Design Patterns

Obiective

- Parcurgerea unor șabloane de proiectare
- Rezolvarea unor probleme practice cu ajutorul lor
- Folosirea unui limbaj orientat pe obiecte pentru implementarea soluțiilor
- Exersarea programării orientate pe obiect

Note preliminare

1. pornim de la premisa că participanții sunt familiarizați cu șabloanele de proiectare (de la școală sau din activitatea practică)
2. participanții sunt programatori într-un limbaj orientat pe obiecte cel puțin la nivel mediu, de preferință Java, C# sau C++
3. participanții sunt familarizați cu UML 2
4. suportul de curs NU este un manual, conține doar scheme, idei pentru a puncta aspecte importante ale trainingului
5. premeditat nu există exemple de cod (se pot găsi extreme de ușor pe Internet, sunt foarte multe cărți pe această temă); abordarea aleasă este de a se prezenta pe scurt șabloane, se vor discuta particularități, context de aplicare, modalități de aplicare, se vor implementa lucruri mai interesante
6. este un prilej de a exersa modul de rezolvare a problemelor orientate pe obiecte, se vor aborda probleme de design și modalități de implementare într-un limbaj orientat pe obiecte

Cuprins

1 Introducere

Design Patterns - Erich Gamma, Richard Helm, John Vlissides, and Ralph Johnson, 1994

Un şablon reprezintă o soluţie comună a unei probleme într-un anumit context.

Descriere:
- Nume
- Problemă
- Soluţie
- Implementări
- Consecinţe

Categorii:
- **Fundamentale**: delegare, interfaţa, clasa abstractă, obiecte imutabile, interfeţe, marker, proxy
- **Creaţionale**: factory method, abstract factory, builder, prototype, singleton, object pool
- **De partiţionare**: filter, compozit, interfaţa read-only
- **Structurale**: adapter, iterator, bridge, façade, flyweight, dynamic linkage, decorator, cache management
- **Comportamentale**: chain of responsibility, command, little language, mediator, snapshot, observer, state, null object, strategy, template method, visitor

Alte tipuri de patterns software:
- **Antipatterns** – soluţii proaste foarte răspândite: god class, singletonitis, basebean (subclasarea unei clase utilitare), abuzul de exception handling
- **Patern-uri GRASP** – General Responsibility Assignment Patterns; indicaţii despre asignarea responsabilităţii claselor, ce clase vor apare în design; ex: low coupling/high cohesion, controller, law of Demeter (don't talk with strangers), creator, expert, polymorphism, etc.
- **Pattern-uri arhitecturale**: MVC, layers, event-driven, microkernel (plugin), microservices
- **GUI Design Patterns**: window per task, disabled irrelevant things, explorable interface, etc.
- **Database Patterns**: decoupling patterns, resource patterns, input and output patterns, cache patterns
- **Concurrency Patterns**: double buffering, lock object, producer-consumer, asynchronous processing, etc.
- **Enterprise systems** (ex. J2EE): Data Access Object, Transfer Objects

Idiom - este legat de un anumit limbaj de programare şi reprezintă o convenţie general acceptată de utilizare a limbajului respectiv.

Exemplu: returnarea unui întreg pentru a semnala ce s-a întâmplat în funcţie.

Frameworks - reprezintă o colecţie de clase care oferă un set de servicii pentru un domeniu particular. Se exportă un număr de clase şi mecanisme pe care utilizatorii le pot adapta.
Exemple: Swing, JavaFX, Windows Forms, Windows Presentation Framework, etc..

Refactoring – proces de schimbare a sistemului software ce nu schimbă comportarea exterioară, dar care schimbă structura internă. Se urmăreşte îmbunătăţirea design-ului după ce codul a fost scris.

2 Principii de design (SOLID)

Robert C. Martin (2000).

Semne ale degradării unui sistem:

- **Rigiditatea** – schimbarea este dificilă, orice schimbare afectează prea multe părți ale sistemului
- **Fragilitatea** – tendința software-ului de a se strica în mai multe locuri în momentul în care se fac schimbări
- **Imobilitatea** – greu de reutilizat în alte aplicații fiindcă soluția este prea particulară
- **Vâscozitatea –** dificultatea de a face cum trebuie un lucru:
  - **La design** – schimbările nu respectă designul, sunt mai ușor de făcut "hacks"
  - **Al mediului de dezvoltare –** de exemplu dacă compilarea durează mult programatorii vor evita să facă schimbări optime care ar duce la compilări lungi; sau lucrul cu sistemul de source control
- **Complexitatea fără rost (gold plating)**
- **Cod duplicat**

Motivul principal: schimbarea/dinamica cerințelor. Design-ul original nu mai face față.

Soluția: managementul dependențelor (dependency firewalls) pentru a stopa propagarea dependențelor prin folosirea de principii și tehnici (design patterns).

2.1 Principiul responsabilității unice (SRP)

O clasă trebuie să aibă un singur motiv de schimbare (= responsabilitate).

Motiv de schimbare = responsabilitatea (apartine) unui actor.

Dacă apar mai multe motive de a schimba clasa se poate considera regândirea ei.

SRP sugerează ca toate clasele trebuie să fie specifice.

Prin generalizare apar interfețele. Acestea sunt descoperite, nu se proiectează.

Abstractizarea = eliminarea irelevantului și amplificarea esențialului.
Principiul reutilizării abstracțiilor = abstracțiile trebuie refolosite.
Abordare: start cu clase concrete, descoperă abstracțiile pe măsură ce se descoperă elementele comune.
Rule of three: nu introduce generalizarea până nu apar 3 situații comune!

Exemple: logging, caching, storage, orchestration.

2.2 Principiul open close (OCP)

Entitățile software (module, clase, funcții) trebuie sa fie deschise pentru extindere și închise pentru modificări.

Motiv: odată făcută publică o clasă ea este folosită de alții (bug fixing este OK).

Entitățile software trebuie să fie scrise pentru a se putea extinde, fără a fi nevoie să fie modificate (cel puțin parțial!).

Preferă compoziția față de moștenire!

Tehnici: strategy, composite, decorator, factory method, moștenire & polimorfism

Exemplu: drawShape(), aplicare polimorfism

Clasele suferă operații (din partea programatorului) CRUD – Create/Read/Update/Delete

OCP sugerează că de la un moment dat o clasă nu mai poate suferi operații de Update & Delete!

Tehnică: strangler – o clasă înlocuiește treptat o alta (care este obsolete)

2.3 Principiul substituției lui Liskov (LSP)

Subclasele trebuie să poată substitui clasele lor de bază.

Implementările trebuie consumate fără a schimba corectitudinea sistemului (corectitudine = să funcționeze).

(Contra)**Exemplu**: Elipsa moștenită de Cerc (Dreptunghi/Patrat) – nu respectă acest principiu!

Simptome ce semnalează violarea LSP:

- Generarea de excepții NotSupportedException
- Downcasts
- Interfețe extrase din clase concrete (spre deosebire de cele rezultate din cel puțin trei exemple concrete)

Principiul reutilizării abstracțiilor indică respectarea LSP.

Design by Contract – contractul clasei de bază trebuie onorat de clasele derivate.

2.4 Principiul segregării prin interfață (ISP)

Mai multe interfețe client specifice sunt mai bune decât o singură interfață generală.

Categoriilor de clienți li se oferă acces la servicii prin intermediul unei interfețe specializate. Schimbarea unei metode nu afectează toți clienții, doar cei care utilizează acea metodă.

Clienții nu trebuie să fie forțați să depindă de metode pe care nu le folosesc!

Clienții definesc interfața (role interface), nu clasa concretă ce implementează interfața (adică interfețe extrase).

2.5 Principiul inversării dependenței (DIP)

Modulele de nivel înalt nu trebuie să depindă de module de nivel jos. Toate trebuie să depindă de abstracții.
Abstracțiile nu trebuie să depindă de detalii, detaliile trebuie să depindă de abstracții.

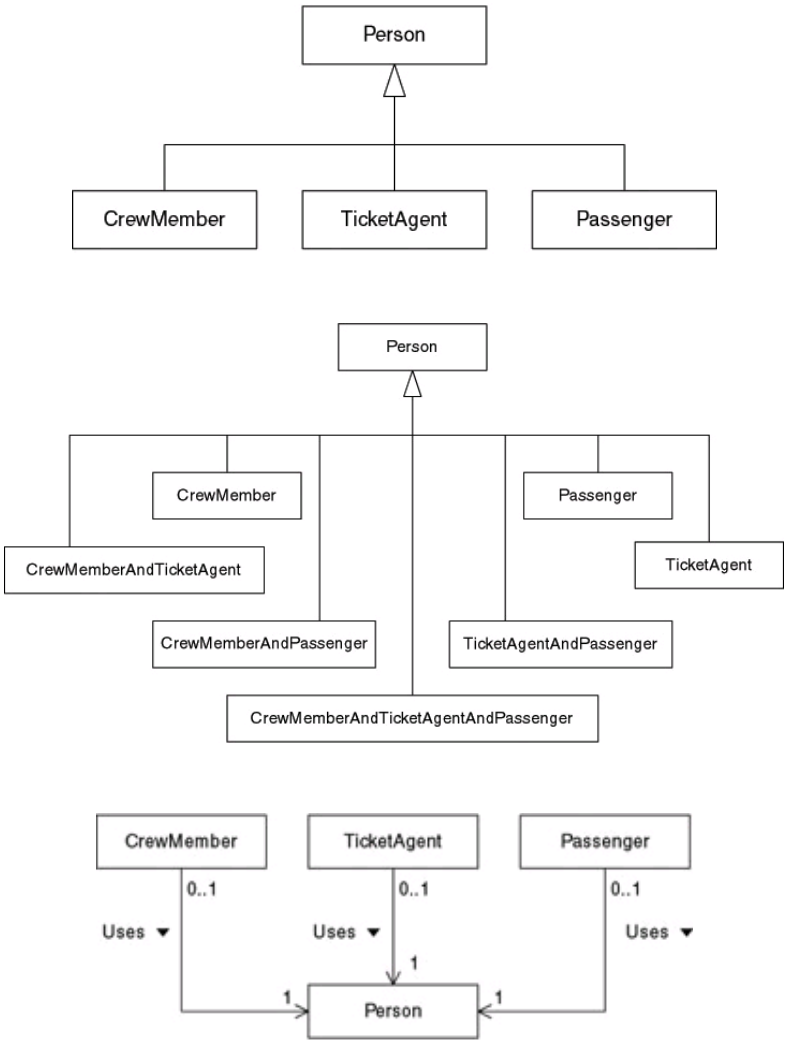
DIP este strategia de a depinde de interfețe, funcții sau clase abstracte, nu de funcții sau clase concrete. Stă la baza proiectării componentelor (COM, EJB, CORBA, etc).

Problema: modulele de nivel înalt depind de module de nivel mai scăzut.

Modulele de nivel înalt lucrează cu concepte de nivel înalt ce nu ar trebui să cunoască detaliile de implementare. Ele ar trebui să lucreze cu abstractizări ale nivelurilor inferioare.
Soluție: dependența de abstractizări; abstractizările se schimbă mai puțin decât implementările concrete.

3 Design Patterns Fundamentale

3.1 Delegare

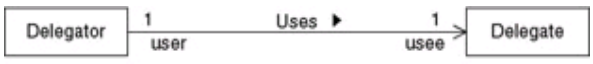


Situaţii:

- Moştenirea este o relaţie statică; pentru implementarea rolurilor este mai potrivită delegarea
- Dacă o clasă doreşte să ascundă metode moştenite de la părinte mai bine foloseşte delegarea
- Nu se subclasează de obicei o clasă utilitară (utility class)

Delegarea este mai greu de implementat, impune o structurare mai slabă decât moştenirea.

Soluţie:



Implementare

Exemplu: mecanismul de tratare a evenimentelor in Swing

Exercitiu: Să se definească o clasa Biblioteca care să modeleze funcţionarea unei biblioteci (intrări de cărţi, casare cărţi, afişarea listei de cărţi). Biblioteca va fi o colecţie de obiecte de tip Carte caracterizate prin nume, autor, ISBN, etc. Comportamentul clasei Biblioteca se va obţine aplicând delegarea asupra unei clase generale, gen Vector, HashTable sau Lista, folosind operaţiile specifice de adăugare/eliminare de elemente, respectiv parcurgere. Clasa delegat va fi definită tot în cadrul temei sau va fi preluată din biblioteca ataşată mediului de programare folosit.

3.2 Obiecte imutabile

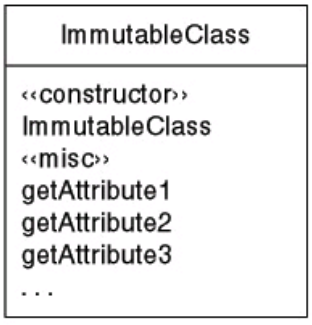
Starea obiectului nu se modifică odată ce a fost creat. Creşte robusteţea şi mentenabilitatea programelor.

Avantaje obiecte imutabile:

- copierea se poate face prin asignare referințe
- se pot refolosi (flyweight pattern), caz în care compararea se face prin testarea referințelor
- funcționează corect nemodificate în context multithreading

Implementare:

- Doar constructorii vor seta membrii dată instanţă.
- Orice metodă ce calculează o nouă stare vor crea o nouă instanţă, nu vor modifica starea obiectului



Exemplu: String din Java (metoda intern())

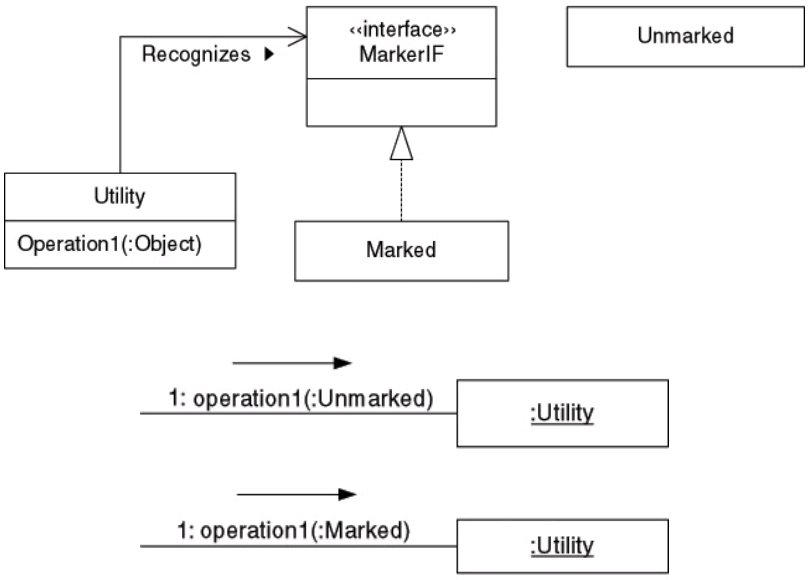
Exerciţiu: scrieţi o clasă Circle ce va avea instanţe imutabile; pentru acestă clasă definiţi metodele move() – mutarea cercului, resize() – modificarea razei cercului.

3.3 Interfeţe marker

O clasă implementează o interfaţă marker pentru a semnala un atribut semantic.

Acest pattern este foarte folosit de clase utilitar care trebuie să ştie ceva despre obiecte fără a presupune că sunt de o anumită clasă.

Exemplu: EqualByIdentity pentru colecţii



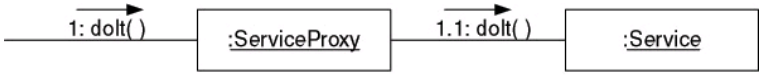
Exemple: interfeţele Serializable, Cloneable

Exerciţiu 1: scrieţi două varinte pentru o clasă (de ex. clasa Complex): una suporta clonare, cealaltă nu.

Exerciţiu 2: implementaţi o colecţie care foloseşte equals() sau == pentru obiectul cautat funcţie de faptul că obiectul implementează sau nu interfaţa marker EqualByIdentity.

3.4 Proxy

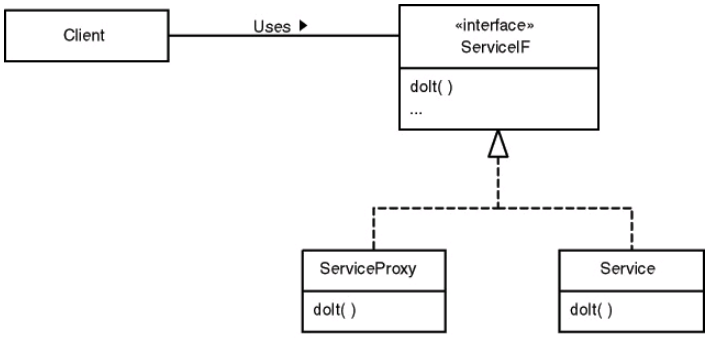
Clienţii unei componente comunică cu un reprezentant al componentei originale. De obicei clienţii nu ştiu că interacţionează cu un proxy, nu cu componenta ce implementeză serviciul.

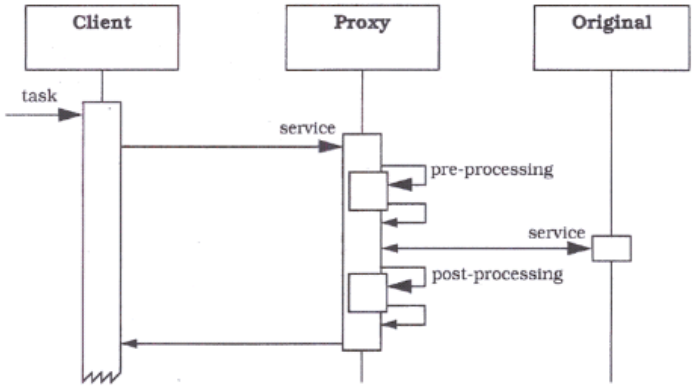


Obiectul proxy poate oferi diverse servicii:

- Decuplează execuţia unei operaţii lungi de apelul clientului
- Remote proxy: creează iluzia că un obiect din alt spaţiu de adrese se află local (RMI, CORBA)

- Protection proxy: controlează accesul la serviciu pe baza unei politici de securitate
- Virtual proxy: creează iluzia că un obiect serviciu există înainte ca acesta să fie creat
- Cache proxy: clienţii locali folosesc informaţii de la componente remote
- Synchronization proxy: accesul multiplu simultan la o componentă trebuie sincronizat
- Counting proxy: pentru evitarea distrugerii accidentale a unei componente, strângerea de statistici (smart reference)
- Firewall proxy: clienţii sunt protejaţi de lumea înconjurătoare





Exerciţiu: lazy clone pentru un Map – clonarea se realizează doar înainte de a modifica colecţia originală.

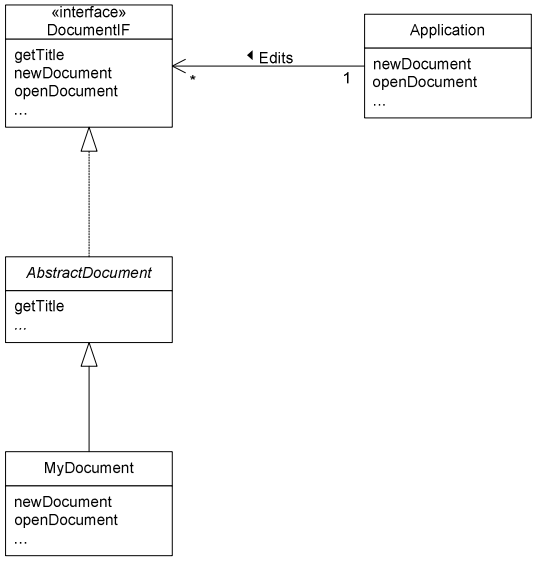
4 Design Patterns Creaţionale

Se ocupă cu modalităţi de creare a obiectelor. Deciziile luate în această etapă vizează de obicei ce clasă se instanţiază sau cine este responsabil cu această sarcină.

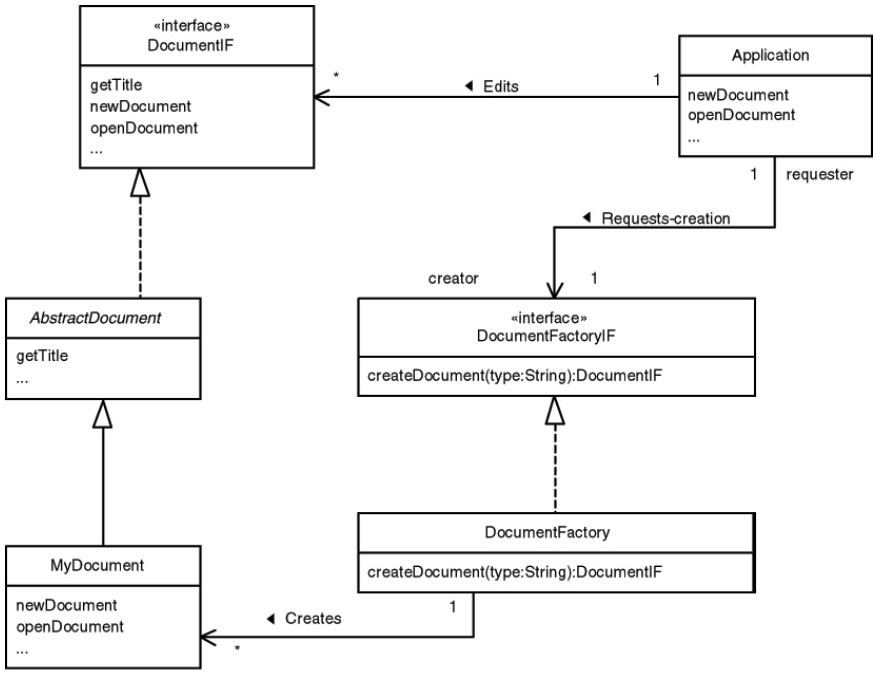
4.1 Factory Method

Este un mod prin care un obiect iniţiază crearea unui alt obiect fără a ştii în prealabil clasa acestuia.

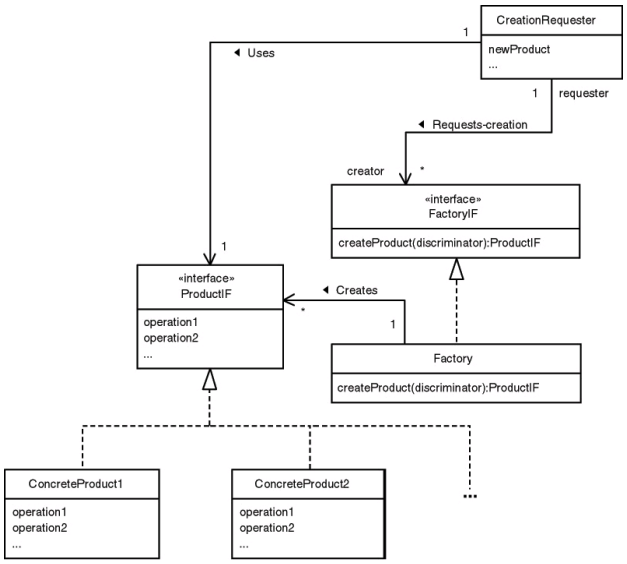
Context



Creare document particular



General



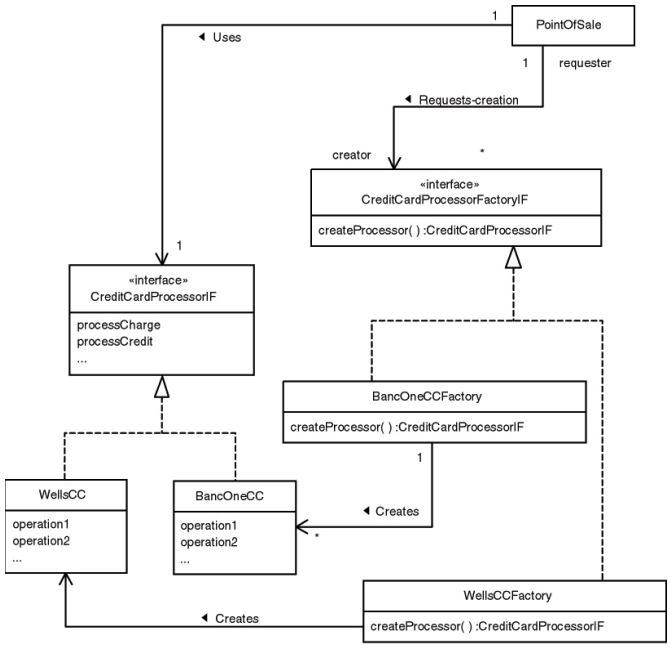
Observaţie: data-driven class determination – clasa care se instanţiază se determină funcţie de informaţia de intrare:

```
Image createImage (String ext) {
   if (ext.equals("gif")
     return new GIFImage();
   if (ext.equals("jpeg"))
     return new JPEGImage();
   ...
}  // createImage(String)
```

Alternative:

- hashed adaptor objects
- reflection

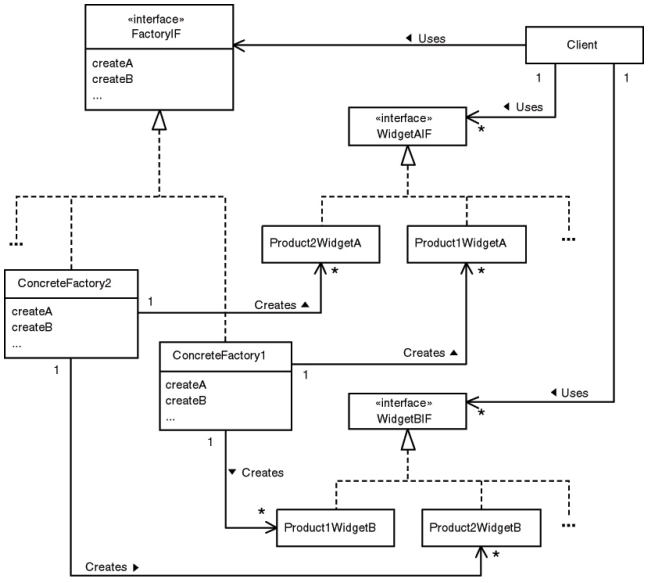
Variantă în care obiectul factory crează acelaşi tip de obiect (layered factory method):



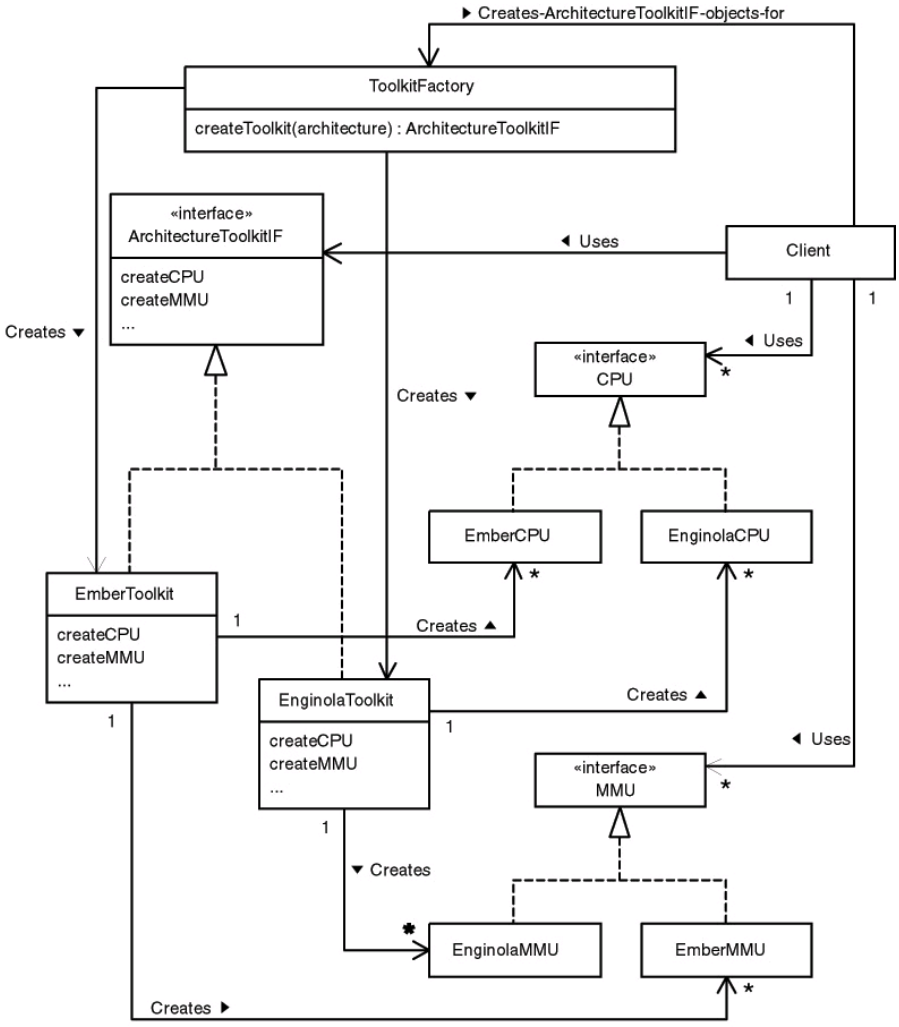
Exerciţiu: daţi un exemplu de aplicare a acestui şablon împreună cu o implementare, de exemplu crearea de obiecte comandă pe baza unui șir de caractere.

4.2 Abstract Factory

Se mai numeşte Kit sau Toolkit. Este o modalitate de a crea o varietate de obiecte de diferite tipuri fără a ştii ce clase se folosesc, dar se asigură că se foloseşte combinaţia de clase corectă.



Exemplu: sistem pentru repararea sistemelor harware; look&feel din Java; seturi de imagini pentru piesele unui joc de șah

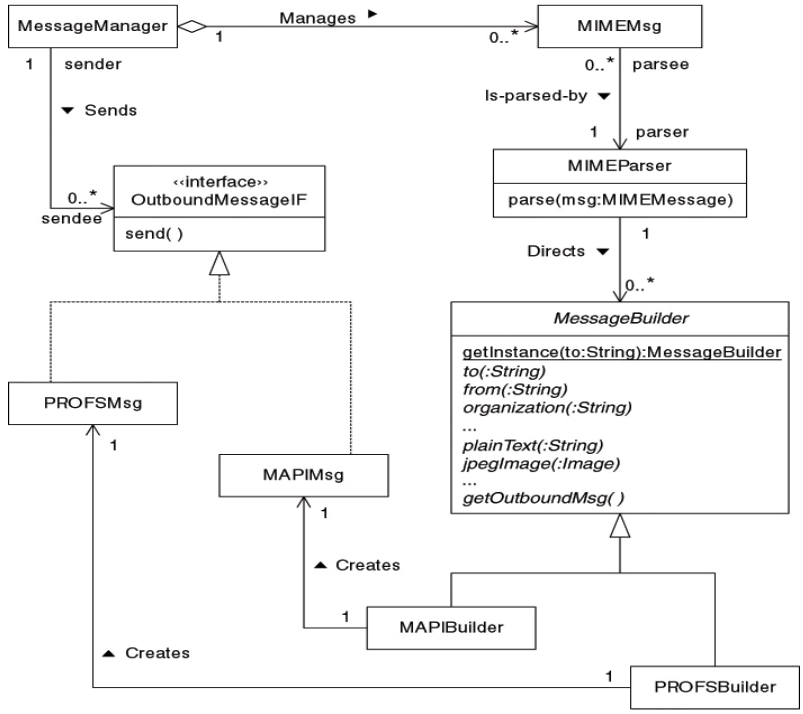


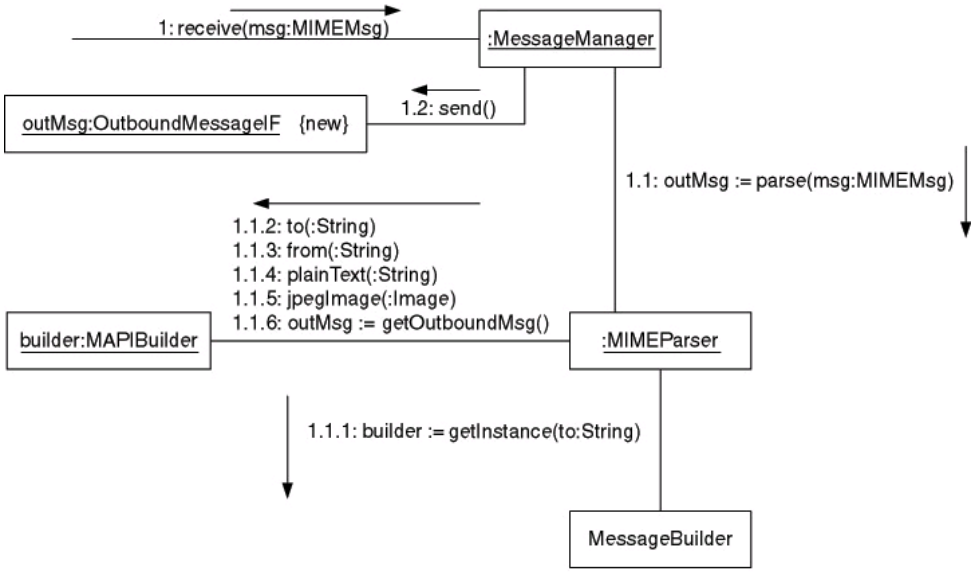
Exerciţiu: implemenaţi exemplul de mai sus, mai exact o funcție care returnează un obiect compozit de tip calculator.

4.3 Builder

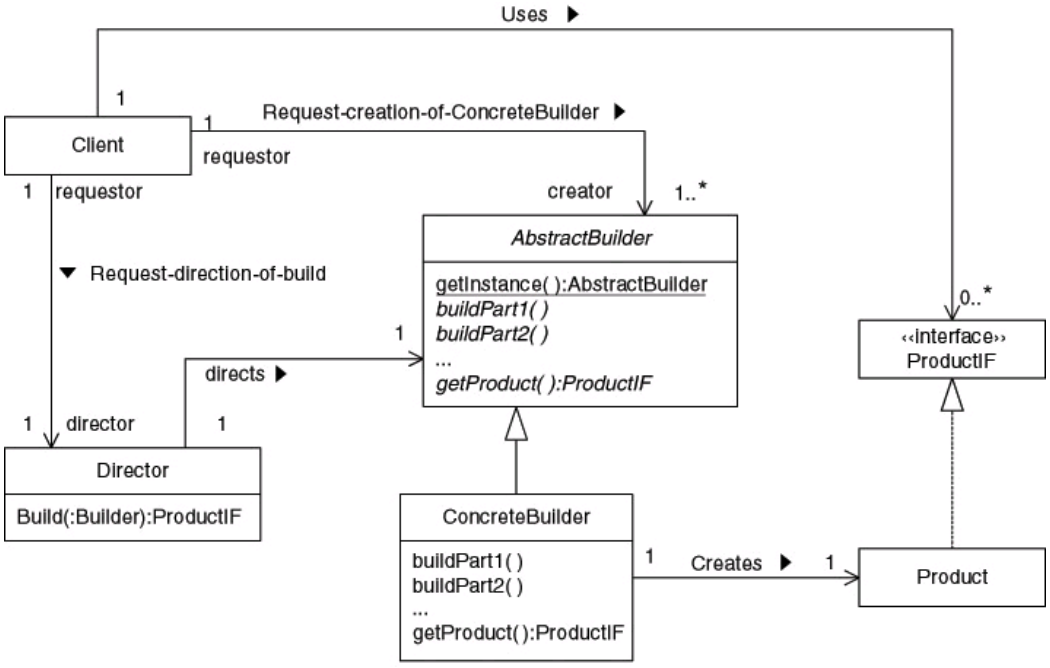
Permite unui client să construiască un obiect complex doar prin specificarea tipului şi conţinutului lui.

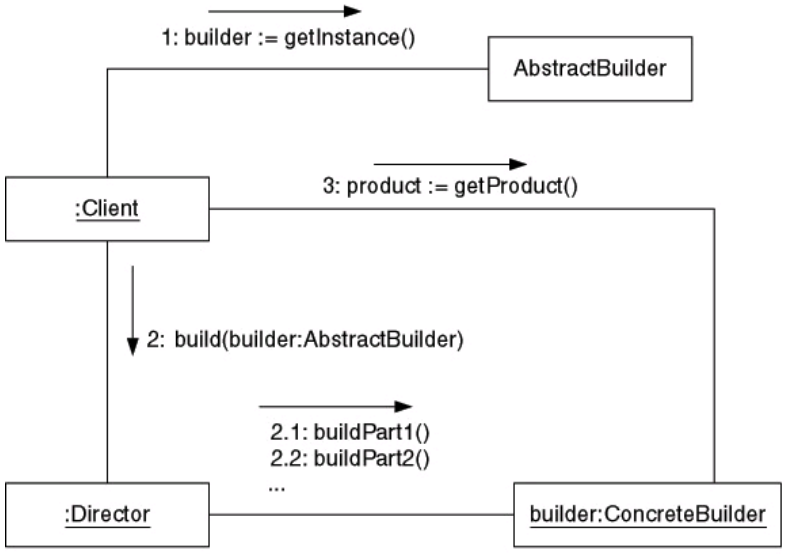
Exemplu:





General



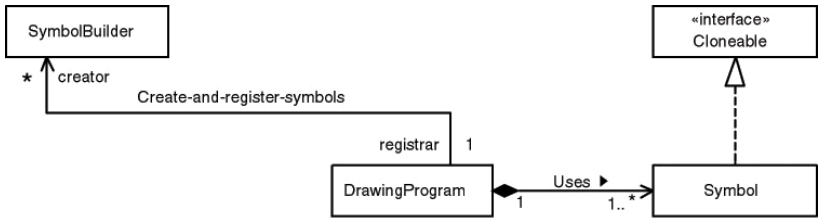


Exerciţiu: daţi o implementare pentru exemplul de mai sus

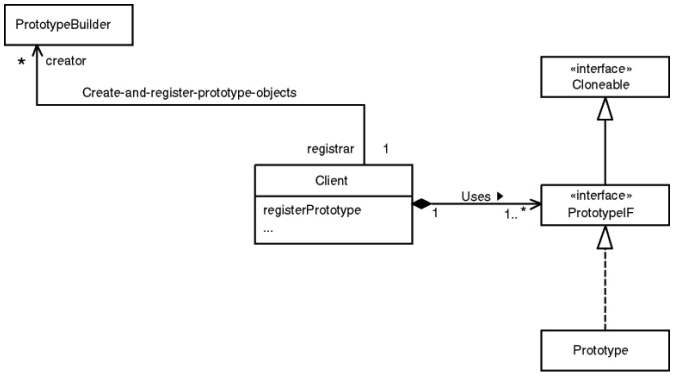
4.4 Prototype

Permite crearea de obiecte fără a ştii exact clasa lor sau detalii de creare a lor (constructor polimorfic).

Context



Soluţie



Copiere:

- Shallow (clone)
- Deep

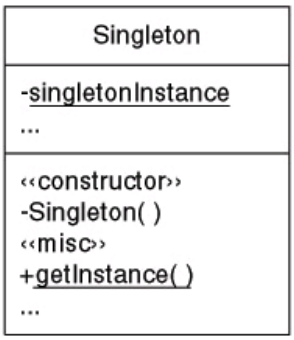
Exerciţiu: aplicaţi acest pattern pentru problema descrisă în context

4.5 Singleton

Asigură că se poate crea doar o singură instanţă a unei clase. Toate obiectele ce au nevoie de un asemenea obiect vor folosi aceeaşi instanţă.

Context: de obicei gestionează resurse

Soluţie



Implementare:

- Constructor privat
- Lazy initialization
- Apel concurent la getInstance()

Utilizare: logger, configurație, accesul la resurse shared, etc.

Discuții:

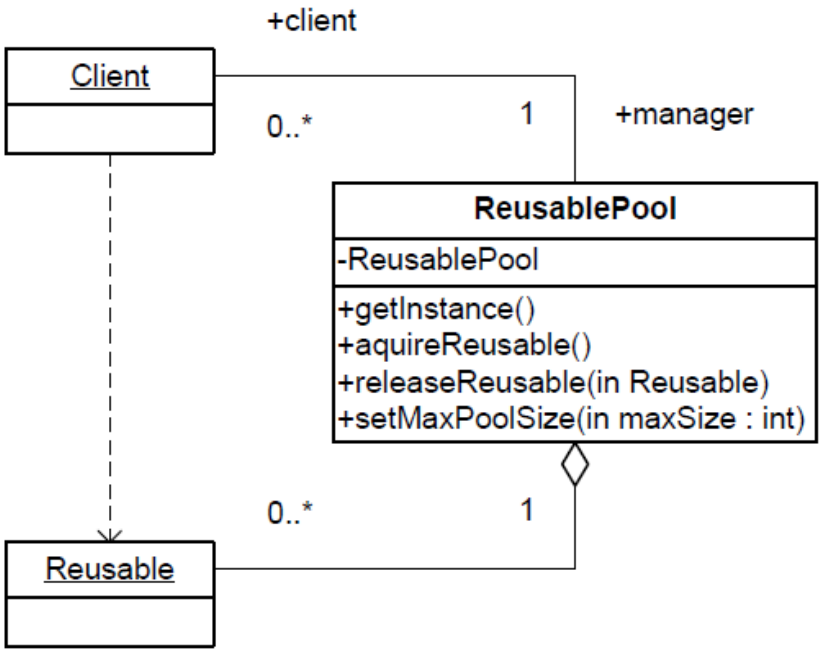
- deosebirea dintre singleton și funcționalitate la nivel de clasă
- comparație cu monostate
- implicatii asupra testelor unitare – dependency injection

Exerciţiu: implementați un singleton care să fie şi thread safe

4.6 Object Pool

Gestionează un set de obiecte pentru a fi refolosite. Se foloseşte când obiectele sunt foarte costisitor de creat sau există un număr limitat de obiecte ce pot exista la un moment dat.

General



Aspecte:

- Cine crează obiectele gestionate
- Câte obiecte se gestionează, dacă există limită (nr fix – array, altfel ArrayList)

Exerciţiu: implementaţi un pool manager de dreptunghiuri

5 Design Patterns de Partiţionare

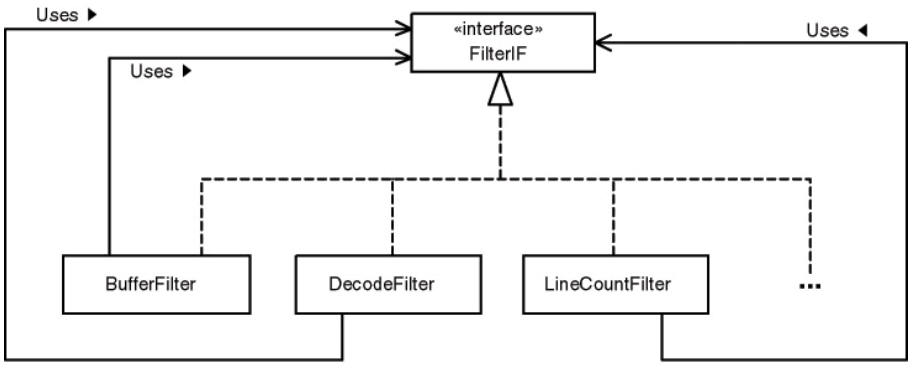
Şabloanele din acest capitol propun metode pentru partiţionarea claselor şi interfeţelor pentru a obţine un design bun.
Strategia aplicată este divide et impera: divizarea unei probleme complexe în probleme mai simple, mai uşor de rezolvat.

5.1 Filtru

Obiectele care au interfeţe compatibile dar realizează transformări sau operaţii diferite pe stream-urile de date pot fi conectate dinamic pentru a realiza operaţii arbitrare.

Context
Se pot defini clase ce fac transformări sau analiză pe stream-uri. Pentru a fi cât mai generale, să poată fi reutilizate, ele se pot scrie pentru a fi cât mai flexibile, instanţele lor să poată fi conectate cât mai uşor.

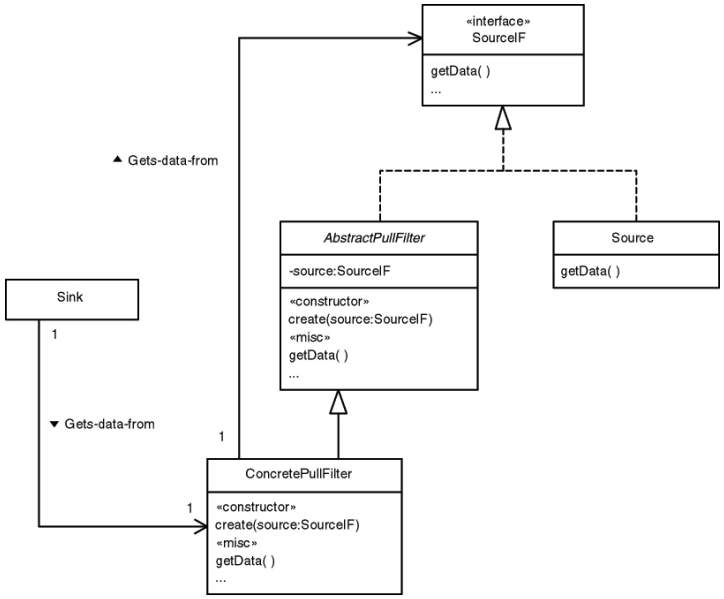
Lucrul acesta se poate realiza prin definirea unei interfeţe comune pe care toate aceste clase le implementează şi pe care le folosesc. Astfel o instanţă a unei clase poate folosi o instanţă a unei alte clase fără a ştii clasa acesteia:



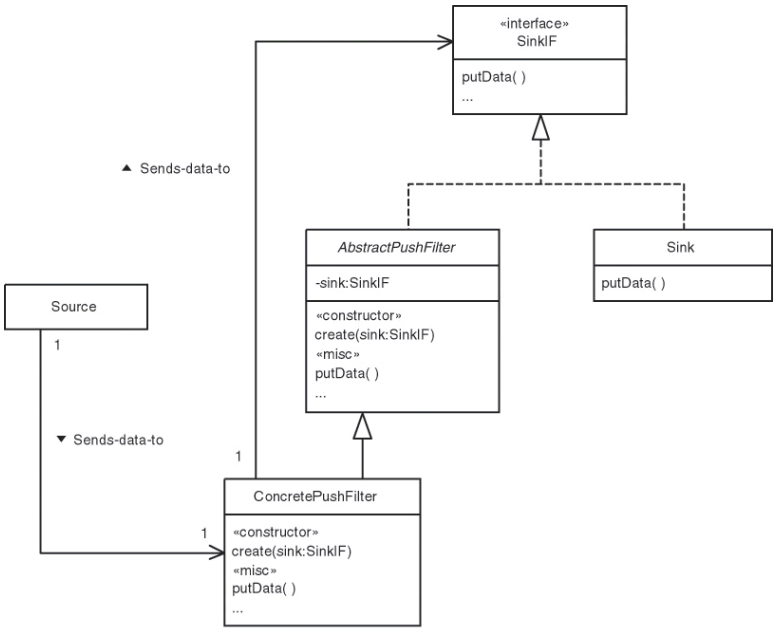
Avantaje:
- Aceste clase se pot refolosi
- Ele se pot combina dinamic
- Folosirea obiectelor este transparentă între ele

Soluţii

1. Filtru Pull



2. Filtru Push



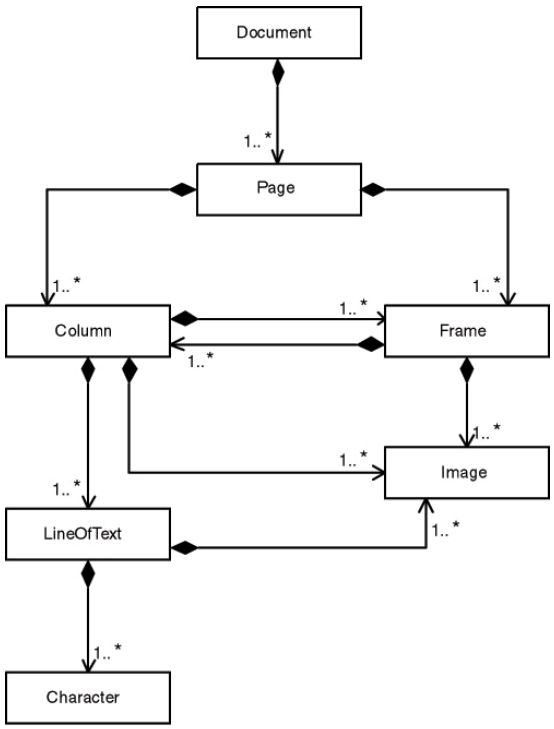
Exemplu: clase filtru din java.io - BufferedReader, FileReader, and LineNumberReader, BufferedWriter, FileWriter, and PrintWriter.

Exerciţiu: construiți o aplicație ce folosește filtre pentru procesarea șirurilor de caractere: PrintFilter, UpperCaseFilter, LengthFilter (elimină cuvintele din string de lungime mai mică decât specificat în constructor).

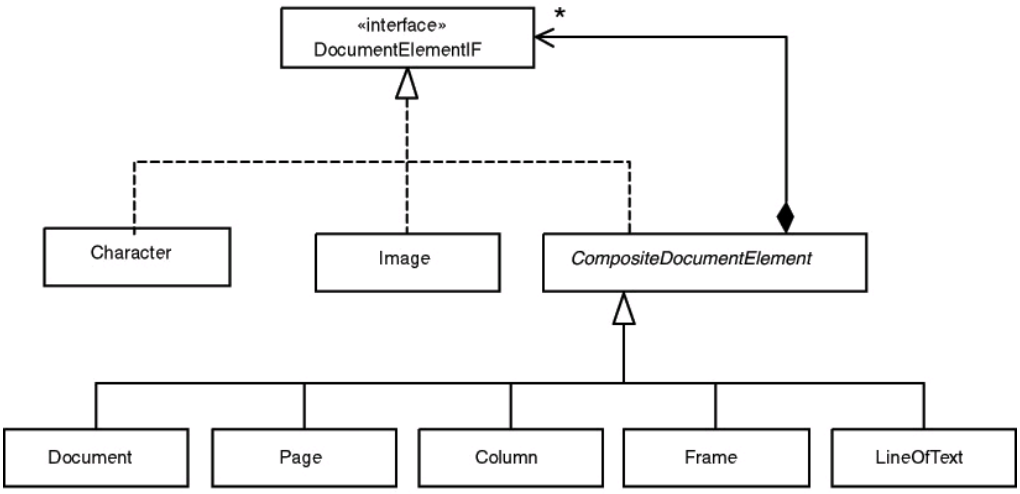
5.2 Compozit

Permite construirea de obiecte complexe prin compunerea recursivă de obiecte similare într-o structură tip arbore. Obiectele din arbore se manipulează într-un stil consistent prin implementarea unei interfeţe comune sau extinderea unei superclase.

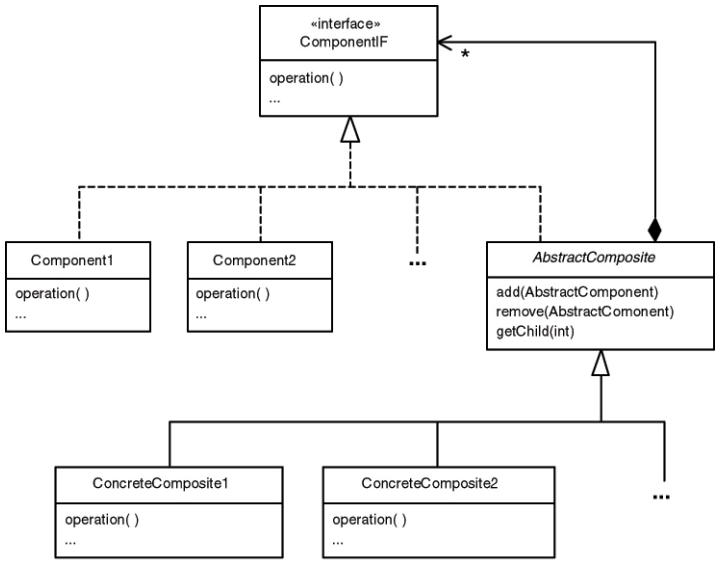
Context



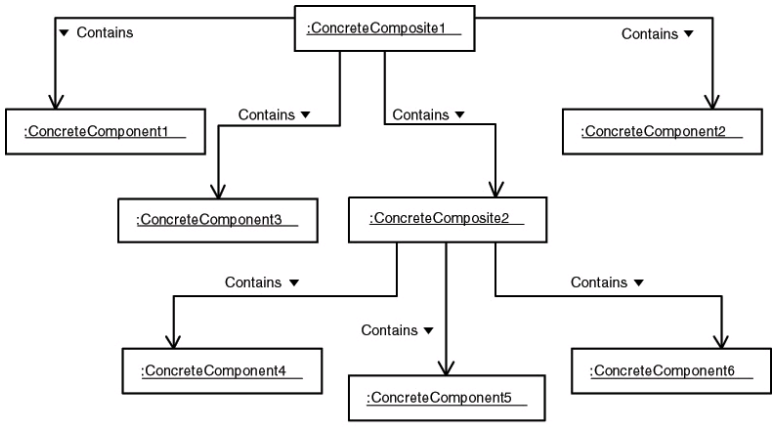
Reorganizare



Soluţie



Exemplu de obiect compozit



Exemplu: java.awt – containere şi controale

Exerciţii:

- aplicați compozit pentru a construi un calculator compus din mai multe componente (CPU, memorie, HDD, etc.). Definiţi o metoda calculeazaPret() pentru calculator ce va aduna pretul fiecarei componente.
- aplicați pattern compozit pentru a grupa forme în editorul grafic 2D; operații move(), arie()

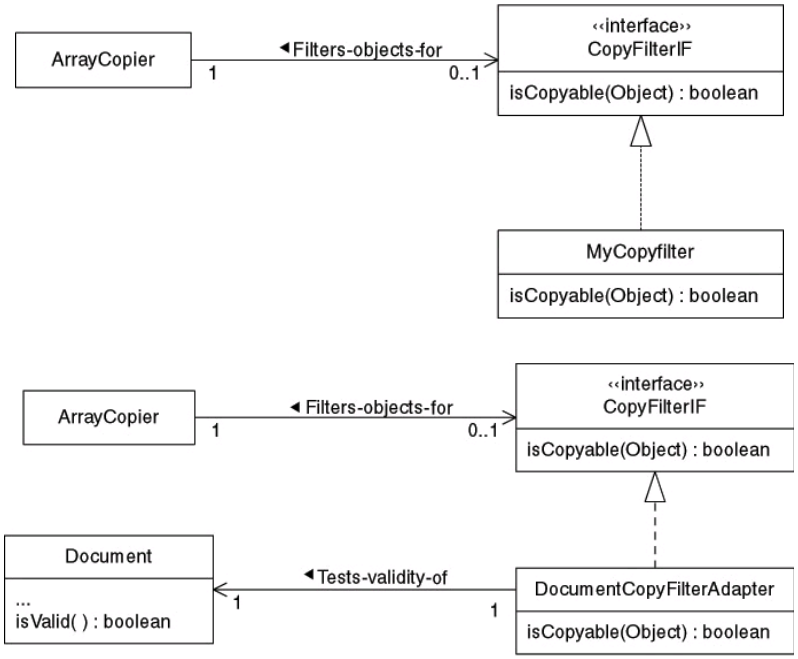
6 Design Patterns Structurale

Acest capitol descrie modalităţi comune în care se pot organiza obiecte de diferite tipuri pentru a lucra împreună.

6.1 Adaptor

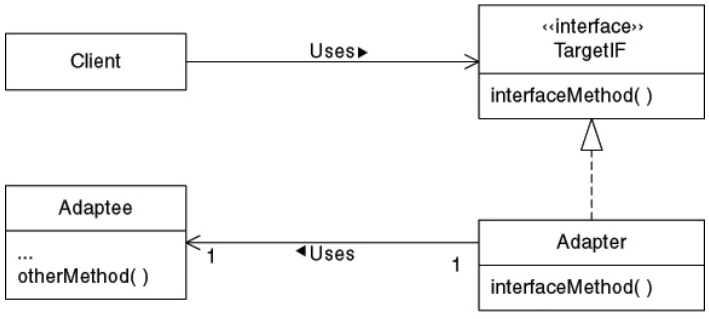
O clasă adaptor implementează o interfaţă cunoscută de clienţii ei pentru a oferi acces la o instanţa a unei clase ce nu este cunoscută de acei clienţi.

Context – criteriul de filtrare se izolează, copierea este generală



Observaţie: clasa este „necunoscută” nu se poate modifica fie că nu avem acces la codul ei, fie că este o clasă generală folosită în multe locuri.

Soluţie



Există **class adapter** – se foloseşte moştenire multiplă: adaptorul moşteneşte adaptatul şi interfaţa folosită de client.

Implementări:

- Adaptor ţine o referinţă la adaptat
- Adaptorul este clasă internă a adaptatului:

```
MenuItem exit = new MenuItem(caption);
exit.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent evt) {
                close();
                …..
        }
}
```

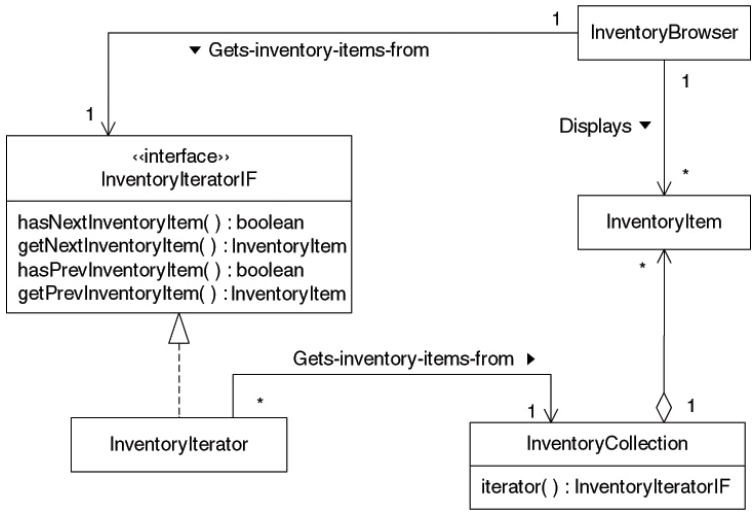
Exemplu: tratarea evenimentelor AWT, Swing

Exerciţiu: ilustraţi acest şablon folosind o clasă internă anonimă sau clasă internă nestatică.

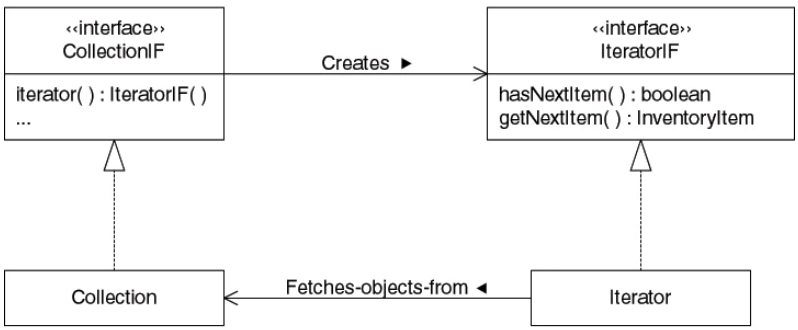
6.2 Iterator

Defineşte o interfaţă ce declară metode pentru accesarea secvenţială a obiectelor dintr-o colecţie. O clasă ce accesează o colecţie doar printr-o asemenea interfaţă este independentă de clasa ce implementează interfaţa şi de clasa colecţiei.

Context



Soluţie



Observaţii:

- Iteratorul se poate personaliza pe capabilităţile colecţiei sau a obiectului compozit
- Se poate implementa ca şi clasă internă
- Probleme dacă se modifică colecţia în timpul traversării
- Iteratorii pot fi interni sau externi – depinde de cine controlează accesul secvenţial; este intern dacă clientul spune doar ce să se facă pentru obiecte (for)
- Se pot defini iteratori specializaţi pe câte un algoritm de parcurgere (iteratori polimorfici generaţi de o metodă factory).

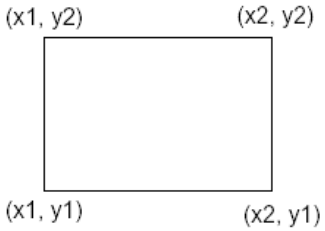
Exemplu: colecţiile din Java

Exerciţiu: pentru o clasă Inventar definiţi un iterator pentru accesul secvenţial la obiectele din inventar. Accesul se poate face după anumite criterii.

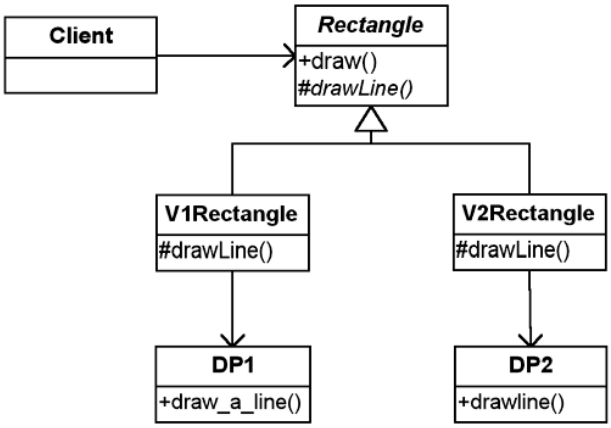
6.3 Bridge

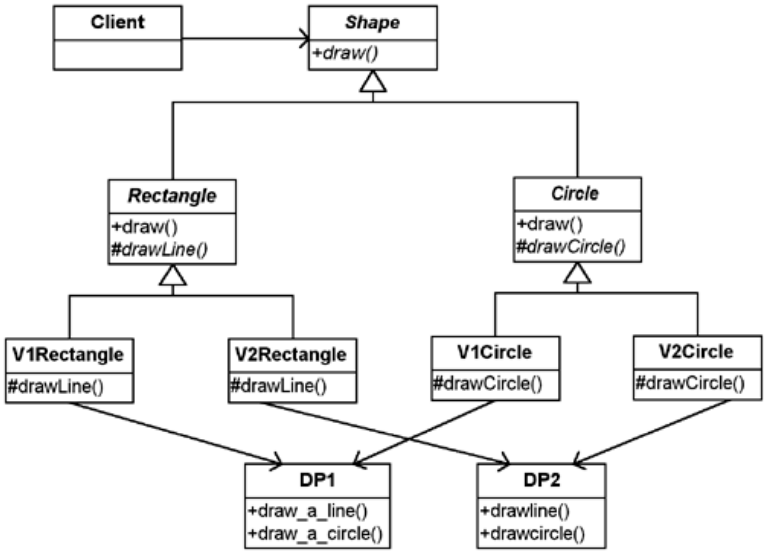
Este folositor când există o ierarhie de abstracţii şi ierarhii corespunzătoare de implementări. Acestea se combină dinamic.

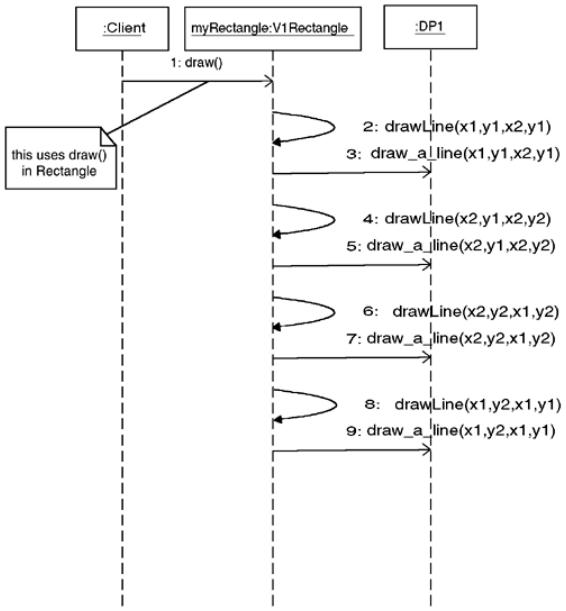
Context

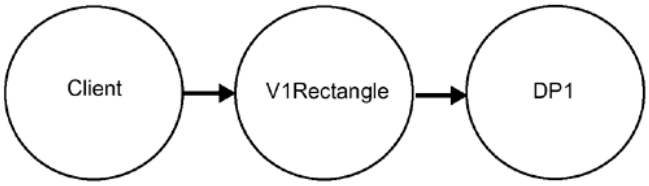


	DP1	DP2
draw a line	`draw_a_line( x1, y1, x2, y2)`	`drawline( x1, x2, y1, y2)`
draw a circle	`draw_a_circle( x, y, r)`	`drawcircle( x, y, r)`





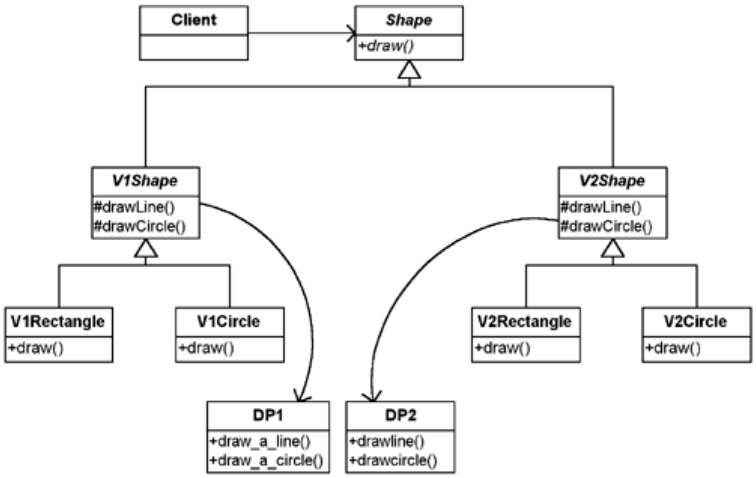


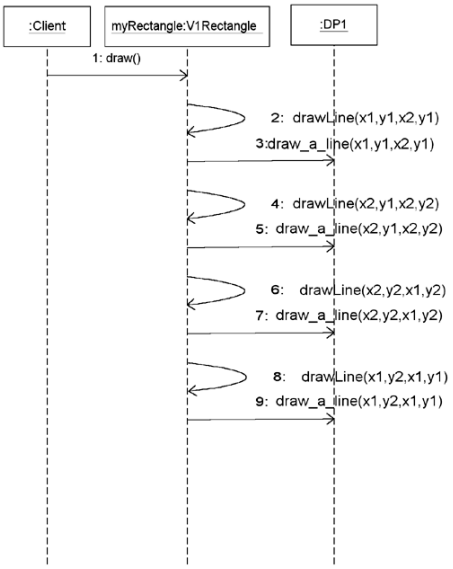


Note:

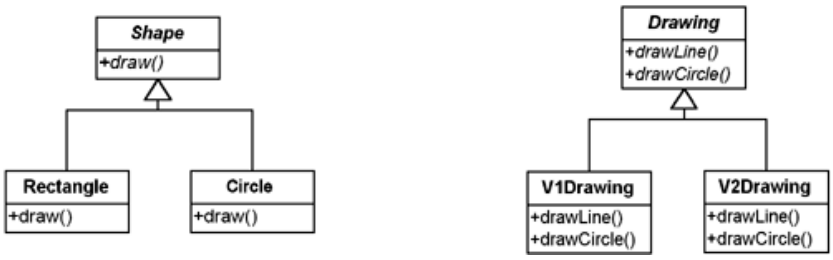
- Mai multe forme, mai multe DP → explozie de clase
- Abstractia (forme) si implementarea (DP) sunt prea cuplate

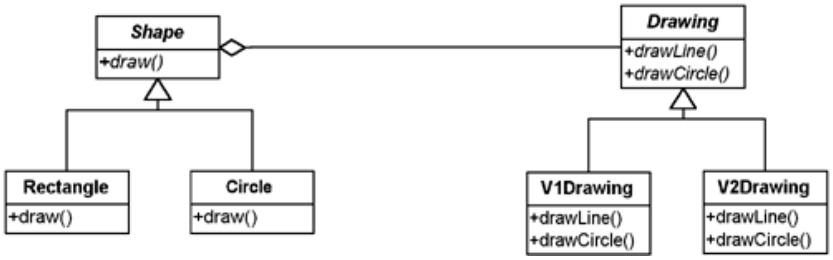
- Design alternativ:

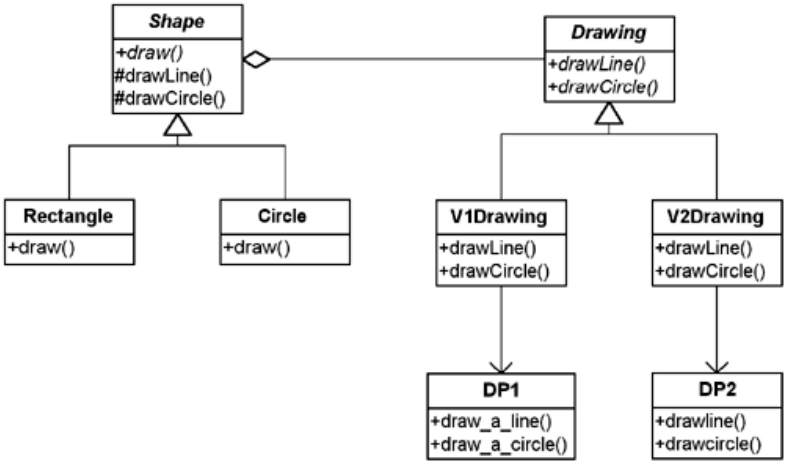


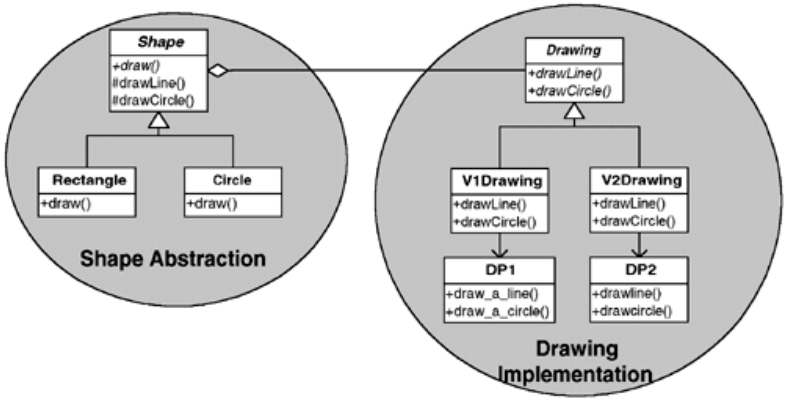


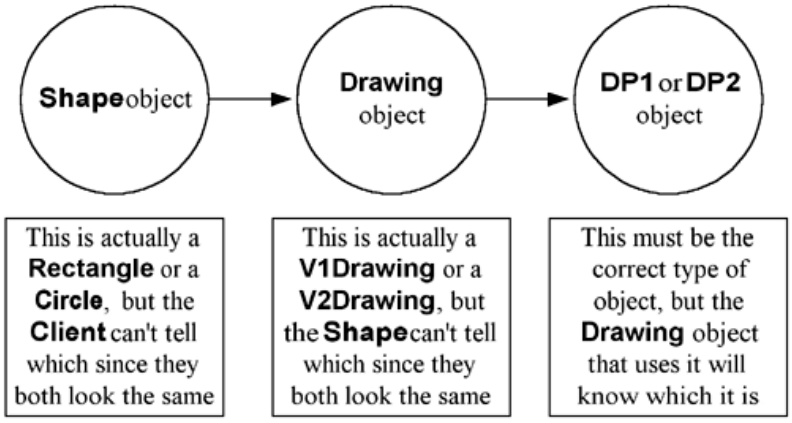
- elementele ce variaza:

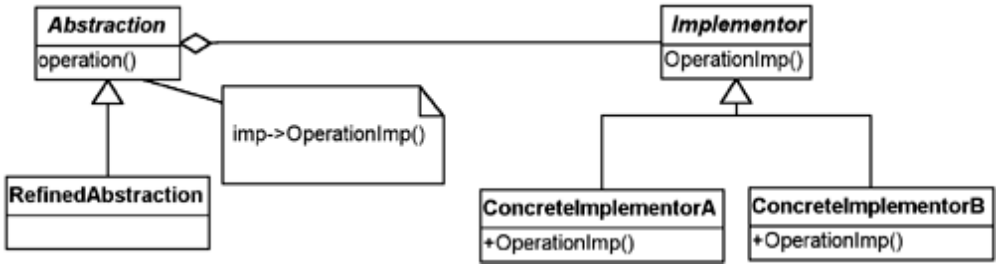

Shape
+draw()
Rectangle
+draw()
Circle
+draw()
Drawing
+drawLine()
+drawCircle()
V1Drawing
+drawLine()
+drawCircle()
V2Drawing
+drawLine()
+drawCircle()


Shape
+draw()
Rectangle
+draw()
Circle
+draw()
Drawing
+drawLine()
+drawCircle()
V1Drawing
+drawLine()
+drawCircle()
V2Drawing
+drawLine()
+drawCircle()


Shape
+draw()
#drawLine()
#drawCircle()
Rectangle
+draw()
Circle
+draw()
Drawing
+drawLine()
+drawCircle()
V1Drawing
+drawLine()
+drawCircle()
V2Drawing
+drawLine()
+drawCircle()
DP1
+draw_a_line()
+draw_a_circle()
DP2
+drawline()
+drawcircle()


Shape
+draw()
#drawLine()
#drawCircle()
Rectangle
+draw()
Circle
+draw()
Shape Abstraction
Drawing
+drawLine()
+drawCircle()
V1Drawing
+drawLine()
+drawCircle()
V2Drawing
+drawLine()
+drawCircle()
DP1
+draw_a_line()
+draw_a_circle()
DP2
+drawline()
+drawcircle()
Drawing Implementation



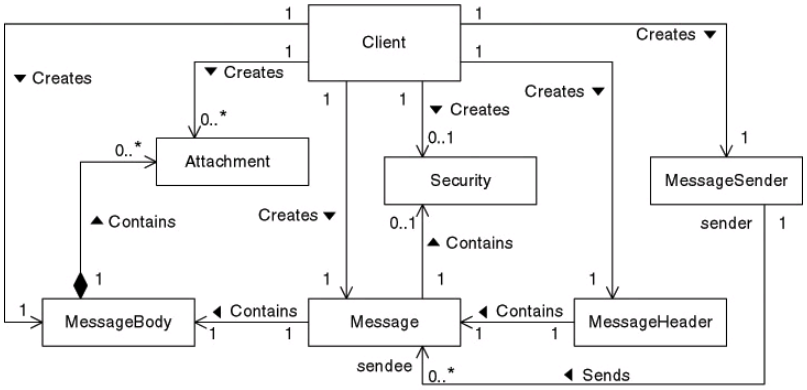


Exerciţiu: Propuneţi o implementare pentru senzori de diferite tipuri (de ex. pentru valori instantanee sau valori medii) produse de diferiți producători.

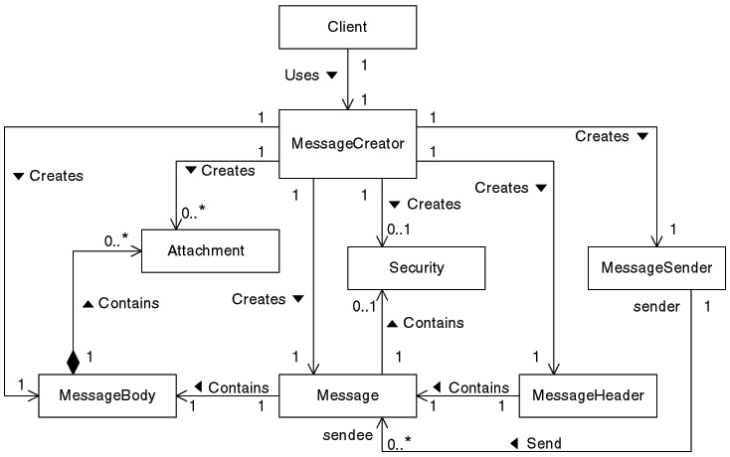
6.4 Façade

Simplifică accesul la un set de obiecte (ce au legătură între ele) prin intermediul unui obiect care ascunde în spate setul de obiecte.

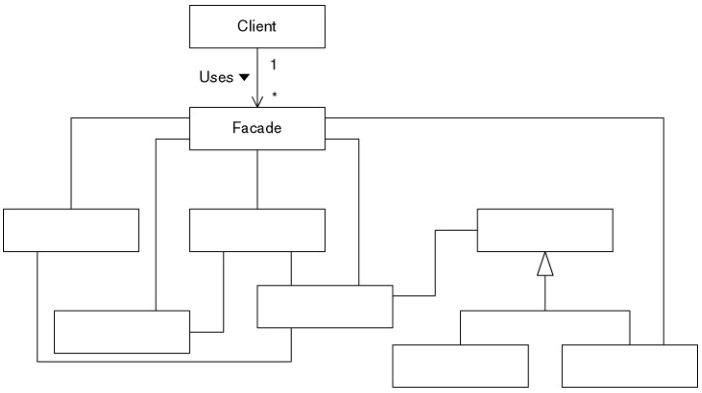
Context



Pentru a ascunde clientului detaliile de construire a mesajului:



Soluţie

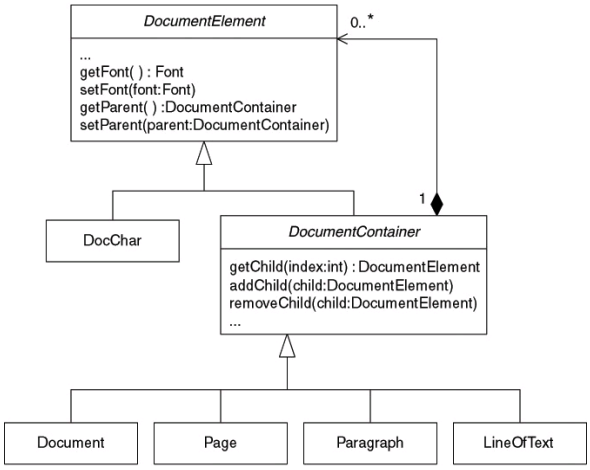


Exerciţiu: Propuneţi o implementare a şablonului façade pentru situaţia din context.

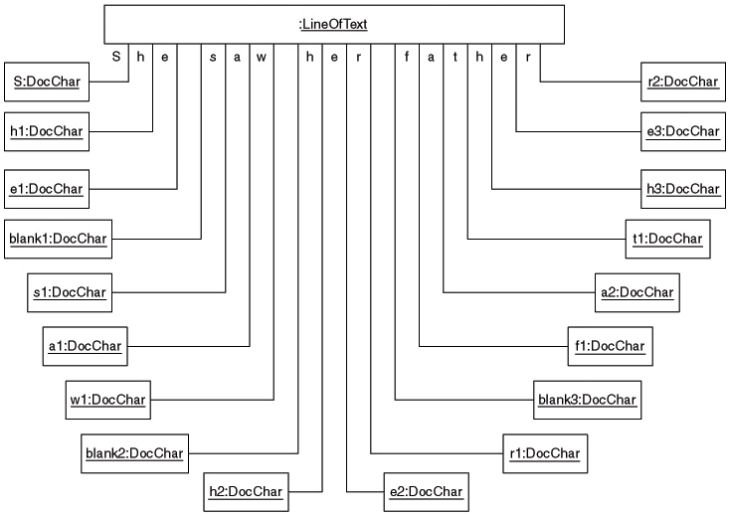
6.5 Flyweight

Dacă instanţele unei clase ce conţin aceeaşi informaţie se pot înlocui una pe alta atunci se pot înlocui instanţe multiple ale aceluiaşi obiect cu o singură instanţă.

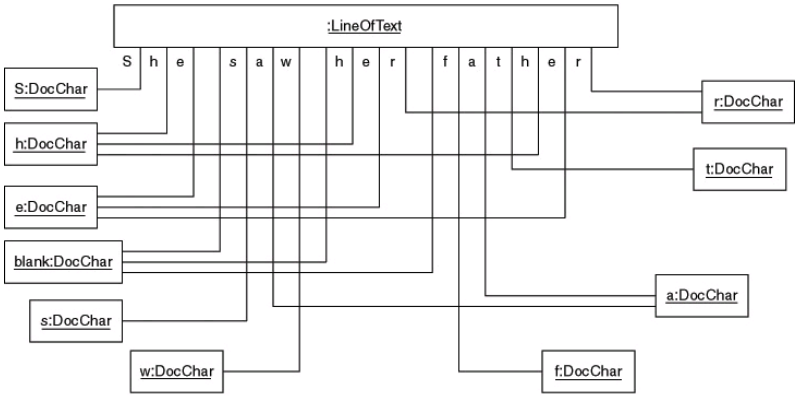
Context



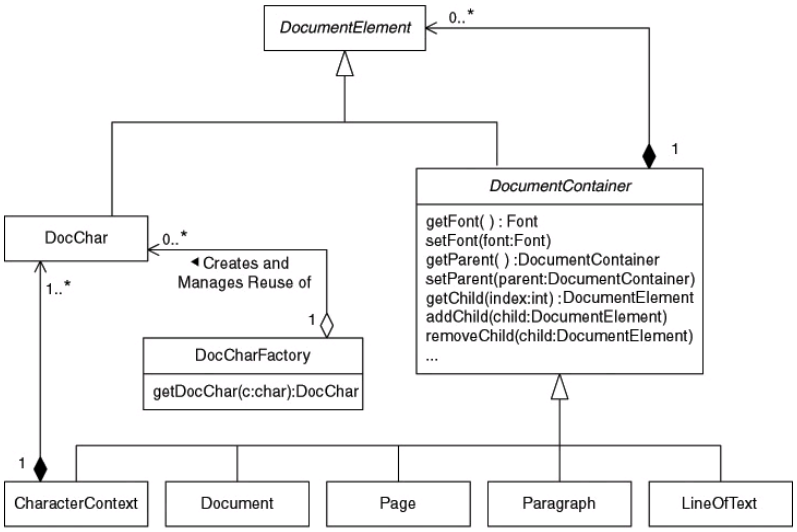
La un moment dat putem avea:



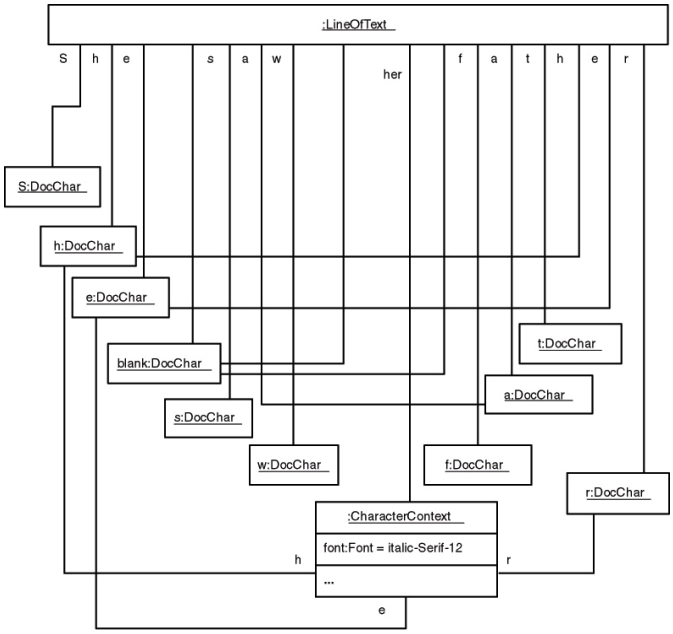
Se poate utiliza un DocChar pentru a reprezenta acelaşi caracter:



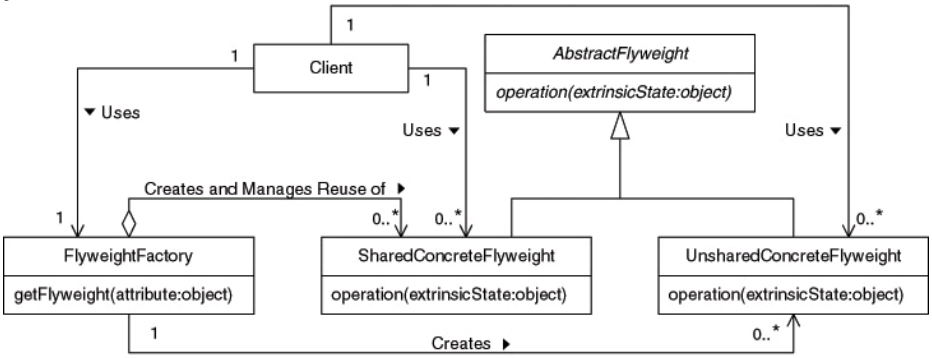
Informaţiile legate de font se scot din DocChar:



Acum organizarea obiectelor arată aşa:



Soluţie



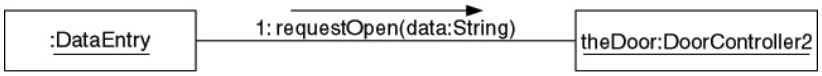
Exemplu: tratarea literalilor String in Java

Exerciţiu: implementaţi soluţia propusă în secţiunea context

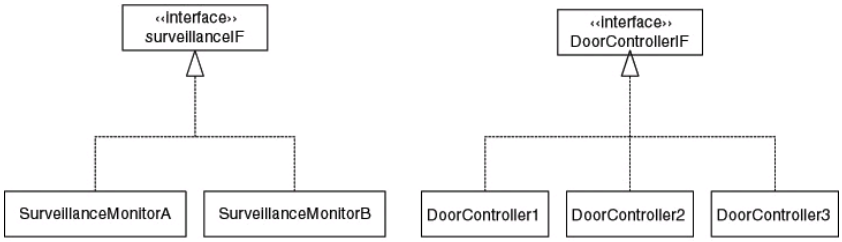
6.6 Decorator

Se mai numeşte şi Wrapper. Acest şablon propune extinderea unui obiect într-un mod transparent pentru clienţii lui prin implementarea aceleiaşi interfeţe ca şi clasa originală şi prin delegarea operaţiilor către clasa originală.

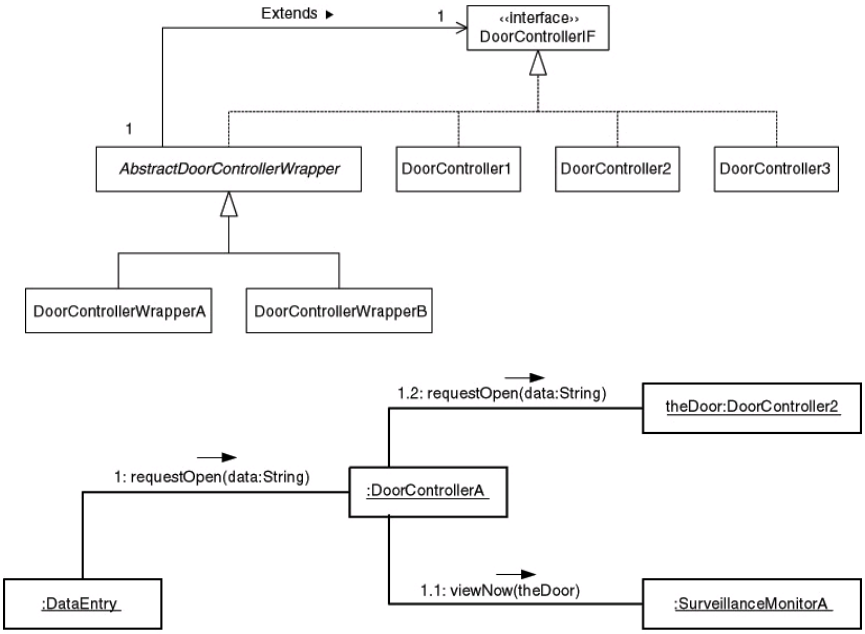
Context
Sistem de securitate pentru controlul accesului într-o clădire.



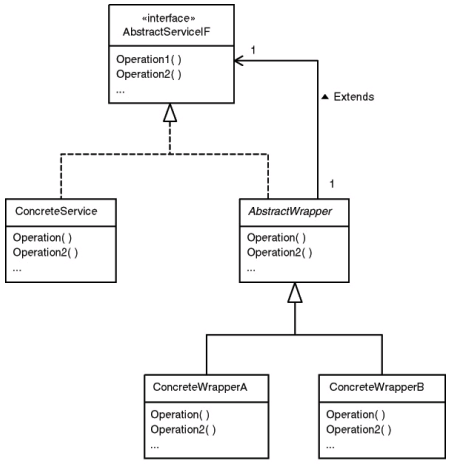
Monitor, camere de luat vederi; problema monitorizării momentului în care o persoană intră pe o uşă.



Rezolvare



General



Note:

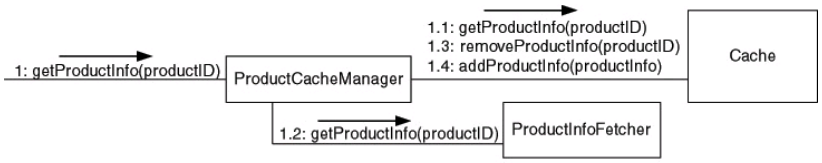
- Dacă există un singur ConcreteService atunci Abstract Wrapper poate să subclaseze ConcreteService
- Dacă există un singur ConcreteWrapper se poate renunţa la AbstractWrapper

Exerciţiu: propuneţi o implementare pentru scenariul descries în secţiunea context.

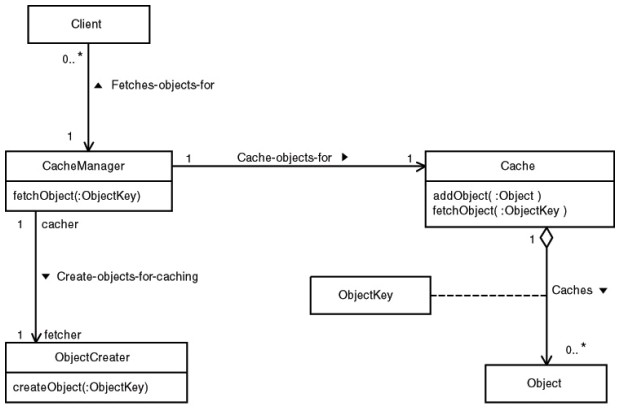
6.7 Cache Management

Permite accesul rapid la obiecte pentru care altfel s-ar consuma mult timp cu crearea sau accesul la ele. Presupune ţinerea unei copii a obiectului ce ar fi costisitor de construit (din diferite motive, de exemplu accesul la o DB).

Context
Informaţii despre produse; politică de gestiune a cache-ului (câte obiecte, care sunt reţinute)



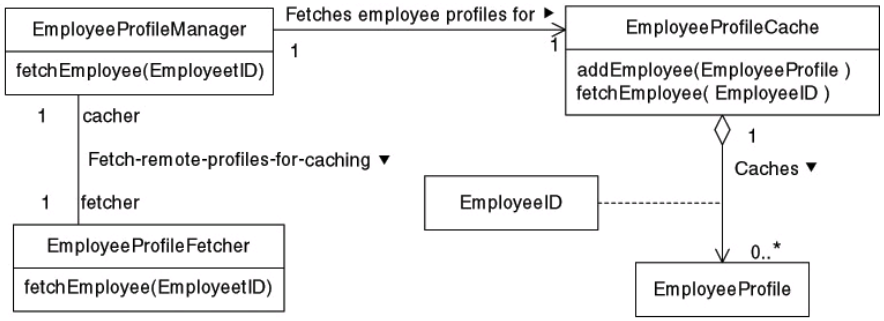
General



Note:

- CacheManager poate avea aceeaşi interfaţă ca şi ObjectCreater pentru a fi transparent pentru client
- Adesea pentru implementarea lui Cache se foloseşte HashMap sau HashTable
- Hit rate
- Read consistency (cache-ul urmăreşte modificările din sursă) – write consistency (sursa urmăreşte modificările din cache)

Exerciţiu: propuneţi o implementare a şablonului cache management pentru cazul prezentat mai jos:


EmployeeProfileManager
fetchEmployee(EmployeetID)
Fetches employee profiles for
1
1
EmployeeProfileCache
addEmployee(EmployeeProfile)
fetchEmployee(EmployeeID)
1
cacher
Fetch-remote-profiles-for-caching
1
fetcher
EmployeeProfileFetcher
fetchEmployee(EmployeetID)
1
Caches
EmployeeID
0..*
EmployeeProfile

7 Design Patterns Comportamentale

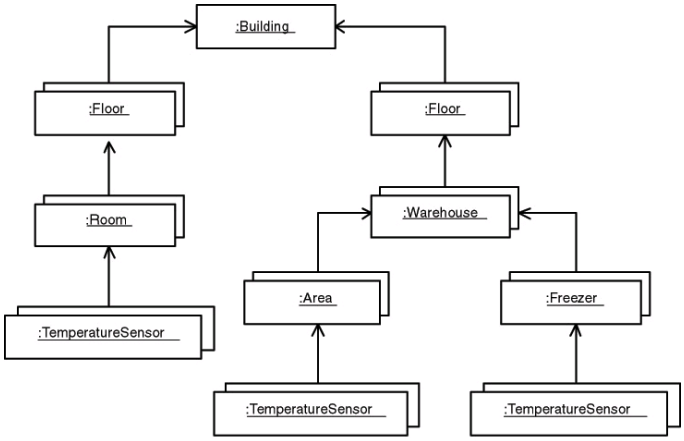
Sunt folosite pentru a organiza, gestiona şi combina comportări.

7.1 Chain of Responsibilities

Permite unui obiect să trimită o comandă fără ca acesta să ştie la cine va ajunge. Acest lucru îl realizează prin pasarea comenzii unui şir de obiecte ce sunt parte dintr-o structură mai mare. Fiecare obiect din şir poate trata comanda şi/sau pasa comanda următorului obiect din şir.

Context – sistem de securitate: senzori ce trimit informaţii ce sunt înregistrate, gestionate stările, generare de alarme. Scalabilitate.

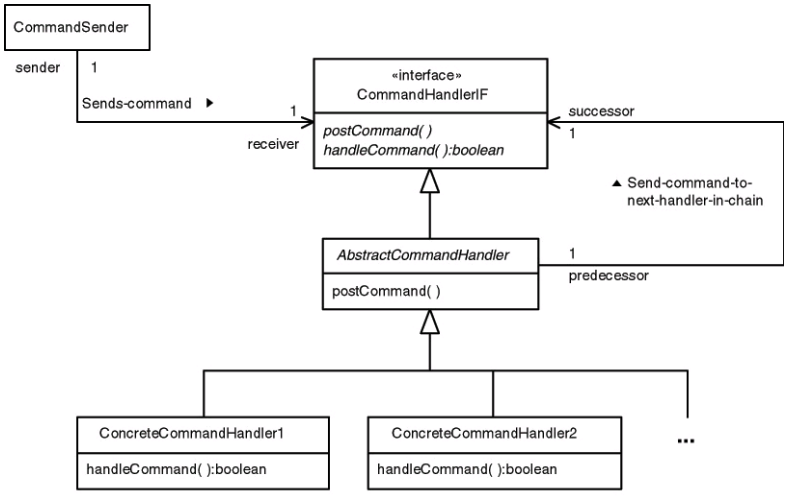
Organizare senzori:



Note:

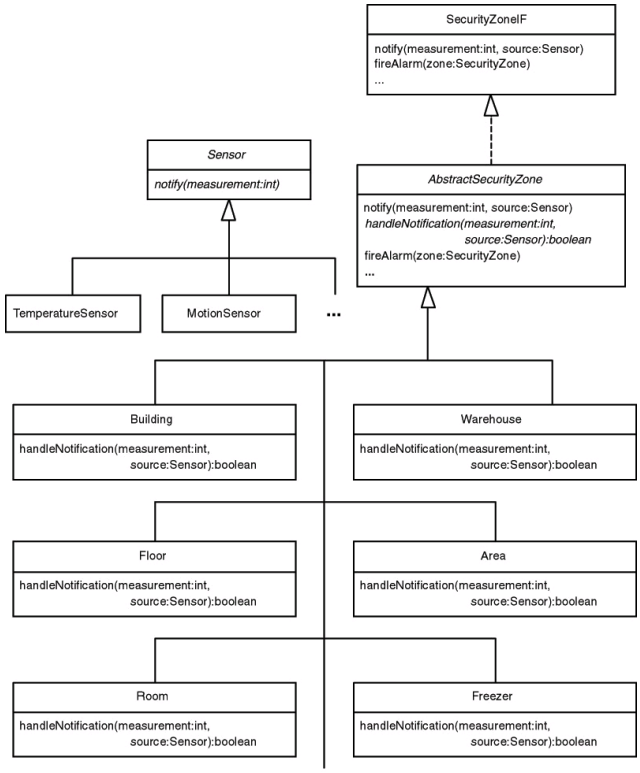
- Obiectul ce trimite comanda nu ştie cine va trata mesajul
- Tratatea mesajelor nu presupune cunoaşterea sursei
- Pot exista mai mulţi interesaţi de mesaj

Soluţie:



Exemplu: trimiterea evenimentelor generate de utilizator către componente

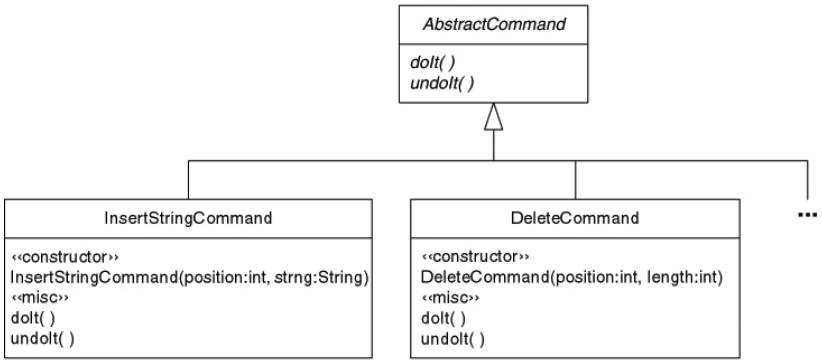
Exerciţiu: propuneţi o implementare pentru scenariul prezentat în secţiunea context.



7.2 Command

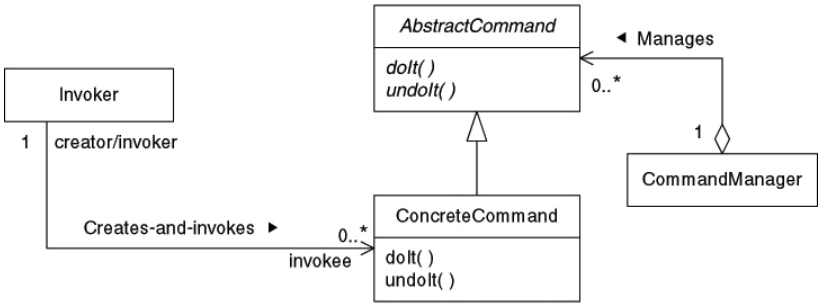
Încapsulează comenzi în obiecte pentru a se controla manipularea lor (selecţie, secvenţiere, execuţia, undo, etc.).

Context – procesor text care permite undo şi redo pentru comenzi.

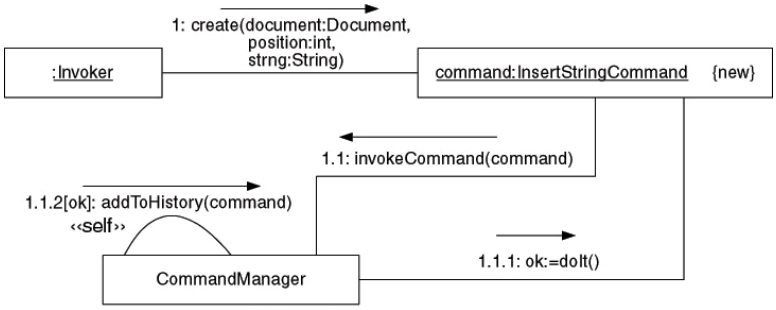


Comenzile sunt ţinute într-un istoric.

Soluţie:



Exerciţiu: oferiţi o implementare pentru scenariul descris în secţiunea context.



Exerciţiu: implementați tranzacții peste grupe de comenzi

7.3 Interpretor (mic limbaj)

Se folosește pentru rezolvarea de problem similare a căror soluție se poate exprima prin combinarea unui număr mic de elemente sau operații.

Exemple:

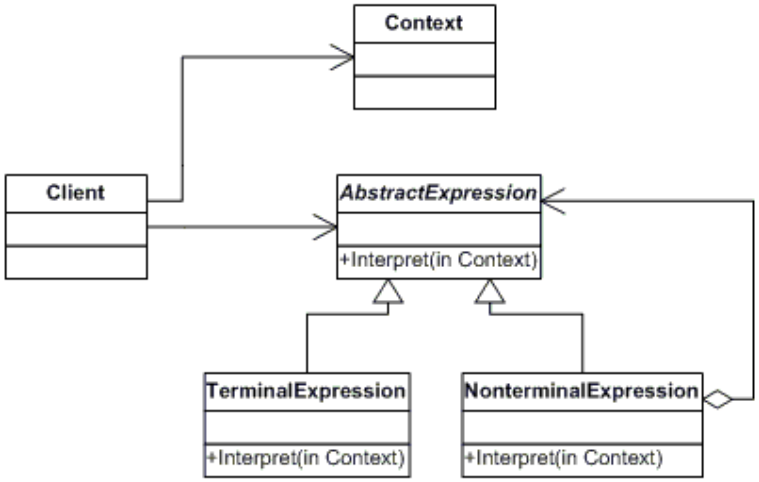
- Căutare potrivirilor într-un șir de caractere cu ajutorul unei expresii regulate
- Evaluarea unei expresii logice: x and y or not z
- Evaluare număr exprimat prin cifre romane
- Transformarea unei expresii din notatie prefix in notatie postfix
- Modalități de afișare a rezultatelor (formatare date): print nume varsta

Context – program pentru căutarea fișierelor ce conțin o combinație dată de cuvinte; se poate define un limbaj pentru exprimarea condiției după care se caută.

search casa or masa and tren

- Cuvinte, “casa, masa” – pentru desemnare combinații de cuvinte
- not, or, and – cuvinte rezervate, precedență
- ( ) – grupare

Analiză lexicală – reguli, cum se identifică elementele (tokens)
Analiză sintactică – reguli pentru identificarea elementelor neterminale; exprimare reguli prin notație Backus-Naur Form (BNF)

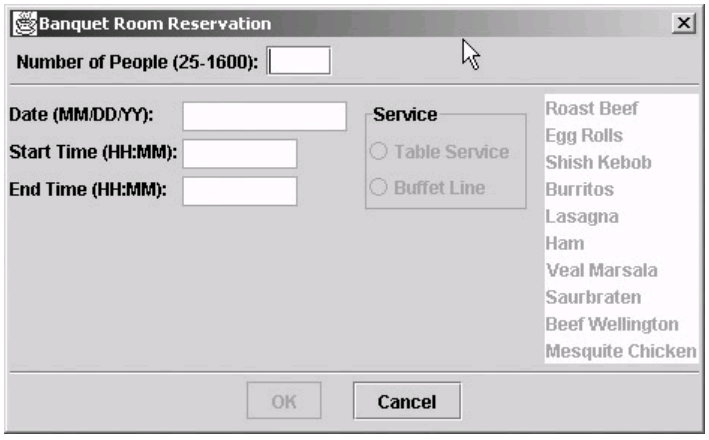


Tema: aplicați acest pattern pentru a genera rapoarte pe baza unei liste de persoane folosind expresii de forma “print prenume nume varsta”, unde print este un verb ce specifica tiparire, ce urmeaza sunt numele campurilor de afisat. Afișarea se face doar cu acele attribute și în ordinea specificată.
Opțional: print nume prenume ASC nume prenume varsta – ordonare după acele câmpuri.

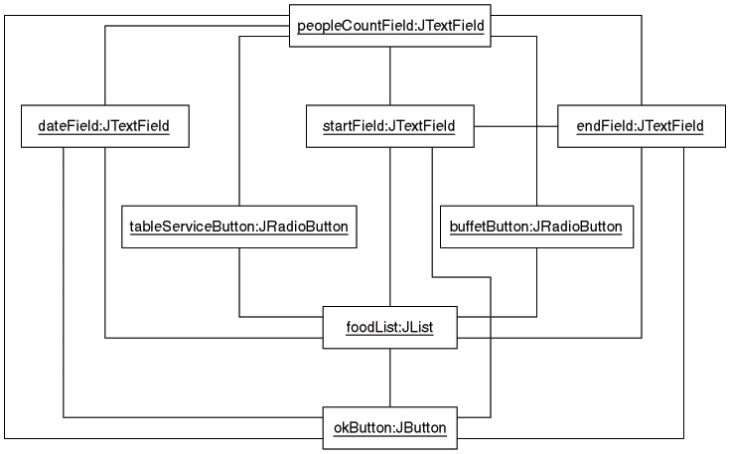
7.4 Mediator

Acest şablon foloseşte un obiect pentru a coordona interdependenţele intre alte obiecte. Logica coordonării stării obiectelor se pune într-un singur loc, nu se distribuie în clasele implicate.

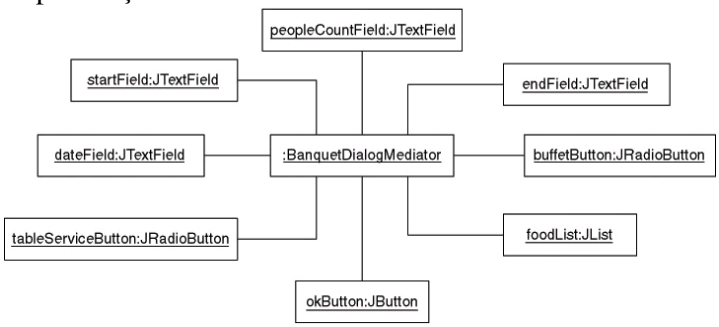
Context – coordonarea dintre stările controalelor



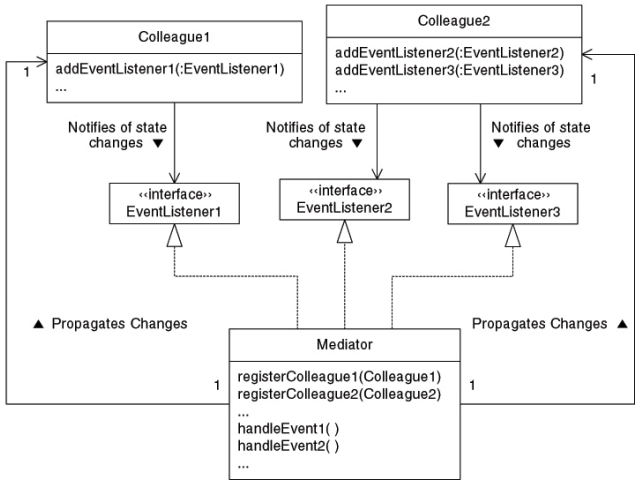
Relaţii dintre obiecte:



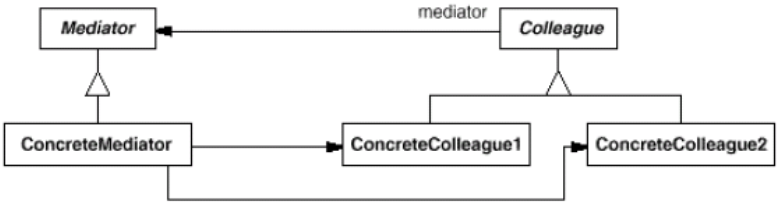
Reorganizare dependenţelor:



Soluţie:



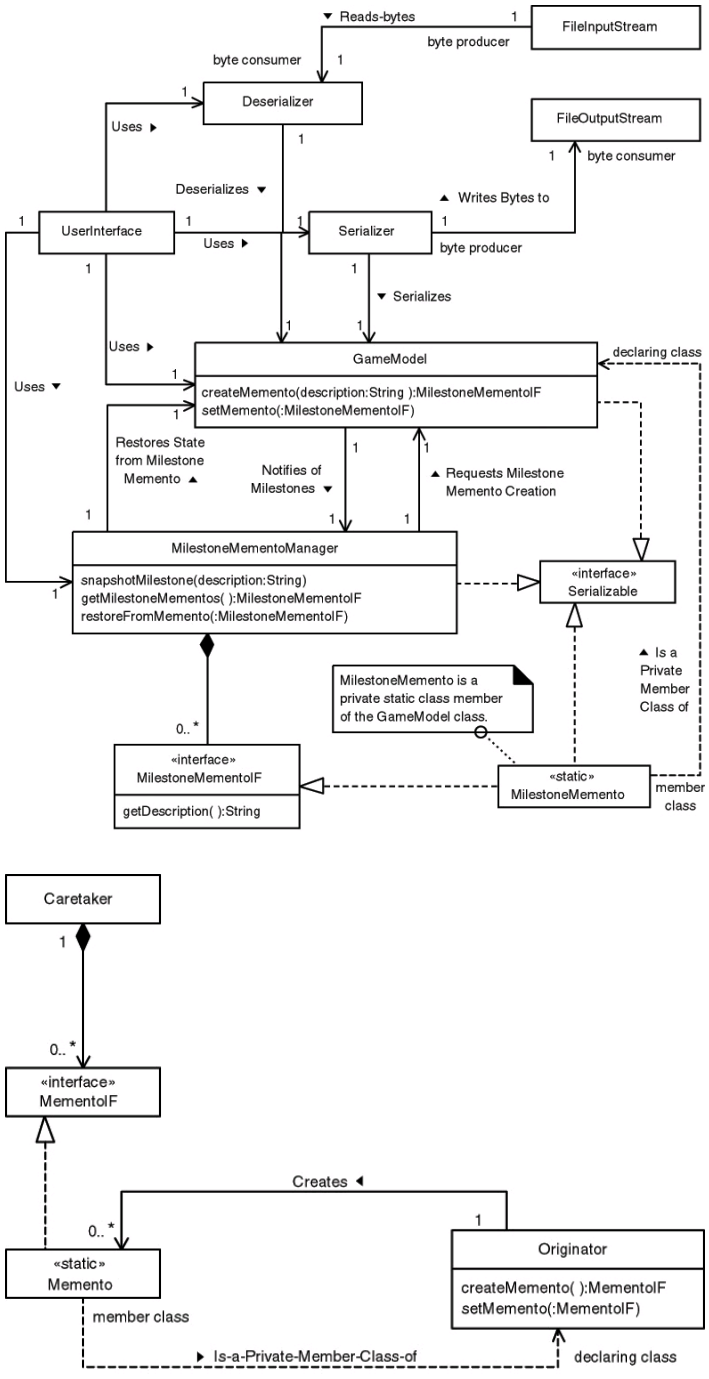
Sau:

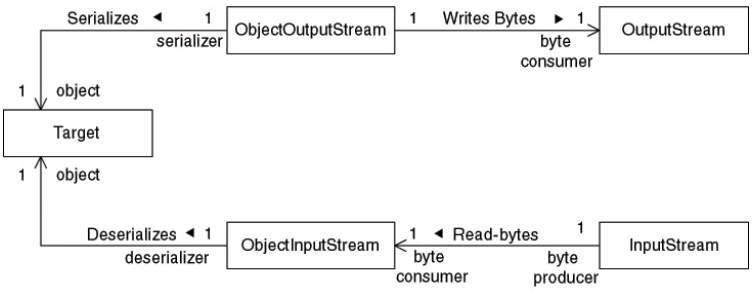


Exerciţiu: daţi un exemplu de mediator oferind şi o implementare

7.5 Snapshot (memento)

Captează starea unui obiect pentru ca ea să se restaureze mai târziu.

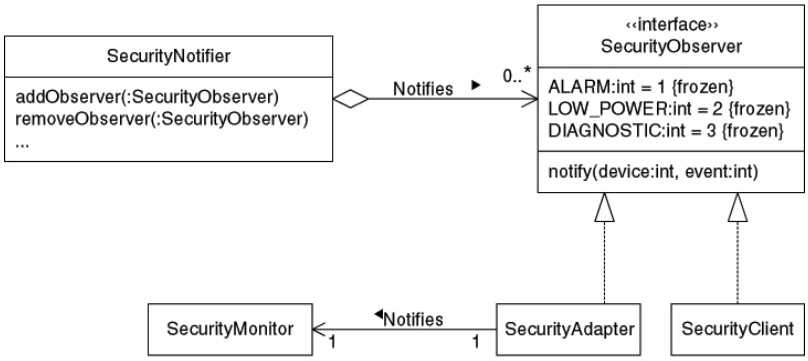




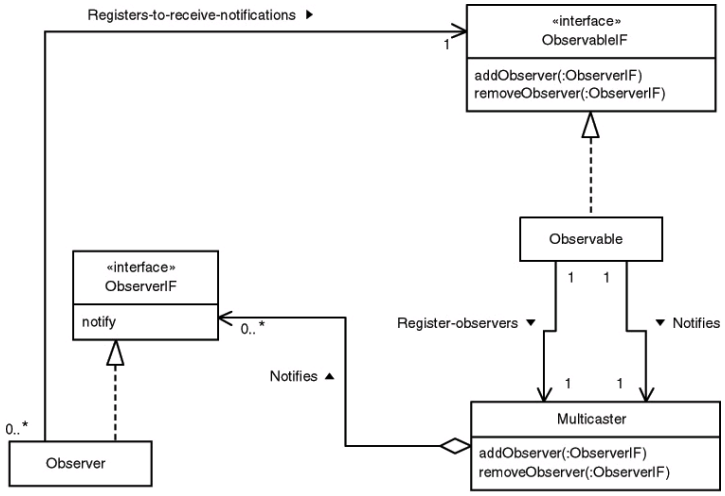
7.6 Observer

Permit obiectelor să înregistreze dinamic dependenţe între ele, astfel un obiect va notifica obiectele interesate că i s-a schimbat starea.

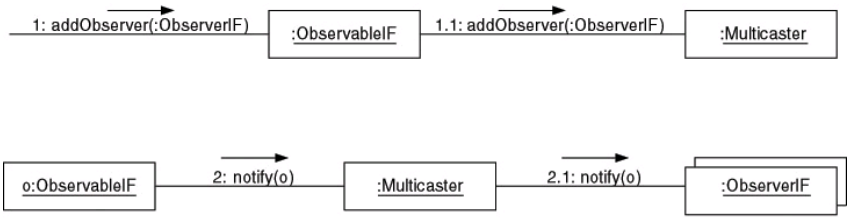
Context



Soluţie



Interacţiune:

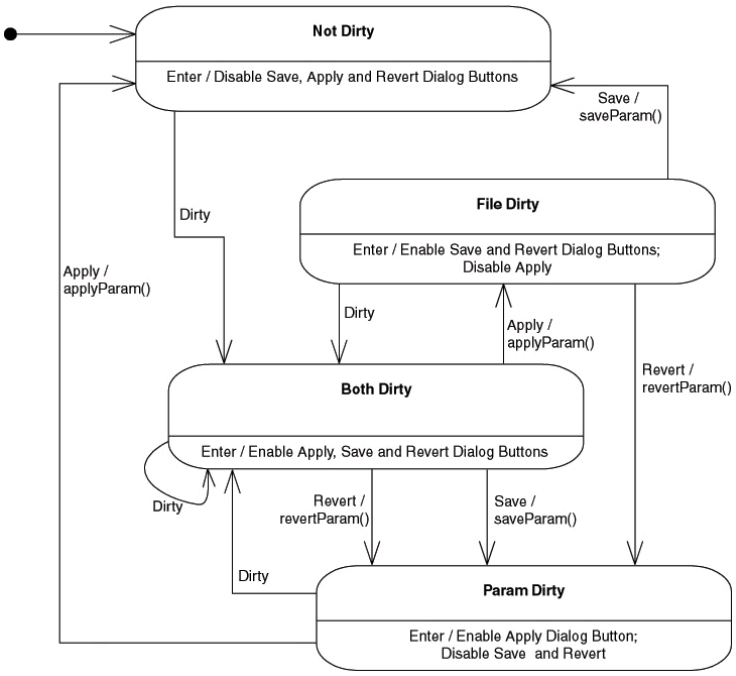


Exerciţiu: daţi un exemplu şi implementaţi acest şablon – (Sudoku)

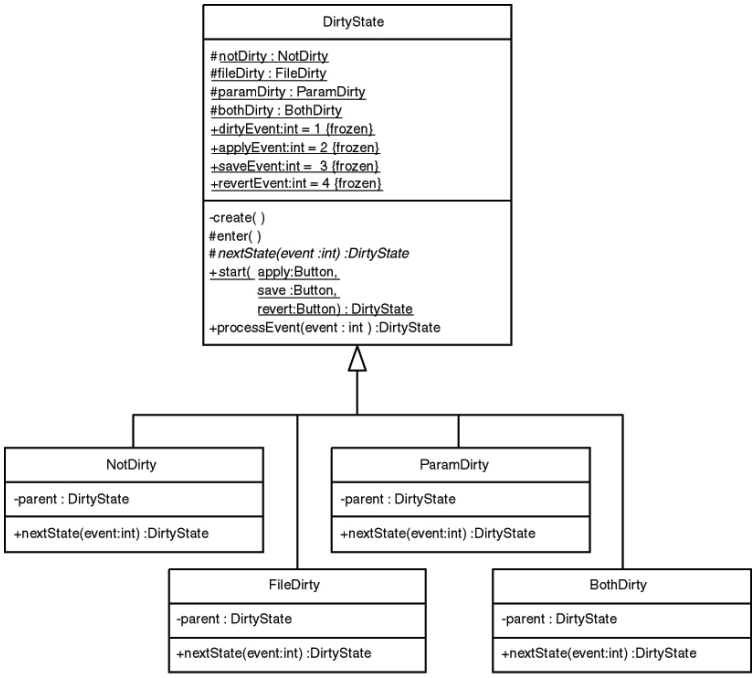
7.7 State

Încapsulează stările unui obiect/sistem ca obiecte discrete.

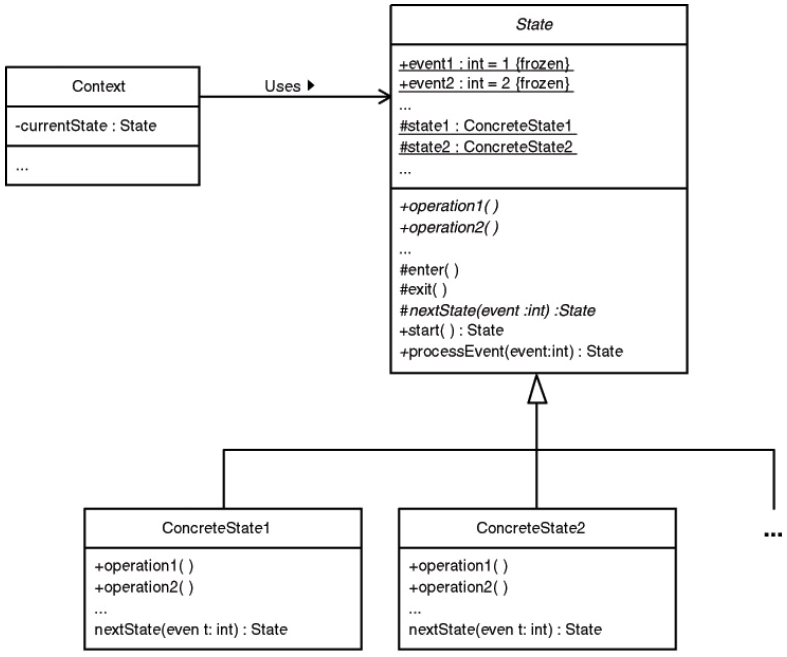
Context – modelarea unui dialog ce editează parametrii unui program; aceştia pot fi salvaţi în memorie sau într-un fişier.



Stările sunt modelate:



General



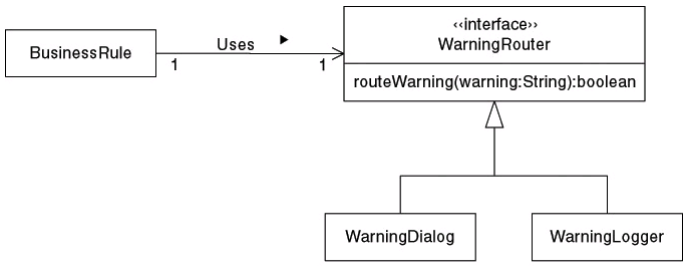
Exerciţii:

- oferiţi o implementare a scenariului descris în context.
- Implementare control bomba
- Implementare control microvawe

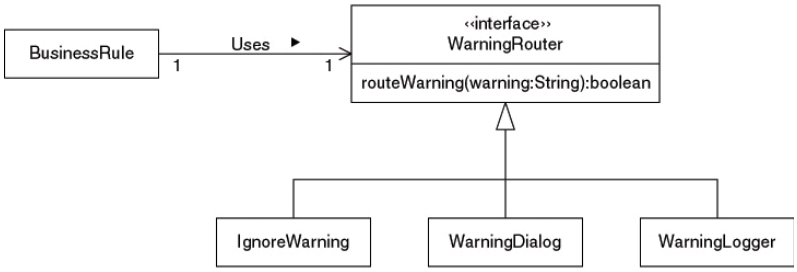
7.8 Null Object

Oferă o alternativă la folosirea lui null pentru a indica absenţa unui obiect delegat. Folosirea lui null presupune un test prealabil înainte de a folosi referinţa; soluţia propusă este de a folosi un obiect special – obiect null – care nu face nimic.

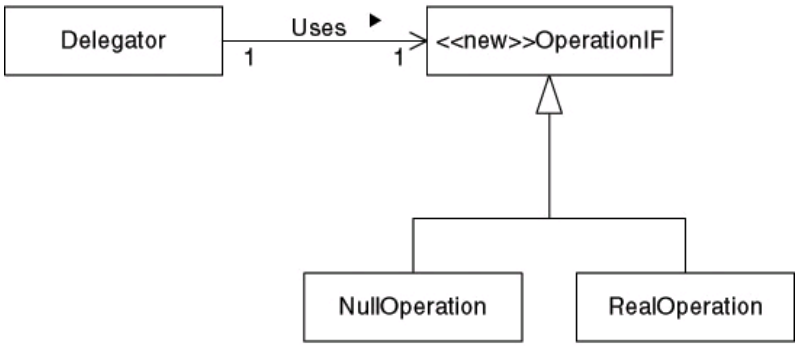
Context



Includem cazul în care se ignoră atenţionarea:



Soluţie

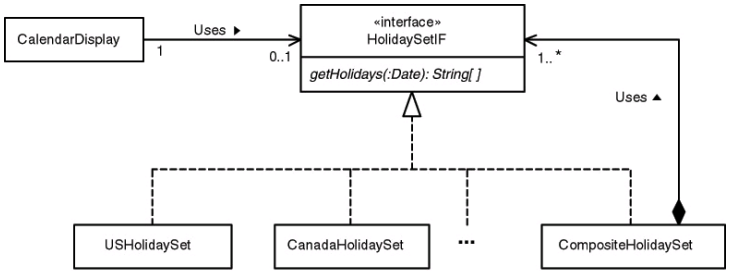


Exerciţiu: aplicaţi acest şablon la cazul din context.

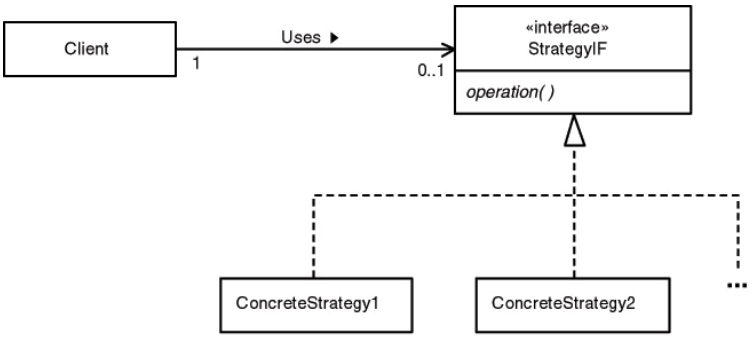
7.9 Strategy

Încapsulează algoritmi (sau comportări) înrudiţi în clase ce implementează o interfaţă comună. Acest lucru permite să selectăm algoritmul prin selecţia obiectului ce implementează algoritmul dorit. Selecţia se poate schimba dinamic.

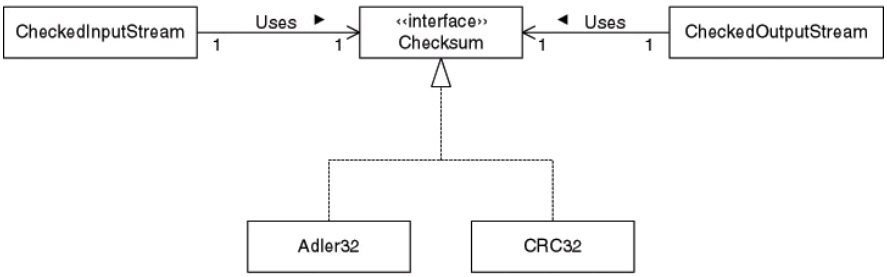
Context



Soluţie



Exemplu: java.util.zip



Exerciţiu: daţi un exemplu de aplicare a acestui pattern, oferiţi o implementare

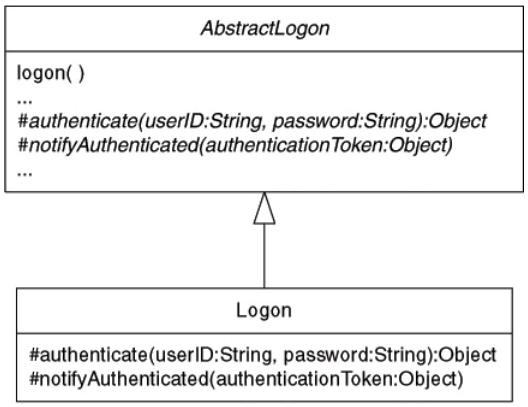
7.10 Template Method

O clasă abstractă conţine doar o parte din logica necesară pentru a face ceva. Clasa se organizează în aşa fel încât metodele ei concrete apelează metode abstracte unde lipsesc părţi din logica implementată. Logica lipsă se furnizează în subclase în metodele ce suprascriu metodele abstracte.

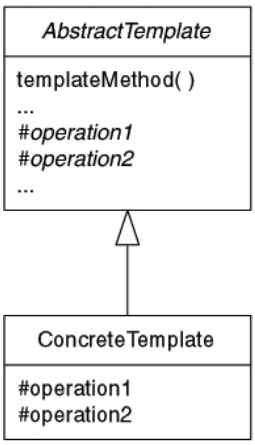
Context – clasă reutilizabilă pentru autentificarea utilizatorilor. Paşi:

1. Cere id şi parolă

2. se face autentificare şi rezultatul este un obiect ce încapsulează informaţia relevantă
3. pe timpul autentificării utilizatorul vede că operaţie este în desfăşurare (formă mouse, imagine sugestivă, etc)
4. notifică aplicaţia de rezultat şi pune la dispoziţie obiectul rezultat



Soluţie

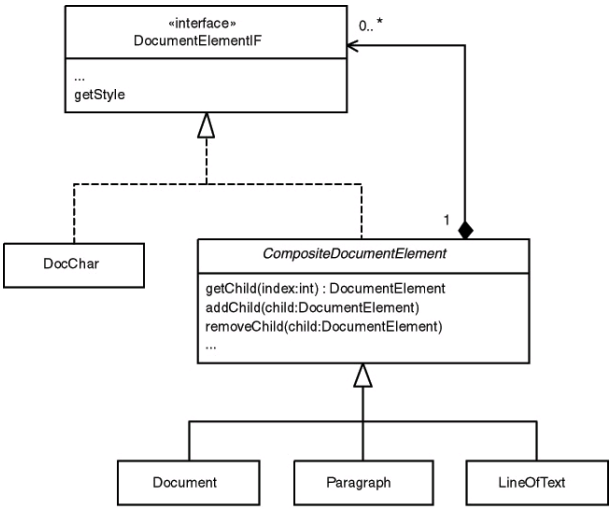


Exerciţiu: daţi un exemplu de aplicare a acestui şablon; oferiţi o implementare.

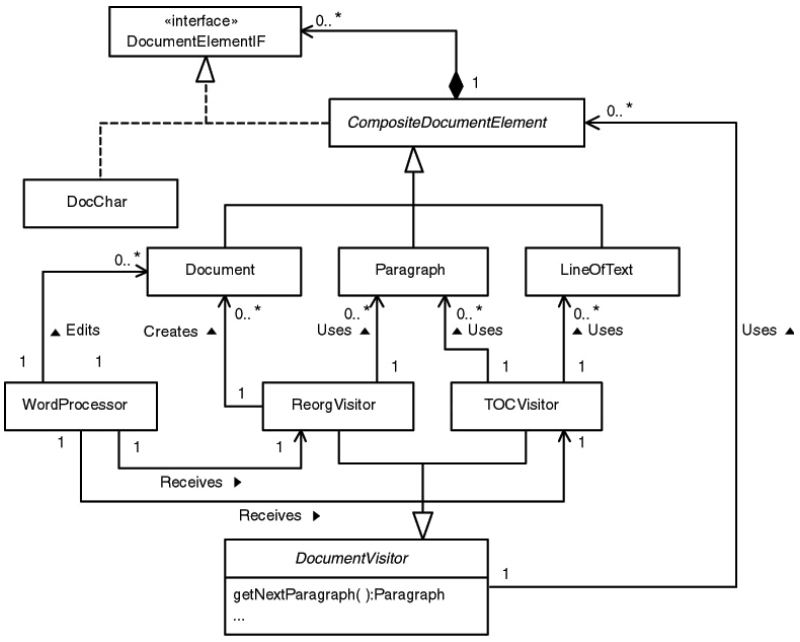
7.11 Visitor

Se aplică la o structură pentru care trebuie definite operaţii; acest şablon sugerează organizarea operaţiilor în clase exterioare structurii de date originale, aceasta din urmă fiind pregătită să suporte această organizare.

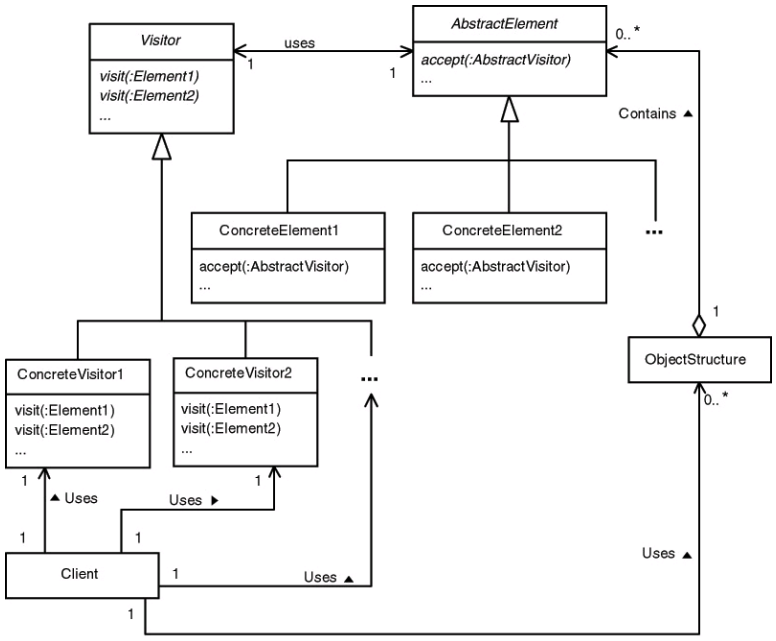
Context – operaţii pe un document



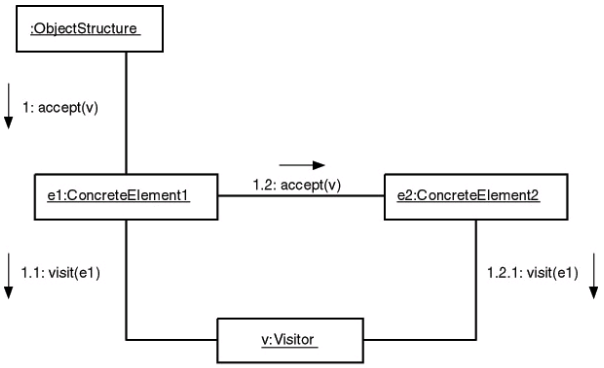
Soluţie



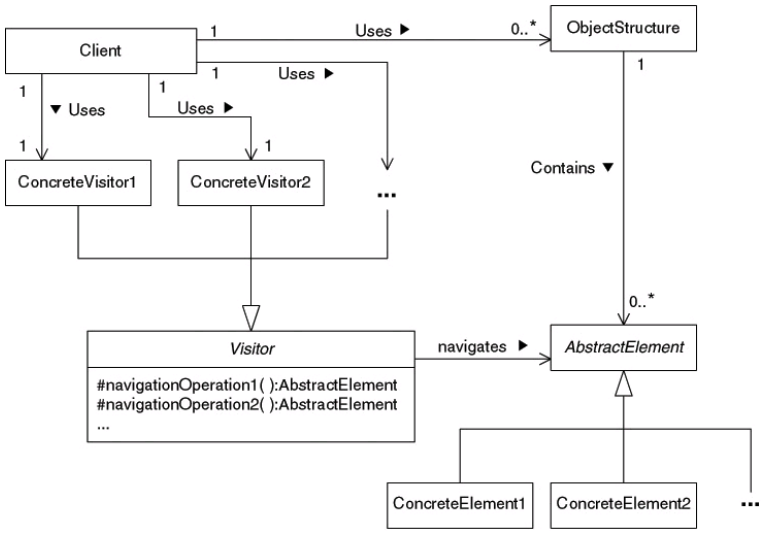
General



Colaborare; structura de date dictează modul de traversare, acelaşi pentru toţi vizitatorii:



Caz în care vizitatorul include şi algoritmul de vizitare:



Exerciţii:

- daţi un exemplu de aplicare a acestui şablon
- calcul arie, pret, greutate pentru un obiect compozit
- double dispatching (operație ce depinde de două obiecte!); aplicare la jocul hartie/foarfeca/piatra

8 Patterns legate de concurenţă

Aceste pattern-uri vizează două tipuri de probleme:

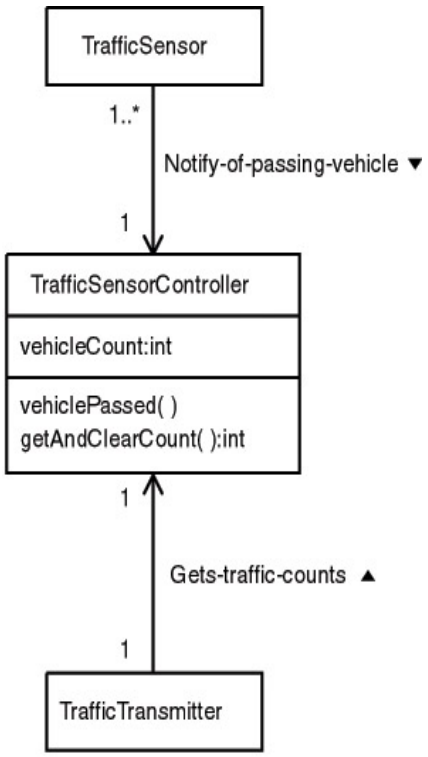
- Accesul concurent la resurse
- Secvenţa operaţiilor pe care le fac mai multe fire de execuţie pe aceleaşi structuri de date

8.1 Execuţia unui singur thread

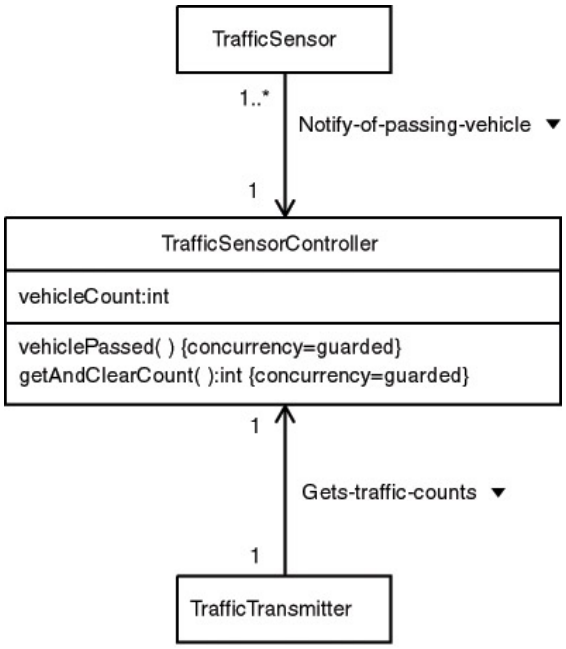
Se mai numeşte şi **secţiune critică**. Accesul la resurse comune se face doar de un thread odată.

Exemplu:

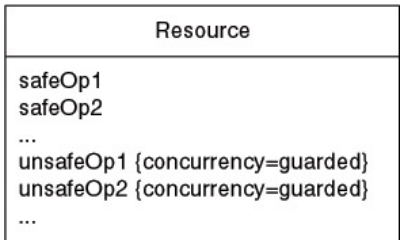
- Fiecare senzor are corespondent o instanţă TrafficSensor; rulează pe thread propriu; exista un senzor pe fiecare bandă a drumului (2, 4, 6...)
- Obiectul de clasă TrafficTransmitter va trimite un mesaj la fiecare minut cu numărul de maşini ce au trecut



Unele metode vor fi sincronizate:

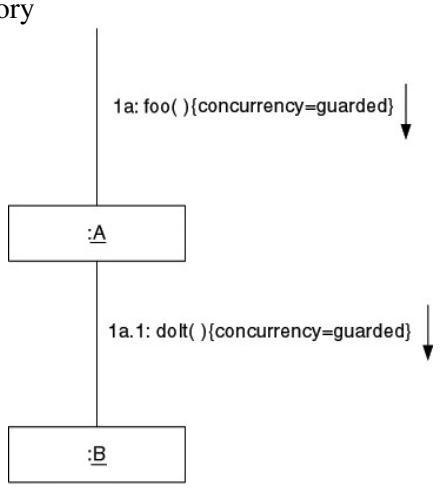


General:

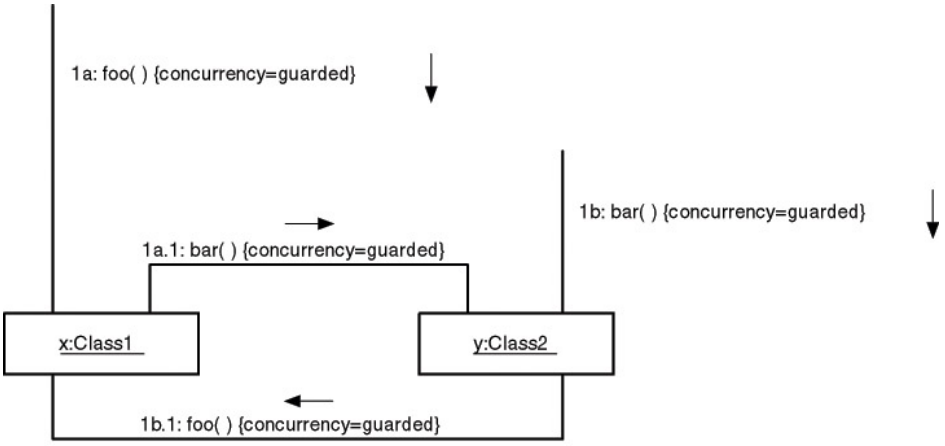


Observaţii:

- Synchronization factory



- Deadlock



Exemplu: java.util.Vector

Exerciţiu: daţi o implementare pentru problema prezentată în context.

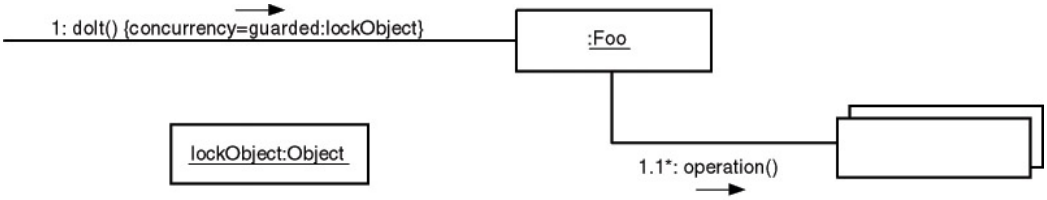
8.2 Lock Object

Dacă dorim accesul exclusiv la mai multe obiecte se poate folosi un asemenea obiect dedicat accesului la acele obiecte, în loc să se achiziţioneze individual fiecare obiect,

Context
Accesul exclusiv la mai multe obiecte la pornirea unui task.

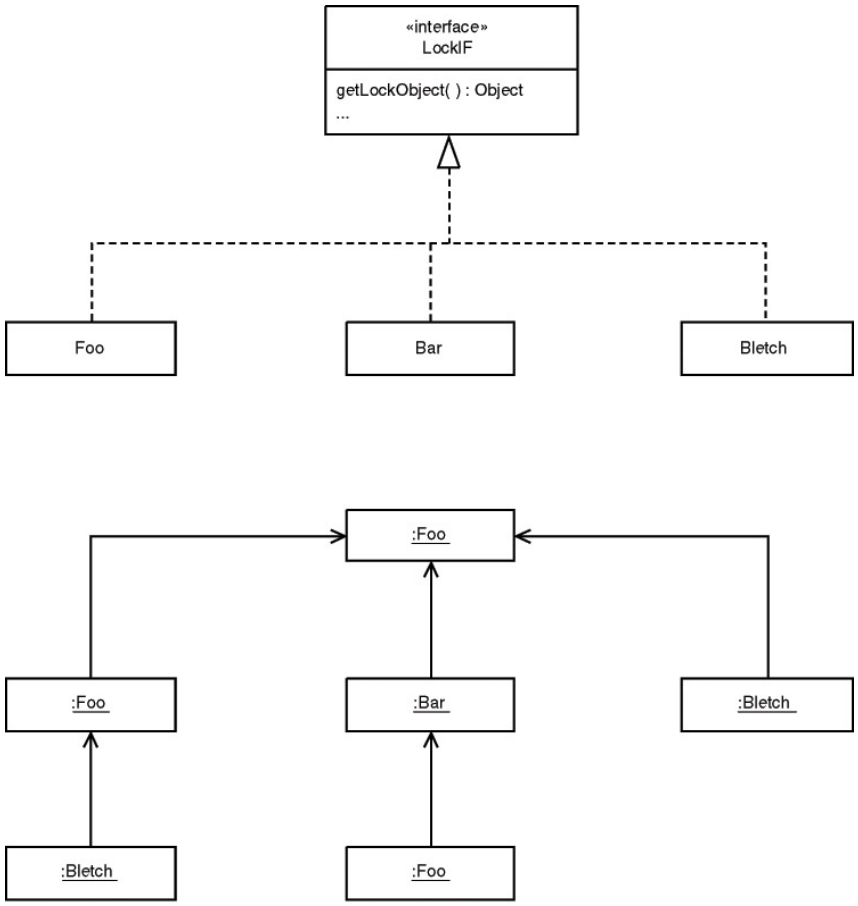
Exemplu: „achiziţionarea" obiectelor dintr-un tablou

General:



Implementare:

Cum se gestionează object lock; ex: obiectele sunt organizate într-un arbore (ca în Swing GUI)

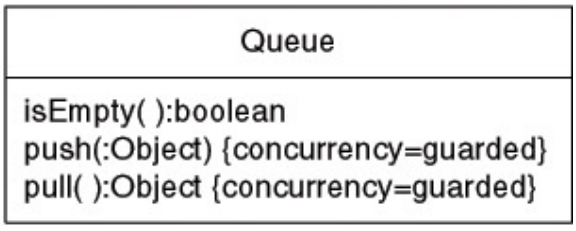


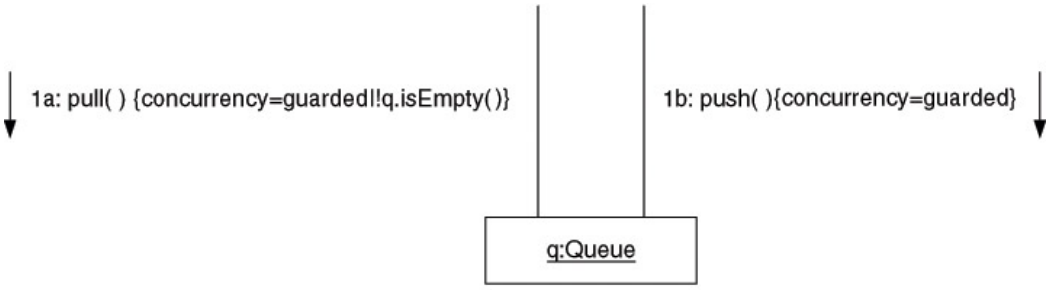
Exercţiu: exemplificaţi propunerea din secţiunea implementare.

8.3 Suspendare cu gardă (Guarded Suspension)

Dacă există o condiţie ce împiedică execuţia unei metode se va suspenda execuţia acesteia din urmă până se îndeplineşte condiţia.

Context: ce se întâmplă dacă se face pull() şi coada (queue) este goală, respectiv se face push() şi coada este „plină”.





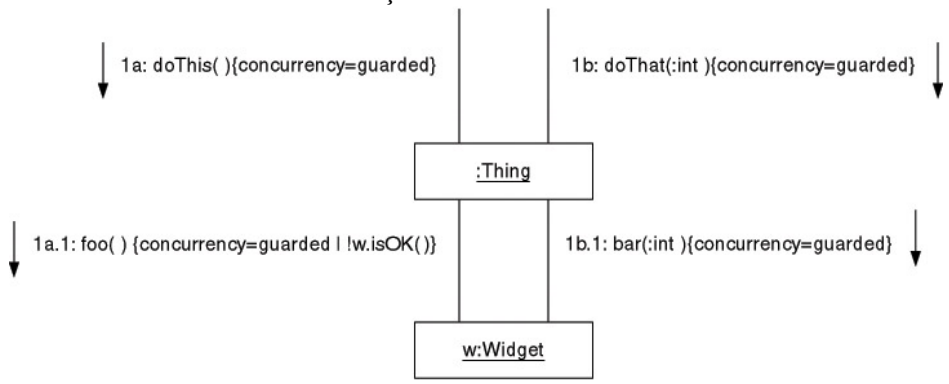
Implementare: wait() şi notify()

```
class Queue {
    synchronized void pull(){
        while (isEmpty()) {
            wait();
        }
        ...
    }

    synchronized void push(x int) {
        ...
        notify();
    }
}
```

Observaţii:

- notifyAll()
- cum se rezolva această situaţie?



Exerciţiu: implementaţi problema de mai sus.

Exemplu: clasa java.net.InetAddress; toate instanţele folosesc un cache comun, accesul la acesta se face prin guarded suspension.

8.4 Balking

Dacă se apelează metoda unui obiect când acesta nu se află în starea potrivită, metoda se va întoarce fără a face nimic (strategie alternativă la guarded suspension).

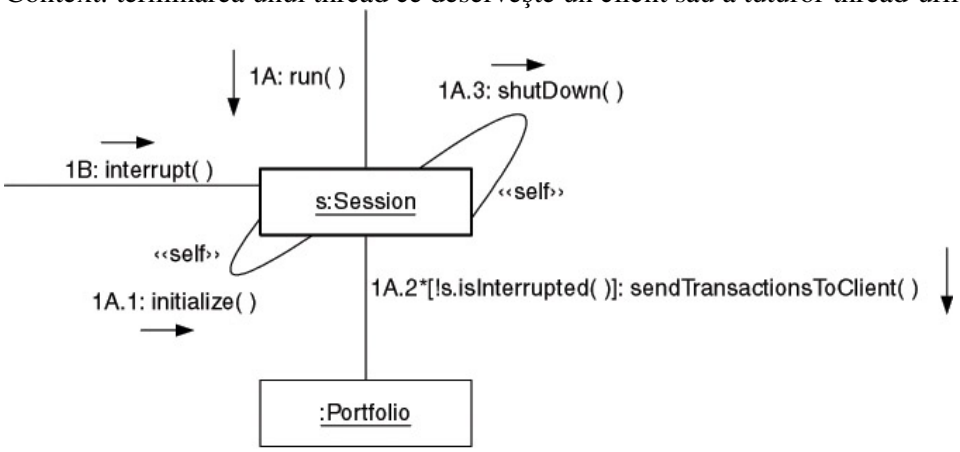
Exemplu: un obiect care citeşte o arhivă zip este solicitat să returneze un anumit fişier. El nu va raspunde solicitării (eventual va returna IllegalStateException).

Exerciţiu: controlul unui reflector prin doua căi: mai multe butoane de start sau un senzor de mişcare. Lumina va dura o perioadă limitată de timp.

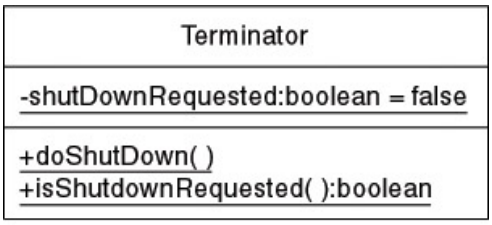
8.5 Terminare în doi paşi (two-phase termination)

Permite terminarea unui thread sau proces în mod organizat prin folosirea unui flag. Thread-ul sau procesul verifică starea acestui flag în puncte strategice ale execuţiei.

Context: terminarea unui thread ce deserveşte un client sau a tuturor thread-urilor



General: clasă pentru terminarea unui thread



Exerciţiu: implementare problemă prezentată în context

8.6 Double Checked Locking

Acest pattern este folosit pentru a reduce overhead-ul asociat achiziţiei unui lock prin testarea unui criteriu de lock într-o manieră nesincronizată, doar dacă criteriul este îndeplinit se va proceda la achiziţia lock-ului.

Exemplu: lazy initialization într-un mediu multi-threading; ilustrare la singleton

```
// Single threaded version
class A {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // other functions and members...
}
// Multithreaded version – overhead inutil după crearea obiectului
class B {
    private Helper helper = null;
    public synchronized Helper getHelper() {
        if (helper == null)
            helper = new Helper();
        return helper;
    }

    // other functions and members...
}


// Double-Checked Locking
class C {
    private Helper helper = null;
    public Helper getHelper() {
        if (helper == null) {
            synchronized(this) {
                if (helper == null) {
                    helper = new Helper();
                }
            }
        }
        return helper;
    }

    // other functions and members...
}
```

Alternative:

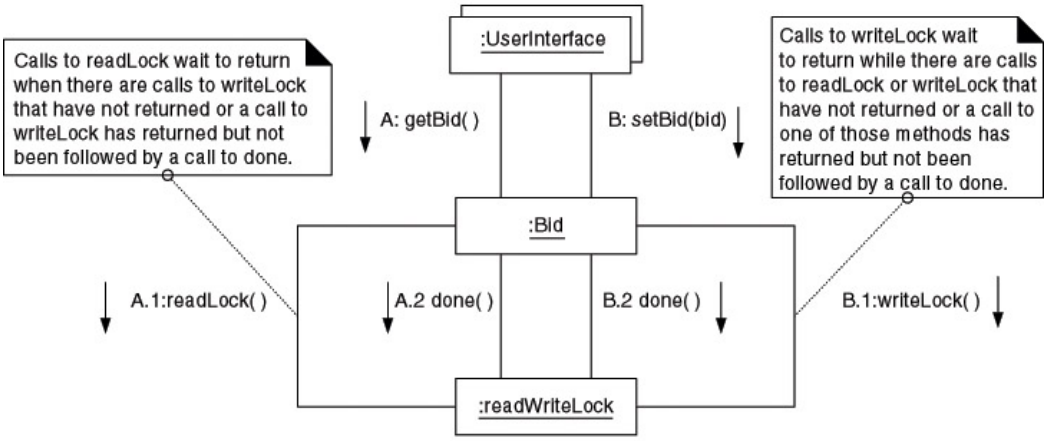
- Eager initialization
- Folosirea unei referinţe statice (se foloseşte lazy initialization!):

```
class MySingleton {
  public static Resource resource = new Resource();
}
```

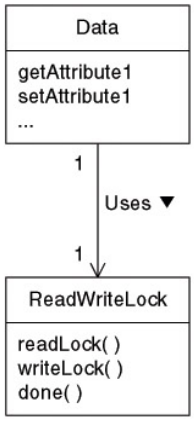
8.7 Read/Write Lock

Permite accesul concurent la citirea unui obiect, scrierea îm obiect este exclusivă.

Context: licitaţie; nu are rost să asigurăm acces exclusiv la citire.



Soluţie:

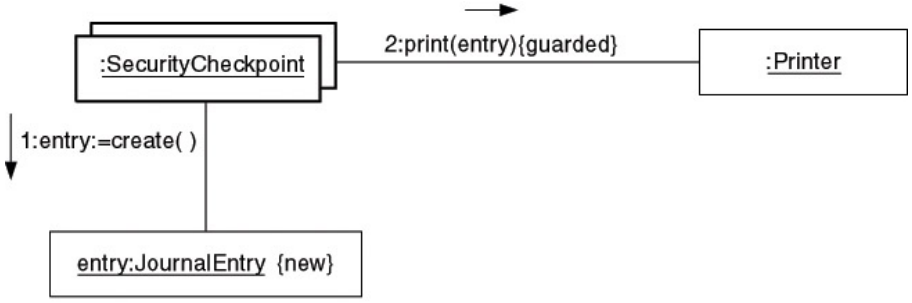


Exerciţiu: implementaţi scenariul prezentat în context

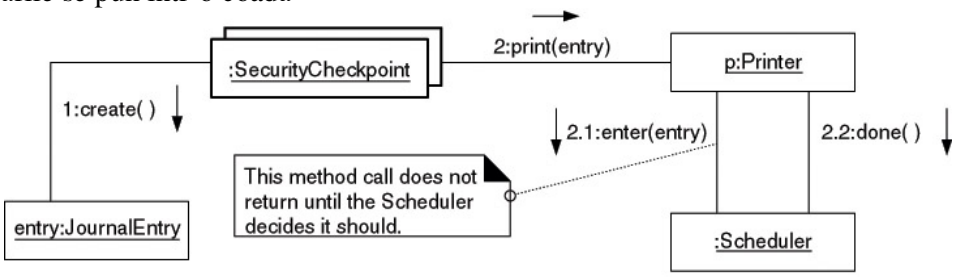
8.8 Scheduler

Controlează ordinea în care firele de execuţie sunt planificate să execute la modul exclusiv cod prin folosirea unui obiect care secvenţiază explicit firele de execuţie. Este un mecanism pentru implementarea unei politici de planificare.

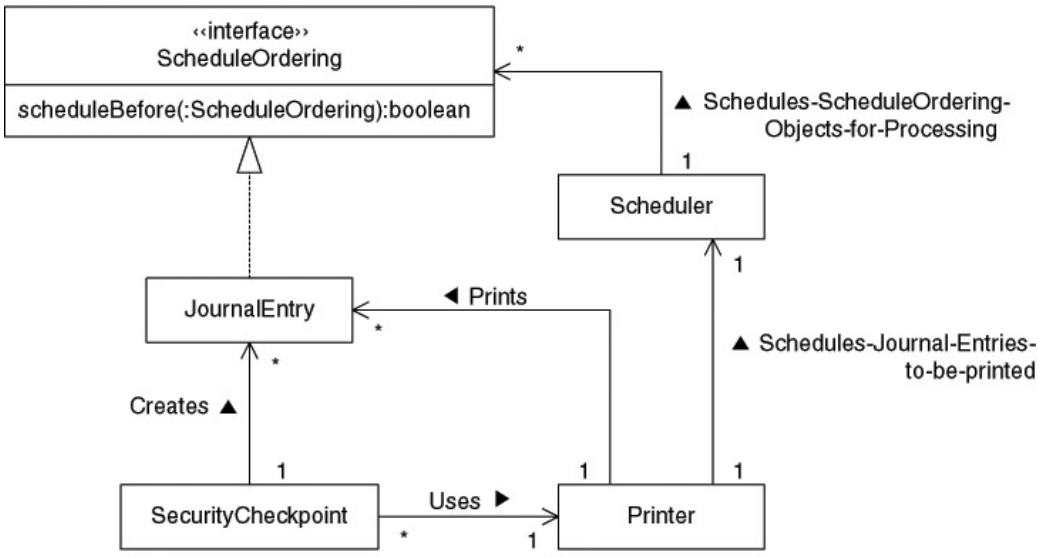
Context
Se înregistrează activitatea de la punctele de verificare (acces permis sau acces refuzat)



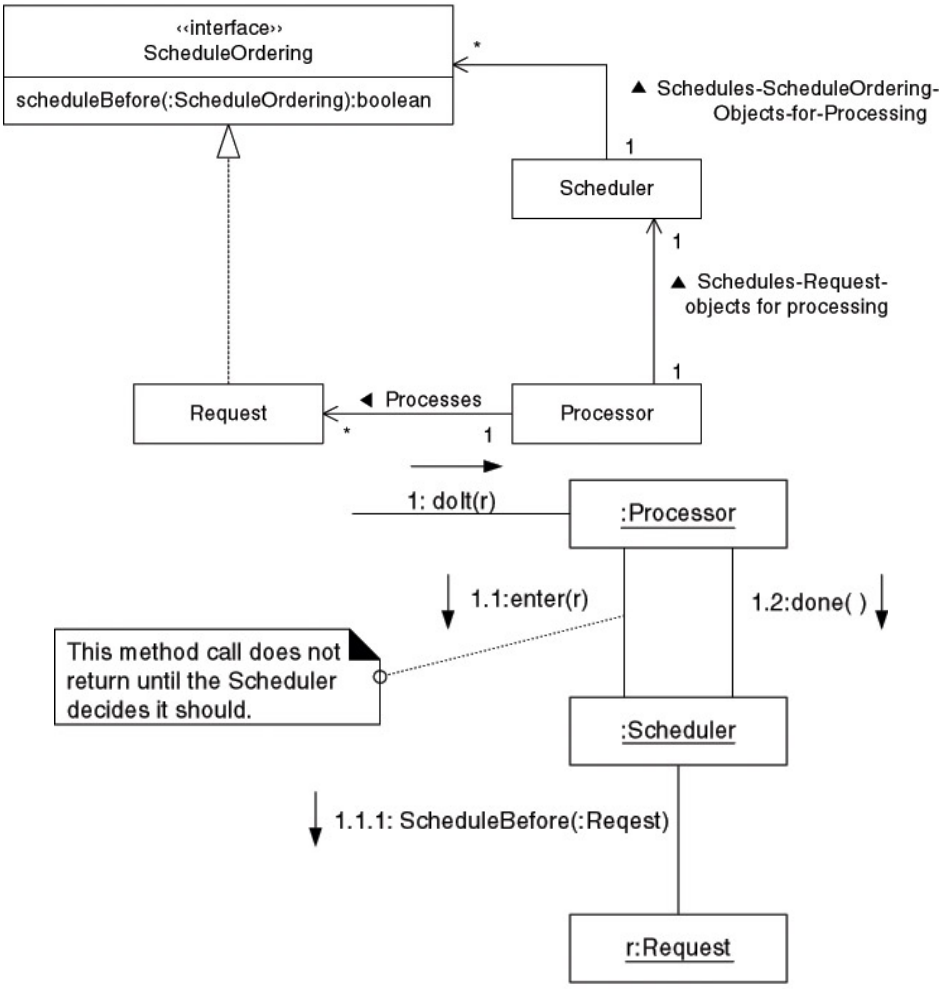
Intrările se pun într-o coadă



Decizia „cine urmează” se poate scoate în afara lui Scheduler:



General:

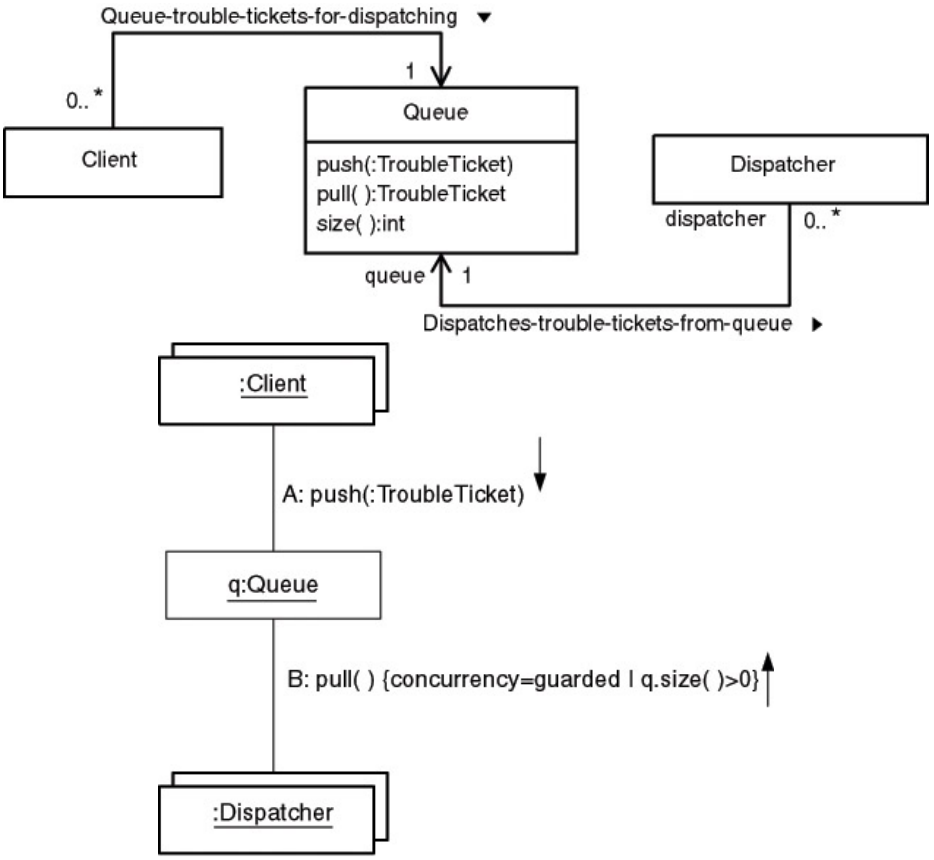


Exerciţiu: oferiţi o implementare pentru problema din context

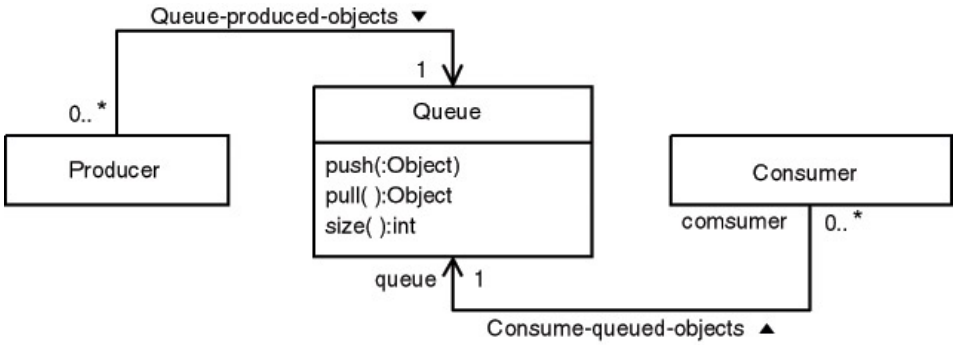
8.9 Producător – consumator

Coordonează în mod asincron producerea şi consumul de informaţii sau obiecte.

Context:



Soluţie:



Observaţie: queue poate avea mărime limitată, producătorul trebuie să aştepte dacă nu are unde să pună obiectele produse

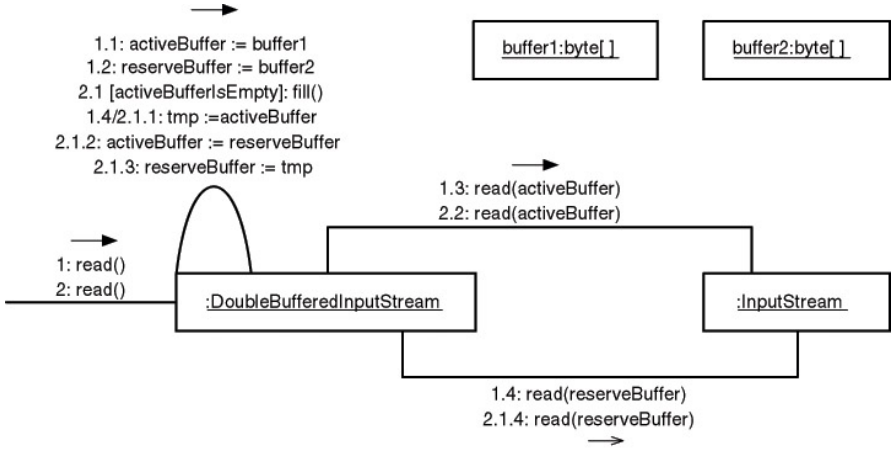
Exerciţiu: implementare problemă prezentată în context.

8.10 Buferizare dublă (Double Buffering)

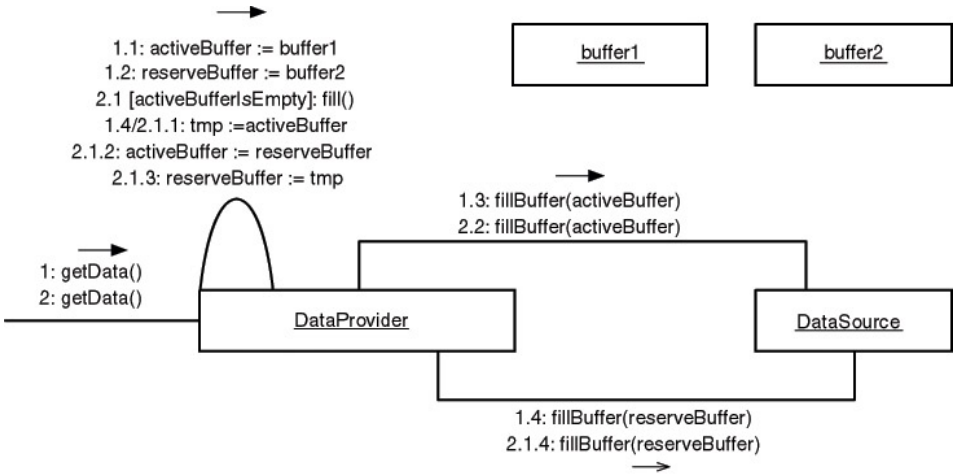
Un obiect consumă date; se evită pierderea timpului la consumarea datelor prin pregătirea lor să fie gata pentru consum. Se adaptează vitezele de procesare a celor două părţi.

Exemplu: afişare imaginii într-un applet

Context: sistem pentru transferul tranzacţiilor de la un POS la o bază de date. Se folosesc două bufere: unul în care se va scrie, celălalt din care se citeşte. Apoi se inversează rolurile.



General:

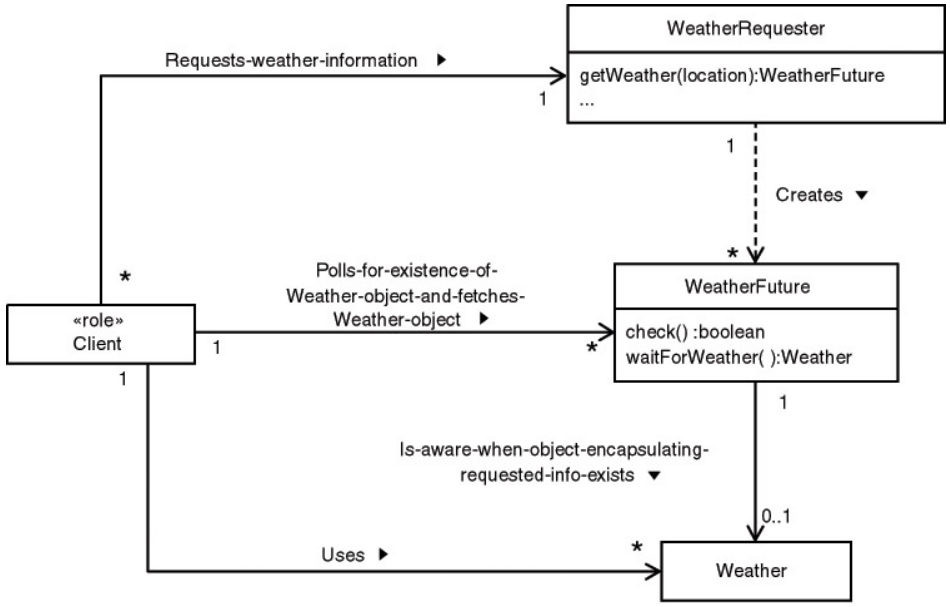


Exerciţiu: implementare

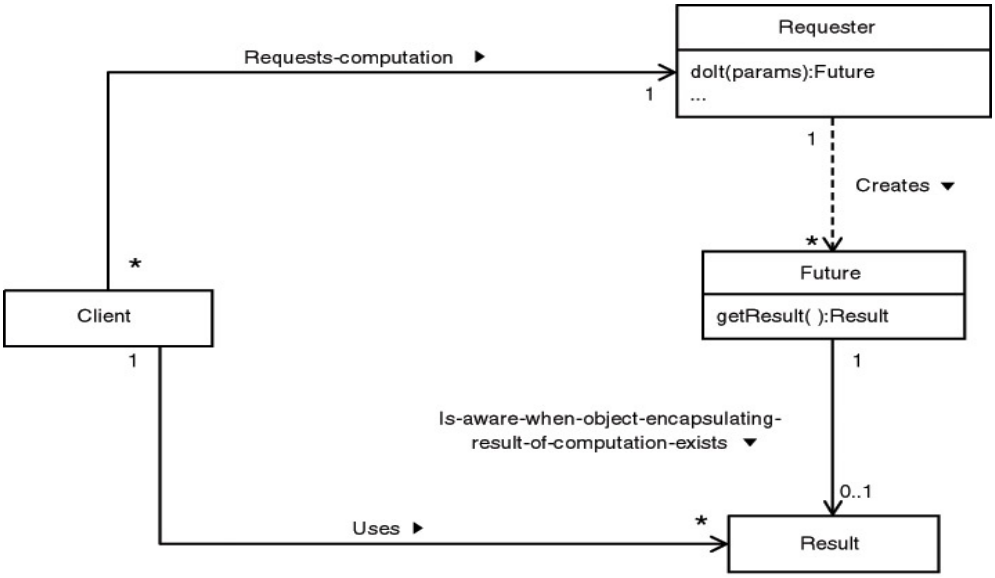
8.11 Viitor (Promisiune)

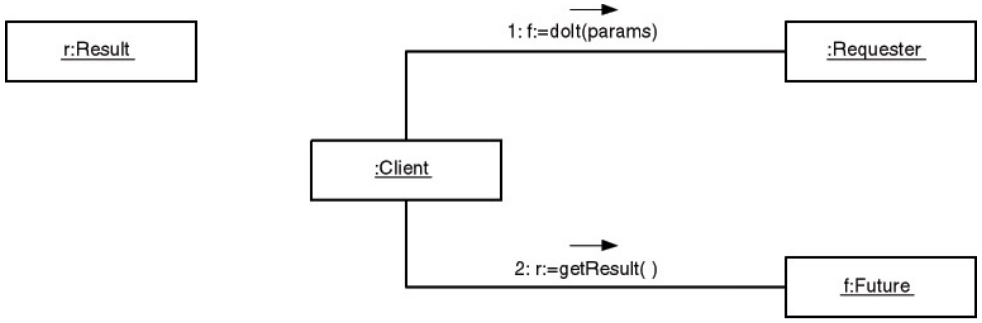
Foloseşte un obiect pentru a încapsula rezultatul unei prelucrări în aşa fel încât ascunde clineţilor faptul că prelucrarea este sincronă sau asincronă. Obiectul are o metodă pentru a oferi rezultatul prelucrării. Aceasta va aştepta rezultatul dacă prelucrarea este asincronă şi este în desfăşurare sau efectuează prelucrarea dacă este sincronă şi nu a fost făcută deja.

Context: clasă pentru a oferi informaţii meteo pentru o anumită locaţie.



General:





8.12 Thread Specific Storage (Thread-Local Storage)

Permit mai multor fire de execuţie folosirea unui variabile logic globale pentru a obţine un obiect local unui thread (fizic aparţin de thread), fără a implica overhead de sincronizare.

Exemplu: errno pentru a semnala ce s-a întâmplat cu o operaţie (intrare sau ieşire); variante: lock wrapper, lock mai complicat (se eliberează doar când thread-ul termină de lucrat cu el).

Java:

```
import java.util.concurrent.atomic.AtomicInteger;

public class UniqueThreadIdGenerator {

  private static final AtomicInteger uniqueId = new AtomicInteger(0);

  private static final ThreadLocal < Integer > uniqueNum =
         new ThreadLocal < Integer > () {
             @Override protected Integer initialValue() {
                 return uniqueId.getAndIncrement();
         }
  };

  public static int getCurrentThreadId() {
         return uniqueNum.get();
  }

} // UniqueThreadIdGenerator
```

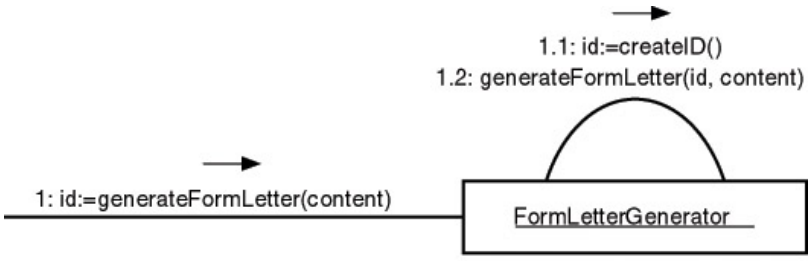
Exerciţiu: oferiţi o implementare pentru o clasă Settings

8.13 Procesare asincronă

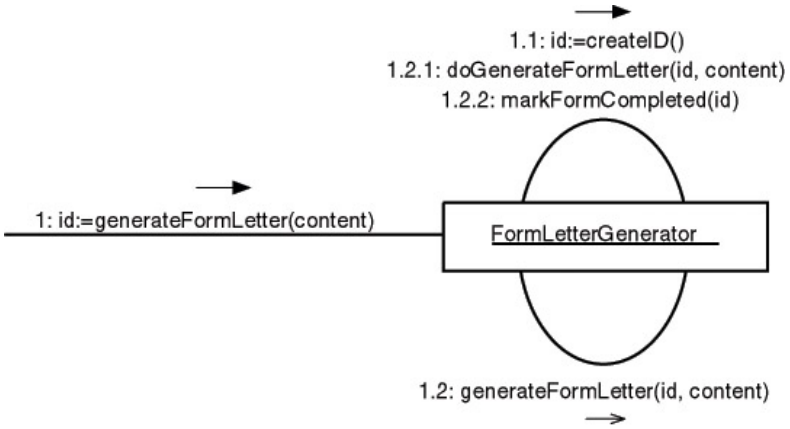
Un obiect primeşte cereri să facă ceva. El nu va onora cererile sincron, le va ţine minte şi le va procesa asincron.

Context:

- sincron

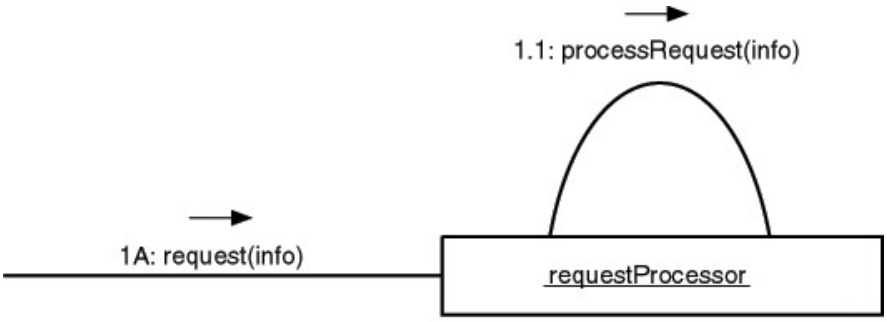


- asincron:



Exemplu: Swing, javax.swing.SwingUtilities, invokeLater()

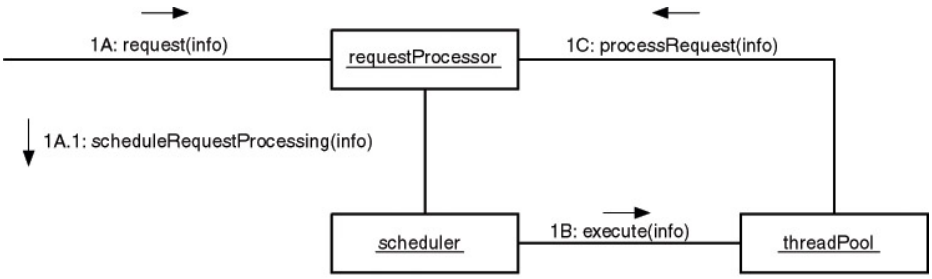
General:



Aspecte:

- Cum se gestionează cererile, când se onorează
- Cum se îndeplinesc cererile:
  - pornirea unui thread nou pt fiecare cerere
  - thread pool

Număr limitat de thread-uri:



8.14 Thread Pool

Este o colecţie de fire de execuţie ce sunt menţinute active pentru a executa task-uri.

Avantaje:

- Reutilizarea thread-urilor economiseşte timpul pentru crearea şi distrugerea lor.
- Se controlează numărul de task-uri ce se procesează în paralel

Exerciţiu: oferiţi o implementare de thread pool, de exemplu pentru execuţia asincronă a unui serviciu (tipărire automată a facturilor)

8.15 Active Object

Decuplează apelul metodelor de execuţia metodelor ce rezidă în propriul thread. Obiectivul este de a introduce concurenţa prin folosirea apelurilor asincrone şi printr-un scheduler pentru a trata cererile.

Exemplu: onorarea comenzilor într-un restaurant: chelneri, comandă, bucătari.

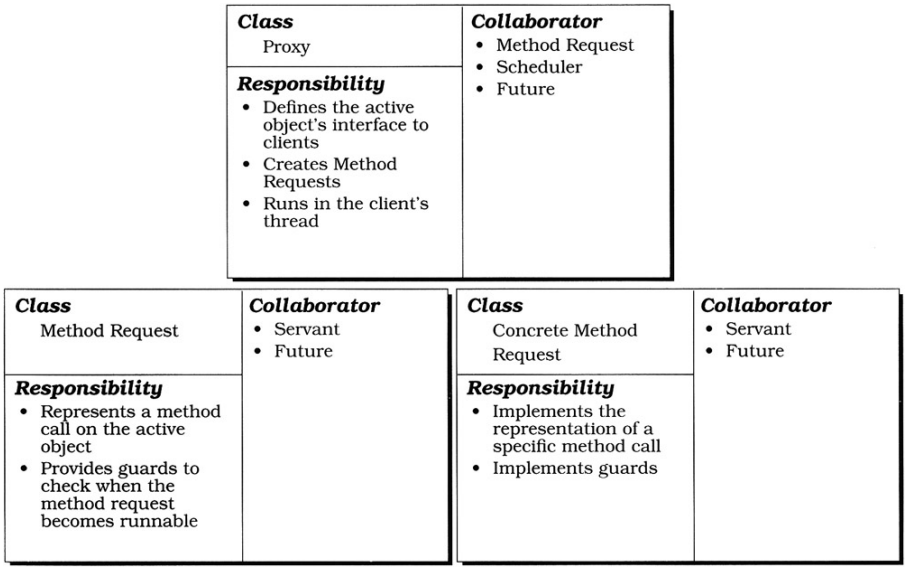
Context: clienţi ce accesează obiecte ce rulează în thread-urile lor proprii.

Soluţie: invocarea metodei se face în thread-ul clientului, execuţia metodei se face într-un thread separat. Cele două părţi sunt reprezentate de:

- un proxy ce reprezintă interfaţa obiectului activ
- un servant ce reprezintă implementarea

Run-time: proxy-ul transformă apelurile făcute de client în cereri ce sunt puse într-o listă destinate unui scheduler. Acesta va scoate cererile apoi şi le va executa pe servant. Clienţii pot obţine rezultatul execuţiei printr-un future returnat de proxy.

Structură

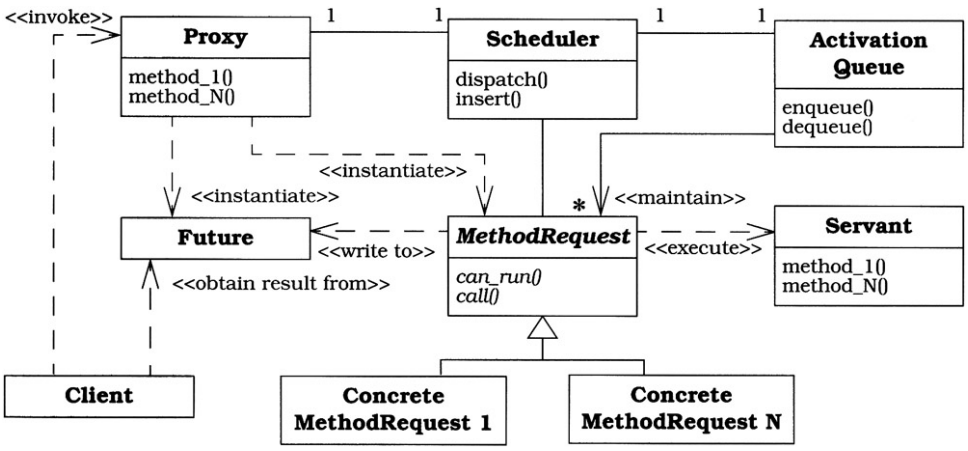


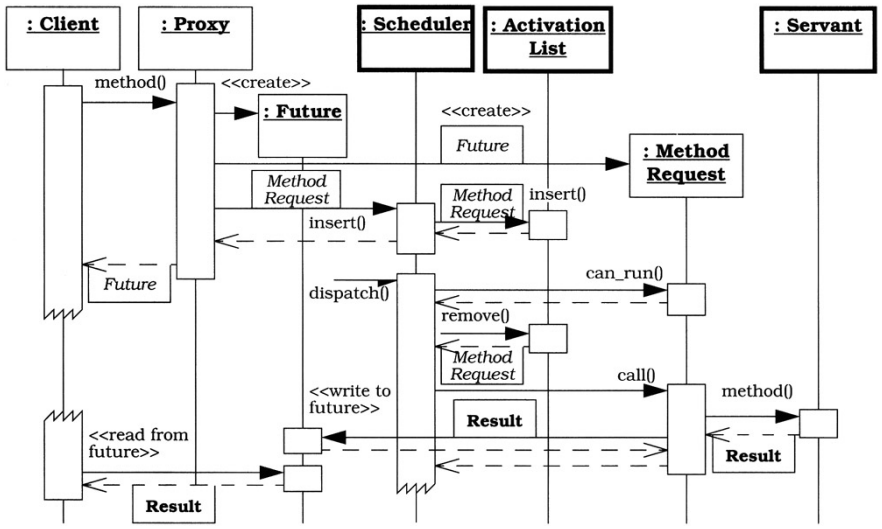
Class	Collaborator
Activation List	
Responsibility	
• Maintains Method Requests pending for execution • Method Requests are inserted by the proxy and removed by the scheduler	

Class	Collaborator
Scheduler	• Activation List • Method Request
Responsibility	
• Inserts Method Requests into the Activation List • Runs in the active object's thread	

Class	Collaborator
Servant	
Responsibility	
• Implements the active object • Runs in the active object thread	

Class	Collaborator
Future	
Responsibility	
• Stores the result of a method call on an active object • Provides a rendezvous point for a client	





8.16 Barrier

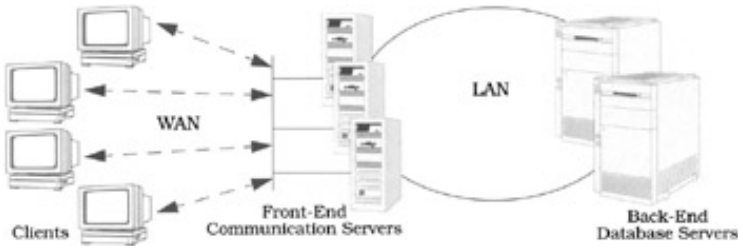
Este o metodă de sincronizare în care un grup de fire de execuție se blochează pană când toate ajung la această entitate.

Exemplu: fire de execuție care se ocupă de inițializarea părților unui sistem; sistemul va fi pregătit doar dupa ce toate firele de execuție au încheiat procedura de inițializare.

8.17 Leader/Followers

Acest pattern arhitectural propune un model concurent eficient în care mai multe fire de execuție deservesc pe rând un set de surse de evenimente. Aceste thread-uri detectează, demultiplexează, livrează şi procesează cererile venite de la sursele de evenimente.

Exemplu: cereri de tranzacţii- agenţii de turism, rezolvare de probleme într-un call center, etc.



Variante pentru procesarea evenimentelor:

- un singur thread (reactor)
- thread pool bazat pe half-sync/half-reactive – overhead

Soluţie: