A jelen anyag doktoranduszok segítségével készült el

Charaf Hassan

Budapest, 2009. október

I. Szolgáltatás-hozzáférési és konfigurációs minták

A „Szolgáltatás-hozzáférési és konfigurációs minták (*Service Access and Configuration Patterns*)" ismertetése során négy olyan mintát mutatunk be, melyek egyedülálló és hálózati alkalmazások szolgáltatásainak és komponenseinek hozzáférésére és konfigurációjára alkalmas alkalmazásfejlesztési interfészek (*Application Programming Interface*), röviden API-k tervezését támogatják. A következő négy mintát mutatjuk be:

- Csomagoló homlokzat (*Wrapper Facade*)
- Komponens konfiguráló (*Component Configurator*)
- Elfogó (*Interceptor*)
- Kiterjesztő interfész (*Extension Interface*)

I.1. Burkoló 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.

Az alkalmazások gyakran íródnak az operációs rendszer nem-OO API-jainak és rendszerkönyvtárainak felhasználásával. Ezek az API-k szokták biztosítani többek közt a hálózati elérést, a szálkezelést, a grafikus felületek kialakítását, vagy akár az adatbázis hozzáférést. A fejlesztők számára mégis sokszor gondot okoz használatuk, mivel:

- Az alacsony szintű API-k használata gyakorta terebélyes kódhoz vezet, aminek gyakori átírása sok hibalehetőséget hordoz. Mindezt OO formában megvalósítva a kód tömörebbé, s így könnyebben fejleszthetővé, átláthatóvá válik. Az OO nyelvek kínálta magas-szintű lehetőségek (konstruktorok, destruktorok, kivétel-kezelés, garbage collection, stb) pedig tovább csökkentik a gyakori programozási hibák számát.
- Az alacsony-szintű API-k használata gyakorta eredményez nem hordozható kódot. Még ugyanazon operációs rendszer vagy fordító különböző verziói közt is jelentkezhetnek különbségek a forráskód vagy a bináris kód inkompatibilitásának következtében. A különböző operációs rendszerek, fordítók vagy hardware felületek közt hordozható szoftverek nagyobb anyagi hasznot biztosíthatnak ugyanazon fejlesztési ráfordítás mellett.
- A platform-specifikus utasítások és direktívák bonyolultabbá teszik a forráskódot, nehézkesebbé teszik a fejlesztést és a karbantartást. Így a szoftver karbantarthatóságának növelése csökkentheti a szoftver életciklusa során fellépő költségeket.
- Az alacsony-szintű API-kat ritkán csoportosítják belül szorosan összetartozó elemeket tartalmazó komponensekbe, mivel (pl. C-ben) nem használnak olyan fogalmakat, mint

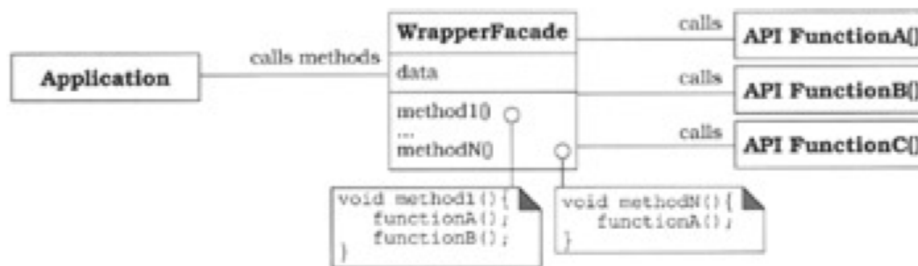
pl. osztály, névtér, csomag. Viszont a jól komponensekbe csoportosított (pl. szálkezelés, IO, felhasználói felület, stb.) API-k elsajátítása, karbantartása és továbbfejlesztése egyszerűbb lenne.

Az előbb felsorolt problémákra jelenthet megoldást az, ha 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.

A *Burkoló homlokzat* mintának két szereplője van:

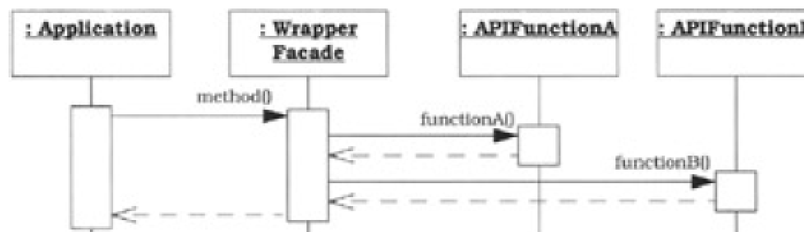
- A *függvények (API function)* a nem-OO API-k építőelemei. Ezek önálló szolgáltatásokat valósítanak meg, melyek adatok manipulációját végzik bemenő paraméterek vagy globális változók alapján.
- A *burkoló homlokzat (wrapper facade)* egy vagy több OO osztály halmaza, melyek magukba foglalják a függvényeket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

Ezeknek az osztályoknak a kapcsolatát mutatja az alábbi UML osztálydiagram:



A *Burkoló homlokzat* minta osztálydiagramja

A *Burkoló homlokzat* (*Wrapper Facade*) osztályaiban elhelyezkedő metódusok továbbítják az alkalmazás (*Application*) egy vagy több API függvényének (*API function*) meghívását a szükséges paraméter adatok átadásával. Ezen adatok többnyire az osztályok privát részében helyezkednek el. A mintának megfelelő működést a következő szekvenciadiagram mutatja:



A *Burkoló homlokzat* minta szekvenciadiagramja

A *Burkoló homlokzat* minta a következő **előnyöket** kínálja:

- Tömör, egybefüggő és robusztus magas-szintű OO interfészek.

- Hordozhatóság és karbantarthatóság.
- Modularitás, újrafelhasználhatóság és konfigurálhatóság.

A mintát a következő **hátrányok** jellemzik:

- A funkcionalitás esetleges csorbulása.
- A teljesítmény csökkenése.
- Különböző programozási nyelvek és fordítók nyújtotta korlátok.

I.2. Komponens konfiguráló (*Component Configurator*)

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. Továbbá a komponens konfiguráló a komponensek különböző processzekhez történő újratársítását is lehetővé teszi a processz leállítása és újraindítása nélkül. Ennek ott van haszna, ahol az alkalmazás vagy rendszer komponenseinek minél rugalmasabb és átlátszóbb inicializálására, felfüggesztésére, újraindítására és leállítására van szükség.

A komponens alapú alkalmazásokban tehát szükséges a komponensek egy vagy több processzhez társítása. Ezzel kapcsolatban a következő szempontok merülnek fel:

- A komponensek funkcionalitásával vagy implementációjával kapcsolatos változások igen gyakoriak bármely komolyabb rendszer vagy alkalmazás esetén. Például újabb algoritmusok vagy architektúrális megoldások születhetnek az alkalmazás fejlődése során. Ezért is lenne fontos, hogy a komponensek implementációja módosítható legyen az alkalmazás fejlesztése és üzemeltetése során. Az egyes komponenseken végrehajtott változtatások minimális hatást kell gyakoroljanak az őket használó komponensekre. Hasonlóképp célszerű az alkalmazás komponenseinek futási időben történő dinamikus inicializálása, felfüggesztése, újraindítása, leállítása vagy épp más komponensre cserélése úgy, hogy ez lehetőleg minél kisebb hatást gyakoroljon a többi alkalmazáshoz tartozó komponensre.
- A fejlesztés kezdetén nem feltétlen ismert a különböző komponensek különböző processzekhez vagy host-okhoz való leghatékonyabb hozzárendelésének módja. Ha idő előtt döntünk a komponensek elrendezésének és működésének ezen szempontjairól, úgy az később a rugalmasság, a rendszer végső teljesítményének, funkcionalitásának és erőforrás felhasználásának rovására mehet. Ezért idővel előnyös lehet (lehetőleg minél később) a szub-optimális komponens konfigurációk átrendezése. Itt különösen fontos, hogy a beavatkozás mértéke minimális legyen, azaz ne járjon például az alkalmazás leállításával.
- Célszerű, hogy a komponensek konfigurációjának, inicializálásának és vezérlésének tipikus feladata kézenfekvő, komponens-független és lehetőleg központosított, automatizált legyen.

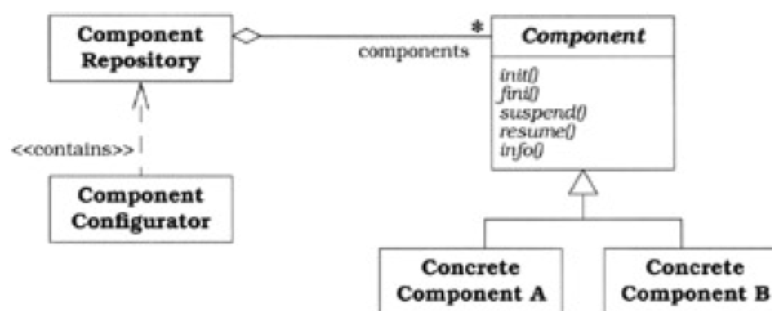
Az előbb felsorolt elvárásokat például úgy valósíthatjuk meg, hogy különválasztjuk a komponensek interfészeit az implementációjuktól, az alkalmazást pedig függetlenítjük attól, hogy a komponens-implementációk mikor lettek bekonfigurálva. Tehát a komponens kínáljon egy egységes interfészt az általa nyújtott 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. A konkrét komponensek legyenek megfelelő konfigurációs egységekbe csomagolva, mint például a dinamikusan csatolható könyvtárak (*Dynamically Linked Library (DLL)*). Ezeket a DLL-eket pedig az alkalmazás már dinamikusan ki/be csatolhatja egy komponens konfiguráló segítségével, amely egy komponens tárat tart nyilván az alkalmazáshoz aktuálisan konfigurált konkrét komponensek karbantartásához.

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. Jellemző vezérlési műveletek például a komponens inicializálása, felfüggesztése, újraindítása és leállítása.
- A *konkrét komponensek (concrete component)* a komponens vezérlő interfészét implementálják a komponens adott típusának megvalósításához. A konkrét komponens ezen felül még alkalmazásspecifikus funkcionalitást megvalósító metódusokat is implementál, például a többi egyenrangú csatolt komponenssel cserélt adatok feldolgozásához. A konkrét komponensek futási időben dinamikusan ki/becsatolható csomagokba vannak szervezve.
- A *komponenstár (component repository)* tarolja a jelenleg alkalmazásba csatlakoztatott konkrét komponenseket. Ez a tár teszi lehetővé a karbantartó rendszerek vagy az alkalmazás adminisztrátorai számára, hogy befolyásolják az alkalmazásba csatlakoztatott konkrét komponensek viselkedését egy központi adminisztratív mechanizmussal.
- A *komponens konfiguráló (component configurator)* a komponenstár felhasználásával a konkrét komponensek (újra)csatolását vezérli. Pontosabban egy olyan script-et interpretál és hajt végre, amely megadja, hogy a meglévő komponensek közül melyeket kell (újra)csatolni az alkalmazásba a DLL-ek dinamikus ki/becsatolásával.

Ezeknek az osztályoknak a kapcsolatát szemlélteti az alábbi osztálydiagram:



A komponens konfiguráló minta osztálydiagramja

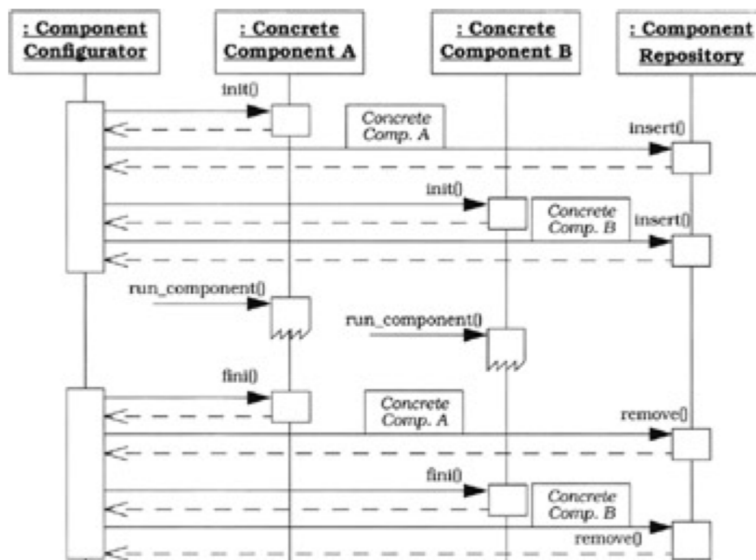
A mintának megfelelő működés lényegében három lépésre bontható:

1. **Komponens inicializálása**: A komponens konfiguráló (*component configurator*) dinamikusan csatol és inicializál egy konkrét komponenst (*concrete component*).

Miután a komponens sikeresen incializálva lett, a komponens konfiguráló behelyezi a komponenstárba (*component repository*). Ez a tár tehát futási időben tárolja a csatolt komponenseket.

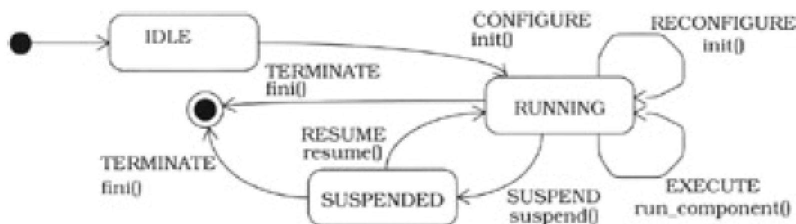
2. **Komponens végrehajtása**: Miután a konkrét komponens csatolva lett az alkalmazáshoz, elvégzi futási idejű tevékenységét, mint például az egyenrangú komponensekkel folytatott üzenetcserét vagy az igényelt szolgáltatások végrehajtását. A komponens konfiguráló ez alatt ideiglenesen bármikor felfüggesztheti és újraindíthatja a konkrét komponenseket, mint például más komponensek (újra)csatolásakor.
3. **Komponens leállítása**: A komponens konfiguráló, mielőtt leállítja azokat a konkrét komponenseket, amelyekre már nincs szükség, még lehetőséget ad számukra, hogy elengedjék az általuk lefoglalt erőforrásokat. A konkrét komponens leállításakor a komponens konfiguráló kiveszi a komponenst a komponens tárból és az alkalmazás címteréből.

A fent leírt működést az alábbi (kibővített) szekvenciadiagram szemlélteti:



A *komponens konfiguráló* minta szekvenciadiagramja

A következő UML statechart diagram azt mutatja be, hogy a komponenskonfiguráló miképpen vezérli egy-egy konkrét komponens életciklusát:



A komponens konfiguráló minta statechart diagramja

A *komponens konfiguráló* minta a következő **előnyöket** kínálja:

- Egységes komponens hozzáférés.
- Központi adminisztrálhatóság.
- Modularitás, tesztelhetőség, újrafelhasználhatóság.
- Dinamikus komponens becsatlakoztatás és vezérlés.
- Komponens konfiguráció hangolása, optimalizálása.

A mintát következő **hátrányok** jellemzik:

- Determinisztikus működés és sorrendi függőségek garanciájának hiánya a komponensek futási idejű dinamikus becsatlakoztatása következtében.
- Megbízhatatlanság és biztonságkritikusság a komponensek futási idejű dinamikus becsatlakoztatása következtében. Ha például hibásan működő komponenst csatlakoztatunk be, úgy az hibás működést eredményezhet a többi komponens, s így az alkalmazás részéről (pl. memóriaterületek nem megfelelő használata következtében). Továbbá a DLL-ek lehetőséget adnak a visszaélésre, hiszen egy „imposztor" akár komponensnek is „álcázhatja" magát.
- Futási idejű overhead és bonyolultabb infrastruktúra.
- Esetleg túl egyszerű komponens interfészek, minthogy az egységes init() és fini() metódusok nem feltétlen alkalmasak túl bonyolult vagy kontextusukkal túlontúl összeforrott komponensek inicializálására és/vagy leállítására.

I.3. Elfogó (Interceptor)

Az *Elfogó* tervezési minta arra való, hogy egy keretrendszert – egy rá építő alkalmazás fejlesztésekor – átlátszó módon egészíthessünk ki olyan alkalmazásspecifikus szolgáltatásokkal, melyeket bizonyos események bekövetkezése „triggerel".

Az olyan keretrendszerek, mint például az ORB-ok, az alkalmazás szerverek és a terület-specifikus szoftver architektúrák nem tudják előre megbecsülni az összes olyan szolgáltatást, amelyet később a felhasználók számára nyújtaniuk kell. Bizonyos keretrendszereket, különösen a fekete-doboz keretrendszereket nem célszerű olyan új szolgáltatásokkal kiegészíteni, melyekre eredetileg nem voltak felkészítve. Hasonlóképp, nem célravezető a keretrendszer használó alkalmazásokra hárítani minden szolgáltatás elvégzését, mivel ez az újrafelhasználhatóság rovására mehet. A keretrendszerek fejlesztőinek ezért tehát a következőket célszerű figyelembe vennie:

- A keretrendszernek képesnek kell lennie újabb kiegészítő szolgáltatások integrációjára anélkül, hogy architektúrájának magján változtatni kelljen.
- Egy alkalmazásspecifikus szolgáltatás integrációja nem befolyásolhatja a meglévő keretrendszer komponenseket, illetve nem teheti szükségessé a keretrendszert használó alkalmazások specifikációjának vagy implementációjának módosítását.
- A keretrendszert használó alkalmazások képesek kell legyenek a keretrendszer viselkedésének monitorozására és vezérlésére.

Az előbb felsorolt szempontoknak a következőképp felelhetünk meg: 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. Ezen felül a keretrendszer

implementációja olyan legyen, hogy ezek a külső szolgáltatások képesek legyenek működésének bizonyos részeit megfigyelni és befolyásolni.

Magyarán: a keretrendszer által feldolgozandó események egy előre megadott halmazához rendeljünk, és tegyünk elérhetővé egy-egy *elfogó interfészt*, hogy az alkalmazások ezen interfész implementációja által olyan külső szolgáltatásokat, úgynevezett *konkrét elfogókat* hozhassanak létre, melyek az említett eseményeket „fogják el", azaz dolgozzák fel alkalmazás-specifikus módon. Minden egyes elfogó számára biztosítsunk tehát egy *diszpécsert*, amely lehetővé teszi az alkalmazás számára, hogy regisztrálja konkrét elfogóikat a keretrendszernél. Így, amikor egy megfelelő esemény bekövetkezik, a keretrendszer értesítheti a diszpécsereket, hogy hívják meg a regisztrált elfogókat (azaz diszpécselje őket, pontosabban hívja meg a megfelelő „callback", avagy „hook" metódusukat). Definiáljunk továbbá olyan *kontextusobjektumokat*, melyek lehetővé teszik a konkrét elfogók számára, hogy az események hatására betekintést nyerjenek és befolyásolják a keretrendszer belső állapotának és működésének bizonyos részét. A kontextusobjektumok tehát olyan metódusokat definiáljanak, melyek hozzáférést engednek a keretrendszer belső állapotához, ezáltal „megnyitva" a keretrendszer implementációját az alkalmazás előtt. Ezeket a kontextusobjektumokat a konkrét elfogók akkor kapják meg, amikor a keretrendszer diszpécseli őket.

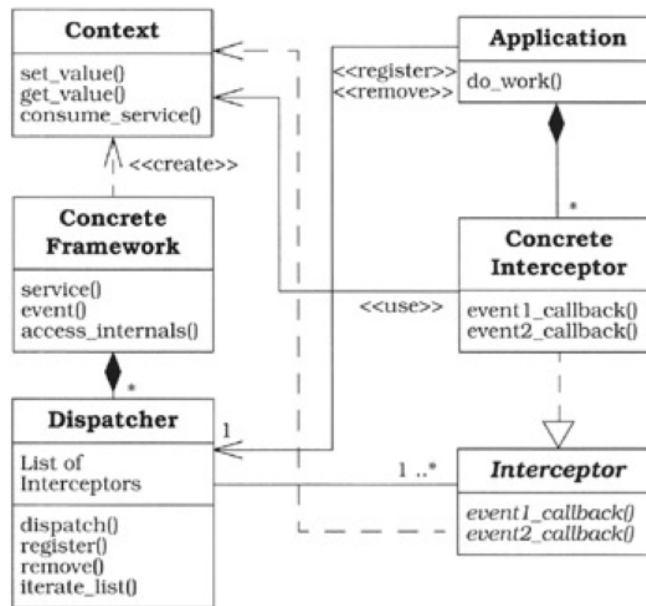
A fentebb ismertetett *Elfogó* mintának tehát a következők a szereplői:

- A *konkrét keretrendszer (concrete framework)* egy generikus és kiterjeszthető architektúrát testesít meg, amely egy adott rendszer (pl. egy ORB, egy Web-szerver vagy egy alkalmazásszerver) szolgáltatásait definiálja.
- 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. Általában egy elfogóra egy diszpécser jut. A diszpécser regisztrációs- és eltávolítási-metódusokat definiál, melyekkel az alkalmazások be-, illetve kiregisztrálhatnak egy-egy konkrét elfogót a konkrét keretrendszernél. A diszpécserek továbbá olyan interfészt is definiálnak, amit a konkrét keretrendszer meghív olyan események bekövetkeztekor, melyek kapcsán a konkrét elfogókat beregisztrálták. 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 diszpécser minden regisztrált elfogóját egy konténerben tárolja.
- 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. A keretrendszer által példányosított kontextusobjektumok a konkrét elfogók „callback" metódusainak meghívásakor adhatók át. Ebben az esetben a kontextusobjektum olyan információt is tárolhat, amely a saját keletkezését kiváltó eseményre vonatkozik. Ellentmondásos

módon kontextusobjektumot akkor is át lehet adni a konkrét elfogónak, amikor azt a diszpécsernél beregisztráljuk. Ebben az esetben a kontextusobjektum kevesebb információt szolgáltat, viszont kisebb overhead-et is okoz.

- Az *alkalmazás (application)* a konkrét keretrendszer felett fut, felhasználva annak szolgáltatásait. Az alkalmazás konkrét elfogókat is implementálhat, melyeket adott események kezelése céljából a konkrét keretrendszernél (pontosabban a diszpécsernél) beregisztrálhat. Amikor ezen események bekövetkeznek, triggerelik a konkrét keretrendszert és diszpécsereit, hogy hívják meg a konkrét elfogók megfelelő „callback" metódusait, melyek az események alkalmazás-specifikus feldolgozását végzik.

Az említett objektumok kapcsolatát mutatja az alábbi osztálydiagram:

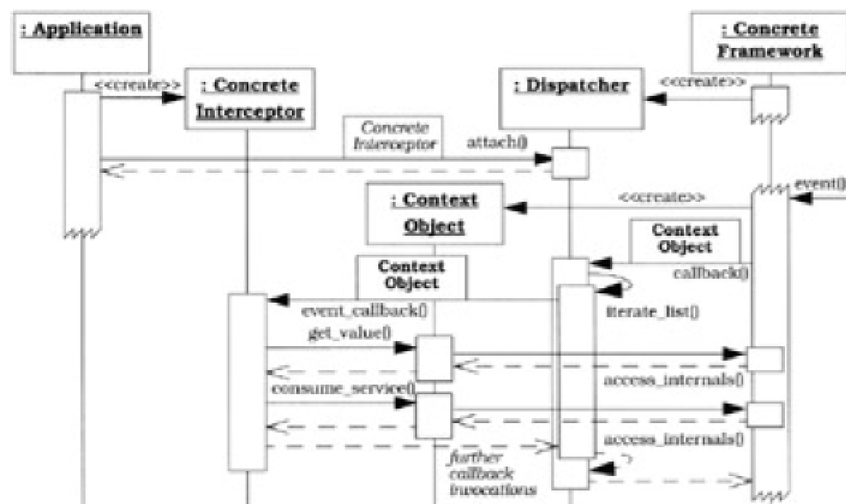


Az *Elfogó* minta osztálydiagramja

Az *Elfogó* minta tipikus működése a következő: 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 esemény-specifikus kontextusobjektumot, amely egyrészt információkat tárol az keletkezését kiváltó eseményre vonatkozóan, másrészt olyan függvényeket tartalmaz, melyek a konkrét keretrendszerhez engednek hozzáférést. Ezt követően a konkrét keretrendszer értesíti a megfelelő diszpécsert az esemény bekövetkezéséről oly módon, hogy paraméterként átadja neki az imént létrehozott speciális kontextusobjektumot. A diszpécser ennek hatására rendre végighalad a konténerében elhelyezkedő, beregisztrált konkrét elfogókon, és – a kapott kontextusobjektummal, mint bemenettel – meghívja „callback" metódusukat. A kontextusobjektum feldolgozását követően a konkrét elfogók opcionálisan meghívhatja a kontextusobjektum metódusait, melyekkel befolyásolhatja a konkrét keretrendszer működését és későbbi esemény-feldolgozását. Végül, miután mindegyik

konkrét elfogó „callback” metódusa visszatért, a konkrét keretrendszer folytatja megszokott működését.

Ezt a működést szemlélteti az alábbi (némileg bővített) szekvenciadiagram:



Az *Elfogó* minta szekvenciadiagramja

Összefoglalva, az *Elfogó* minta a következő **előnyöket** kínálja:

- Kiterjeszthetőség és rugalmasság.
- Az elfogók által megvalósított szolgáltatások (mint egyfajta „aspektus”) különválasztása az alkalmazás logikájától.
- Keretrendszerek monitorozásának és vezérlésének támogatása.
- Rétegek szerinti szimmetria, mivel a keretrendszer szimmetriáját (pl. kliens-szerver; kérdés-válasz) kihasználva a szolgáltatások megfelelő párjaikkal egyetemben valósíthatók meg.
- Az elfogók alkalmazás-szintű újrafelhasználhatósága.

A következő **hátrányok** jellemzik:

- Komplex tervezési megfontolások (pl. adott keretrendszerre épített alkalmazáshoz mennyi/milyen elfogó és diszpécser tartozzon?)
- Hibás elfogók akár az egész alkalmazás működését blokkolhatják.
- Teljesítmény visszaesés vagy holtpont jöhet létre, ha elfogási lavinák keletkezhetnek, amikor az elfogók a konkrét keretrendszer állapotához nyúlnak, s így esetleg újabb olyan eseményeket idéznek elő, melyek hatására ugyanez ismétlődik.

I.4. Kiterjesztő interfész (*Extension Interface*)

A *Kiterjesztő interfész* tervezési minta lehetővé teszi, hogy egy komponens több interfészt exportáljon, és ezzel megakadályozza az interfészek felduzzadását. Ezen túl lehetővé teszi a

komponens funkcióinak módosítását és bővítését anélkül, hogy a kliens kódját módosítani kellene.
Ugyanazon komponenshez több kiterjesztő interfész is tartozhat, melyek mindegyike egy-egy adott komponens és kliensek közti kapcsolódást definiál. Olyan alkalmazások esetén hasznos, ahol a komponensek interfészei idővel megváltozhatnak.

Az alkalmazással szemben támasztott követelmények változása az alkalmazáskomponensek funkcionalitásának megváltozását, bővülését vonhatja magával. Vannak esetek, amikor az összes interfészváltozás már akkor megjósolható, mielőtt még megkezdődne a komponens alapú alkalmazás fejlesztése. Ekkor úgynevezett bázis interfészek hozhatók létre, melyek metódusai leszármaztatással, illetve polimorfizmussal bővíthetők a későbbiekben. Ettől eltérő esetben azonban nem adhatók ilyen stabil bázis interfészek, hiszen a követelmények előre nem megjósolható módon változnak azok után, hogy a komponensek elkészültek és be lettek integrálva az alkalmazásba. Óvatlan esetben ezek a változások akár a kliens oldali kód használhatatlanságát is eredményezhetik. Ráadásul az is előfordulhat, hogy a követelmények ilyetén megváltozása által implikált új komponens-funkcionalitást csak kevés alkalmazás fogja kihasználni, míg a többi alkalmazás, amely a komponenst használja, kénytelen lesz olyan szolgáltatásokat is biztosítani, melyeket gyakorlatilag sohasem használ, és csak szükségtelen overhead-et okoznak számára. – Az említett problémák elkerülésére a komponenseket célszerű úgy megtervezni, hogy fel legyenek készítve a várható és nem várható változtatásokra egyaránt. Ebben az esetben a következőket célszerű figyelembe venni:

- Ha a komponensek interfészei nem változnak, úgy a komponensek implementációjában bekövetkező változás nem okozhat problémát a kliens oldali kódban.
- 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.

A fenti elvárásoknak úgy tudunk a legjobban megfelelni, ha a klienseket úgy alakítjuk ki, hogy a komponenseket különböző interfészeken keresztül érjék el (ahol az interfészek a komponens kliensek felé játszott különböző szerepeinek felelnek meg), ne pedig egyetlen komponenst használjanak, amelyik mindegyik szerepét egyetlen interfészbe vagy implementációba sűríti.

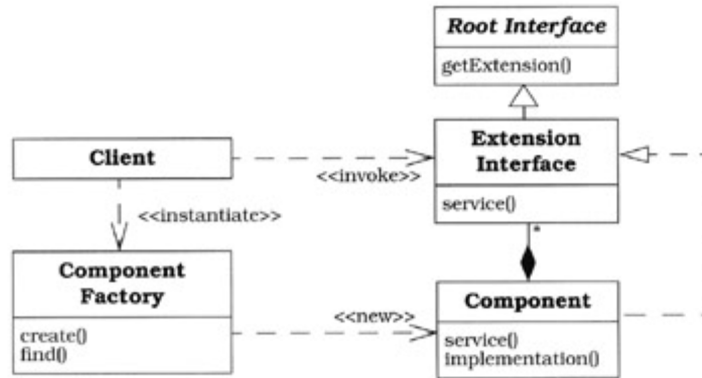
Magyarán: „exportáljuk” a komponens funkcionalitását *kiterjesztő interfészeken* keresztül, melyek mindegyike műveletek egy-egy szemantikusan összefüggő halmazának felel meg. Minden komponensnek legalább egy kiterjesztő interfészt kell megvalósítania. A komponens meglévő funkcionalitásának módosítása vagy kiterjesztése céljából inkább hozzunk létre újabb kiterjesztő interfészeket, mintsem hogy a meglévőket módosítsuk. Sőt, mi több, a klienseket úgy programozzuk, hogy a komponenseket ne az implementációjukon, hanem az interfészeiken keresztül érjék el. Ebből következően a kliensek csak a komponensek különböző – kiterjesztő interfészek által megtestesített – szerepeitől függenek.

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

A *Kiterjesztő interfész* mintának tehát lényegében öt szereplője van:

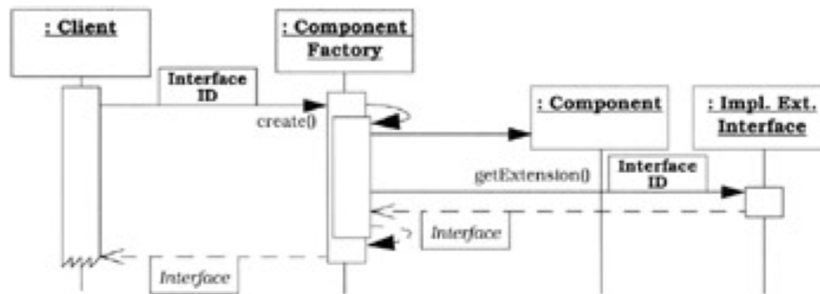
- A *komponensek (component)* különböző típusú szolgáltatásspecifikus funkcionalitást valósítanak meg. Ez a funkcionalitás gyakorta több független szerepre bontható, melyek mindegyike szemantikusan összefüggő műveletek egy halmazát határozza meg.
- A *kiterjesztő interfészek (extension interface)* a 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. Ez a megállapodás szabja meg tehát a kiterjesztő interfész metódusai meghívásának protokollját, például az elfogadható paraméter-típusokat, a metódusok meghívásának sorrendjét, stb.
- A *gyökér interfész* (*root interface*) egy speciális kiterjesztő interfész, mely háromféle funkcionalitást biztosít: (1) Az alapfunkcionalitást, amelyet mindegyik kiterjesztő interfésznek támogatnia kell (például a kliens által igényelt kiterjesztő interfész kiadása). Ez a funkcionalitás tehát meghatározza azokat az alapvető mechanizmusokat, amelyeket a komponensnek implementálnia kell ahhoz, hogy a kliens képes legyen a komponens kiterjesztő interfészei közti navigációra; (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. Nyilván, ha a kiterjesztő interfészek a gyökér interfész leszármazottai, akkor mind megvalósítják az általa megvalósított funkciókat, így bármelyikük képes a vele kapcsolatos komponenshez tartozó tetszőleges kiterjesztő interfész kiadására.
- Adott komponenstípushoz tartozó *komponensgyár (component factory)* szolgál arra, hogy a kliens számára kiadja a gyökér interfész referenciáját. A komponensgyár különválasztja a komponensek létrehozását és inicializálását a feldolgozásuktól. Ha tehát egy kliens új komponenspéldányt szeretne létrehozni, úgy ezzel a kérésével a komponensgyárhoz fordul. Miután a komponens sikeresen létrejött, a komponensgyár a komponens egy kiterjesztő interfészével tér vissza. A komponensgyár továbbá lehetővé teheti a kliensek számára azt is, hogy megadott típusú gyökér interfészt igényeljenek. Ezen felül biztosíthatnak még meglévő komponenspéldányok referenciáinak keresésére és visszaadására szolgáló funkcionalitást is.

Ezeknek az osztályoknak a kapcsolatát mutatja az alábbi osztálydiagram:



A *Kiterjesztő interfész* minta osztálydiagramja

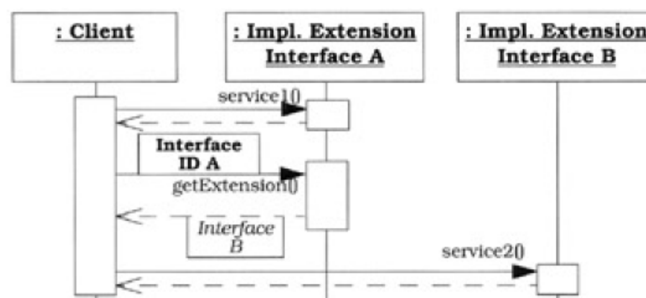
A *Kiterjesztő interfész* minta két legfontosabb kollaborációját mutatjuk be: (1) a kliensek új komponenseket hoznak létre, és beszereznek egy gyökér interfészt. Ezt mutatja a következő (bővített) szekvenciadiagram:



A *Kiterjesztő interfész* minta szekvenciadiagramja (1. eset)

Első lépésben tehát a kliens (*client*) egy új komponens (*component*) létrehozását kéri: ennek során a komponens által megvalósított (*interface ID-val azonosított*) kiterjesztő interfészt (*impl.ext.interface*) igényel a komponensgyártól (*component factory*). Ennek hatására a komponensgyár létrehozza az új komponenst, és lekéri gyökér interfészének referenciáját. Ezt követően a komponensgyár igényli az adott kiterjesztő interfészt a gyökér interfésztől, majd a kapott kiterjesztő interfész referenciájával tér vissza a klienshez.

A (2)-es eset a kliensek és a kiterjesztő interfészek közti tipikus kollaboráció. Ezt mutatja a következő (bővített) szekvenciadiagram:



A *Kiterjesztő interfész* minta szekvenciadiagramja (2. eset)

Az ábrán az látható, amint a kliens az A kiterjesztő interfészen (ami akár gyökér interfész is lehet) hív meg egy metódust. Ennek hatására az A kiterjesztő interfész implementációja (maga a komponens) végrehajtja a kijelölt metódust, és visszatér annak eredményével (ha van) a klienshez. Ezek után a kliens meghívja az A kiterjesztő interfész getExtension() metódusát, melynek bemenetéül egy olyan paramétert ad, amely a kívánt (az A-tól eltérő) kiterjesztő interfészt azonosítja. Mivel a getExtension() metódust a gyökér interfész definiálja, ezért mindegyik kiterjesztő interfész támogatja. Így tehát az A kiterjesztő interfész implementációja (maga a komponens) megkeresi a kívánt – mondjuk – B kiterjesztő interfészt, és visszaadja a rá mutató referenciát a kliensnek. Ezt követően a kliens meghív egy metódust a B kiterjesztő interfészen, minek hatására az interfész implementációja (maga a komponens) végrehajtja a kijelölt metódust, és visszatér annak eredményével (ha van) a klienshez.

A *Kiterjesztő interfész* minta a következő **előnyöket** kínálja:

- Kiterjeszthetőség.
- Komponensek szerepeinek különválasztása.
- Polimorfizmus közös interfészből való leszármazás nélkül.
- Komponensek és kliensek különválasztása az implementáció és interfész különválasztásával.
- Interfész aggregáció és delegáció támogatása.

A mintát a következő **hátrányok** jellemzik:

- Megnövekedett komponens tervezési és implementációs igény.
- A kliens programozása bonyolultabb.
- Újabb indirekció és futási idejű overhead.

II. Eseménykezelési minták

II.1. Bevezetés

A hálózati kommunikáció megvalósításánál, azon belül is elsősorban a szervereknél a szinkron feladat végrehajtás nem járható út. Elengedhetetlen a szolgáltatás kérelmek párhuzamos kiszolgálása.
Ha limitált a kapcsolatok száma és az egyes kapcsolatok közötti adatátvitel minimális, akkor lehetséges, hogy minden kapcsolatot külön szál szolgáljon ki. Azonban ha a szálak száma túl nagy, akkor a taszkváltások magas száma miatt lecsökken a rendszer teljesítménye. Ha pedig a szálak között nagymértékű az adatátvitel, akkor a szinkronizálások miatt akadozik a kiszolgálás.
Ezekben az esetekben az eseményalapú kezelés lehet a megoldás.

II.2. Reactor minta

A Szolgáltatáskérések szétosztása minta lehetővé teszi az esemény-vezérelt alkalmazások számára olyan szolgáltatáskérések szétosztását és irányítását, amelyek egy vagy több klienstől érkeznek be.
A minta jól alkalmazható helyzetekben, amikor olyan esemény-vezérelt alkalmazásról van szó, amely több szolgáltatáskérést fogad egyszerre, de azokat szinkron módon és sorosan, egymás után dolgozza fel. Az elosztott rendszerek eseményvezérelt alkalmazásainak, például kiszolgálóknak, fel kell készülniük a szolgáltatáskérések egyidejű kezelésére, még akkor is, ha végül sorosan dolgozzák fel őket. Így mielőtt a szolgáltatásokat végrehajtanák, az alkalmazásnak szét kell osztania és a megfelelő szolgáltatás megvalósításhoz kell irányítania a beérkezett kéréseket. A probléma megoldásához négy követelményt kell teljesíteni:

- Skálázhatóság és késleltetés javítása. Az alkalmazás nem blokkolhat, valamint nem zárhat ki kéréseket.
- Az átviteli kapacitás maximalizálása. A felesleges környezetváltásokat, CPU-k közötti adatátviteleket ki kell küszöbölni.
- A már meglévő szétosztási eljárásokhoz és irányítási metódusokhoz egyszerűen lehessen új vagy javított szolgáltatásokat hozzárendelni.
- Az alkalmazás kódjának védettnek kell lennie azoktól a következményektől, amelyek a többszálas és szinkronizációs mechanizmusok bonyolultságából adódnak.

A megoldáshoz 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. 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) használ, mikor a jelző eseményekre várakozik. Ezek bekövetkeztekor értesíti a reaktort, amely szinkron módon elindítja az eseményhez rendelt eseménykezelőt, amely magát a szolgáltatást nyújtja.

Felépítés

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. Amikor egy jelző esemény előáll egy eseményforrásnál, a jelző esemény bekerül a hozzárendelt kezelő sorába, valamint a kezelő „kész” állapotú lesz.
- Szinkron esemény szétválasztó, amely egy vagy több jelző esemény bekövetkezésére vár a kezelők egy halmazán (kezelőhalmaz). A függvény addig blokkolt állapotban van, amíg valamely kezelőhalmazon egy esemény „kész” jelzést nem kap.
- 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. Minden konkrét eseménykezelő olyan kezelőhöz van hozzárendelve, amely az alkalmazáson belül azonosítja a szolgáltatást. A gyakorlatban ez azt jelenti, hogy a konkrét eseménykezelők implementálják a kampó eljárásokat, amelyek a hozzájuk rendelt kezelőiken érkezett jelző események feldolgozásáért felelősek.
- 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). A reaktor a saját szinkron esemény szétosztóját használja a jelző események bevárására. Amikor ezek az események bekövetkeznek, a rektor először demupltiplexálja a jelző eseményt, majd elindítja a megfelelő kampó eljárást az esemény feldolgozására.

Figyeljük meg, hogy a Szolgáltatáskérések szétosztása minta hogyan „fordítja meg” a vezérlés irányát az alkalmazásban. A reaktor dolga, nem pedig az alkalmazásé, hogy várjon a jelző eseményekre, szétossza azokat, majd elindítsa a megfelelő kampó eljárást. A gyakorlatban a konkrét eseménykezelők nem hívják meg a reaktort, hanem a reaktor indítja el a konkrét eseménykezelőket, amelyek „reagálnak” az specifikus esemény bekövetkeztére. Ezt a vezérlési irány megfordulást nevezik Hollywood elvnek. Így az alkalmazás-fejlesztők feladata csupán a konkrét eseménykezelők implementálása és a regisztrálásuk a reaktornál. A reaktor szétosztó és irányító mechanizmusa újra felhasználható.

Működés

Az alkalmazás regisztrál egy konkrét eseménykezelőt a reaktornál. Az alkalmazás ekkor azt is jelzi, hogy milyen jelző esemény típusról szeretne az eseménykezelő értesítést kapni a reaktortól. A reaktor utasítja az eseménykezelőt, hogy adja meg a belső kezelőjét (internal handle). Ez a kezelő azonosítja a jelző esemény forrását a szinkron esemény szétosztó és a számítógép felé. Miután minden eseménykezelőt regisztráltak, az alkalmazás elindítja a reaktor eseményhurokját. Ekkor a reaktor minden egyes regisztrált eseménykezelő kezelőjét egy kezelőhalmazba gyűjti össze. Ezután meghívja a szinkron esemény szétosztót, amely várakozik a kezelőhalmazon bekövetkező jelző eseményekre. A szinkron esemény szétosztó visszatér a reaktorhoz, amikor egy vagy több kezelő „kész”-é válik. A reaktor ezután a kezelőket „kulcsok”-ként alkalmazva felderíti a megfelelő eseménykezelő(ke)t, és elindítja a hozzájuk rendelt kampó eljárás(oka)t. A jelző esemény típusa paraméterként átadható, amely felhasználható további alkalmazás specifikus feldolgozások végrehajtására. Miután a megfelelő eseménykezelő elindult, feldolgozza a meghívott szolgáltatást.

Implementáció
A Szolgáltatáskérések szétosztása minta implementációja két rétegre bontható:

- Szétosztó/irányító infrastruktúra réteg komponensek. Ez a réteg általános, alkalmazás független stratégiákat valósít meg a jelző események eseménykezelőkhöz való szétosztásakor, és utána a hozzárendelt eseménykezelő kampó eljárások elindításkor.
- Alkalmazás réteg komponensek. Ez a réteg konkrét eseménykezelőket definiál, amelyek a saját alkalmazás specifikus kampó eljárásaikat valósítják meg.

Az implementáció egy lehetséges megvalósításának lépései:

1) Az eseménykezelő interfészének definiálása
   a) Meghatározása az irányításkor elindítani kívánt célnak
      i) Eseménykezelő objektumok
      ii) Eseménykezelő függvények
   b) Az eseménykezelő irányításakor/elindításakor alkalmazott interfész stratégiájának meghatározása
      i) Külön-eljárású irányító interfész stratégia
      ii) Többes-eljárású irányító interfész stratégia
2) A reaktor interfészének definiálása
   a) Két paraméterű
   b) Három paraméterű
3) A reaktor interfészének implementálása
   a) Reaktor implementáció hierarchia kialakítása
   b) Szinkron esemény szétosztó mechanizmus kiválasztása
   c) A szétosztó tábla implementálása
   d) A konkrét reaktor implementáció definiálása
4) Az alkalmazás által igényelt reaktorok számának meghatározása
5) A konkrét eseménykezelők implementálása
   a) A konkrét eseménykezelők állapotát leíró házirend meghatározása
   b) Stratégia implementálása a konkrét eseménykezelők kialakítására az eseménykezelőkből
      i) Hard-coding
      ii) ii. Generikus
   c) A konkrét eseménykezelők funkcionalitásának implementálása

Módosítási lehetőségek
Annak érdekében, hogy a Szolgáltatáskérések szétosztása minta támogassa a konkurens működést, az újra-belépést vagy az időzítés alapú eseményeket, bizonyos módosításokat kell tenni. Ha a reaktor a fő ciklust egyszálú módon futtatja, akkor nincs szükség zárokra, mivel az eseménykezelők kampó eljárásai sorosan futnak le. Ám az egyszálú szétosztó/irányító működhet többszálas alkalmazásban is. Ekkor annak ellenére, hogy a reaktor fő eseményhurokját csak egy szál futtatja, több szál regisztrálhat és távolíthat el kezelőket a reaktornál. A reaktor által meghívott eseménykezelők pedig konkurens módon megoszthatják állapotukat más szálakkal is. Ezért a következő problémákat kell orvosolni:

- a versenyhelyzet megakadályozása,
- holtpont megakadályozása és
- a reaktor eseményhurkának várakozó szálait explicit módon kell értesíteni.

A lehetséges módosításkor szükséges új komponensek:

- Konkurens eseménykezelők: Az egyszálú eseménykezelőkhöz képest a konkurens eseménykezelők külön-külön szálon futhatnak.
- Konkurens szinkron esemény szétosztók: Az olyan szinkron esemény szétosztók, amelyek képesek konkurens módon működni, a kezelőket visszaadják az őket meghívó egyik szálnak, majd a kezelő pedig meghívja a megfelelő kampó eljárást, így az áteresztőképesség megnő a párhuzamos működés következtében. Hátránya viszont, hogy a reaktor implementációja sokkal összetettebbé és sokkal kevésbé hordozhatóvá válik.
- Újra-belépő reaktorok: Általánosságban a konkrét eseménykezelők csak „reagálnak" a meghívásuk esetén, ők maguk nem hívják meg a reaktor eseményhurokját. Ám bizonyos esetekben szükség lehet adott események lekérdezésére a reaktor eseményhurkának meghívása által. Egy gyakori stratégia a reaktor újra-belépővé módosítására az, hogy a szétosztó táblában levő kezelőhalmaz állapot információit a run-time stack-be másoljuk, mielőtt a szinkron esemény szétosztót meghívjuk.
- Időzítő és I/O események integrált szétosztója: A reaktorba épített időzítő mechanizmus lehetővé teszi idő-alapú konkrét eseménykezelők regisztrálását, amelyek egy megadott későbbi időpontban meghívják a megfelelő időzítő eseményt.

Néhány alkalmazás
InterViews, Xt toolkit, ACE Reactor Framework, ORB Core, Call Center Management, Project Spectrum, Receiving Phone Calls.

Következmények
A Szolgáltatáskérések szétosztása minta a következő előnyökkel jár:

- Modularitás, újra felhasználhatóság és széles körű beállíthatóság
- Hordozhatóság
- Durvaszemcsés konkurenciakezelés (A soros végrehajtás eredményeképpen nem kell törődni a bonyolult szinkronizációs eljárásokkal)

A Szolgáltatáskérések szétosztása minta hátrányai:

- Korlátozott felhasználhatóság
- Nem megszakításos működés
- A hibakeresés és tesztelés bonyolult

II.3. Proactor

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 elosztott környezetben működő esemény-vezérelt alkalmazások, például szerverek, teljesítménye nagymértékben javítható a szolgáltatáskérések aszinkron feldolgozása által. Amikor a szolgáltatás aszinkron feldolgozása befejeződik, az alkalmazásnak az operációs rendszer által továbbított megfelelő végrehajtási eseményt kezelnie kell annak érdekében,

hogy jelezze az aszinkron számítássorozat végét. Egy alkalmazás például elirányíthat egy végrehajtási eseményt egy belső komponenshez, amely az aszinkron feldolgozást végzi. Ez a komponens a végeredményt továbbíthatja egy külső komponensekhez ugyanúgy, mint a belső eredetei kiszolgálóhoz. Ehhez az aszinkron számítási modellhez négy követelményt kell tejesíteni:

- A skálázás és késleltetés javítása. Az alkalmazásnak párhuzamosan több végrehajtási eseményt kell feldolgoznia anélkül, hogy a sokáig tartó műveletek késleltetnék más műveletek feldolgozását.
- Az átvitel maximalizálása. Minden felesleges környezetváltást, szinkronizációs műveletet és CPU-k közötti adatátvitelt kerülni kell.
- A már meglévő szétosztási eljárásokhoz és irányítási metódusokhoz egyszerűen lehessen új vagy javított szolgáltatásokat hozzárendelni.
- Az alkalmazás kódjának védettnek kell lennie azoktól a következményektől, amelyek a többszálas és szinkronizációs mechanizmusok bonyolultságából adódnak.

A megoldáshoz 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.

Ez a gyakorlatban azt jelenti, hogy az alkalmazás által kínált minden szolgáltatást el kell látnunk egyrészt aszinkron műveletekkel, melyek kezelőkön keresztül elkezdik a szolgáltatáskérések feldolgozását, továbbá végrehajtási kezelőkkel, amelyek feldolgozzák a végrehajtási eseményeket az aszinkron műveletek eredményeivel együtt. Az aszinkron műveletet egy kezdeményező (initiator) hívja meg, amelyet egy aszinkron műveleti feldolgozó (asynchronous operation processor) futtat. A művelet befejeződésekor a feldolgozó a művelet eredményét tartalmazó végrehajtási eseményt egy végrehajtási eseménysorba helyezi.

A soron egy aszinkron esemény szétosztó áll rendelkezésre, amelyet a proaktor hív meg. Amikor az aszinkron esemény szétosztó eltávolítja a végrehajtási eseményt a sorából, a proaktor elvégzi a szétosztást és a továbbküldést az eseményen az aszinkron művelethez rendelt alkalmazás specifikus végrehajtási kezelőhöz. Ez pedig feldolgozza az aszinkron művelet eredményeit, majd esetlegesen további aszinkron műveleteket hív meg, amelyek ugyanezen feldolgozási láncot követik.

Felépítés

Az Aszinkron műveletek feldolgozása minta kilenc részből áll:

- Kezelők, amelyeket az operációs rendszer szolgáltat olyan elemek azonosítására, amelyek végrehajtási eseményeket generálnak. A végrehajtási események keletkezhetnek külső szolgáltatáskérésekre adott válaszként (pl. távoli alkalmazások adatkéréseinél), vagy az alkalmazás belső műveleteire adott válaszként (pl. időzítők jelzései vagy aszinkron I/O rendszerhívások).
- Az aszinkron műveletek általában sokáig tartó műveleteket jelentenek, amelyek a szolgáltatások implementációiban szerepelnek. A meghívás után a meghívója vezérlőszálának blokkolása nélkül lefut, így lehetővé teszi a hívónak más műveletek elvégzését. Abban az esetben, ha várakoznia kell egy esemény bekövetkeztére, akkor a futás addig késletetésre kerül.

- 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 végrehajtási események az aszinkron műveletek befejeződésekor keletkeznek.
- 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. Amikor a művelet befejeződik, a feldolgozó előállítja a megfelelő végrehajtási eseményt, majd ezt elhelyezi azon kezelő végrehajtási eseménysorába, amely a műveletet futtatja.
- 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. A kezdeményező gyakran feldolgozza az általa meghívott aszinkron művelet eredményét, így ilyenkor egy konkrét végrehajtási kezelő szerepét is játssza.

Az aszinkron műveletek feldolgozása minta komponensei proaktív entitásokként működnek. Az alkalmazáson belüli vezérlési- és adatfolyamot aszinkron műveletek meghívása által proaktívan vizsgálják. Az aszinkron műveleti feldolgozó és a proaktor a végrehajtási eseménysoron keresztül működnek együtt. A sort használják a kapott végrehajtási események visszaadására a megfelelő konkrét végrehajtási kezelőhöz, és azok kampó eljárásai kezdeményezésére. A végrehajtási esemény feldolgozása után a végrehajtási kezelő proaktívan kezdeményezhet egy új aszinkron műveletet.

Működés

A Proakor mintában a következő komponensi együttműködések fordulnak elő:
Egy alkalmazás, amely a kezdeményező szerepét játssza, meghívja az aszinkron műveletet az aszinkron műveleti feldolgozón egy adott kezelőn keresztül. Az adatok paraméterein kívül a kezdeményező egyéb végrehajtási paramétereket is átad, például a végrehajtási kezelőjét, illetve a végrehajtási eseménysor kezelőjét.
Ezután a művelet és a kezdeményező egymástól függetlenül futnak. A gyakorlatban a kezdeményező új aszinkron műveleteket hívhat meg, mialatt a többi fut. Abban az esetben, ha egy művelet egy távoli alkalmazástól szeretne szolgáltatáskérést kapni, a műveletet az aszinkron műveleti feldolgozó a kérés beérkeztéig felfüggeszti.
Amikor az alkalmazás készen áll feldolgozni az aszinkron műveletek eredményeként kapott végrehajtási eseményt, meghívja a proaktor eseményhurokját. Ez az eljárás várakozik a végrehajtási eseménysorára érkező eseményekre. A sorból való eltávolítás után a proaktor eljárása szétosztja az eseményt és továbbítja a sor végrehajtási kezelőjéhez. Ez pedig elindítja a végrehajtási kezelő megfelelő kampó eljárását, átadva az aszinkron művelet eredményét.

A konkrét végrehajtási kezelő feldolgozza a kapott végrehajtási eredményt. Abban az esetben, ha a végrehajtási kezelő az eredményeket visszaadja a hívójának, két eset lehetséges. A cél végrehajtási kezelő szintén lehet kezdeményező, ebben az esetben az eredmény visszajuttatása a hívóhoz több munkát nem igényel, mert ő maga a hívó. Abban az esetben viszont, ha egy távoli alkalmazás, vagy egy belső komponens kérte az aszinkron műveletet, a végrehajtási kezelő meghívhat egy aszinkron írási műveletet az átviteli kezelőn annak érdekében, hogy az eredményeket visszaadja a távoli alkalmazásnak.
Miután a végrehajtási kezelő befejezte a feldolgozást, más aszinkron műveletet indíthat el, amely esetén a fent bemutatott ciklus újrakezdődik.

Implementáció

Az Aszinkron műveletek feldolgozása minta implementációja két rétegre bontható:

- Szétosztó/irányító infrastruktúra réteg komponensek. Ez a réteg általános, alkalmazás független stratégiákat valósít meg az aszinkron műveletekhez. Szétosztást és irányítást is végez a végrehajtási eseményeken, amely során az aszinkron műveletektől érkező végrehajtási eseményeket eljuttatja a hozzájuk rendelt végrehajtási kezelőkhöz.
- Alkalmazás réteg komponensek. Ez a réteg aszinkron műveleteket és konkrét végrehajtási kezelőket definiál, amelyek a saját alkalmazás specifikus szolgáltatás feldolgozásokat valósítanak meg.

Az implementáció egy lehetséges megvalósításának lépései:

1. Aszinkron műveletek és végrehajtási kezelőkben különálló alkalmazás szolgáltatások tervezése
2. A végrehajtási kezelők interfészének definiálása
    a. Az aszinkron műveletek eredményeinek szállítási eljárásának definiálása
    b. A továbbítási célpont típusának meghatározása
    c. A végrehajtási kezelő továbbítási interfészének stratégiájának definiálása
        i. Különálló továbbítási interfész stratégia
        ii. Többszörös továbbítási interfész stratégia
3. Az aszinkron műveleti feldolgozó implementálása
    a. Az aszinkron művelet interfészének definiálása
        i. A hordozhatóság és rugalmasság maximalizálása
        ii. Többszörös végrehajtási kezelők, proaktorok és végrehajtási eseménysorok hatékony és lényegre törő kezelése
    b. Az aszinkron műveleti feldolgozó mechanizmusának kiválasztása
4. A proaktor interfészének definiálása
5. A proaktor interfészének implementálása
    a. A proaktor implementáció hierarchiájának kifejlesztése
    b. Végrehajtási eseménysor és aszinkron esemény szétosztó mechanizmusának kiválasztása
        i. FIFO szétosztás
        ii. Szelektív szétosztás
    c. Annak meghatározása, hogy hogyan irányítsuk a végrehajtási eseményeket a végrehajtási kezelőkhöz

    d. Annak meghatározása, hogy hogyan indítsuk el a kampó eljárásokat a kijelölt végrehajtási kezelőkön
    e. A konkrét proaktor implementáció definiálása
6. Az alkalmazás proaktorainak számának meghatározása
7. A konkrét végrehajtási kezelők implementálása
    a. A konkrét végrehajtási kezelők állapotának karbantartásának házirendjének meghatározása
    b. A konkrét végrehajtási kezelők konfigurációjának mechanizmusának kiválasztása
    c. A végrehajtási kezelők funkcionalitásának implementálása
8. A kezdeményezők implementálása

Módosítás lehetőségek

Aszinkron végrehajtási kezelők. Amikor a konkrét végrehajtási kezelőt elindítják, a proaktor szálát kéri kölcsön a végrehajtás feldolgozásához. Ez a modell viszont korlátozza a konkrét végrehajtási kezelőt abban, hogy rövid idejű szinkron feldolgozásokat hajtson végre azért, hogy a teljes válaszidő jelentős csökkenését elkerülje. Ennek feloldásához a végrehajtási kezelőknek kezdeményezőként kell működniük, és hosszú idejű aszinkron műveleteket kell azonnal meghívniuk, a szinkron végrehajtási feldolgozás helyett.

Konkurens aszinkron esemény szétosztók. Egy jobban skálázható stratégia lehet egy szálkészlet létrehozása, amely egy közös aszinkron esemény szétosztón osztozik, így a proaktor a végrehajtási kezelők szétosztását és irányítását konkurens módon végezheti.

Osztott végrehajtási kezelők. A kezdeményezők több aszinkron műveletet indíthatnak el párhuzamosan, amelyek mindegyike ugyanazon konkrét végrehajtási kezelőn osztozik. A helyes működéshez azonban minden osztott kezelőnek egyértelműen meg kell tudnia határoznia, hogy melyik aszinkron művelet fejeződött már be. Ebben az esetben a kezdeményező és a proaktor kénytelen együttműködni annak érdekében, hogy a művelet-specifikus állapot információt az egész aszinkron feldolgozási életcikluson keresztül felügyelje.

Aszinkron műveleti feldolgozó emuláció. Számos platform (pl. hagyományos UNIX verziók és a Java Virtual Machine) nem kínál aszinkron műveleti lehetőséget az alkalmazások számára. Vannak technikák, amelyek segítségével emulálni lehet egy aszinkron műveleti feldolgozót. Egy általános megoldás konkurencia mechanizmusok használata műveletek futtatására a kezdeményezők blokkolása nélkül. Három tevékenységet kell megvalósítani a többszálú aszinkron műveleti feldolgozóhoz:

- műveleti végrehajtás indítása,
- aszinkron művelet futtatása és
- műveleti végrehajtás kezelése.

Számos egyéb módosítások is lehetséges, mint a Szolgáltatáskérések szétosztása mintánál, például az időzítő és I/O műveletek szétosztása, valamint konkurens konkrét végrehajtási kezelők támogatása.

Néhány alkalmazás

Végrehajtási portok a Windows NT-ben, az aszinkron I/O műveletek POSIX AIO családja, ACE Proactor Framework, operációs rendszer eszköz meghajtó programjainak megszakítás-kezelő mechanizmusa, telefonhívás kezdeményezés hangpostán keresztül.

Következmények
Az Aszinkron műveletek feldolgozása a következő előnyökkel jár:

- Az egyes problémakörök különválasztása
- Hordozhatóság
- A konkurensen működő mechanizmusok beágyazása
- A száltechnika és a konkurencia szétválasztása
- Teljesítmény
- Az alkalmazás szinkronizációjának egyszerűsödése

A minta hátrányai:

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

II.4. Asynchronous Completion Token (ACT)

A probléma
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.

A megoldás
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.

A minta egyszerű leírása

1) Az aszinkron művelet létrehozásakor az ACT létrehozása.
2) Az aszinkron művelet futása során az ACT változatlanul őrződik.
3) A feldolgozás után az eredménnyel együtt az ACT is visszakerül a hívóhoz.
4) A hívó az ACT alapján meghívja az eredménykezelőt.

A minta részletes leírása

A struktúra elemei

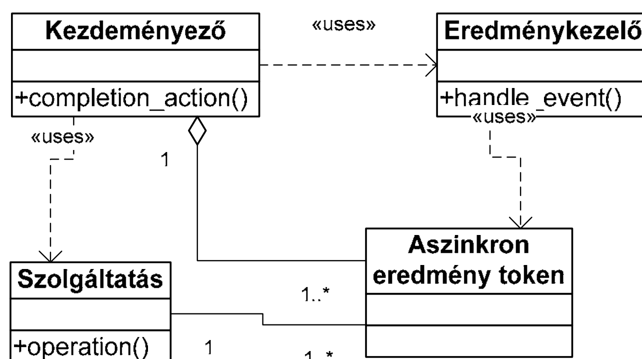
Szolgáltatás
A szolgáltatás olyan műveleteket takar, amelyek aszinkron meghívhatóak.

Kezdeményező
A kezdeményező hívja meg az aszinkron szolgáltatásokat. Továbbá a válaszokat eljuttatja az eredménykezelőkhöz, amely feldolgozza a szolgáltatás eredményeit.

Aszinkron eredmény token

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.

Eredménykezelő
Az eredménykezelő funkcionalitása megegyezik a proactor mintában szereplővel.



Az aszinkron eredmény token strukturális felépítése

A folyamat
Az előző strukturális ábra alapján a következő folyamat zajlik le:

1) Mielőtt a kezdeményező meghívná a szolgáltatást, előtte az aszinkron művelethez legenerálja az aszinkron eredmény tokent, amely majd azonosítja az eredménykezelőt.
2) A szolgáltatás meghívásakor a kezdeményező az ACT-t is átadja a művelet paramétereivel együtt.
3) A kezdeményező további műveleteket is elindíthat hasonló módon.
4) Az aszinkron művelet végén a szolgáltatás visszaadja az eredményeket a kezdeményezőnek az ACT-vel együtt.
5) A kezdeményező az ACT alapján azonosítja az eredménykezelőt és átadja a szolgáltatás válaszát.
6) Az eredménykezelő elvégzi az alkalmazás specifikus műveleteket az adatokkal.

Jellemzői

Előnyök
A minta elsősorban 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 mintánál tárgyaltakkal.

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

Implementáció

Implementációs irányelvek

A főbb implementációs irányelvek megegyeznek a proactor mintánál tárgyaltakkal. Ezt természetesen kiegészíti az ACT implementációja.
Az ACT-t reprezentálhatjuk egy objektummal, illetve a rá mutató pointerrel, de használhatunk egy statikus tömböt is, ahol egy indexszel hivatkozhatunk rá. Azonban mindkét esetben gondoskodnunk kell a felszabadításról. Ezt célszerűen a kezdeményezőnek kell elvégeznie. Ha az eredménykezelőre bíznánk, akkor a programozók könnyen elfelejtkeznének róla.

Támogatások az operációs rendszerekben

Az aszinkron I/O műveleteket támogató operációs rendszerek támogatják az ACT-t is. Vagyis az MS Windows, a Posix alapú Unix és a Linux is ide tartozik.

II.5. Acceptor-Connector

A probléma

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.
Ugyanakkor a kapcsolat felépítésének metódusa szinte minden kapcsolat alapú hálózati protokollnál egyforma, így tekinthetjük alkalmazás független, újra felhasználható kódnak.
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. Hiszen ebben az esetben bármelyik oldal lehet a kezdeményező (aktív).
Arra is szükségünk lehet, hogy könnyen új szolgáltatásokat adjunk a szerverhez a korábbi szolgáltatások zavarása nélkül, így a megoldásunkba ezt is célszerű bele venni.

A megoldás

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.
Az így kialakult struktúrában van egy komponens, amely a kapcsolatot kezdeményezi, és egy másik, ami fogadja. Akár kezdeményező, akár fogadó szerepet tölt be az alkalmazás a kapcsolat felépítésekor meghívódik a szolgáltatás, amely a hálózati kommunikációt folytatja.
Ez a struktúra megoldja a felvázolt problémákat, célokat.

A minta egyszerű leírása

Az alkalmazásunk működhet aktív és passzív módban is.
Passzív mód:

1) Az acceptor várja a kapcsolatokat.
2) Bejövő kapcsolat esetén az acceptor felépíti a kommunikációs csatornát.
3) Meghívódik a szolgáltatás és a továbbiakban ő folytatja a kommunikációt.

Aktív mód:

1) A connector felépíti a kommunikációs csatornát.
2) Meghívódik a szolgáltatás és a továbbiakban ő folytatja a kommunikációt.

A minta részletes leírása

A struktúra elemei

Szolgáltatás

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, így a lényegi hálózati kommunikációt ő folytatja. A kommunikációhoz tárol egy csatorna leírót, amelyet a connector-tól vagy az acceptor-tól vesz át. Emellett tartalmaz egy indító függvényt, amellyel a szolgáltatás kezelése lényegében beindul.

Acceptor

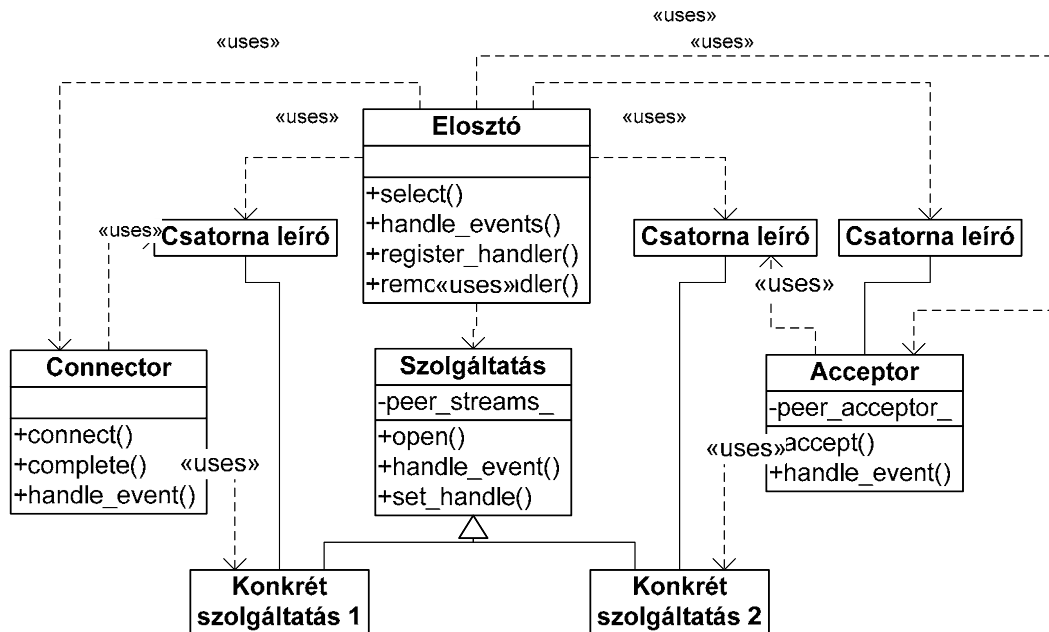
Az acceptor valósítja meg a passzív kapcsolat felépítést végző komponenst.
Az inicializációja során létrehoz egy passzív végpontot, amely várja a kapcsolódásokat. TCP esetén ez azt jelenti, hogy létrehoz egy socketet, hozzáköti (bind) egy porthoz és bekapcsolja a szerver módot (listen). Ezt követően vár a kapcsolódásokra (accept).
Amikor egy kapcsolat beérkezik, akkor létrehoz egy kommunikációs csatornát a túloldallal (kliens socket). Ezt követően elindítja a szolgáltatás kezelőt, amely majd a kommunikációt folytatja és átadja neki a csatorna leíróját. Majd elindítja a szolgáltatás kezelőt.

Connector

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.
TCP esetén ez azt jelenti, hogy a connector megpróbál kapcsolódni a szerverhez (connect). Amikor ez sikerül létrejön egy összekapcsolt socket, amelyen keresztül elérhető a másik fél.
Amikor létrejött a kapcsolat, akkor elindítja a szolgáltatás kezelőt és átadja a csatorna leíróját.
Mivel a kapcsolat kezdeményezése és a kapcsolódás kezelése elkülönül a mintában, ezért a connector akár aszinkron módon is felépítheti a csatornákat.

Elosztó

Az elosztó feladata, hogy a hálózati kommunikáció eseményeit fogadja és a megfelelő modulokhoz továbbítsa.



Az acceptor-connector minta strukturális felépítése

A folyamat

Passzív mód:

1) Az acceptor (fogadó) inicializációja.
2) Az acceptor várja a kapcsolódásokat.
3) Bejövő kapcsolat esetén az acceptor felépíti a csatornát. Ezt követően létrehoz egy új szolgáltatás kezelőt és csatolja hozzá a csatorna leírót.
4) Az acceptor meghívja a szolgáltatás aktivációs függvényét.
5) A szolgáltatás ezt követően a kapott csatorna leírón keresztül kommunikál a másikalkalmazással.
6) Ha a szolgáltatás befejeződik, akkor minden erőforrás felszabadul.

Aktív mód:

1) A connector (kezdeményező) felépít egy kapcsolatot a másik féllel.
2) A connector létrehoz egy új szolgáltatás kezelőt és csatolja hozzá a csatorna leírót.
3) A connector meghívja a szolgáltatás aktivációs függvényét.
4) Lásd passzív mód 5.
5) Lásd passzív mód 6.

Aszinkron aktív mód:

Megegyezik az aktív móddal, azonban a connector aszinkron módon indítja el a kapcsolat felépítést és regisztrálja magát, hogy megkapja a kapcsolat felépítés végéről az értesítést.

Jellemzői

Előnyök

A minta elválasztja az alkalmazás független kapcsolat felépítést a alkalmazás függő protokoll implementációtól. Így az alkalmazás független részek újra felhasználhatóak más programoknál.
Ugyanakkor a minta lehetővé teszi, hogy a szolgáltatás implementációja ugyanaz maradhasson a kapcsolat felépítésének irányától függetlenül.
Emellett az acceptor-connector minta azt is lehetővé teszi, hogy a kapcsolatok felépítése aszinkron módon történjen, így egyszerre nagy mennyiségű hoszttal tudunk felépíteni kapcsolatot abban az esetben is, ha a hálózati késleltetés nagy. Egy szinkron megoldás ezt nem tenné lehetővé.
A minta továbbfejlesztésével még egy további előnyhöz is juthatunk, amelyre a Unix alapú rendszerek inetd daemonja mutat példát. Mivel a minta használatával a kapcsolat felépítést központosítjuk több szolgáltatás számára, ezért lehetőség nyílik a hozzáférési korlátozások központi konfigurációjára is. Így nem szükséges külön-külön a szolgáltatásokba implementálni azt. Természetesen a kapcsolat felépítésénél csak TCP szintű információk álnak rendelkezésünkre, így a döntést az alapján kell elvégezni. A magasabb szintű azonosítás már a szolgáltatás feladata.

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. Így egy ilyen egyszerű alkalmazást csak feleslegesen bonyolítana.

Implementáció

Implementációs irányelvek

Az acceptor-connector minta implementációját a következő részekre bonthatjuk fel:

- Szükségünk van egy mechanizmusra, amely lehetővé teszi, hogy ha egy kapcsolódási kérés érkezik a passzív végponthoz (szerver socket), akkor a kapcsolat felépülhessen a végpontok között (kliens socket). Az operációs rendszerek rendelkeznek ezen mechanizmus támogatásával, így ezzel többnyire nem kell foglalkoznunk.

- Szükségünk van egy eseménykezelő/szétosztó mechanizmusra is. Egyrészt le kell kezelni a kapcsolódások eseményét passzív végpont esetén, illetve a kapcsolódási próbálkozás eredményét aktív végpont és aszinkron kapcsolódás esetén. Továbbá az acceptor-connector minta csak a kapcsolódást határozza meg. Ezt követően az egyes szolgáltatásoknak még kommunikálnia kell a másik féllel, így a kommunikációs eseményeket is kezelnünk kell. Ezért ehhez a korábban megismert reactiv és proactiv mintákat vagy szálakat használhatunk.

- Definiálnunk kell:
    - A szolgáltatás interfészt
    - Az acceptor interfészt
    - És a connector interfészt

- Természetesen nem elég az interfészek definíciója. Le is kell implementálnunk az acceptor-t és a connector-t. TCP esetén minden jelentősebb operációs rendszer teljes támogatást nyújt ezeknek a megvalósításához.

- Implementálnunk kell a szolgáltatás kezelőket. Ez magában foglalja a szolgáltatás inicializációs rutinjait, az alkalmazási rétegbeli protokoll implementációját és a szolgáltatás algoritmusait. A szolgáltatások párhuzamos futtatásához alkalmazhatjuk a reactor mintát vagy szálakat.

Támogatások az operációs rendszerekben

Minden jelentősebb operációs rendszer tartalmaz szolgáltatásokat, amelyekkel az acceptor-connector minta leimplementálható.
A Unix/Linux operációs rendszereknél az inetd (xinetd, rinetd) daemon ezt a mintát valósítja meg, így használata esetén elegendő már csak a szolgáltatásokat leimplementálnunk. A kapcsolat felépítést az inetd megoldja.

III. Konkurenciakezelési minták

III.1. Bevezetés

Az elmúlt évtizedben a számítógépek feldolgozási sebessége 3-4, míg a hálózati elemek áteresztő képessége 6-7 nagyságrenddel nőtt. Hogy ezt a hatalmas mértékű fejlődést a szoftverek minél jobban ki tudják használni, a szoftverfejlesztésben paradigmaváltásra volt szükség. Ennek folytán az utóbbi években előtérbe kerültek a hálózati alapon működő, elosztott architektúrájú szoftverek.

Az elosztott architektúra rengeteg olyan kérdést vet fel, melyek a hagyományos egyszálú alkalmazásoknál egyáltalán nem merülnek fel, vagy megoldásuk triviális.

- A hálózati *szolgáltatások elérési módja és konfigurációja* olyan feladatokat állít a tervező elé, melyek korrekt megoldására új módszerek szükségesek.
- Az *eseménykezelés* a rendszer elosztott volta miatt szintén új kihívások elé néz.
- Maga a *konkurencia* ténye is jelentősen elbonyolítja egy szoftver kifejlesztését. A konkurencia kezelés olyan új tervezési minták kidolgozását tette szükségessé, melyek leegyszerűsítik, átláthatóvá és hatékonnyá teszik az elosztott szoftverek fejlesztését.
- A megosztott erőforrások elérését egy többszálú alkalmazásban szinkronizálni kell. A *szinkronizáció* implementációjára is léteznek új tervezési minták, melyek megkönnyítik és hatékonyabbá teszik az ilyen feladatok megoldását.

III.2. Az Active Object architekturális minta

Az Active Object tervezési minta lényege, hogy elválasztja az eljáráshívást az eljárás végrehajtásától.

Az Active Object tervezési mintával kapcsolatban leggyakrabban a „pincér az étteremben” hasonlatot szokták emlegetni. A pincérek az étteremben minden vendéghez odamennek és felveszik a rendelésüket, majd továbbítják azt a konyha felé. Ezután folytatják a következő vendéggel, és így tovább. A konyha viszont nem biztos, hogy a rendelések sorrendjében fogja elkészíteni az ételeket, hogy időt és/vagy pénzt takarítson meg. Amíg az étel elkészül, mind a vendégek, mind a pincérek foglalkozhatnak mással: a pincérek kiszolgálnak más vendégeket, a vendégek beszélgethetnek, elintézhetnek egy-egy telefont, stb. Ha kész az étel, a pincér kiviszi őket a vendégeknek.

Probléma

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. Nem fordulhat tehát elő az, hogy az egyik fogyasztó egy termelőre várva blokkolja a központi (kommunikációt biztosító) objektumot, és ezzel a többi termelő/fogyasztó kommunikációját.

Ebben a példában a központi objektum egy megosztott erőforrás, ezért metódusainak elérését szinkronizálnunk kell. Felmerülnek továbbá a következő követelmények:

- Ha a meghívott metódus mögött egy feldolgozás-intenzív feladat áll, akkor nem engedhető meg, hogy a feldolgozás idejére a hívó blokkolódjon.
- A megosztott erőforrás szinkronizált elérését egyszerűen programozhatóvá kell tenni.
- A hardver/szoftver platformban rejlő párhuzamosítási lehetőségeket transzparens módon használjuk ki.

Megoldás

A problémát úgy oldjuk meg, hogy a metódus meghívását és végrehajtását külön processzekre bízzuk.

Az Active Object tervezési minta 6 komponenst definiál.

1) A kliens processzében futó proxy objektum fogadja a kliens kéréseit (metódushívásait)
2) 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.
3) 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.
4) Az aktív objektum száljában futó ütemező (scheduler) feladata, hogy eldöntse, hogy az aktivációs lista mely elemét hajtsa végre legközelebb. Az ütemező különböző logikák alapján dönthet: pl. a kérések listába bekerülési sorrendje alapján, a kérések tulajdonságai alapján, a saját állapota alapján, vagy ezek egy kombinációját alkalmazva.
5) A kiszolgáló (servant) végrehajtja az ütemező által kiválasztott kérést. A kiszolgálóban implementált funkciók mindegyike megfelel a kliensekben futó proxy által kínált metódusok egyikének. A kiszolgáló az aktív objektum része, az ő száljában fut.
6) Amikor a kliens egy hívást kezdeményez a proxyban, kap egy válasz kezelő objektumot (future), amelyen keresztül a kliens megkaphatja a kérés eredményét. Amint a kliensnek szüksége van az eredményre, valamilyen módon randevúznia kell a válasz kezelő objektummal: vagy blokkol vagy pollingol amíg el nem készül az eredmény.

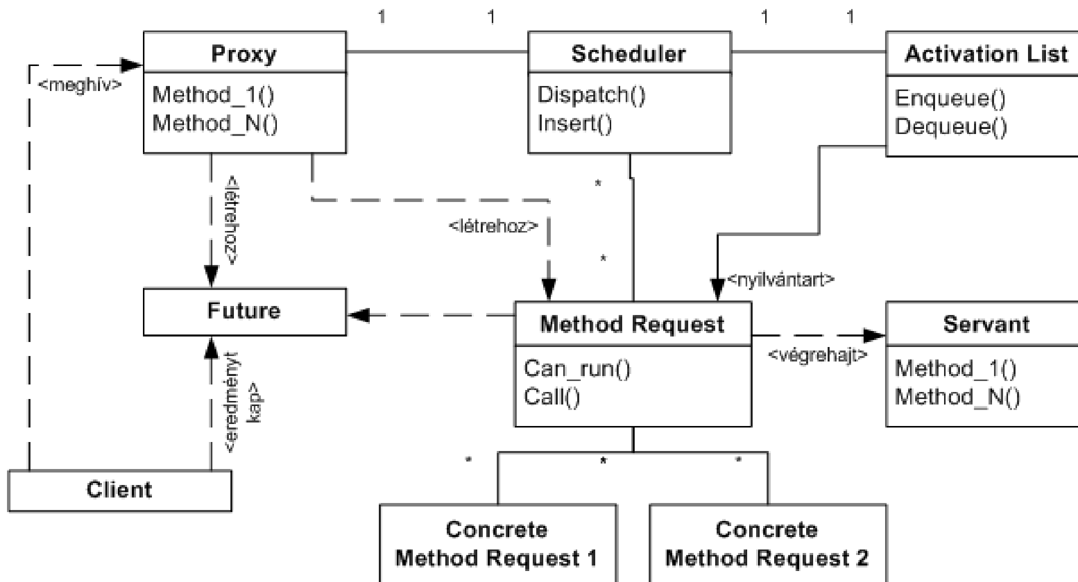
Az alábbi táblázatban összefoglaltam a fenti komponensek tulajdonságait.

Az Active Object minta által definiált komponensek összefoglalása

Objektum	**Leírás**	**Kapcsolódó objektumok**					
		Proxy	**Meth. Req.**	**Act. List**	**Scheduler**	**Servant**	**Future**
Proxy	• Az aktív objektum interfészét definiálja a kliensek felé • Method Request objektumokat hoz létre • A kliens szálban fut		**X**	**X**			**X**
Method Request	• Egy eljáráshívást képvisel az aktív objektumban					**X**	**X**

Objektum	Leírás	Kapcsolódó objektumok					
		Proxy	Meth. Req.	Act. List	Scheduler	Servant	Future
	• Ún. guard eljárásokat biztosít annak megállapítására, mikor válik futtathatóvá egy eljáráshívás						
Concrete Method Request	• Egy konkrét eljáráshívás reprezentációját implementálja • Implementálja az alaposztály guard metódusait					X	X
Activation List	• A végrehajtásra váró (és futtatás alatt lévő) method requesteket tartja nyilván • A method requesteket a proxy helyezi be a listába és a scheduler veszi ki onnan		X				
Scheduler	• Method requesteket szúr be az aktivációs listába • Futtatja az aktív objektum szálját		X	X	X		
Servant	• Implementálja az aktív objektumot • Futtatja az aktív objektum szálját		X				
Future	• Tárolja az eljáráshívás eredményét • Randevú lehetőséget biztosít a kliens számára						

Az alábbi ábrán az Active Object tervezési minta osztálydiagramja látható.



Az Active Object tervezési minta osztálydiagramja

Az Active Object dinamikus működése 3 fázisra bontható:

- Method Request létrehozása és beütemezése
  A kliens meghívja a proxy egyik eljárását. A proxy erre létrehozza a megfelelő Concrete Method Request objektumot, ami eltárolja az eljáráshíváshoz szükséges kontextust. A létrehozott objektumot továbbadja a Schedulernek, aki beszúrja az Activation List-be. A kliens visszakap egy Future objektumot, melyen keresztül majd elérheti az eljáráshívás eredményét.

- A Method Request-nek megfelelő eljárás végrehajtása
  A Scheduler a kliensektől független külön szálon fut. Folyamatosan figyeli az aktivációs listát, és új végrehajtandó kérések esetén eldönti, hogy melyik kérést futtassa először. A végrehajthatóságot a requestek guard metódusán keresztül ellenőrzi. A kiválasztott method requestet kiveszi a listából, és meghívja a Servant megfelelő végrehajtó metódusát.
- Befejezés
  A Servant által elkészített eredményt a Scheduler eltárolja a Future objektumban, és a továbbiakban figyeli tovább az aktivációs listát. A kliens a Future objektumon keresztül elérheti az eredményt.

Előnyök, hátrányok

Az Active Object tervezési minta egyszerűsíti a konkurencia kezelését, transzparens módon oldja meg a párhuzamosítást, valamint külön kezelhetővé teszi az eljáráshívást és – végrehajtást.

Hátrányai közé sorolható a Scheduler implementációjától függő mértékű overhead (kontextus váltás, szinkronizáció, adatmozgatási idő) és a nehezebb debuggolhatóság (a nem determinisztikus működés miatt).

Alkalmazás példa

Ezt a tervezési mintát alkalmazza például a Symbian SDK. Itt egy ún. active scheduler játssza a Scheduler szerepét. A CActive absztrakt alaposztály az alábbi metódusokon keresztül valósítja meg a tervezési mintát:

- SetActive(): jelzi a scheduler felé, hogy egy új request érkezett
- RunL(): Ezen virtuális metódus implementálásával valósíthatjuk meg a Servant-ot
- DoCancel(): Ezen virtuális metódus implementálásával szabhatjuk meg, mi történjen, ha megszakítják az aktív objektum működését.

A Symbian SDK-ban nem feleltethető meg minden fent tárgyalt objektumnak egy-egy Symbian SDK-beli objektum, a Symbian aktív objektumok filozófiája mégis megegyezik a fent leírtakkal.

Példa Symbian Active Objectre

```
class CExampleActiveObject : public CActive {
public:
   CExampleActiveObject(CExampleServiceProvider* aServiceProvider);
   ~CExampleActiveObject;
   void IssueRequest(CMethodRequest);
private:
   void DoCancel();
   void RunL();
   TInt RunError(TInt aError);
private:
   CExampleServiceProvider* iServiceProvider;
   CExampleMethodRequest iRequest;
};



CExampleActiveObject::CExampleActiveObject(CExampleServiceProvider* aServiceProvider) :
CActive(EPriorityStandard) {
```

```
    iServiceProvider = aServiceProvider; // Store pointer to service provider
    CActiveScheduler::Add(this) ; // add to active scheduler
}


void CExampleActiveObject::RunL() {
// implement Servant
}


void CExampleActiveObject::DoCancel() {
// implement Cancel
}
```

A fenti kódrészlet egy CExampleActiveObject nevű aktív objektumot definiál, mely a Symbian SDK által biztosított CActive alaposztályból örököl (a CActive interfészt imlpementálja). A konstruktorban az objektum hozzáadja magát a központi Active Scheduler-hez. Az Active Scheduler a programozó számára nem jelent többletmunkát – ezt a framework biztosítja. Az Activation List viszont nem része a frameworknek, tehát ha a kliens egyszerre egynél több kérést is küldhet, akkor erről gondoskodnunk kell az aktív objektumunk implementációjában (kell bele egy várakozási sor, és a RunL végén meg kell vizsgálni, hogy vár-e még valaki…).

A Servant megvalósításához implementálnunk kell a RunL metódust.

Az IssueRequest() metóduson keresztül tud a kliens kérést kezdeményezni. Itt az IssueRequest argumentumai között kell szerepeltetnünk a minta szerinti Method Request objektumot (vagy az annak megfelelő információt hordozó paramétereket), és ezt tárolni kell egy tagváltozóban is, hogy később a RunL-ben felhasználhassuk. Az IssueRequest() végül meg kell, hogy hívja a CActive alaposztály SetActive() metódusát, hogy jelezze a Scheduler felé az új kérés érkezését. A Future objektum nincs beépítve a CActive filozófiájába – erről a programozónak kell gondoskodni.

Látható, hogy az itt bemutatott példa másképp valósítja meg az Active Object tervezési mintát, mint ahogyan azt a 0 pontban szemléltettem. A Symbian Active Object mégis alkalmas ezen tervezési minta bemutatására, hiszen a lényeget ez is megvalósítja: elválasztja (külön szálakra bontja) az eljárás hívást és az eljárás végrehajtást.

III.2. A Monitor Object architekturális minta

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 Monitor Object tervezési mintával kapcsolatban gyakran hozzák fel egy terhelt gyorsétterem példáját. A vásárlók (kliensek) egyenként mennek a pénztárhoz és leadják rendeléseiket. Egyszerre mindig csak egy vásárló beszél a pénztárossal. Ha egy vásárló

rendelése nem teljesíthető azonnal, akkor félreáll (vagy leül egy asztalhoz) amíg a pénztáros nem szól neki, hogy elkészült az általa kért étel.

Probléma

Tekintsük a 0 pontban ismertetett helyzetet! Termelő és fogyasztó folyamatok kommunikálnak egymással egy központi objektumon keresztül. A központi objektum tartalmaz egy üzeneteket tartalmazó várakozási sort, amelynek van egy enqueue() és egy dequeue() metódusa egy-egy üzenetnek a várakozási sorba való beszúrására, ill. kivételére. Fontos, hogy ezek a metódusok szinkronizáltak legyenek, hogy a várakozási sor mindig konzisztens állapotban maradjon. Nem engedhetjük meg, hogy pl. két termelő folyamat által hívott enqueue() metódus végrehajtása átlapolódjon, mert az implementációtól függően előfordulhat, hogy a második enqueue() hívás által beszúrt üzenet felülírja az előzőt, így az egyik üzenet elveszik.

Ebben a problémakörben a következő követelményeket fogalmazhatjuk meg:

- Az objektum orientált szemlélet megköveteli, hogy az objektumokat csakis az interfész metódusaikon keresztül érhessük el. Ezt könnyen kiterjeszthetjük, hogy kivédjük a versenyhelyzet okozta negatív hatásokat. Az interfész metódusoknak biztosítani kell, hogy egy időben csak egy metódust lehessen futtatni.
- Hogy egyszerűsítsük a többszálú alkalmazások fejlesztését, oldjuk meg, hogy ne a klienseknek kelljen szinkronizálni a megosztott objektumhoz való hozzáférést, hanem ezt a feladatot végezze el maga a megosztott objektum.
- Ha egy metódus meghívása a kliensben blokkolást eredményez, akkor oldjuk meg, hogy a kliens önként lemondhasson a metódus meghívásáról. Ennek főként a deadlock-ok elkerülésében van szerepe.
- Ha egy metódus futása az előbbi módon megszakad, akkor is konzisztens állapotban kell hagynia az objektum állapotát.

Megoldás

A probléma megoldásának lényege, hogy a megosztott objektum metódusait egyszerre csak egy kliens szál futtathatja.

A megosztott objektumot definiáljuk *monitor objektum*ként. A montior objektumot a kliensek csak a *szinkronizált interfész metódus*ain keresztül érhetik el. A versenyhelyzetet elkerülendő egy időben csak egy szinkronizált interfész metódus futását engedélyezzük. A szinkronizációt a monitor objektum egy ún. *monitor lock*kal oldja meg. A szinkronizált metódusok annak eldöntésére, hogy mikor szakítsák meg, ill. folytassák futásukat, ún. *monitor feltételek*et értékelnek ki.

Ennek megfelelően a Monitor Object tervezési minta 4 komponenst definiál:

1) A monitor objektum egy vagy több interfész metódust exportál. Minden kliens ezen metódusokon keresztül érheti csak el a monitor objektumot. A metódusok a kliens threadjében futnak, mivel a monitor objektumnak nincs külön threadje.
2) A szinkronizált metódusok egy-egy exportált metódust implementálnak thread-safe módon. Egyszerre csak egy ilyen metódus futhat egy monitor objektumon belül.
3) Minden monitor objektum tartalmaz egy monitor lockot. A szinkorinzált 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.
4) A monitor feltételek egy monitor objektumon belül azt határozzák meg, hogy egy metódus mely esetekben függessze fel a futását (engedjen előre másokat) és mely

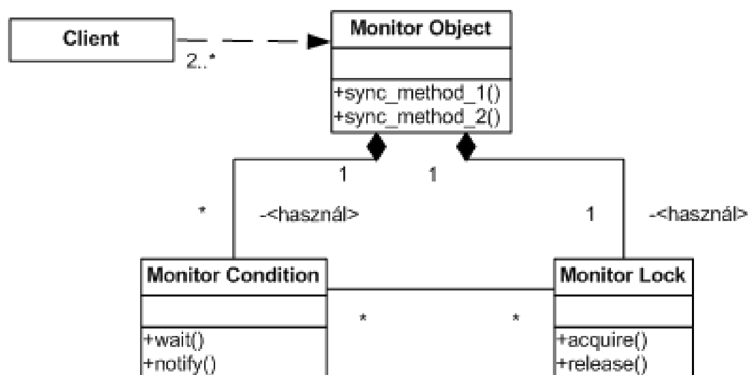
esetekben folytassa működését. Ezzel lehetővé tesszük, hogy a metódusok végrehajtása ne feltétlenül a kliensek „érkezési” sorrendjében történjen.

Az alábbi táblázatban összefoglaltam a fenti komponensek tulajdonságait.

A Monitor Object tervezési minta által definiált komponensek összefoglalása

Objektum	**Leírás**	**Kapcsolódó obj.**			
		Mon. Object	**Sync. Method**	**Mon. Lock**	**Mon. Cond.**
Monitor Object	• A konkurens módon elért objektum		**X**		
Synchronized Method	• A monitor objektum egy publikusan elérhető metódusát implementálja			**X**	**X**
Monitor Lock	• A kölcsönös kizárást valósítja meg az objektumon belül				
Monitor Condition	• Lehetővé teszi, hogy a szinkronizált metódusok megállapítsák, mikor kell felfüggeszteni/folytatni a futásukat			**X**	

Az alábbi ábrán a Monitor Object tervezési minta osztálydiagramja látható.



A Monitor Object tervezési minta osztálydiagramja

A Monitor Object dinamikus működése a következőképpen néz ki:

- Szinkronizált metódus meghívása és sorosítása
  A T_1 kliens thread meghívja a monitor objektum egy szinkronizált metódusát. A metódus először megpróbálja megszerezni a monitor objektumhoz tartozó lockot. Ha egy T_2 thread már megszerezte ez előtt a lockot, akkor annak T_2 által történő elengedéséig a T_1 thread blokkolódik.
- A szinkronizált metódust futtató szál felfüggesztése
  Ha a szinkronizált metódus blokkolódik, várhat valamelyik monitor feltételének teljesülésére (wait). Ez a monitor objektumból való ideiglenes kilépést jelent, megengedve, hogy más threadek futtassák a szinkronizált metódust.
- A monitor feltétel jelzése
  A szinkronizált metódus jelezni tudja egy feltétel teljesülését az összes olyan thread felé, ami előtte felfüggesztette saját futását. Ezek ennek hatására felébrednek, és valamelyik közülük megszerzi a lockot, és futtathatja a megfelelő szinkronizált metódust.

- A szinkronizált metódust futtató szál folytatása
  Ha egy szálnak jelzik a monitor feltétel teljesülését, akkor ott folytatja futását, ahol abbahagyta. A felébresztést az operációs rendszer ütemezője végzi. Ekkor a lock megszerzésének folyamata újraindul, és ennek sikerességétől függően a thread újra felfüggesztődik vagy elkezdheti futtatni a szinkronizált metódus lényegi részét.

A Monitor Object minta egy variációjának tekinthető az *időzített szinkron eljáráshívás*. Itt a kliens threadek megadhatják, hogy mennyi ideig hajlandóak várni egy eljáráshoz tartozó lock megszerzésére.

A Stratégiai lockolás tervezési minta segítségével a Monitor Object minta hatékonyabbá, robosztusabbá és újrafelhasználhatóvá tehető pl. többféle lock- és monitor feltétel típus bevezetésével.

Előnyök, hátrányok
Az a filozófia, hogy egy metódust egyszerre csak egyvalaki futtathat, 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.

A monitor lock viszont központi szerepe miatt korlátozhatja az alkalmazás skálázhatóságát. A monitor objektum továbbá egymásba ágyazva tartalmazza a szinkronizációt és a funkcionalitást biztosító kódot, ami növeli az objektum kódjának bonyolultságát. Problémákba ütközünk akkor is, ha örökölni akarunk egy monitor objektumból, és az örökölt osztálynak másfajta lockolási stratégiát kellene megvalósítani, mint amilyet az alaposztály használ. Ezt a problémát más minták (stratégiai lockolás, thread-safe interface) beépítésével lehet enyhíteni.

Alkalmazás példa
Az alábbi perl kódrészlet egy ütemező várakozási sorát valósítja meg. Az ütemező a sor elején álló folyamatnak prioritásának megfelelő időszeletet biztosít. Az időszelet lejártakor a soron következő folyamat egy jelzést küld a futó folyamatnak, mire az várakozó státuszba kerül, és a soron következő folyamat futhat. Ha egy folyamat a rá szabott időszelet vége előtt befejeződik, akkor küld egy jelzést a várakozási sorban utána következő folyamatnak, ezzel felébreszti őt a várakozó státuszból.

A monitor lock szerepét egy file valósítja meg, melyet *flock* hívásokkal lockolunk. Ebben a file-ban tároljuk az élő (futó vagy várakozó) folyamatok processz azonosítóit.

Az alábbi procedúrák közül a kliens a PushQueue(), RemoveFromQueue() és a CanRun() metódusokat hívhatja.

- A PushQueue() regisztrálja a klienst (beszúrja a processz azonosítóját a sor végére).
- A RemoveFromQueue() törli a klienst a sorból és jelez a következő kliensnek, hogy ő következhet.
- A CanRun() metódus megvizsgálja, hogy a hívó várakozó kliensnek kell-e még tovább várakoznia, és egyben megvalósítja a prioritásarányos időszelet kiosztást.

Példa a Monitor Object minta alkalmazására (Perl)

```
package Queue;
```

```
# ReadQueue: a várakozási sor felolvasása file-ból
sub ReadQueue {
  my $mode=shift;

  my $arr=[];
  return  $arr if (! -f $ProcessQueue);
  my $VAR1;
  if($mode !~ /no_lock/ || ! -r $FILE) {
    open $FILE,"+<$ProcessQueue";      # a várakozási sor megnyitása
    flock ($FILE,LOCK_EX);            # lock megszerzése
  }
  undef $/;
  seek $FILE,0,0;
  my $Queue=eval(<$FILE>);
  if($mode !~ /no_unlock/) {
    close $FILE;
    flock $FILE,LOCK_UN if $mode !~ /no_unlock/;  # lock elengedése
  }
  return $Queue;
}


# WriteQueue: a várakozási sor visszaírása file-ba
sub WriteQueue {
  my $Queue=shift;
  my $elem=shift;
  my $mode=shift;

  push @$Queue,$elem if $elem;
  if($mode !~ /no_lock/ || ! -w $FILE) {
    open $FILE,">$ProcessQueue";
    flock ($FILE,LOCK_EX);        # lock megszerzése
  }
  truncate $FILE,0;
  seek $FILE,0,0;
  print $FILE Dumper($Queue);
  if($mode !~ /no_unlock/) {
    close $FILE;
    flock($FILE,LOCK_UN);         # lock elengedése
  }
  return $Queue;
}


# RemoveFromQueue: egy folyamat végeztekor törli magát a
# várakozási sorból, és felébreszti az utána következőt
sub RemoveFromQueue{
  # A várakozási sor felolvasása a lock elengedése nélkül
  my $Queue=ReadQueue('no_unlock');
  my @q = grep ($_!=$$, @$Queue);
  # A várakozási sor visszaírása a lock megszerzése nélkül
  WriteQueue(\@q,'','no_lock');
  if(@q) {
    kill USR2 => $q[0];   # a soron következő folyamat felébresztése
  }
}


# CanRun: a kliens hívja meg a célból, hogy megállapítsa futhat-e
```

```
# ha az eredmény hamis, akkor újra meg kell hívnia ezt a metódust
# később
sub CanRun {
  my $queue      = shift;
  my $prioarr    = shift;
  my $queueindex = shift;


  if ( $queueindex == 0 ) {         # a sor elején van a kliens
    return 1;         # futhatunk, mert mi vagyunk a sor elején
  } elsif ( $queueindex == 1 ) {
    # mi vagyunk a második helyen->befolyásolhatjuk az előttünk lévőt
    my $firstpid = $queue->[0];
    # kivárjuk az előttünk lévő processz időszeletének végét
    select(undef, undef, undef, $prioarr->[0]/5.0);
    my $queue2 = &MinimalizeQueue;  # a sor újraolvasása
    if ( $queue2->[0] == $firstpid ) {   # Ha még mindig ő van elöl
      push(@$queue2,shift(@$queue2));    # Menjen a sor végére
      &WriteQueue($queue2);             # sor visszaírása
      # Jelezzük az előttünk lévőnek, h. elvettük tőle a futási jogot
      kill USR1 => $firstpid;
      # Letelt az elöl lévő processz ideje, most mi következünk
      return 1;
    } else {     # időközben az előttünk lévő proc. terminálódott
      return 1;  # Közben magától terminált -> jöhetünk mi
    }
  } else {
    # Nem futhatunk, mert nem mi vagyunk a sor elején
    # és a második helyen sem
    select(undef, undef, undef, $prioarr->[0]/5.0); # sleep
    return 0;
  }
}

# PushQueue: A hívó kliens beszúrása a várakozási sorba
sub PushQueue {
  &WriteQueue(&ReadQueue('no_unlock'),$$,'no_lock');
}
```

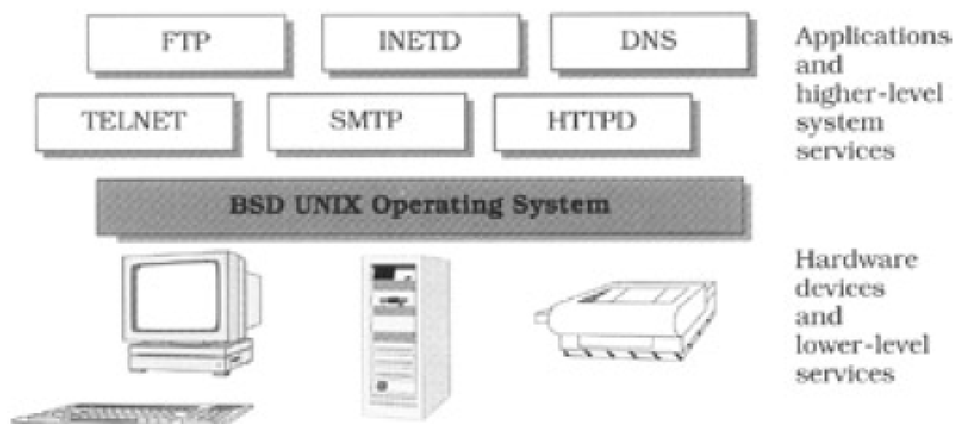
III.3. Half-Sync/Half-Async

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.

Példa

A teljesítmény érzékeny konkurens alkalmazások, mint a telekommunikációs kapcsolórendszerek vagy repülőgépes irányítórendszerek, a szinkron és aszinkron feldolgozás egy keverékét végzik hogy különböző típusú alkalmazásokat, rendszerszolgáltatásokat vagy hardvert koordináljanak. Hasonló jellemzőkkel rendelkeznek a rendszer-szintű szoftverek, mint az operációs rendszerek.
A BSD UNIC operációs rendszer egy példa egy konkurens rendszerre, mely a kommunikációt koordinálja szabvány internetes alkalmazás szolgáltatások között, mint az FTP, INETD, DNS,

TELNET, SMTP és HTTPD, és hardveres I/O eszközök, mint a hálózati interfészek, lemezvezérlők, végfelhasználói terminálok és nyomtatók.



A BSD UNIX operációs rendszer bizonyos szolgáltatásokat aszinkron dolgoz fel, hogy a teljesítményt maximalizálja. A protokollfeldolgozoás a BSD UNIX kernelben például aszinkron fut, mert az I/O eszközök interruptok által működnek melyeket a hálózati interfész hardver hoz létre. Ha a kernel ezeket az aszonkron interruptokat nem kezeli azonnal, a hardver eszközök hibásan működhetnek, eldobnak csomagokat, vagy memóriabuffereket rontanak el.

Bár a BSD operációs rendszer magját aszinkron interruptok működtetik, nehéz alkalmazásokat és magaszintű rendszerszolgáltatásokat fejleszteni aszinkron mechanizmusok használatával, mint például a interruptok vagy signalok. Különösen az aszinkron programok implementálásához, validálásához, hibakereséséhez és karbantartásához szükséges erőfeszítés túl magas lehet. Például az aszinkrónia rejtett időzítési problémákat és versenyfeltételeket okozhat, amikor egy interrupt váratlanul megszakít egy futó számítást.

Hogy az aszinkron programozás bonyolultságát elkerülje, a magasszintű BSD UNIX szolgáltatások szinkron futtatnak több folyamatot. Például az FTP vagy TELNET internetes szolgáltatások, melyek a read() és write() rendszerhívásokat használják blokkolhatják a várakozó I/O tevékenységek befejeződését. Az I/O blokkolás cserébe lehetővé teszi a fejlesztőknek, hogy állapotinformációkat és végrehajtási történetet implicit megtartsanak a szálaik futási-idejű stackjében, mint külön adatstruktúrában melyet a fejlesztőknek kell explicit menedzselniük.

Egy operációs rendszer kontextusában, mindazonáltal, a szinkron és aszinkron feldolgozás nem teljesen független. Különösen az alkalmazási-szintű internet szolgáltatások, melyeket szinkron hajtanak végre a BSD UNIX-ban, együtt kell dolgozniuk a kernel-szintű protokollfeldolgozással, melyek aszinkron futnak. Például egy http szerver által meghívott szinkron read() rendszerhívás indirekt együttműködik az ethernet hálózati interfészen érkező adatok fogadásával és protokollfeldolgozásával.

A kulcs kihívás a BSD UNIX alatti fejlesztésben az aszinkron és szinkron folyamatok strukturálása, hogy mint a programozási egyszerűséget, mind a rendszerteljesítményt növelje. Különösen a szinkron alkalmazások fejlesztőit kell az aszinkron programozás komplex részleteitől megvédeni. Mégis, az rendszer teljesítményének összességében nem szabad romlania nem hatékony szinkron feldolgozási mechanizmusok használata miatt a BSD UNIX magban.

Kontextus

Konkurens rendszerek, melyek mint szinkron és aszinkron feldolgozást végző szolgáltatásokat futtatnak melyek egymással kommunikálnak.

Probléma

A konkurens rendszerek gyakran tartalmaznak szinkron és aszinkron feldolgozó szolgáltatások egy keverékét. Nagy az ösztönzés a rendszerprogramozóknak, hogy aszinkróniát használjanak a rendszer teljesítményének növekedéséhez. Az aszinkron programozás általánosságban hatékonyabb, mert a szolgáltatásokat direktben aszinkron mechanizmusokra lehet leképezni, mint például a hardveres interrupt vagy szoftveres signal kezelőkre.

Viszont a felhasználói programozók oldaláról meg nagy az ösztönzés, hogy szinkron feldolgozást használjanak, hogy a programozási folyamat egyszerűsítésére. A szinkron programok általában egyszerűbbek, mivel bizonyos szolgáltatásokat kényszeríteni lehet, hogy jól-definiált pontokon mozogjanak a feldolgozási szekvenciában.

Az alábbi két erőt kell feloldani szoftver architektúrák specifikálásakor, melyek egyszerre hajtanak végre szinkron és aszinkron szolgáltatásokat:

- Az architektúrát úgy kell megtervezni, hogy az alkalmazásfejlesztők, akik a szinkron feldolgozás egyszerűségét akarják, ne kelljen az aszinkrónia bonyodalmaival törődniük. Hasonlóan, rendszerfejlesztők, akik a teljesítményt akarják maximalizálni, ne kelljen a szinkron feldolgozás nem hatékony elemeit használniuk.
- Az architektúrának lehetővé kell tennie, hogy a szinkron és aszinkron szolgáltatások egymással kommunikáljanak anélkül, hogy a programozási modelljüket bonyolítanánk vagy indokolatlanul lerontsuk a teljesítményüket.

Bár az igény a programozási egyszerűségre és jó teljesítményre ellentmondásnak tűnik, lényeges, hogy ezeket az erőket feloldjuk a konkurens rendszerek egy típusában, kifejezetten a nagyléptékű vagy komplexek esetében.

Megoldás

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.

Részletesebben: A magas szintű szolgáltatásokat, mint a hosszútávú adatbázislekérdezések vagy fájlátvitelek, szinkron dolgozzuk fel külön szálakban vagy folyamatokban, hogy egyszerűsítsük a konkurens programozást. Viszont az alacsonyszintű szolgáltatásokat, mint hálózati interfészről jövö interruptok által vezérelt rövid-életű protokoll kiszolgálók, aszinkron dolgozzuk fel, hogy fejlesszük a teljesítményüket. Ha a különböző rétegekben elhelyezkedő szinkron és aszinkron szolgáltatásoknak kommunikálniuk kell vagy feldolgozásukat szinkronizálni, a *sorbanálló* réteg segítségével küldhetnek egymásnak üzeneteket.

Struktúra

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 internet szolgáltatások az operációs rendszer példánkban külön alkalmazási folyamatokban futnak. Ezek a folyamatok hívják a read() és write() függvényeket hogy szinkron I/O-t végezzenek az internetes szolgáltatásuk kérésére.

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.
Az I/O eszközök és protokollok feldolgozása a BSD UNIX operációs rendszer kernelben aszinkron interrupt kezelőkkel történik. Ezek a kezelők befejezésig futnak, ami azt jelenti, hogy nem blokkolnak vagy szinkronizálják a futásukat más szálakkal, amíg be nem fejeződnek.

Osztály Szinkron szolgáltatás réteg	**Együtt működő** Sorbanállás réteg
Felelősség Végrehajt szinkron magas-szintű feldolgozó szolgáltatásokat	

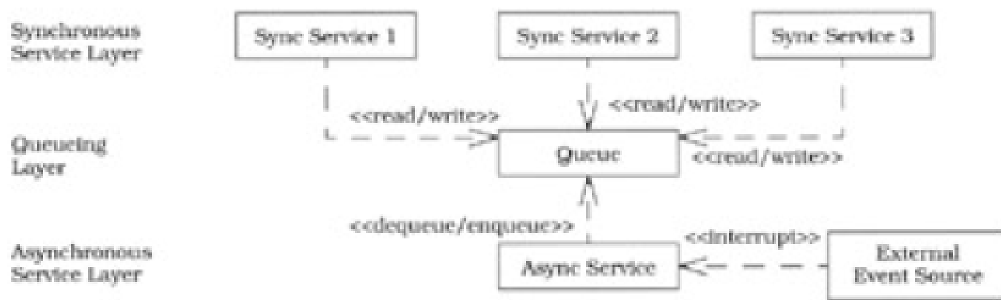
Osztály Aszinkron szolgáltatás réteg	**Együttműködő** Sorbanállás réteg, Külső eseményforrás
Felelősség Végrehajt aszinkron alacsony-szintű feldolgozó szolgáltatásokat	

A sorbanállási réteg biztosítja a szinkron és aszinkron rétegek közti kommunikációs mechanizmust. Például az adat és irányítási információt tartalmazó üzeneteket az aszinkron szolgáltatások létrehozzák, majd bufferelik a sorbanállási rétegben, hogy a szinkron szolgáltatások kinyerjék onnan, és fordítva. A sorbanállási réteg felelős egy réteg szolgáltatásait értesíteni, amikor üzenetet küldtek nekik egy másik rétegből. A sorbanállási réteg tehát lehetővé teszi az aszinkron és szinkon rétegenek, hogy „termelő/fogyasztó” mintájára beszéljenek egymással, hasonlóan a Pipes and Filters mintánál használt struktúrához.
A BSD UNIX operációs rendszer biztosít egy Socket réteget. Ez a réteg szolgál egy bufferelő és értesítő pontként a szinkron internet szolgáltatás alkalmazási folyamatok és az aszinkron interrupt-vezérelt I/O hardver szolgáltatások között a BSD UNIX rendszermagban.
A külső eseményforrások hoznak létre eseményeket, melyeket az aszinkron szolgáltatási réteg kap meg és dolgozik fel. Az operációs rendszerekben a külső események gyakori forrásai lehetnek a hálózati interfészek, lemezvezérlők és végfelhasználói terminálok.

Osztály Sorbanállási réteg	**Együtt működő** Szinkron szolgáltatás réteg, Aszinkron szolgáltatás réteg
Felelősség Bufferelést biztosít a szinkron szolgáltatás réteg és az aszinkron szolgáltatás réteg között	

Osztály Külső esemény forrás	**Együttműködő** Aszinkron szolgáltatási réteg
Felelősség Az aszinkron szolgáltatás réteg által elfogadott és feldolgozott eseményeket generál	

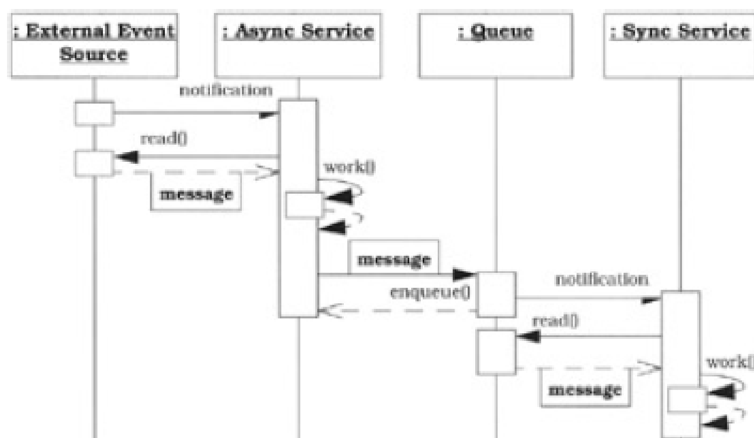
Az alábbi osztálydiagram mutatja az említett résztvevők struktúráját és a köztük fennálló kapcsolatokat.



Dinamika

A Half-Sync/Half-Async minta szinkron és aszinkron rétegei üzenettovábbítással kommunikálnak egymással a sorbanállási rétegen keresztül. Az interakciók három fázisát írjuk le, mely akkor történik, ha 'lentről-fel' input esemény érkezik a külső eseményforrásoktól.

- Aszinkron fázis. Ebben a fázisban az input külső forrásai az aszinkron réteggel lépnek kapcsolatba egy aszinkron esemény jelentés segítségével, mint például egy interrupt vagy signal. Ha az aszinkron szolgáltatások befejezték az input feldolgozását, kommunikálhatnak eredményeikről a kijelölt szolgáltatásoknak a szinkron rétegben a sorbanállási rétegen keresztül.
- Sorbanállási fázis. Ebben a fázisban a sorbanállási réteg buffereli az aszinkron rétegből jövő inputot a szinkron rétegnek és jelzi a szinkron rétegnek, hogy az input elérhető.
- Szinkron fázis. Ebben a fázisban a megfelelő szolgáltatások a szinkron rétegben begyűjtik és feldolgozzák az inputot melyet az aszinkron rétegbeli szolgáltatások helyeztek a sorbanállási rétegbe.



A rétegek közti interakció és a minta résztvevői megfordíthatóak hogy egy 'fentről le' szekvenciát alakítsanak amikor a kimenet a szinkron rétegben futó szolgáltatásokból érkezik.

Implementáció

Ez a fejezet írja le a szükséges tevékenységeket melyekkel implementálható a Half-Sync/Half-Async minta és alkalmazható magas-szintű alkalmazások konkurencia-

architektúrájának strukturálására, mint például webszerverek, adabázis szerverek és alacsonyszintű rendszerek, mint a BSD UNIX operációs rendszer. Ezért bemutatunk néhány példák különböző szakterületről.

1. A rendszer szétbontása összesen három rétegre: szinkron, aszinkron és sorbanállási. Az alábbi három altevékenység segítségével határozható meg, hogy miként bontsuk szét a rendszert a Half-Sync/Half-Async mintának megfelelően.
    1. A magas-szintű és/vagy hosszú-időtartamű szolgáltatások azonosítása és egy szinkron réteggé való konfigurálása. Sok szolgáltatás a rendszerben könnyebben implementálható, ha szinkron feldolgozással programozzák őket. Ezek a szolgáltatások gyakran relatíven magas-szintű vagy hosszú-időtartalmú feldolgozást végeznek, mint például nagyméretű folyamok átvitele egy webkiszolgálón vagy komplex lekérdezés egy adatbázisban. A szinkron réteg szolgáltatásainak ezért külön folyamatban vagy szálban kell futniuk. Ha az adat nem elérhető, a szolgáltatások blokkolhatják a válaszokra váró sorbanállási réteget, a peer-to-peet alkalmazási kommunikációs protokollokon keresztül.

        Mindegyik internetes szolgáltatás a BSD UNIX operációs rendszer példában különböző alkalmazási folyamatban fut. Mindegyik alkalmazási folyamat az internetes szolgáltatáshoz kapcsolt protokollal kommunikál a klienseivel. Az I/O műveletek ezen folyamatokban megvalósíthatóak a TCP socketeken való blokkolással majd a BSD UNIX kernel aszinkron I/O folyamatainak befejeződésére való várakozással.
    2. Az alacsony-szintű vagy/és rövid-időtartamú szolgáltatások azonosítása és egy aszinkron réteggé konfigurálása. A rendszer bizonyos szolgáltatásai nem blokkolhatnak hosszabb ideig. Ezek a szolgáltatások tipikusan alacsony szinten vagy rövid-idejű feldolgozást végeznek, melyek külső eseményforrásokkal állnak kapcsolatban, mint például végfelhasználói terminálok vagy interrupt vezérelt hálózati interfészek. A hatékonyság érdekében ezen eseményforrásokat gyorsan kell kezelni, hogy ne blokkolják a szálat, mely kiszolgálja őket. Az ő szolgáltatásaikat aszinkron értesítések vagy interruptoknak kellene indítaniuk, melyek külső forrásokból jönnek, és befejeződésig futnak, ahol a sorbanállási rétegbe tehetnek üzeneteket a végeredményeiket illetően.

        A mi operációs rendszer példánkban az I/O eszközmeghajtók és komunikációs protokollok a BSD UNIX kernelben aszinkron hardver interruptokra reagálva történnek. Minden aszinkron művelet a kernelben befejeződésig fut, adat vagy irányítási információt tartalmaz adatokat téve a Socket rétegbe, ha kommunikálnia kell egy alkalmazási folyamattal melye egy internetes szolgáltatást futtat a szinkron rétegben.
    3. Rétegközti kommunikációs stratégiák azonosítása és konfigurálásuk egy sorbanállási rétegbe. A sorbanállási réteg egy mediator, ami szétbontja a kommunikációt a szinkron és aszinkron rétegekbeli szolgáltatások között. Ezért ezek a szolgáltatások nem érik el egymást közvetlenül, csak a sorbanállási rétegen keresztül. A kommunikációhoz kapcsolódó stratégiákat,

mint például a (de)multiplexálás, bufferelés, üzenetküldés és folyamatirányítás, a sorbanállási réteg hatja végre. Az aszinkron és szinkron rétegek szolgáltatásai használják ezeket a sorbanállási stratégiákat, hogy protokollokat implementáljanak üzenetek küldésére a szinkron és aszinkron réteg között.

A mi BSD UNIX operációs rendszer példánkban a Socket mechanizmus definilja a sorbanállási réteget a szinkron interneteskiszolgáló alkalmazási folyamatok és az aszinkron operációs rendszer kernel között. Mindegyik internetes szolgáltatás egy vagy több Socketet használ, melyek sorok, amit a BSD UNIX tart karban, hogy bufferelje az üzeneteket melyeket az alkalmazási folyamatok között és a TCP/IP protokollstack és hálózati hardvereszközök között váltottak a kernelben.

2. Szolgáltatások implementálása a szinkron réteghez. Magas-szintű és/vagy hosszú időtartalmű szolgáltatások a szinkron rétegben sokszor multi-threading vagy multi-processing segítségével lesznek implementálva. A szálakhaz hasonlítva, a folyamatok több állapotinformációt tartalmaznak de létrehozásuk, szinkronizálásuk, ütemezésük és közöttük levő kommunikáció lebonyolítása nagyobb többletterhelést jelent. Szinkron szolgáltatások implementálása különböző szálakban, folyamatok helyett, ezért egyszerűbb és hatékonyabb alkalmazásokat jelent.

   Multi-threading csökkenti az alkalmazás robosztusságát mivel egy folyamaton belüli szálak nem védettek egymástól. Például egye hibás szál elronthatja a szálak között megosztott adatot a folyamaton belül, ami hibás eredményekhez vezet, leállítja a folyamat működését vagy a folyamat végtelen idejű várakozását okozza. A robosztusság növelése érdekében az alkalmazás-szolgáltatásokat külön folyamatokban lehet implementálni.

   Az internetes szolgáltatások a BSD UNIX példánkban külön folyamatokban lettek implementálva. A konstrukció növeli robosztusságukat és megakadályozza az illetéktelen hozzáférést bizonyos erőforrásokhoz, például más felhasználók által birtokolt fájlokhoz.
3. Szolgáltatások implementálása az aszinkron réteghez. Alacsony szintű és/vagy rövid-időtartamú szolgáltatások az aszinkron rétegben nem mindig rendelkeznek dedikált szálirányítással. Helyette a szálat máshonnan kell kölcsönözniük, mint például az operációs rendszer „idle thread”-je vagy külön interrupt verem. Hogy megfelelő válaszidőt biztosítsanak más rendszerszolgáltatásoknak, mint például a magas-prioritású hardver interruptok, ezeknek a szolgáltatásoknak szinkron kell futniuk és nem blokkolhatnak hosszabb időperiódusokra. Az alábbi két stratégia az aszinkron szolgáltatások futásának indítására használhatóak:
   1. Aszinkron interruptok. Ezt a stratégiát gyakran használják aszinkron szolgáltatások fejlesztéséhez, melyeket direktben a hardverinterruptok triggerelnek külső eseményforrásokból, mint a hálózati interfészek vagy lemezvezérlők. Ebben a stratégiában amikor az esemény történik a külső eseményforráson, az interrupt jelez az eseményhez rendelt kezelőnek, hogy dolgozza fel az eseményt teljesen.

Bonyolult konkurens rendszerekben lehet, hogy szükséges az interrputoknak egy hierarchiáját definiálni, hogy a kevésbé kritikus kezelőket megszakíthassák a magasabb prioritásúak. Ahhoz, megakadályozzuk az interruptkezelőknek, hogy elrontsák a megosztott állapotot amíg mások éppen hozzájuk férnek, az aszinkron réteg által használt adatstruktúrákat védeni kell, például interrupt prioritás emelésével.

A BSD UNIX kernel kétszintű interrupt sémát használ, hogy a hálózati csomagfeldolgozást kezelje. Idő-kritikus folyamatok magas prioritási fokon futnak, a kevéssé kritikus szoftverek alacsony prioritáson. Ez a kétszintű interruptséma megakadályozza a szoftver protokoll feldolgozás overheadjét, hogy késleltesse a magas-prioritású hardverinterruptokat.

2. Proaktív I/O. Ezt a stratégiát gyakran használják olyan aszinkron szolgáltatások fejlesztésére melyek magaszintű operációs rendszer API-n alapszanak, mint a Windows NT overlapped I/O és I/O completion portok vagy a PSOX aio_* családja az aszinkron rendszerhívásoknak. Ebben a stratégiában az I/O műveletek egy aszinkron művelet processzor által vannak végrehajtva. Amikor az aszinkron művelet befejeződik, az aszinkron művelet processzor generál egy befejeződés eseményt. Ezt az eseményt aztán elküldi az esemény kezelőjéhez, mely feldolgozza az eseményt befejezésig.

   Például a web szerver a proactor mintában mutat be egy alkalmazást, ami proaktív I/O mechanizmust használ, a Windows NT API rendszerhívással. Ez a példa aláhúzza a tényt, hogy az aszinkron feldolgozás és a Half-Sync/Half-Async mintát lehet magasabb szintű alkalmazásokhoz használni, melyek nem érik el a hardver eszközöket direktben.

Mindezek az aszinkron feldolgozó stratégiák osztják a kényszert, hogy a kezelő nem blokkolhat hosszabb időperiódusra anélkül hogy elrontaná a külső eseményforrásokból érkező események feldolgozását.

4. A sorbanállási réteg implementálása. Miután a szolgáltatások az aszinkron rétegben befejezik a külső eseményforrásokból érkező input feldolgozását, tipikusan beteszik az eredményüzeneteket a sorbanállási rétegbe. A megfelelő szolgáltatás a szinkron rétegből ezt követően ki fogja venni ezeket az üzeneteket a sorbanállási rétegből és feldolgozza őket. Ezek a szerepek megfordulnak az output feldolgozásánál. Két kommunikációval kapcsolatos stratégiát kell definiálni amikor implementáljuk a sorbanállási réteget:

   1. A bufferstratégia implementálása. Szolgáltatások az szinkron és szinkron rétegben nem érik el egyás memóriáját direktben – helyette, üzeneteket váltanak a sorbanállási rétegen keresztül. Ez a sorbanállási réteg buffereli az üzeneteket, hogy a szinkron és aszinkron szolgáltatások futhassanak konkurensen, ahelyett, hogy egyfajta 'állj-és-várj' folyamatirányító protokollon keresztül lépésenkénti lockolással fussanak. A bufferelési stratégiának tehát implementálnia kell a sorbarendezést, szerializációt, jelzéseket és a folyamatirányítási stratégiát. Fontos, hogy a Strategy minta alkalmazható, hogy egyszerűsítsük az alternatív stratégiák konfigurációját.

- Sorbarandezési stratégia implementálása. Egyszerű sorbanállási rétegek tárolják az üzeneteket amikor azok megérkeznek, vagyis „first-in, first-out” (FIFO) elven működnek. Az első üzenet, amit a szolgáltatások betettek a sorba az egyik rétegből lesz az első üzenet, amit egy szolgáltatás el fog venni a másik rétegből. A FIFO sorrendezést egyszerű implementálni, de prioritás felcserélődéshez vezethet, ha a magas-prioritású üzeneteket alacsony-prioritású üzenetek után tettek be. Ezért kifinomultabb sorbanállási stratégiák használhatók, hogy az üzeneteket prioritássorrendben nyerjük vissza.
- Szerializációs stratégia implementálása. Az aszinkron és szinkron rétegek szolgáltatásai futhatnak konkurensen. A sort ezért szerializálni kell, hogy a kölcsönös kizárási állapotokat elkerüljük, amikor az üzeneteket konkurensen teszik be és veszik ki. Ezt a szerializációt gyakran egyszerű szinkronizációs mechanizmusokkal implementálják, mint a mutexek. Az ilyen mechanizmusok biztosítják, hogy az üzeneteket be lehet tenni és ki lehet venni a sorbanállási réteg üzenetbufferjéből anélkül, hogy elrontanánk a belső adatstruktúrát.
- Értesítési stratégia implementálása. Szükséges lehet egyik rétegben levő szolgáltatás értesítése, ha üzenetet címeztek neki hogy egy másik rétegből megérkezzen. Az értesítési stratégia, melyet a sorbanállási réteg biztosít, gyakran sokkal kifinomultabb és komolyabb szinkronizációs mechanizmusokkal van implementálva mint a szemaforok vagy állapotváltozók. Ezek a szinkronizációs mechanizmusok képesek értesíteni a szinkron vagy aszinkron rétegbeli megfelelő szolgáltatást amikor adat érkezik hozzájuk a sorbanállási rétegben. A Variációk fejezetben felvázolunk néhány más értesítési stratégiát aszinkron signalokon és interruptokon alapozva.
- Folyamatirányítási stratégia implementálása. A rendszer nem fordíthat végtelen mennyiségű erőforrást az üzenetek bufferelésére a sorbanállási rétegben. Ezért szükséges a szinkron és aszinkron réteg közti átadott adatmennyiség szabályozása. A folyamatirányítás egy technika, mely megakadályozza a szinkron rétegeket, hogy elárasszák az aszinkron rétegeket nagyobb mennyiségben, mint ahogy az üzeneteket továbbítani és sorbaállítani lehet a hálózati interfészeken.
- A szolgáltatások a szinkron rétegben blokkolhatnak. Egy tipikus hibája a folyamatirányítási szabálzatnak, hogy a szinkron szolgáltatásokat alvásba küldi ha bizonyos számnál több üzenetet hoz létre és állít sorba. Miután az aszinkron szolgáltatási réteg kiüríti a sort egy bizonyos szint alá, a sorbanállási réteg felébresztheti a szinkron szolgáltatást, hogy az folytassa a feldolgozást.
- Ezzel ellentétben, az aszinkron szolgáltatások nem blokkolnak. Ha túl sok üzenetet hoznak létre, az általános folyamatirányítási szabályzat engedélyezi a sorbanállási rétegnek, hogy eldobjon üzeneteket, amíg a szinkron szolgáltatások befejezzék a sorukban álló üzeneteik feldolgozását. Ha az üzenetek egy megbízható kapcsolat-központú átviteli protokollal vannak kapcsolatban, mint a TCP, a küldők time-outolnak végül és újraküldik a kidobott üzeneteket.

2. (De)multiplexáló mechanizmus implementálása. A Half-Sync/Half-Async minta egyszerű implementációiban, mint a Leader/Follower minta Példák fejezetében bemutatott OLTP szerverek, csak egy sor van a sorbanállási

rétegben. Ezt a sort az összes szolgáltatás megosztva használja az aszinkron és szinkron rétegben, és bármelyik szolgáltatás feldolgozhat bármilyen kérést. Ez a konfiguráció enyhíti a szükséget egy kifinomult (de)multiplexáló mechanizmus iránt. Ebben az esetben az általános implementáció definiál egy singleton sort, ahova az összes szolgáltatás beilleszt illetve kivesz üzeneteket.
A Half-Sync/Half-Async minta bonyolultabb implementációiban az egy rétegbeli szolgáltatásoknak szüksége lehet hogy bizonyos üzeneteket a másik rétegben bizonyos szolgáltatásoknak küldjenek. A sorbanállási rétegnek tehát több sorra van szükséges, például szolgáltatásonként egyre. Több sorral kifinomultabb demultiplexáló mechanizmusra van szükség, hogy biztosítsuk, hogy a különböző rétegek közti szolgáltatások közt váltott üzenetek a megfelelő sorba kerülnek. Az általános implementáció valamilyen fajta (de)multiplexáló mechanizmust használ, mint például egy hash tábla hogy az üzeneteket a megfelelő sor(ok)ba rakja.
A Monitor Object és Active Objectben definiált Message_Queue komponensek különféle stratégiákat mutatnak ahogy a sorbanállási réteget implementálni lehet:

- A Monitor Object minta biztosítja, hogy egy sorban egyszerre egy metódus hajtódik végre, függetlenül attól, hogy hány szál hívja a sor metódusait konkurensen, úgy, hogy mutexeket és állapotváltozókat használ. A sor futtatja a metódusait a kliens szálakban, vagyis a szálakban, melyek a szinkron és aszinkron szolgáltatásokat futtatják.
- Az Active Object szétválasztja a metódushívásokat a sorban a metódusvégrehajtásoktól. Több szinkron és aszinkron szolgáltatás tehát hívhat metódust a sorban konkurensen. A metódusok külön szálakban futnak a szinkron és aszinkron szolgáltatás szálaitól.

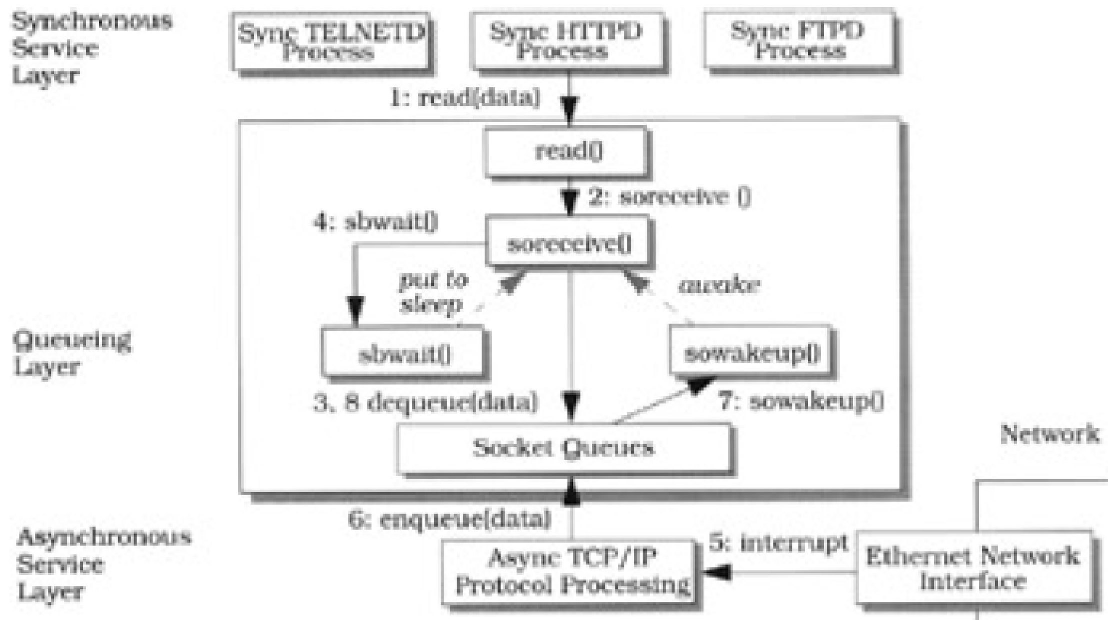
A Lásd Még fejezetei az Active Object és Monitor Object mintáknak megvitatják ezen minták sorbanállási rétegben való használatának az előnyeit és hátrányait.

Megoldott Példa
A különféle minták, mint a Proactor, Scoped Locking, Strategized Locking és Thread-Safe Interface mutatják be a különféle aspektusait a web szerver alkalmazások tervezésének. Ebben a fejezetben egy szélesebb rendszerkontextust fedezünk fel, ahol a webszerverek futhatnak, úgy, hogy felvázoljuk, ahogy BSD UNIX operációs rendszer alkalmazza a Half-Sync/Half-Async mintát hogy egy HTTP GET kérést kap Eternetről a TCP/IP protokollstacken keresztül.
A BSD UNIX egy példája egy olyan operációs rendszernek, amelyik nem támogatja hatékonyan az aszinkron I/O-t. Ezért nem lehetséges egy Web szervert a Proactor minta segítségével implementálni. Helyette, felvázoljuk hogyan koordinálja a szolgáltatásokat és a kommunikációt a BSD UNIX a szinkron alkalmazási folyamatok és az aszinkron operációs rendszer kernel között.
A következőket tárgyaljuk részletesen:

- A read() szinkron rendszerhívás a Web szerver alkalmazás által (a HTTPD folyamat)

- Az aszinkron fogadása és protokollfeldolgozása az Ethernet hálózati interfészről érkező adatnak.
- Szinkon befejezősédese a red() hívásnak, mely visszaadja az irányítást és egy GET kérést a HTTPD folyamatnak.

Ezek a lépések láthatók az alábbi ábrán:



Ahogy az ábrán látható, a HTTPD folyamat meghívja a read() rendszerhívást egy kapcsolódott socketkezelőn, hogy megkapja a HTTP GET kérést a TCP csomagba zárva. A HTTPD folyamat szemszögéből nézve a read() rendszerhívás szinkron, mert a folyamat, aki a read()-et hívja, blokkol amíg a GET kérés adat vissza nem tér. Ha az adat nem azonnal elérhető, a BSD UNIX kernel alvásba küldi a HTTPD folyamatot, amíg az adat meg nem érkezik a hálózatból.

Sok aszinkron lépés történik a szinkron read() rendszerhívás implemenálásához. Bár a HTTPD folyamat alhat amíg az adatra várakozik, a BSD UNIX kernel nem alhat, mivel más alkalmazási folyamatok, mint a FTP vagy TELNET szolgáltatások és I/O eszküzük a kernelben igényelik szolgáltatásait, hogy konkurensen és hatékonyan fussanak.

Miután a read() rendszerhívás megtörtént, az alkalmazási folyamat átvált kernel módba és elkezdi a kiváltságos utasítások futtatását, melyek a szinkronban a BSD UNIX hálózati alrendszerébe irányítják. Végül az alkalmazási folyamat szálirányítása a kernel soreceive() függvényben fejeződik be. Ez a függvény dolgozza fel a különböző típusú socketek bemeneteit, mint a datagram socketek és stream socketek úgy, hogy a socketsorból adatokat küld az alkalmazási folyamatoknak. Az soreceive() függvény tehát definiálja a határt a szinkron alkalmazási folyamat réteg és az aszinkron kernel réteg között a kimenő csomagoknak.

Két út van, ahogy a HTTPD folyamat read() rendszerhívását kezelheti a soreceive(), a Socket és a socketsorban levő adat mennyiségétől függően:

- Teljesen szinkron. Ha a HTTPD processz által igényelt adat a socket sorában van, a soreceive() függvény átmásolja azonnal és a read() rendszerhívás szinkron befejeződik

- Félig szinkron és félig aszinkron. Ha a HTTPD processz által igényelt adat még nem elérhető, a kernel hív egy sbwait() függvényt, mely a folyamatot alvásba teszi, amíg a kért adat meg nem érkezik.

Miután az sbwait() alvásba tette a folamatot, a BSD UNIX ütemezője átvált egy másik folyamat kontextusába, mely kész a futásra. A HTTPD folyamat nézőpontjából a read() rendszerhívás szinkron futásúnak tűnik. Amikor az igényelt adatot tartalmazó csomag(ok) megérkez(nek), a kernel aszinkron feldolgozza őket, ahogy az alábbiakban részletezzük. Amikor elég adatot helyeztek a socket sorába hogy a HTTPD folyamat kérése teljesüljön, a kernel felébreszti ezt a folyamatot és befejezi a read() rendszerhívását. Ez a hívás szinkron visszatér, hogy a HTTPD folyamat elemezze és végrehajtsa a GET kérést.
A BSD UNIX kerneljében a teljesítmény maximalizálása érdekében minden protokollfeldolgozás aszinkron van végrehajtva, mivel az I/O eszközöket hardverinterruptok irányítják. Például a csomagok melyek az Ethernet hálózati interfészről érkeznek a kernelbe lesznek küldve interrupt kezelőkön keresztül, melyet az Ethernet hardver aszinkron indított. Ezek a kezelők különféle eszközöktől kapnak csomagokat és indítják az ezt követő aszinkron feldolgozását a magasabb szintű protokolloknak, mint az IP és TCP. Végül az alkalmazás adatokat tartalmazó érvényes csomagat sorba állítják a Socket rétegben, ahol a BSD UNIX kernel ütemezi és szállítja a várakozó HTTPD folyamatnak, hogy az szinkron elfogyassza ezt az adatot.
Például a HTTPD folyamat read() rendszerhívásához társított ’half-async’ feldolgozás akkor kezdődik, amikor az Ethernet hálózati interfészről érkezik egy csomat, ami triggerel egy aszinkron hardver interruptot. Minden bejövő csomagfeldolgozást az interruptkezelő kontextusában viszik véghez. Egy interrupt alatt, a BSD UNIX kernel nem alhat vagy blokkolhat, mert nincs alkalmazási folyamat kontextus és nincs dedikált szálirányítás. Ezért az Ethernet interruptkezelő ’kölcsönzi’ a kernel szálirányítását. Hasonlóan, a BSD UNIX kernel kölcsönzi az alkalmazás folyamatok szálirányítását, amikor rendszerhívásokat indítanak.
Ha egy csomag egy alkalmazási folyamatnak van célozva, felküldik a szállítási rétegnek, mely további protokollfeldolgozást végez, mint a TCP szegmensek újraösszeállítása és visszaigazolása. Végülis, a szállítási réteg hozzáadja az adatot a fogadási socket sorhoz, és sbwakeup()-ot hív, mely ra bejövő csomagok számára a határt jelenti az aszinkron és szinkron rétegek között. Ez a hívás ébreszti fel a HTTPD folyamatot, mely az soreceive()-ben aludt adatra várva azon a socketsoron. Ha a HTTPD folyamat által igényelt adat megérkezett, az soreceive() bemásolja a HTTPD által biztosított bufferbe, lehetővé téve ezáltal a rendszerhívásnak hogy az irányítást a webszervernek visszaadja. A read() hívás ezért szinkronnak tűnik a HTTPD folyamat perspektívájából, hiába aszinkron feldolgozás és kontextusváltás történt amíg ez a processz aludt.

Változatok

Aszinkron Irányítás Szinkron Adat I/O-val. Az Implementáció fejezetben bemutatott HTTPD Webszerver ’behúzza’ az üzeneteket a sorbanállási rétegből az ő irányába, ezáltal kombinálja az irányító és adat tevékenységeket Néhány operációs rendszer platformon, azonban, lehetséges az irányítást és az adatot úgy szétválasztani, hogy a szolgáltatások a szinkron rétegben *aszinkron* érthesíthetők, ha üzeneteket tettek a sorbanállási rétegbe. A fő haszon ebben a variánsban, hogy a magasabb-szintű ’szinkron’ szolgáltatások reszponzívabbak, mivel aszinkron lehet őket értesíteni.
A UNIX signal-vezértel I/O mechanizmusok implementálják ezt a változatát a Half-Sync/Half-Async mintának. A UNIX kernel használja a SIGIO signált, hogy ’tolja’ az irányítást egy magasabbszintű alkalmazási folyamatnak, amikor az adat megérkezik egyik

socketén. Amikor egy folyamat megkapja az irányítási jelzést aszinkron, behúzhatja szinkron az adatot a socket várakozási rétegéből a read()-el.
Az aszinkron irányítás használatának hátránya, természetesen, hogy a magas szintű szolgáltatás fejlesztőinek szembe kell néznie az aszinkronitás bonyolultságával, ahogy a Problémák fejezetben vázoltuk.
Half-Async/Half-Async. Ez a változat kibővíti az előző változatot az aszinkron irányítási értesítések és adatoperációk propagálásával egészen a magasszintű szolgáltatásokig a szinkron rétegben. Ezek a magasszintű szolgáltatások tehát képesek lesznek az alacsonyszintű aszinkron mechanizmusok hatékonyságának előnyeit kihasználni.
Például a POSIX valósidejű programozási specifiációban definiált signal interfészek támogatják ezt a variánst. Egy bufferpointert lehet átadni a signal kezelő függvénynek, melyet az operációs rendszer küldött, amikor az a valósidejű esemény megtörtént. A Windows NT támogat hasonló mechanizmusokat az overlapped I/O és completion I/O portokon. Ebben az eseteben, amikor az aszinkron művelet befejeződik, az hozzá társított overlapped I/O struktúra jelzi, hogy melyik művelet fejeződött meg és hogy ad-e tovább adatot. A Proactor minta és az Asyncrhronous Completion Token minta írja le, hogyan tervezzünk alkalmazásokat, hogy hasznát vegyék az aszinkron műveleteknek és az overlapped I/O-nak.
A hátránya ennek a változatnak hasonló az előző változatéhoz. Ha a legtöbb vagy minden szolgátatás aszinkron műveletek által van hajtva, a konstrukciót jobban lehet a Proaktor minta használatával modellezni, mint a Half-Sync/Half-Async mintával.
Half-Sync/Half-Sync. Ez a variáns biztosítja az alacsony szintű szolgáltatások szinkron feldolgozását. Ha az aszinkron réteg többszálú, a szolgáltatásai autonóm futnak és használják a sorbanállási réteget hogy üzeneteket küldjenek a szinkron szolgáltatási rétegnek. Ennek a változatnak az előnye, hogy az aszinkron réteg szolgáltatásai egyszerüsíthetőek, mert blokkolhatnak annélkül hogy más szolgáltatásokra hatással lennének ebben a rétegben.
A mikrokerneles operációs rendszerek mint a Mach vagy Amoeba, tipikusan használják ezt a változatot. A mikrokernel mint egy külön multi-thread folyamat fut és üzenetet cserél az alkalmazási folyamatokkal. Hasonlóan a többszáló operációs rendszer makrokernelek mint a Solaris, támogatni tudnak szinkron I/O műveleteket a kernelben.
A kernelt többszálúvá tevése segítségével lehet poll-olt interruptokat implementálni, ami csökkenti a kontextusváltások mennyiségét a magas-teljseítményű folyamatos médiarendszerekben, azzal, hogy egy kernel szálat dedikál hogy a megosztott memória területet vizsgálja rendszeres időközönként. Ellentétben az egyszálú operációs rendszerekkel, mint a BSD UNIX, ahol az alacsonyszintá kernel szolgáltatásokat aszinkron I/O használatra korlátoznak és a szinkron multi-programozást csak magaszintű alkalmazásfolyamatoknak támogatják.
A hátránya a szinkron alacsonyszintű feldolgozás engedélyezésének természetesen az, hogy ez az overheadet növeli, ezzel az általános rendszerteljesítményt jelentősen csökkentve.
Half-Sync/Half-Reactive. Objektumorientált alkalmazásokban a Half-Sync/Half-Async mintát a composite arhitektúrális mintával lehet implementálni, ami kombinálja a Reactor mintát a Thread Pool változatával az Active Object mintának. Ebben a gyakori variánsban a reactor eseménykezelői alakítják az aszinkron réteg szolgáltatásait és a sorbanállási réteget meg az active object aktivációs listájával lehet implementálni. Az active object szálkészletéből az ütemező által elküldött szolgág alkotják a szolgáltatásokat a szinkron rétegben. Ennek a változatnak az előnye az egyszerűsítés amit megenged. Ezt az egyszerűséget úgy éri el, hogy eseményeket demultiplexálja, majd továbbküldi egy egy-szálas reactornak, ami az active object száltartalékában folyó konkurens eseményfeldolgozásról le van választva.
Az OLTP szerverek, amit a Leader/Follower minta Példa fejeztében ismertettünk használja ezt a változatot. Az aszinkron szolgáltatási réteg használja a Reactor mintát hogy demultiplexáljon tranzakciókéréseket több klienstől és továbbítsa eseménykezelőknek. A

kezelők kéréseket küldenek a sorbanállási rétegnek, mely egy aktivációs lista a Monitor Object mintával implementálva. Hasonlóan, a szinkron szolgáltatási réteg használja a thread pool variánsát az Active Object mintának, hogy szétszórja a kéréseket az aktivációs listáról a dolgozó szálak pooljába, mely a kliensektől érkező kéréseket szolgálják ki. Az active object szálkészletének minden szála szinkron blokkolhat mivel mindegyik rendelkezik a saját stackjével.

Ennek a variánsnak a hátránya, hogy a sorbanállási réteg több kontextusváltást, szinkronizációt, adatallokálást és adatmásolást okoz, ami szükségtelen bizonyos alkalmazásoknak. Ilyen esetekben a Leader/Follower minta hatékonyabb, kiszámíthatóbb és skálázhatóbb út lehet hogy a konkurens alkalmazásunkat strukturáljuk, mint a Half-Sync/Half-Async minta.

Ismert használatok

UNIX hálózati alrendszerek. A BSD UNIX hálózati alrendszer és a UNIX STREAMS kommunikációs keretrendszer használja a Half-Sync/Half-Async mintát, hogy rendezze a konkurens alkalmazási folyamatok I/O architektúráját és az operációs rendszer kerneljét. Az I/O ezekben a kernelekben aszinrkon és interruptok által triggerelt. A sorbanállási réteg Socket layer által van BSD UNIX-ban implementálva és a Steam Head-ek segítségével a UNIX STREAMS-ben. Az alkalmazási folyamatok számára az I/O szinkron.

A legtöbb UNIX daemon, mint a TELNETD és FTPD, alkalmazási folyamatokként vannak fejlesztve, melyek a read() és write() rendszerhívást használják szinkron. Ez a szerkezet védi az alkalmazásfejlesztőket az kernel által feldolgozott aszinkron I/O bonyolultságától. Bár vannak hibrid mechanizmusok, mint a UNIX SIGIO signal, amit arra haszálnak, hogy szinkron I/O feldolgozást triggereljen aszinkron irányítási értesítésekből.

CORBA ORB-ok. Az MT-Orbix a Half-Sync/Half-Async minta egy változatát használja hogy CORBA távoli eljárásokat küldjön egy konkurens szerveren. Az MT-Orbix ORB Core-jában egy külön szálat társítanak minden socket kezelőhöz, mely egy klienshez van kapcsolódva. Mindegyik szál szinkron blokkol kiolvasva a CORBA kérést a klienstől. Amikor egy kérés megérkezik, akkor demultiplexálják és beszúrják a sorbanállási rétegbe. Egy active object szál a szinkron rétegben ekkor felébred, kiveszi a sorból a kérést és befejezésig feldolgozza úgy egy felfele irányuló CORBA szolgahívással.

ACE. Az ACE keretrendszer használja a 'Half-Sync/Half-Reactive' változatát a Half-Sync/HAlf-Async mintának egy alkalmazás-szintű gatewayben, mely üzeneteket irányít egy elosztott rendszer tagjai között. Az ACE_Reactor egy ACE implementációja a Reactor mintának, ami demultiplexálja a jelző-eseményeket és társított eseménykezelőket az aszinkron rétegben. Az ACE Message_Queue osztály implementálja a sorbanállási réteget, míg az ACE Task osztály implementálja a szálkészlet változatát az Active Object mintának a szinkron szolgáltatási rétegben.

Conduit. A Conduit kommunikációs keretrendszer implementálja a Choices operációs rendszer projektben az objektumorientált változatát a Half-Sync/Half-Async mintának. Az alkalmazási folyamatok szinkron aktív objektumok, egy Adapter Conduit szolgál mint sorbanállási réteg és a Conduit mikro-kernel aszinkron működik, a hardvereszközökkel interruptokon keresztül kommunikálva.

Éttermek. Sok étterem használja a Half-Sync/Half-Async minta egy változatát. Például éttermek gyakran alkalmaznak kiszolgálókat, akik üdvözlik a vendégeket és figyelik a sorrendet, ahogy majd leültetik őket, ha az étterem nagyon forgalmas és sorban kell őket állítani az asztalra várakozás közben. Ez a kiszolgáló „megosztott" minden vendég között, és nem tölthet sok időt egyetlen csoporttal sem. Miután a vendégek leültek egy asztalhoz, az adott asztalhoz dedikált pincér szolgálja ki őket.

Következmények
A Half-Sync/Half-Async mintának a következő **előnyei** vannak:
Egyszerűsítés és teljesítmény. A magas-szintű feldolgozó szolgáltatások programozása egyszerűsödik anélkül hogy az alacsonyszintű szolgáltatások teljesítményét lerontanánk. A konkurrens rendszerek gyakran nagy számú és változatosságú magaszintű szolgáltatással rendelkeznek mint alacsonyszintűvel. Szétválasztani a magas-szintű szolgáltatásokat az alacsonyszintű aszinkron feldolgozó szolgáltatásoktól egyszerüsíti az alkalmazásprogramozást, mert a komplex konkurenciairányítás, interruptkezelés és időzítési szolgáltatások logalizálhatóak az aszinkron rétegen belül. Az aszinkron réteg ugyancsak kezeli az alacsonyszintű részleteket, melyek nehezek lehetnek az alkalmazásfejlesztőknek, hogy robosztus módon programozzák, mint például az interruptkezelés. Ezen felül az aszinkron réteg kezelni tudja az interakciót a hardverspecifikus komponensekkel mint a DMA, memóriamenedzsment és I/O eszközregiszterek.
Szinkron I/O használata ugyancsak képes a programozás egyszerűsítésére, és növelheti a teljesítményt a többprocesszoros platformokon. Például a hosszútávú adatátvitelek, mint egy nagy orvosi kép letöltése egy hierarchikus tárkezelő rendszerben egyszerüsíthető és hatékonyan végrehajtható szinkron I/O segítségével. Különösen, a egy processzort az adatátvitelt végző szálhoz kapcsolhatunk. Ez lehetővé teszi, hogy az adott CPU utasítás és adatcache-jét hogy a teljes képátvitel művelethez rendeljük.
Problémák széválasztása. A szinkronizációs szabályok minden rétegben szétválasztottak. Nem kell minden rétegnek ugyanazt a konkurencia irányítási stratégiát használnia. Egy egy-száló BSD UNIX kernelben például az aszinkron szolgáltatási réteg implementálj a szinkronizálást alacsonyszintű mechanizmusok segítségével, mint például a CPU interruptszintjeinek növelésével és csökkentésével. Ezzel ellentétben az alkalmazási folyamatok a szinkon kiszolgáló rétegben implementálják a szinkronziálást magasszintű mechanizmusokkal, mint monitor objektumok, vagy szinkronizált üzenetsorok.
Legacy könyvtárak, mint az X Windows vagy öregebb RPC eszközkészletek gyakran nem újrahívhatóak. Több irányítószál nem tudja tehát ezeket a könyvtári függvényeket konkurensen meghívni ezzel versenyhelyzetet okozva. A teljesítmény növelése érdekében, vagy hogy kihasználjuk a több CPU adta előnyöket, szükséges lehet a nagyméretű adatátviteleket vagy adatbázislekérdezéseket külön szálba tenni. Ez esetben a Half-Sync/Half-Reactive változata a Half-Sync/Half-Async mintának alkalamzható, hogy szétbontsa az egyszáló részeket az alkalmazásnak többszálú részekké.
Például egy alkalmazás X Windows GUI feldolgozása futhat egy reactor irányítása alatt. Hasonlóan, a hosszú adatátvitelek futhatnak egy active object száltartalékának irányítása alatt. A szinkronizációs szabályok az alkalmazás minden rétegében való szétválasztásával a Half-Sync/Half-Async minta segítségével, a nem újrahívható függvények is tudnak működni a továbbiakban anélkül, hogy a meglevő kódjukat megváltoztatnánk.
Rétegbeli kommunikáció központosítása. A rétegen belül kommunikáció egy elérési pontba lett központosítva, mivel miden interakciót a sorbanállási réteg közvetít. A sorbanállási réteg bufferelo a két réteg között küldött üzeneteket. Ez eltünteti a zárolás és szinkronizáció bonyolultságait, melyek amúgy szükségesek lennének, ha a szinkron és aszinkron szolgáltatási rétegek objektumokat akarnának elérni direktben egymás memóriaterületén.
A Half-Sync/Half-Async mintának a követekző **hátrányai** vannak:
A határ átlépésének hátrányát vonhatja maga után a kontextusváltás, szinkronizáció és adatmásolási overhead, amikor adatot visznek át a szinkron és az aszinkron réteg között a sorbanállási réteg segítségével. Például a legtöbb operációs rendszer a Half-Sync/Half-Async mintát úgy implementálja, hogy a sorbanállási réteget a felhasználói-szintű és kernel szintű területek védelmi határára teszi. Lényeges teljesítményromlást okozhat ennek a határnak az átlépése.

Ennek az overheadnak a csökkentésének egy lehetősége, hogy megosztanak egy memóriarészt a szinkron szolgáltatási réteg és az aszinkron szolgáltatási réteg között. Ez a ’nincs-másolás’ szerkezet teszi lehetővé a két réteg direkt adatcseréjét anélkül, hogy adatot kéne a sorbanállási rétegbe, vagy abból kimásolni.
Léteznek kiegészítések a BSD UNIX I/O alrendszeréhez, melyek minimalizálják a réteghatár átlépésének hátrányait, hogy pollolt interruptuokat használnak, hogy a folytonos média I/O folyamok kezelését fejlesszék. Ez a megközelítés egy buffermenedzsment rendszer definiál, amely engedélyezi a hatékony lap újra-térképezést és megosztott memória mechanizmusokat, melyet az alkalmazási folyamatok, a kernel és az eszközei között használnak.
A magasszintű alkalmazási szolgáltatások nem feltétlenül nyernek az aszinkron I/O hatékonyságából. Az operációs rendszer vagy az alkalmazási keretrendszer interfészeinek szerkezetétől függően lehetséges, hogy a magaszintű szolgáltatások nem képesek az alacsony szintű I/O eszközöket hatékonyan használni. A BSD UNIX operációs rendszer, például megakadályozza az alkalmazásokat, hogy biztonyos típusú hardvert hatékonyan kezeljenek, még ha ezen I/O külső források támogatják az számítás és kommunikáció aszinkron átlapolását.
Hibakeresés és tesztelés bonyolultsága. A Half-Sync/Half-Async minta használatával írt alkalmazások előidézhetik ugyanazokat a hibakeresési és tesztelési kihívásokat, melyeket a Proactor és Reactor minták Következmények fejezetében ismertettünk.

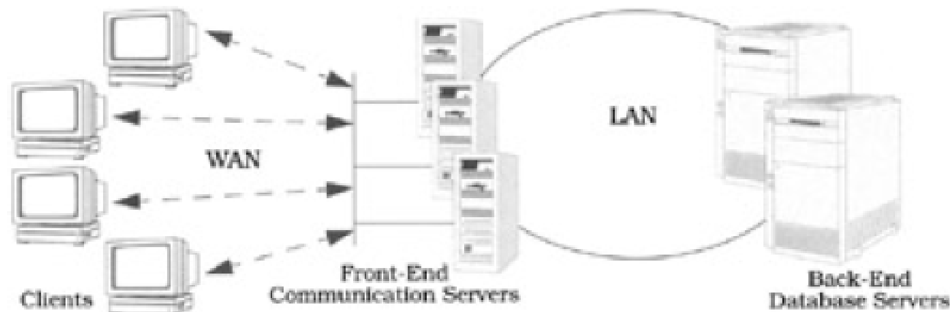
Lásd még
A Proactor mintát úgy is lehet tekinteni, mint a Half-Sync/Half-Async minta egy kibővítését, mely az aszinkron irányítás és adatműveleteket propagálja fel, a magasszintű szolgáltatásoknak. Általánosságban, a Proactor mintát akkor kell használni, ha az operációs rendszer platformja támogatja az aszinkron I/O-t hatékonyan és az alkalmazásfejlesztőket nem zavarja az aszinkron I/O programozási modell.
A reaktor mintát lehet az Active Object mintával együtt használni a Half-Sync/Half-Async minta Half-Sync/Half-Reactive változatának implementálásához. Hasonlóan, a Leader/Follower mintát is lehet használni a Half-Sync/Half-Async helyett ha nincs szükség sorbanállási rétegre az aszinkron és szinkron rétegek között.
A Pipes and Filters minta ír le pár általános elvet a termelő-fogyasztó kommunikáció implementlására egy szoftverrendszer komponensei között. A Half-Sync/Half-Async minta bizonyos konfigurációi ezért a Pipes and Filters minta példányainak tekinthetők, ahol a szűrő egész rétegeket tartalmaz a kifinomultabb szolgáltatásokból. Ezen felül, a szűrő tartalmazhat aktív objektumokat, melyek tartalmazhatnak Half-Sync/Half-Reactive vagy Half-Sync/Half-Sync változatokat.
A rétegek minta írja le az általános elvet, hogy a szolgáltatásokat külön rétegekbe választjuk szét. A Half-Sync/Half-Async minta ezért a Rétegek minta egy specializációjaként is tekinthetjók, aminek a célja a szinkron feldolgozás és aszinkron feldolgozás szétválasztása egy konkurrens rendszerben mindkét típusú szolgáltatás számára egy dedikált réteg bevezetésével.

III.4. Leader/Followers

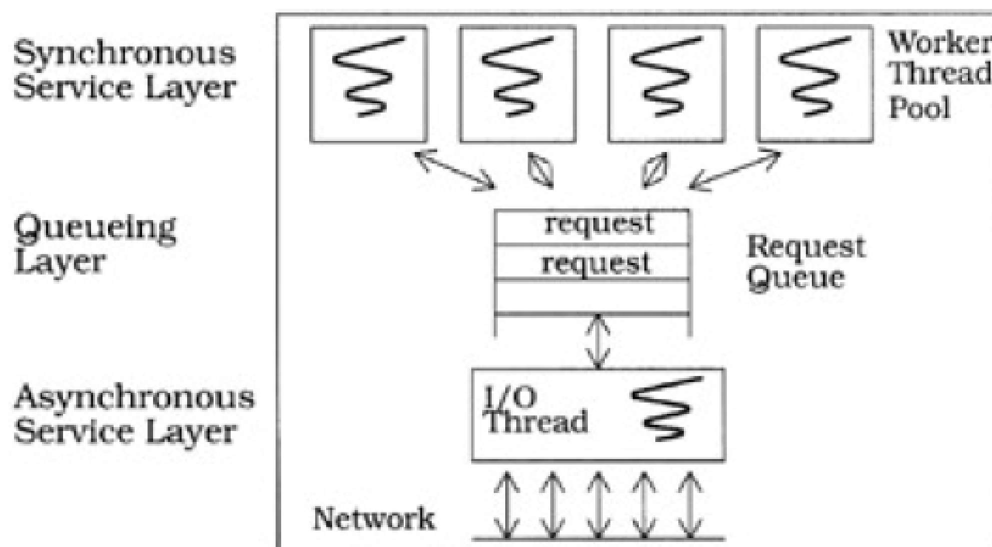
A Leader/Follower architekturális minta egy hatékony konkurenciamodellt biztosít, ahol több szál felváltva megoszt egy esemény forrás halmazt, hogy felderítsen, demultiplexáljon, elküldjön és feldolgozzon szolgáltatás kéréseket, melyek az eseményforrásokon történnek.

Példa

Vegyünk egy többrétegű, nagykapacítású inline tranzakció feldolgozó rendszert (OLTP). Ebben a szerkezetben a frontend kommunikációs szerverek routolják a tranzakciós kéréseket a távoli kliensektől, mint az utazó ügynökök, vagy igényfeldolgozó központok vagy eladási terminálok, a back-end adatbázisszerverekig, melyek feldolgozzák a kéréseket tranzakcióban. Miután a tranzakció befejeződik, az adatbázisszerver visszaadja az eredményeket az összekapcsolt kommunikációs kiszolgálóhoz, mely ezután továbbítja az eredményeket vissza a származtató távoli kliensnek. Ezt a többrétegű architektúra hivatott a rendszer általános áteresztőképességének és biztonságának növelni, terheléselosztás és redundancia segítségével. Ugyancsak tehermentesíti a back-end szervereket a távoli kliensek különféle kommunikációs protokolljainak menedzselésétől.



Egy mód az OLTP szerverek implementálására, hogy egy egyszálú eseményfeldolgozó modellt használunk a Reactor minta alapján. Ez a modell azonban szerializálja az eseményfeldolgozást, mely rontha a rendszer általános teljesítményét, amikor hosszú-futásidejű vagy blokkoló kliens kérés eseményeket kezel. Hasonlóan, az egyszálú szerverek nem hasznosítják transzparens módon a multiprocesszoros platformokat.
Egy gyakori stratégia az OLTP szerverek teljesítményéneknövelésére, hogy többszálú konkurenciás modellt használunk, melyek a különféle kliensek kéréseit és a kapcsolódó válaszok feldolgozását egyidejűleg végzik. Például többszálúvá tehetjük az OLTP backend szervert egy szálkészlet létrehozásával a Half-Sync/Half-Async minta Half-Sync/Half-Reactive mintája alapján. Ebben a felépítésben az ILTP backend szerverek tartalmazzák a dedikált hálózati I/O szálakat, melyek használják a select() esemény demultiplexálóját, hogy várjanak az eseményekre melyek egy socketkezelő halmazon követeznek be melyeket a frontend kommunikációs szerverekhez kapcsoltak.



Amikor tevékenység történik a halmazbeli kezelőkön, a select() visszaadja az irányítást a hálózati I/O szálnak, és jelzi, hogy a halmaz melyik socketkezelőinek vannak várakozó eseményei. Az I/O szálak ezután kiolvassák a tranzakciós kéréseket a socket kezelőkből, eltárolják őket dinamikusan lefoglalt kérésekkés ée beszúrják ezeket a kéréseket a szinkronizált üzenetsorba, melyet a Monitor Object minta használatával implementáltak. EZ az üzenetsort aztán dolgozó szálak egy készlete szolgálja ki. Amikor egy dolgozó szál elérhető a készletből, kivesz egy kérést a sorból, elvégzi a meghatározott tranzakciót, majd visszaad egy választ a front-end kommunikációs szervernek.
Bár a fent leírt szálkezelő modellt sok konkurens alkalmazásban használják, túlzottan nagy overheadet okoz, amikor nagykapacitású szerverekben használják, mint a mi OLTP példánkban. Például egy kisebb terhelés a Half-Sync/Half-Reactive szálkészletben dinamikus memóriafoglalást von maga után, több szinkronizációs operátorral és egy kontextusváltást, hogy a kérésüzenetet átadjunk a hálózati I/O szál és a dolgozó szál között. Ezek az overheadek még a legjobb esetbeli válaszidőt is szükségtelenül magassá teszi. Ezen felül az OLTP backend szerverek többprocesszoron futnak, lényeges méretű overhead jöhet létre a processzorcache-ek koherenciaprotokolljából mely a szálak közti kérésátvitelekhez szükségesek.
Ha az OLTP backend szerverek egy, az aszinkron I/O-t hatékonyan támogató operációs rendszer platformon futnak, a Half-Sync/Half-Reactive szálkészletet le lehet cserélni egy teljesen aszinkron szálkészlettel a Proactor minta alapján. Ez az alternatíva csökkenti a szinkronizáció, kontextusváltás és cachekoherencia overhead nagyrészét, azzal, hogy eltünteti a hálózati I/O szálat. Sajnos sok operációs rendszer nem támogatja az aszinkron I/O-t és amelyek igen, gyakran nem hatékonyan támogatják. Pedig alapvetően fontos, hogy a nagykapacitású OLTP szerverek a a kéréseket hatékonyan demultiplexálják a szálakhoz, hogy azok konkurensen feldolgozzák az eredményeket.

Kontextus

Egy eseményvezérelt alkalmazás, ahol egy eseményforrásra érkező több szolgáltatáskérést hatékonyan kell feldolgozni több szál segítségével, melyek megosztják az eseményforrásokat.

Probléma

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 az OLTP példánkban, melyek egyidejűleg érkeznek. Ahhoz, hogy ezt a problémát hatékonyan kezeljük, három erőt kell feloldani:

- A szolgáltatáskérések több eseményforrásból érkezhetnek, mint például több TCP/IP socketkezelő, melyek mindegyikét egy klienshez osztottak ki. A kulcs tervezési erő ezért, hogy meghatározzuk hatékony demultiplexálási kapcsolatokat a szálak és az eseményforrások között. Különösen, egy szál rendelése minden eseményforráshoz nem megvalósítható az alkalmazások vagy az operációs rendszer és hálózati platformok skálázási korlátozásai miatt.
  A mi OLTP szerveralkalmazásunknak valószínűleg nem praktikus hogy egy külön szálat rendelünk minden socketkezelőnek. Különösen ahogy a kapcsolatok száma

jelentősen növekedik, ez a szerkezet nem skálázható hatékonyan sok operációs rendszer platformon.

- A teljesítmény maximalizálása érdekében a konkurrenciával kapcsolatos overhead kulcs forrásait, mint a kontextusváltás, szinkronizáció és cache koherenciamenedzsment, minimalizálni kell. Különösen az olyan konkurrencia modellek, melyek dinamikusan foglalnak memóriát minden egyes, több szál között átadott kérésnek, növelik az overhead-et lényegesen a konvencionális többprocesszoros operációs rendszereken.
- A mi OLTP rendszerünk implentálása a Half-Sync/Half-Reactive szálkészlet változattal, ahogy a Példa fejezetben körvonalaztuk, dinamikus memóriafoglalást igényel a hálózati I/O szálakban, hogy tárolják a bejövő tranzakciós kéréseket az üzenetsorba. Ez a szerkezet számos szinkronizációs és kontextusváltást von maga után, hogy a kéréseket beszúrjuk vagy kivegyük az üzenetsorból, ahogy a Monitor Object mintánál bemutattuk.
- Több szálnak mely eseményeket demultiplexál egy megosztott eseményforrás, halmazon koordinálnia kell, hogy megakadályozza a versenyhelyzeteket. Versenyhelyzet akkor jön létre, ha több szál próbál elérni vagy módosítani egy bizonyos típusú eseményforrást egyidejűleg. Pédául, egy szálkészlet nem használhatja a select()-et konkurensen hogy demultiplexáljon egy socketkezelő halmazt, mert az operációs rendszer hibásan fog több, a select()-et hívó szálat értesíteni , amikor I/O események várakoznak ugyanazon a socketkezelőn. Ezen felül, a bájtfolyamközpontú protokollok esetén, mint a TCP, ha több szál hív read()-et vagy write()-ot ugyanazon a socketkezelőn, akkor adatot veszt vagy ront el.

Megoldás

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.

Részletesebben: Veszünk egy szálkészlet mechanizmust, mely lehetővé teszi, hogy több szál koordinálja egymást és védre a kritikus fázisokat, míg észlelik, demultiplexálják, elküldik és feldolgozzák az eseményeket. Ebben a mechanizmusban, egy szál egy időpillanatban – a vezető (leader) – várakozik, hogy egy esemény történjen az eseményforrás halmazon. Ez idő alatt a többi szál – a követők (followers) – felsorakozhatnak és kivárják a sorukat, hogy vezetővé váljanak. Miután az jelenlegi vezető észlel egy eseményt az eseményforrás halmazból, először kijelöl egy követő szálat, hogy legyen egy új vezető. Aztán eljátssza a feldolgozó szál szerepét, mely demultiplexálja és elküldi az eseményt a kijelölt eseménykezelőnek, mely elvégzi az alkalmazásspecifikus esménykezelést a feldolgozó szálban. Több feldolgozó szál képes az eseményeket konkurrensen kezelni, amíg az aktuális vezetőszál új eseményre várakozik a szálak között megosztott eseményforrás halmazból. Miután kezelte az eseményt, a feldolgozó szál visszavált követő szerepbe, és vár, hogy újra vezetőszál legyen.

Felépítés

Négy kulcsszereplője van a Leader/Followers mintának:

A *kezelőket* az operációs rendszer biztosítja hogy azonosítsák az eseményforrásokat, mint a hálózati kapcsolatok vagy nyitot fájlok, melyek eseményeket generálnak és sorba tesznek. Az

események külső forrásból is jöhetnek, mint a connect vagy read események, melyeket egy szolgáltatásnak küldenek a kliensek, vagy belső események, mint a timeoutok. 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 eg műveletet kezdeni a halmazbeli kezelőn anélkül, hogy az a művelet blokkolna.
Az OLTP szerverk két típusú eseményben érdekeltek – connect és read események- melyek a bejövő kapcsolatokat és a tranzakciókéréseket reprezentálják. Mind a frontend, mind a backend szerverek fenntartanak külön kapcsolatot minden egyes klienshez, ahol a frontend szerverek kliense úgy nevezett távoli kliens és a frontend szerverek pedig kliensei a backend szervereknek. Mindegyik kapcsolat egy eseményforrás, melyet a szerverben külön socketkezelő reprezentál. A mi OLTP szerverünk a select() eseményt használja demultiplexálásra, mely azaonsítja azokat a kezelőket, melyek eseményforrásai rendelkeznek várakozó eseményekkel, így az alkalmazás képes I/O műveleteket meghívni ezeken a kezelőkön anélkül hogy blokkolná a hívó szálakat.

Osztály Kezelő és kezelőhalmaz	**Együtt működő**
Felelősség A kezelő azonosítja az eseményforrást az operációs rendszerben A kezelőhalmaz a kezelők egy gyűjteménye	

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. Főleg konkrét eseménykezelők implementálják a horogmetódusokat melyek a kezelőn érkezett eseményeket dolgozzák fel.

Osztály Eseménykezelő	**Együtt működő** Kezelő
Felelősség Definiál egy interfészt a kezelőn történő események feldolgozására	

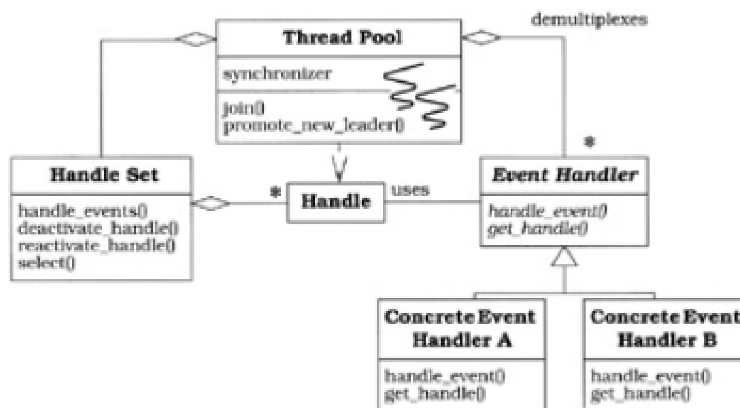
Osztály Konkrét eseménykezelő	**Együttműködő** Kezelő
Felelősség Alkalmazási szolgáltatást definiál Feldolgozza a kezelőre érkező eseményeket alkalmazás specifikus módon Egy feldolgozó szálban fut	

Például, a konkrét eseménykezelők az OLTP frontend kommunikációs szerverekben távoli kliens kéréseket fogadnak és validálnak, majd továbbítják a kéréseket a backend adatbázis szerverekhez. Hasonlóan, a konkrét eseménykezelők a backend adatbázisszerverekben tranzakciókérelmeket kapnak a frontend szerverekből, hogy írják/olvassák a megfelelő adatbázisrekordot tranzakcióvégrehajtásra és térjenekvissza eredményekkel a frontend

szerverek részére. Minden hálózati I/O műveletet socketkezelőkön keresztül hajtanak végre, mely azonosítja a különféle eseményforrásokat.
A Leader/Followers minta szívében van a szálkészlet, mely a szálak egy csoportja, 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. Ezen szálak egyikét kiválasztják a vezetőnek, mely várakozik, hogy egy esemény történjen a kezelőhalmaz egyik kezelőjén. 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.

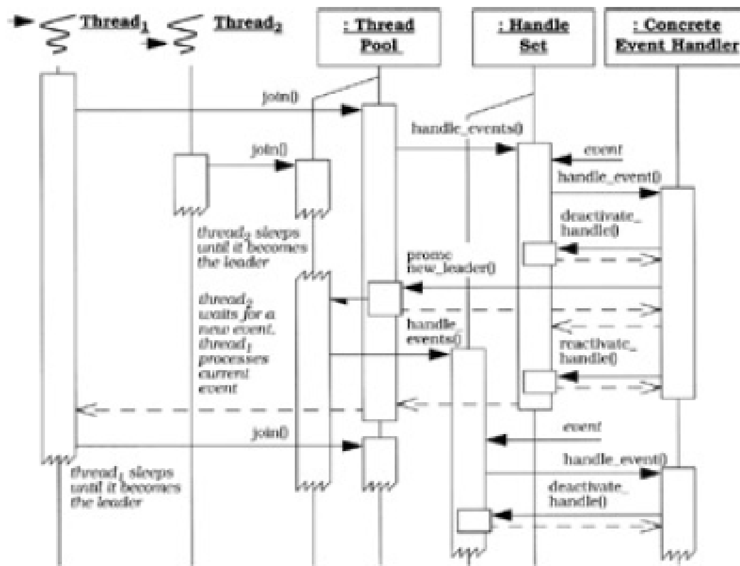
Osztály Szálkészlet	**Együttműködő**
Felelősség Szálak melyek háromféle szerepet játszhatnak: a vezető eseményekre vár, a feldolgozó feldolgozza az eseményeket, a követú várakozik, hogy vezető legyen. Tartalmaz szinkronizálót	Kezelőhalmaz Kezelő Eseménykezelők

Minden, a Leader/Follower minta segítségével tervezett OLTP szervernek lehet egy szálkészlete, mely tranzakciókérések feldolgozására vár, melyek a kezelőhalmaz által azonosított eseményforrásokon érkeznek. Tetszőleges időpillanatban, több szál a készletből dolgozhat fel tranzakciós kéréseket és küldheti vissza az eredményeket a klienseiknek. Egy szál a készletből az aktuális vezető, mely várakozik az új connect vagy read esemény érkezésére a szálak által megosztott kezelőhalmazba érkezve. Amikor ez megtörténik, a vezetőszál feldolgozószállá változik és kezeli az eseményt, míg a készlet egyik követőszálát új vezetővé léptetik elő.
A következő osztálydiagram illusztrálja Leader/Followers minta résztvevőinek struktúráját. Ebben a struktúrában, több szál osztja a szálkészlet, eseménykezelő és kezelőhalmaz résztvevőinek azonos példányait. A szálkészlet biztosítja a szálak helyes és hatékony koordinációját.

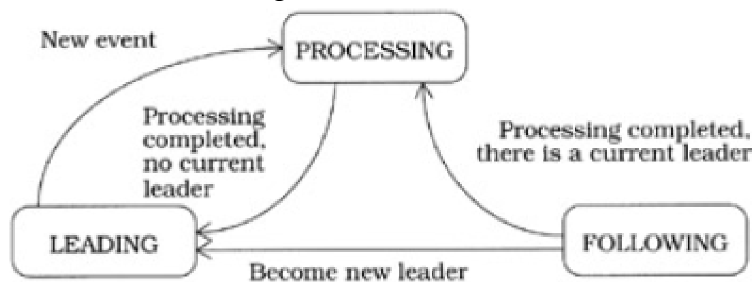


Dinamika
A Leader/Follower mintabeli együttműködés négy fázisra osztható szép:

- *Vezető szál demultiplexálás.* A vezető szál várakozik egy esemény bekövetkezésére a kezelőhalmazt egyik kezelőjén. Ha nincs aktuálisan vezetőszál, például mert az események gyorsabban jönnek, mint ahogy a szálak kiszolgálni képesek őket, az operációs rendszer belsőleg sorbaállítja az eseményeket, amíg egy vezetőszál elérhető.
- *Követő szál előléptetés.* Miután a vezetőszál észlelte az új eseményeket,a szálkészlet segítségével választ egy követő szálat, hogy az új vezető legyen.
- *Az eseménykezelő demultiplexálja az eseménykezelést*. Miután előléptették a követőszálat, hogy új vezetővé váljon, az előző vezető a feldolgozó szerepet játssza ezután. Ez a szál konkurensen demultiplexálja az észlelt eseményt az eseményhez kapcsolat kezelőhöz és aztán elküldi a horogmetódust hogy feldolgozza az eseményt. A feldolgozószál a vezetőszállal konkurensen fut és midnen más szállal a feldolgozó állapotban.
- *Újracsatlakozás a szálkészlethez*. Miután a feldolgozószál az eseménykezelést befejezte, újracsatlakozhat az eseménykészlethet és várhat hog egy újabb eseméynt dolgozzon fel. A feldolgozó szál azonnal vezetővé válik, ha nincs éppen vezető szál. Egyébként a feldolgozó szál visszatér hogy a követő szerepet játssza és várakozik a szálkészlet szinkronizálójánál, amíg a vezető elő nem lépteti.



A szál állapotváltozásait az alábbi diagramon ábrázoltuk:



Implementáció

Hat tevékenység segítségével lehet a Leader/Followers mintát implementálni:

1. *Kezelő és kezelőhalmaz mechanizmusok kiválasztása.* A kezelőhalmaz a kezelők egy gyűjteménye, melyeket a vezetőszál arra használ, hogy várakoznak egy eseményre az eseményforrás halmazban. A fejlesztők gyakran az operációs rendszer által biztosított kezelőket és kezelőhalmaz mechanizmusokat választják, ahelyett hogy a semmiből implementálnák őket. Három altevékenység segít a kezelők és kezelőhalmaz mechanizmusok kiválasztásában:
    1. *A kezelők típusainak meghatározása*. Két általános kezelőtípus van.
        - *Konkurens kezelők.* Ez a típus lehetővé teszi több szál számára, hogy elérjen egy kezelőt az eseményforrásokhoz *konkurensen* anélkül hogy versenyhelyzetet okozna mely elrontja, elveszti vagy ételmetlenné teszi az adatokat. Például a record-orientált protokollok Socket API-ja, mint az UDP, lehetővé teszi több szál számára, hogy meghívja a read() vagy write() műveletet ugyanazon a kezelőn konkurensen.
        - *Iteratív kezelők.* Ebben a típusban több szál egy kezelőt *iteratívan* érel, mert a konkurens hozzáférés versenyhelyzetet teremt. Például a bájtfolyam-központú protokollok Socket API-ja, mint a TCP, nem garantálja, hogy a read() vagy write() műveletek tiszteletben tartják az alkalmazás-szintű üzenethatárokat. Ezért, elrontott vagy elvesztett adatot eredményezhet ha az I/O műveletek a Socketen nincsnek megfelelően szerializálva.
    2. *A kezelőhalmaz típusának meghatározása.* Két általános típusan van a kezelőhalmazoknak:
        - *Konkurens kezelőhalmaz*. Ezen a típuson lehet konkurensen eljárni, mint például egy szálhalmaz által. Minden alkalommal, amikor lehetővé válik egy művelet elindítása egy halmazbeli kezelőn a művelet blokkolása nélkül, a konkurens kezelőhalmaz visszaadja a kezelőt az egyik őt hívó szálnak. Például a Win32 WaitForMultipleObjects() függvénye támogatja a konkurens kezelőhalmazokat azáltal, hogy egy szálkészletet engedélyez, hogy várjanak ugyanarra a halmazra egyidejűleg.
        - *Iteratív kezelőhalmaz*. Ez a típus visszatér hívójához, ha lehetséges egy utasítást elindítani *egy vagy több* kezelőn a halmazban anélkül hogy a műveletek blokkolnának. Bár az iteratív kezelőhalmaz képes több kezelőt visszaadni, egyszerre csak egy szál képes egyszerre meghívni. Például a select() és poll() függvények támogatják az iteratív kezelőhalmazokat. Ezért a szálkészlet nem használhatja a select() vagy poll() hívásokat események demultiplexálására konkurensen ugyanazon a kezelőhalmazon, mert több szálat értesíthetnek ugyanazokkal a várakozó I/O eseményekkel, mely hibás működéshez vezethet.

    Az alábbi táblázat összegzi a jellegzetes példákat a konkuresn és iteratív kezelők és kezelőhalmazok minden kombinációjára.

	Konkurens kezelők	Iteratív kezelők
Konkurens kezelőhalmaz	UDP Socketek + WaitForMultipleObjects()	TCP Socketek + WaitForMultipleObjects()

Iteratív kezelőhalmaz	UDP Socketek + select().poll()	TCP Socketek + select().poll()

3. *Egy adott kezelő és kezelőhalmaz kiválasztásának következményeinek meghatározása.* Általában a Leader/Followers mintát annak megakadályozására használják, hogy több szál elrontson vagy elveszítsen adatokat hibásan, mint például read műveletet hívjanak egy megosztott TCP bájtfolyam socketen konkurensen vagy egy select()-et hívnak egy megosztott kezelőhalmazon konkurensen. Azonban néhány alkalmazást nem kell ezen problémáktól védeni. Különösen, ha a kezelő és a kezelőhalmaz mechanizmusai mindketten konkurrensen, a következő implementációs tevékenységek nagyrésze kihagyható.
   Ahogy az 1.1-es és 1.2-es implementációs tevékenységnél tárgyaltuk, a hálózati programozási API-k és protokollok bizonyos kombinációinak szemantikája engedélyei a konkurens I/O műveleteket egy megosztott kezelőn. Például a Socket API UDP támogatása biztosítja hogy egy teljes üzenet mindig vagy olvasva vagy írva van egy vagy egy másik szál által, annak kockázata nélkül, hogy részleges read() vagy adatromlás az átlapolt write() miatt bekövetkezne. Hasonlóan, bizonyos kezelőhalmaz mechanizmusok, mint a Win32 WaitForMultipleObjects() függvény hívásonként egy kezelőt ad vissza, melyek lehetővé teszi, hogy egy kezelőszálkészlet konkurensen meghívja.
   Ezekben a helyzetekben, lehetséges a Leader/Followers minta implemenálása egyszerűen az operációs rendszer szálütemezőjét használva a szálak (de)multiplexálására, kezelőhalmazokhoz, és robosztus kezelőkhöz, mely esetben a 2-6 implementációs tevékenység kihagyható.
4. *Egy eseménykezelőt demultiplexáló mechanizmus implementálása*. Azon felül, hogy egy eseménydemultiplexáló várakozik egy vagy több eseményre hogy a kezelőhalmazán, mint a select() bekövetkezzen, a Leader/Followers minta implementációjának az eseményeket is demultiplexálnia kell az eseménykezelőköz és elküldeni a horogmetódusukat, hogy feldolgozzák az eseményeket. Általában két alternatív stratégiát lehet használni ezen mechanizmus implementálásához:
   - *Program az alacsonyszintű operációs rendszer események demultiplexálási mechanizmusához.* Ebben a stratégiában az operációs rendszer kezelőhalmaz demultiplexáló mechanizmusa direktben kerül használatba. Ezért a Leader/Followers implementációnak fenn kell tartania egy demultiplexálási táblázatot ami a <*kezelő, eseménykezelő, eseménytípusok*> hármasokat tartalmazó menedszer. Mindegyik kezelő kulcsként működik, mely összekapcsolja a kezelőket az eseménykezelőket az demultiplexálási táblázatban, mely ugyancsak tárolja az eseménytípusokat, mint a connect és read, amit minden eseménykezelő fel fog dolgozni. Ennek a táblázatnak a tartalmát átkonvertálják kezelőhalmazzá, amit egy natív demultiplexálási mechanizmusnak adnak át, mint a select() vagy WaitForMultipleObjects().
     A Reactor minta 3.3-as implementálási tevékenysége mutatja miként lehet egy demultiplexáló táblázatot implementálni.

- *Program egy magaszintű esemény demultiplexáló mintához.* Ebben a stratégiában a fejlesztők magaszintű mintákat használnak ki mint a Reactor, Proactor vagy Wrapper Facade. Ezek a minták segítik a Leader/Followers minta implementációjának egyszerűsítését és a szükséges erőfeszítést csökkenteni, ami a natív operációs rendszer kezelőhalmaz demultiplexáló mechanizmusok direkt programozásából véletlenül adódhat. Ezen felül a magas-szintű minták használata egyszerűvé teszi a rendszer I/O és a demultiplexáló aspektusainak szétválasztását a konkurenciamodelljétől ezáltal a kódtöbbszöröződést és karbantartási nehézségeket csökkentését.

  A mi OLTP szerver példánkban az eseményt a konkrét eseménykezelőhöz kell demultiplexálni, ami a socketkezelővel volt társítva ami az eseményt megkapta. A Reactor minta támogatja ezt a tevékenységet, ezért használható, hogy egyszerüsítése a Leader/Followers minta implementációját. A Leader/Followrs minta kontextusában azonban a reactor csak *egy* kezelőt demultiplextál egyszerre a társított konkrté eseménykezelőhöz, függetlenül attól, hogy hány kezelőnek van várakozó eseménye. Csak egy kezelő egyidejű demultiplexálása maximalizálhatja a konkurenciát a készlet szálai között és egyszerüsítheti a Leader/Followers minta implementációját úgy, hogy a várakozó eseményekhez rendelt különböző sorok kezelésének szükségét csökkenti.

2. *A kezelőhalmazban levő kezelők ideiglenes (de)aktiválására protokoll implementálása.* Amikor egy esemény beérkezik, a vezető három lépést végez el:
   1) Deaktiválja a kezelőhalmazból ideiglenesen a kezelőt
   2) Előlépteti egy követőszálat hogy új vezető legyen és
   3) Folytatja az esemény feldolgozását

   A kezelő deaktiválása a kezelőhalmazból elkerüli a versenyhelyzeteket, melyek az új vezető megválasztása és az esemény feldolgozásának elkezdése között telne el. Ha az új vezető ugyanarra a kezelőre várakozik a kezelőhalmazból ez alatt az időintervallum alatt, egy eseményt demultiplexálhatna másodszor is, mely hibás működés, mert a küldés már folyamatban van. Miután az eseményt feldolgozták, a kezelőt újra-aktiválják a kezelőhalmazba, mely lehetővé teszi a vezetőszálnak, hogy rajta vagy más aktivált kezelőkön esemény következzen be az eseményhalmazban.

   A mi OLTP példánkban a kezelődeaktiválási és újraaktiválási protokollt a Reactor interfészének kiterjesztésével lehet megadni, ahogy a Reactor minta 2-es implementálási tevékenységében lett megadva:

```
class Reactor {
public:
        // ideiglenesen deaktiválja a <HANDLE>-t
        // a belső kezelőhalmazból

        void deactivate_handle (HANDLE, Event_Type);

        // Újra aktiválja a korábban deaktivált
        // <Event_Handler>-t a belső kezelőhalmazba.
```

```
        void reactivate_handle (HANDLE, Event_Type);
        // …

};
```

3. *Szálkészlet implementálása.* Hogy egy követő szálat vezetővé léptessünk elő és hogy emeghatározzuk mely szál az aktuális vezető, a Leader/Followers mintának egy szálkészletet kell menedzselnie. Ez implementáció egyenes útja, ha az összes szál a halmazban egyszerűen vár egy szinkronizálón, ami egy szemafor vagy állapotváltozó. Ebben a szerkezetben nem számít hogy melyik szál dolgozik fel egy eseményt, amíg az összes szál a készletben melyek eseményeket osztanak meg sorosítva vannak.
   Például az alább mutatott LF_Thread_Pool osztályt lehet használni a backend adatbázisszerverekhez az OLTP példánkban:

```
class LF_Thread_Pool {
public:

        // Konstruktor.
        LF_Thread_Pool (Reactor *r): reactor_ (r) { }

        // A szálak hívják a <join>-t hogy várjon a kezelőhalmazra
        // és demultiplexálja az eseményeket az eseménykezelőikre.
        void join (Time_Value *timeout = 0);

        // Előlépteti a követőszálat, hogy
        // az új vezető legyen
        void promote_new_leader ();
        // Támogatja a <HANDLE> (de)aktivációs protokollt.
        void deactivate_handle (HANDLE, Event_Type et);
        void reactivate_handle (HANDLE, Event_Type et);

private:
        // Pointer az esemény demultiplexáló/elküldőre.
        Reactor *reactor_;
        // A vezetőszál id-je, melyet
        // NO_CURRENT_LEADER-re állítanak ha nincs vezető.
        Thread_Id leader_thread_;

        // A követőszál várakozik erre az állapotváltozóra
        // amíg vezetővé nem választják.
        Thread_Condition followers_condition_;

        // A belső állapothoz való hozzáférést sorosítja.
        Thread_Mutex mutex_;
};
```

   Az LF_Thread_Pool konstruktora cacheli a neki átadott reaktort. Alapból ez a reaktor implementáció használja a select()-et, mely támogatja az iteratív kezelőhalmazokat. Ezért, a LF_Thread_Pool felelős a select()-et hívó szálak sorosításáért a reaktor kezelőhalmazában.
   Az alkalmazási szálak hívják a join()-t, hogy várakozzon a kezelőhalmazra, és demultiplexálja az új eseményeket az asszociált eseménykezelőikre. Ahogy a 4-es implementációs műveletben mutatjuk, ez a metódus nem tér vissza a hívójához, amíg az alkalmazás nem terminál vagy a join() timeoutol. A promote_new_leader() metódis

előlépteti az egyik követőszálat a halmazból, hogy az új vezető legyen, ahogy az 5.2-es implementációs tevékenységben mutatjuk.
A deactivate_handle() metódus és a reactive_handle() metódus deaktiválja és újraaktiválja a kezelőket a reaktor kezelőhalmazában. Ezen metódusok implementálása egyszerűen továbbítja a reactor interfaceben megadott ugyanilyen metódisokat, ahogy a 2-es implementációs tevékenységben mutattuk.
Megjegyezzülk, hogy egy szinkronizáló állapotváltozó followers_condition van a szálkészlet szálai között megosztva. Ahogy a 4-es és 5-ös implementációs tevékenységben látható, az LF_Thread_Pool a Monitor Object mintát használja.

4. *Protokoll implementálása mely a szálaknak lehetővé teszi a szálkészletbe való kezdeti belépést (majd későbbi újrabelépést).*
   Ez a protokoll a következő két esetben használatos:
   - A készlet szálainak kezdeti létrehozásakor, melyek begyűjtik és feldolgozzák az eseményeket és
   - Amikor egy feldolgozószál befejeződik és kész egy új esemény kezelésére

   Ha nem elérhető a vezető, akkor a feldolgozó szál azonnal vezetővé válhat. Ha van már vezetőszál, akkor követővé válhat a szálkészlet szinkronizálójánál várakozva.
   A backend adatbázisszervereink a LF_Thread_pool join() metódusát implementálhatják úgy, hogy várakozik a kezelőhalmazra majd demultiplexálja az új eseményeket a hozzájuk kapcsolt eseménykezelőkre

```
void LF_Thread_Pool::join (Time_Value *timeout) {
        // Scoped Locking idiómát használ, hogy megszerezze a mutexet
        // automatikusan a konstruktorban
        Guard<Thread_Mutex> guard (mutex_);

        for (;;) {
                while (leader_thread_ != NO_CURRENT_LEADER)
                // Alszik és elengedi a <mutex>-et automatikusan.
                followers_condition_.wait (timeout);

                // Felveszi a vezető szerepet.
                leader_thread_ = Thread::self ();

                // Ideiglenesen elhagyja a monitort, hogy a
                // követőszálak csatlakozhassank a készlethez.
                guard.release ();

                // Miután vezetővév válik, a szál a
                // reaktort használja az eseménykre váráshoz.
                reactor_->handle_events ()'

                // Újra belép a monitorba, hogy sorosítsa a tesztet
                // a <leader_thread_>-nek amíg a ciklusban van
                guard.acquire ();
        }
}
```

   A for cikluson belül a hívó szál a vezető, feldolgozó és követő szál szerepek között váltakozik. A ciklus első felében várakozik, amíg vezető lehet, mely ponton a reactor mintát használja, hogy a megosztott kezelőhalmazon egy eseményre várjon. Amikor a reactor egy eseményt észlel a kezelőn, demultiplexálja az eseményt a kapcsolódó

eseménykezelőre és elküldi a handle_event() metódust hogy egy új vezetőt válasszon és feldolgozza az eseményt. Miután a reactor demultiplexál egy eseményt, a szál újra felveszi a vezető szerepet. Ezek a szerepek addig futnak a ciklusban, amíg az alkalmazás nem terminál vagy időtúllépés történik.

5. *Követő előléptető protokoll implementálása.* Rögtön azután hogy a vezető érzékel egy eseményt, de még azelőtt, hogy demultiplexálja az eseményt egy eseménykezelőre és feldolgozza azt, egy követő szálat új vezetővé kell előléptetnie. Két altevékenységre lehet szükség ezen protokoll implementációjához:
   1. *A kezelőhalmaz szinkronizációs protokoll implementálása.* Ha a kezelőhalmaz iteratív és mi vakon léptetünk elő egy új vezetőszálat, lehetséges, hogy az új vezetőszál megpróbálja ugyanazt az eseményt kezelni, amit az előző vezetőszál észlelt és éppen a feldolgozásának a közepén van. Hogy ezt a versenyhelyzetet elkerüljük, ki kell vennünk a kezelőt a kezelőhalmazból mint lehetséges választható kezelő, mielőtt egy követőt új vezetővé léptetnék elő és elküldenénk az eseményt a konkrét eseménykezelőjének. A kezelőt aztán újra aktiválni kell, miután az eseményt elküldték és feldolgozták.

      Egy alkalmazás implementálhat konkrét eseménykezelőket, melyek az Event_Handler osztály alosztájai, ahogy a Reactor minta 1.2-es implementációs tevékenységében ismertettük. Hasonlóan, a Leader/Followers implementáció használhatja a Decorator mintát, hogy egy LF_Event_Handler osztályt hozzon létre, mely „dekorálja" az Event_Handlert. Ez a dekorátor lépteti elő az új vezetőszálat és aktiválja/deaktiválja a kezelőt a reaktor kezelőhalmazában, a konkrét eseménykezelőknek transzparensen.

```
class LF_Event_Handler : public Event_Handler {
public:
        LF_Event_Handler (Event_Handler *eh, LF_Thread_Pool *tp)
                : concrete_event_handler_ (eh),
                thread_pool_ (tp) { }

        virtual void handle_event (HANDLE h, Event_Type et) {
        // Ideiglenesen deaktiválja a kezelőt
        // a reaktorban, hogy elkerülje a versenyhelyzetet.

        thread_pool_->deactivate_handle (h, et);
        // Előlépteti a követőosztályt hogy új vezető legyen.
        thread_pool_->promote_new_leader ();

        // Elküldi az alkalmazásspecifikus esemény
        // feldolgozó kódot.

        concrete_event_handler_->handle_event (h, et);

        // Újraaktiválja a kezelőt a reaktorban.
        thread_pool_->reactivate_handle (h, et);
        }

private:
        // Az <Event_Handler> ezen használata játssza a
        // <ConcreteComponent> szerepet a Decorator
```

```
        // mintában, melyet az alkalmazásspecifikus
        // funkcionalitás implementálására használnak.
        Event_Handler *concrete_event_handler_;

        // Az <LF_Thread_Pool> példánya.
        LF_Thread_Pool *thread_pool_;
};
```

2. *Az előléptető protokoll sorrendezésének meghatározása.* Több sorrendezőstratégia létezik, hogy meghatározzuk melyik követőszálat léptessük elő.
   - *LIFO sorrend.* Sok alkalmazásban nem számít hogy melyik követő szálat léptetik elő legközelebb, mert minden szál ekvivalens társ. Ebben az esetben a vezető a szálakat előléptetheti az *utolsó-be,első-ki (last-in,first-out LIFO)* sorrendben. A LIFO sorrend maximalizálja a CPU cache affiinitását azáltal, hogy biztosítha hogy a *legrövidebb* időt várakozó szálat léptetik előre legelőször, mely jó példája a Fresh Work Before Stale mintának.
     A cache affinitás növeli a rendszerteljesítményt ha a legutoljára blokkolt szál gyakorlatilag ugyanazt a kódot és adatot futtatja, amikor újra futásra ütemezik. A LIFO előléptetési protokoll implementálása további adatstruktúrát idényel, mint például a várakozó szálak veremje, mint sem a natív operációs rendszer egy szinkronizációs objektumáthasználnánk mint például egy szemafor.
   - *Prioritás sorrend.* Néhány alkalmazásban, különösen valósidejű alkalmazásokban, a szálak különböző prioritással futnak. Ebben az esetben, ezért, szükséges, hogy a követőszálat a prioritása alapján válasszák vezetővé. Ezt a protokollt valamiféle prioritássor segítségével implementálhatjuk, mint például a heap. Bár ezt a protokoll bonyolultabb mint a LIFO, szükséges lehet a követőszálakat prioritásuk szerint előléptetni, hogy a prioritásinverziót minimalizáljuk.
   - *Implementáció által definiált sorrend.* Ez a sorrend a leggyakoribb amikor a kezelőhalmazt operációs rendszer szinkronizálókkal implementálják, mint szemafórok vagy állapotválozók, melyek gyakran küldik a szálakat egy implementáció által definiált sorrendben. Ezen protokoll előnye, hogy a natív operációs rendszer szinkonizálókra hatékonyan képeződik le.
     A mi OLTP backend szervereink használhatják az alábbi egyszerű protokollt, hogy követőszálakat előléptessenek a natív operációs rendszer állapotváltozóinak sorbarendezésétől függetlenül:

```
void LF_Thread_Pool::promote_new_leader () {
        // A Scoped Locking idiómát használjuk hogy
        // a mutexet automatikusan a konstruktorban
        // megszerezzük.
        Guard<Thread_Mutex> guard (mutex_);
        if (leader_thread_ != Thread::self ())
        throw /* ...csak a vezető léptethet elő... */;

        // Jelzi, hogy már nem vagyunk vezetők
        // és értesíti a <join> metódust hogy léptesse elő
```

```
		// a következő követőt.
		leader_thread_ = NO_CURRENT_LEADER;
		followers_condition_.notify ();
.
		// El kell engedni a mutexet a destruktorban.
	}
```

Ahogy az 5.1-es implementációs tevékenységnél látható, a promote_new_leader() metódus az LF_Event_Handler dekorátor hívja meg mielőtt továbbítja a konkrét eseménykezelőnek mely feldolgozza az eseményt.

---

6. *Eseménykezelők implementálása*. Az alkalmazásfejlesztőknek el kell dönteniük, hogy milyen tevékenységeket kell elvégezni, amikor a konkrét eseménykezelő horogmetódusát meghívja a feldolgozó szál a Leader/Followers minta implementációjában. A Reactor minta 5-ös implementációs tevékenysége leírja a konkrét eseménykezelők implementálásával kapcsolatos különféle problémákat.

Megoldott Példa

A Példa fejezetben bemutatott OLTP backend adatbázis használhatja a Leader/Followers mintát, hogy egy szálkészletet implementáljon, ami hatékonyan demultiplexálja az I/O eseményeket a socket kezelőkről az ő eseménykezelőikre. Ebben a szerkezetben nincs kijelölt hálózati I/O szál. Helyette a szálak a szálak egy készlete előre le van foglalva az adatbázisszerver inicializálásánál:

```
const int MAX_THREADS = /* ... */;

// Elődeklaráció.
void *worker_thread (void *);

int main () {
	LF_Thread_Pool thread_pool (Reactor::instance ());
	// A passzív-módú Acceptor kódját lehagytuk.
	for (int i = 0; i < MAX_THREADS - 1; ++i)
		Thread_Manager::instance ()->spawn
			(worker_thread, &thread_pool);
	// A főszál részt vesz a szálkészletben.
	thread_pool.join ();
};
```

Ezek a szálak nincsenek egyetlen socketkezelőhöz sem kötve. Ezért, minden szál ebben a készletben felváltva játssza el a hálózati I/O szál deladatát az LF_Thread_Pool::join() függvény hívásával

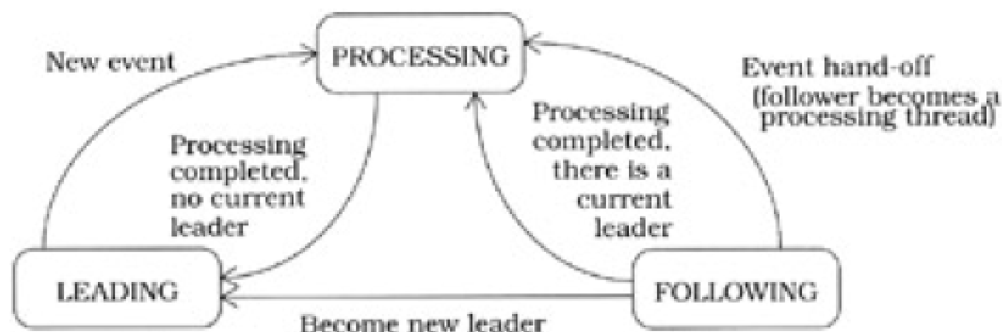
```
void *worker_thread (void *arg) {
	LF_Thread_Pool *thread_pool =
		static_cast <LF_Thread_Pool *> (arg);

	// Minden dolgozó részt vesz a szálkészletben.
	thread_pool->join ();
};
```

Ahogy a 4-s implementációs tevékenységnél mutattuk, a join() metódus csak a vezetőszálnak engedélyezi hogy a Reactor singleton segítségével select()-et hívjon aa Socketkezelők

megosztott halmazán, melyek az OLTP backend szerverekez vannak kapcsolódva. Ha a kérések megérkeznek, amikor minden szál elfoglalt, sorba lesznek állítva a socketkezelőkön, amíg a szálak a készletben elérhetőek lesznek a kérések végrehajtására.
Amikor egy kérés esemény megérkezik, a vezetőszál deaktiválja ideiglenesen a socketkezelőt hogy a select() kezelőkészletébe bele kerüljön-e, előléptet egy követő szálat, hogy új vezető legyen, majd folytatja a kérés esemény feldolgozását mint feldolgozó folyamat. Ez a feldolgozó szál ezek után beolvassa a kérést egy bufferben, mely vagy a futási idejű veremben van, vagy a Tread-Specific Storage minta segítségével lett elhelyezve. Minden OLTP tevékenység egy feldolgozó szálban történik. Ezért nincs szükség további kontextusváltásra, szinkronizációra, adatmozgatásra amíg a folyamat be nem fejeződik. Amikor befejezi a kérés kezelését, a feldolgozószál visszatér hogy a követő szál szerepét játssza és várakozik a szinkronizálón a szálkészletben. Ezen felül, a socketkezelő melyet feldolgoztak újra aktiválódik a Reactor singleton kezelőhalmazában, hogy a select() tudjon várakozni az I/O eseményekre hogy bekövetkezzenek vagy más Socketen a kezelőhalmazon.

Változatok

Kötött Kezelő/Szál Kapcsolatok. A minta korábbi fejezeteiben előre nem összekötött kezelő/szál kapcsolatokat mutattunk be, ahol nincs fix kapcsolat a szálak és a kezelők között. Ezért, tetszőleges szál feldolgozhat tetszőleges eseményt, mely a kezelőhalmaz tetszőleges kezelőjén történik. A kötetlen kapcsolatokat általában akkor használják, amikor adolgozó szálak egy csoportja felváltva demultiplexál egy megosztott kezelőhalmazt.
A Leader/Followers minta egy változata kötött kezelő/szál kapcsolatokat használ. Ebben a változatban minden minta az ő kezelőjéhez van kötve, melyet arra használja, hogy adott eseményeket dolgozzon fel. A kötöt kapcsolatokat általában az alkalmazás kliensoldalán használják, amikor egy szál várakozik egy socketkezelőre egy válaszért, melyet egy kétirányú kérésként küldött egy szerverhez. Ebben az esetben a kliensalkalmazás szála elvárja, hogy az válasz eseményt is ezen a kezelőn dolgozzák fel, ugyanazon szál által, mint az eredeti kérést.
A kötött kezelő/szál kapcsolatok variánsnál, ezért a vzetőszálnak a szálkészletben át kell adnia egy eseményt egy követőszálnak, ha a vezetőnek nincs meg a szükséges kontextusa az esemény feldolgozására. Miután a vezető észlel egy új seményt, ellenőrzi az eseményhez kapcsolt kezelőt, hogy mely szál felelős a feldolgozásáért.
Ha a vezető felfedezi, hogy ő felelős az eseményért, akkor előléptet egy követőszálat hogy új vezető legyen. Azonban, ha az eseményt egy másik szálnak szánták, a vezető átadja az eseményt a megjelölt követőszálnak. Ez a követőszál aztán ideiglenesen kiiktatja a kezelőt és feldolgozza az eseményt. Eközben a jelenlegi vezetőszál folytatja a várakozást a kezelőhalmazon egy másik esemény bekövetkezésére.
Az alábbi ábra mutatja be az új váltási mechanizmust a követő és feldolgozó állapot között:



A Leader/Follower szálkészletet lehet implicit karbntartani, például egy szinkronizáló segítségével mint egy szemafor vagy állapotváltozó, vagy egy konténer és a Manager minta segítségével. A választás nagyban attól függ, hogy a vezetőszálnak explicit kell-e értesítenie egy követőszálat az esemény átadásról.
Sorosítási kényszerek egyszerűsítése. Vannak operációs rendszerek, ahol több vezetőszál várakozhat a kezelőhalmazon egyszerre. Például a Win32 WaitForMultipleObjects() támogatja a konkurens kezelőhalmazt, mely lehetővé teszi egy szálkészletnek, hogy ugyanazon a kezelőhalmazon konkurensen várakozzon. Ezért, egy ilyen képességű szálkészlet ki tudja használni egy többprocesszoros hardver előnyeit, hogy több eseményt konkurensen kezeljen, míg más szálak az eseményekre várakoznak
Két változata van a Leader/Follower mintának, melyet alkalmazni lehet hogy több vezetőszálat engedjünk meg, hogy egyidejűleg aktív legyen:

- *Leader/followers kezelőhalmazonként.* Eza változat a konvencionális Leader/Followers implementációt alkalmazza több kezelőhalmazra elkülönítve. Például miden szálat egy adott kezelőhalmazhoz társítanak. Ez a változat különösen hasznos olyan alkalmazásokban, ahol több kezelőhalmaz elérhető. Azonban ez a változat korlátozza a szálakat, hogy egy specifikus kezelőhalmazt használjanak.
- *Több vezető és több követő.* Ebben a változatban mintát kibővítik, hogy támogasson több egyidejű vezetőszálat, ahol a vezetőszálak bármelyike várakozhat bámelyik kezelőhalmazon. Amikor egy szál újra belép a szálkészletben, megnézi, hogy van-e vezető minden kezelőhalmazhoz kapcsolva. Ha van egy kezelőhalmaz vezető nélkül, akkor az újra belépő szál lehet annak a kezelőhalmaznak a vezetője azonnal.

Hibrid Szálkapcslatok. Néhány alkalmazás hibrid felépítést használ, mely implentálja mind a kötött és kötetlen kezelő/szál kapcsolatokat egyidejűleg. Így, az alkalmazás néhány kezelője dedikált szálakkal rendelkezik bizonyos eseményekre, míg más kezelők tetszőleges szál által feldolgozhatóak. Ezért, a Leader/Followers egy variánsa használja az esemény átadási mechanizmust, hogy a szálak egy részhalmazát értesítse attól a kezelőtől függően, melyen az esemény bekövetkezett.
Például az OLT frontend kommunikációs szerveren több szál használhatja a Leader/Follower mintát hogy új kérés eseményekre várakozzon a kliensektől. Így lehetnek szálak, melyek a kérésekre adott válaszokra várnak, melyet ők hívtak meg a backend szervereken. Valójában a szálak mindkét szerepet eljátsszák életük során, szálakként indulva melyek új bejövő kéréseket küldenek el, majd a kéréseket a backend szerverekhez rendelik, hogy kielégítség a kliensalkalmazás követelményeit, végül várakoznak a választa a backend szerverekről.
Hibrid kliens/Szerverek. Egy komplex rendszerben, ahol a társalkalmazások mind kliens és szerver szepeket is játszanak, fontos, hogy a kommunikációs infrastruktúra feldolgozza a bejövő kéréseket, miközben egy vagy több válaszra várakozik. Másképpen a rendszer holtpontba kerülhet, mert egy kliens minden szálát válaszokra várásra dedikálja.
Ebben a változatban a szálak és kezelők kötése dinamikusan változik. Például, egy szál kezdetben nem kötöt, de egy beérkező kérés feldolgozása közben az alkalamzás felfedezi, hogy egy társ által biztosított szolgáltatásra van szüksége az elosztott rendszerben. Ebben az esetben a kötetlen szál elküld egy új kérést az alkalmazáskód végrehajtása közben, gyakorlatilag a kezelőhöz kötve magát, mely elküldi a kérést. Később, amikor a válasz megérkezik és a szál befejezi az eredeti kérést, újra kötetlenné válik.
Alternatív eseményforrások és nyelők. Vegyünk egy rendszert, ahol az események nem csak kezelőkről, hanem más forrásokból is be tudnak érkezni, mint például megosztott memória, vagy üzenetsorok. UNIXban például nincs eseménydemultiplexálási funkció mely várakozni tud az I/O eseménykre, szemaforeseményekre vagy/és eseménysorokra egyidejűleg. De, egy

szál képes blokkolva várakozni események egy típusára egyszerre. Így a Leader/Followers mintát kilehet terjeszteni, hogy több mint egy eseménytípusra várakozzon egyidejűleg.:

- Egy vezetőt társítanak minden eseményforráshoz – ellentétben egy vezetőszállal az egész rendszernek
- Amikor az eseményt megkapjuk, de mielőtt feoldolgoznánk, a vezetőszálnak választania kell egy követőt, hogy ezen az eseményforráson várakozzon.

A hátránya ennek a változatnak, hogy a résztvevő szálak számának midnenképpen nagyobbnak kell lennie, mint az eseményforrások száma. Így ez a megközelítés nem olyan jól skálázható ahogy az eseményforrások száma növekszik.

Ismert használatok

ACE Thread Pool Reactor keretrendszer. Az ACE keretrendszer egy objektumorientált keretrendszer implementációját adja a Leader/Follower mintának, melyet 'szálkészlet reaktor'-nak hívnak (ACE_TP_Reactor), mely demultplexál eseményeket eseménykezelőkre a szálak egy készletében. A szálkészlet reaktor használatakor az alkalmazás előre létrehoz egy fix számú szálat. Amikor ezek a szálak meghívják az ACE_TP_Reactor handle_event() metódusát, egy szál vezető lesz és várakozik egy eseményre. A szálak kötetlenek az ACE thread pool keretrendszerben. Ezért, miután a vezető észlel egy eseményt, tetszőleges szálat előléptet, hogy új vezető legyen majd demultiplexálja az eseményt a megfelelő eseménykezelőnek.

CORBA ORB-ok és webszerverek. Sok CORBA implementáció, beleértve a Chorus COOL ORB-ot és TAO-t, használja a Leader/Followers mintát mind a kliensoldal kapcsolati modellhez, mind a szerveroldali konkurenciamodellhez. Ezen felül a JAWS Webszerver használja a szálkészlet modellt az olyan operációsrendszer platformokra, melyek nem engedélyezik az egyidejű accept() hívást a passzív-módú socketkezelőkre.

Tranzakciómonitorok. A kedvelt tranzakciómonitorok, mint a Tuxedo, tradícionálisan folyamatonkénti alapon működnek, például a tranzakciók mindig egy folyamattal vannak társítva. Hétköznapi OLTP rendszerek magas teljesítményt és skálázhatóságot igényelnek, bár a tranzakciók processz alapú futtatása lehet, hogy nem tudja ezeket a kívánalmakat teljesíteni. Ezért, a következő generációs tranzakciós szolgáltatások, mint például a CORBA Transaction Service implementációi kötött Leader/Follower kapcsolatokat használnak a szálak és a tranzakciók között.

Taxi standok. A Leader/Followers mintát használják a mindennapi életben, hogy sok repülőtéri taxistandot működtessenek. Ebben az esetben a taxisok játsszák a szálak szerepét, a vonalban első taxival mint a *vezetővel* és a többi taxivan mint a *követőkkel*. Így az utasok megérkezése a taxistandnál jelenti az események megjelenését, melyet demultiplexálni kell a taxikhoz,tipikusan FIFO sorban. Általában, ha egy taxi képes egy utast kiszolgálni, akkor ez a helyzet a *kötetlen* kezelő/szál kapcsolatoknak felel meg, ahogy a fő Implementáció fejezetben bemutattuk. Ha azonban bizonyos taxik szolgálnak ki bizonyos utasokat, ez a helyzet a kötött kezelő/szál kapcsolatoknak felel meg, ahogy a Változatok fejezetben bemutattuk.

Következmények

A Leader/Followers mintának több **előnye** van:

Teljesítménynövekedés. Összehasonlítva a Half-Sync/Half-Reactive szálkészlettel melyet a Példa fejezetben mutattunk be, a Leader/Followers minta a következő módon növeli a teljesítményt:

- Fejleszti a CPU cache affinitását és eltünteti a dinamikus memóriafoglalás és buffermegosztás szükségét a szálak között. Például egy feldolgozó szál a beolvashatja

a kérést a futási idejű vermén allokált bufferhelyre, vagy a Thread-Specific Storage minta által lefoglalt memórián.

- Minimalizálja a zárási többletszámításokat úgy, hogy nem cserél adatot a szálak között,ezáltal csökkenti a szálszinkronizációt. Kötött kezelő/szál kapcsolatoknál a vezetőszál demultiplexálja az eseményeket az eseménykezelőkre a kezelő értékétől függően. A kérést aztán kiolvasa a kezelőről a követőzszál mely az eseményt feldolgozza. Kötetlen kapcsolatoknál a vezetőszál olvassa az eseményt a kezelőről és dolgozza is fel.
- Minimalizálja a prioritásinverziót, mert nincs külön sorbanállás a szerveren. Ha valós idejű I/O rendszereken használják, a Leader/Follower szálkészlet modell csökkentheti a nemdeterminizmus forrásait a szerver kérésfeldolgozásában jelentősen.
- Nincs szükség kontextusváltásra minden egyes eseménynél, ezzel csökkentve az eseményküldési válaszidőt. Fontos, hogy a követőszál vezetővé előléptetéséhez *van* szükség kontextusváltásra. Ha két esemény érkezik egyidejűleg, ez növeli a második esemény küldési válaszidejét, de a teljesítmény nem rosszabb, mint a Half-Sync/Half-Reactive szálkészlet implementációknál.

Programozási egyszerűség. A Leader/Follower minta egyszerűsíti a konkurens modellek programozását, hol több szál fogadhat kéréseket, dolgozhat fel válaszokat és demultiplexálhat kapcsolatokat egy megosztott kezelőhalmazon.
A Leader/Follower mintának az alábbi **hátrányai** vannak:
Implementációs bonyolultság. A fejlettebb Leader/Follower változatokat nehezebb implementálni, mint a Half-Sync/Half-Reactive szálkészleteket. Különösen, amikor többszálú kapcsolat multiplexelésére használják, a Leader/Followers mintának követő szálak egy készletét kell karbantartania melyek kérések feldolgozására várnak. Ezt a halmazt frissíteni kell, amikor egy követőszálat előléptetnek vezetővé és amikor egy szál újra csatlakozik a követő szálak készletébe. Minden művelet konkurensen, nem meghatározott sorrendben történet. Így a Leader/Follower minta implementáció hatékony, míg biztosítja a műveletek atomicitását.
Rugalmasság hiánya. A Half-Sync/Half-Async minta Half-Sync/Half-Reactive változatán alapuló szálkészlet modell lehetővé teszi a sorbanállási rétegben várakozó eseményeknek, hogy kidobják őket, vagy új prioritást adjanak nekik. Hasonlóan, a rendszer több különböző sort tud fenntartani melyet a szálak dolgálnak ki különböző prioritásokkal, hogy csökkentsék a versenyt és prioritásinverziót az különböző prioritású események között. A Leader/Followers modellben aznonban nehéz kidobni vagy újrasorrendezni eseményeket, mert nincs explicit sor. Ezen funkcionalitás biztosításának egy lehetősége, hogy szolgáltatások töb szintjét kínáljuk többszintű Leader/Follower csoportokkal az alkalmazásban, mindegyik egy adott prioritású szálat kiszolgálva.
Hálózati I/O szűk keresztmetszetek. A Leader/Followers minta, ahogy a Implementáció fejezetben bemutattuk, sorosítja a feldolgozást hogy egy szálat enged egy időpillanatban, hogy várakozon a kezelőhalmazon. Néhány környezetben, ez a konstrukció szűk keresztmetszetté válhat, ha csak egy szál demultiplexálja az I/O eseményeket. Gyakorlatban ez általában nem szokott probléma lenni, mert a legtöbb I/O intenzív feldolgozást az operációs rendszer kernelje végzi. Ezért, az alkalmazásszintű I/O műveleteket gyorsan el lehet végezni.

Lásd még

A Reactor minta gyakran adja a Leader/Followers minta implementációk magját. Azonban a Reactor mintát lehet a Leader/Follower mintha helyett használni, amikor minden egyes eseménynek csak rövid feldolgozási időre van szüksége. Ebben az eseteben a további ütemezési komplexitása a Leader/Follower mintának felesleges.
A Proactor minta definiál egy másik modellt aszinkron eseménybefejezések konkurrenst demultiplexálására. A Leader/Followers minta helyett használható:

- ha az operációs rendszer támogat hatékony aszinkron I/O-t és
- ha a programozóknak nem okoz gondot a Proactor mintához csatolt aszinkron inverziója az irányításnak.

A Half-Sync/Half-Async és az Active Object minták két további alternatívát jelentenek a Leader/Followers mintának. Ezek a minták jobb választások mint a Leader/Follower minta:

- Ha van további szinkronizáció vagy sorrendezési kényszerek melyeket a kérések újrasorrendezésével lehet kezelni még azelőtt, hogy a készletbeli szálak feldolgozhatnák őket és/vagy
- Ha az eseményforrásokra nem lehet várni egy egyedülálló eseménydemultiplexer által hatékonyan.

A Controlled Reactor minta tartalmazza a teljesítménymenedzsert amely a szálak haszálatát irányítja az eseménykezelőknek, a felhasználó specifikációja alapján és egy alternatíva lehet amikor irányított teljesítmény egy fontos cél.

IV. Szinkronizációs tervezési minták

IV.1. Bevezető

Többszálú alkalmazások fejlesztése összetettebb, mint a szekvenciális programoké. A nehézségek és különbségek abból a kérdéskörből fakadnak, hogy objektumainkhoz konkurensen több szál is hozzáférhet, ami nem kívánt módon rendszerünk állapotterének hibás megváltoztatásához, nem várt eredményhez, illetve esetlegesen futási hibához vezethet.

Négy tervezési mintát mutatunk be, amik a konkurens rendszerek zárolási problémáit

egyszerűsítik: *Scoped Locking*, *Strategized Locking*, *Thread-Safe interface* és a *Double-Checked Locking Optimization*.

A *Scoped Locking* C++ idióma garantálja, hogy a lockolás automatikusan megtörténik, ha lefutás elér egy adott hatókört (scope), és automatikusan feloldódik, ha elhagyja a scope-ot, a visszatérési úttól függetlenül.

A *Strategized Locking* tervezési minta a stratégia minta specializációja, ami felparaméterezi szinkronizációs folyamatokat azokban a komponensekben, amik kritikus részeiben a konkurens hozzáférés nem megengedett. Ha a módszert C++-ban implementáljuk, akkor gyakran az előbb említett *Scoped Locking* technikát is felhasználjuk a tervezésben.

A hátralévő két mintát a szinkronizációs mechanizmusok hatékonyságának és robusztusságának javítására alkalmazzuk.

A *Thread-Safe Interface* mintával a zárolásból fakadó overheadet minimalizálhatjuk, és biztosíthatjuk, hogy a komponensen belüli metódushívások ne okozzanak belső deadlockot azzal, hogy újra zárolnának olyan esetben, amikor a lock még nem feloldott.

A *Double-Checked Locking Optimization* minta a szinkronizációs overheadet minimalizálja, abban az esetben, amikor a komponens kritikus szakaszai egyszerű lockolást igényelnek a program lefutása során.

Kiegészítésként megjegyzendő, hogy a taglalt minták mindegyike felhasználható a konkurencia implementálásához használt tervezési elvek javításához. Egyéb, szinkronizációs témában felhasználható minták a *Code Locking*, *Data Locking* és a Reader/Writer Locking.

IV.2. Scoped Locking

A webszerver-alkalmazások általában regisztrálják, hogy az egyes *URL*-ekhez a látogatók hányszor fértek hozzá adott idő alatt. A latency csökkentésére a webszerverhez tartozó processz a letöltésszámlálót egy memória-rezidens komponensben tárolja egy diszk fájl helyett. A webszerverek tipikus példái a többszálú alkalmazásoknak, mivel a párhuzamos működéssel a feldolgozás hatékonyságát lehet növelni. A letöltésszámláló problémája esetünkben az, hogy a konkurens frissítést elkerülve a számlálásért felelős publikus

metódusokat szerializálni kell. Ennek egy lehetséges módja, hogy minden ilyen publikus metódus lefoglalja és felszabadítja a zárolást.

Problémák

Az említett 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. Pl C++ programnyelv esetén a következő esetekben lehetséges a metódusból való kilépés: return, break, continue vagy goto. Ezeken kívül előfordulhat nem várt kivételdobás, ami a kritikus hatókörön kívülre vezet.

Megoldások

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 Guard osztályban pointerrel célszerű megoldani a lockolást, továbbá érdemes egy flaget is elhelyezni az osztályban, ami azt jelöli, hogy a guard sikeresen végrehajtotta a zárolási műveletet. A flaget az osztály destruktorában is használjuk, így futási időben felmerülő hibáktól védhetjük meg magunkat, ha felszabadításkor figyeljük, hogy a guard tényleg maga állította-e be a zárat. A kritikus szekció védéséhez az adott programszakaszt egy hatókörbe kell zárni – abban az esetben, ha ez meg nem történt meg -, és a scope első utasításaként példányosítani kell a Guard osztályunkat, így a konstruktorban automatikusan lefut a zárolás. Amikor elhagyjuk a kritikus szakasz hatókörét, a guard objektumunk automatikusan megszűnik létezni, lefut a destruktora, és a zárolás ezzel együtt feloldódik.

Megjegyzendő, hogy a C++ szemantika miatt a zárak akkor is feloldódnak, ha a kritikus szakaszban kivételdobás történt. Az implementáció hátránya lehet, hogy explicit módon nem tudjuk feloldani a zárolást, anélkül, hogy a blokkot elhagynánk.

A *Scoped Locking* módszer esetén minden típusú lockhoz különböző guardot definiálunk, ami kevéssé hatékony, hibákhoz vezethez, illetve a programunk vagy a komponensünk memóriafelhasználását is növeli. Emiatt egy gyakran használt változata a Scoped Locking mintának a parametrikus típusa vagy polimorfikus verziója a *Strategized Locking* mintának.

Kód:

```
class Guard {
private:
Mutex *lockObj; //Mutató a szinkronizációs objektumra.
bool lockOwned; //Jelzi a lock sikeres birtokba vételét.
public:
//Konstruktor.
Guard(Mutex &lock) : lockObj(&lock), lockOwned(false) {
lockObj->Acquire();
lockOwned = true;
}
//Destruktor.
~Guard() {
```

```
//Elengedjül a lock-ot, ha a birtokba vétel sikeres volt.
if (lockOwned)
lockObj->Release();
}
};
```

Ha a *Guard* osztályt egy metódusban lokálisan megpéldányosítjuk, akkor a metódusból való visszatéréskor a *Guard* destruktora automatikusan meghívódik és elengedi a szinkronizációs objektumot

Összegzés

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.

Ha a metódus, ami a *Scoped Locking*-ot használja rekurzívan hívja önmagát, „*self-deadlock*" történik, ha a zárolás nem rekurzívan van megvalósítva. A *Thread-Safe Interface* tartalmaz megoldást a hiba kiiktatására: a minta garantálja, hogy csak interfész metódusok használják a *Scoped Locking*-ot, az implementációs metódusok nem.

A *Scoped Locking* a C++ nyelv adottságait használja, ezért nem integrálható operációs rendszerspecifikus rendszerhívások esetében; a zárak adott esetben nem szabadulnak fel automatikusan, amikor szál vagy process abortál vagy véget ér egy kritikus szekción belül. Hasonlóan, a zárak nem lesznek felszabadítva, ha a standard C longjmp() függvényt hívják, mivel a függvény nem hívja meg a C++ objektumok destruktorát.

A *Scoped Locking* minta egy olyan guard objektumot használ, ami explicit módon nem használunk a hatókörön belül, mert a destruktor csak implicit módon szabadítja fel a zárat. Sajnálatos módon néhány C++ fordító ***„statment has no effect"*** figyelmeztetést dob, amikor a guard objektumokat definiáljuk, de nem használjuk a hatókörön belül.

IV.3. Strategized Locking

A Strategized Locking tervezési minta parametrizálja a szinkronizációs mechanizmusokat, amik a komponens kritikus szakaszait védik a konkurens eléréstől.

Nagy teljesítményű webszerverek fejlesztésénél kulcskérdés a fájl cache használata, ami az URL eléréseket memóriaképekre vagy fájl elérésekre képezi le. Ha a kliens egy olyan URL-t kér le, ami a cacheben tárolódik, a webszerver a fájltartalmat azonnal a klienshez tudja továbbítani a lassabb másodlagos tároló elérése nélkül. Egy hordozható webszerver-kód fájlcache implementációjának hatékonyan kell futnia többszálú és egyszálú operációs rendszereken egyaránt. Ennek egy lehetséges megoldása több különböző fájl cache osztály implementálása. Ez a két implementáció egy komponenscsalád része, amik csak a szinkronizációs stratégiájukban különböznek. Az egyik komponens egyszálú fájlcachet implementált lockolási műveletek nélkül. A másik komponens fájlcachet implemenált, ami

zárolást használ a többszálúság miatt, hogy a konkurens cache-használatot kivédje. Eltérő implementációk karbantartása és bővítése a későbbiekben problémás lehet.

Problémák

Azoknak a komponenseknek, amik többszálú környezetben futnak, a kritikus szakaszaikat védeniük kell a konkurens klienshozzáféréstől. A szinkronizációs mechanizmusok integrálása során két tényt kell figyelembe vennünk:

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.

Megoldások

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). Példányosítsuk ezeket a csatlakoztatható típusokat a komponensen belül, ami használni tudja az objektumokat a metódusainak hatékony szinkronizálására.

A legtöbb komponensnek viszonylag egyszerű szinkronizációs igényei vannak, amik általános zárolási módszerekkel leírhatóak. Ezek a szinkronizációs technikák együttesen kezelhetőek polimorfizmust vagy paraméteres típusokat alkalmazva. Általánosságban elmondható, hogy a polimorfizmus akkor használandó, ha a zárolási technikák csak futási időben ismertek. Paraméteres típusokat akkor kell használni, ha a zárolási stratégia fordítási időben ismert.

Feltételezve, hogy a Scoped Locking mintát használjuk, a zárolási stratégia kialakítása a következő lépésekből áll. Ahhoz, hogy egy komponenst alternatív zárolási mechanizmusokkal lássunk el, a módszerek konkrét implementációinak abstract interfésszel kell rendelkezni általános leírással a zárak lefoglalására és felengedésére polimorfizmuson vagy paraméteres típusokon alapulva. A polimorfizmus módszere során egy polimorfikus objektumot hozunk létre, ami dinamikus kötésben tartalmazza a lefoglalás és zár elengedése metódusokat. Minden konkrét zárolást ebből az alaposztályból származtatunk, a metódusok felülírásával konkrét lockolási módokat hozunk létre. Ha a C++ fordító támogat default template argumentumot, akkor célszerű beállítani egy LOCK-ot az általános zárolási módhoz.

A paraméteres típussal történő megvalósítás során meg kell győződnünk arról, hogy minden konkrét zárolás ugyanazt a módszert alkalmazza a zárak foglalására és felengedésére. Általános megoldás a problémára, ha a Wrapper Facade patternt használjuk.

Miután a szinkronizációs módszereket kialakítottuk, a komponensek elérhetik ezeket a mechanizmusokat, hogy a kritikus szakaszaikat védjék. .

Kód:

```
//Az absztrakt szülő osztály.
class Lock {
public:
virtual void Acquire() = 0;
virtual void Release() = 0;
}
```

```
//Az egyszálas megvalósítás.
class SingleThreadedLock : Lock {
public:
virtual void Acquire() {}
virtual void Release() {}
}
//A többszálas megvalósítás.
class MultiThreadedLock : Lock {
private:
//A szinkronizációs objektum.
Mutex lock;
public:
virtual void Acquire() {
lock.Acquire();
}
virtual void Release() {
lock.Release();
}
}
```

Összegzés

Három előnye van a *Strategized Locking* pattern alkalmazásának. Magától értetődő adott **konkurenciamodell beállítása és testre szabása egy komponenshez**, mivel a szinkronizációs aspektusok előre eldönthetőek. **Ha nem létezik megfelelő zárolási módszer egy új konkurens modellhez, a lockolási stratégiák családja bővíthető a létező kód módosítása nélkül**. Egyszerű bővítéseket hozzáadni, illetve hibákat javítani a komponenseken, mivel minden konkurenciamodellhez csak egy implementáció tartozik. **Azokat a komponenseket, amiket a minta felhasználásával fejlesztünk kevésbé fognak függni specifikus szinkronizációs működésektől. Emiatt újrafelhasználhatóbbak**.

A *Strategized Locking* design pattern hátrányai az *obtructive locking* és az ún. *over-enginerring*.

Néhány fordító számára hátrányos lehet a polimorfikus stratégia alkalmazása, annak ellenére, hogy flexibilis a tervezhetőség szempontjából. Néhány esetben ez a „túltervezettség", a zárolási módok kiszervezése a komponenshez nehézségeket okozhat. Kevésbé gyakorlott programozók esetleg rossz zárolási típussal paraméterezik fel a komponenst, ami nem várt fordítási és futási eredményekhez vezethet. Hasonlóan, csak egy típusú szinkronizációs módszer lenne szükséges egy adott típusú komponenshez. Ebben az esetben a rugalmas működés hátrányos és szükségtelen. Általánosságban a pattern akkor a leghatékonyabb, ha a gyakorlati alkalmazás azt mutatja, hogy a komponens viselkedése ortogonális a zárolási stratégiára.

Kiegészítés

Java-ban a fő szinkronizációs módszer a monitor. A Java nyelv nem nyújt támogatást a szokásos konkurenciakezelő módszerekhez mint a mutex vagy a szemafor. Habár, lehetőség van különböző konkurencia primitívek implementálására. Pl. a util.concurrent csomag számos zárolási lehetőséget definiál. A primitívek implementációja felhasználható zárolási módszerként, hogy alkalmazás specifikus eseteket támogassunk. A paraméteres típusok támogatásának hiánya miatt csak a polimorfikus módszer terjeszthető ki Java-ra. Ebben az esetben a Java implementációk hasonlóak lesznek a C++-osokkal.

IV.4. 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. Ha thread-safe komponenseket tervezünk, akkor a fejlesztőnek figyelembe kell venni a komponensen belüli metódushívások miatti *self-deadlockot* és a szükségtelen zárolási overheadet is. (Pl a lookup() metódus meghívja az insert()-et, ami szintén lockolni akar.)

Problémák

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. Ebben az esetben a következő problémák megoldatlanok a többszálú komponensek esetében.
A thread-safe komponenseket úgy kell megtervezni, hogy elkerüljük a self-deadlockot. A thread-safe komponenseket úgy kell megterveznünk, hogy csak minimális zárolási overheadet tartalmazzanak. Pl., hogy megakadályozzuk a versenyhelyzetet. Ha rekurzív komponens zárolást választunk a self-deadlock
elkerülésére, szükségtelen overheadet jelenthet a zárak többszöri foglalása és felszabadítása komponensen belüli metódushívások között.

Megoldások

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) metódusok 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.

Kód:

```
class ThreadSaafeInterfacceClass {
private:
//A szinkrronizációt biztosító objektum.
MultiThreaadLock lock;
//A műveletet végrehajtó metódus.
long ImplementationMethod(longg data) {
//A műveletet itt hajtjuk végre...
return result;
}
public:
//Az interfész metódus.
long InterfaceMethod(long data) {
//Szinkronizáláss.
Guard guard(lock);
//A műveletet végrehajtó metódus meghívása.
```

```
return ImplementtationMethod(data);
}
}
```

Összegzés
Három előnye van a *Thread-Safe Interface* pattern alkalmazásának. A minta biztosítja, hogy selfdeadlock nem következik be komponensek közti metódushívások során. A minta garantálja, hogy zárakat nem foglalunk le és szabadítunk fel szükségtelenül a futás során. Továbbá a lockolás és a funkcionalitás elkülönítése mindkét aspektus egyszerűsítésében segíthet.
A módszer alkalmazásának négy hátránya van. Minden interfészmetódus legalább egy implementációs metódust igényel, ami megnöveli a komponenst és a futás során a metódushívások számát.
Az idióma egymagában nem oldja fel a self-deadlock problémáját. Abban az esetben, ha a kliens meghívja az A komponens interfészmetódusát, ami utána az implementációs metódusba továbbítódik, ott meghívva egy B komponens interfészmetódusát deadlock keletkezhet, ha a B metódusa meghívja az A interfészmetódusát, ami ismét zárolni próbál.
Az objektum-orientált programozási nyelvek (mint a C++, Java) osztályszintű hozzáférést biztosítanak.
Ennek eredményeként egy objektum a publikus interfészén keresztül meghívhat az osztályon belüli más objektumok privát metódusait, továbbítva a lockolási információkat is. Emiatt a fejlesztőnek figyelemmel kell lenni arra, hogy ne hívjon meg más objektumok privát metódusait az osztályon belül.
A *Thread-Safe Interface* minta megakadályozza, hogy több komponens ugyanazt a lock-ot használja.
Emiatt a szinkronizációs overhead megnő, mivel több zárat is létre kell hozni. Emiatt a deadlockok megtalálása is nehézkesebbé válik. Továbbá, a minta a zárolást a komponensszintnél alacsonyabban végzi, ami a zárolási műveletek növekedését, ezzel együtt csökkenő teljesítményt eredményez.

Kiegészítés
A *Thread-Safe Interface* pattern a *Decorator* patternhez köthető, ami kiterjeszti az objektumot dinamikus felelősségekkel. A fő különbség az, hogy a *Decorator* pattern a dinamikus kiegészítő felelősségek hozzáadására alkalmazható, míg a *Thread-Safe Interface* a metódusfelelősségek statikus elkülönítésére való a komponensosztályokban.
Azoknak a komponenseknek, amik a *Strategized Locking* pattern alapján készültek a Thread-Safe Interface megvalósítását is felhasználhatják, hogy biztosítsák, hogy a komponens robusztusan és hatékonyan fog működni a választott zárolási stratégiától függetlenül.
A Java a metódusszintű lockolást monitor objektumokon keresztül valósítja meg a *synchronized* kulcsszó segítségével. Javaban a monitorok rekurzívak. A self-deadlock problémája nem történhet meg addig, amíg ugyanazt a monitort használjuk. A lockolási overhead problémája a Java Virtual Machine típusától függ. Ha egy specifikus JVM nem hatékonyan implementálja a monitorokat, és a monitorok rekurzívan zárolódnak, a *Thread-Safe Interface* esetleg segíthet a futás idejű teljesítmény javításában.

IV.5. 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 teszik szükségessé *threadsafe* módon egyszeri alkalommal a program futása során.

A *Singleton* pattern biztosítja, hogy egy osztályból csak egy példány létezhet, és globális hozzáférést biztosít a példányhoz. Az alkalmazások a statikus instance() metódust használják, hogy lekérjék a Singletonra mutató pointert és meghívjanak publikus metódusokat ezen keresztül. A *Singleton* pattern kanonikus implementációja nem kielégítő olyan platformokon, amik preemptív multi-taszkingot vagy valós hardveres párhuzamosítást használnak. A *Singleton* konstruktor többször is hívható, ha több preemptív szál hívja az instance() metódust egyidejűleg inicializálása előtt; illetve több szál futtatja a Singleton konstruktor dinamikus inicializálását a kritikus területen belül.
Kedvezőbb esetben a konstruktor többszöri hívása memory leak-et okoz. Rosszabb esetben ellenőrizhetetlen lefutást okozhat, ha az inicializálás nem egyértelmű. Ahhoz, hogy a kritikus szekciót védjük a konkurens hozzáféréstől alkalmazhatjuk a *Scoped Locking* idiómát, hogy automatikusan lefoglaljuk és elengedjuk a mutex zárolást.
A patternt úgy alakítjuk át, hogy minden instance() hívás során lockolunk, ezzel biztosítjuk, hogy a kritikus szakasz csak egyszer fusson le. Guard beiktatásával kiküszöbölhetjük a zárolásból fakadó overheadet.
Sajnálatos módon ez a megoldás nem nyújt thread-safe inicializálást, mivel többszálú alkalmazásokban a versenyhelyzet a Singleton többszöri inicializálását eredményezheti. Pl. két szál egyszerre vizsgálja az „instance_ == 0" feltételt. Minkettő igazzal tér vissza, az egyik a guardon keresztül lockol, és a másik blokkolódik. Miután az első szál inicializálja a Singletont és elengedi a zárat, a blokkolt szál ismételten inicializálni fogja a Singletont.

Problémák
A konkurens alkalmazásoknak biztosítaniuk kell, hogy a kódjuk bizonyos része szerializáltan fusson, hogy elkerüljék a versenyhelyzetet, amikor elérnek és módosítanak megosztott erőforrásokat. Általános módja a versenyhelyzet elkerülésének zárak, mutex-ek alkalmazása. Minden szálnak, ami belép a kritikus szakaszba, először zárat kell igényelnie. Ha a zárat valamelyik más szál birtokolja, a szál várakozni kényszerül, amíg a zárat felszabadítják.
A vázolt szerializációs módszer kevésnek bizonyulhat olyan objektumok vagy komponensek számára, amik egyszeri inicializálást igényelnek. A Singleton példában 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.

Megoldások
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.
A *Double-Checked Locking Optimization* minta három lépésben implementálható. Először fel kell ismernünk azt a kritikus kódrészt, ami csak egyszer fut le. A zárolási logika implementálása során szerializáljuk a kritikus kód egyszer futtatható részét. A zárolási logika implementálásához felhasználhatjuk a *Scoped Locking* idiómát, hogy megbizonyosodjunk róla, hogy a zárolás automatikusan megtörténik, amikor a megfelelő területre ér a futás, illetve automatikusan felszabadul, ha elhagyjuk a scope-ot. A Scoped Locking idióma alapján zárolással oldjuk meg, hogy a Singleton konstruktora szerializált legyen. A zárat inicializálni kell az egyszer futtatandó kód hívása előtt. C++- ban egy lehetséges módja annak, hogy

megbizonyosodjunk a zár inicializálásáról statikus objektumként való definiálása. Sajnos azonban a C++ nyelv specifikációja nem garantálja olyan statikus objektumok inicializálásának sorrendjét, amik különböző fájlokban találhatók. Eredményeként különböző C++ fordítók és linkerek inkonzisztensen viselkedhetnek, és a zárat nem inicializálják első hozzáféréskor.
A probléma megoldására jobb módszer az *Object Lifetime Manager* pattern alkalmazása. A minta egy hordozható objektummenedzser komponenst definiál, ami a globális vagy statikus objektumok életciklusát a következőképpen szervezi: az object manager első használatukkor létrehozza az objektumokat, illetve meggyőződik arról, hogy az objektumokat megfelelően megszüntették a program befejeződésekor. Pl. a zárat az object manager felügyeletével is elhelyezhetjük, ami meggyőződik arról, hogy a zár inicializált mielőtt a singleton megkísérli használni a zárat. Az object manager hasonlóan törölheti a singletont, amikor a program leáll. Ezzel megakadályozza a memória és az erőforrások szivárgását, ami egyébként megtörténhet a Singleton patternnel.

Kód:

```
class SingletonClass {
private:
//Mutató az objektumra.
static SingletonClass *instance;
//A szinkronizációt biztosító objektum.
static MultiThreadLock lock;
public:
static SingletonClass *GetInstance() {
//Az első ellenőrzés.
if (instance == 0) {
//Szinkronizálás.
Guard guard(lock);
//A második ellenőrzés.
if (instance == 0)
instance = new SingletonClass();
}
return instance;
}
}
```

A fenti C++ nyelvű példa C# nyelven szinkronizálás használata nélkül is megvalósítható az alábbi módon:

```
public sealed class SingletonClass
{
private static SingletonClass instance = new SingletonClass();
static SingletonClass()
{
}
private SingletonClass()
{
}
public static SingletonClass Instance
{
get { return instance; }
}
}
```

Összegzés

Két előnye van a *Double-Checked Locking Optimization* minta használatának. A módszer minimalizálja a zárolási overheadet a két első belépés flag vizsgálatával. Miután a flag beállítódott az első vizsgálat meggyőződik arról, hogy szükséges-e még további zárolás. A

második vizsgálat azt ellenőrzi, hogy a kritikus szakasz csak egyszer futott-e le (versenyhelyzet).
Ennek ellenére három hátránya van a *Double-Checked-Locking Optimization* patternnek. Ha az instance_ pointert flagként olvassuk a singleton implementációban, a singleton instance_ pointert atomi műveletként kell írni és olvasni, ugyanis előfordulhat, hogy a többciklusú írás/olvasás során más szálak is hozzáférnének az instance_-hoz, ami hibás mutatóértéket eredményezhet, ez illegális memóriahozzáférésként jelentkezhet. Ilyen esetek olyan rendszerekben fordulhatnak elő, amik pl a 32- bites pointereket 16 bites word buszon továbbítják, amihez két memóriaolvasási művelet szükséges.
Ebben az esetben word flaget kell alkalmazni, hogy biztosítsuk a hardverszintű atomi műveletet.
Bizonyos multi-processzoros platformok (pl COMPAQ Alpha, Intel Itanium) erőteljesen alkalmaznak memóriacachelési optimalizációkat. Ezeken a rendszereken a *Double-Checked Locking Optimization* pattern implementálása a megfelelő módosítások nélkül nem lehetséges. A megfelelő használhatósághoz CPU-specifikus utasítások szükségesek (pl. cache flush).
Mivel CPU-specifikus utasítások szükségesek az implementáláshoz, a pattern Java alkalmazásokban nem alkalmazható.

Kiegészítés
A Double-Checked Locking Optimization minta a Lazy Evaluation pattern thread-safe változata. Ezt a mintát gyakran alkalmazzák olyan programozási nyelvekben, melyekben konstruktor használata nem támogatott, ezért a pattern alkalmazása biztosítja a komponensek inicializálását mielőtt az állapotukhoz hozzáférnénk.

V. Összefoglaló

A minták múltja, jelene és jövője

„Jósolni nehéz, kifejezetten a jövőt”
Yogi Berra, Baseball Hall of Fame filozófus

James Burke tudományfilozófus szereti megjegyezni, hogy a történelem ritkán történik a megfelelő sorrendben vagy megfelelő időben, de a történész feladata, hogy úgy nézzen ki, mintha így lenne. A Pattern-Oriented Software Architecture (POSA) könyvsorozat megírásának egyik előnye, hogy lehetőségünk nyílik a korábbi jövendöléseinket újra átnézni, összefoglalni, hogy mi is történt és megvizsgálni miért történt úgy.

Ez a fejezet átnézi az 1996-os előrejelzéseinket a POSA sorozat első kötetében megjelent „merre tartanak a minták” témában. A System of Patterns [POSA1]. Megbeszéljük a minták valós irányait, melyet az elmúlt 4 év alatt bejártak, elemezzük hogy hol is tartanak ma és felülvizsgáljuk a jövőről alkotott víziónkat utólagos bölcsességgel.

Mi történt az elmúlt 4 évben

A *System Of Patterns*ben *[POSA]* megjósoltuk, hogy szerintünk miként fognak a minták fejlődni. Most, négy évvel később újra végigmegyünk a kapcsolódó fejezet ugyanazon részen és összefoglaljuk mi is történt valójában. Idézzük a kapcsolódó munkák nagyrészét ez alatt, de a listánk nem kimerítő. További hivatkozások taláhatóak a http://hillside.net/patterns/ címen.

Minták

1996-ban úgy jósolták, hogy az elkövetkezendő években a szoftverfejlesztés kölünféle területein sok mintát fognak (újra)felfedezni és dokumentálni. Úgy reméltük, hogy ezek a minták, a mintákkal foglalkozó irodalom sok hézagját fogják betömni. Az előrejelzés ezen része mindenképpen igazzá vált, bár az irodalomban még maradtak hézagok, kijfezetetten ez elosztott, hibatűrő, tranzakciós, valósidejű és begyázatott rendszerek, középrétegek és alkalmazások körében.

A minták irodalma jelentősen növekedett az elmúlt négy évben. Például, az Egyesült Államokban és Európában tartott PLoP és EuroPLoP konferenciák virágzottak és új ágakkal bővültek, beleértve az Egyesült Államokbeli ChilliPLoP és az ausztál KoalaPLoP konferenciákat. Ezeket a konferenciákat az *„szerzői műhely”* elvre alapozták. Egy szerzői műhelyben a szerzők elolvassák egymás mintáit és mintanyelveit, majd megbeszélik előnyeiket és hátrányaikat, hogy tartalmuk és stílusuk fejlődését elősegítsék.

Az elmúlt négy évben mintákat tartalmazó könyvek jelentek meg, melyek tématerületek széles körét ölelik fel, beleértve:

- Minták szerkesztett gyűjteményei a szoftverfejlesztés számos területéről.

- Egy termékeny könyv az egészségügy és vállalati pénzügy szakterületén való analízis mintákról Martin Fowlertől.
- A *Patterns for Concurrent and Networked Objects* könyv, mely az architektúrális és tervezési mintákra koncentrál konkurrens és hálózati közesréteget tartalmazó keretrendszerek és alkalmazások számára.
- Szoftver fejlesztési minták és minták a szoftverkonfiguráció menedzsmentre.
- Minták egy meghatározott programnyelvre, mint a Smalltalk és Java, vagy egy bizonyos köztesréteg specifikusak, mint a CORBA.
- Úgynevezett „*anti-minták*", melyek olyan gyakori „*megoldásokat*" mutatnak be szoftverfejlesztési problémákra, melyek ésszerűnek tűnnek ugyan, de különféle okok miatt használhatatlanok.

Fontos megjegyezni, hogy ezek a könyvek mind céljukban, mind minőségükben különböznek. Ezen publikációk értékelése és bírálata azonban nincs ezen fejezet hatáskörében – ezt az ön ítéletére és az online értékelésekre hagyjuk, mint például amiket az amazon.com-on lehet találni. Mindazonáltal, az elmúlt négy évben született publikációk mennyisége a szoftver kutatói és fejlesztői közösségek minták irányában tanusított növekvő érdeklődését árulja el.

Többet a leginkább széles körben alkalmazott minták közül keretrendszerek dokumentálására használják vagy használták. Ezek a minták keretrendszerek absztrakt leírásának tekinthetők, melyek segítik a fejlesztőkat az ő szoftverarchitekúrájuk újrahasznosításában. A keretrendszerek más szemszögből minták megtestesüléseinek is tekinthetők, melyek lehetővé teszik a direkt tervezést és kód újrahasznosítást. A tapasztalat azt mutatja, hogy az érett keretrendszerekben a minták nagyon sűrűn találhatók meg.

Mintarendszerek és Mintanyelvek

Ahogy a minták a szoftveres közösségben egyre ismertebbé váltak, egyre növekvő mennyiségű munka foglalkozott a minta nyelvekkel. Olyan korán, mint 1995, a mintákkal foglalkozó közösség vezető szerzői kezdték el a minták gyűjteményét dokumentálni specifikus szoftverfejlesztési szakterületekre, különösen a telekommunikációra. Ezek a minták sokkal közelebbi kapcsolatban álltak, mint a korábban publikált önálló minták. Valójában, több minta olyan szorosan kapcsolódott, hogy elszigetelten nem is léteztek. A mintákat mintanyelvekbe szervezték, ahol mindegyik nyelvbeli mintára építenek és összeszövik további mintákkal.

Több publikált mintanyelv is megjelent, ahogy a jegyzetben bemutatott minták is egy mintanyelvet alkotnak, vagy precízebben fogalmazva egy mintanyelv magját adják, mellyel elosztott objektumú köztesrétegek, alkalmazások és szolgáltatások fejleszthetők. A jóslat, hogy a mintanyelveken végzett munka növekedni fog, így igazzá vált. Mégis lényeges hézagok maradtak az irodalomban, nagyrészt az átfogó mintanyelvek kidolgozásához szükséges erőfeszítés miatt.

Végeztek munkát a minták kategóriákba, mintarendszerekbe való osztályozásán. Általánosságban azonban az elmúlt négy év alatt a mintanyelvek a mintakatalógusoknál vagy mintarendszereknél sokkal népszerűbbekké váltak. A mintákkal foglalkozó közösség a mintanyelveket találta a legígéretesebb útnak szorosan kapcsolódó mintahamazok dokumentálására.

Az előző fejezetekben a bemutatott minták mintanyelv aspektusait már ismertettük. Azon kívül, hogy egy elosztott objektumú köztesrétegek és konkurrens vagy hálózati alkalmazások építésére szolgáló mintanyelv alapjait definiáljuk, a mintákat egy mintarendszerbe is rendezhetjük.

Ez a besorolási terv egy érdekes elméleti feladatot ad a mintatér rendszerezésével. Mindegyik mintát osztályozhatjuk, és egy mezőhöz rendelhetjük egy többdimenziós mátrixban, ahol a mátrix mindegyik dimenziója egy konkrét mintatulajdonságot jelöl. Ha izolált problémákat kell megoldani, ez az osztályozás adhat hasznos minta-alapú megoldásokhoz hgyors hozzáférést. A következő táblázat egy lehetséges módot mutat, ahogy a mintákat egy konkurrencia és hálózatkezelési mintarendszerré rendszerezni lehet.

	Architektúrális Minta	**Tervezési Minta**	**Idióma**
Alap architektúra	Broker Layers Microkernel		
Kommunikáció	Pipes and filters	Abstract Session Command Processor Forwarder-Receiver Observer Remote Operation Serializer	
Inicializálás		Activator Client-Dispatcher-Server Evictor Locator Object Lifetime Manager	
Szolgáltatáselérés és konfiguráció	Interceptor	Component Configurator Extension Interface Half Object plus Protocol Manager-Agent Proxy Wrapper Facade	
Eseménykezelés	Proactor	Acceptor-Connector	

	Reactor	Asynchronous Completion Token Event Notification Observer Publisher-Subscriber	
Szinkronizáció	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
Konkurrencia	Half-Sync/Half-Async Leader/Follower	Active Object Master-Slave Monitor Object Producer-Consumer Scheduler Two-Phase Termination Thread-Specific Storage	

A minták rendszezése specifikus területek vagy tulajdonságok szerint nem fejezi ki egy konkrét mintahalmazban létező kapcsolatokat és függőségeket, egy bizonyos szintig persze. Ezek a kapcsolatok és függőségek minta alkalmazhatóságát befolyásolják más minták jelenlétében, mivel nem minden minta kombinálható értelmesen tetszőleges másikkal.

Például a Thread-Specific Storage nem használható a Leader/Follower szálkészletével, mivel nem feltétlenül lesz fix asszociáció a készletben levő szálak és az időben feldolgozott események között. Általánosságban tehát fontos a „*megfelelő*" mintakombinációkat felismerni, valós életbeli rendszer tervezésekor, mert ezek a rendeszerek sok problémától függenek, amit meg kell oldani.

A nagy mintarendszerek hajlamosak a bonyolultságra, mivel minél több mintaterületet ölel föl egy mintarendszer, annál valószínűbb, hogy egy minta több kategóriához lesz beosztva. Ennek a jelenségnek a legjobb példája a Wrapper Facade, mivel használják konkurrens és hálózatkezelő rendszerek, grafikus interfészek és komponensek építésére is. Háromszor is megjelent a mintarendszerünkben. Ahogy a minták száma növekszik, a mintarendszerek tele lesznek az ilyen ismétlődő minták bejegyzéseivel, mely az olvasásukat és használatukat nehézzé teszi.

Egy lehetséges megoldás erre a problémára, hogy bizonyos szakterületekhez kisebb mintarendszereket specifikálunk, mintsem egy univerzális mintarendszert. Példának lehetne felhozni a bemutatott mintarendszert a konkurrencia és hálózatkezelésre, vagy a komponens létrehozás mintarendszerét. Ha ezt a megközelítést alkalmazzuk a könyvben bemutatott mintákra, akkor azt vesszük észre, hogy a mintarendszer struktúrálisan nagyon hasonlít a már ismertetett mintanyelvre.

Végül, az eredmény mintarendszer nem fejezi ki a minták közti kapcsolatokat olyan jól ahogy az általunk definiált mintanyelv teszi. A minták inkább önálló szigetek maradnak, és a mintarendszer így kevésbé hasznos a konkurrens és hálózati alkalmazások vagy köztesrétegek teljes szoftverrendszerének minta-alapú fejlesztésében.

A tapasztalatunk az, hogy a könyvben bemutatott minták egy mintanyelvvé rendszerezése hatásosabb, mint mintarendszerré való osztályozásuk. Sok kutatást kell még végezni, hogy az összes, a mintanyelv teljessé tételéhez szükséges mintát felismerjük, dokumentáljuk és integráljuk.

Metódusok és eszközök.

1996-ban úgy jósolták, hogy a mintákat használó különféle eljárásokon és eszközökön végzett munka növekedni fog, néhány tapasztalt fejlesztő szkeptikus véleménye ellenére. A a szoftverkutatók szünet nélkül dolgoztak a témán, eljárásokat kihozva a meglevő rendszerekben szereplő minták felismerésére, valamint számos eszközt fejlesztve a minták szoftverfejlesztésben és visszafejtésben való alkalmazásához. Az mintákat támogató automatizált szoftveres eszközök kutatása nem hatott még a szoftvereszöközök fejlesztőire, bár van néhány újkeletű eredmény. Például a minták dokumentálása az UML-be lett integrálva. Hasonlóan, eszközök tűnnek fel, melyek metamodellek segítségével leírt mintákból automatikusan generálnak kódot.

Algoritmusok és Adatstruktúrák

Növekvő érdeklődést jósoltak a minták algoritmusok, adatstruktúrák és más minták közti kapcsolatokban való alkalmazásában. Azt várták, hogy ezt az erőfeszítést majd az algoritmusok és adatstruktúrák mint minták (egy speciális mintaformátumbeli) dokumentálása kísér majd.

Mára több publikációt és könyvet írtak, mely integrálja a mintákat az egyetemi algoritmusokba és adatstruktúra tantárgyakba. Általában azonban a mintákat kutató közösség ezen munkájának célpontja inkább az egyetemi oktatás és nem a szakirodalomba való direkt hozzájárulás volt.

Minták formalizálása

Az elmúlt négy évben sok kutatást végeztek a minták formalizálásán. Ez a munka azonban nem fejtette ki direktben a hatását a mintákat kutató közösségre vagy szoftveres alkalmazásukra.

A hatás hiányára az egyik ok, hogy a minta inkább egy séma, mely kapcsolódó problémák egy halmazát oldja meg, és ezért lehetséges ismételten implementálni anélkül, hogy szükségszerűen ugyanolyan legyen minden egyes alkalommal. A formalizmusok célja ezzel ellentétben egy konkrét fogalmat leírása olyan precízen, ahogy csak lehetséges. Következésképpen a mintákkal járó variációk leírására használt a formalizmusok nagyok és bonyolultak, melyek nehezen érthetők és használhatók. Sajnos ez elveszíti a minták használatának előnyét, mely a leghatékonyabban fejleszti – és nem akadályozza – a szoftverfejlesztő csapat tagjai közti kommunikációt.

Minták helyzete ma

A mintákkat kutató közösség napjainkra jelentőset fejlődött ahhoz képest ahol négy évvel ezelőtt volt. Konkrétan, a dokumentált minták és mintanyelvek mennyisége nagyobb és diverzebb mint négy évvel ezelőtt volt - és még mindig növekszik.

Különböző csoportok a közösségben azonban más mintaparadigmákat részesítenek előnyben, beleértve:

- Cristopher Alexander elméletorientált megközelítése, minták és mintanyelvek *fentről-le* elemzése
- A Gang-of-Four és POSA által elsőnek alkalmazott *lentről-fel* mérnök-orientált megközelítés
- A *speciális-célú* mintákat javasló szerzők megközelítése, mint például az *anti-minták*

-

A minták elméleti megértésén felül, a mintáat kutató közösség jobban átlátja a minták gyakorlatbeli működését. 1996-ra a mintákat a kutatók és fejlesztők izolált csoportjai opportunista módon alkalmazták. Az elmúlt években a mintákat sokkal rendszerezettebben alkalmazták gyártás alatt levő rendszerekben, a szoftveripar szakterületeinek és témáinak széles körében, beleértve:

- Alkalmazás keretrendszerek
- Valós-idejű CORBA köztesrétegek, nagyteljesítményű webkiszolgálók, hálózatmenedzsment, konkurrencia és szinkronizáció irányítás
- Be- és kimeneti feldolgozás
- Teljesítménymérés és optimalizáció
- Szoftverbiztonság és megbízhatóság
- Gyakorlati tanulmányok, melyek a minták alkalmazásának előnyeit és hátrányait vizsgálják
- A Java programozási nyelv és osztálykönyvtárai, third-party keretrendszerei és virtuális gépének specifikációja, implementációja és fejlesztői közössége

A minták növekvő hatására további bizonyíték az a mérték, mellyel a mai fejlesztők nem csak használják őket a terveikben – amit már jóval a könyvekben való bemutatásuk előtt tettek – hanem ahogy a mintákat explicit használják a mindennapi munkájukban. A minták egy közös szótárat biztosítanak, melyet fejlesztők és kutatók tömören ötletcserékhez tervezési leírásokban, problémamegoldásokban, rendszerpecifikus tulajdonságokat betanításához és szoftverrendszerek dokumentálásához használnak.

Természetesen fontos védekezni a túlzott lelkesedés ellen, mely esetén a mintákat a fejlesztők vakon használják, ahol csak lehet. Míg a minták növelik a rendszerterv rugalmasságát és újrahasznosíthatóságát, növelhetik a költségeket. Néhány minta implementálása például több tervezési és programozási erőfeszítést igényel, vagy több kódot eredményez, amit karban kell majd tartani. Néhány minta implementálása extra kerülőutakat tartalmazhat, melyek többlet teljesítményt igényelnek bizonyos platformokon. A menedzsereknek és fejlesztőknek ezért mindig ki kell értékelniük a minták hasznát és terheit mielőtt alkalmazzák őket, kifejezetten, ha egy egyszerű osztály vagy függvény is megfelel a követelményeknek.

Minták jövője

A jövőben reméljük a minták iránti érdeklődés lényeges megnövekedését, ahogy a kutatók és főárambeli szoftver projektek továbbra is használják a minta-orientált paradigmákat, eljárásokat és folyamatokat. Ez a fejezet kifejti a víziónkat minden egyes nagyobb témakörben, melyeket a febtiekbeb értékeltünk, és körvonalazza a kulcsproblémákat, melyeket a jövő kutatásaiban előre látunk. Ugyancsak további néhány kategóriát hozzáadunk – szoftverfejlesztési folyamatok és oktatás – hogy reflektáljunk az ígéretes irányokra, melyeket a mintáat kutató közösség nem vizsgált eddig.

Minták

A mintákkal foglalkozó irodalom az elmúlt négy évben a konkrét mintákra és mintanyelvekre koncentrált legnagyobb hatással, melyek gyakran objektum orientált alkalmazási keretrendszerekből származtak. Előre nézve úgy gondoljuk, hogy a mintákat kutató közösség tovább bővíti ezt a tradíciót. A következő generációs objektum orientált alkalmazások és keretrendszerek például a mintákat explicit testesítik meg. A mintákat továbbiakban is használni fogják keretrendszerek szerkezetének és tartalmának dokumentálására. Más kulcstémák és szakterületek melyeknek a konkrét mintáknak használata hasznos, a következők:

- *Elosztott objektumok*: Sok, köztesréteggel és konkurrens vagy hálózati objektumokat tartalmazó alkalmazással kapcsolatos mintát dokumentáltak az elmúlt évtizedben, beleértve azokat, melyek ebben a jegyzetben vannak. A következő lépés az *elosztott* objektumokhoz kapcsolódó minták dokumentálása, kiterjesztve a korábbi munkákat olyan elosztott témákra, mint a távoli szolgáltatás elhelyezése, partícionálása, elnevezési és könyvtárszolgáltatások, terheléselosztás, megbízhatóság és biztonság.

  Növekvő számú elosztott objektumnak szükséges például magas szintű megbízhatóságot biztosítania a kliens programok vagy végfelhasználók felé. A CORBA Fault Tolerance specifikáció és az azt implementáló ORBok elfogadásával a

fejlesztőknek több lehetőségük van, hogy a hibatűrő elosztott objektumokkal kapcsolatos tapasztalataikat rögzítség minták formájában, mint ahogy a múltban volt.

- *Valós-idejű és elosztott rendszerek.* Növekvő számú számítástechnikai rendszer elosztott, beleértve a járművek fedélzeti rendszereit és egyéb gépjármű-központú alkalmazást, gyári automatizált felszerelés irányítási rendszereket, repülési informatikát és mobil számítástechnikai eszközöket. Ezen rendszerek nagyrésze szigorú erőforráskorlátozásoknak van kitéve, kifejezetten memóriahasználati- és időkorlátozásoknak.

A jóminőségű valós-idejű rendszerek fejlesztése továbbra is nagyon nehéz, egyfajta *'feketemágia'* marad. Viszonylag kevés mintát publikáltak ezen a területen. Úgy gondoljuk, hogy a minták mennyisége a valós-idejű és beépített rendszerek területén növekedni fog. A hajtóerő az objektumtechnológia felnőtté válása a kapcsolódó fejlesztőeszközökkel, eljárásokkal és technikákkal (beleértve a mintákat) együtt, melyek sikeressé teszik a fejlesztést és gyakrabban alkalmaznak majd ezeken a szakterületeken.

- *Mobil rendszerek*: A vezetéknélküli hálózatok mindennapivá válnak, a beépített számítástechnikai eszközök pedig egyre kisebbek, könnyebbek és több helyen alkalmazhatóak. Így, a mobil rendszerek hamarosan sokféle felhasználói kommunikációs és számítástechnikai igényt fognak kiszolgálni. A mobil rendszerek alkalmazási területei közé tartoznak a mindenhol elérhető számítástechnika, mobil ágensek, személyes asszisztens programok, pozíció-függő információellátás, távoli orvosi diagnosztika és teleradiológia, otthoni és irodai automatizálás. Ezen felül internetes szolgáltatások lesznek elérhetőek mobil rendszerekről a webböngészéstől kezdve az online banki szolgáltatásokig.

A mobil rendszerek sokféle kihívással rendelkeznek, mint például az alacsony vagy változó sávszélesség és jelerősség kezelése, a kapcsolat és szolgáltatásminőség gyakori megszakadásának kezelése, eltérő protokollok, cache konzisztencia megőrzése kapcsolat nélküli hálózati végpontok között. Úgy gondoljuk, hogy a mobilrendszer-fejlesztők dokumentálni fogják tapasztalataikat minta formában, hogy segítsék a legjobb szoftverfejlesztési praktikák terjedését ezen a területen.

- *Üzleti tranzakciós és e-kereskedelem rendszerek:* Sok üzleti információs rendszer, mint a könyvelés, raktárkezelés, számlázó rendszerek, tranzakciókon alapszanak. A tranzakciók feldolgozási szabályai komplexek, de rugalmasnak kell lenniük, hogy új üzleti gyakorlatokat vagy összeolvadásokat ki tudjanak fejezni. Az üzleti rendszereknek egyre nagyobb mennyiségű online tranzakciót kell kiszolgálniuk, ahogy azt Lead/Followers mintánál tárygaltuk.

Az e-kereskedelem weben való felemelkedése kiteszi a business-to-business rendszereket direktben a fogyasztók felé. Ezen rendszerek fontossága ellenére viszonylag keveset írtak az elemzésükről, architektúrájukról vagy tervezési mintáikról.

Úgy várjuk, hogy minták száma az e-kereskedelem és tranzakciókezelés területén növekedni fog.

- *Commercial-off-the-shelf (COTS) alapú elosztott rendszerek szolgáltatásminősége:* Az elosztott rendszereknek, mint a streaming video, internet telefónia, nagy léptékű interaktív szimulációs rendszerek, egyre szigorúbb szolgáltatásminőségi (QoS) követelményeik vannak. Kulcs QoS követelmények közé tartoznak a hálózati sávszélesség és késleltetés, CPU sebesség, memória elérési idő és energiaszint. A fejlesztési ciklus idejének és költségeinek csökkentése érdekében, az ilyen elosztott rendszereket egyre inkább többrétegű COTS hardver, operációs rendszer és köztesréteg komponensek segítségével fejlesztik.

  Történelmileg azonban a COTS-alapú rendszereket nehéz konfigurálni, hogy egyszerre *több* QoS követelménynek is megfeleljenek, mint a biztonság, időbeliség *és* hibatűrés. Ahogy a fejlesztők és integrátorok tovább sajátítják el az end-ot-end QoS garanciák biztosításának nehézségeit, lényeges, hogy dokumentálják a sikeres mintákat, hogy ezzel másokat segítsenek a független QoS tulajdonságokkal rendelkező QoS-alapú elosztott rendszerek konfigurálásában, felügyeletében és irányításában.

- *Reflektív köztesréteg:* Ez a kifejezés olyan technológiák gyűjteményét írja le, melyeket autonóm vagy félig-autonóm elosztott alkalmazások és rendszerek erőforrásainak menedzselésére és irányítására terveztek. A reflektív köztesréteg technikák lehetővé teszik az alkalmazás viselkedésének dinamikus változását; a szoftver magjának, hardver protokolljainak, szabályainak és mechanizmusainak adaptálását a többi alkalmazás vagy végfelhasználó ismeretében vagy azok nélkül. Mint az elosztott rendszerek szolgáltatásminőségénél, a minták itt is kulcsszerepet fognak játszani a legjobb eljárások dokumentálásában, hogy reflektív köztesréteg alapú szoftverek hatékony alkalmazásait elősegítsék.

- *Optimizációs elvű minták:* Sok minta a meglevő mintairodalomból a szoftverek minőségi tényezőire koncentrált a teljesítmény helyett. Ez elfogadható lehet olyan szakterületeken ahol a nem-funkcionális követelmények, mint használhatóség vagy kiterjeszhetőség mindenek felett áll, más szakterületek – kifejezetten az elosztott és beágyazott valós-idejű rendszerek – a hatékonyságot, skálázhatóságot, megjósolhatóságot és megbízhatóságot tartják legfontosabbnak a többi szoftver tulajdonság közül.

  Ezeken a szakterületeken tehát növekvő odafigyelést fogunk látni az optimizációs elvű mintákra, melyek a komplex szoftverrendszerek optimizációs szabályait dokumentálják. Mégis, a mintáat kutató közösségben van némi vita arról, hogy az optimizációs elvű minták valódi minták vagy csak elvek.

A fent ismertetett mintabányászati tevékenységen túl, növekvő számú publikációt fogunk látni melyek szerkesztett gyűjteményei, egy szakterülethez kapcsolódó konkrét mintáknak.

Például Linda Rising szerkesztett az IEEE Communication Magazine egy különszámát, és egy szerkesztett könyvgyűjteményt, mely telekommunikációs szakterület mintáival és mintanyelveivel foglalkozik.

Mintanyelvek

Az egyedülálló minták dokumentálásán túl látni fogjuk a minták rendszerezését teljes mintanyelvekké. Valószínüsítjük továbbá, hogy meglevő egyedülálló mintákat vagy összetett mintákat integrálják meglevő vagy új mintanyelvekbe.

Azonban, ahogy egyre több tapasztalat gyűlik ösze nagy architektúrális minták implementálásából, mint a Reactor vagy Proactor, valószínüsítjük, hogy kisebb minták, mint az 'Event Demultiplexor' vagy 'Continuation-passing Event-driven System' fognak feltűnni. Ezek minták elősegítik a nagy architekturális minták realizálást kisebb mintákra való szétbontásukkal. Ez a szétbontás egy másik út a mintanyelvek definiálására az olyan rendszerekhez, melyek strutúráját az éppen szétbontott architektúrális minták definiálják.

A mintanyelvek trendje két erő összefolyásából származik:

- Minnél több minta van dokumentálva, egyes szakterületek mintákkal való fedése fokozatosan egyre teljesebb
- Ahogy a minták egy szakterületen belül teljessé válnak, valószínű, hogy ezek a minták a szakterületekben másik mintákra építenek vagy függnek tőlük.

Ez az összedolgozás oka annak, hogy egy szakterület újonnan dokumentált mintái általában a meglevő minták hibáit vagy alproblémáit oldják meg. Ahogy ezek a kapcsolatok nyilvánvalóvá válnak, a szakterülethez tartozó számos mintát össze lehet kapcsolni, hogy egy mintanyelvet alkossanak. A mintákról a mintanyelvekre való váltás ambíciózus és nem triviális. A fő akadály a mintanyelvek növekedésének a dokumentálásukhoz szükséges erőfeszítés mértéke. A mintanyelveket könnyebb használni és alkalmazni, mint írni, mert a szerzőinek kell:

- szakértőknek lennie az adott szakterület technikai témáiban
- nagymennyiségű időt a tapasztalataik érthető formában való dokumentálására szánni, ami a nyelvbeli összes mintát azonosítja és az egyes mintákat a nyelvbe integrálja.

Szerencsére az egyedülálló és összetett mintákat a mintákat kutató közösség egyre inkább dokumentálja, legtöbbször gyakori keretrendszerek alapján. Eredményképpen egyre több anyag válik elérhetővé, hogy mintanyelvekké szőjék őket, enyhítve ezzel a mindent a semmiből való létrehozás szükségét.

Szerzők új generációja emelkedik ki a konferenciák különféle szerzői műhelyeiből. Idővel reméljük, hogy ezek a szerzők kitöltik a kulcs hézagokat a meglevő mintairodalomban.

Tapasztalatjelentések, metódusok és eszközök

Ahogy megjegyeztük az előző fejezetben, sok tanulmányt tettek közzé mostanra, a minták ipari szoftverfejlesztési tapaszalataiból. Úgy gondoljuk, hogy az ilyen tanulmányok száma növekedni fog, kifejezetten az olyan szakterületeken, ahol efféle tanulmányok még nem elérhetőek manapság, beleértve az elektromos orvosi képfeldolgozást, valósidejű repülésirányítást és globális internet e-kereskedelmi rendeszereket.

A dokumentált tapasztalat ilyen formájú összegyűlése lehetővé teszi kutatóknak és fejlesztőknek, hogy általános, szakterületfüggetlen alapelveket állapítsanak meg, melyek a minták használatát hatékonnyá teszik. Ezen elvek nagyja csak *implicit* lett korábbi projektekben megértve és alkalmazva. Megfelelő mennyiségű elérhető tapasztalati tanulmány mellett valószínüsítjük, hogy a mintákkal foglalkozó közösség letisztítja és dokumentálja ezeket a központi elveket *explicit* hogy elősegítse a fejlesztők képességét a minták sikeres alkalmazására.

Ugyancsak valószínüsítjük, hogy ezek az alapvető elvek össze lesznek szőve, hogy egy *'eljárás'* vagy elvhalmazt alkossanak a minta-alapú szoftverfejlesztésben. Ez az eljárás vagy elvhalmaz kiegészíti és teljessé teszi a meglevő szoftverfejlesztési eljárásokat, mint például a Unified Software Development processt vagy dokumentációs metódusokat mint az UML. Az UML modellek manapság nagyrészt azt a szoftertervek *hol* és *mit*-jét adják meg. A holnap minta-orientált metódusai és elvei abban is segítenek, hogy megindokoljuk, hogy egyes modellek *miért* alkalmasabbak az adott szakterület kulcsproblémájának megoldására.

Úgy hisszük, hogy a legsikeresebb minta-orientált metódusok és elvek *'lentről-fel'* készülnek majd el, a mintákat haszáló szakértő felhasználók és szoftvertervezők kollektív tapasztalatából általánosítva. Ennek a következtető folyamatnak nagyobb esélye van a sikerre, mint olyan megközelítéseknek, melyek a minta-orientált metódusokat és elveket *'fentről-le'* próbálják definiálni. Hasonlóan, ha a szoftverfejlesztésben a tapasztalat-központú eljárások és elvek elérhetővé válnak, úgy reméljük, hogy azok implementálva lesznek szoftervejlesztőeszközökben. Az ilyen eszközöknek nagyobb elfogadottságnak kéne örülniük, mint a jelenben, mert ők jobban automatizálják a tapasztalatból származó bevált gyakorlatot.

Minták Dokumentációja

Úgy valószínüsítjük, hogy a könyvek maradnak a fő médiája a mintákkal kapcsolatos konkrét információ terjesztésének. Azonban reméljük a web egyre nagyobb használatát, valamint web-alapú protokollokat és eszközöket, mint a HTML vagy XML, melyek mekönnyítik a minták dokumentálását és mintanyelvekbe való kapcsolását. A korábbi erőfeszítéseknek voltak sikereik izolált helyzetekben, mint a Portland Pattern Repository Ward Cunningham WikiWikiWeb-jében. The korszerű eljárások ezen a területen fejlődhetnek az alábbi három trendnek köszönhetően:

- a webelérés általánosá válása
- az olcsó és jóminőségű elektromos könyvolvasók felemelkedése
- hatékonyabb e-kereskedelmi eljárások a szerzők jutalmazására

Minták Formalizálása és Mintanyelvek

A minták fogalmának formalizálásán végzett kutatások biztosan folytatódni fognak, és olyan formalizmusokkal lesznek kiterjesztve, melyek segítik az átfogó mintanyelvek leírását. Nem világos azonban, hogy ennek a munkának nagyobb hatása lesz-e a jövőben, mint amilyen most van. Azt valószínüsítjük, hogy hasznos munkát fognak végezni a minták egy adott rendszer vagy kontextusbeli konkrét példányainak formalizálásában, hogy dokumentáljanak *specifikus* minta implementációkat.

Bizonyos formális módszerek, mint a design-by-contract, segíthet, hogy elkerüljük a minták nem megfelelő implementációját egy minta alkalmazása során. A minta példányok formalizálásának így gyakorlati haszna is lehet a szoftverfejlesztésben, hogy növelje a rendszer tervek és implementáció minőségét.

Szoftverfejlesztési eljárások és szerveződések

Sok munka, melyet a mintanyelveken végeztek, a szoftverfejlesztési eljárások és szerveződések tökéletesítésére koncentrált. Néhány szoftverfejlesztő csapat alkalmazta ezeket a mintákat, hogy munkahelyük minőségét és produktivitásukat növelje. Ez a munka ugyancsak hatással van az akadémiai szoftverfejlesztési eljárásokat kutató közösségre. Valószínüsítjük, hogy több integrációt fogunk látni a minták és a szoftverfejlesztési eljárások és szerveződések között az elkövetkezendő években.

A minták különösen hasznosnak tűnnek ’lightweight’ szoftverfejlesztésie eljárásokban, mint például a nyílt-forráskódú vagy eXtreme Programming (XP) folyamatok. Ezen felül a refaktorizációs folyamatok is sikeresen alkalmaznak mintákat, ami elősegíti a meglevő kód újratervezését és fejleszti annak modularitását, karbantarthatóságát és újrahasznosíthatóságát.

Oktatás

1996-ban nem látták előre azt a jelentős erőfeszítést, ami az úgynevezett pedagógiai minták dokumentálásán végbement. A pedagógiai minták célja emberek hatákony oktatási elveinek rögzítése. Ezen felül, a mintákat már széles körben használják szoftvertervezési és programozási egyetemi kurzusokon. Használják szoftver szakértők képzésében és újraképzésében is. Rövidtávon azt várjuk, hogy sok egyetemi kurzus fogja használni és alkalmazni a mintákat, hogy kulcsfogalmakat adjon át a szoftvertervezés bevált gyakorlatából és helytálló elveiből. Nagymennyiségű tapasztalat gyűlt így össze a minták sikeres alkalmazásáról. Az akadémiai intézetek remélhetőleg rendszerezik ezt a tapasztalatot, hogy segítsék a szoftverfejlesző szakérők következő generációjának oktatását a minták alkalmazására és azok hatékony dokumentálására. Amikor ezek a diákok végeznek és elkezdik szakmai karrierjüket, a minták még jobban általánosak lesznek, mint manapság.

Hosszútávú víziónk

Távolabbra tekintve, úgy hisszük, hogy a minták, mintanyelvek és keretrendszer komponensek ismerete végül egy olyan pontig fejlődik, ahol a szoftverfejlesztők olyan eljárásokkal és eszközökkel rendelkeznek majd, mint a biológusok. Kísérletezés és modellezés évszázadai után, a biológusok teljesen vagy részlegesen megfejtették néhány organizmus genetikus kódját. Eme tudás által vezetve, kezdik megismerni a DNS szekvenciákban található azon alapvető elemeket, melyek különböző organizmusokban egyformák, és azokat melyek egyedivé teszik őket.

A biológusok ezt az információt alkalmazzák, hogy a DNS manipulációjára úgy technikákat fejlesszenek ki, gyógymódot találjanak az öröklődő állapotokra, mint például a cisztás fibrózis. Ilyen kontextusban a tudósok:

- a DNS szekvencia alapvető tulajdonságait fedezik fel
- új technikákat hoznak létre, hogy ezeket a szekvenciákat biztonságosan és etikusan manipulálják

Mind a biológiai és szoftveres rendszerek nagyon komplexek. Könnyebben megérthetőek és irányíthatóak, ha alapvető komponenseikre szétbontjuk őket, mint a DNS szekvenciák, gének vagy minták, mintanyelvek. Úgy hisszük, hogy amint a központi szoftver mintákat és mintanyelveket megértik és újrahasznosítható komponensek és keretrenszerek formájában megtestesítik, nagyléptékű komplex szotfverrendszereket lehet majd sokkal megjósolhatóbban fejleszteni.

Úgy gondoljuk, hogy az minták és mintanyelvek általános kutatása és fejlesztése végül az alap szoftver tulajdonságokban átütő felfedezésekhez fog vezetni. Cserébe ezek a *'szoftver DNS'* felfedezések új eljárások, folyamatok, eszközök, komponensek, architektúrák és nyelvek feltalálását fogják ösztönözni. Ezek a termékek lehetővé teszik számunkra, hogy a nagyléptékű szoftverek komplexitását a mai technológiáknál sokkal hatékonyabban lehessen megtervezni.

Konkurrencián és hálózatokon túl

A korábbi fejezetekben megmutattuk, hogy a bemutatott minták miként definiálják egy mintanyelv alapjait, melyet elosztott objektumú köztesréteg, és konkurrens vagy hálózati alkalmazásokra lehet használni. Míg ez a mintanyelv kihangsúlyozza ezen minták használatát egy konkrét szakterületen, sok mintát lehet ezen kívül is alkalmazni.

A minták *Probléma* és *Ismert problémák* részének elemzése megmutatja, hogy a minták hatóköre sok esetben szélesebb, mint amit ez a jegyzet implikál. Néhány minta, például a Wrapper Facade, átalánosan alkalmazható akármilyen szakterületen, ahol szükséges objektum orientált osztályinterfészekkel egységbe zárni az önálló funkciókat és az adatokat. Más mintákat különféle rendszerekben megjelenő tipikus problématípusokra alkalmazunk. az Extension Interface például egy többszerepű komponens által biztosított funkcionalitáshoz való kiterjeszthető elérés tervezését mutatja meg.

Ebben a fejezetben tehát körvonalazunk szakterületeket a konkurrencia és hálózaton témakörén túl, melyekre a bemutatott minták alkalmazhatóak lehetnek.

Grafikus felhasználói felület

Több bemutatott mintát használnak grafikus felhasználói felületek tervezésére és implementálására:

- A Wrapper Facade-et gyakran használják a különböző GUI könyvtárak implementációs részleteinek az alkalmazás fejleszői elől való elrejtéséhez. A GUI

könyvtárakban való alkalmazásának két kiemelkedő példája a Microsoft Foundation Classes (MFC) és a Java Swing Library

- A Reactor minta variánsait haszálták, hogy grafikus felhasználói felületek eseménykezelését megszervezzék. Az Interviews Dispatcher keretrendszer implementálja például a Reaktor mintát, ahol az alkalmazás központi eseményciklusát definiálja és menedzseli a kapcsolatait egy vagy több fizikai GUI kijelzővel. A Reaktor mintát használja az Xt toolkit és az X Windows disztribúcióban.

Komponensek

Több mintát használunk a komponensek és komponens-alapú fejlesztés kontextusában:

- A Wrapper Facade minta specifikálja alacsonyszintű komponensek összefüggő kollekciójának implementálását és különféle környezetekben való alkalmazását, mint például szálkezelő vagy processzközötti kommunikációra használt komponensekben.

- A Component Configurator minta támogatja a komponens implementációk dinamikus konfigurálását és újrakonfigurálását. Azon kívül, hogy az operációs rendszerek eszközmeghajtóinak dinamikus installácójának alapjául szolgál, ugyancsak ez a minta az alapja a Java appletek dinamikus letöltésének és konfigurálásának.

- Az Interceptor minta ismerteti a kibővíthető komponensek és alkalmzások építésének mechanizmusát. A *Ismert használatok* rész listázza az általános komponens modelleket, melyek alkalmazzák ezt a mintát, mint a Microsoft Component Object Modellje (COM), Enterprise Java Beans (EJB) és a Corba Component Model (CCM).

- Az Extension Interface minta definiál egy általános mechanizmust komponensek tervezésére és klienseik számára a szolgátatásaik elérésére. Minden köznapi komponens szabvány, mint a COM, EJB vagy CCM implementálja ezen mintának egy variánsát.

Általános Programozás

Néhány minta vagy idióma általános programozásban is alkalmazható:

- A Scoped Locking egy specializációja az általános biztonságos erőforrás beszerzés és elengedés C++ programozási technikájának. Ez a technika, egy általánosabb kontextusban bemutatva ismert az 'Object-Construction-is-Resource-Acquisition' néven.

- A Double-Checked Locking Optimization védhet kódot, amit csak egyszer lehet futtatni, például az inicializálási kódot.

Összegezve, hét különböző a fentiekben tárgyalt minta és idióma – Wrapper Facade, Reactor, Component Configurator, Interceptor, Extension Interface, Scoped Locking és a Double-

Checked Locking Optimization – nyilvánvalóan alkalmazható a konkurrencia és hálózatkezelés szakterületén kívül. Ha analizáljuk a jól-tervezett szoftverrendszereket, minden bizonnyal találunk más szakterületeket is, ahol ezek vagy más minták alkalmazhatóak. Bár a jegyzetben a mintákat elsősorban konkurrens és hálózatkezelő rendszerek fejlesztésének kontextusában mutattuk be, fontos észrevenni, hogy ezek a minták megoldhatják más szakterületek visszatérő problémáit.

Egy elbúcsúzó gondolat a jövendöléséről

Ahogy a baseball *Hall of Fame* rezidens filozófusa, Yogi Berra olyan ékesszólóan mondta: „*jósolni nehéz, kifezetten ha a jövőről van szó'*. Nem minden jóslat, amit 1996-ban tettek vált valóra, különösen a minta-orientált szoftvereszközök széleskörű elterjedéséről. Felülvizsgált előrejelzésünk egy egészséges adag bizonytalanságot tartalmaz. Előrejelzésünk kifejezetten az alábbiakra lett alapozva:

- ismereteink a mintáat kutató közösségről
- a közösség folyó és tervezett érdekeltségei és kutatási tevékenységük – már amenyire ismerjük őket és
- a jövőbeli kutatási irányok, melyeket gyümölcsözőnek tartunk, vagy mi vezetünk

Úgy gondoljuk hogy előrejelzéseink nagy része valósággá válik. A minták és mintanyelvek jövője olyan irányokat is vehet, mint például a minta-alapú modell ellenőrzés, melyeket ma nem gondolunk. Vegyék ezeket az előrejelzéseket úgy, mint egy lehetséges víziót a mintákról és jövőjükről folytatott sok dialógus között, de nem, mint egy tévedhetetlen jós próféciáját.

Irodalom

„Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, Volume 2" A könyv a John Wiley & Sons kiadónál jelent meg 2000-ben.