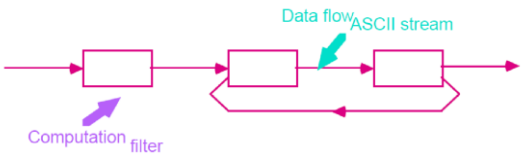
Szoftverarchitektúrák ZH

Tartalom

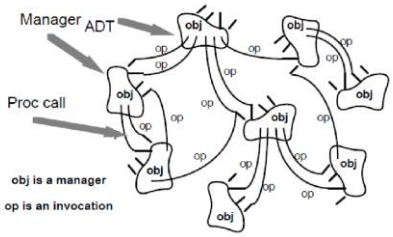
Bevezetés

*Közös architekturális stílusok***

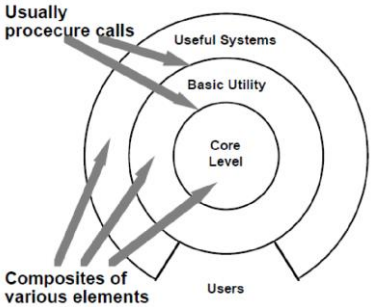
- Komponensek és konnektorok
- Pipes and Filters
  - Egyszerűsített pipeline
  - Batch szekvenciális végrehajtás
  - Unix Shell jó példa erre
  - Definiált bemenet-kimenet megközelítés



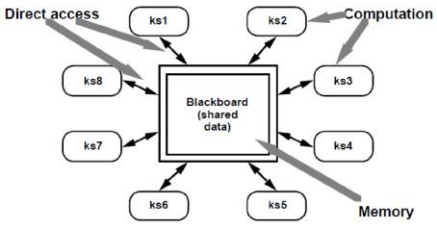
- Adatabsztrakció és objektumorientáció
  - Dekompozíció
  - Védett interakció



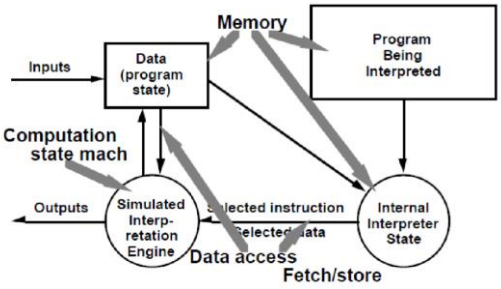
- Eseményvezérelt, implicit hívás
  - Implicit hívás
  - Reaktív viselkedés
  - Szelektív broadcast
  - Újrafelhasználhatóság
  - Hátránya: Az eseményt létrehozó komponens nem tudja befolyásolni azt, hogy hogyan reagál a célkomponens.
- Rétegezett rendszerek
  - Magasabb absztrakció
  - Újrafelhasználhatóság
  - Rugalmasság, skálázhatóság



- Repositories
  - Knowledge sources
  - Blackboard Data Structure
  - Control



- Táblavezérelt interpreterek
    - Virtuális gépek
    - Híd a program és az alatta levő hardver között
    - Hordozhatóság, Migrálhatóság.



*Top 10 architekt hiba amire vigyázni kell***

1. Scoping Woes.
2. Not Casting Your Net Widely.
3. Just Focusing on Functions.
4. Box and Line Descriptions.
5. Forgetting That It Needs to be Built.
6. Lack of Platform Precision.
7. Making Performance and Scalability Assumptions.
8. DIY Security.
9. No Disaster Recovery.
10. No Backout Plan.

A minta ismertetés sablonja

- a: név, becenév
- b: kontextus
- c: probléma
- d: megoldás
- e: statikus viselkedés (osztálydiagram)
- f: dinamikus viselkedés (szekvenciadiagram)
- g: implementációs kérdések
- h: hol használták?
- i: mérleg: előnyök-hátrányok

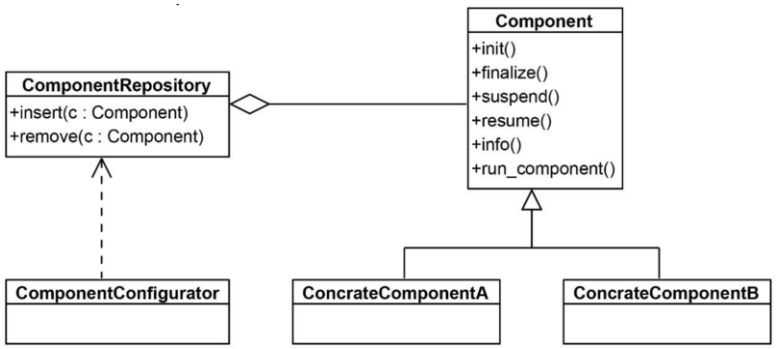
Szolgáltatáshozzáférés és -konfiguráció

*Component Configurator***

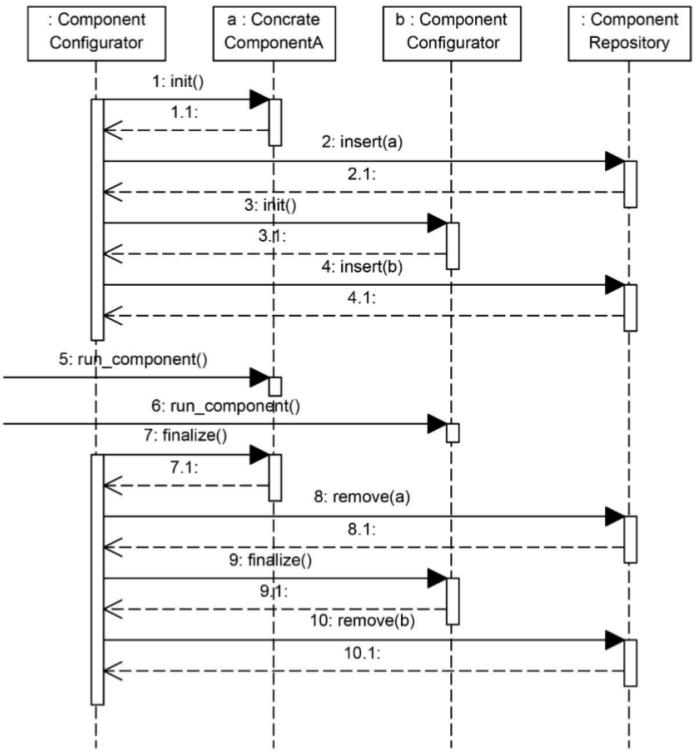
- a: Edző (Coach)
- b: A Component Configurator minta lehetővé teszi, hogy az alkalmazások a különböző komponens implementációkat futási időben csatlakoztassák illetve leválasszák (az alkalmazás

módosítása, újrafordítása vagy statikus újralinkelése nélkül). Ezen kívül ez a minta támogatja a komponensek átkonfigurálását különböző alkalmazásfolyamatokhoz a folyamatok újraindítása nélkül.

- c: probléma
  - A komponens funkcionalitása változhat, a teljesítmény bizonyulhat nem elegendőnek.
  - Közös adminisztrációs feladatok
- d: megoldás
  - Válasszuk szét a komponensek interfészeit az implementációjuktól, továbbá a komponens kínáljon egy egységes interfészt az általa nyújtott szolgáltatás vagy funkcionalitás konkrét típusának konfigurációjára és vezérlésére.
- e: Négy szereplő van:
  - Komponens: Egységes interfészt definiál a konfigurációra és vezérlésre.
  - Konkrét komponensek: A komponens konkrét implementációi.
  - Komponenstár: Komponenstároló.
  - Komponensbeállító: Főnök, Coach vezérlő.



- f:
  - Komponens elindítás: A komponensbeállító dinamikusan csatol egy komponenst az alkalmazáshoz, és inicializálja azt. A sikeres inicializáció után a komponensbeállító a komponenst hozzáadja a komponenstárhoz.
  - Komponens használat: Az alkalmazáshoz való csatlakozás után a komponens különböző feladatokat végezhet (pl. más komponensekkel való üzenetváltás, kérések kiszolgálása). Eközben a komponensbeállító ideiglenesen felfüggesztheti, illetve újraindíthatja a komponenst (erre pl. a többi komponens átkonfigurálása esetén lehet szükség).
  - Komponens leállítás: A komponensbeállító leállítja azokat a komponenseket, amelyekre nincsen szükség a továbbiakban. Leállítás során a komponenseknek fel kell szabadítaniuk a fogvatartott erőforrásaikat. A sikeres leállás után a komponensbeállító eltávolítja a komponenst a komponenstárból, és az alkalmazás címteréből.
- f:



- g:
- h: Windows Service Control Manager (SCM). Java Applets, dynamic download init, start, stop.
- i: Mérleg

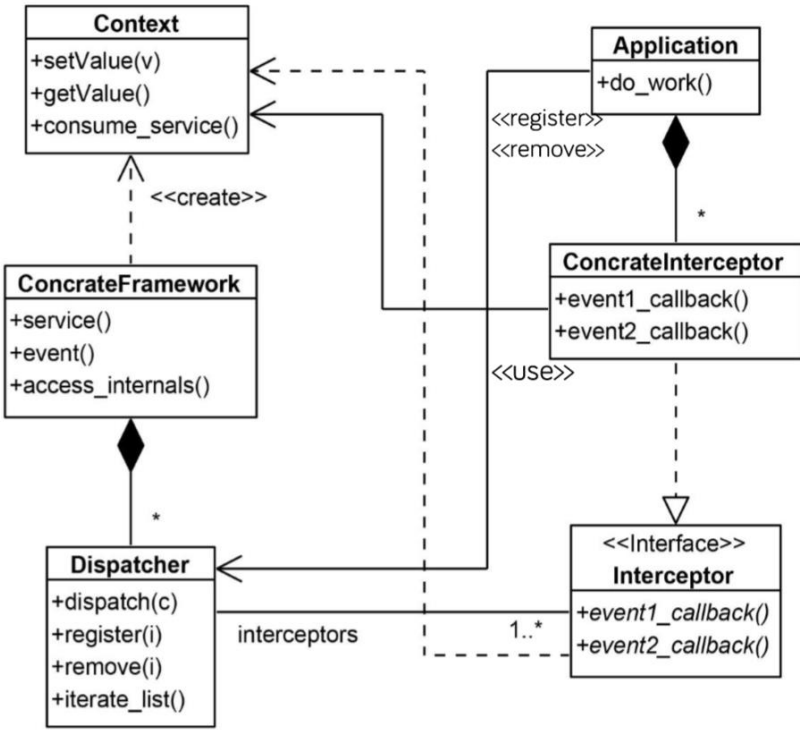
Előnyök	Hátrányok
Egységes komponenshozzáférés	Biztonság
Központi adminisztrálhatóság	Teljesítmény overhead
Dinamikus komponenscsatlakozás -vezérlés	Bonyolultság
Modularitás, optimalizálás	

*Interceptor***

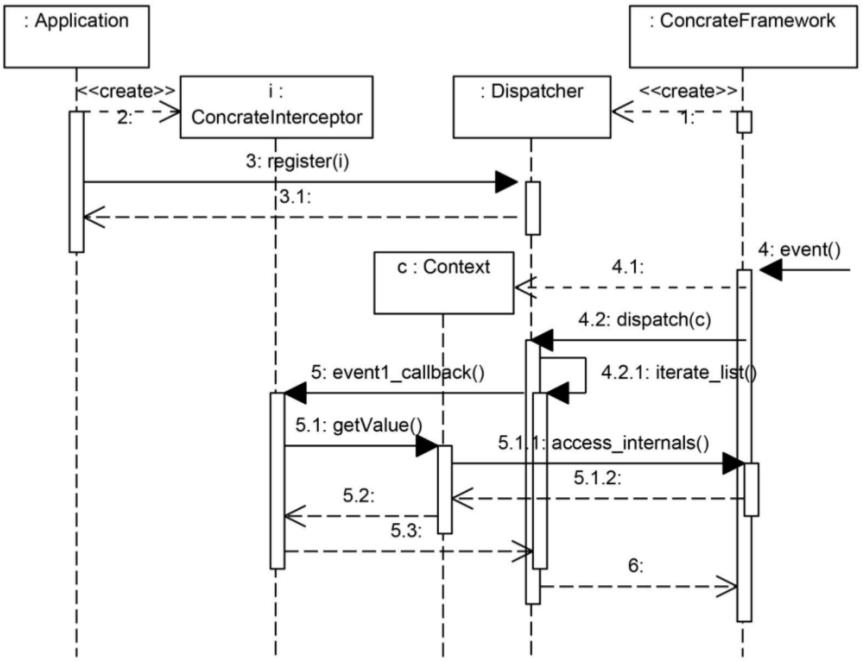
- a: Címváltozás, levéltovábbítás
- b: Az interceptor tervezési mintát alkalmazva átlátszó módon egészíthetünk ki keretrendszereket különböző szolgáltatásokkal. Szolgáltatásokat rendelhetünk hozzá a keretrendszer bizonyos belső eseményeihez. Az események előfordulásakor ezek a szolgáltatások automatikusan lefutnak.
- c: Milyen problémákat oldunk meg?
    - Az új szolgáltatások legyenek egyszerűen, a keretrendszer módosítása nélkül hozzáadhatóak.
    - Az új szolgáltatások hozzáadása ne módosítsa se a keretrendszer eddigi működését, se a keretrendszer funkcióinak igénybevételének a módját.
    - A keretrendszer működése legyen monitorozható.
- d: Definiáljunk a keretrendszer belsejében, bizonyos eseményekhez rendelt beavatkozási pontokat (interception points). Minden beavatkozási ponthoz definiáljunk egy-egy külön interfészt (interceptor interface), amely egy (vagy akár több) callback metódust tartalmaz. Rendeljünk továbbá minden beavatkozási ponthoz egy dispatcher-t, amely lehetővé teszi a kliensek számára, hogy szolgáltatásokat (azaz interception interface implementációkat) rendelhessenek hozzá az adott beavatkozási ponthoz. Amennyiben a keretrendszer elér egy beavatkozási pontot, vagyis belső működése során bekövetkezik egy bizonyos esemény, akkor a beavatkozási ponthoz rendelt minden szolgáltatásnak le kell futnia. Esetenként szükség lehet arra is, hogy a szolgáltatás módosítsa a keretrendszer belső állapotát. Ehhez

nyissuk meg a keretrendszert a csatlakoztatható szolgáltatások számára. Minden beavatkozási ponthoz definiáljunk egy kontextusobjektumot, amelyen keresztül a keretrendszer belső működése egy meghatározott fokig szabályozható, és a beavatkozási pontok elérésekor adjuk át ezeket a kontextusobjektumokat a szolgáltatásoknak.

- e: Hat szereplő van:
    - Konkrét keretrendszer
    - Interceptor
    - Konkrét Interceptor
    - Dispatcher
    - Context Object
    - Application



- f: Az alkalmazás (application) példányosít egy konkrét elfogót (concrete interceptor), amely egy adott elfogó interfészt implementál, majd a konkrét elfogót beregisztrálja a megfelelő diszpécsernél (dispatcher), melyet a konkrét keretrendszer (concrete framework) példányosít. A konkrét keretrendszer ezek után egy olyan eseményt érzékel, amelyet el kell fogni. Esetünkben minden egyes ilyen eseményhez egy-egy speciális kontextusobjektum (context object) tartozik. Tehát a konkrét keretrendszer példányosítja az eseményspecifikus kontextusobjektumot, amely egyrészt információkat tárol az keletkezését kiváltó eseményről, másrészt a konkrét keretrendszerhez hozzáférést biztosító függvényeket tartalmaz. Ezt követően a konkrét keretrendszer értesíti a megfelelő diszpécsert az esemény bekövetkezéséről úgy, hogy paraméterként átadja neki az imént létrehozott speciális kontextusobjektumot. A diszpécser ennek hatására rendre végighalad a konténerében elhelyezkedő, beregisztrált konkrét elfogókon, és – bemenetként a kapott kontextusobjektummal – meghívja callback() metódusukat. A kontextusobjektum feldolgozását követően a konkrét elfogó opcionálisan meghívhatja a kontextusobjektum metódusait, melyekkel befolyásolhatja a konkrét keretrendszer működését, és későbbi eseményfeldolgozását. Végül, miután mindegyik konkrét elfogó callback() metódusa visszatért, a konkrét keretrendszer folytatja megszokott működését.
- f:



- g:
- h: Komponens alapú alkalmazásszerverek, CORBA, COM, Web Browsers, Angular hibakezelés
- i: Mérleg Előnyök Hátrányok

Előnyök	Hátrányok
Kiterjeszthetőség, Rugalmasság	Komplex tervezés
Monitorozhatóság	Hibás interceptorok problémát okoznak
Újrafelhasználhatóság	Teljesítmény Overhead
Aspektus alapú megközelítés támogatása	

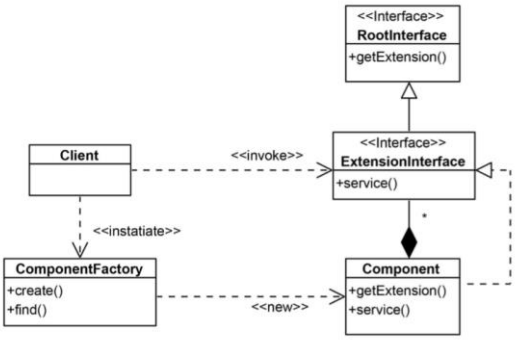
*Extension Interface***

- a: Extension Interface
- b: Olyan alkalmazáskörnyezetben hasznos, ahol a komponensek interfészei fejlődnek az idő folyamán.
- c: probléma
    - Ha a komponensek interfészei nem változnak, akkor a komponensek implementációjában bekövetkező változás nem okozhat problémát a kliens-oldali kódban.
- c: probléma
    - A kliens-oldali kódban az sem okozhat gondot továbbá, ha a fejlesztők újabb, külsőleg látható szolgáltatásokkal egészítik ki a komponenst. Ideális esetben ekkor még a kliens-oldali kód újrafordítása sem szükséges.
    - A komponensek funkcionalitásának megváltoztatása viszonylag egyszerű legyen, és ne okozza se a meglévő komponens interfészek túlbonyolódását, se belső architektúrájuk instabilitását.
    - Távolról, illetve lokálisan ugyanazon az interfészen keresztül legyenek elérhetők a komponensek. Ha a komponensek, és klienseik elosztva helyezkednek el különböző hálózati csomópontokon, úgy a komponensek interfésze és implementációja legyen különválasztva.
- d: Megoldás
    - Exportáljuk a komponens funkcionalitását kiegészítő interfészeken keresztül, melyek mindegyike műveletek egy-egy szemantikusan összefüggő halmazának felel meg.

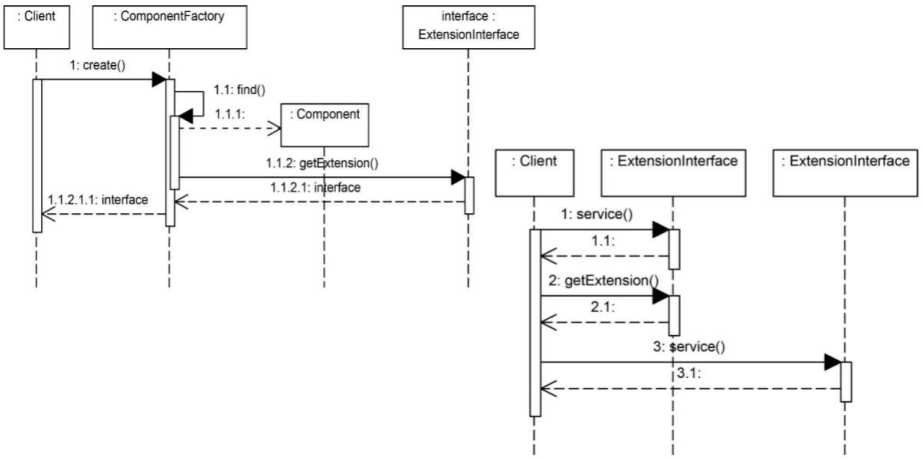
Minden komponensnek legalább egy kiegészítő interfészt meg kell valósítania. A komponens meglévő funkcionalitásának módosítása, vagy kiegészítése céljából inkább hozzunk létre újabb kiegészítő interfészeket ahelyett, hogy a meglévőket módosítanánk. Sőt, a klienseket úgy programozzuk, hogy a komponenseket ne az implementációjukon, hanem az interfészeiken keresztül érjék el.

- Ahhoz, hogy a kliensek számára lehetővé tegyük komponens-példányok létrehozását és a komponensek kiegészítő interfészeinek elérését, vezessünk be egy újabb indirekciót. Vezessünk be egy-egy komponens üzemet (component factory), amely az adott típusú komponens példányosítására szolgál. A komponens üzem egy gyökér interfész (root interface) referenciát is adjon vissza, amely alapján a kliensek a komponens további kiegészítő interfészeit is elérhetik. Ezt például úgy érhetjük el, hogy mindegyik kiegészítő interfészt a gyökér interfészből származtatjuk, amely az összes kiegészítő interfészre jellemző funkcionalitást valósítja meg, pl. képes tetszőleges kiegészítő interfész szolgáltatására.

- e: Öt szereplő van:
  - Komponensek: különböző típusú szolgáltatásspecifikus funkcionalitást valósítanak meg.
  - Kiegészítő interfészek: a komponens implementációjának különböző szerepeit exportálják.
  - Komponens üzem: Component Factory
  - Gyökér interfész: A komponens életciklus funkcionalitását kínálja, minden interfész ebből származik.
  - Kliens alkalmazás: Igénybe veszi a komponens szolgáltatásait interfészeken keresztül.
- e: Statika, struktúra



- f:



- g:
  - Determine the stability of the design and the long-term application requirements.
  - Analyze the domain and specify a domain-specific component model.

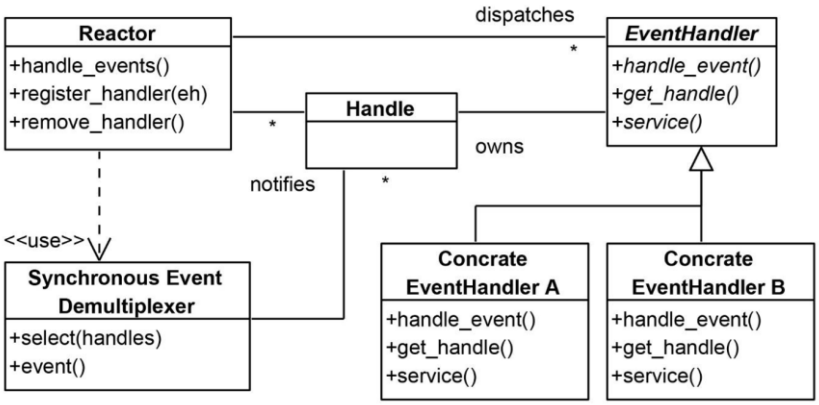
    - Specify the root interface.
    - Implement the mechanism to retrieve extension interfaces.
    - Decide which functionality the factory will export.
- h:
    - Microsoft COM/COM+
    - CORBA 3
    - EJB
    - OpenDoc
- i: Mérleg

Előnyök	Hátrányok
Kiterjeszthetőség	Ráfordítás növekedése komponenstervezés és implementáció terén
Szemantikus aspektusok szétválasztása	Teljesítmény overhead
Poliformizmus	Kliens komplexitásának növekedése
Interfész aggregáció és delegáció	

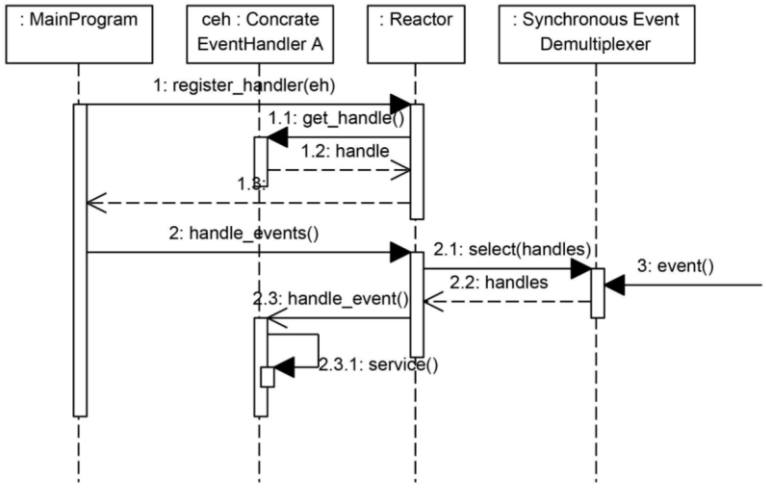
Eseménykezelés

*Reactor***

- a: Dispatcher, Notifier, „Telefonközpont”
- b: Egy eseményvezérelt alkalmazás, ami több eseménykérést fogad egyszerre és feldolgozza őket szinkron módon, sorosan.
- c: probléma
    - A skálázhatóság növelése, a késleltetés csökkentése.
    - A kiszolgálási kapacitás növelése, a fölösleges kapcsolgatások csökkentése.
    - A szinkronizációs mechanizmusok minimalizálása.
- d: Megoldás:
    - A beérkező jelző események szétosztását szét kell választani a szolgáltatáson belüli alkalmazásspecifikus feldolgozásától. Minden szolgáltatás számára az alkalmazás egy külön eseménykezelőt (event-handler) kínál, amely feldolgozza az adott forrástól érkező adott típusú eseményt. Az eseménykezelők regisztrálják magukat egy Reaktornál, amely szinkron esemény szétválasztást (synchronous event demuliplexing, SED) használ, mikor a jelző eseményekre várakozik. Ezek bekövetkeztekor a SED értesíti a reaktort, amely szinkron módon elindítja az eseményhez rendelt eseménykezelőt, amely magát a szolgáltatást nyújtja.
- e: Öt szereplő van:
    - Reactor: A központi elem. Regisztrálja az eseményekhez az eseménykezelőket.
    - Synchron Event Demultiplexer: Szétválogatja az érkező eseményeket.
    - Event Handle: Eseményleíró
    - Event Handler: Absztrakt osztály az eseménykezelő struktúra leírására.
    - Concrete Event Handler: Több van belőle, konkrét eseménykezelő implementációk.
- e: Statika, struktúra



- f:
  - Az alkalmazás elindulásakor a konkrét eseménykezelők regisztrálnak a reactornál a register_handler() függvényhívásokkal.
  - A Reactor összegyűjti az eseményleírókat a get_handler() függvényhívásokkal.
  - Miután minden eseménykezelő regisztrált, a Reactor elindítja az eseménykezelő ciklust a handle_events() függvény meghívásával. Ebben a Reactor meghívja a szinkron eseményelosztót az összegyűjtött eseményleírók listájával.
  - A szinkron eseményelosztó visszatér, amikor egy vagy több esemény érkezett a kezelendő eseményleíróval.
  - A Reactor megkapja az eseményt feldolgozásra. Az eseményleíró alapján azonosítja az eseménykezelőt, és meghívja a handle_event() függvényét.
  - Az eseménykezelő beolvassa az adatokat, elvégzi a kérés kiszolgálását, és visszatér a válasszal.
- f:

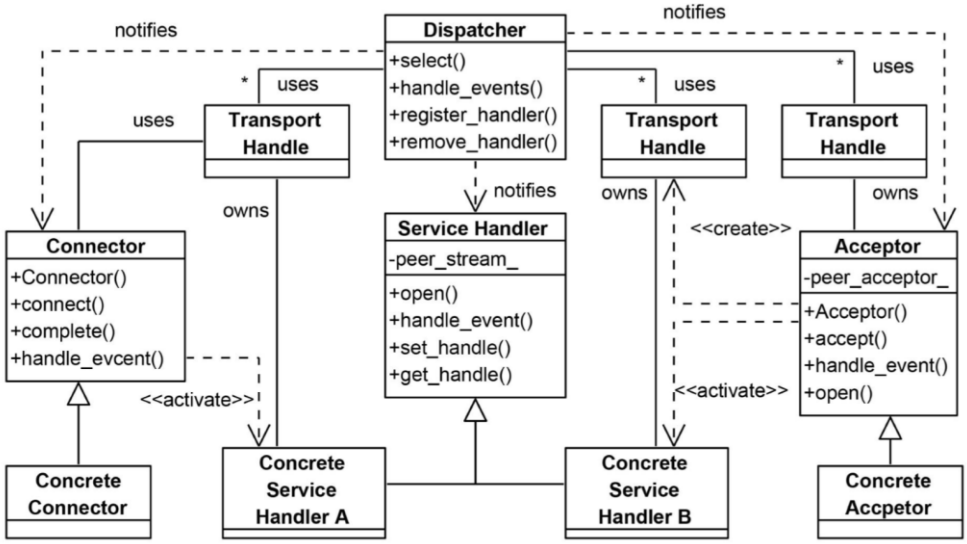


- g:
  - Eseménykezelő interfész definiálása.
  - Eseménykezelési „dispatch” interfész stratégia meghatározása.
  - A Reactor interfész definiálása és implementálása.
  - SED mechanizmus meghatározása.
  - Az alkalmazásban alkalmazandó Reactorok számának eldöntése.
- h:
  - Windows eseménykezelési mechanizmus
  - Xwindows eseménykezelési mechanizmus
  - CORBA ORB Core
  - Call Management System
- i: Mérleg

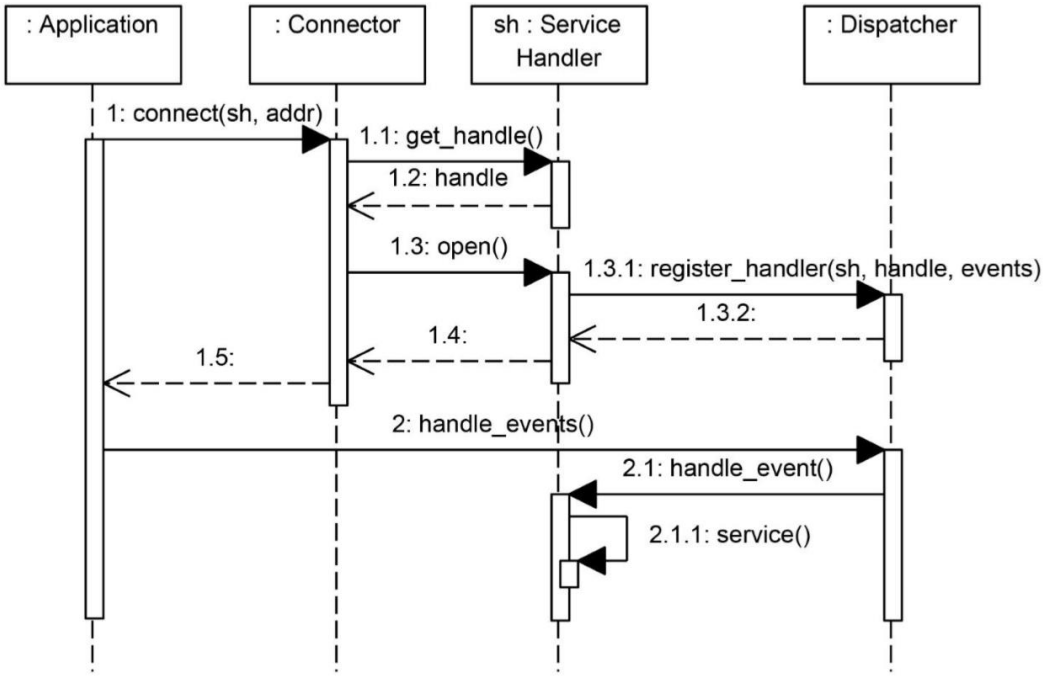
Előnyök	Hátrányok
Modularitás, újrafelhasználhatóság és konfigurálhatóság	Szinkron várakozás
Szemantikus aspektusok szétválasztása	Debuggolás nehézsége
Hordozhatóság	
Kontrolálható konkurenciakezelés	
Egyszerűség	

*Acceptor-Connector***

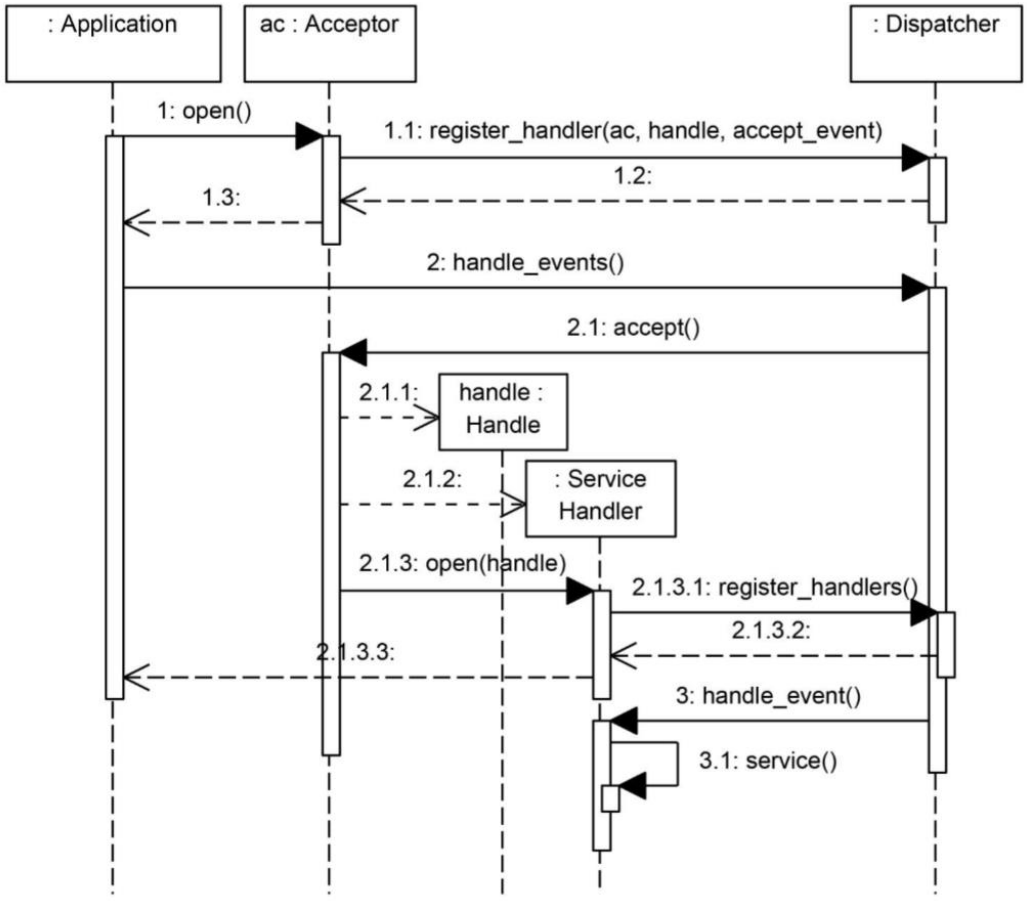
- a: „Manager-Secretary”
- b: Egy rendszerben, ahol több szolgáltatási csomópont van, szétválasztjuk a kapcsolatépítés kezelését a végrehajtó szolgáltatásoktól.
- c: probléma:
  - A szolgáltatási csomópontok tehermentesítése a kapcsolatépítéstől
  - Bővítési lehetőség biztosítása
  - A kapcsolatfelépítési késleltetésének minimalizálása
- d: Megoldás
  - Külön csomópontban valósítjuk meg a kapcsolatfelvételt a szolgáltatás végrehajtásától. A fogadó (acceptor) passzív szereplő, és vár egy kapcsolatfelvételi kérésre, a kezdeményező (connector) pedig aktív.
- e: Nyolc szereplő van:
  - Acceptor
  - Concrete Acceptor
  - Service Handler
  - Concrete Service Handler
  - Connector
  - Concrete Connector
  - Dispatcher: Központi elem
  - Transport Handle
- e: Statika, struktúra



- f: Dinamika



- f: Dinamika-2



- g:
  - A Dispatcher megvalósításában felhasználhatjuk a Reactor vagy Proactor egyes részeit.
- h:
  - UNIX Network Superservers
  - Web Browsers
  - CORBA ORB
- i: Mérleg

Előnyök	Hátrányok
Hordozhatóság, újrafelhasználhatóság, kiterjeszthetőség	Többletköltség, komplexitás
Hatékonyság	Dedikált funkció
Robusztusság	Single point of failure
Könnyű implementáció	

Szinkronizáció

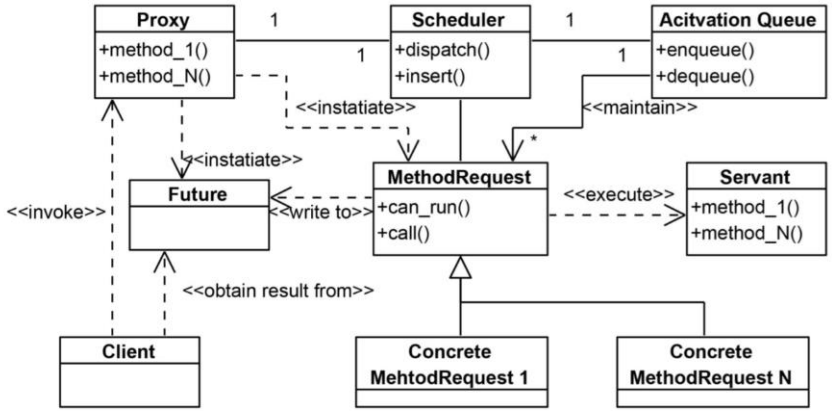
Konkurenciakezelés

*Active Object***

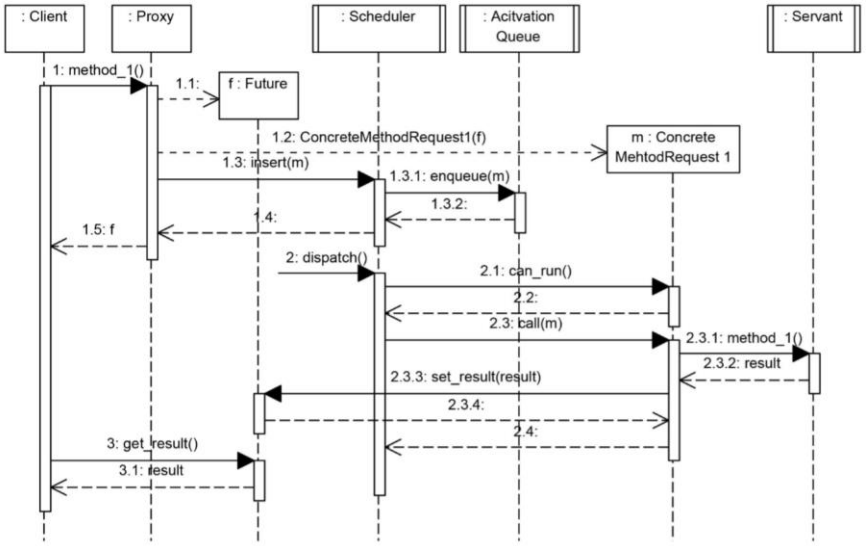
- a: Concurrent Object, „Chef in a Restaurant”
- b: A kliensek meghívnak különböző szálakban lévő objektumokat (azoknak a metódusait). Az Active Object minta szétválasztja a metódus futtatását a metódus hívásától annak érdekében, hogy konkurens viselkedést vigyen a rendszerbe, és egyszerűsítse az olyan objektumokhoz való hozzáférést, amik külön szálban futnak.
- c: probléma
    - Ahelyett, hogy egy egyszálú passzív objektumot használna, ami ugyanabban a szálban hajtja végre a metódust, amelyikben a kliens azt meghívta, egy konkurens objektum ezt a saját szálában teszi meg. Azonban ilyen konkurens objektumok esetén szinkronizálnunk kell az objektum metódusaihoz és adataihoz való hozzáférést, ha ezt az objektumot egyszerre több kliens képes elérni. Ebben az esetben a következő kényszereket kell betartanunk:
        - A konkurens objektumon meghívott erőforrás-, vagy feldolgozásintenzív metódus nem blokkolhatja az objektumot korlátlanul, ezzel degradálva az objektum szolgáltatásait.
        - Az olyan kliensoldali metódushívásokat egy konkurens objektum esetében, amikre különböző szinkronizációs kényszerek érvényesülnek, átlátszó módon kell sorosítani és ütemezni.
        - Az alkalmazásokat úgy kell megtervezni, hogy az aktuális hardver- és szoftverplatform párhuzamosítási lehetőségeit átlátszó módon tudja kezelni.
- d: Megoldás
    - A metódus végrehajtását szétválasztjuk a metódus hívásától. Míg a metódus hívása klienst kezelő szálban történik, addig a metódus végrehajtásának egy másik szálban kell megtörténnie. Részletesebben kifejtve egy proxy reprezentálja az aktív objektum interfészét, és egy servant objektum implementálja az aktív objektumot. A proxy és a servant külön szálban futnak, így a metódushívás és a metódus végrehajtás tud konkurensen működni. A proxy fut a klienst kezelő szálban, míg a servant egy másikban. Futás közben a kliens szálban futó proxy alakítja át a konkrét metódushívásokat metódushívás kérésekké (method request), amiket az ütemező végrehajtási listájába (activation list) el is tárol. Az ütemező esemény-feldolgozó ciklusa folyamatosan fut, ugyanabban a szálban, amiben a servant objektum is, és folyamatosan leveszi a kéréseket a végrehajtási listáról, és elküldi a servantnak. A kliens egy future objektumon keresztül tudja a későbbiekben elérni a metódus végrehajtásának eredményét, amit a proxy küld vissza.
- e: Nyolc szereplő van:
    - Client
    - Poxy: interfészt biztosít a kliensek számára.

- Method Request
- Concrete Method Request
- Scheduler
- Activation List
- Servant
- Future

- e: Statika, struktúra



- f: Dinamika



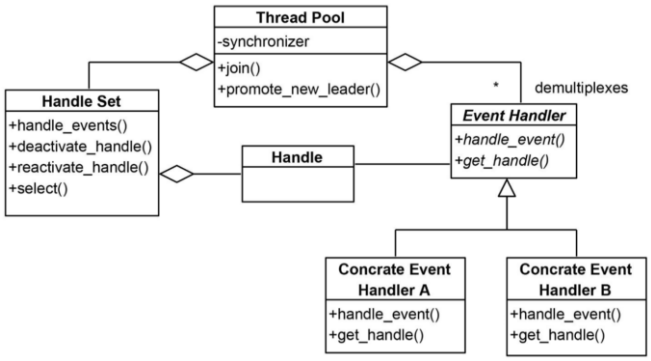
- g:
  - Servant
  - Scheduler
  - Activation List
  - Method Request
- h:
  - Java JDK 1.3 mechanizmus a konkurens futtatásra classes java.util.Timer and java.util.TimerTask
  - Symbian Párhuzamosítás
  - Automatic Call Distribution (Siemens) > Chef in a Restaurant
- i: Mérleg

Előnyök	Hátrányok
Enhances application concurrency and simplifies synchronization complexity	Performance overhead
Transparently leverages available parallelism	Debuggolás nehézsége

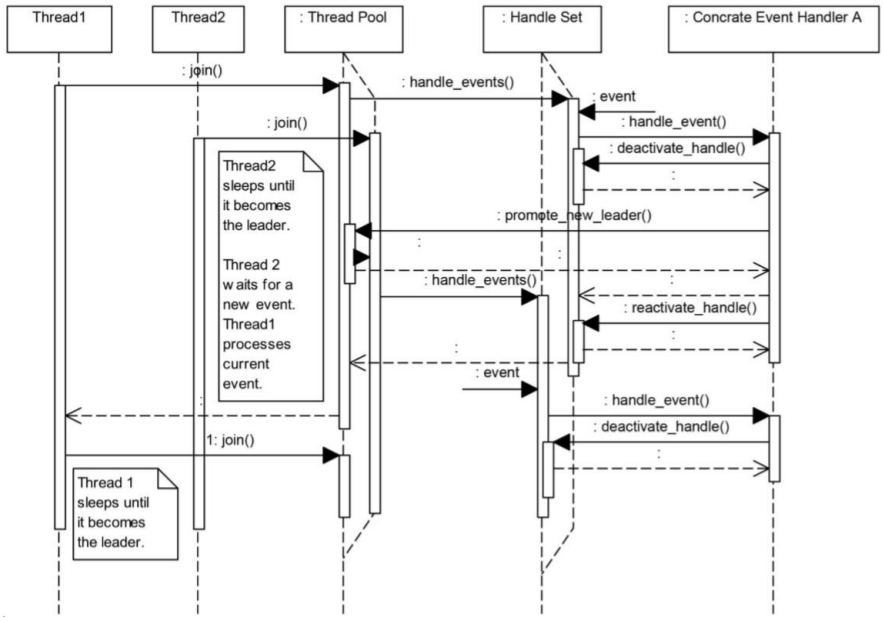
Method execution order can differ from method invocation order	

*Leader-Followers***

- a: „Taxi Stands”
- b: Leader/Followers architekturális minta egy hatékony konkurenciakezelési modellt definiál, ahol többszálú, megosztott eseményhalmazt kezelnek. Ide tartozik az eseménydetektálás, -válogatás és -végrehajtás.
- c: probléma
- Nagyszámú különböző típusú események kezelése többszálú környezetben. A cél egy hatékony modell definiálása a fenti feladat elvégzéséhez.
- d: Megoldás
    - Definiáljunk egy szálpoolt, ahol várakoznak a végrehajtó szálak. A pool tetején egy Leader thread vár egy esemény érkezésére, a többi szál (Followers) passzívan vár. Ha a Leaderhez érkezik egy esemény, kijelöli a poolból a következő leadert, és elkezdi végrehajtani az esemény funkcionalitását. Ha elvégezte a feladatát, visszakerül a poolba Followerként.
- e: Öt szereplő van:
    - Handle
    - Handle Set
    - Event Handler
    - Concrete Event Handler
    - Thread pool
- e: Statika, struktúra



- f: Dinamika



- g: -
- h:
    - CORBA ORB
    - Web Servers
    - Transaction Monitor
    - Taxi Stands
- i: Mérleg

Előnyök	Hátrányok
Egyszerűsített konkurenciakezelés	Rugalmatlanság
Performancia	Hálózati hiba esetén nehézkes
Egyszerű programozás	

Minták összefoglalása

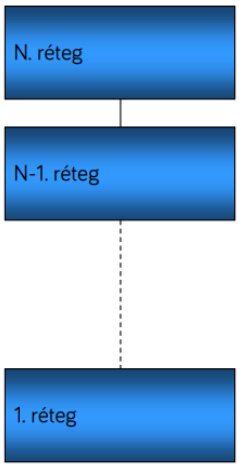
	Architectural Pattern	Design Pattern	Idiom
Base Line Architecture	Broker Layers Microkernel		
Communication	Pipes and Filters	Abstract Session Command Processor Forwarder-Receiver Observer Remote Operation Serializer	
Initialization		Activator Client-DispatcherServer Evictor Locator Object Lifetime Manager	
Service Access and Configuration	Interceptor	Component Configurator Extension Interface Half Object plus Protocol Manager-Agent Proxy Wrapper Facade	
Event Handling	Reactor Proactor	Acceptor-Connector Asynchronous Completion Token	

		Event Notification Observer Publisher-Subscriber	
Synchronization	Object Synchronizer	Balking Code Locking Data Locking Guarded Suspension Double-Checked Locking Optimization Reader/Writer Locking Specific Notification Strategized Locking Thread-Safe Interface	Scoped Locking
Concurrency	Half-Sync/HalfAsync Leader/Followers	Active Object Master-Slave Monitor Object Producer-Consumer Scheduler Two-phase Termination Thread-Specific Storage	

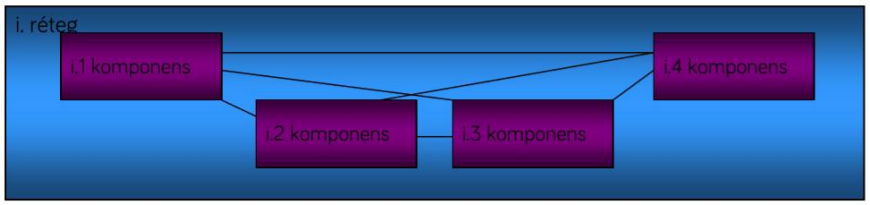
Architektúrális minták

*Rétegelés és szigorú rétegelés, előny, hátrány**

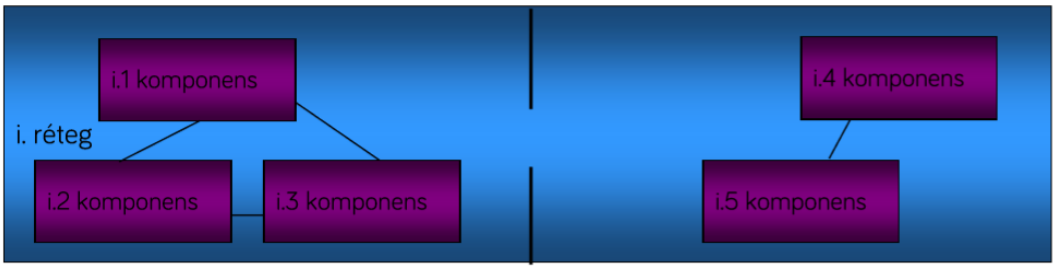
- Rétegelés
  - Legalapvetőbb szervezési elv
  - Hol fordul elő?
    - A batch világban nem volt
    - OS felépítése: driver - OS (API) – alkalmazás
    - Hálózati protokollok
    - Vállalati információs rendszerek
    - Stb.
- Szigorú rétegelés



- Szigorú rétegelés
  - Az i. réteg felelőssége
    - szolgáltatásokat nyújtani az i+1. rétegnek
    - alfeladatokat szolgáltatni az i-1. rétegnek
  - Az egyes rétegeken belül azonban tetszőleges függőségi viszonyok:



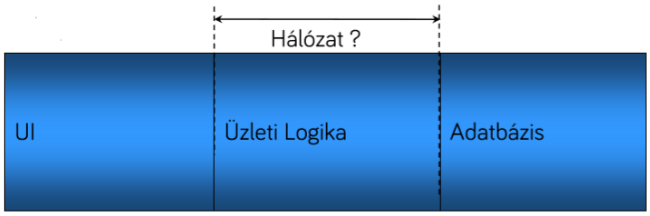
- Függőleges partícionálás
  - Az egymástól független komponenseket egy rétegen belül csoportosíthatjuk:



  - Előnyök
    - Túl komplex a feladat: vezessünk be egy új absztrakciós szintet (új rétegbe), és oldjuk meg erre építve!
    - A részek egymástól függetlenül cserélhetők, fejleszthetők.
    - Egy rétegre építve sok szolgáltatás építhető (többször is felhasználható).
    - Egy réteg mögött kicserélhető, bővíthető a többi réteg, így az adott réteg szolgáltatásai új kontextusban is felhasználatók.
    - Egy réteg működése önmagában is megérthető, nem kell a többit is megérteni.
  - Hátrányok
    - Egyszerűbb feladatnál felesleges komplexitás
    - A változások sok esetben kaszkádoltan végigvonulnak az összes rétegen, több helyen kell módosítani.
    - Pl. felveszünk egy új adatbázis mezőt, akkor a GUI és az adatbázis réteg között minden réteget módosítani kell.
    - Teljesítmény
- Információs rendszerek
  - Nincsenek rétegek
    - Batch rendszerek
  - Kétrétegű architektúra
    - Kliens-szerver
  - Háromrétegű architektúra
- Kétrétegű architektúra
  - Kliens-szerver rendszerek, ma is elterjedten használt
  - Megosztott adatbázis (file-ok, DBMS)
  - Alkalmazások (hagyományos vagy 4GL nyelven)
  - Felépítés:

  - Előnyök
    - Az adatkarbantartás elkülöníthető az alkalmazásoktól.

        - A már meglévő adatokat több szempontból (nézetből) is megjeleníthetjük.
        - Kitűnő RAD támogatás, adatkötés
    - Hátrányok
        - Ha az adatintegritás kezelését „bedrótozzuk” az alkalmazásba
            - Keveredik a GUI-val
            - Ha több alkalmazást írunk, akkor duplikálódik az üzleti logika, több helyen kell karbantartani, stb.
            - Az adatbázist nem lehet megváltoztatni a régi alkalmazásokkal való kompatibilitás miatt
        - Az adatok integritása jól csak az adatbázis szintjén megoldható
            - Nincs mindig lehetőség tárolt eljárások írására
            - Ha van: nem objektumorientált, korlátozott lehetőségek
        - Az alkalmazás az adatbázis fizikai felépítését is figyelembe kell vegye, pedig neki csak a szemantikát kellene.
        - Optimalizálás denormalizálással
        - Megjelent a webes világ: jó lenne, ha az üzleti logikához csak egy webes megjelenítést lehetne kapcsolni a vastag kliens mellé: ezt nem teszi lehetővé.
    - Mikor használjuk
        - Egyszerűbb alkalmazások esetében használjuk bátran
        - Ekkor nem éri meg extra rétegekkel vacakolni
        - Ha nem akarunk üzleti logikát több alkalmazásban, alrendszerben felhasználni (vagy ha igen, akkor azt nem tudjuk/akarjuk tárolt eljárásokban implementálni)
- Háromrétegű architektúra
    - Rétegei
        - Felhasználói felület (Presentation)
        - Üzleti logika (Business, Domain)
        - Adatforrás (Data Source, Data Access)
            - Általában adatbázis, de lehet egy külső rendszer, ami szolgáltatást nyújt (webszolgáltatás, más alkalmazás, üzenetalapú rendszer)
    - Felépítés:

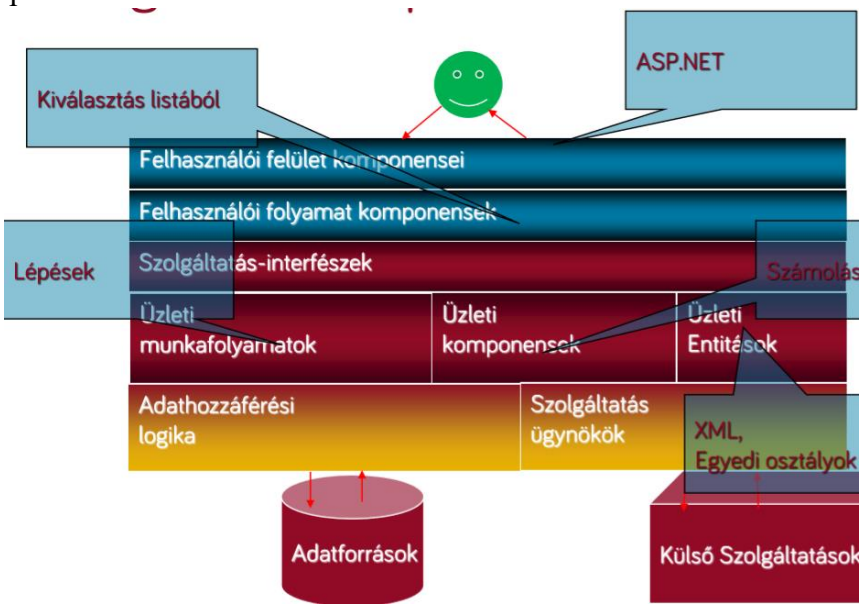


    - Az alkalmazáslogikai alréteg - opcionális
        - Egy interfész (homlokzat) a tartománymodellhez
        - Az interfész illeszkedik a GUI vezérlőelemeihez
        - A tartomány típusairól logikai típussá konvertál
    - Előnyök
        - Az üzleti logikai komponensek
            - Nincsenek szétszórva a GUI-ban
            - Mivel független a GUI-tól, könnyebben újrafelhasználhatók (pl. webes és vastag kliens)
            - Nincsenek duplikált részek
            - Automatizált tesztek könnyebben könnyebben készíthetők (pl. NUnit)

        - Az alkalmazás kevésbé függ a fizikai adatszerkezettől
            - A referenciális integritás alkalmazásfüggetlenül kezelhető
            - Az adatbázis átszervezhető az alkalmazástól függetlenül
        - Tranzakciókezelés a DBMS feladata – ne nekünk kelljen implementálni.
        - Ipari támogatottság
    - Hátrányok
        - Extra komplexitás, többletmunka
        - Nagyobb erőforrásigény
        - Az üzleti logikai komponenseket le kell képezni relációs modellre, ami nehéz.
            - Az objektumorientált adatbázisok jó felhasználási területe, de ezek teljesítőképessége kétséges.
- „Rétegek” és „rétegek”
    - Layers and tiers, avagy logikai és fizikai rétegek
    - Layer
        - Logikai réteg
        - Ezt tervezzük először, erről volt eddig szó
        - Célszerű lehet külön modulba (pl. .NET szerelvény vagy Java package) kitenni
    - Tier
        - Fizikai réteg. Más gépen fut, hálózati kommunikáció
        - A logikai régekre mondjuk meg, mely gépeken fussanak.
    - Ajánlások
        - Ne keverjük a kettőt, gyakran a „profik” is összekeverik
        - Minél kevesebb fizikai réteg – teljesítmény
        - Webes világban adott, előny a központi telepítés és karbantartás

*Rétegek tervezése**

- Rétegek és komponensek



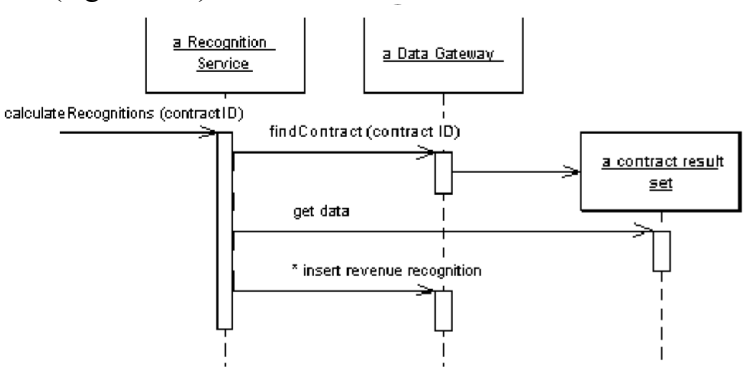
- Üzleti logikai réteg tervezése
    - Tervezési ajánlások
        - Azonosítsuk a szükséges komponenstípusokat!
        - Az egyes rétegek komponenseit tervezzük konzisztensre!
        - Vizsgáljuk meg a komponensek közötti kommunikációt, mielőtt fizikailag szétválasztanánk!
        - Minél kevesebb reprezentáció legyen ugyanarra az adatra!

- Különítsük el az adatbiztonsághoz és a többi rendszerhez kapcsolódó részeket!

  - Három alapvető minta a szervezésre
    - Transaction Script
    - Domain Model
    - Table Module

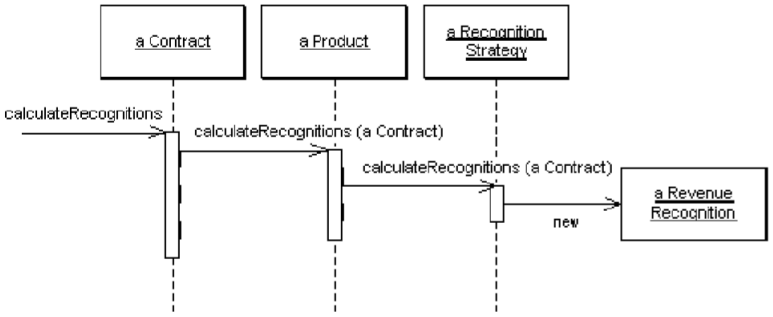
*Transaction Script**

- Legegyszerűbb, klasszikus, „procedurális” megoldás
- Egy eljárás minden műveletnek (felhasználói tranzakció), amit a felhasználó végrehajthat
  - UI rétegtől megkapja az adatokat
  - Tipikusan adatbázisba ír és onnan olvas
  - Számításokat végez, hívhat más szolgáltatásokat
- Előnyök
  - Egyszerű megérteni
  - A tranzakciók határai jól definiáltak
- Példa: hozam számítás (algoritmus)



*Domain Model**

- Objektumorientált megközelítés
- OO dekompozíció, együttműködő objektumok

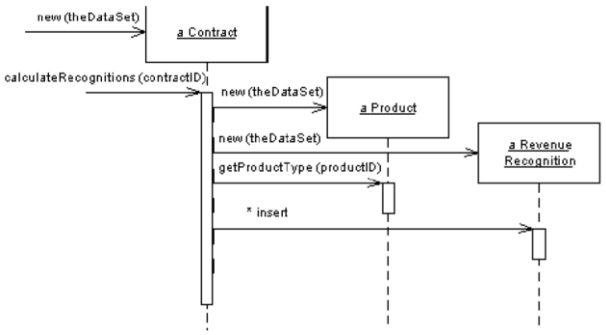


- Előnyök
  - Az OO megközelítés minden előnye
    - A komplexitást jobban kezeli
    - Karbantarthatóság, bővíthetőség, stb.

        - Interfészek használata
        - Polimorfizmus
        - Tervezési minták
- Hátrányok
    - Aki a klasszikus, procedurális világból érkezik: nehéz megszokni az OO gondolkodásmódot
    - Az üzleti objektumokat le kell képezni az adatbázis relációs modelljére
        - Komplex feladat
        - Ha nem használunk OO adatbázist: csomó plusz munka
- Előny a példa kapcsán: könnyebb bővíthetőség, ha be kell vezetni egy új számítási algoritmust:
    - Pontosan definiált, mit és hol (le kell származtatni egy új osztályt)
    - Csak egy helyen kell módosítani

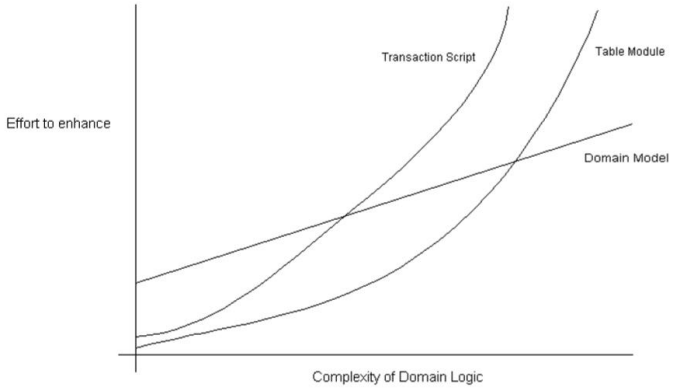
*Table Module**

- RecordSet ill. DataSet központú megközelítés. Microsoft technológiákra jellemző.
- Domain modelhez hasonlít de:
    - Domain model esetében külön objektum minden árucikknek.
    - Table module esetében csak egy objektum, ami kap egy RecordSet vagy DataSet objektumot, amin dolgozik.
- Példa:



- Előnyök
    - A fejlesztőkörnyezet támogatása
        - Adatkötés UI vezérlőelemekhez
        - XML támogatás (DataSet)
- Hátrányok
    - Nem igazán objektumorientált
        - Külön osztályba került az adat (RecordSet) és a rajta dolgozó műveletek.
        - Nem működnek az OO koncepciók, pl. a polimorfizmus, így a karbantarthatóság, kiterjeszthetőség romlik.

TS, DM, DT összehasonlítás



- Általában keveredik a TS és a DM, az számít, melyik a domináns.

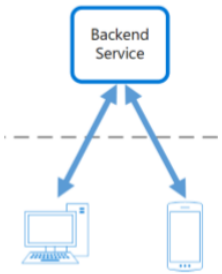
Felhő architektúrák

Kihívások

- Rendelkezésre állás
- Adatkezelés
- Design VS implementation
- Üzenetkezelés
- Felügyelet
- Teljesítmény és skálázás
- Rugalmasság
- Biztonság

*Backends for Frontends**

- b: egy webes alkalmazás üzleti logikáját a fejlesztés során leválasztjuk egy „backend” szolgáltatásba. Később, ahogy felmerül az igény mobilkliensek kiszolgálására, a meglévő szolgáltatásra építünk
- c: probléma:
    - A webes és mobil alkalmazások képességei eltérőek, ezért másféle háttérszolgáltatásokat igényelnek



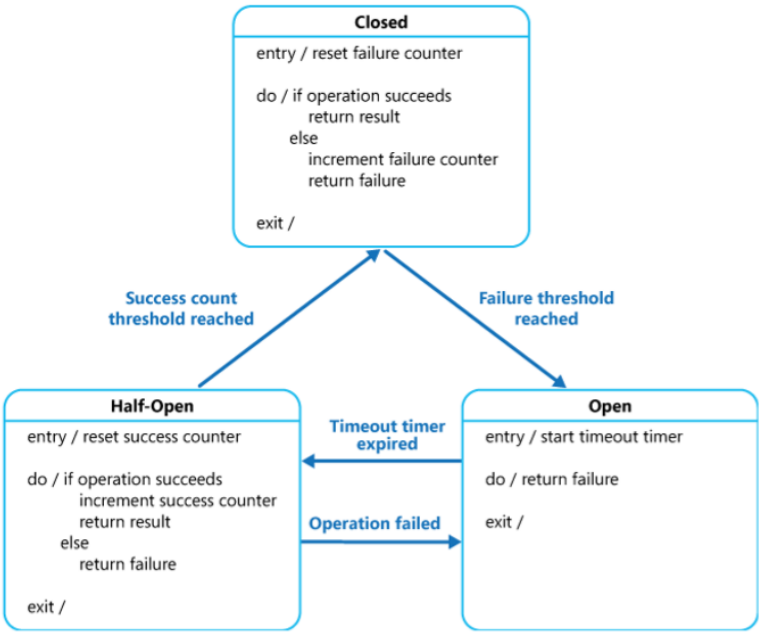
    - A webes és a mobil frontend egymással vetélkedő követelményeket támaszt a backend szolgáltatással szemben
    - Ez már akár 3 fejlesztő csapatot is érintő komplex egyeztetésekhez és kompromisszumokhoz vezethet
- d: megoldás válasszuk szét a webes és mobil backendet és optimalizáljuk őket a frontend technológiához igazítva
- e,f,g: --

- h: A frontendeket optimálisabb eltérő backend technológiával támogatni A backendben elszaporodnak a frontend specifikus elágazások
- i:

Előnyök	Hátrányok
Frontendhez igazodó (hatékonyabb) backend	Kód duplikáció
Gyorsabb fejlesztés	Hasonló igényű Frontendek mellett nem biztos, hogy megéri
	Fejlesztési overhead

*Circuit Breaker**

- a: circuit breaker (megszakító)
- b: egy elosztott környezetben a tranziens hibák kezelésére sokszor újra próbálkozást (retry minta) alkalmazunk. Ez azonban nem mindig működik
- c: Probléma: amennyiben a hiba nem tranziens, az ismétlési kísérletekkel túlterhelhetjük a hívott szolgáltatást/infrastruktúrát mellyel további leállási láncot idézhetünk elő, illetve megnehezítjük a szolgáltatás helyreállítását
- d: Megoldás: a megszakító minta proxy-ként illeszkedik be a szolgáltatás hívásba. Normál üzemben átengedi a hívásokat (closed állapot), amennyiben viszont hibát észlel, elkezdi számolni az újra próbálkozási kísérleteket és egy adott határ túllépését követően blokkolja az újabb próbálkozásokat (open állapot). Ez az állapot egy adott ideig (timeout) tart, amelyet követően egy átmeneti (half open) állapotba kerül a rendszer. Ez az állapot (adott számú hibás próbálkozásig) ismét átengedi a hívásokat. Sikeres hívásokat követően a rendszer végül visszaáll closed állapotba.
- e,f:



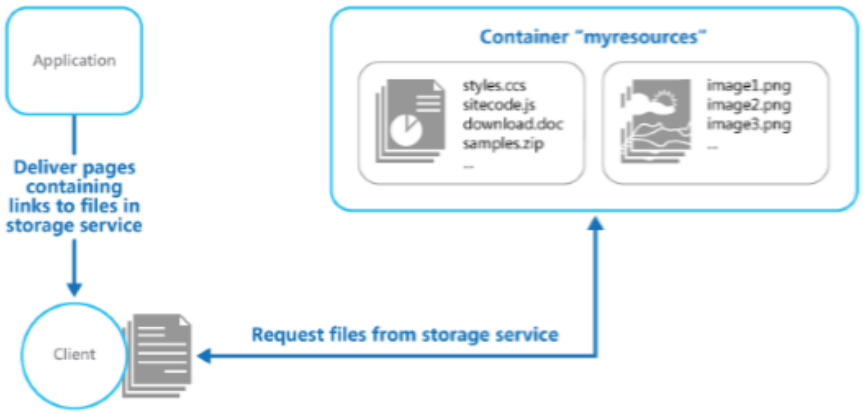
- g: Az open állapot fenntartásának ideje dinamikusan is növelhető
  - Az alkalmazásnak képesnek kell lennie a kapott kivételek kezelésére
  - Circuit breaker felelőssége a naplózás
  - A paraméterezésnek igazodnia kell a kapcsolódó szolgáltatás helyreállítási mintájához
  - Circuit breaker értelmezheti is az üzeneteket és adott hibák esetén azonnal open állapotba válthat
- i:

Előnyök	Hátrányok

Szolgáltatások túlterhelésének megelőzése	Lokális erőforrások elérése esetén általában túl nagy overhead
Azonnali hibajelzés (open állapotban)	

Static Content Hosting

- a. Static content hosting
- b: Egy webes környezetben gyakran szolgálunk ki olyan kéréseket, melyek eredménye általában egy változatlan, statikus fájl (képek, HTML fájlok stb.)
- c: Probléma: a webszerverek sokszor a felhőszolgáltatás legdrágább komponensei, a statikus fájlok közvetlen kiszolgálása a webszerverről más, dinamikus kérések elől veszi el az erőforrásokat
- d: Megoldás: szervezzük ki a statikus tartalmakat ezek kiszolgálására dedikált szolgáltatásokba (blob storage, CDN stb.).



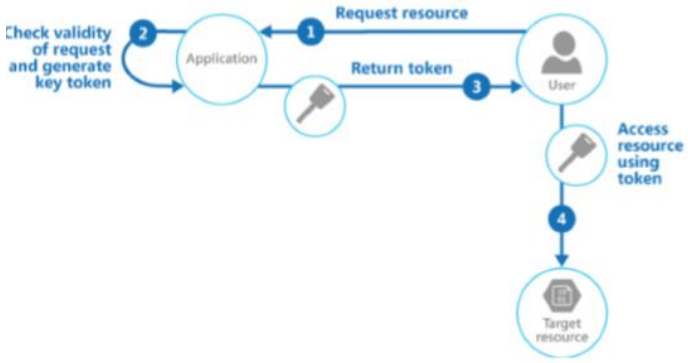
- e, f:
  - A statikus fájlokat telepítjük az arra dedikált szolgáltatásba
  - Alkalmazásunkban módosítjuk a hivatkozásokat az érintett fájlok esetében
- g. :
  - A hosztoló szolgáltatásnak támogatnia kell a webes alkalmazással azonos végponttípusokat (HTTP / HTTPS)
  - A hosztoló szolgáltatás nem feltétlenül támogatja saját domain nevek használatát
  - Kezelni kell a statikus fájlok telepítését az alkalmazástelepítésekkel együtt
  - A legjobb teljesítmény érdekében érdemes CDN-t alkalmazni
  - A védett erőforrásokhoz a hozzáférést szabályozni kell (lásd „valet key minta”)
- i:

Előnyök	Hátrányok
Nagyobb átbocsátóképesség az alkalmazásszerveren	A hosztoló szolgáltatás nem feltétlenül támogatja saját domain nevek kezelését
Statikus tartalmak hatékonyabb kiszolgálása	A hozzáféréskezelés nem feltétlenül megoldott
	A hosztoló szolgáltatás IP címe régiónként változhat
	Az alkalmazás telepítése komplexebbé válik

*Valet Key**

- a: Valet Key

- b: Webes környezetben a klienseknek gyakran kell adatfolyamokat, fájlokat letölteni a szerverről, vagy éppen feltölteni a szerverre. Ezen műveletek során a kliens jogosultságait az alkalmazásszerver ellenőrzi. A gyakorlatban a konkrét fájlok sokszor eleve olyan szolgáltatásban kerülnek tárolásra mely akár a közvetlen, kliens általi manipulációt is támogatná.
- c: Probléma: A fájlok, adatfolyamok kiszolgálása értékes erőforrásokat vesz el az alkalmazásszerverünktől. Jó ötlet lenne megengedni, hogy a kliens közvetlenül a végső tárhellyel kommunikálva végezze a le/feltöltést ezzel azonban elveszíthetnénk az ellenőrzést az adatok felett. Egy megbízhatatlan kliens visszaélve a lehetőséggel hozzáférhetne olyan adatokhoz is, melyekre nem szeretnénk ha rálátása lenne.
- d: Megoldás: Az adathozzáférés ellenőrzését úgy kell megoldanunk, hogy az adatokat kiszolgáló rendszert ne kelljen közvetlenül bekapcsolnunk a saját egyedi jogosultságkezelési rendszerünkbe. A megoldás során a kliens számára olyan – időben korlátos – kulcsokat állítunk ki, melyeket az adatkiszolgáló önmagukban tud validálni, s ez alapján engedélyezni, vagy letiltani az igényelt műveletet.
- e,f:



- g. :
  - A kulcs érvényessége időben korlátozható, maga a kulcs visszavonható (pl. a sikeres műveletet követően)
  - A kulcs egy erőforrásra, vagy akár erőforráscsoportra is vonatkozhat
  - Titkokat mindig titkosított csatornán adjunk át
- h: Azure Blob, SAS token Amazon S3
- i:

Előnyök	Hátrányok
Alkalmazásszerver tehermentesítése	Korlátozottan alkalmazható, ha az adatok tartalmát validálni/transzformálni kell
Jobb skálázhatóság	Meglévő alkalmazások utólagos átírása körülményes
Adatmozgatási költségek csökkentése	Elsősorban nagy mennyiségű fájlműveletnél jelent érdemi előnyt

Enterprise Architecture Methodologies

*Zachmann Framework***

- Rule 1: Columns have no order
- Rule 2: Each column has a simple, basic model
- Rule 3: Basic model of each column is unique
- Rule 4: Each row represents a distinct view
- Rule 5: Each cell is unique

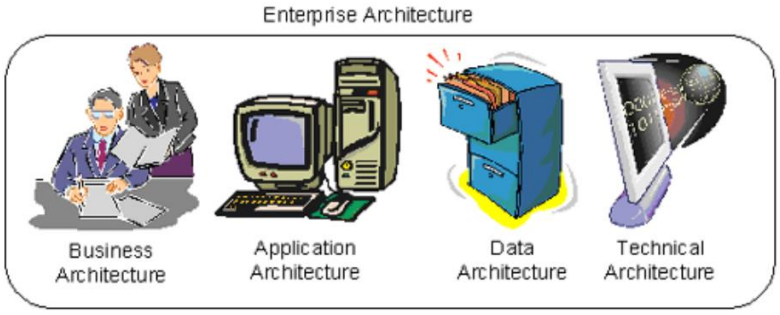
- Rule 6: Combining the cells in one row forms a complete description from that view

VA Enterprise Architecture	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*	Based on work by John A. Zachman
SCOPE (CONTEXTUAL) *Planner*	Things Important to the Business Entity = Class of Business Thing	Processes Performed Function = Class of Business Process	Business locations Node = Major Business Locations	Important Organizations People = Major Organizations	Events Significant to the Business Time = Major Business Event	Business Goals and Strategy Ends/Means = Major Business Goals	SCOPE (CONTEXTUAL) *Planner*
ENTERPRISE MODEL (CONCEPTUAL) *Owner*	Semantic Model Ent = Business Entity Rel = Business Relationship	Business Process Model Proc = Business Process I/O = Business Resources	Business Logistics System Node = Business Location Link = Business Linkage	Work Flow Model People = Organization Unit Work = Work Product	Master Schedule Time = Business Event Cycle = Business Cycle	Business Plan End = Business Objective Means = Business Strategy	ENTERPRISE MODEL (CONCEPTUAL) *Owner*
SYSTEM MODEL (LOGICAL) *Designer*	Logical Data Model Ent = Data Entity Rel = Data Relationship	Application Architecture Proc = Application Function I/O = User Views	Distributed System Architecture Node = IS Function Link = Line Characteristics	Human Interface Architecture People = Role Work = Deliverable	Processing Structure Time = System Event Cycle = Processing Cycle	Business Rule Model End = Structural Assertion Means = Action Assertion	SYSTEM MODEL (LOGICAL) *Designer*
TECHNOLOGY MODEL (PHYSICAL) *Builder*	Physical Data Model Ent = Segment/Table Rel = Pointer/Key	System Design Proc = Computer Function I/O = Data Elements/Sets	Technology Architecture Node = Hardware/Software Link = Line Specifications	Presentation Architecture People = User Work = Screen Format	Control Structure Time = Execute Cycle = Component Cycle	Rule Design End = Condition Means = Action	TECHNOLOGY MODEL (PHYSICAL) *Builder*
DETAILED REPRESENTATIONS (OUT-OF-CONTEXT) *Sub-Contractor*	Data Definition Ent = Field Rel = Address	Program Proc = Language Statement I/O = Control Block	Network Architecture Node = Addresses Link = Protocols	Security Architecture People = Identity Work = Job	Timing Definition Time = Interrupt Cycle = Machine Cycle	Rule Design End = Sub-Condition Means = Step	DETAILED REPRESENTATIONS (OUT-OF-CONTEXT) *Sub-Contractor*
FUNCTIONING ENTERPRISE	Data Ent = Rel =	Function Proc = I/O =	Network Node = Link =	Organization People = Work =	Schedule Time = Cycle =	Strategy End = Means =	FUNCTIONING ENTERPRISE
	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*	

		Data	Function	Network	People	Time	Motivation
REQUIREMENTS ANALYSIS	Scope Investor	List of Things Important to Business Entity=Class of Business Thing	List of Processes the Business Performs Function=Class of Business Process	List of Locations Important to Business Node=Major Business Location	List of Organizations Important to Business Agent=Class of Agent	List of Events Significant to Business Time=Major Business Event	List of Business Goals/Strategies End/Means=Major Business Goal
	Enterprise Model Owner	e.g., Entity Relationship Diagram Ent=Business Entity Rel=Business Rule	e.g., Function Flow Diagram Function=Business Process	e.g., Logistics Network Node=Business Location Link=Business Linkage	e.g., Organization Chart Agent=Org Unit Work=Work Product	e.g., Master Schedule Time= Business Event Cycle=Business Cycle	e.g., Business Plan End=Business Objectives Means=Business Strategy
DESIGN	Information System Model Designer	e.g., Data Model Entity=Data Entity Relationship= Data Relationship	e.g., Data Flow Diagram Funct=Appl Function Arg=User Views	e.g., Distributed System Architecture Node=Info Sys Funct Link=Line Char	e.g., Human Interface Structure Agent=Role Work=Job	e.g., Processing Structure Time=Trigger Cycle=Component Cycle	e.g., Knowledge Architecture End=Criterion Means=Option
DEVELOPMENT	Technology Model Builder	e.g., Data Design Entity=Segment/Row Relationship=Pointer/Key	e.g., Structure Chart Funct=Computer Funct Arg=Screen/Device Formats	e.g., System Architecture Node=Hardware/ System Software Link=Line Specification	e.g., Human/ Technology Interface Agent=User Work=Job	e.g., Control Structure Time=Execute Cycle=Component Cycle	e.g., Knowledge Organization End=Condition Means=Action
	Components Subcontractor	e.g., Data Definition Description Ent=Fields Rel=Addresses	e.g., Program Funct=Language Stmts Arg=Control Blocks	e.g., Network Architecture Node=Addresses Link=Protocols	e.g., Security Architecture Agent=Identity Work=Transaction	e.g., Timing Definition Time=Interrupt Cycle=Machine Cycle	e.g., Knowledge Definition End= Means
	Functioning System	e.g., Data	e.g., Function	e.g., Network	e.g., Organization	e.g., Schedule	e.g., Strategy

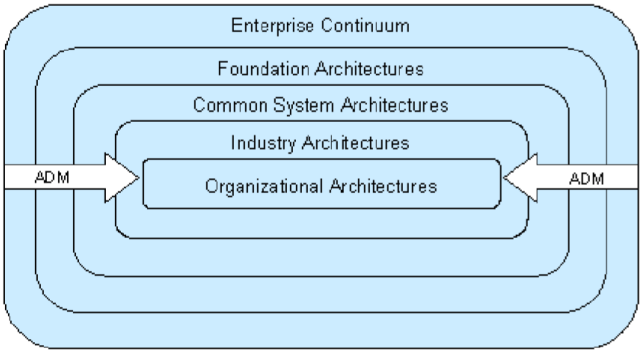
- Strength- End result (table)
- Weaknesses – No process

*The Open Group Architecture Framework (TOGAF)***

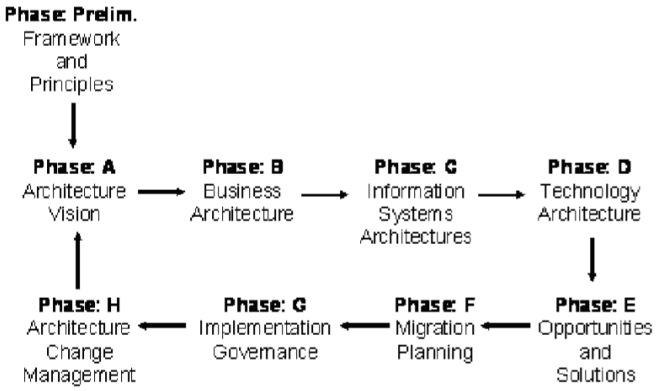


1. Business architecture—Describes the processes the business uses to meet its goals
2. Application architecture—Describes how specific applications are designed and how they interact with each other
3. Data architecture—Describes how the enterprise datastores are organized and accessed
4. Technical architecture—Describes the hardware and software infrastructure that supports applications and their interactions

- Enterprise Continum



- TOGAF – Arch. Development Method



- Lépések
  - 1. Develop baseline data-architecture description
  - 2. Review and validate principles, reference models, viewpoints, and tools
  - 3. Create architecture models, including logical data models, datamanagement process models, and relationship models that map business functions to CRUD (Create, Read, Update, Delete) data operations
  - 4. Select data-architecture building blocks
  - 5. Conduct formal checkpoint reviews of the architecture model and building blocks with stakeholders
  - 6. Review qualitative criteria (for example, performance, reliability, security, integrity)

    - 7. Complete data architecture
    - 8. Conduct checkpoint/impact analysis
    - 9. Perform gap analysis
- Strengths and Weaknesses
    - Strengths
        - Provides a process for developing an architecture. > Flexible so can be tailored to a company's organization
    - Weakness
        - Open/generic, no specific end result

Dokumentálás

*Dokumentáció típusok**

- Tervezési dokumentációk
    - Követelmények
    - Specifikáció
    - Rendszerterv
- Fejlesztői dokumentációk
    - Architektúra
    - Implementáció
    - Tesztelés
- Felhasználói dokumentáció
    - Konfigurálás és telepítés
    - Karbantartás
    - Felasználói

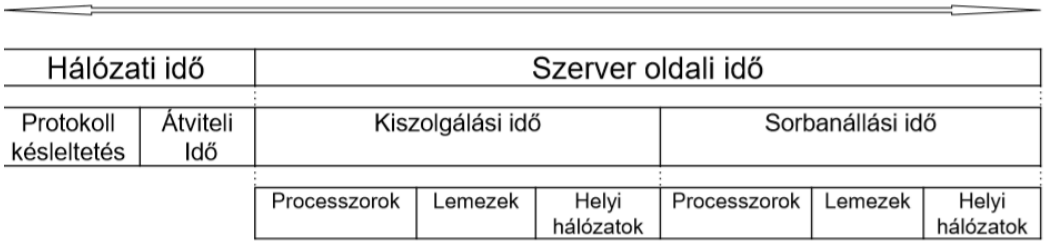
*Dokumentáció felépítése**

- A dokumentáció belső felépítése
    - A belső felépítése függ a rendszerrel szemben támasztott követelményektől.
    - Absztraktnak és érthetőnek kell lennie. Alapvetően két célt szolgál: Az új fejlesztők könnyen tudják megérteni és lehessen alapul venni a rendszerterv készítése során
    - Egy architektúra dokumentációja egyszerre legyen előíró és leíró. Előíró a tervezőkkel kapcsolatban és leíró a felhasználók számára
- Érdekcsoportok- Stakeholderek
- Dokumentációval szemben támasztott legfontosabb követelmények (fejlesztői)
- Dokumentációval szemben támasztott legfontosabb követelmények (felhasználói)
- Nézet
- Fontosabb nézetek dokumentálása
- 3 lépcsős módszer a számunkra fontos nézetek kiválasztásához
- Információ mélysége
- Kombináljuk a nézeteket

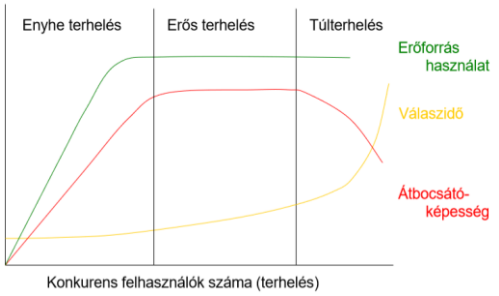
Teljesítmény

*Teljesítmény mérőszámok, ábra enyhe és erős túlterhelés melletti viselkedésről***

- Válaszidő
  - Kérés teljesítéséhez szükséges idő
  - Szerver oldalon
    - Kérés megérkezésétől válasz elküldéséig
- Kliens oldalon
  - Kérés elküldésétől válasz megérkezéséig
  - Válasz első byte-jáig (TTFB: Time To First Byte)
  - Válasz utolsó byte-jáig (TTLB: Time To Last Byte)
- Válaszidő



- Átbocsátóképesség
  - Rendszer által egységnyi idő alatt teljesített tranzakciók (munkaegységek) száma
- Erőforrás-használat
  - Memória
  - CPU
  - Merevlemez
  - Hálózat
  - Stb.



- Teljesítménybefolyásoló tényezők
  - Teljesítménymetrikák sok tényezőtől függnek
  - Teljesítménybefolyásoló tényezők keresése
    - Statisztikai módszerek
    - Hipotézisvizsgálatok
    - Stb.
- Kiértékelési folyamat választása

	Analitikus modellezés	Szimuláció	Mérés
Életciklus szakasz	Bármikor	Bármikor	Prototípus
Szükséges idő	Alacsony	Közepes	Változó
Eszközök	Elemző	Szimulációs nyelvek	Műszerezés
Pontosság	Alacsony	Közepes	Változó
Optimalizálás	Egyszerű	Közepes	Bonyolult
Költség	Alacsony	Közepes	Magas
Eladhatóság	Alacsony	Közepes	Magas

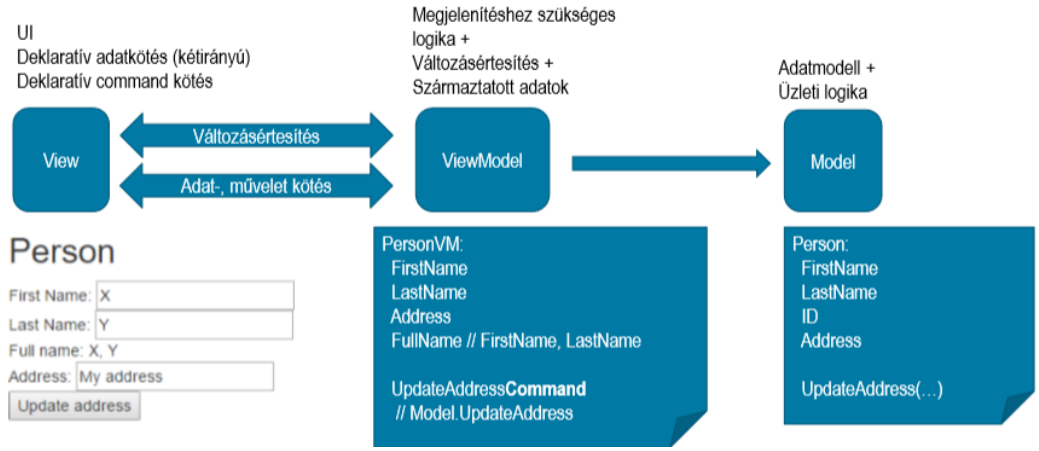
*Kapacitástervezési lépések***

- Kapacitás tervezési lépések
  - 1. A környezet pontos azonosítása
  - 2. A működési terhelés jellemzése
  - 3. A terhelési modell paramétereinek azonosítása
  - 4. A jövőbeni terhelés evolúciója
  - 5. A performancia modell fejlesztése
  - 6. A performancia modell validálása
  - 7. A szolgáltatás performanciájának predikálása
  - 8. Jövőbeni Szenáriók alakulása

Webes architektúrák

*MVVM –Model-View-ViewModel**

- Architektúrálisminta
- Más mintákkal szoros kapcsolat (Model-View-Controller, Model-ViewPresenter, Document-View, PresentationModel)
- Eredetileg WPF-hezdefiniálták
- Célja: eseményvezérelt programozási környezetben a nézet és a logika szétválasztása



- Speciális programozási környezet
  - Deklaratív adatkötés –kétirányú
  - Változásértesítés
  - Deklaratív command kötés –kétirányú
- View és ViewModel elszapárálása Tesztelhetőség
- Értékelés
  - Előnyök
    - Tesztelhetőség
    - Karbantarthatóság
    - Kód újrafelhasználás
    - Fejlesztők és UI designerek együttműködésének segítése
  - Hátrányok
    - Egyszerű UI →Overkill
    - Bonyolult UI →Memória

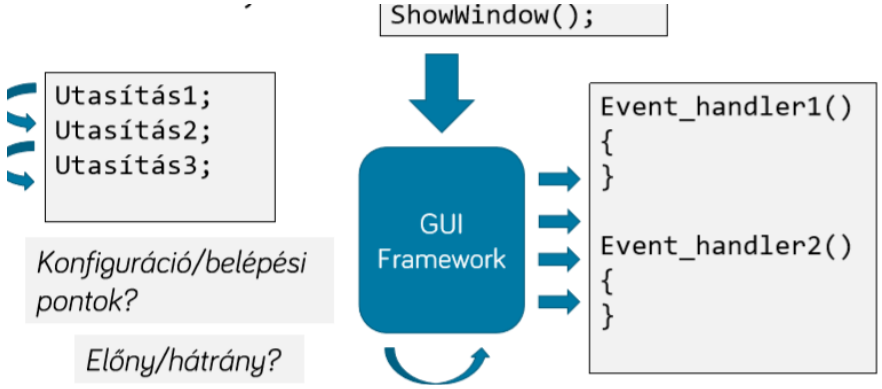
IoC

SOLID elvek

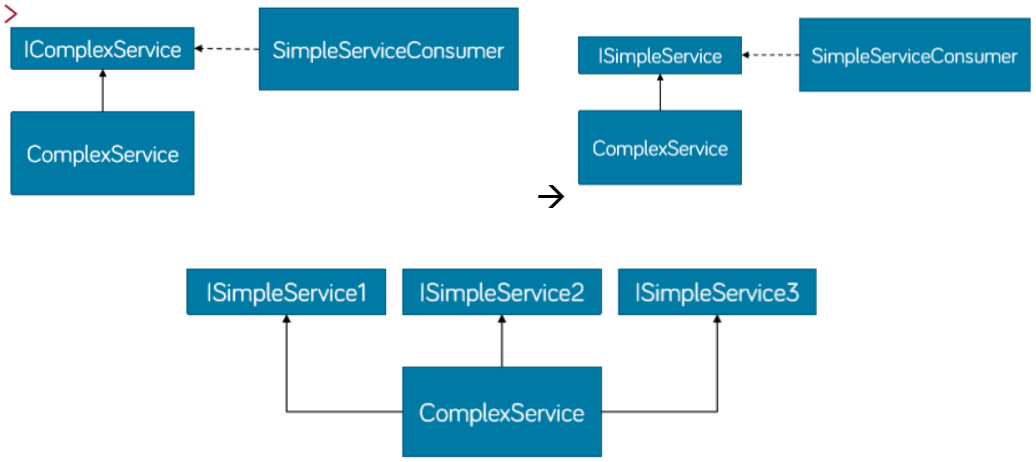
S	**Single Responsibility principle** Egy osztály csak egyetlen funkcionalitásért legyen felelős, de azért teljes mértékben (encapsulation).
O	**Open-closed principle** Minden modul nyitott a kiterjesztésre, de zárt a módosításra. (Absztrakció, interfészek, öröklés.)
L	**Liskov Substitution principle** Leszármazott típus állhat egy típus helyén. (Interfészek, öröklés.)
I	**Interface segregation principle** Egy komponens ne függjön olyan funkcióhalmaztól (metódusoktól), amelyet nem használ.
D	**Dependency inversion principle** Ne legyen szoros függőség a konkrét implementációk között, helyette az interfészek hivatkozzanak egymásra.

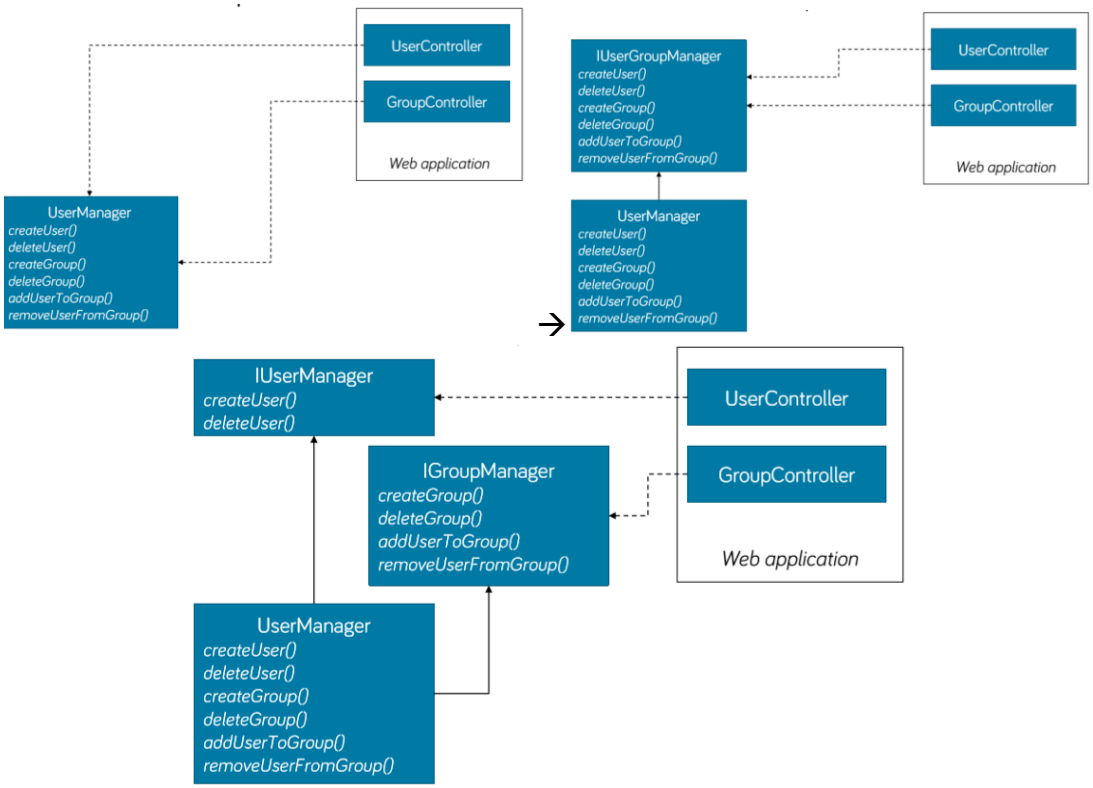
Inversion Control (IoC)

- Flow inversion: a saját kódunk meghívásának szabályozása (pl. Procedurális kód vs. GUI alkalmazás)

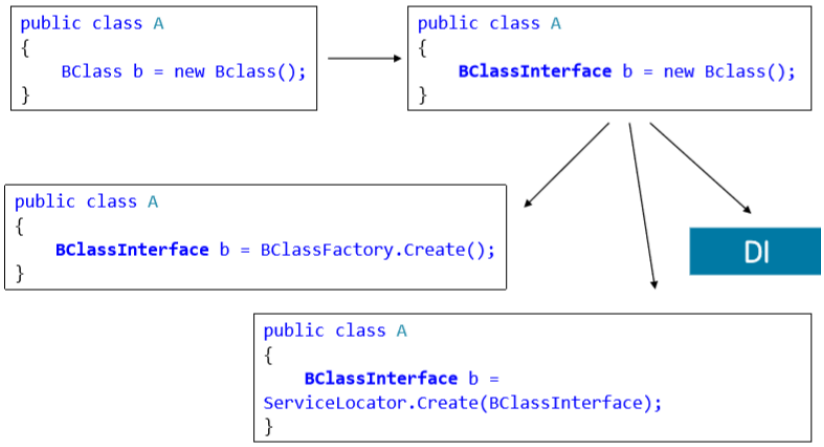


- Interface inversion:
  - Egy interfész felhasználója szabja meg, hogy mi legyen az interfészen
  - Egy komponens inkább több interfészt valósítson meg, minthogy túl bonyolult interfészeket vezessünk be





- Create inversion

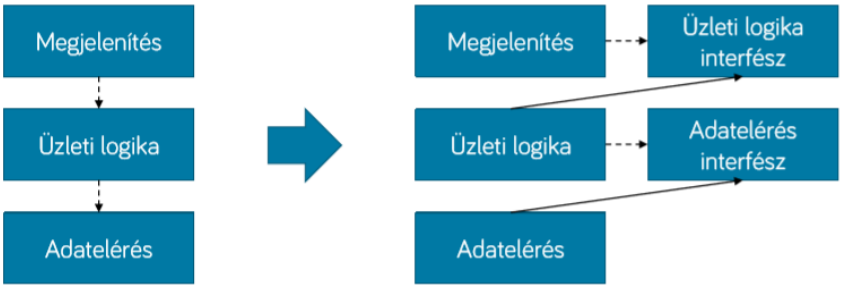


Lazán csatolt komponensek

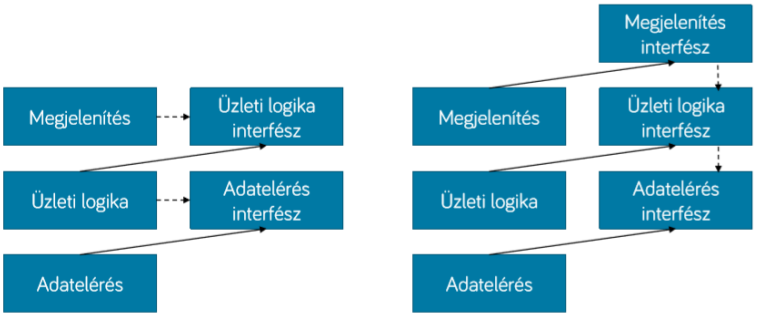
- Függőség (dependency):
  - A komponens függ B komponenstől, ha B változása után A-t is változtatni kell (pl. újrafordítani)
  - Futásidejű függőség feloldás / futásidejű kötés (late binding): egy komponens a függőségeire azok interfészével hivatkozik. Csak futási időben dől el, hogy melyik valós komponens kerül betöltésre.
- Laza csatolás
  - Lazán csatolt komponensek: egy komponens változtatásakor csak azokat a további komponenseket kell változtatni, amelyek valóban szorosan hozzátartoznak
    - A laza csatolás az absztrakción alapul: interfészeket vezetünk be, ahol lehet ezeket használjuk a mögöttes megvalósítás elrejtésével
  - (Interfészeket használunk, de példányosítani mégis kell valahol – ld. később)
  - A laza csatolás előnyei
    - Könnyebb kiterjeszthetőség

        - Tesztelhetőség –Egységteszt (unit test) vs. integrációs teszt –Mocking lehetősége
        - Lehetővé teszi a függőségek futásidőben történő feloldását
        - Támogatja a párhuzamos fejlesztést és karbantarthatóságot
    - Szorosan csatolt komponensek mindig rosszak?
        - Nem ~ feladat mérete
        - Refaktorálás lehetősége
        - Kell minden komponenshez mindig interfész? (…legyen legalább két implementációja egy interfésznek)
- Rétegzett architektúra
    - A rétegek cserélhetők, de a komponensek szorosan csatoltak
    - Rétegzett architektúra ≠ lazán csatolt komponensek

- Rétegzett architektúra
    - Absztrakció bevezetése
    - Ez már lazán csatolt?

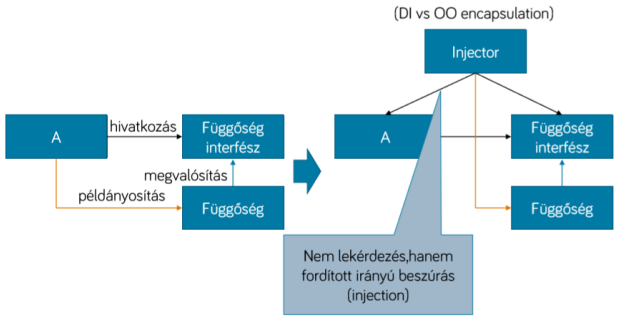


- Rétegzett architektúra
    - Lazán csatolt esetben:
        - Nem csak az alsóbb rétegek nyújtanak interfészt (tesztelhetőség)
        - Az interfészek hivatkoznak egymásra

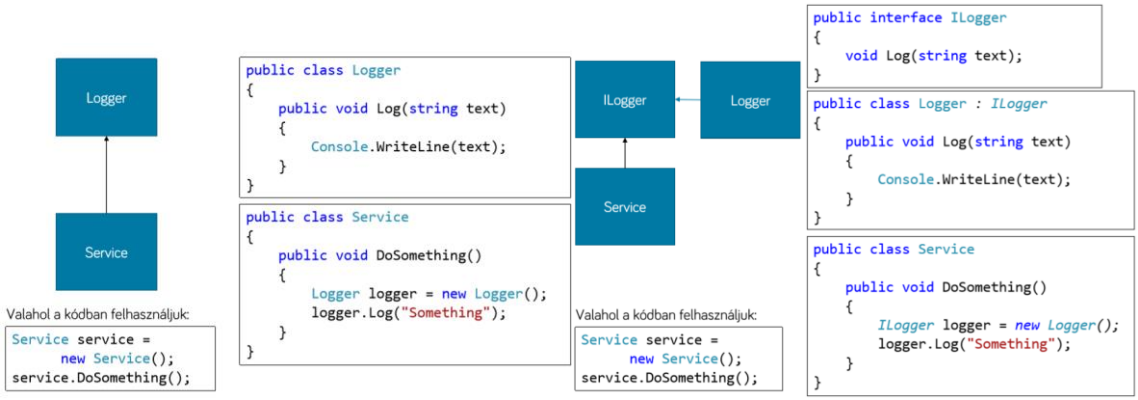


*Dependency Injection***

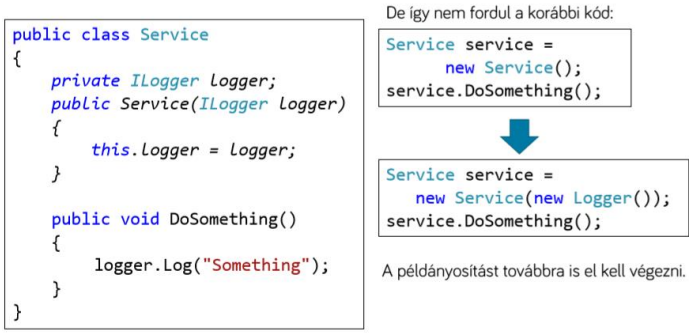
- Olyan tervezési elvek, minták összessége, amelyek lehetővé teszik lazán csatolt komponensek fejlesztését.
- Injector komponens (singleton):



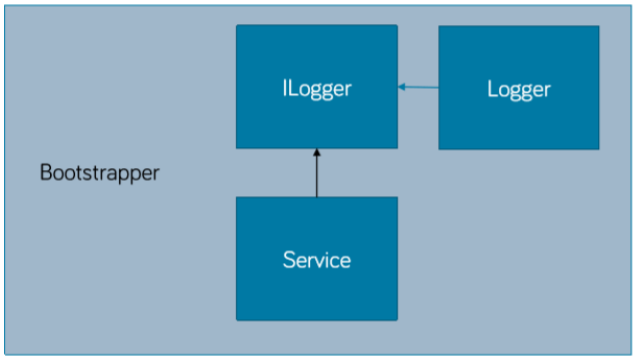
- Minták a DI megvalósítására:
  - Constructor injection
  - Interface (method) injection
  - Setter (property) injection
  - Service locator
  - Ambient context
  - Factory tervezési minta
  - …
- IoC konténer:
  - Mivel a DI megvalósítja a IoC alapelvet, az Injectort szoktuk IoC konténernek hívni
  - Feladatai:
    - Ezen keresztül lehet függőség betöltése, életciklus kezelés
    - Automatikus függőség feloldás
- DI a gyakorlatban
  - Manuális DI



  - Constructor Injection



    - Hol példányosítsunk?
      - Külön komponens (független a Service-től, Logger-től), ami feloldja a függőségeket
      - Csak a példányosítást szervezzük ki, a logikát (DoSomething meghívását) ne!



- Bootstrap komponens

```
public class AppBootstrapper
{
    private static AppBootstrapper _Current;
    public static AppBootstrapper Current
    {
        get
        {
            if (_Current == null)
            {
                _Current = new AppBootstrapper();
                _Current.ComposeObjects();
            }
            return _Current;
        }
    }
    public Service Service;
    public ILogger Logger;
    public void ComposeObjects()
    {
        Logger = new Logger();
        Service = new Service(Logger);
    }
}
```

A logika meghívása:

```
Service service =
AppBootstrapper.Current.Service;
service.DoSomething();
```

Feladatok:
- Példányosítás
- Függőségek feloldása
- Életciklus menedzsment

- DI konténer
  - Globálisan elérhető
  - Visszaad olyan komponenseket, amelyekben már feloldotta a függőségeket •DI keretrendszerek
    - A példában a DI konténerünkben a függőségek feloldását kézzel kell implementáljuk
      - Ha pl. constructor injection helyett más módon valósítjük meg a DI-t, akkor frissíteni kell a bootstrapper kódját
    - Vannak általános keretrendszerek, amelyek ezt automatikusan megoldják
- Setter (property) injection

```
public class Service : IService
{
    public ILogger Logger { get; set; }
    public Service()
    {
     }

    public void DoSomething()
    {
        Logger.Log("Something");
    }
}

public interface IService
{
    void DoSomething();
}
```

```
public void ComposeObjects()
{
    Logger = new Logger();
    var service = new Service();
    service.Logger = Logger;
    this.Service = service;
}
```

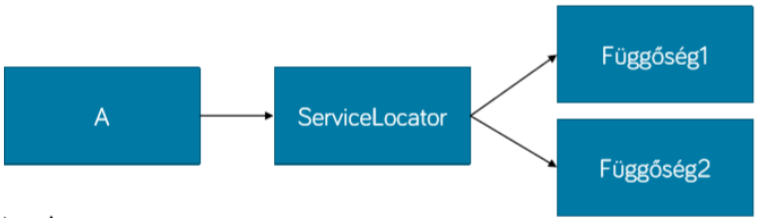
Manuálisan kell frissíteni, mert a DI megvalósítása változott.

- Interface (method) injection

```
public void ComposeObjects()
{
    Logger = new Logger();
    var service = new Service();
    service.Inject(Logger);
    this.Service = service;

}
```

- További DI minták
  - Service locator ~ DI
    - Egy komponens gondoskodik a függőségek betöltéséről
    - De a hívó félnek kell specifikálnia, mit és hol keressen



- Ambient context
  - ~Statikusan gettereken keresztül elérhető objektumok (pl. Thread.CurrentThread)
- Factory tervezési minta:
  - A példányosítás helyett Factory metódusokat hívunk
- Tesztelés – unit tesztekkel
  - NUnit + Moq

```
[TestFixture]
public class ServiceTests
{
    [Test]
    public void Test_DoSomething()
    {
        var loggerMock = new Mock<ILogger>();
        loggerMock
            .Setup(logger => logger.Log(It.IsAny<string>()))
            .Callback<string>((text) =>
                 Console.WriteLine("MOCKED " + text));
        var sut = new Service(loggerMock.Object);
        Assert.DoesNotThrow(sut.DoSomething);
    }
}
```

- DI keretrendszerek
- Nagyon sokféle, általában más szolgáltatásokat is nyújtanak
- .NET-hez, pl: Unity, Ninject, Spring.NET, StructureMap, MEF
- Java: Spring, Guice, Dagger (Android)
- Cél: a függőségfeloldás minél nagyobb mértékű automatizálása