Bevezetés

A szoftverrendszerek méretének és bonyolultságának növekedésével a tervezési kérdések már túlmutatnak az algoritmusok és adatstruktúrák létrehozásán, a teljes rendszer megtervezése és specifikációja egy új típusú feladattá vált. A strukturális kérdések a következőket foglalják magukban: a teljes felépítés és a globális vezérlő szerkezet; a kommunikációs protokollok, szinkronizáció, és az adatelérés; funkcionalitás hozzárendelése az egyes tervezési elemekhez; a rendszer fizikai elosztása; a tervezési elemek komponálása; skálázhatóság és teljesítmény; a megfelelő tervezési módszerek kiválasztása. Ez a szoftverarchitektúra szintű tervezés.

Egyre jobban nyilvánvaló, hogy a hatékony szoftverfejlesztéshez szükség van architektúrális szofvertervezési eszközökre. Először is fontos, hogy felismerjünk gyakori mintákat, hogy a rendszerek közötti magas-szintű kapcsolatokat megérthessük, és így az új rendszereketa régiek alapján tudjunk felépíteni. Másodszor, a megfelelő architektúra kiválasztása gyakran kritikus a szoftver rendszer tervezésének sikerességét tekintve; a rossz tervezési döntések katasztrofális következményekkel járhatnak. Harmadszor, a szoftverarchitektúrák alapos megértése lehetővé teszi a mérnök számára, hogyelvek alapján válasszon a különböző tervezési módszerek között. Negyedszer, egy architeturális rendszerleírás gyakran nélkülözhetetlen egy összetett rendszer elemzéséhez és magasszintű tulajdonságainak leírásához.

Általános Architektúra Típusok

A továbbiakban megvizsgálunk néhány jellegzetes, széles körben használt architektúrális stílust. A célunk, hogy illusztráljuk az architekturális lehetőségek gazdag választékát és hogy megmutassok egy stílus használatának hátrányát más stílusokhoz képest. Ahhoz, hogy a módszerek közötti különbségeket jól le tudjuk írni, szükségünk van egy közös keretrendszerre, amelyen keresztül nézhetjük azokat. A használandó keretrendszer lényege, hogy egy rendszer architektúráját különböző számítási komponensek (vagy egyszerűen komponensek) gyűjteményeként kezeljük, amikhez hozzátesszük a komponensek közötti interakciók (konnektorok) leírását. Grafikusan ábrázolva ez az absztrakt leírás egy olyan gráfot eredményez, amelyben a csomópontok reprezentálják a komponenseket,az élek pedig a konnektorokat. Látni fogjuk, hogy az élek által reprezentált interakciók lehetnek akár eljáráshívások, eseményküldések, adatbázis lekérdezések, vagy akár csővezetékek is.

Csővezetékek és szűrők (pipes and filters)

A csővezeték és szűrő stílusban mindegyik komponensnek vannak bemenetei és kimenetei. Egy komponens beolvassa a bemenetein érkező adatfolyamot majd a kimenetein adatfolyamokat hoz létre, ezáltal kézbesítve a végeredmény egy példányát eredeti sorrendben

Ą komponenseket „szűrőknek” (filter) nevezzük. A konnektorok pedig csővezetékként viselkednek, az adatfolyamot továbbítják egy szűrő kimeneteiről más szűrők bemeneteire. Ezért a konnektorokat „csővezetékeknek” (pipe) nevezzük. A stílus legfontosabb invariánsai között meg kell említeni, hogy a szűrőknek független entitásoknak kell lenniük: nem szabad az állapotokat megosztani különböző szűrők között. Egy másik fontos tulajdonság, hogy a szűrők nem tudják azonosítani a csővezetékeiket.

A szűrők és csővezetékek típusú rendszereknek sok fontos tulajdonsága van. Először is, a tervező könnyen megértheti a rendszer teljes bemenet-kimenet viselkedését, hiszen azok a különálló szűrők bement-kimenet működésén alapulnak. Másodszor, támogatják az újrafelhasználást, mert tetszőleges két szűrő összekapcsolható, amennyiben összeegyeztethetők az adatfolyam definícióik. Harmadszor, a rendszerek könnyen karbantarthatók, ill. kibővíthetők, mert új szűrőket lehet a rendszerhez adni, illetve a régi szűrők kicserélhetők újakra. Negyedszer, a rendszeren speciális elemezéseket lehet végrehajtani, például az áteresztőképesség, vagy deadlock vizsgálata. Végül, az ilyen típusú rendszerek támogatják a konkurens végrehajtást. Minden szűrő egy külön feladatként definiálható, amik esetleg párhuzamosan is végrehajthatók.

Megvannak a hátrányai is. Először is, ezek a rendszerek gyakran mégis az adatok kötegelt feldolgozását eredményezik., továbbá különösen alkalmatlanok interaktív alkalmazások készítésére. Másodszor, karban kell tartani a kapcsolatot minden olyan rendszer között, amelyek csővezetékekkel kapcsolódnak, még akkor is, ha azok teljesen elkülönülnek. Harmadszor, az implementációtól függően, az adatátvitelben néha egy alacsonyabb szintet kell használni, ami azt eredményezi, hogy minden szűrőben külön kódolni és dekódolni kell az adatot. Ez egyrészt a teljesítmény csökkenéséhez, másrészt az egyes szűrők komplexitásának a növekedéséhez vezet.

Adatabsztraktció és Objektum-Orientált Szervezés

Ebben a típusban az adat reprezentáció és az adathoz tartozó primitív műveletek egy absztrakt adattípusba (objektum) lettek összefogva. Az objektumok függvényeken és eljáráshívásokon keresztül lépnek kapcsolatba egymással. Ennek az architektúra típusnak két fontos eleme a következő: (a) az objektum felelőssége, hogy megőrizze saját reprezentációjának integritását (általában azáltal, hogy a típusban definiált invariánsokat ellenőrzi), (b) az adatreprezentáció el van rejtve a többi objektum elől.

Mivel az objektum elrejti a megvalósítás részleteit a kliensek elől, ezért lehetővé válik az implementáció módosítása anélkül, hogy a kliensek változnának. Továbbá, mivel az adatokat kezelő függvények egy halmazát magával az adattal együtt kezeljük, a tervezőnek lehetősége nyílik a különböző problémákat szétosztani egymással együttműködő részek számára.

A legjelentősebb hátrány, hogy az objektumoknak pontosan ismerniük kell egymást ahhoz, hogy kapcsolatba tudjanak lépni egymással. További mellékhatások is megjelenhetnek: ha A használja B-t és C szintén használja B-t, akkor C hatása B-re nem várt hatással lehet A-ra is és fordítva.

Esemény-alapú, Implicit Függvényhívás

Az ötlet, ami az implicit hívás mögött áll az, hogy az eljárások közvetlen meghívása helyett, a komponens eseményeket tud kiváltani. A rendszer más komponensei feliratkozhatnak ezekre az eseményekre, azáltal, hogy hozzárendeli az egyik eljárását az eseményhez. Amikor az esemény¸bekövetkezik, a rendszer maga hívja meg az összes eljárást, amit korábban regisztráltak az adott eseményhez. Így, egy esemény publikálása „implicit” módon eredményezi a különböző modulok eljárásainak meghívását.

Architektúrális szempontból, a komponens egy implicit függvényhívások esetében egy modul, aminek a interfésze nemcsak egy eljárások gyűjteményétől (és a hozzájuk tartozó absztrakt adattípusokból) áll, hanem események is szerepelnek. Az eljárásokat a hagyományos módon is meg lehet hívni. Azonban, a komponensek regisztrálhatják néhány eljárásukat a rendszer eseményeire. Ez azt eredményezi, hogy amikor az események bekövetkeznek, a hozzájuk tartozó eljárások meghívásra kerülnek. Ezért, a konnektorok az implicit függvényhívások esetében lehetnek hagyományos függvényhívások, illetve esemény és függvény összerendelések is.

Az egyik fontos haszna az implicit hívások alkalmazásának az, hogy nagyban támogatja az újrafelhasználhatóságot. Bármilyen komponens lehet integrálni a rendszerbe, egyszerűen azáltal, hogy feliratkozik a rendszer eseményeire. A másik fontos előnye, hogy az implicit hívások megkönnyítik a rendszer evolúciót. Bármely komponenst lecserélhetünk egy másikra, ez nem befolyásolja a rendszer más komponenseinek interfészét.

Hátránya az implicit hívásoknak az, hogy a komponensek lemondanak annak vezérléséről, hogy a számítás hogyan hajtódik végre a teljes rendszerben. Egy másik probléma az adatok átadásával kapcsolatban jelentkezik. Bizonyos esetekben az adatot az eseményekkel együtt át lehet adni. Azonban más esetekben a rendszernek egy közös, megosztott tároló helyet kell biztosítani az együttműködéshez. Ebben az esetben a rendszer teljesítménye és az erőforrás kezelés igen komoly kérdéseket vethet fel. Végezetül, a rendszer helyességének bizonyítása nehézségekbe ütközhet, hiszen egy eljárásnak, ami egy eseményt hoz létre a jelentése függ attól, hogy milyen környezetben teszi mindezt.

Rétegelt Rendszerek

A rétegelt rendszer hierarchikusan van kialakítva, minden rége egy adott szolgáltatást nyújt a felette rétegnek és kliensként igénybe veszi az alatta lévő réteg szolgáltatásait. A konnektorokat azok a protokollok határozzák meg, amelyek rögzíti, hogy az gyes rétegek hogyan léphetnek egymással kapcsolatba. Topológiai kényszereket lehet megfogalmazni, például korlátozni a kommunikációt a szomszédos rétegekre.

A rétegelt rendszereknek sok kívánatos tulajdonsága van. Először is, támogatják, hogy a tervezés során különböző absztrakciós szintekkel dolgozzunk. Ez lehetővé teszi az implementálást végzőknek, hogy egy összetett problémát szétbontsanak egymás után következő lépések sorozatára. Másodszor, támogatják a rendszer kiterjeszthetőségét. Akárcsak a csővezetékeknél, egy réteg legfeljebb a felette és az alatta lévővel van kapcsolatban, ezért egy réteg megváltoztatása legfeljebb két másik réteg megváltoztatását vonja maga után. Harmadszor, támogatják az újrafelhasználhatóságot. Akárcsak az absztrakt adattípusoknál, az egyes rétegek különböző implementációit könnyedén kicserélhetjük, amennyiben ugyanazt az interfészt mutatják a szomszédos rétegek felé.

Nem minden rendszert lehet azonban rétegelten struktúrálni, továbbá nehézséget jelent megtalálni a megfelelő absztrakciós szintet is.

Adattárolók

Az adattároló architektúrális típusban két nagyjából eltérő típusú komponens található: egy központi adatstruktúra reprezentálja az aktuális állapotot és több különálló komponens dolgozik ezen a központi tárolón.
A vezérlési elv megválasztása alapján több alkategóriát alakíthatunk ki. Ha az adatátviteli típusok olyanok, hogy a tranzakciók eredményeként bizonyos folyamatokat kiválaszt és elindít (triggerel) a rendszer, akkor az adattár lehet egy hagyományos adatbázis. Amennyiben a központi adattár aktuális állapota alapján dől el, hogy milyen folyamatok indulnak el, akkor az adattároló ún. blackboard rendszer.
Az egyes adatforrások akkor lesznek meghívva, ha a blackboard rendszer állapotának megváltozása triggereli ezeket. A vezérlés valós helye, vagyis implementációja, lehet az adatforrásokban, magában a blackboardban, egy különálló modulban, vagy ezek kombinációiban.

Tábla-vezérelt Értelmezők

Az értelmező motor mind az értelmező definícióját, mind saját futásának aktuális állapotát tartalmazza. Tehát, egy értelmezőnek általában négy komponense van: az értelmező motor, ami elvégzi az értelmezést, memória, ami tartalmazza a pseudo-kódot, amit értelmezni kell, a magának a motornak a vezérlési állapota, és a szimulált program aktuális állapota.
Az értelmezőket általában virtuális gépekben használjak, hogy áthidalják azokat a különbségeket, amik a hardveres számítási működés illetve a program által elvárt számítási működés között fennállnak.

További ismert architektúrák

- Elosztott folyamatok: Az elosztott rendszerek alkalmazása számos ismert szervezési módszert alakított ki több-processzes rendszerek építéséhez. Néhány megoldás topológiai kérdésekre ad választ (pl gyűrű vagy csillag szervezés), míg mások inkább a folyamatok közti kommunikációs protkokollt írják le (pl heartbeat algoritmusok). Ismert tagja: kliens-szerver.
- Főprogram-subrutin felépítés: A modularizálhatóságot nem támogató nyelveken írt rendszerek gyakran egy főprogramból és az aköré csoportosuló szubrutinokból állnak. A főprogram vezérli a szubrutinokat, általában valamilyen ciklus található benne, ami valamilyen sorrendben végrehajtja a szubrutinokat.
- Szakterület-specifikus szoftver architektúrák: Azzal, hogy az architektúrát egy területre specializáljuk, lényegesen növelni tudjuk a struktúrák kifejező erejét. Sőt, sok esetben az architektúrát oly mértékben lehet specializálni, hogy csupán az architektúrális leírás alapján a futtatható rendszer (részben) automatizáltan generálható lesz.
- Állapot átmenetes rendszerek: Számos reaktív rendszer egy állapotátmeneti rendszerrel írható le. Az ilyen rendszerek állapotokból és átmenetekből állnak, amelyek az egyik állapotból a másikba viszik a rendszert.
- Folyamat szabályzó rendszerek: Valamilyen valós fizikai folyamat szabályozására építettek. Leegyszerűsítve, az ilyen rednszerek egy visszacsatolásból állnak, amiben a szernzorokból érkező input alapján a folyamat szabályzó előállítja a kimenetet, ami meghatározza a környezet új állapotát.

Minták

[Minták leírása elvileg: :név (becenév), probléma, megoldás, statika, dinamika (szekvencia diagramok TODO), implementáció kérdések, mérleg, hol alkalmazzák? (nem mindegyiknél volt minden)]

Szolgáltatás-hozzáférési és konfigurációs minták

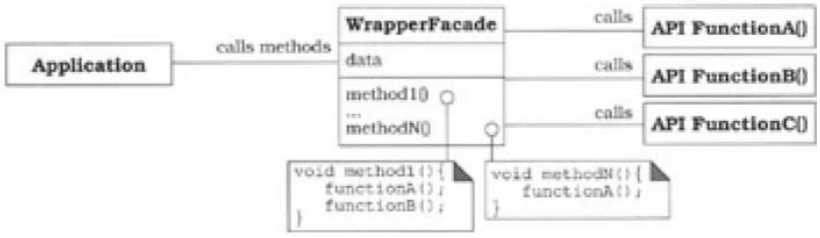
Csomagoló homlokzat (Wrapper Facade)

A Burkoló homlokzat tervezési minta arra való, hogy a nem-OO (objektumorientált) API-k függvényeit és adatait hordozható és karbantartható OO osztály-interfészekbe „burkolja". Olyan karbantartható és továbbfejleszthető alkalmazások esetén célszerű tehát használni, melyek nem-OO API-k hozzáférési mechanizmusait és szolgáltatásait veszik igénybe.

Kerüljük a nem-OO API-k közvetlen meghívását. Inkább az egymással összefüggő függvények és adatok számára hozzunk létre egy vagy több burkoló homlokzat osztályt, melyek metódusaikba foglalják ezeket a függvényeket és adatokat.

Statika: A Burkoló homlokzat mintának két szereplője van:

- A függvények (API function) a nem-OO API-k építőelemei. Ezek önálló szolgáltatásokat valósítanak meg, melyek adatok manipulációját végzik bemenő paraméterek vagy globális változók alapján.
- A burkoló homlokzat (wrapper facade) egy vagy több OO osztály halmaza, melyek magukba foglalják a függvényeket és a hozzájuk kapcsolódó adatokat. Minden osztály egyedi szerepet valósít meg.



Dinamika: Az osztályban elhelyezkedő fgv-ek továbbítják a megfelelő paraméterekkel a hívásokat a nem-OO API fgv-eknek.

Előnyök: Tömör, egybefüggő és robusztus magas-szintű OO interfészek, Hordozhatóság és karbantarthatóság, Modularitás, újrafelhasználhatóság és konfigurálhatóság.

Hátrányok: A funkcionalitás esetleges csorbulása, teljesítmény csökkenése, különböző programozási nyelvek és fordítók nyújtotta korlátok.

Komponens konfiguráló (Component Configurator)

Becenév: edző

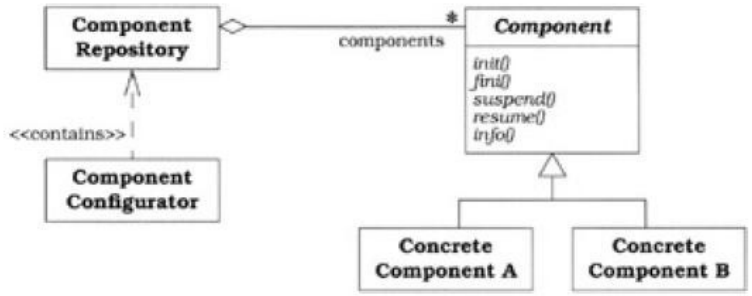
A Komponens konfiguráló tervezési minta arra való, hogy lehetővé tegye az alkalmazás számára komponensek implementációinak futási idejű ki- és becsatolását (link/unlink) az alkalmazás módosítása és újrafordítása nélkül. Ennek ott van haszna, ahol az alkalmazás vagy rendszer komponenseinek minél rugalmasabb és átlátszóbb inicializálására, felfüggesztésére, újraindítására és leállítására van szükség.

Különválasztjuk a komponensek interfészeit az implementációjuktól, az alkalmazást pedig függetlenítjük attól, hogy a komponens-implementációk mikor lettek bekonfigurálva. Tehát a komponens kínáljon egy egységes interfészt az általa nyújtott szolgáltatás vagy funkcionalitás konkrét típusának konfigurációjára és vezérlésére, a konkrét komponensek pedig ezen interfész implementációjaként álljanak elő. Ekkor az alkalmazás vagy annak adminisztrátora futási időben informálódhat az egyes bekonfigurált konkrét komponensekről, illetve dinamikusan inicializálhatja, függesztheti fel, indíthatja újra vagy állíthatja le őket ezen interfészek felhasználásával.

Statika: A komponens konfiguráló mintának a következő objektumok a szereplői:

- A komponens (component) egy egységes interfészt definiál, amely a komponens implementációja által nyújtott szolgáltatás vagy funkcionalitás típusának konfigurációjára és vezérlésére használható.
- A konkrét komponensek (concrete component) a komponens vezérlő interfészét implementálják a komponens adott típusának megvalósításához, vlamint alkalmazásspecifikus funkcionalitást megvalósító metódusokat is implementál.
- A komponenstár (component repository) tarolja a jelenleg alkalmazásba csatlakoztatott konkrét komponenseket.

- A komponens konfiguráló (component configurator) a komponenstár felhasználásával a konkrét komponensek (újra)csatolását vezérli.



Dinamika:

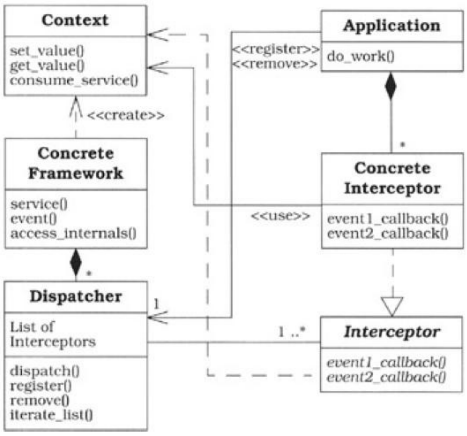
1. Komponens inicializálása a konfigurátor által, a komponens a repo-ba kerül.
2. Komponens végrehajtása, igényelt szolgáltatások biztosítása
3. Komponens leállítása, konfigurátor engedi, hogy az erőforrásokat elengedjék, majd kiveszi a tárból

Előnyök: Egységes komponens hozzáférés. Központi adminisztrálhatóság. Modularitás, tesztelhetőség, újrafelhasználhatóság. Dinamikus komponens becsatlakoztatás és vezérlés. Komponens konfiguráció hangolása, optimalizálása. Bővíthetőség, Robosztusság, Rugalmasság (órán csak ezek voltak)
Hátrányok: Determinisztikus működés és sorrendi függőségek garanciájának hiánya a komponensek futási idejű dinamikus becsatlakoztatása következtében. Megbízhatatlanság és biztonságkritikusság a komponensek futási idejű dinamikus becsatlakoztatása következtében. Futási idejű overhead és bonyolultabb infrastruktúra.
Használják: Eclipse, SCM

Elfogó (Interceptor)

Becenév: elfogó
Az Elfogó tervezési minta arra való, hogy egy keretrendszert átlátszó módon egészíthessünk ki olyan alkalmazásspecifikus szolgáltatásokkal, melyeket bizonyos események bekövetkezése „triggerel". Az alkalmazás oly módon egészítse ki a keretrendszert, hogy a keretrendszerbe előre definiált interfészeken keresztül úgynevezett „külső (out-of-band)" szolgáltatásokat regisztrálunk, melyeket aztán a keretrendszer bizonyos események bekövetkezésekor triggerel.
Statika: Az Elfogó mintának tehát a következők a szereplői:

- A konkrét keretrendszer (concrete framework) egy generikus és kiterjeszthető architektúrát testesít meg.
- Az elfogók (interceptor) a konkrét keretrendszer kapcsán lehetséges valamely eseményhez vagy események egy halmazához tartoznak. Az elfogók olyan „hook" metódusok szignatúráit definiálják, melyeket a konkrét keretrendszer a megfelelő események bekövetkeztekor automatikusan meghív a diszpécseren keresztül.
- A konkrét elfogók (concrete interceptor) specializálják az elfogó interfészeket, és implementálják azok „hook" metódusát, amivel alkalmazás-specifikus módon kezelni tudják az említett eseményeket.
- A diszpécsereket (dispatcher) a konkrét keretrendszer definiálja a konkrét elfogók konfigurálása és triggerelése céljából, hogy ily módon az elfogók képesek legyenek adott események bekövetkezését kezelni. A diszpécser regisztrációs- és eltávolítási-metódusokat definiál. Amikor tehát a konkrét keretrendszer jelzi a diszpécsernek egy-egy ilyen esemény bekövetkeztét, a diszpécser meghívja az összes esemény kapcsán beregisztrált konkrét elfogó megfelelő „callback", avagy „hook" metódusát.
- A kontextusobjektumokat (context object) a konkrét elfogók használják arra, hogy hozzáférjenek a konkrét keretrendszer bizonyos aspektusaihoz, illetve vezérelhessék azt. A kontextusobjektumok „hozzáférési (accessor) metódusai" férnek hozzá a konkrét keretrendszer információihoz, míg a „változtató (mutator) metódusai" arra valók, hogy a konkrét keretrendszer viselkedését vezéreljék.
- Az alkalmazás (application) a konkrét keretrendszer felett fut, felhasználva annak szolgáltatásait.

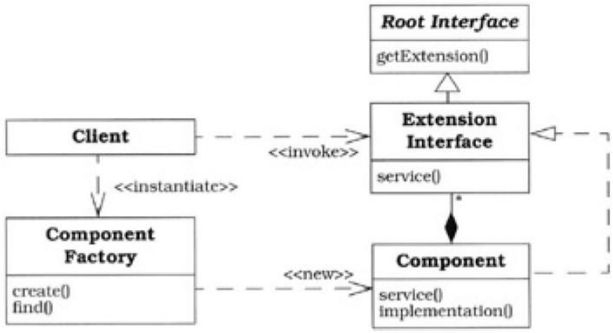


Dinamika: Az alkalmazás példányosít konkrét elfogót, amit a diszpécser beregisztrál. esemény hatására triggerel a keretrendszer, létrehozza az aktuális kontextus objektumot és értesíti a diszpécsert (azaz elküldi neki a kontextust). A diszpécser a beregisztrált konkrét elfogókon végighalad, és meghívja callbackjeiket.
Előnyök:Kiterjeszthetőség és rugalmasság. A szolgáltatások különválasztása az alkalmazás logikájától. Keretrendszerek monitorozásának és vezérlésének támogatása. Rétegek szerinti szimmetria. Az elfogók alkalmazás-szintű újrafelhasználhatósága. Monitorozhatóság.
Hátrányok: Komplex tervezési megfontolások. Hibás elfogók akár az egész alkalmazás működését blokkolhatják. Teljesítmény visszaesés vagy holtpont jöhet létre, ha elfogási lavinák keletkezhetnek. Overhead.
Használják: Alkalmazás szerverekben, Com+, Java EE, CORBA, Eclipse

Kiterjesztő interfész (Extension Interface)

A Kiterjesztő interfész tervezési minta lehetővé teszi, hogy egy komponens több interfészt exportáljon, és ezzel megakadályozza az interfészek felduzzadását. Olyan alkalmazások esetén hasznos, ahol a komponensek interfészei idővel megváltozhatnak. Az alkalmazással szemben támasztott követelmények változása az alkalmazáskomponensek funkcionalitásának megváltozását, bővülését vonhatja magával.
Tehát: „exportáljuk" a komponens funkcionalitását kiterjesztő interfészeken keresztül, melyek mindegyike műveletek egy-egy szemantikusan összefüggő halmazának felel meg. Minden komponensnek legalább egy kiterjesztő interfészt kell megvalósítania. A komponens meglévő funkcionalitásának módosítása vagy kiterjesztése céljából inkább hozzunk létre újabb kiterjesztő interfészeket, mintsem hogy a meglévőket módosítsuk.
Statika: A Kiterjesztő interfész mintának tehát lényegében öt szereplője van:

- A komponensek (component) különböző típusú szolgáltatásspecifikus funkcionalitást valósítanak meg.
- A kiterjesztő interfészek (extension interface) a komponens implementációjának különböző szerepeit exportálják. A komponens által implementált minden egyes szerephez egy-egy kiterjesztő interfész tartozik. A kiterjesztő interfész specifikálja azt a „megállapodást (contract)", ami megszabja, hogy a kliensek miképpen használhatják a komponens funkcionalitását.
- A gyökér interfész (root interface) egy speciális kiterjesztő interfész, mely háromféle funkcionalitást biztosít: (1) Az alapfunkcionalitást, amelyet mindegyik kiterjesztő interfésznek támogatnia kell (2) területfüggetlen funkcionalitást, amelyek például a komponensek életciklusát szabályozzák; (3) területspecifikus funkcionalitást, amelyet az adott terület minden komponensének tudnia kell.
- A kliensek (client) kizárólag kiterjesztő interfészeken keresztül érik el a komponensek nyújtotta funkciókat. Miután a kliens hozzájutott a gyökér interfész referenciájához, képes egy adott komponens bármely kiterjesztő interfészének lekérésére.
- Adott komponenstípushoz tartozó komponensgyár (component factory) szolgál arra, hogy a kliens számára kiadja a gyökér interfész referenciáját.



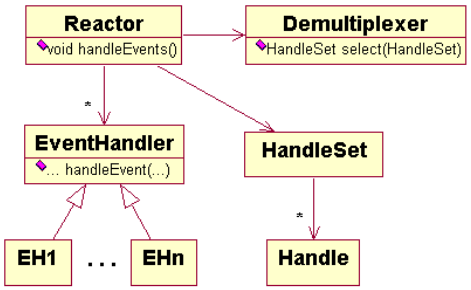
Előnyök: Kiterjeszthetőség. Komponensek szerepeinek különválasztása. Polimorfizmus közös interfészből való leszármazás nélkül. Komponensek és kliensek különválasztása az implementáció és interfész különválasztásával. Interfész aggregáció és delegáció támogatása.
Hátrányok: Megnövekedett komponens tervezési és implementációs igény. A kliens programozása bonyolultabb. Újabb indirekció és futási idejű overhead.
Használják: COM, COM+, CORBA, Java EE,

Eseménykezelési minták

Reactor

"Zh esélyes." -Hassan
Becenév: Dispatcher, Notifier, Telefonközpont
Feladata a szinkron eseménykezelés. Lehetővé teszi az esemény-vezérelt alkalmazások számára olyan szolgáltatáskérések szétosztását és irányítását, amelyek egy vagy több klienstől érkeznek be. A beérkező jelző eseményeket (indication event) szinkron módon kell kezelni, majd szét kell választani a szétosztást és az irányítási eljárásokat a szolgáltatáson belüli alkalmazás-specifikus feldolgozástól. Fel kell készülni, hogy egy szolgáltatáshoz egyidejűleg több kérés is befut. Ekkor az a megoldás, hogy megfelelő helyre szétosztják a kéréseket, majd sorosan feldolgozzák.
Statika: A Szolgáltatáskérések szétosztásának mintája öt főbb részt tartalmaz:

- Kezelők, amelyeket az operációs rendszer biztosít olyan eseményforrások azonosítására (pl. hálózati kapcsolatok vagy megnyitott fájlok), amelyek jelző eseményeket generálhatnak.
- Szinkron esemény szétválasztó, amely egy vagy több jelző esemény bekövetkezésére vár a kezelők egy halmazán.
- Az eseménykezelő egy interfészt specifikál egy vagy több kampó eljárás (hook method) számára. Ezek az eljárások egy olyan halmazát reprezentálják, amelyek az alkalmazás specifikus események feldolgozására szolgálnak.
- A konkrét eseménykezelők specializálják az eseménykezelőt, és az alkalmazás által nyújtott szolgáltatás specifikus szolgáltatását implementálják.
- A reaktor egy interfészt definiált, amely lehetővé teszi az alkalmazásoknak, hogy regisztrálják vagy eltávolítsák a hozzájuk rendelt kezelőik eseménykezelőit, valamint futtassák az alkalmazás eseményhurokját (event loop).

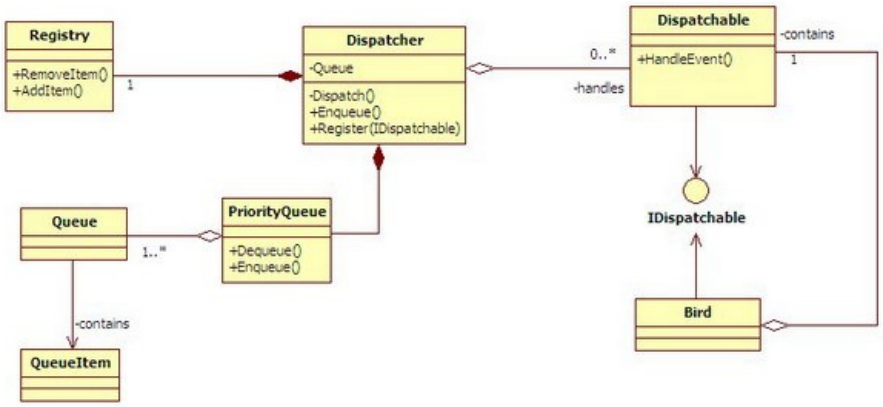


Dinamika: Alkalmazás regisztrál reaktornál és jelzi, milyen eseményekről kér értesítést. Reaktor utasítja az eseménykezelőt, hogy adja meg a kezelőjét, majd ezeket a reaktor kezelőhalmazba gyűjti, és meghívja a szétosztót, ami várakozik. Ha egy kezelő „KÉSZ" lesz, szétosztó értesíti reaktor, ami meghívja az eseménykezelők callback-jeit.
Előnyök: Modularitás, újra felhasználhatóság és széles körű beállíthatóság. Hordozhatóság. Durvaszemcsés konkurenciakezelés.Skálázhatóság
Hátrányok: Korlátozott felhasználhatóság (szinkron működés). Nem megszakításos működés. A hibakeresés és tesztelés bonyolult

Proactor

Becenév: posta, üzenetrögzítő
Az Aszinkron műveletek feldolgozása architekturális mintázat lehetővé teszi az esemény-vezérelt alkalmazásoknak, hogy hatékonyan szétosszák és irányítsák a kért szolgáltatásokat, amelyeket aszinkron műveletek végrehajtása triggerel. Így teljesítménybeli előnyöket érhetünk el a konkurencia által anélkül, hogy bizonyos hátrányokat is elszenvednénk. Az Aszinkron műveletek feldolgozása minta olyan helyzetekben alkalmazható jól, amikor az esemény-vezérelt alkalmazás aszinkron módon fogadja és dolgozza fel a több helyről beérkező szolgáltatáskéréseket.
Az alkalmazás szolgáltatásait kétfelé kell osztanunk: sokáig tartó műveletek, amelyek aszinkron módon futnak, és végrehajtási kezelők, amelyek feldolgozzák az előbbiek eredményeit, miután azok befejeződtek. Integrálni kell azon végrehajtási események szétosztását, amelyek az aszinkron műveletek befejeződésekor továbbítódnak, és a végrehajtási kezelőkhöz történő irányításukat is, amelyek végül feldolgozzák őket. Ezt a végrehajtási esemény szétosztást és irányítást külön kell választani a végrehajtási kezelők végrehajtási események alkalmazás specifikus feldolgozásától.
Statika: Az Aszinkron műveletek feldolgozása minta kilenc részből áll:

- Kezelők, amelyeket az operációs rendszer szolgáltat olyan elemek azonosítására, amelyek végrehajtási eseményeket generálnak.
- Az aszinkron műveletek általában sokáig tartó műveleteket jelentenek, amelyek a szolgáltatások implementációiban szerepelnek.
- A végrehajtási kezelők interfészeket specifikálnak, amelyek egy vagy több kampó eljárással (hook method) rendelkeznek. Ezek az eljárások olyan műveletek halmazát jelentik, amelyek az alkalmazás specifikus végrehajtási eseményekben található visszaadott információt dolgozzák fel.
- A konkrét végrehajtási kezelők specializálják a végrehajtási kezelőt, hogy egy az alkalmazás által nyújtott adott szolgáltatást definiáljanak, amely az örökített kampó eljárás(oka)t implementálja. A kampó eljárások feldolgozzák a végrehajtás eseményben található eredményeket.
- Az aszinkron műveleteket az aszinkron műveleti feldolgozó hívja meg, amely gyakran már eleve implementálva van az operációs rendszer kerneljében.
- Az aszinkron esemény szétosztó olyan függvény, amely a végrehajtási eseménysorba helyezett végrehajtási eseményekre várakozik, amelyeket aztán eltávolít a sorból és visszatér a hívójához.
- A Proaktor eseményhurkot szolgáltat egy alkalmazás folyamatnak vagy szálnak. Ebben a proaktor meghívja az aszinkron művelet szétosztót, amely a bekövetkező végrehajtási eseményekre vár. Ezek bekövetkezésekor a szétosztó visszatér, majd a proaktor demultiplexálja az eseményt az eseményhez rendelt végrehajtási kezelőhöz, és elindítja a megfelelő kampó eljárást a kezelőn a feldolgozáshoz.
- A kezdeményező egy, az aszinkron műveletet meghívó alkalmazáshoz képest lokális entitás.



Előnyök: Az egyes problémakörök különválasztása, Hordozhatóság, A konkurensen működő mechanizmusok beágyazása, A száltechnika és a konkurencia szétválasztása, Teljesítmény, Szinkronizáció egyszerűsödése

Hátrányok: Korlátozott alkalmazhatóság, A programozás bonyolultsága, Az aszinkron módon futó műveletek ütemezése, vezérlése és megszakítása

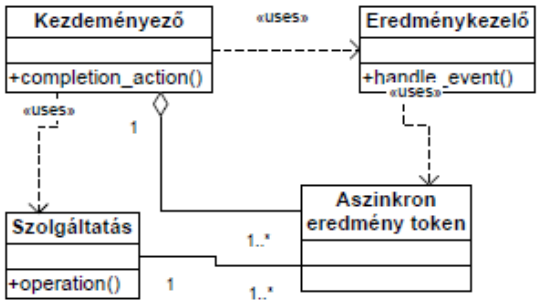
Asynchronous Completion Token (ACT)

Az előző, proactor mintát hatékonyabbá tehetjük az eredmény elosztó algoritmus átalakításával. A feladat csak annyi, hogy az aszinkron művelethez hozzákötjük az eredménykezelő elérhetőségét és így nem szükséges azt visszakeresni.

Minden aszinkron művelethez a létrehozás során létrehozunk egy aszinkron eredmény tokent (ACT), amely tartalmazza azokat az adatokat, amely az eredménykezelőt egyértelműen azonosítja. Ez a token a művelet végrehajtása során változatlanul megmarad és a végén az eredmény-eseménnyel együtt visszakerül a kezdeményezőhöz. Így a kezdeményező könnyen meghívhatja az eredménykezelőt.

Statika: A struktúra elemei:

- A szolgáltatás olyan műveleteket takar, amelyek aszinkron meghívhatóak.
- A kezdeményező hívja meg az aszinkron szolgáltatásokat. Továbbá a válaszokat eljuttatja az eredménykezelőkhöz, amely feldolgozza a szolgáltatás eredményeit.
- Az aszinkron eredmény token olyan információkat tartalmaz, amelyek egyértelműen azonosítanak egy eredménykezelőt a kezdeményező számára. A kezdeményező állítja össze és átadja a szolgáltatásnak, amikor meghívja. A szolgáltatás változatlanul csatolja az eredmény-eseményhez. Ezt követően a kezdeményező már könnyen azonosíthatja az eredménykezelőt, amelynek továbbadja az eredmény-eseményt feldolgozásra.
- Az eredménykezelő funkcionalitása megegyezik a proactor mintában szereplővel. (meghívja a callback-eket)



Dinamika: Kezdeményező tokent generál, majd meghívja a szolgáltatást, átadva a tokent. További műveleteket is indíthat ugyanígy. Aszinkron művelet végén visszaadja az eredményt és az ACT-t, ami alapján azonosítja az eseménykezelőt. Átadja neki az adatokat, amin az elvégzi a spec. műveletet.

Előnyök: A minta az eredmény-események szétosztását segíti a korábbi proactor mintához képest. Így a legnagyobb előnye ott mutatkozik meg, hogy nincs szükség bonyolult táblázatokban keresgélni az eredménykezelőket, hanem egyszerűen továbbítható az esemény. A további előnyei megegyeznek a proactor-éval.

Hátrányok: A fejlesztők az ACT felszabadítása során hibákat követhetnek el ezzel memóriaszivárgást okozva.

Használják: CORBA

Acceptor-Connector

Becenév: Manager Secretary

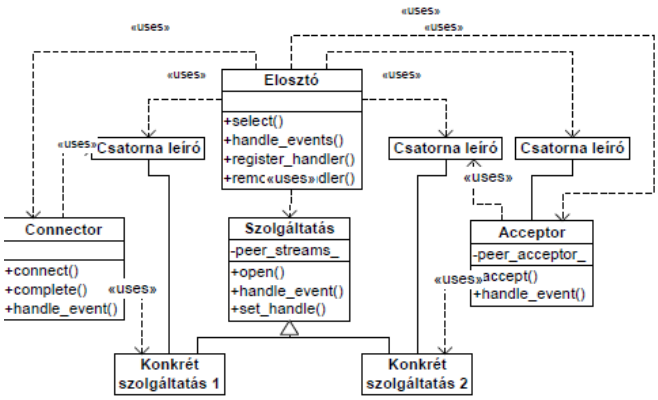
A felek kapcsolódásának implementációját célszerű leválasztani a kommunikációt kezelő konkrét részekről, mivel az az adatátvitel szempontjából irreleváns. Az alkalmazásnak inkább a kommunikációra kellene koncentrálnia és nem a kapcsolat felépítésre. Ha egy hálózati komponens viselkedhet kliensként és szerverként is, vagy egyszerre mindkettőként, akkor szükségünk vagy egy megoldásra, amely alkalmazkodik a megváltozott szerepekhez és kezeli a kapcsolat felépítést.

Ehhez külön választjuk a kapcsolat felépítéséért felelős részeket és a kommunikációs protokollimplementáló részeket. Mivel a kapcsolat felépítésénél mindig van egy aktív és egy passzív fél, ezért mindkettőt le kell implementálnunk, míg a tényleges kommunikációt folytató rész mindkét esetben lehet ugyanaz.

Statika: A struktúra elemei:

- A szolgáltatás kezelő a hálózati kommunikáció két résztvevőjéből az egyik oldalt implementálja. Egy konkrét szolgáltatás implementáció lehet kliens vagy szerver, vagy mindkettő peer-to-peer kommunikáció esetén. A szolgáltatás kezelő tartalmazza az alkalmazás szintű protokoll megvalósítását.

- Az acceptor valósítja meg a passzív kapcsolat felépítést végző komponenst. Az inicializációja során létrehoz egy passzív végpontot, amely várja a kapcsolódásokat.
- A connector komponens valósítja meg az aktív kapcsolat felépítést. A kapcsolat kezdeményezésekor jelzi milyen szolgáltatást szeretne használni. A túloldalon egy acceptor-nak kell várnia, amellyel létrehozzák a kommunikációs csatornát.
- Az elosztó feladata, hogy a hálózati kommunikáció eseményeit fogadja és a megfelelő modulokhoz továbbítsa.



Dinamika:

- Aktív mód esetén connector felépíti a csatornát, majd meghívódik a szolgáltatás, innentől ő kommunikál.
- Passzív mód esetén acceptor vár, majd bejövőnél felépíti a kapcsolatot, és meghívja

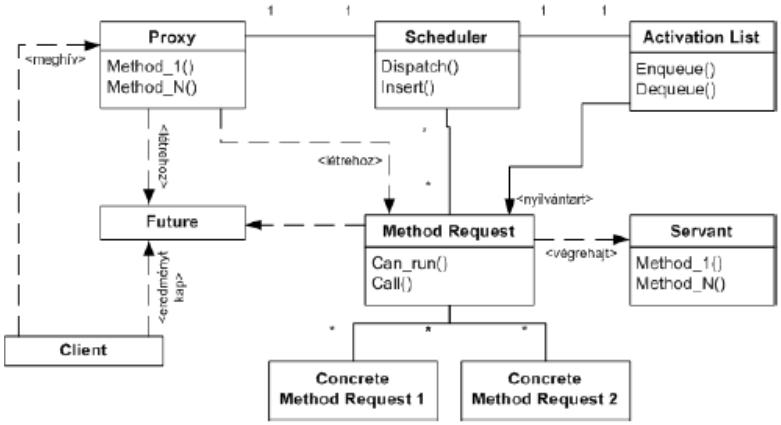
Előnyök: Elválasztja az alkalmazás független kapcsolat felépítést , így az alkalmazás független részek újra felhasználhatóak más programoknál. Lehetővé teszi, hogy a kapcsolatok felépítése aszinkron módon történjen, így egyszerre nagy mennyiségű hoszttal tudunk felépíteni kapcsolatot. Hatékony, robosztus, könnyű implementán, hordozhatói.
Hátrányok: Egy egyszerű kliens esetén, amelynek a feladata az, hogy egy szolgáltatáshoz kapcsolódjon, az acceptor-connector minta implementációja nagyobb lehet, mint a teljes eredeti program. Overhead
Használják: Machine to machine rendszerek, CORBA.

Konkurenciakezelési minták

Active Object

Hassan kedvence.
Becenév: Pincér az étteremben
Az Active Object tervezési minta lényege, hogy elválasztja az eljáráshívást az eljárás végrehajtásától. Tekintsünk egy többszálú alkalmazást, ahol termelő és fogyasztó folyamatok kommunikálnak egy központi objektumon keresztül. Meg kell oldani, hogy se a termelő, se a fogyasztó folyamatok ne blokkolják egymás kommunikációját.
A problémát úgy oldjuk meg, hogy a metódus meghívását és végrehajtását külön processzekre bízzuk.
Statika: Az Active Object tervezési minta 6 komponenst definiál.

- A kliens processzében futó proxy objektum fogadja a kliens kéréseit (metódushívásait)
- Amikor a kliens meghívja a proxy egyik metódusát, a proxy létrehoz egy method request objektumot, ami a kérés/hívás tulajdonságait és kontextusát hivatott tárolni. A proxy összes szinkronizálandó metódusához implementálni kell a method request alaposztály egy-egy konkrét alosztályát, melyek közül értelemszerűen a meghívott metódusnak megfelelő típusú objektumot hozza létre a proxy.
- A létrehozott method request objektum bekerül egy aktivációs listába (activation list). Ez a lista a folyamatban lévő kliens kéréseket tartja nyílván. Mivel ez a lista nyilvánvalóan megosztott használatú, elérését szinkronizálni kell.
- Az aktív objektum száljában futó ütemező (scheduler) feladata, hogy eldöntse, hogy az aktivációs lista mely elemét hajtsa végre legközelebb.
- A kiszolgáló (servant) végrehajtja az ütemező által kiválasztott kérést.
- Amikor a kliens egy hívást kezdeményez a proxyban, kap egy válasz kezelő objektumot (future), amelyen keresztül a kliens megkaphatja a kérés eredményét.

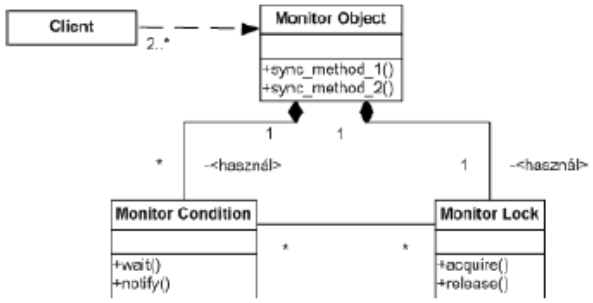


Dinamika: Concrete Method Req-et létrehoz a proxy (amit kliens hívott), az tárolja az adatokat és a kontextust. Továbbadja schedulernek, ami listába teszi, klien kap egy Future-t, amibe majd az eredmény lesz lekérezhető.
Scheduler másik szálon fut, kiválasztja active list-ből a futtatni kívántat, majd meghívja a Servant metódusát. eredményt a Scheduler a Future-ba tárolja, amit elér a kliens.
Előnyök: Egyszerűsíti a konkurencia kezelését, transzparens módon oldja meg a párhuzamosítást, valamint külön kezelhetővé teszi az eljáráshívást és -végrehajtást. Egyszerű, rugalmas, gyors, hatékony és jól bevált.
Hátrányok: A Scheduler implementációjától függő mértékű overhead és nehezebb debuggolhatóság. Proxy: SPF!

Monitor Object

Becenév: gyors étterem
A Monitor Object tervezési minta biztosítja, hogy egy objektumnak egy időben csak egy metódusa futhasson. Erre olyan esetekben lehet szükség, amikor egy többszálú alkalmazásban egy megosztott objektumot többen szeretnének elérni. Ekkor biztosítani kell, hogy a megosztott objektum állapota minden esetben konzisztens maradjon, és ne befolyásolhassa azt a többszálúság miatt kialakuló versenyhelyzet.
A probléma megoldásának lényege, hogy a megosztott objektum metódusait egyszerre csak egy kliens szál futtathatja.
Statika: A Monitor Object tervezési minta 4 komponenst definiál:

- A monitor objektum egy vagy több interfész metódust exportál. Minden kliens ezen metódusokon keresztül érheti csak el a monitor objektumot.
- A szinkronizált metódusok egy-egy exportált metódust implementálnak thread-safe módon. Egyszerre csak egy ilyen metódus futhat egy monitor objektumon belül.
- Minden monitor objektum tartalmaz egy monitor lockot. A szinkronizált metódusok ezt használják a metódus-futások sorosítására. Minden szinkronizált metódusnak meg kell szereznie a lockot futás előtt és el kell engednie azt a futás befejeződéskor.
- A monitor feltételek egy monitor objektumon belül azt határozzák meg, hogy egy metódus mely esetekben függessze fel a futását (engedjen előre másokat) és mely esetekben folytassa működését. Ezzel lehetővé tesszük, hogy a metódusok végrehajtása ne feltétlenül a kliensek „érkezési” sorrendjében történjen.



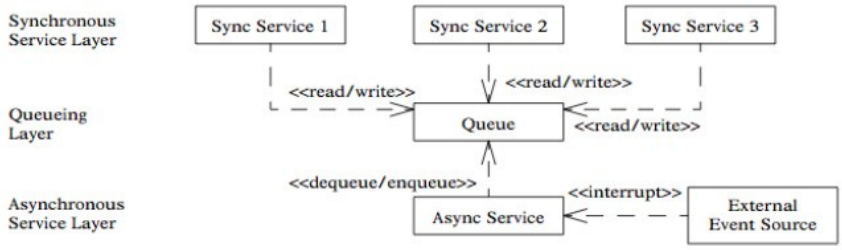
Dinamika: T1 szál meghívja Monitor Object egy metódusát, ami megpróbál lock-ot szerezni. Ha nem tud, blokkolódik addig. Blokkolódás esetén várhat MO feltétel teljesülésre, ezzel átengedve másnak a lehetőséget (ideiglenes kilépés). Ha a várt feltétel teljesül, adott szálakat felébreszti, és aki lock-ot szerez, az futtathatja a metódust.

Előnyök: Egyszerűbbé teszi az implementációt. A monitor feltételek bevezetése leegyszerűsíti és átláthatóvá teszi a thread-ek ütemezését, elkerülve azt, hogy a felfügesztett thread-eknek pollingolni kelljen a monitor objektumot amíg a feltétel nem teljesül. Nincs overhead.
Hátrányok: Korlátozhatja az alkalmazás skálázhatóságát. Növeli az objektum kódjának bonyolultságát. Problémákba ütközünk akkor is, ha örökölni akarunk egy monitor objektumból.

Half-Sync/Half-Async

Becenév: foglalt étterem
A Half-Sync/Half-Async architektúrális minta kettéválasztja konkurrens rendszerekben a szinkron és aszinkron szolgáltatásfeldolgozást, hogy egyszerűsítse a programozást a teljesítmény indokolatlan csökkentése nélkül. A minta bevezet két egymással kommunikáló réteget, egyet az aszinkron és egyet a szinkron szolgáltatás feldolgozáshoz.
Lényege, hogy hardverfejlesztők aszinkron műveleteket szeretnek, szoftveresek meg szinkronokat. Ezzel a mintával megoldható, hogy mindenki úgy programozzon, ahogy szeretne. Erre a megoldás, hogy szétbontjuk a rendszer szolgáltatásait két rétegbe, szinkron és aszinkron rétegekbe, és hozzáadunk egy sorbanállás réteget közéjük, hogy közvetítse a kommunikációt a szinkron és aszinkron régetek szolgáltatásai között.
Statika: A Half-Sync/Half-Async minta követi a Rétegek mintát, és négy résztvevőt tartalmaz:

- A szinkron szolgáltatás réteg magas szinten feldolgozó szolgáltatásokat futtat. A szinkron réteg szolgáltatásai különb szálakon vagy folyamatokban futnak, melyek blokkolhatnak míg a műveleteiket végzik.
- Az aszinkron szolgáltatás réteg az alacsony szintű feldolgozó szolgáltatásokat végzi, melyek tipikusan egy vagy több külső eseményforrásból erednek. Az aszinkron réteg szolgáltatásai nem blokkolhatnak míg műveleteiket végzik anélkül, hogy indokolatlanul rontanák a többi szolgáltatás teljesítményét.
- A sorbanállási réteg biztosítja a szinkron és aszinkron rétegek közti kommunikációs mechanizmust.
- A külső eseményforrások hoznak létre eseményeket, melyeket az aszinkron szolgáltatási réteg kap meg és dolgozik fel.



Dinamika:

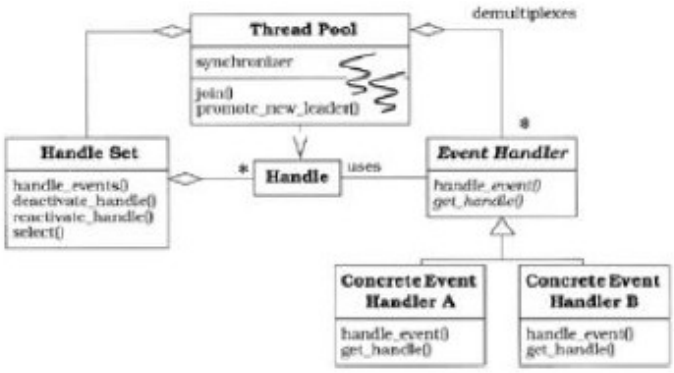
- Aszinkron fázisban a külső forrás az aszinkron réteggel lép kapcsolatba, pl. interrupt. Ha aszinkron befejezte a feldolgozást, sorbanállásin keresztül kommunikálhat a szinkronnal.
- Sorbanállási fázisban a réteg buffereli a kéréseket.
- Szinkron fázisban a szinkron műveletek begyűjtik az adatokat és feldolgozzák, majd „visszfelé mindez”.

Előnyök: Egyszerűsítés és teljesítmény. Problémák széválasztása. Rétegbeli kommunikáció központosítása.
Hátrányok: A határ átlépésének hátrányát vonhatja maga után. A magasszintű alkalmazási szolgáltatások nem feltétlenül nyernek az aszinkron I/O hatékonyságából. Hibakeresés és tesztelés bonyolultsága.
Használják: Unix Network, ORB, EJB

Leader/Followers

A Leader/Follower architekturális minta egy hatékony konkurenciamodellt biztosít, ahol több szál felváltva megoszt egy esemény forrás halmazt, hogy felderítsen, demultiplexáljon, elküldjön és feldolgozzon szolgáltatás kéréseket, melyek az eseményforrásokon történnek. A többszálúság egy átlagos technika olyan alkalmazások implementálására, melyek több eseményt konkurensen dolgoznak fel. Azonban elég nehéz nagyteljesítményű többszálú alkalmazásokat implementálni. Ezek az alkalmazások gyakran különböző típusú események nagy mennyiségét dolgozzák fel, mint a connect, read és write események, melyek egyidejűleg érkeznek.
A probléma megoldása: Vegyünk egy szálkészletet melyeket megosztunk eseményforrások egy halmazán hatékonyan úgy, hogy váltva demultiplexálják az eseményeket, melyek ezen eseményforrásokról érkeznek és szinkron továbbküldik az eseményeket az alkalmazásszolgáltatásoknak, melyek feldolgozzák őket.
Statika: Négy kulcsszereplője van a Leader/Followers mintának:

- A kezelőket az operációs rendszer biztosítja hogy azonosítsák az eseményforrásokat, mint a hálózati kapcsolatok vagy nyitott fájlok, melyek eseményeket generálnak és sorba tesznek.
- A kezelőhalmaz egy olyan gyűjteménye a kezelőknek, melyek használhatók arra, hogy várakoznak egy vagy több esemény bekövetkezésére egy halmazbeli kezelőre. A kezelőhalmaz visszatér a hívójához amikor lehetséges egy műveletet kezdeni a halmazbeli kezelőn anélkül, hogy az a művelet blokkolna.
- Egy eseménykezelő ad meg egy interfészt egy vagy több horogmetódusból áll (hook method). Ezek a metódusok reprezentálják az elérhető művelethalmazt, melyek alkalmazásspecifikus eseményeket dolgoznak fel, mely a eseménykezelők által kiszolgált kezelőkön történik.
- A konkrét eseménykezelők specializálják az eseménykezelőket, hogy specifikus szolgáltatásokat implementáljanak, melyeket az alkalmazás kínál.



A Leader/Followers minta szívében van a szálkészlet, mely megosztja a szinkronizálót, mint például egy szemafor vagy állapotváltozó, és implementál egy protokollt a különböző szerepek közti váltás koordinálására. Egy vagy több szál játssza a követő szerepet és felsorakoznak a szálkészlet szinkronizálónál várva, hogy a vezető szerepbe kerüljenek. Amikor egy esemény megtörténik, az aktuális vezető szál előlépteti az egyik követőszálat, hogy az új vezető legyen. Az eredeti vezető aztán konkurensen játssza a feldolgozó szál szerepét, mely demultiplexálja az eseményeket a kezelőhalmazról egy megfelelő eseménykezelőbe majd elküldi a kezelő horogmetódusát hogy kezelje az eseményt. Miután egy feldolgozó szál befejezi az esemény kezelését, visszatér a követő szerep játszásába és várakozik, hogy a szálkészlet szinkronizálójánál a sorára, hogy vezetőszál legyen újra.
Dinamika: Vezető szál várakozik a kezelőhalmazban lévő egyik kezelő eseményének bekövetkezésére. Ha észlelte, előléptet egy követő szálat vezetővé. eseménykezelő demultiplexálja az eseménykezelést, a vezető szál (konkurensen) játssza a feldolgozó szerepét, és elküldi a horogmetódust, hogy feldolgozza az eseményt. Ennek befejeztével követőként csatlakozik a thread pool-hoz.
Előnyök: Teljesítménynövekedés. Programozási egyszerűség
Hátrányok: Implementációs bonyolultság. Rugalmasság hiánya. Hálózati I/O szűk keresztmetszetek.
Használják: webszerver, copy-paste

Szinkronizációs tervezési minták

Scoped Locking

A védett kódrészeket, melyek konkurens lefutását megakadályozni szeretnénk, lockolással kell védenünk, melyeket gondosan fel kell szabadítanunk, ha a lefutás elhagyta a kritikus szakaszt. Azonban nem feltétlenül egyszerű a szakaszból kilépést nyomon követni, ugyanis a metódusunk több pontról is visszatérhet. Ezen kívül előfordulhat nem várt kivételdobás, ami a kritikus hatókörön kívülre vezet.
Megoldás: Definiáljunk egy ún. Guard osztályt, aminek a konstruktora automatikusan végrehajtja a zárolást, amikor a lefutás a hatókörbe ér, és aminek a destruktora felszabadítja azt, amikor elhagyjuk a kritikus hatókört.
A Scoped Locking módszer megnövekedett robusztusságot garantál. Az idióma alkalmazásával a lockolás biztosítása és feloldása automatikusan történik, amikor a program lefutása során belép illetve elhagyja a kritikus szakaszt. Az idióma a konkurens alkalmazások robusztusságát növeli azzal, hogy megszűnteti a szinkronizációból és a többszálúságból fakadó hibalehetőségeket. Két hiányossága a módszernek a rekurzív hívások során fellépő deadlock illetve a nyelvspecifikus szemantikából adódó korlátozások, minden típushoz külön osztály kell.

Strategized Locking

A Strategized Locking tervezési minta parametrizálja a szinkronizációs mechanizmusokat, amik a komponens kritikus szakaszait védik a konkurens eléréstől.
Azoknak a komponenseknek, amik többszálú környezetben futnak, a kritikus szakaszaikat védeniük kell a konkurens klienshozzáféréstől. Különböző típusú alkalmazások különböző szinkronizációs stratégiákat igényelnek (olvasási/írási zárolások, szemaforok stb.). Emiatt szükség van arra, hogy a komponensek szinkronizációs mechanizmusait testre szabhatóvá tegyük az adott alkalmazás szükségletei szerint.
Megoldás: Parametrizáljuk egy komponens szinkronizációs aspektusait oly módon, hogy „csatlakoztathatóvá" tesszük. Minden típus egy adott szinkronizációs stratégiát ír le (pl. mutex, reader/writer, lock, szemafor, null lock). Ehhez kell egy ősosztály, amiből a különböző implementációkat származtatjuk. Példányosítsuk ezeket a csatlakoztatható típusokat a komponensen belül, ami használni tudja az objektumokat a metódusainak hatékony szinkronizálására.
Előnyök: Konkurenciamodell beállítása és testre szabása egy komponenshez. Zárolási módszer egy új konkurens modellhez, a lockolási stratégiák családja bővíthető a létező kód módosítása nélkül. Újrafelhasználhatóság.
Hátrányok: az obtructive locking és az ún. over-enginerring.

Thread-Safe Interface

A Thread-Safe Interface minta minimalizálja a zárolási overheadet és biztosítja, hogy a komponensen belüli metódusok ne kerüljenek „self-deadlock"-ba azzal, hogy újra le akarják foglalni a már foglalt zárat. A többszálú komponensek gyakran tartalmaznak több publikusan elérhető interfészmetódust és privát metódusokat, amik megváltoztathatják a komponensek állapotát. A versenyhelyzet megelőzése érdekében a komponensbe ágyazott zárolással szerializálhatjuk a metódushívásokat. Ez a megoldás abban az esetben működik, ha a metódusok nem hívják egymást.
Megoldás: Minden olyan komponenst, ami komponensen belüli metódushívásokat tartalmaz két elv alapján tervezzünk. Minden interfészmetódus (pl. C++ publikus metódusok) csak komponens zárakat foglal le/szabadít fel. Miután a zárolás megtörtént az interfészmetódus az implementált metódushoz továbbítódik, ami ellátja az aktuális metódus funkcionalitását. Az implementációs metódus visszatérése után az interfészmetódusnak fel kell szabadítani a zárakat, mielőtt visszatérne a vezérlés a meghívotthoz. Az implementációs metódusok (pl. C++ privát és protected) csak akkor futnak le, ha az interfész metódusok hívták meg azokat. Így biztosíthatjuk, hogy a szükséges zárolásokkal hívtuk meg a metódusokat, és soha nem foglalnak és engednek el zárakat. Az implementációs metódusok továbbá soha nem hívhatnak meg interfészmetódusokat, mivel ezek foglalják le a zárakat.
Előnyök: Nincs selfdeadlock. Zárakat nem foglalunk le és szabadítunk fel szükségtelenül. Lock és funkcionalitás elkülönül.
Hátrányok: Megnövekedett kód és metódushívás. Szinkronizációs overhead. Csökkenő teljesítmény az alacsony szintű zárak miatt.

Double-Checked Locking Optimization

A Double-Checked Locking Optimization design pattern a szinkronizációs overheadet csökkenti abban az esetben, ha a kód kritikus területei zárak használatát teszi szükségessé threadsafe módon egyszeri alkalommal a program futása során.
A mutex szerializációs módszer kevésnek bizonyulhat olyan objektumok vagy komponensek számára, amik egyszeri inicializálást igényelnek. A Singleton esetben a kód kritikus szekcióját csak egyszer kell futtatni az inicializálás során; annak ellenére, hogy minden metódushívás lefoglalja és felengedi a zárat, ami overheadet okoz. Az overhead elkerülése érdekében a konkurens alkalmazások fejlesztői globális változókat használnak a Singleton minta alkalmazása helyett. Ennek a megoldásnak két hátránya is van. Nem hordozható, mivel a különböző fájlokban definiált globális objektumok definíciós sorrendje gyakran nem specifikált. Ezen kívül erőforrás-igényes megoldás, mivel a globális változóink akkor is létrejönnek, ha nem is használjuk azokat.
Megoldás: Vezessünk be egy flaget annak jelzésére, hogy a kritikus szekció lefuttatása szükséges-e. Ha nem kell lefuttatni, akkor a kritikus szakaszt a futás során nem érintjük, így elkerülhetjük a szükségtelen zárolásból adódó overheadet.
Előnyök: Zárolási overhead minimalizálása. Versenyhelyzet megoldása.
Hátrányok: Illegális memóriahozzáférés keletkezhet. Bizonyos rendszerekben nem implementálható

Peer-to-peer rendszerek

Definíció: Olyan rendszer, mely a résztvevők erőforrásait (sávszélesség, háttértár, processzoridő) használja fel a szolgáltatások nyújtására és nem valamilyen központi komponensét (szerverét). Általában akkor használjuk, ha minden résztvevő egyben kliens és szerver is.
Felhasználási területek: Fájl-/adatcserélés (BitTorrent, Gnutella), Kommunikáció (Skype), Audió/videó streaming, Elosztott számítási rendszerek (setiHome,grid.com), Elosztott keresés (YaCy)

- Tisztán (pure) P2P: minden csomópont egyenrangú, teljesen azonos funkciót lát el. Kevésbé érzékeny a „meghibásodásokra"
- Hibrid P2P: a rendszer valamilyen funkcióját központosított komponensek valósítják meg
    - Tipikusan központi nyilvántartás a peer-ekről
    - Centralizált, többszintű („semi-centralized") hálózatok

- Nem strukturált: Tetszőleges két peer között lehet kapcsolat
    - Kommunikáció elárasztással
    - Pl. Gnutella
- Strukturált: A peer-ek valamilyen struktúra szerint kapcsolódnak egymáshoz, így téve hatékonyabbá a kommunikációt
    - Pl. Elosztott hash táblák (DHT)

Szemantikus réteg: P2P tipikus probléma: hova csatlakozzunk?
Megoldás: szemantikus réteg a tiszta P2P hálózat fölé

- A peer-ekről lehet tudni, milyen „jellegű" adatot tárolnak, szoktak letölteni, stb. (szemantikus adat)
- Amikor egy peer le akar tölteni valamit, olyan peer-ek címét kapja először vissza amely „jellegében" leginkább hasonlókat szokott letölteni
- Pluszmunka (overhead) a szemantikus réteg karbantartása miatt, de sok esetben megéri

BitTorrent

Fájlcserélő protokoll, nagy mennyiségű statikus adat megosztására.

- Skálázható: a rendelkezésre álló sebesség növekszik minden újonnan csatlakozó peer-el
- Hatékony: nagy adatmennyiség gyors mozgatása sok résztvevő között
- Robosztus: peer-ek kiesése esetén is működik tovább

Az álományok darabokra vannak osztva, egy darab letöltése után egyből visszaosztja azt. SHA1 hash ellenőrzés. A tracker feladata a peer-ek (peer: leecher vagy seeder) összehozása (IP:port). A torrent file egy kisméretű, metaadatokat és infohasht tartalmazó file, ez alapján lehet hozzáférni a megosztott tartalomhoz, és ez alapján történik az azonosítás. Ha a tracker kiesik, leáll a szolgáltatás. Megoldások:

- Multi-tracker: egyszerű megoldás, jobb rendelkezésre állás, terheléseloszlás
- Peer exchange (PEX): peer-ek megosztják az infokat egymással
- Trackerek helyett elosztott hash tábla (DHT) tárolja a peereket

Elosztott hash táblák (DHT)

Céljuk kulcs-érték párok tárolása és visszakeresése P2P módon. Overlay (Virtuális hálózati topológia egy meglévő hálózat felett) hálózat, IP (Internet) felett, struktúrált P2P rendszerek. Felhasználás pl. fájlcserélés, elosztott fájlrendszerek, P2P SIP. Egy hash függvény értékkészletét elosztja a peerek között. Egy peer azokat a fájlokat/adatokat tárolja, amiknek a nevének/azonosítójának a hash értéke hozzá tartozik. A többi fájlt/adatot a szomszédos peereken keresztül találjuk meg.
Chord:

- Decentralizált: teljesen elosztott P2P rendszer, minden peer egyenlő, nincs központi elem
- Skálázhatóság: logaritmikus költségű keresés (peerek számához mérten)
- Rendelkezésre állás: mindig megtalálható az a peer akinél a keresett adat van
- Load balancing: a hash függvény egyenletesen rendeli hozzá az adatokat az egyes peerekhez

Rendeljünk minden peer-hez egy ID-t, és rendezzük őket körbe. Belépés esetén elég megkérdezni egy tagot, aki megmondja hol a helyünk, kilépéskor szomszédnak szólunk. Válasszuk úgy a kulcsokat, hogy azok értékkészlete megegyezzen a peer ID-k értékkészletével: ID(kulcs) = hash(kulcs) Egy adott <kulcs,érték> párt abban a peer-ben tároljunk, melyre távolság(GUID,kulcs) minimális. Tárolt kulcsok elosztásakor a belépő peer megkapja a szomszéd kulcsainak egy részét, kilépéskor átadja.
A körben keresés nem hatékony (O(n)), vezessünk be további kapcsolatokat! Finger table: táblázat, melynek i. sorában egy hivatkozás van egy peer-re ami legalább 2^(i-1) távolságra van, így már O(log n) lesz.
DHT BitTorrent esetén: cél a trackertől való függés csökkentése

- Kulcs: a BitTorrent fájl infohash-e (SHA-1)
- Kulcshoz tárolt adat: adott torrenthez tartozó peer-ek címei

Közösségi hálózatok

Funkcionalitás: kapcsolat nyilvántartás, tartalom megosztás, közösségi alkalmazások. Folyamatos fejlődés, növekedés, hatalmas felhasználói bázis, nehéz: hibamentes működés.
A közösségi hálózat gyakorlatilag egy framework, a kapcsolatokat felhasználhatjuk alkalmazások fejlesztéséhez. Miért jó ez? Mert könnyebben publikálhatjuk, vizsgálhatjuk a felhasználókat és van már mögöttes rendszer.

OpenSocial

Általános APIk egy halmaza web alapú közösségi hálózatokhoz (JS API, perzisztencia API, személyek kezelése, tevékenységek kezelése)

- iGoogle: modulok felvihetők, statisztikát biztosít
- iWiW: 2010től biztosít sandbox-ot

Rétegezett architektúrák

Használjuk, ha...

- ...a rendszer alacsony és magasabb szintű funkciók keveréke
- ... szeretnénk, hogy a kód változása ne gyűrűzzön végig a rendszeren
- ... fontos a modularitás, hogy
    - az egyes részek könnyen cserélhetők legyenek
    - könnyen hordozható legyen a rendszer (platformfüggetlenség)

Szigorú rétegelés: Az i. réteg felelőssége szolgáltatásokat nyújtani az i+1. rétegnek és alfeladatokat szolgáltatni az i-1. rétegnek. azonban az egyes rétegeken belül azonban tetszőleges függőségi viszonyok. (pl: vitruális futtató környezet szigorú rétegelés, API-k: nem szigorú rétegelés)
Függőleges partícionálás: Az egymástól független komponenseket egy rétegen belül csoportosíthatjuk. Előnye: a részek egymástól függetlenül cserélhetők, fejleszthetők.
Forgatókönyvek:

- Fentről le
- Lentről fel
- Megszakított - hiba esetén
- Megszakított - lalsóbb rétegek kommunikációja
- Két régetelt arch kommunikációja (hálózati stack)

Kétrétegű architektúra

Megosztott adatbázis (file-ok, DBMS) + Alkalmazások (hagymányos vagy 4GL nyelven)
Előnyök: Az adatkarbantartás elkülöníthető az alkalmazásoktól. A már meglévő adatokat több szempontból (nézetből) is megjeleníthetjük.
Hátrányok: Az adatok integritása jól csak az adatbázis szintjén megoldható – ha nincs lehetőség tárolt eljárásokra, akkor ez problematikussá válhat. Ha az adatintegritás kezelését „bedrótozzuk” az alkalmazásba, nem lehet megváltoztatni a régi alkalmazásokkal való kompatibilitás miatt. Optimalizálás denormalizálással. Az alkalmazás az adatbázis fizikai felépítését is figyelembe kell vegye, pedig neki csak a szemantikát kellene

Háromrétegű architektúra

Külső séma (Alkalmazás) + Koncepcionális séma (Tartomány) – üzleti logika + Belső séma (Adatbázis)

Előnyök: 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 (Microsoft),Sun
Hátrányok: Kevesen használják. Nagyobb erőforrásigény. Többletmunka. Az objektum-orientált adatbázisok jó felhasználási területe, de ezek teljesítőképessége kétséges

MVC

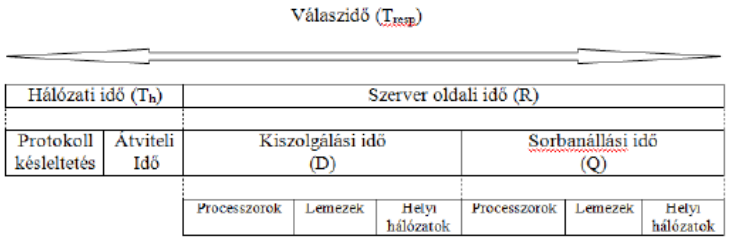
Model – alkalmazáslogika
View – megjelenítés
Controller – interakció: kommunikáció a felhasználóval
Előnyök: Többféle nézete ugyanannak az adatnak. Szinkronizált nézetek, Függetlenül cserélhető View-k és Controller-ek Lehetséges framework
Hátrányok: Komplexitás növekszik. Túl sok szükségtelen frissítés. Túl elszigetelt a Model és a Controller.

Document-View

Az MVC architektúra egy változata, a Microsoft alkalmazza az MFC-ben

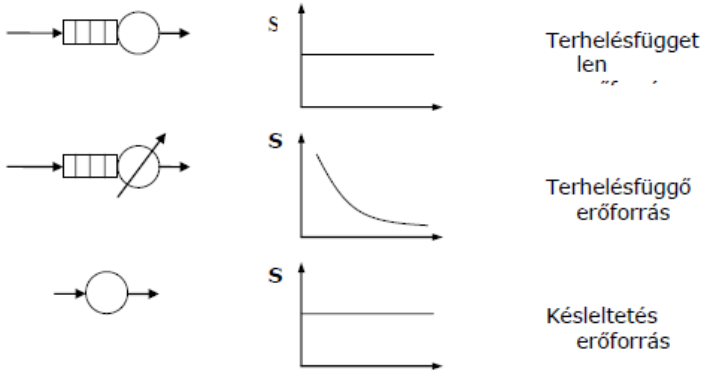
Teljesítmény

- Válaszidő (Response Time): A kérés teljesítéséhez szükséges idő. Pontosabban ez értelmezhető a szerver oldalon, vagyis a kérés megérkezésétől a válasz elküldéséig eltelt időként; illetve a kliens oldalon, vagyis a kérés elküldésétől a válasz megérkezéséig eltelt időként.
- Átbocsátó képesség (Throughput): A rendszer által egységnyi idő alatt teljesített tranzakciók száma.
- Erőforrás-használat: Az adott erőforrás (memória, merevlemez, CPU, hálózat) használatát jellemző mérőszámok.



A sorbanállási rendszerek főbb jellemzőit. Az erőforrásoknak három alapvető típusát különböztethetjük meg, ahogy azt az alábbi ábra is mutatja.

- Terhelésfüggetlen erőforrások esetén a kiszolgálási idő (amely nem tartalmazza a várakozási sorban eltöltött időt) nem függ az erőforrás terheltségétől, vagyis a várakozási sorban lévő kérések számától, a sorhossztól. Képletben kifejezve, ha n jelöli az erőforráshoz tartozó sor hosszát, S a kiszolgálási időt, S(n) = S.
- Terhelésfüggő erőforrás esetén a kiszolgálási idő függ a sorhossztól.
- A késleltetés erőforrás speciális abban a tekintetben, hogy nem tartozik hozzá várakozási sor. A kiszolgálási idő így nem is függhet a sorhossztól, ezáltal megegyezik az egység válaszidejével.



A kérések osztályokba sorolhatók. Fontos jellemzője az egyes osztályoknak, hogy a hozzájuk tartozó kérések száma a rendszerben állandó-e. Ez alapján megkülönböztetünk nyílt és zárt kérésosztályokat. A nyílt osztályú kérések érkezhetnek a rendszeren kívülről, és el is hagyhatják azt, míg a zárt osztályokba sorolt kérések a rendszeren belül keringenek, számuk nem változik. A vegyes sorbanállási rendszerek mindkét típusú kérésosztályból szolgálnak ki kéréseket.

Teljesítménymérés

- Mérések kliens oldalon: az ilyen típusú mérések elvégzése valamilyen kliens emulátorral (más néven load tester) történhet.
- Mérések az operációs rendszer szintjén: tipikusan az erőforrás használatot szokás itt mérni, ennek elvégzéséhez a különféle operációs rendszereken megfelelő eszközök állnak rendelkezésre.
- Mérések az alkalmazás szerverben: a legtöbb alkalmazás szerver gyártó igen sokféle adat monitorozását teszi lehetővé futási időben. Ezen működés engedélyezéséhez speciális, monitorozó módban kell elindítani az alkalmazás szervert.
- Más szervereken végzett mérések: az alkalmazás szerver más szerverekkel együttműködve szolgálja ki a klienseket. Ezek a webszerver, adatbázis szerver, esetleg messaging szerver. Az ezekben végzett mérésekhez általában az adott szerver gyártója kínál fel eszközöket.
- Protokoll lehallgatás: megfelelő eszközökkel monitorozhatjuk a rendszer egyes elemei között folyó kommunikációt.
- Műszerezés (Instrumentation): ez alatt azt a módszert értjük, amikor az alkalmazás forráskódjába beillesztünk olyan sorokat, amelyek lemérik és naplózzák a kód bizonyos részeinek futási idejét.