

Modele de proiectare a aplicațiilor de întreprindere

Cursul 6 - 7

Sumar

- Aplicații concurente
- Modele de proiectare pentru aplicații concurente

Aplicații concurente

- Fire multiple de execuție
- Execuție asincronă
- Partajarea resurselor între mai multe fire de execuție sau procese
- Sincronizare
- Planificare
- Gestiunea evenimentelor

Probleme uzuale

- Resurse partajate
 - Blocaje (deadlock)
- Ordinea operațiilor
 - Dependențe între operații

Concepte specifice

- Proces
- Fir de execuție
- Resurse partajate
- Secțiune critică
- Resurse critice
- Sincronizare execuției
- Excludere mutuală
 - Semafoare
 - Mutex (Semafor binar)
- Variabile condiție
 - blocarea firului curent
 - notificare deblocare
- Blocare/Impas (Deadlock)
- Planificarea execuției

Java

- **ExecutorService**
 - Multime de fire de execuție
 - Metode execute(), shutdown()
- **Executors.newFixedThreadPool(n) -> ExecutorService**
- **wait, notify, notifyAll**
 - Firul curent va aștepta (wait) pînă când este obiectul este notificat (notify, notifyAll)
- **volatile**
 - garanteaza vizibilitatea unui obiect între fire de executie
- **Lock/ReentrantLock**
 - lock(), unlock()
- **Condition**
 - boolean awaitUntil(Date deadline)throws InterruptedException
 - Firul curent va astepta pînă când acesta este semnalat sau întrerupt, sau se ajunge la termenul specificat
 - awaitUninterruptibly()
 - Firul curent va aștepta pînă când acesta este semnalat

C#

- Thread
- Monitor
- Lock

Modele de proiectare pentru aplicații concurente

- Double-Checked Locking
- Single Threaded Execution
- Lock Object
- Scheduler
- Future
- Active Object
- Half-Sync/Half-Async
- Monitor Object
- Producer/Consumer
- Two-Phase Terminator
- Double Buffering
- Balking
- Guarded Suspension
- Reactor
- Read Write Lock
- Thread Pool
- Thread-Specific Storage
- Asynchronous Processing
- Leader/Followers

Double Checked Locking

Problema

- Există un acces concurent în crearea de obiecte
 - În cazul în care se dorește crearea unei singure instanță a aceleiași clase (de exemplu, modelul Singleton), poate nu este suficient să se realizeze o singură verificare pentru existența instanței, atunci cînd există mai multe fire de execuție
- Există un acces concurent la o metodă
 - Comportamentul acesteia se modifică în funcție de anumite constrângeri, care se modifică în cadrul acestei metode

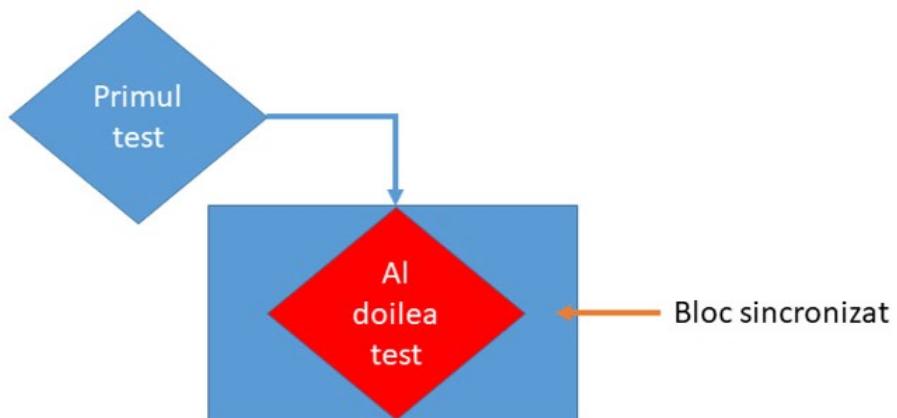
Scop

- Reducerea efortului de obținere a unei blocări, verificînd mai întâi criteriul de blocare, fără a obține, de fapt, blocarea
- Dacă verificarea criteriului de blocare indică faptul că este necesară blocarea, logica reală de blocare va continua

Implementare

- Utilizare la inițializarea întîrziată (singleton)
- Sincronizarea se realizează la nivel de bloc și nu de metodă

Diagrama



Single Threaded Execution

Problema

- Accesul la date/resurse partajate conduce la rezultate eronate în cazul apelurilor concurențiale
- Prevenirea apelurilor concurențiale la resurse sau la date
- Exemplu
 - Accesul la datele membre ale unui obiect

Scop

- Asigurarea că operațiile care nu se pot realiza corect într-un context concurențial nu se execută astfel
- Sincronizarea unor secvențe de cod care nu pot fi executate în mod concurrent

Diagrama de clase

ResursaPartajata	
metodaSigura1 metodaSigura2 metodaSigura3 metodaNesigura1 {guarded} metodaNesigura2 {guarded}	<ul style="list-style-type: none">• metodaSigura<ul style="list-style-type: none">• poate fi apelată în siguranță din mai multe fire de execuție• metodaNesigura<ul style="list-style-type: none">• nu poate fi apelată în siguranță din mai multe fire de execuție• se permite un singur apel la un moment dat

Implementare

- Declararea metodelor ca fiind sincronizate
- Utilizarea altor mecanisme specifice de sincronizare

Dezavantaje

- Performanțele codului se pot degrada
- Pot apărea situații de blocaj (deadlock)

Lock Object

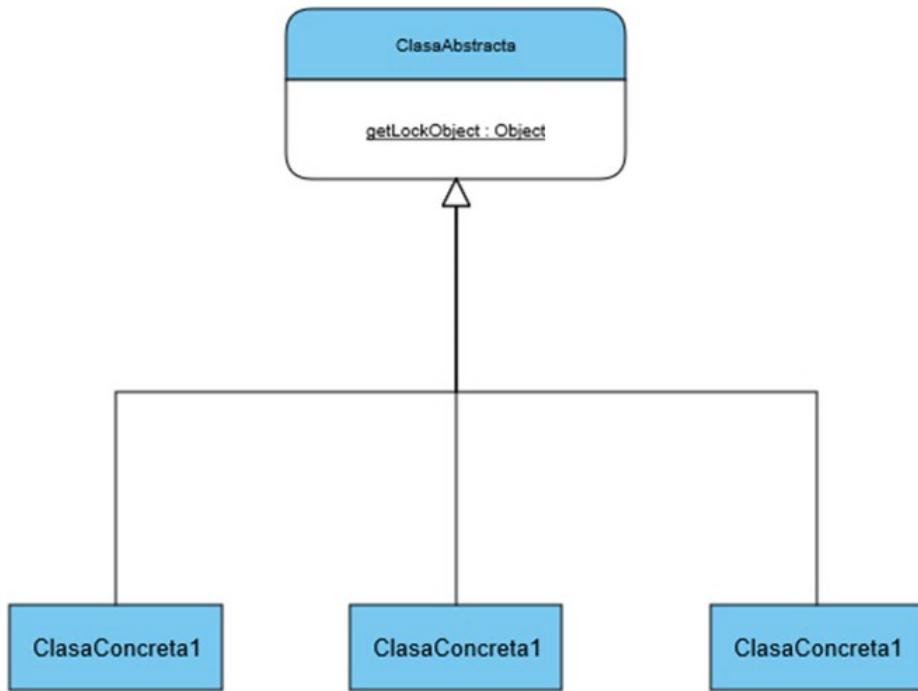
Problema

- Accesul la date/resurse partajate conduce la rezultate eronate în cazul apelurilor concurențiale
- Prevenirea apelurilor concurențiale
- Blocarea unui set arbitrar de obiecte crește complexitatea și scade eficiența

Scop

- Accesul exclusiv al unui fir de execuție la mai multe obiecte
- Utilizarea unui singur obiect care permite blocarea accesului
- Rafinare a modelului Single Threaded Execution

Diagrama de clase



Future

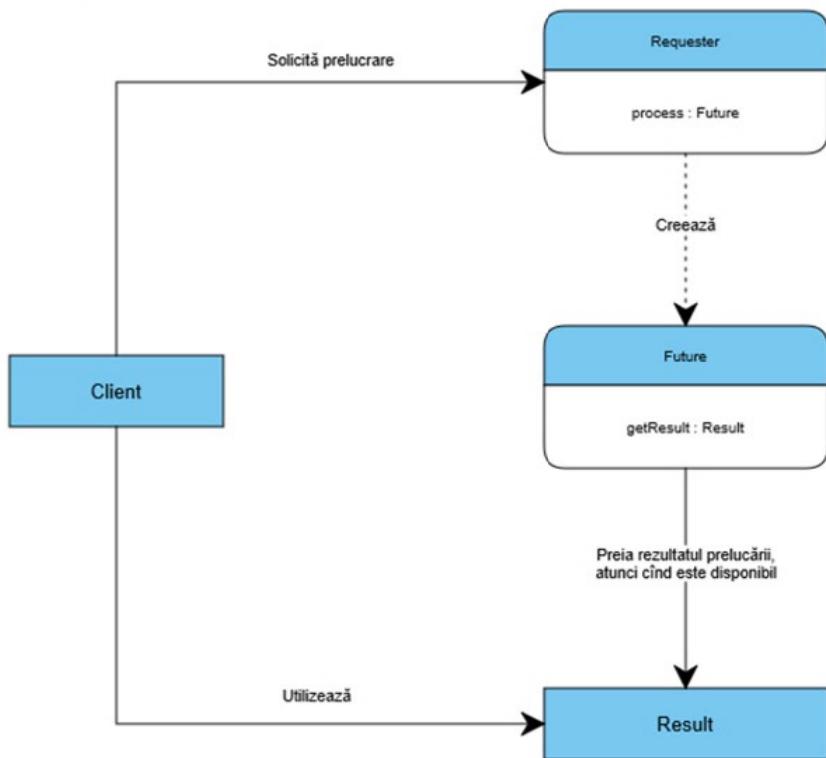
Problema

- Efectuarea unor prelucrări în alt fir de execuție față de cel care utilizează rezultatele
- Necesitatea preluării rezultatelor după ce obținerea acestora, independent de modul de lucru (sincron sau asincron)
- Utilizarea unui model de tip **Observer** ar putea fi implementat, dar cu acces dintr-un singur fir de execuție
 - Creștea complexitățea codului

Scop

- Încapsularea rezultatului unei prelucrări și punerea la dispoziție a unui mecanism care să ascundă modul de prelucrare (sincron/asincron)
- Rezultatul este accesibil prin intermediul unei metode dedicate
 - Dacă rezultatul este disponibil, va fi returnat imediat
 - Dacă rezultatul nu este disponibil, se aștepta pînă va fi deveni disponibil

Diagrama



Implementare

• **Result**

- Încapsulează rezultatul unei prelucrări

• **Client**

- Inițiază procesul de efectuare a prelucrărilor
- Preia rezultatul

• **Requester**

- Inițiază prelucrările
- Returnează un obiect de tip **Future**

• **Future**

- Încapsulează rezultatul unui obiect de tip **Result**

Scheduler

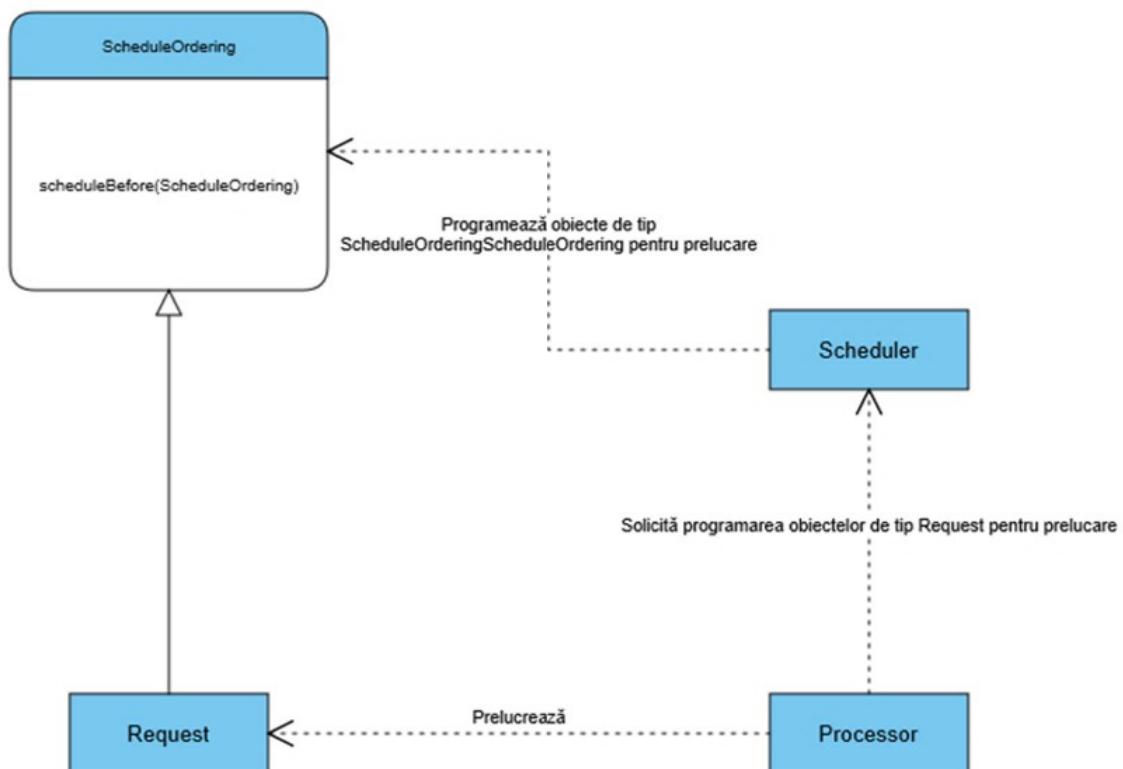
Problema

- La un moment dat, mai multe fire de execuție trebuie să acceseze o resursă, dar doar un singur fir de execuție poate accesa acea resursă
- Există restricții cu privire la ordinea în care firele de execuție pot accesa o resursă

Scop

- Furnizarea unui mecanism generic pentru programarea firelor de execuție
- Controlează ordinea de în care se execută firele

Diagrama de clase



Implementare

- **SchedulerOrdering**

- Interfață care permite stabilirea modalității de selecție a firului care va fi executat

- **Request**

- Încapsulează o cerere ce va fi prelucrată de un obiect de tip **Processor**
 - Implementează interfața **SchedulerOrdering**

- **Processor**

- Efectuează prelucrările reprezentate de obiectele de tip **Request**
 - Este prelucrat un singur obiect la un moment dat
 - Deleagă responsabilitatea programării prelucrării unui obiect de tip **Scheduler**

- **Scheduler**

- Clasă dedicată planificării procesării obiectelor de tip **Request** de către obiectele de tip **Processor**

Active Object

Problema

- Metodele invocate pe un obiect în același timp nu ar trebui să blocheze întregul proces pentru a preveni degradarea calității serviciilor oferite de alte metode
- Accesul sincron la obiectele partajate trebuie să fie simplu
- Aplicațiile trebuie concepute pentru a utiliza în mod transparent paralelismul disponibil pe o platformă hardware/software

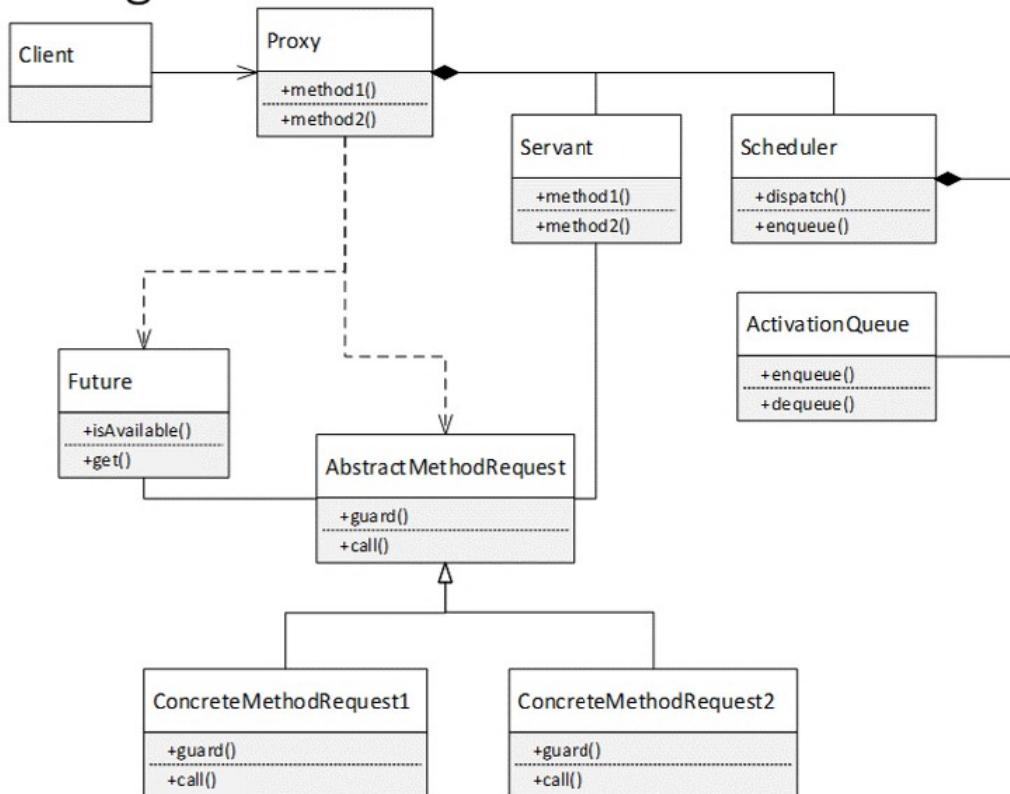
Scop

- Decuplarea execuției unei metode de invocarea acesteia
- Îmbunătățirea concurenței
- Simplificarea accesului la obiecte
- Se pot utiliza pentru implementare modelele Future și Asynchronous Processing

Implementare

- Clientul apelează o metodă prin intermediul unui obiect de tip proxy
- Dacă se așteaptă un rezultat, obiectul de tip proxy creează un obiect, asociat acestuia
- Obiectul de tip proxy creează o cerere către metoda care urmează să fie apelată
- Pentru a evita condițiile de concurență, cererile primite de la clienți sunt plasate în coadă și manipulate de un planificator
- Planificatorul alege un obiect în așteptare și îl lansează în execuție
- Este responsabilitatea obiectului să știe ce să facă atunci când se invocă
- Dacă metoda returnează un obiect, acesta este transmis către client
- Uzual, se utilizează transmiterea de mesaje

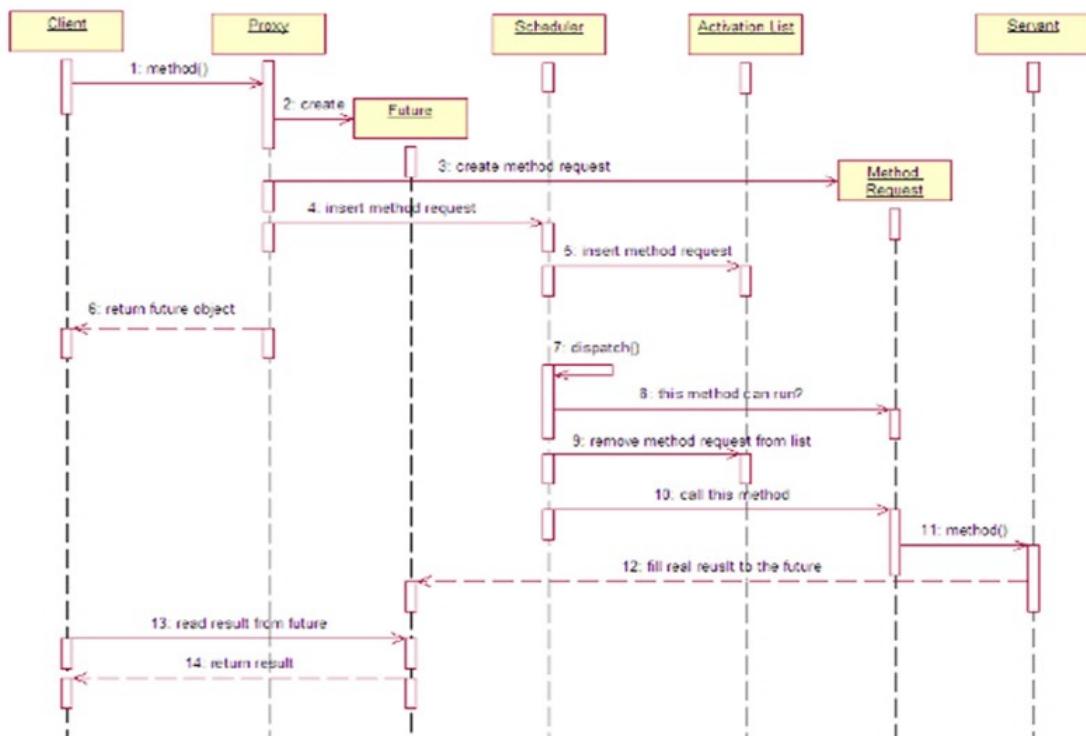
Diagrama de clase



Componente

- **Proxy**
 - Interfață care permite accesul clientilor la un obiect
- **AbstractMethodRequest, ConcreteMethodRequest**
 - Declararea și implementarea metodelor apelate prin intermediul obiectului de tip Proxy
- **Servant**
 - Implementează cererile de metode
- **Scheduler**
 - Recepționează cererile de apel la metode și le repartizează către Servant în momentul în care sunt disponibile
- **ActivationQueue**
 - Gestionează lista de așteptare a cererilor de metode
- **Future/Callback**
 - Permite accesul clientilor la rezultatele metodelor apelate prin intermediul obiectului de tip Proxy

Diagrama de stare



Avantaje

- Reducerea complexității codului:
 - Odată ce modelul este implementat, codul poate fi tratat ca un singur fir
- Nu este nevoie de sincronizare suplimentară
 - Solicitările simultane sunt serializate și manipulate de un singur fir intern

Dezavantaje

- Impactul asupra performanțelor
 - Planificarea apelurilor și gestiunea cererilor pot fi costisitoare din punct de vedere al memoriei
 - Contextul non-trivial se poate modifica
- Impactul asupra codului
 - Necesară crearea unui framework redus
 - Există mai multe componente (Proxy, Scheduler, Servant etc.)

Monitor Object

Problema

- Multe aplicații conțin obiecte ale căror metode sunt invocate simultan de clienți mulți
- Aceste metode modifică adesea starea obiectelor lor
- Pentru ca astfel aplicații concomitente să se execute corect este necesară sincronizarea și planificarea accesului la obiecte

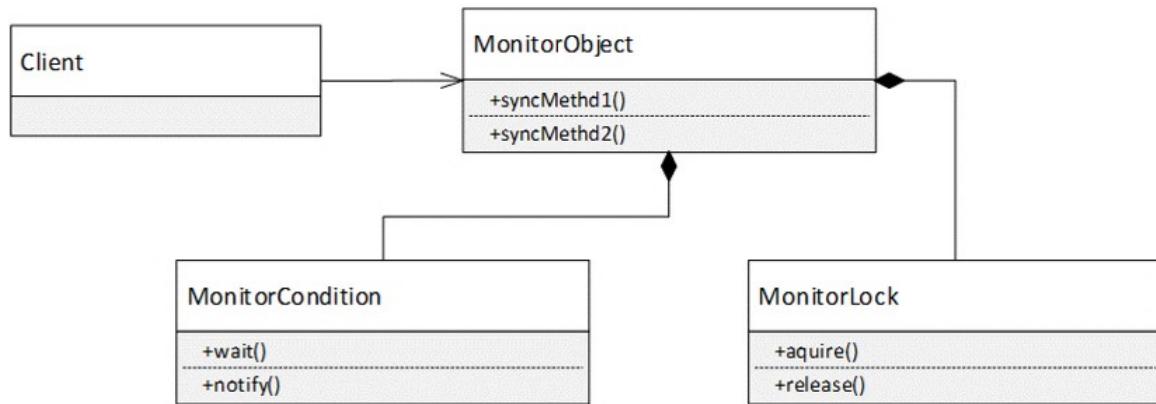
Scop

- Sincronizarea executării concurențială a metodelor pentru a se asigura că în cadrul unui obiect se execută o singură metodă
- Permite metodelor unui obiect de a programa în mod cooperativ secvențele de execuție ale acestora

Implementare

- Se definește un obiect responsabil cu sincronizarea
- Se asociază un obiect de blocare
- Se implementează un mecanism de suspendare și reluare a execuției
- Spre deosebire de modelul de proiectare *Active Object*, în cazul *Monitor Object*, nu există un fir separat de control
- Fiecare solicitare primită este executată în firul de control al clientului propriu-zis iar, până la revenirea metodei, accesul este blocat
- La un singur interval de timp, o singură metodă sincronizată poate fi executată într-un singur monitor

Diagrama de clase



Componente

- **MonitorObject**

- Definește obiectul care este accesat concurrent

- **SynchronizedMethod** (`syncMethd1()`, `syncMethd2()` etc.)

- Implementează metodele care disponibile în cadrul obiectului accesat concurrent

- **MonitorLock**

- Asigură că o singură metodă va fi executată la un moment dat

- **MonitorCondition**

- Stabilește circumstanțele în care se execută metodele sincronizate

Avantaje

- La un moment dat, doar o singură metodă sincronizată rulează în cadrul unui obiect
- Separarea blocării (realizată la nivel scăzut) de implementarea sincronizării
- Capacitatea de a suspenda și de a relua executarea în cadrul unei metode
- Mecanism de blocare bine cuplat pentru o performanță sporită

Dezavantaje

- Apelarea lui *wait* într-o stare instabilă
- Nu este eliberată blocarea atunci când apare o excepție
- Nu se realizează sincronizarea metodei când este necesar

Producer Consumer

Problema

- Decuplarea sistemul prin separarea efortului în două procese: producere și consum
- Abordează problema diferitelor durate necesare pentru a produce un rezultat sau a consuma o resursă

Scop

- Model clasic de concurență
- Reduce legătura dintre producător și consumator prin separarea identificării de executarea operațiilor
- Permite producerea secvențială și prelucrarea obiectelor, în mod coordonat, într-o aplicație concurrentă

Implementare

- Producătorul
 - Generează elemente
 - Stochează elementele într-o structură partajată
- Consumatorul
 - Preia elementele din structura partajată
 - Utilizează elementele

Implementare

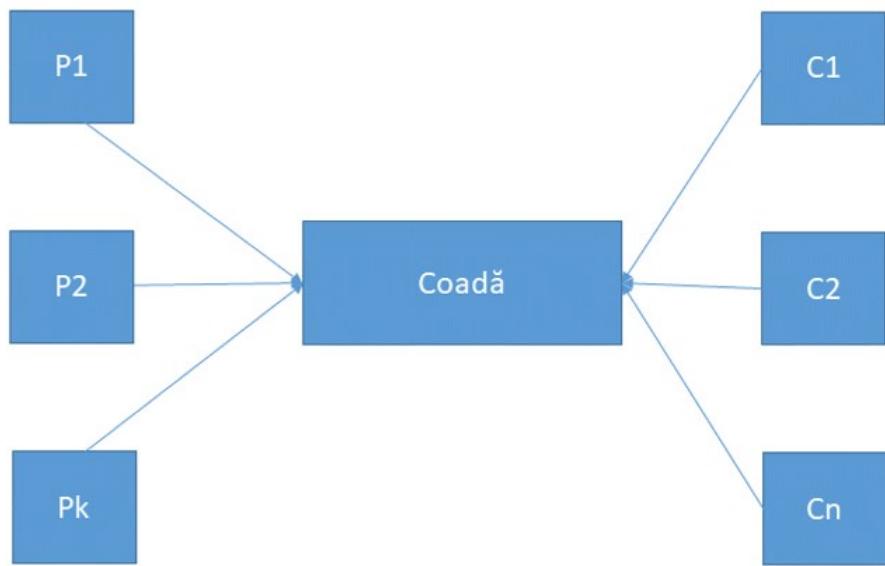
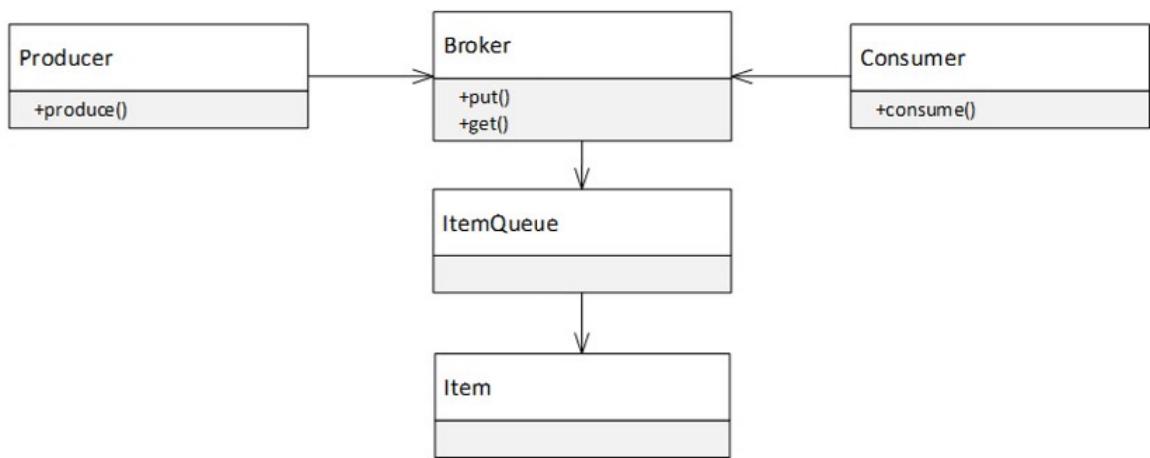


Diagrama de clase



Componente

- **Producer**

- Produce elemente și le adaugă în structura partajată

- **Consumer**

- Utilizează elementele și le elimină din structura partajată

- **Item**

- Elemente produse de *Producer* și utilizate de *Consumer*

- **ItemQueue**

- Stochează elementele produse de *Producer* și utilizate de *Consumer*

- **Broker**

- Gestioneaază accesul la structura partajată (*ItemQueue*)

Half-Sync/Half-Async

Problema

- Există sisteme care au următoarele caracteristici:
 - Trebuie efectuate sarcini ca răspuns la evenimentele externe care apar asincron (ex. întreruperi hardware)
 - Este ineficient să se dedice un fir separat de control pentru a efectua operații de I/O sincron pentru fiecare sursă externă de eveniment
 - Sarcinile de nivel mai înalt în sistem pot fi simplificate semnificativ dacă operațiile de I/O sunt efectuate sincron
- Una sau mai multe operații dintr-un sistem trebuie să ruleze într-un singur fir de control
 - Alte operații pot beneficia de mai multe fire de execuție

Problema

- Sistemele concurente includ atât servicii asincrone cât și sincrone
- Implementările asincrone sunt, în general, mai eficiente
 - anumite servicii pot fi asociate direct pe mecanisme asincrone (manipulatoare de întreruperi hardware sau de semnale software)
- Implementările sincrone simplifică efortul de programare
 - anumite servicii pot fi constrânsă să ruleze la puncte bine definite în secvență de prelucrare

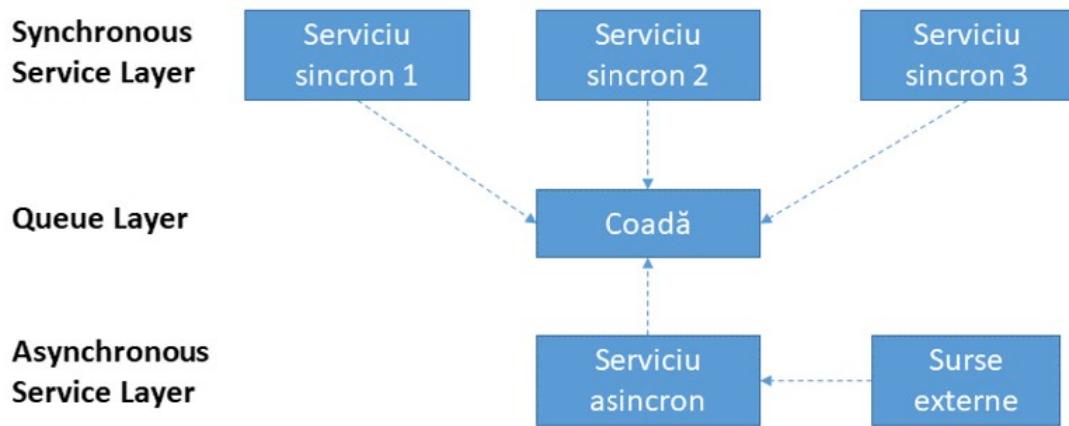
Scop

- Deconectarea procesărilor asincrone și sincrone în sistemele concurente
- Introducerea a două niveluri de comunicare (unul pentru asincron, altul pentru sincron)
 - Simplificarea scrierii programelor
 - Fără reducerea necorespunzătoare a performanței

Componente

- **Synchronous Service Layer (SSL)**
 - Execută procesarea de nivel înalt în mod sincron
- **Asynchronous Service Layer (ASL)**
 - Execută procesarea de nivel scăzut în mod asincron
- **Queue Layer (QL)**
 - Asigură o zonă tampon între nivelurile sincron și cel asincron
- **External Events Source**
 - Generează evenimente recepționate și procesate de ASL

Diagrama de componentă



Two-Phase Terminator

Problema

- Existența unui proces sau fir de execuție care rulează nedefinit
- În cazul apariției unor situații excepționale, terminarea forțată ar conduce la efecte neprevăzute
- Este necesar ca firul de execuție sau procesul să aibă timp, înainte de terminare, să elibereze resursele și să finalizeze anumite operații

Scop

- Permite închiderea ordonată a unui fir de execuție sau a unui proces în cadrul unei aplicații concurentă
- Firul de execuție poate elibera resursele înainte de încheierea execuției

Diagrama de clase

Terminator
-solicitareIncheiereExecutie
esteSolicitataIncheiereaExecutiei incheieExecutia

- **esteSolicitataIncheiereaExecutiei**
 - Dacă returnează true, firul eliberează resursele și își încheie execuția
- **incheieExecutia**
 - firul își încheie execuția după eliberarea resurselor

Double Buffering

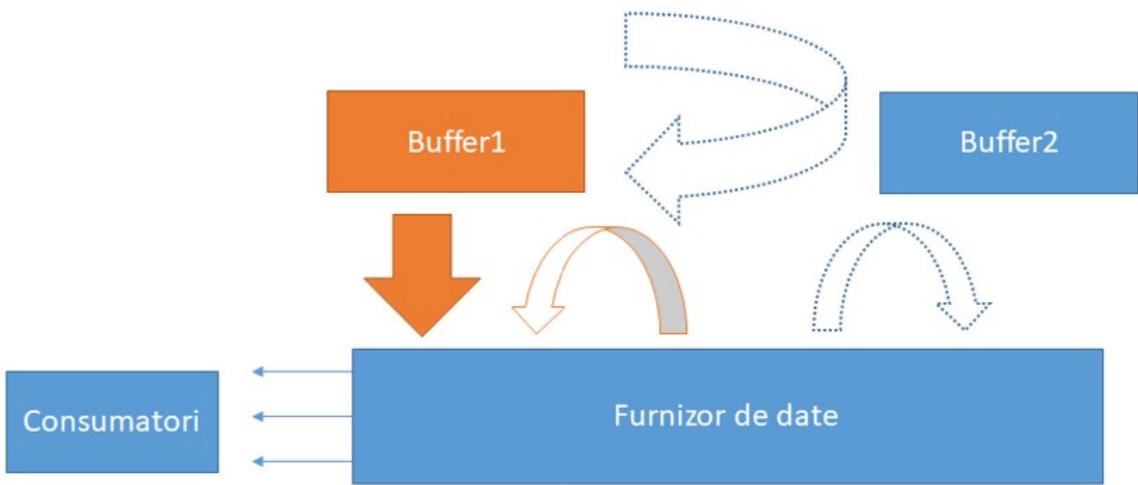
Problema

- Un obiect utilizează foarte multe date la un moment dat
- Există un obiect care produce date; acesta nu are control asupra momentului în care datele vor fi utilizate
- Pot apărea probleme legate de performanță, dacă datele nu sunt disponibile pentru prelucrare

Scop

- Evitarea întârzierilor în prelucrarea datelor prin producerea în mod asincron a datelor
- Asigurarea datelor înainte de necesitatea prelucrării acestora
- Formă specializată a modelului de proiectare **Producer-Consumer**

Diagrama



Balking

Problema

- Un obiect poate avea o stare care nu permite apelul unei metode
- Execuția metodei nu poate fi amînată, aceasta trebuind să fie executată imediat
- Dacă obiectul nu se află într-o anumită stare, apelul unei metode nu este necesar

Scop

- Permite unui fir de execuție să finalizeze prelucrările dacă o condiție nu a fost îndeplinită
- O metodă nu va fi apelată, dacă nu sunt îndeplinite anumite condiții

Diagrama de clase

ObiectProjejat	
stare	
actualizeazaStare {guarded} metodaProtejata {guarded}	<ul style="list-style-type: none">• actualizeazaStare<ul style="list-style-type: none">• este utilizată pentru modificarea stării obiectului• se apelează doar dacă sănt îndeplinite anumite condiții• metodaProtejata<ul style="list-style-type: none">• este apelată dintr-un fir, iar execuția continuă doar dacă este îndeplinită o anumită condiție

Implementare

- Se testează starea unui obiect (în mod sincron), iar dacă aceasta nu corespunde unei condiții, se ieșe din metodă
- Ieșirea din metodă se poate realiza prin
 - Returnarea unei valori
 - Lansarea unei excepții

Guarded Suspension

Problema

- Metodele unei clase sănătate marcate ca fiind sincronizate
- Starea unui obiect face imposibilă execuția completă a unei anumite metode sincronizate
- Dacă s-ar executa metoda în starea curentă a obiectului ar putea apărea un blocaj (deadlock)

Scop

- Permite unui proces să aștepte pînă când au fost îndeplinite anumite precondiții înainte de procesare
- Similar cu **Balking**, cu diferența că se așteptă îndeplinirea condiției și nu se ieșe din funcție

Diagrama de clase

ObiectProjejat
stare
actualizeazaStare {guarded} metodaProtejata {guarded}

- **actualizeazaStare**
 - este utilizată pentru modificarea stării obiectului
 - se apelează doar dacă sînt îndeplinite anumite condiții
- **metodaProtejata**
 - este apelată dintr-un fir, iar execuția este suspendată pînă cînd este îndeplinită o anumită condiție

Implementare

- Suspendarea metodei, dacă nu este îndeplinită condiția
- La îndeplinirea condiției, firele suspendate își pot relua execuția
- Utilizarea metodelor de tipul wait() și notify()

Read/Write Lock

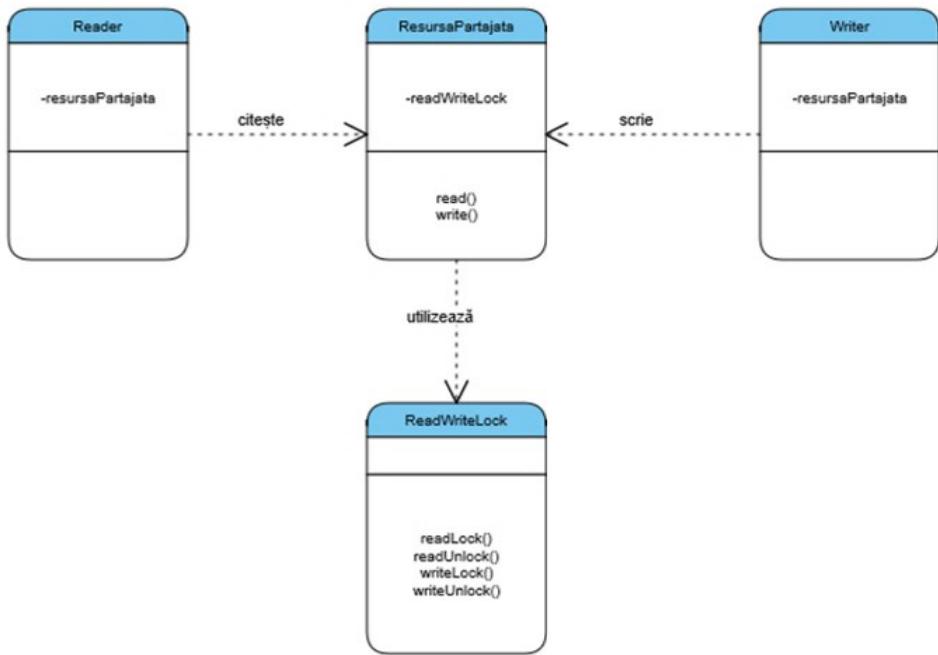
Problema

- Într-un mediu concurențial, citirea stării unui obiect în timpul modificării acesteia poate conduce la inconsistență
- Frecvența citirilor este mai mare decât frecvența scrierilor

Scop

- Permite citiri simultane și scrieri exclusive în cadrul unei aplicații concurente
- Nu se poate scrie în momentul efectuării unei citiri
- Formă specializată de **Scheduler**

Diagrama de clasă



Thread Pool

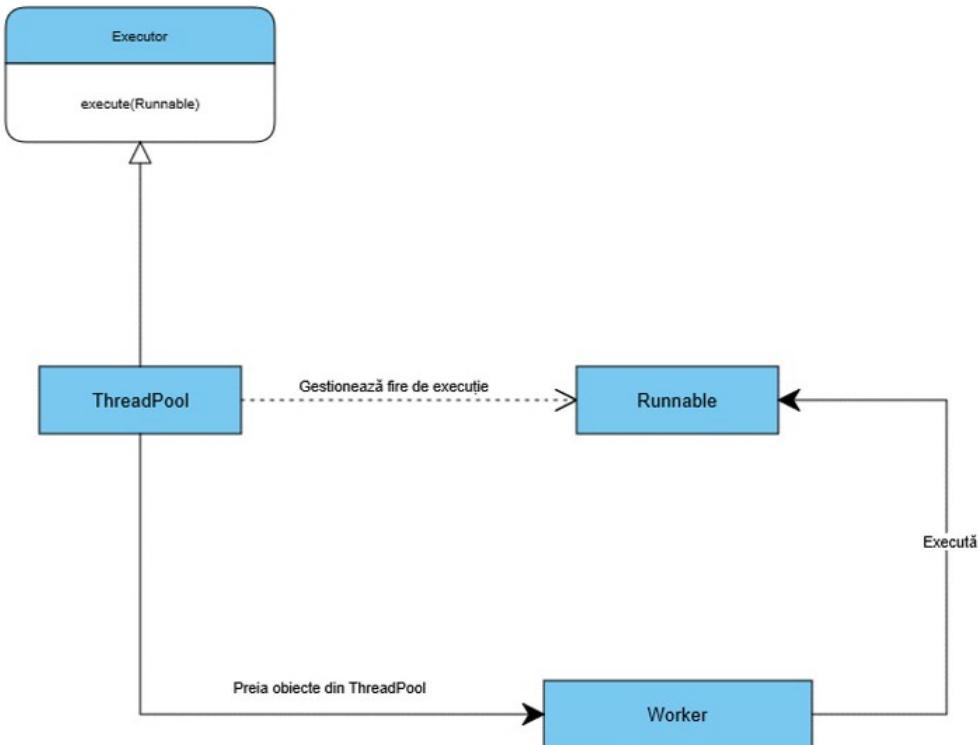
Problema

- Necesitatea efectuării unor operații independente, în fire de execuție distincte
- Costul creării unor fire noi de execuție este ridicat
- Există un număr optim de fire de execuție care pot rula la un moment dat
 - Prea multe fire -> probleme de performanță
 - Prea puține fire -> resurse neutilizate

Scop

- Reutilizarea firelor de execuție
- Evitarea efortului de creare de noi fire de execuție
- Gestiunea numărului de fire existente la un moment dat

Diagrama de clase



Implementare

• Executor

- Interfața definește o metodă care primește ca parametru un obiect de tip **Runnable**, în vederea execuției acestuia

• ThreadPool

- Implementează interfața **Executor**
- Gestioneză o listă de fire de execuție pentru procesare sarcinilor transmise
- Limitează numărul de obiecte de tip **Worker**

• Runnable

- Interfață ce permite execuția într-un fir distinct

• Worker

- Au asociat un fir de execuție
- Primesc obiecte din **ThreadPool** și le execută

Asynchronous Processing

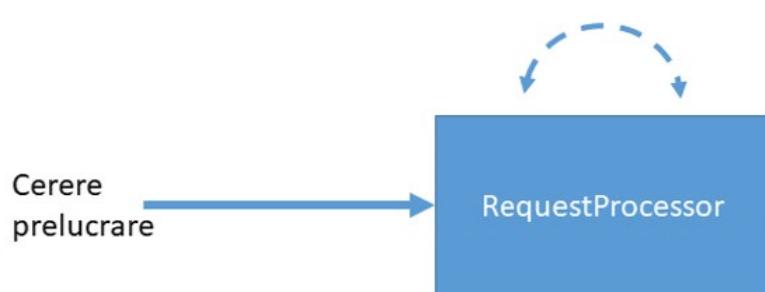
Problema

- Un obiect primește solicitări pentru anumite prelucrări
- Solicitările sunt asincrone, iar clientii nu trebuie să aștepte rezultatul
- Este posibil ca solicitările să fie primite în timpul efectuării unor prelucrări

Scop

- Prelucrarea asincronă a unor solicitări prin secvențializarea acestora
- Se pot utiliza pentru implementare modelele
 - Thread Pool
 - Scheduler
 - Producer-Consumer

Diagrama

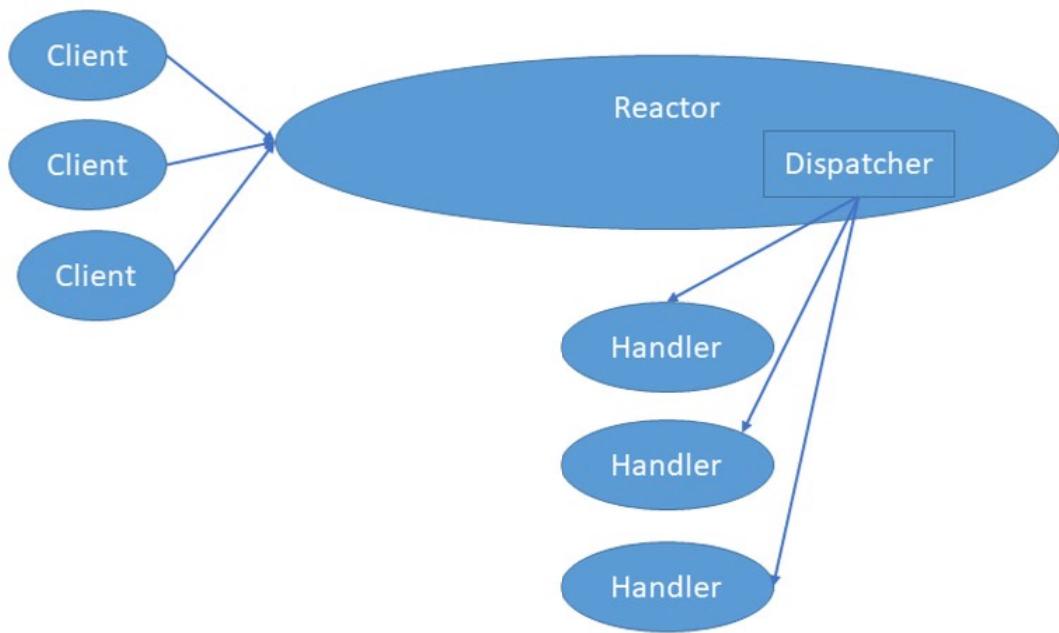


Reactor

Problema

- Aplicațiile bazate pe evenimente trebuie să gestioneze diferite cereri, destinate unor servicii specifice, în mod asincron
- Cererile recepționate în mod asincron trebuie distribuite către serviciile specifice
- Este necesară menținerea scalabilității

Diagrama



Implementare

- **Reactor**

- Rulează într-un fir de execuție distinct
- Reacționează la solicitările de prelucrare (I/O) prin distribuirea acestora către obiectul de tip **Handler** potrivit

- **Handler**

- Prelucrează operațiile de I/O
- Prelucrările se realizează operații fără blocare

Thread-Specific Storage

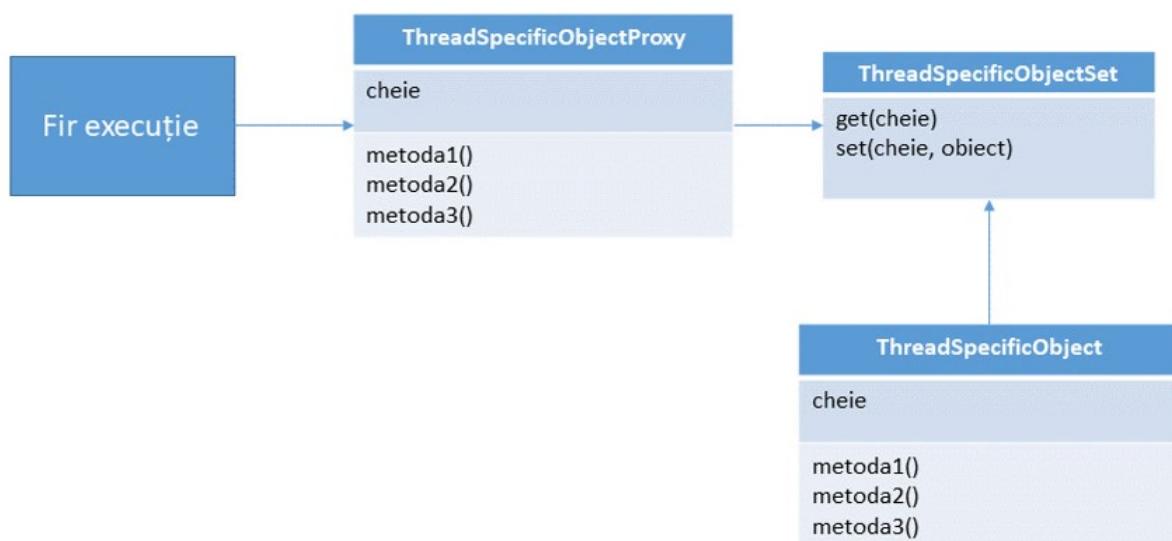
Problema

- Partajarea resurselor într-un context concurențial
- Evitarea blocajelor prin utilizarea obiectelor de sincronizare

Scop

- Accesul firelor de execuție la un obiect local acestora, prin intermediul unui singur punct, global
- Nu implică un mecanism de sincronizare la fiecare acces
- Se realizează copii locale al obiectelor

Diagrama de clase



Implementare

- **ThreadSpecificObject**

- Obiecte accesibile firelor de execuție
- Accesul la aceste obiect se realizează prin intermediul unei chei

- **ThreadSpecificObjectProxy**

- Permite accesul clientilor la obiectele de tip **ThreadSpecificObject**
- Simulează obiectele reale

- **ThreadSpecificObjectSet**

- Gestionează obiectele **ThreadSpecificObject** asociate unui fir de execuție

Referințe

- .NET Design Patterns, <http://www.dofactory.com/net/design-patterns>
- Data & Object Factory, *Gang of Four Software Design Patterns*, Companion document to Design Pattern Framework™ 4.5, 2017
- Data & Object Factory, *Patterns in Action* 4.5, A pattern reference application, Companion document to Design Pattern Framework™ 4.5, 2017
- Design Patterns | Object Oriented Design, <http://www.oodesign.com/>
- Design patterns implemented in Java, <http://java-design-patterns.com/patterns/>
- M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, R. Stafford, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002
- E. Freeman și alții, *Head First Design Patterns*, O'Reilly, 2004
- M. Grand, Patterns in Java, Volume 1—A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, John Wiley & Sons, 2002
- M. Grand, Patterns in Java, Volume 3 - Java Enterprise Design Patterns, John Wiley & Sons, Inc., 2002
- J.D. Meier et al, Application Patterns, <http://apparch.codeplex.com/wikipage?title=Application%20Patterns>, 2009
- D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, Volume 2, John Wiley & Sons, 2000
- A. Shivets, *Design Patterns Made Simple*, <http://sourcemaking.com>
- Stack Overflow, <https://stackoverflow.com/>
- D. Trowbridge et. al, *Enterprise Solution Patterns Using Microsoft .NET*, Version 2.0, Microsoft, 2003