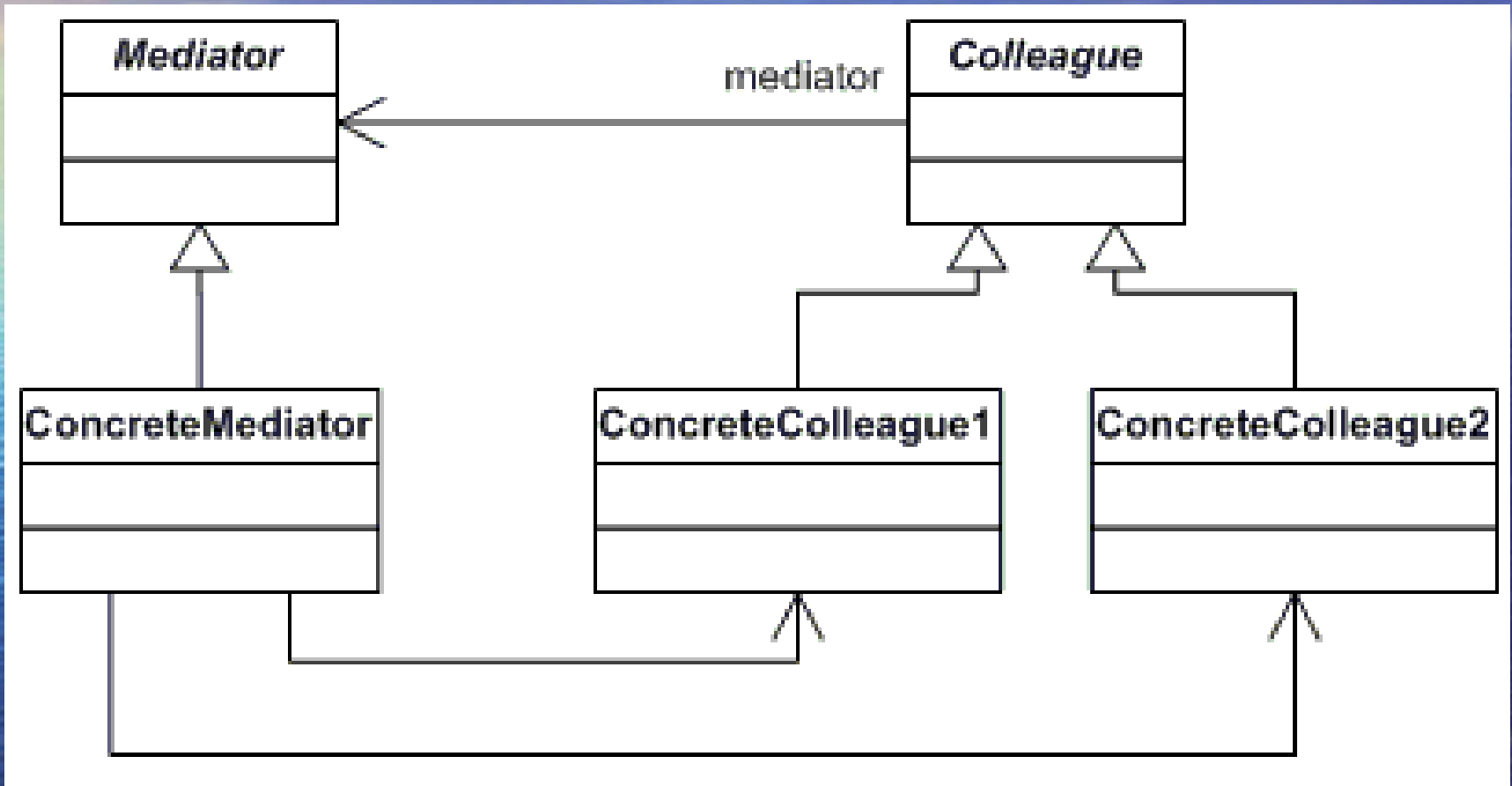
- Több egymással kapcsolatot tartó objektum kommunikációjának egységbe (objektumba) zárása

- Mikor érdemes a Mediator mintát használni?
  - Ha az objektumok közti kommunikáció igen összetett, strukturálatlan és nehézen érthető a létrejött kapcsolatrendszer.
  - Ha egy objektum újrahasznosíthatósága bonyolult lenne, mert sok más objektummal áll kapcsolatban.
  - Ha az egyes osztályok között elosztott viselkedési paraméterek kezelhetetlenek lennének alosztályok bevezetése nélkül.

Behaioral Patterns - Mediator

- Mediator
  - Definiál egy felületet, amelyen keresztül az objektumokkal kommunikálhat.
- Concrete Mediator
  - A kapcsolódó objektumokat koordinálja.
  - Ismeri társait és fenntartja velük a kapcsolatot.
- Colleague classes
  - Ismeri a Mediatort (és senki mást a Colleaguek közül!)
  - Ha egy másik Colleague-val szeretne kommunikálni, megkeresi a Mediatort

Behaioral Patterns - Mediator



Behaioral Patterns - Mediator

- **Együttműködés**
  - A résztvevők (Colleagues) jelzést küldenek, majd fogadnak a Mediatortól. A Mediator valósítja meg az interakciót a megfelelő Colleague-k között.

- **Következmények**
  - Leegyszerűsíti és biztosítja a kommunikációt az objektumok között.
  - Megköti, hogyan kooperáljanak az objektumok.
  - Központosítja az irányítást

Behaioral Patterns - Observer

- Egy objektumtól (annak állapotától) függő tetszőleges számú objektum értesítése az állapot megváltozásáról.

- Akkor alkalmazzuk, ha
  - az objektum struktúra sok, különböző interface-szel rendelkező objektumot tartalmaz, és az objektumokon végrehajtandó operáció a konkrét osztálytól függ;
  - sok különböző és egymással össze nem függő operációt akarunk végrehajtani az objektumokon, és nem akarjuk "beszennyezni" az osztályokat ezekkel az operációkkal;
  - az objektum struktúrát definiáló osztályok csak ritkán változnak, ellenben gyakran akarunk rajtuk új operációt definiálni.

Behaioral Patterns

- **Flyweight:** Objektumok „újrahasznosítása” /megosztása nagy számú, kis méretű adat/objektum kezelésére.
- **State:** Az objektum viselkedésének (interfészének) megváltoztatása a belső állapot függvényében.
- **Visitor:** Objektumhalmazon elvégzendő különböző műveletek objektumba zárása, a objektumhalmaz ill. a benne szereplő objektumok változtatása nélkül

Minta alkalmazás előnyök

- Használatukkal a tervezés felgyorsítható, a „kerék újrafeltalálása” elkerülhető
- Bevált gyakorlat és tervezési stratégiák dokumentálása, újrafelhasználásának elősegítése
- Tetszőleges absztrakciós szinten megjelenhet
- Funkcionális és nem funkcionális követelményeket is tartalmazhat

Formalizáláuk

Tervezési minták: UML diagram részletek, forráskód minták, „elbeszélő szövegrészek”

Formalizálás szükségessége:

- Célok tisztázása,
- Érthetőség növelése,
- Tool támogatás

érdekében.

Formalizálási torekvések:

- UMLaut
- LePUS

LePUS

Jellemzők:

- Vizuális formalizmus OO rendszerekhez
- Formális
- Tömör
- Tool támogatás

Célok:

- Tervezési minták leírására
- Minta viszonyok meghatározására
- Létező mintákból újak leszármaztatása

UMLaut

- Cél:
  - Minta résztvevők együttműködéseinek megfogalmazása
- Probléma:
  - Bizonyos tervezési mintákra vonatkozó kényszerek OCL-ben sem fogalmazhatók meg, lásd pl. Visitor minta
- Megoldás:
  - UML 1.3 metamodell módosítások tervezési minták modellezése érdekében

Felhasznált irodalom, referenciák

- http://www.inf.bme.hu
- www.inf.bme.hu/ooret/1999osz/DesignPatterns
- http://informatika.szif.hu/infoportal/tervezeselemzes.pdf
- http://www.dofactory.com/patterns/Patterns.aspx
- http://www.neumann-centenarium.hu/kongresszus/eloadas/ea55.pdf
- http://home.mit.bme.hu/~volgyesi/BRI/AspectOrientedProgramming.pdf
- http://www.mee.hu/06etech/2004/2004_01/pdf/03.pdf
- http://www.xsys.hu/letoltes.html#designpatterns
- http://rs1.sze.hu/~raffai/org/
- http://informatika.szif.hu/infoportal/tervezeselemzes.pdf