Design Patterns

442A
Horsa Alexandru
Todicescu Alexandru
Toma Crina
Uta Cosmin Alexandru

443A
Botoc Catalin

Cuprins

Introducere....3
Model de proiectare....3
Descrierea modelelor de proiectare....3
Cum rezolva modelele de proiectare problemele unui proiect....4
Utilizarea mecanismelor refolosite....6
Modele de interfata....7
Facade....7
Adapter....10
Bridge....12
Composite....14
Modele de responsabilitate....17
Singleton....17
Observator....19
Mediator....20
Proxy....21
Lant de responsabilitati....23
Flyweight....24
Modele creationale....27
Abstract Factory....27
Builder....29
Factory Method....31
Prototype....32
Modele operationale....33
Template Method....33
State....34
Strategy....37
Command....38
Interpreter....39
Modele de extensibilitate....40
Decorator....40
Iterator....43
Visitor....46
Bibliografie....51

Introducere

Toma Crina

442A

Proiectarea de software orientat pe obiect este grea, dar proiectarea de software reutilizabil orientat pe obiect este chiar mai grea. Trebuiesc gasite obiecte pertinente , grupate pe clase in functie de granularitate, clasele de interfata si ierarhiile mostenite sa fie definite , si stabilite legaturile cheie dintre ele.Este de preferat sa se evite reproiectarea sau cel putin sa se minimalizeze.Inainte ca proiectarea sa ia sfarsit , designerii experimentati de obicei incearca sa o reutilizeze, modificand parametrii de fiecare data.

Proiectarea de modele simplifica reutilizarea de proiecte si arhitecturi.Poate imbunatatii documentatia si mentenanta sistemelor existente furnizand specificatiile claselor si obiectelor ce interactioneaza in mod explicit.

Model de proiectare

Fiecare model descrie o eroare ce are loc in mod repetat si solutia la problema , in asa fel incat poti folosii aceasta solutie de un milion de ori fara sa o faci de doua ori la fel.Un model are patru elemente esentiale:

1. Numele modelului : este o variabila folosita pentru a descrie o eroare de proiectare.
2. Problema : descrie cand trebuie aplicat modelul, eroare in sine si contextul ei.
3. Solutia : descrie elementele ce alcatuiesc proiectul , relatiile dintre ele , responsabilitatile si colaborarea.
4. Consecintele: reprezinta rezultatele puneri in aplicare a modelului.

Descrierea modelelor de proiectare

Descrierea se face folosind un format consistent.Fiecare model este impartit in sectiuni in functie de urmatoarele sabloane.

a. Numele modelului si clasificarea sa : un nume corect este vital pentru ca el va deveni parte din vocabularul proiectului.
b. Intentia (Intent) : O scurta declaratie ce raspunde la intrebari despre relatiile, intentia si erorile unui model de proiectare.
c. De asemenea cunoscut ca (Also Known As) : Alte denumiri cunoscute pentru modelul in cauza, daca exista.

d. Motivatia (Motivation) : reprezinta un scenariu ce ilustreaza o eroare a modelului si solutii pentru rezolvarea sa .

e. Aplicabiliate : situatiile in care poate fi aplicat modelul de proiectare.

f. Structura : reprezentarea grafica a claselor in model folosind o notatie baza pe Object Modeling Technique (OMT).

g. Participanti : clasele si/sau obiectele participante la modelul de proiectare si responsabilitatile lor.

h. Colaborari : cum colaboreaza participantii pentru a indeplini responsabilitatile.

i. Consecinte : cum suporta modelul obiectivele.

j. Implementarea : cum se realizeaza implementarea modelului .

k. Codul Sursa : fragmente de cod ce ne arata cum se poate implementa modelul in C++ sau Smalltalk

l. Utilizari cunoscute : exemple de modele folosite in sisteme reale.

m. Modele de legatura : diferentele si legaturile intre modele de proiectare.

Cum rezolva modelele de proiectare problemele unui proiect

Modelele de proiectare rezolva multe din problemele proiectarii orientate pe obiect pe care designerii le intalnesc.Avem cateva probleme si cum modelul de proiectare le rezolva:

1. Gasirea obiectului corect : programele cu proiectare pe obiect sunt formate din obiecte.Procedurile folosite se numesc “metode” sau operatiuni.Un obiect efectueaza o operatiune cand primeste un mesaj de la utilizator.

   Partea cea mai grea la proiectarea orientata pe obiect, reprezinta descompunerea unui sistem pe obiecte.Este foarte dificil acest lucru de realizate din cauza factorilor existenti in numar mare: incapsularea , granularitatea , dependency , flexibilitatea , performanta , evolutia , refolosirea , etc.

   Modelele de proiectare ne ajuta la identificarea abstractiilor mai putin observabile si a obiectelor ce le poate captura.

2. Determinarea granularitatii obiectului

   Obiectele pot varia in in functie de marime si de numar.Pot reprezenta tot incepand cu partea de hardware pana la aplicatii.Cum decidem ce reprezinta obiect si ce nu?

   Modelul Facade (208) descrie cum sa reprezinte un susbsistem complet ca obiect , Flyweight (218) descrie cum sa suporte un numar mare de obiect la cea mai fina granularitate.

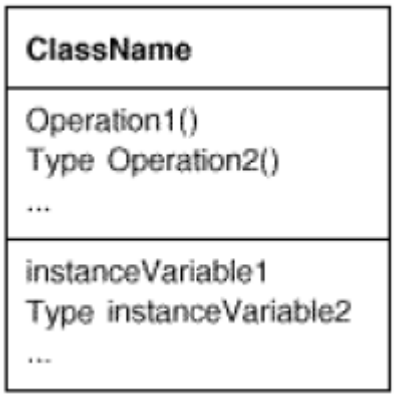
Alte modele descriu modalitati specifice de descompunere a obiectelor in obiecte mai mici: Abstract Factory (99) , Builder (110) , Visitor (366), Command (263).

3. Precizarea interfetelor obiectelor

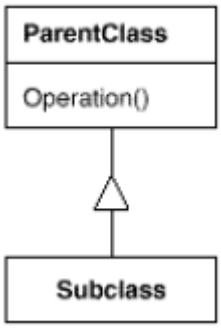
Modelul de proiectare ne ajuta sa definim interfetele identificand elementele cheie si tipurile de date transmise prin interfata.De asemenea este posibil ca un model sa iti spuna ce sa nu pui in interfata (The Memento -316-). Modelele de proiectare specifica de asemenea si relatia dintre interfete.De obicei au nevoie ca anumite clase sa aiba interfate similare .

4. Precizarea implementarii obiectului

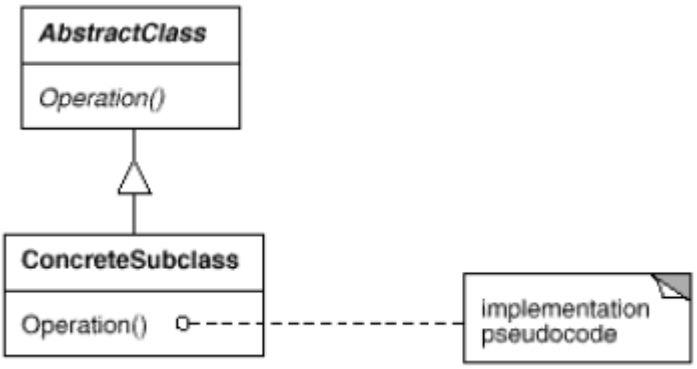
Implementarea unui obiect este definita de catre clasa sa.Clasa specifica data interna a obiectului si defineste operatiile pe care obiectul le poate realiza.



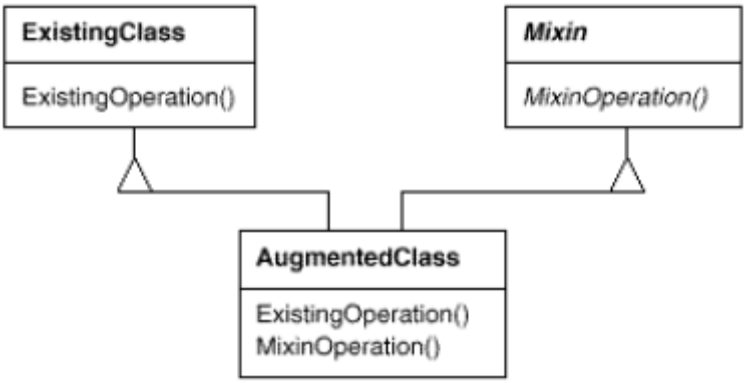
Noi clase pot fi definite in functie de alte clase existente folosind mostenirea de la acestea.Cand o sub-clasa mosteneste de la clasa parinte , ea include toate definitiile si operatiile pe care clasa parinte le executa.



O clasa abstracta are rolul principal de a defini o interfata obisnuita pentru sub-clasa sa.Operatiunile pe care o clasa abstracta le declara dar nu le implementeaza se numesc operatiuni abstracte.



O clasa de mixiune este o clasa realizata pentru a procura o interfata optionala sau functionalitate pentru alte clase.Clasele de mixiune au nevoie de o mostenire multipla.



Utilizarea mecanismelor refolosite

Majoritatea persoanelor inteleg conceptul de obiecte , interfate , clase si mostenire. Greutatea consta in folosirea lor pentru a crea software flexibil si refolosibil, de aceea avem nevoie de modele de proiectare.

Proiectare

Reproiectarea afecteaza in mare parte software-ul sistemului, iar schimbarile neasteptate sunt invariabil de scumpe.

Modelele de proiectare ne ajuta sa evitam acest lucru asigurand ca sistemul sa se poata modifica in modalitati specifice.

Cauzele reproiectarii si modelele folosite:

1. Crearea unui obiect specificand o clasa explicita. (Abstract Factory , Factory Method , Prototype)

2. Dependenta de operatiuni specifice . (Chain of Responsability , Command)
3. Dependenta de hardware si platforma software. (Abstract Factory , Bridge)
4. Dependenta de reprezentarea obiectelor sau implementarea lor. (Abstract Factory , Bridge , Memento , Proxy)
5. Dependenta de algoritmi. (Builder , Iterator , Strategy , Template Method , Visitor)
6. Cuplare stransa . (Abstract Factory , Bridge , Chain of Responsability , Command , Facade , Mediator , Observer)
7. Extinderea functionalitatii folosind sub-clase. (Bridge , Chain of Responsability, Composite , Decorator , Observer , Strategy)
8. Neputinta de a modifica o clasa . (Adapter , Decorator , Visitor).

Folosirea unui model de proiectare:

1. Citeste modelul odata pentru o analiza generala.
2. Mergi inapoi la capitolele Structuri , Participanti si Colaborari si citeste.
3. Vizualizeaza Codul Proba pentru a vedea un exemplu de model de proiectare.
4. Alege nume pentru participantii la proiectarea modelului ce conteaza in contextul aplicatiei.
5. Defineste clasele
6. Defineste numele aplicatiilor specifice pentru operatiunile din model
7. Implementeaza operatiunile ce vor indeplinii responsabilitatile si colaborarile din model.

Modele de interfata

Toma Crina
442A

Facade

Intentia : ofera o interfata unita pentru un set de interfate dintr-un subsistem. **Motivatia** : structurarea unui sistem in mai multe subsisteme ajuta la reducerea complexitatii.Un tel

comun este de a minimiza comunicatiile si dependentele dintre subsisteme.

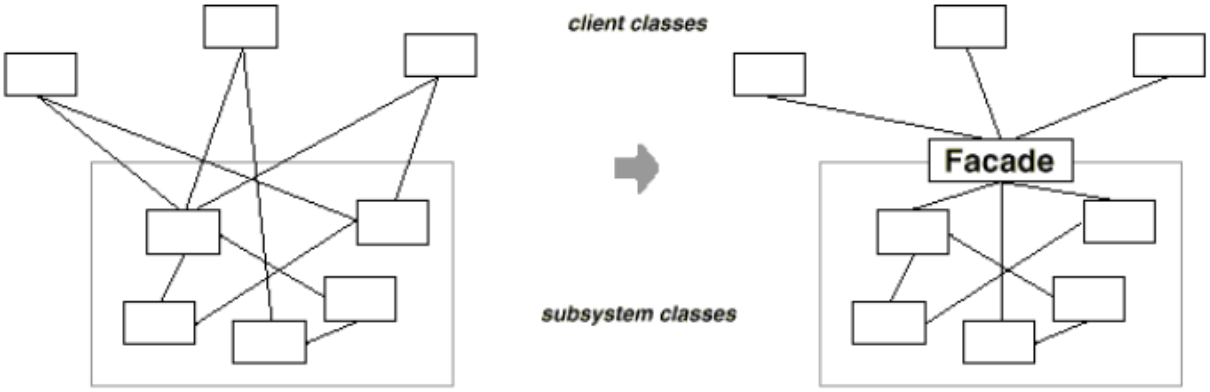
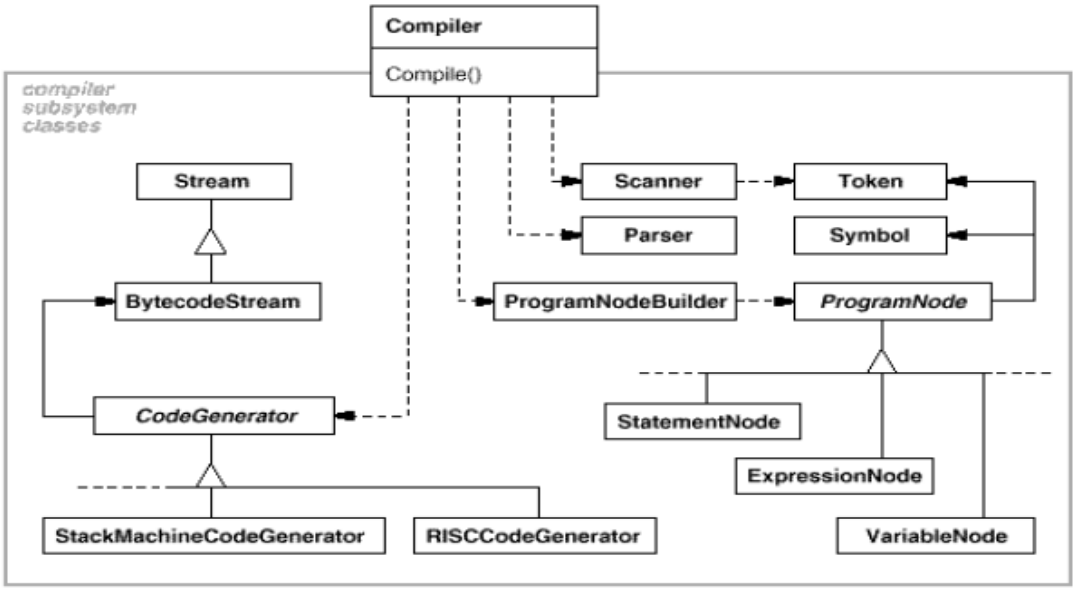


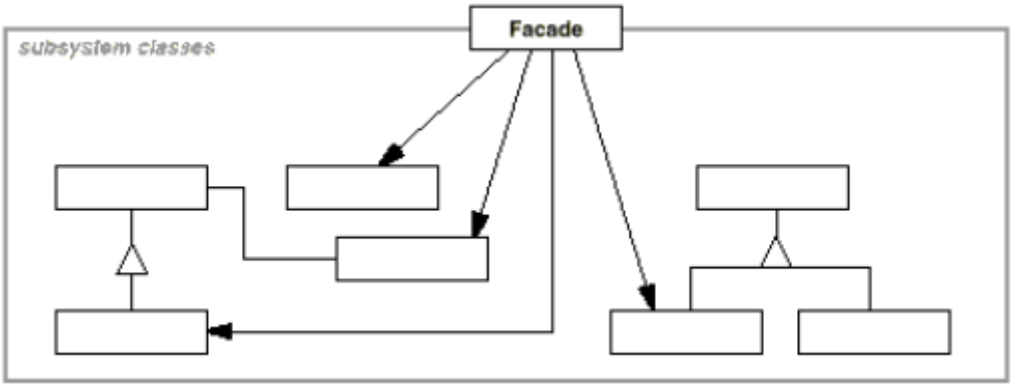
Diagrama Compiler



Aplicabilitate - folosim modelul Facade atunci cand:

- Vrem sa oferim o interfata simpla unui subsistem complex
- Exista multe dependente intre clienti si implementarea claselor unei abstractii
- Vrem un subsistem etajat.

Structura



Participanti:

- Facade (compiler)
- Subsystem classes (Scanner , Parser , ProgramNode)

Colaborari :

- Clientii comunica cu subsistemele trimitand cereri catre Facade , care la randu lui le trimite subsistemului de obiecte corect.
- Clientii care folosesc Facade , nu au nevoie sa acceseze subsistemele obiectelor in mod direct.

Consecinte:

- Protejeaza clientii de componentele subsistemului
- Promoveaza legaturile slabe dintre susbsistem si client
- Nu impiedica aplicatiile sa foloseasca susbsistemul de clase , daca au nevoie.

Implementare:

- Reduce legaturile client-susbsistem
- Sistem public versus sistem de clase privat

Utilizari cunoscute:

- ObjectWorks\Smalltalk compiler
- ProgrammingEnvironment
- InspectObject\InpsectClass

- FileSystemInterface\Domain

Modele asemanatoare:

- Abstract Factory
- Mediator
- Singletons

Adapter

Intentia : convertirea interfatei unei clase in interfata pe care clientul o asteapta.

Motivatie : Uneori o clasa set de instrumente , proiectata pentru a fi reutilizata nu este reutilizabila pentru ca interfata nu se potriveste cu interfata specifica domeniului de care aplicatia are nevoie.

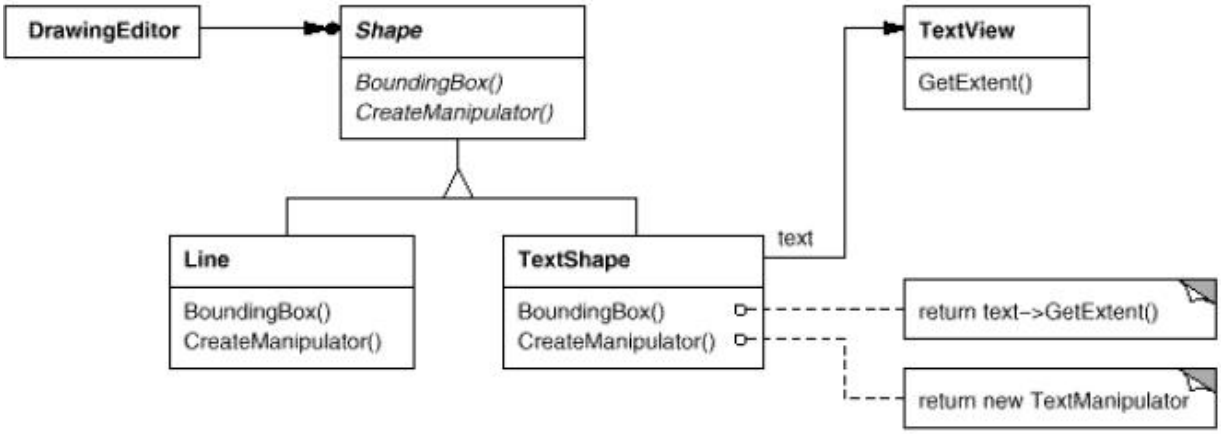


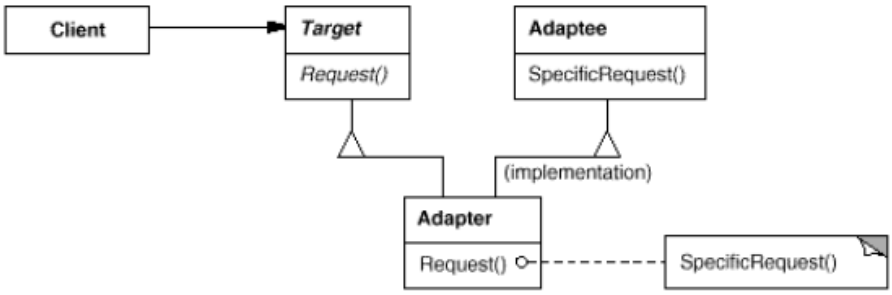
Diagrama ce reprezinta cazul obiectului adaptor.

Aplicabilitate - folosim modelul Adapter atunci cand:

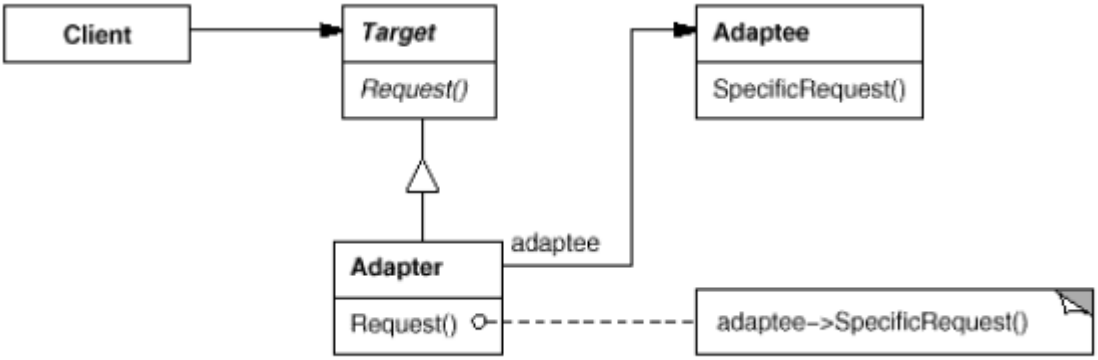
a. Vrem sa folosim o clasa existenta si interfata pe care o folosim nu se potriveste cu cea de care avem nevoie.
b. Vrem sa cream o clasa reutilizabila ce va coopera cu alte clase neprevazute
c. Vrem sa folosim mai multe sub-clase existente dar este imposibil de adaptat interfata lor. (doar in cazul obiectului adaptor)

Structura

a. Adaptor de clasa:



b. Obiect adaptor



Participanti

a. Target (forma)
b. Client
c. Adaptee
d. Adapter (forma textului)

Consecinte

Adaptor de clasa:

a. Adapteaza Adaptee la Target angajandu-se la o clasa concreta Adapter.
b. Permite Adapter sa ocoleasca cateva din comportarile lui Adaptee
c. Introduce un singur obiect

Obiect Adaptor:

a. Permite folosirea unui singur Adapter in acelasi timp cu mai multi Adaptee
b. Incalcarea comportamentului lui Adapter se face mai greu

Implementare

a. Implementarea adaptorilor de clasa in C++.
b. Adaptori conectabili

Interfata “ingusta” duce la trei implementari :

a. Folosind operatiuni abstracte
b. Folosind obiecte delegate
c. Parametrizand adaptorii

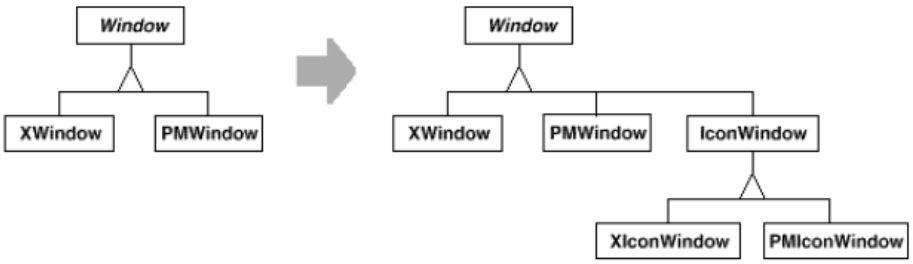
Bridge

Intentia : decupleaza o abstractie de pe implementare astfel incat cele doua sa varieze independent.

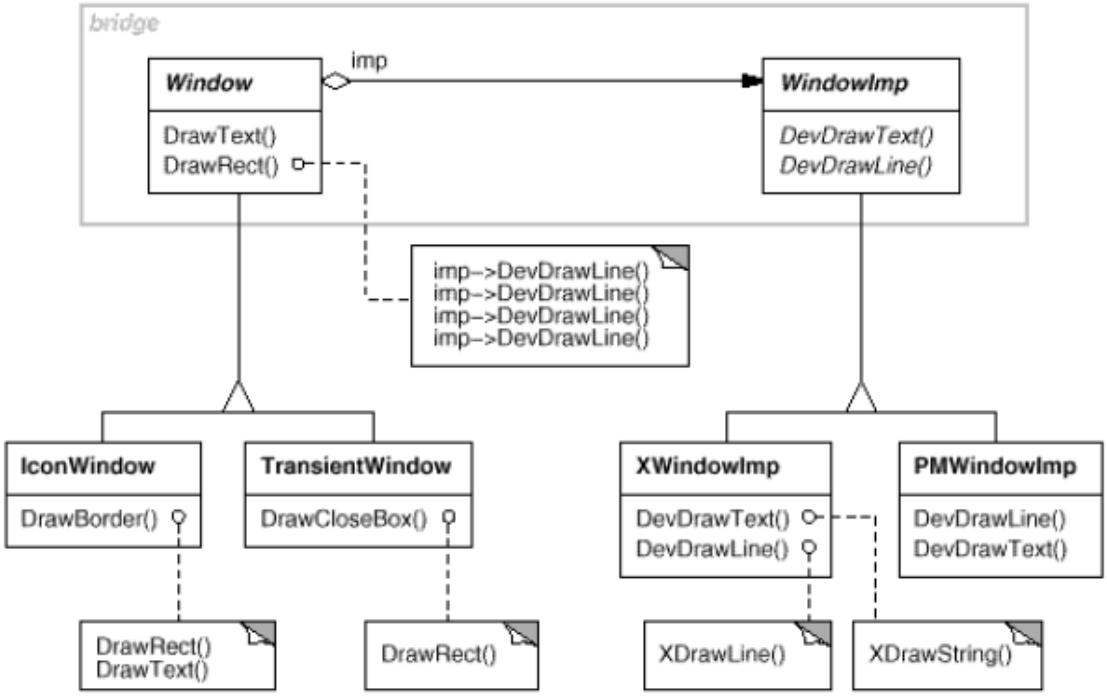
Motivatie : Cand o abstractie poate avea una din mai multe posibile implementari , modul normal de acomodare este de a folosi mostenirea.

Folosirea mostenirii vine totusi cu doua dezavantaje.

a. Nu este convenabil sa extindem abstractia Window pentru a acoperi mai multe tipuri de ferestre sau platforme.



b. Face codul clientului dependent de platforma.

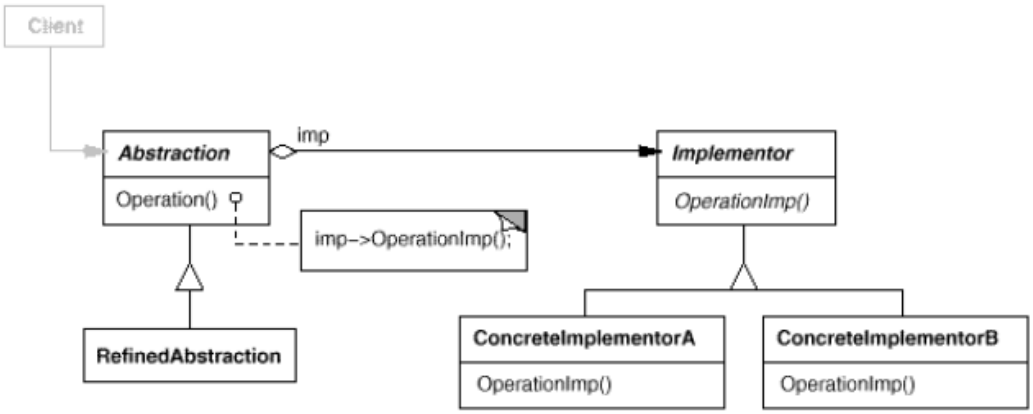


Aplicabilitate

Folosim modelul Bridge atunci cand:

a. Cand vrem sa evitam o legatura permanent intre abstractie si implementare.
b. Atat abstractiile cat si implementarile lor trebuie sa poata fi extinse prin sub-clase.
c. Schimbari in implementarea unei abstractii nu trebuie sa afecteze clientul.
d. (C++) vrem sa ascundem implementarea unei abstractii complet , de client.
e. Exista o proliferare de clase dupa cum se vede in digrama motivatiei de mai sus.
f. Vrem sa impartim o implementare intre mai multe obiecte , si acest lucru trebuie ascuns de client

Structura



Participanti

a. Abstractia
b. Abstractia rafinata
c. Implementor
d. Implementor cert

Consecinte

- Decuplarea interfetei si a implementarii
- Extensibilitate imbunatatita
- Ascunde detaliile implementarii de client

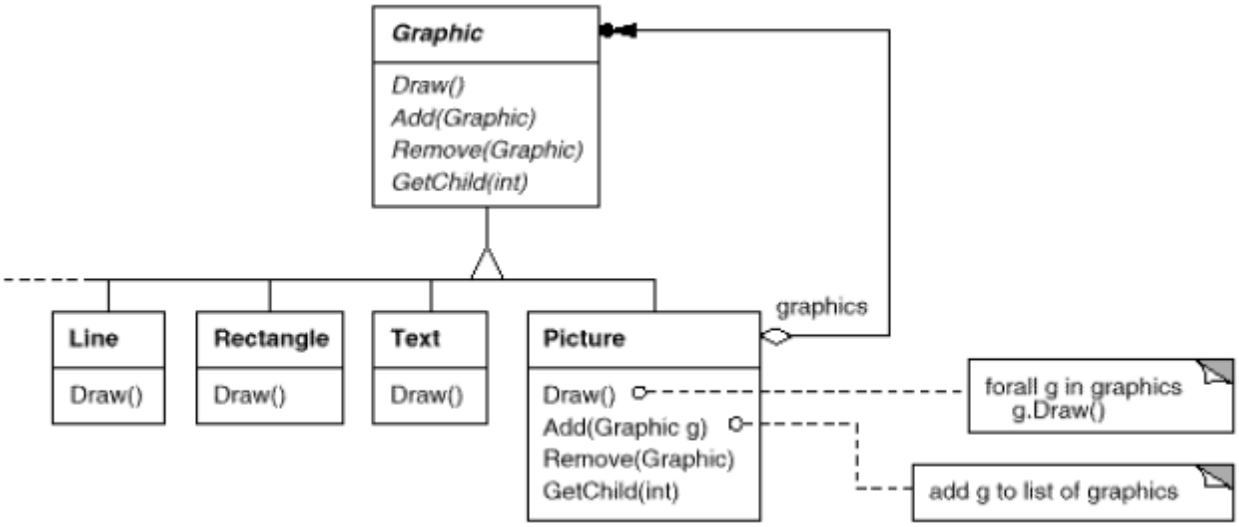
Implementare

- Un singur implementor
- Crearea implementorului de obiect, corect
- Impartirea implementorilor
- Folosirea de mostenire multipla

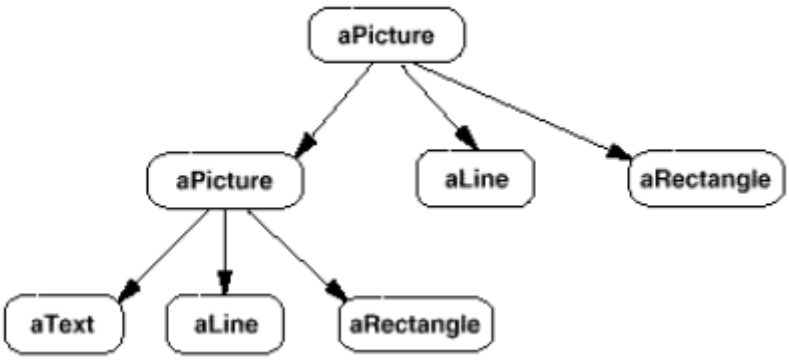
Composite

Intentia : compune obiecte structuri principale pentru a reprezenta parti intregi din ierarhie.

Motivatia: Aplicatii grafice cum sunt editori de desene si capturi schematice de sistem permit utilizatorilor sa construiasca diagrame complexe din componente simple.Modelul Composite descrie cum trebuie folosita o compozitie in asa fel incat clienti sa nu poata distinge acest lucru.



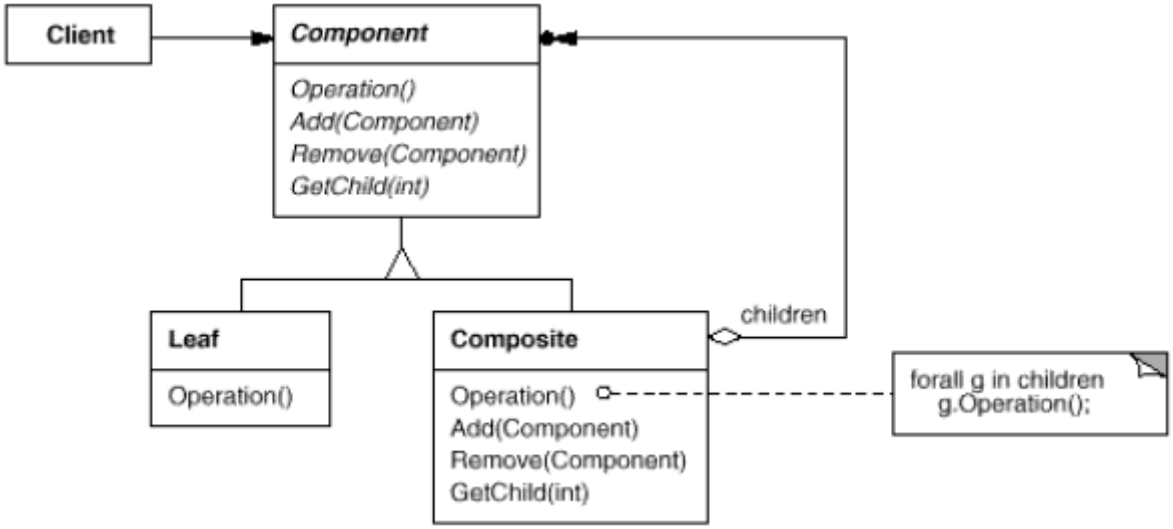
Urmatoarea diagrama ne arata o structura compozit tipica a unor obiecte Grafice compuse recursiv.



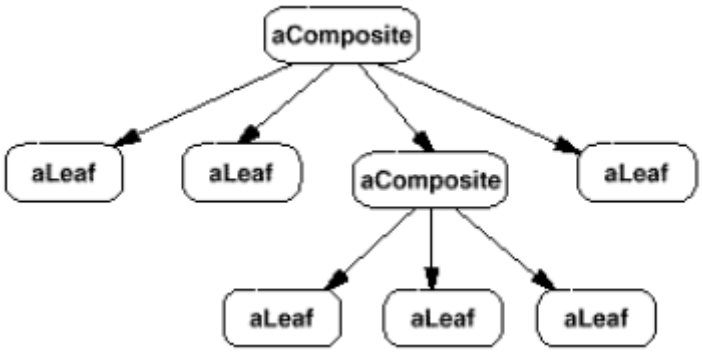
Aplicabilitate – folosit modelul Composite atunci cand:

- Vrem sa reprezentam parti intregi din ierarhia unui obiect
- Vrem clientii sa poata ignora diferentele dintre compozitii de obiecte si obiecte individuale.

Structura



O structura tipica Composite de obiecte arata in felu urmator:



Participanti

- Component (Graphic)
- Leaf (Rectangle , Line , Text ,)
- Composite (Picture)
- Client

Colaborari : clientii folosesc interfata de clasa Component pentru a interactiona cu obiecte in structura compozit.

Consecinte

- Defineste clasele ierarhice consistente din obiecte primitive si compozite
- Clientul este simplificat
- Adaugarea de noi componente este mai simpla
- Poate face designul prea general

Modele de responsabilitate

Todicescu Alexandru
442A

Singleton

Scop

Scopul este de a asigura faptul ca o clasa are o singura instanta si de a furniza un singur punct de acces global catre aceasta instanta.

Motivatie

Este important pentru unele clase sa aiba doar o singura instanta. De exemplu, desi pot exista mai multe imprimante intr-un sistem, ar trebui sa existe un singur spooler de imprimanta. De asemenea, ar trebui sa existe doar un singur sistem de fisiere si un singur manager de ferestre. Un filtru digital va avea un singur convertor A/D.

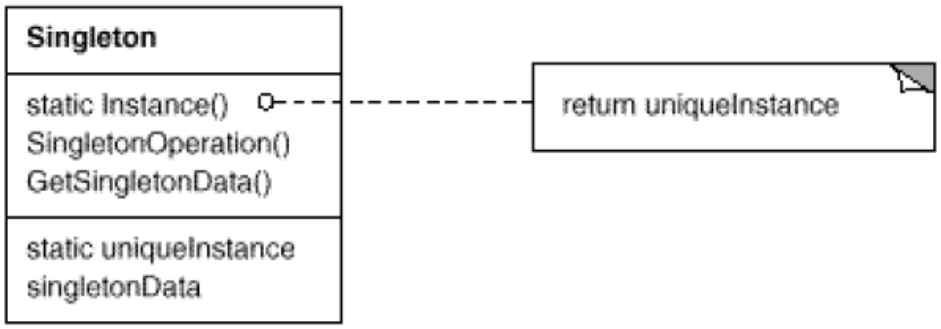
Cum ne asiguram de faptul ca o clasa are o singura instanta si aceasta instanta este usor de accesat? O variabila globala face ca obiectul sa fie accesibil global, dar nu ne opreste din a instantia de mai multe ori clasa.

O solutie mai buna este de a face responsabila chiar clasa de urmarirea singurei sale instante. Clasa poate asigura faptul ca nicio alta instanta nu se mai poate crea (prin interceptarea cererilor de creare a noi obiecte) si poate furniza o metoda de a accesa instanta. Acesta este modelul Singleton.

Aplicabilitate

Se utilizeaza modelul Singleton cand trebuie sa existe o singura instanta a unei clase si aceasta trebuie sa fie accesibila clientilor dintr-o locatie bine cunoscuta.

Structura



Consecinte

Avantaje

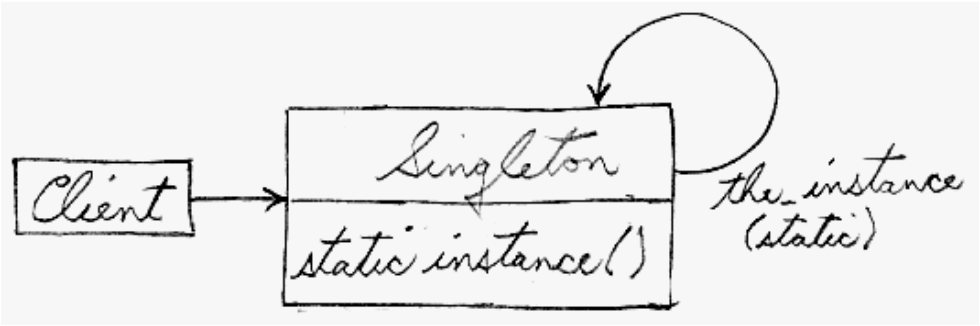
- Singleton este singura clasa care poate crea o instanta a sa. Singura modalitate de creare este prin apelarea metodei statice oferita de ea.
- Referinta nu trebuie transmisa tuturor obiectelor interesate. Fiecare obiect ce are nevoie de Singleton va apela metoda statica.

Dezavantaje

- sablonul Singleton poate provoca probleme de threading, ce depind de implementare. Trebuie controlata cu atentie initializarea singletonului într-o aplicatie multithreading. Fara un control adecvat, aplicatia va beneficia de "razboaie ale firelor de executie".

Cum trebuie facut?

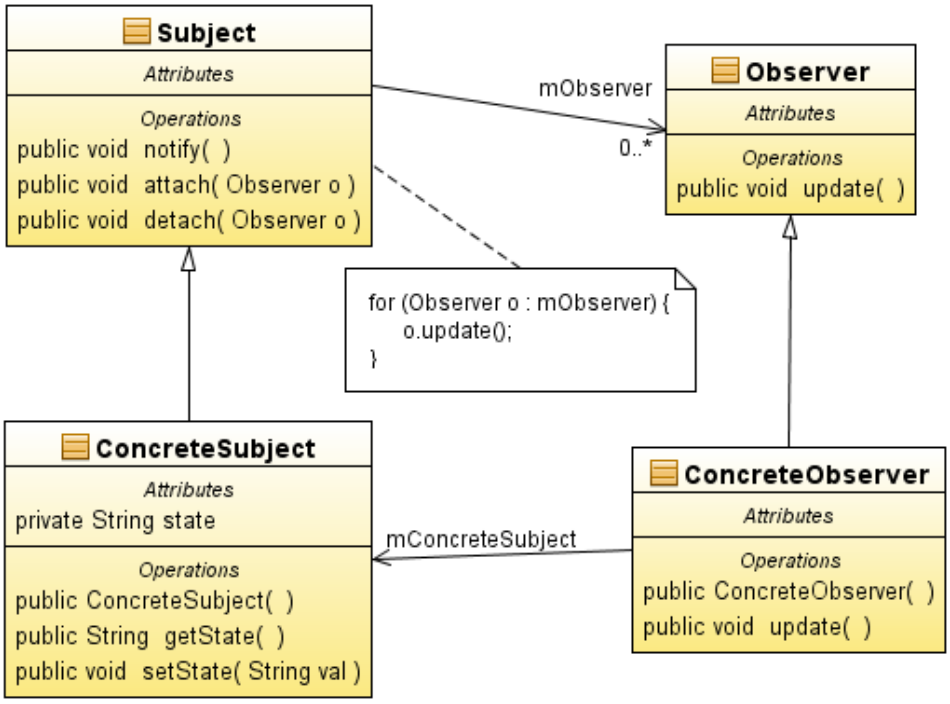
1. Se defineste un atribut private de tip static in clasa care implementeaza acest model
2. Se defineste o functie publica "accesoare" de tip static pentru acces
3. Se face initializarea(crearea la prima utilizare) in functia accesoare
4. Se definesc toti constructorii protected sau private
5. Clientii pot folosi doar functia accesoare pentru a manipula obiectul Singleton.



Observator

Sablonul observator este un sablon de proiectare in care un obiect gestioneaza o lista cu proprii dependenti, pe care ii anunta automat de eventualele modificari de stare, de obicei prin apelarea anumitor metode.

De multe ori acest sablon este folosit pentru implementarea sistemelor distribuite.



Clasa Subject este o clasa abstracta (sau o interfata) ce asigura atasarea si scoaterea observatorilor. Clasa contine pe langa o lista privata cu observatori si urmatoarele metode:

- attach( ) – adauga un nou observator in lista;
- detach( ) – elimina un observator din lista;
- notifyObserver( ) – anunta fiecare observator asupra unei schimbari de stare prin apelarea metodelor update( ) ale acestora.

Clasa ConcreteSubject este elementul de interes al observatorilor. Ea trimite o notificare tuturor observatorilor prin apelarea metodei notifyObserver( ) din clasa ei parinte. Clasa ConcreteSubject contine pe langa interfata Subject metoda GetState ce returneaza starea subiectului de observat.

Clasa Observer defineste o interfata pentru anuntarea tuturor observatorilor asupra modificarilor survenite in subiect. Interfata consta intr-o metoda ce va fi suprascrisa de fiecare observator concret.

Clasa ConcreteObserver gestioneaza o referinta catre clasa ConcreteSubject si contine operatia update( ). Cand acesta operatie este apelata de catre subiect, ConcreteObserver apeleaza operatia GetState a subiectului pentru a-si actualiza informati privind starea subiectului. Operatia update( ) poate primi eventual parametri cu informatii generale ale evenimentului aparut, informatii utile observatorului.

Sablonul observator este utilizat atunci cand modificarea starii unui obiect afecteaza alte obiecte si nu se stie la momentul scrierii codului exact ce obiecte vor trebui anuntate.

Exemplu de folosire a sablonului observer: implementrarea modurilor de lucru, cand intr-o fereastra de dialog utilizarea unui obiect poate presupune disponibilizarea sau indisponibilizarea altor obiecte. In terminologia acestui sablon toate obiectele de pe un formular sunt observatori, iar subiectul concret de observat este chiar formularul. Fiecare obiect in parte nu trebuie sa modifice direct celelalte obiecte de pe formular pentru ca ar trebui sa cunoasca multimea acestora.

Aceeasi functionalitate a modurilor de lucru ar putrea fi implementata si cu ajutorul sablonului mediator.

Mediator

Sabloanele comportamentale (behavioral patterns) caracterizeaza modul in care obiectele si clasele interactioneaza si isi distribuie responsabilitatile. Ele se pot aplica obiectelor sau claselor.

Sabloanele comportamentale ale claselor utilizeaza mostenirea pentru descrierea unor algoritmi si fluxuri de control:

- Interpreter
- Template Method

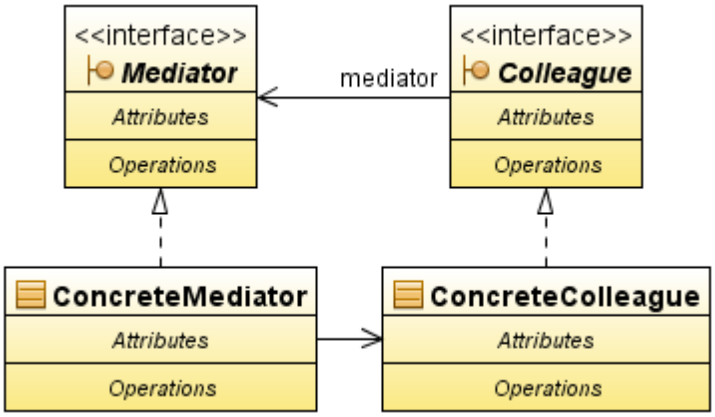
Sabloanele comportamentale ale obiectelor descriu modul in care un grup de obiecte coopereaza pentru a indeplini o sarcina ce nu ar putea fi efectuata de un singur obiect:

- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

De obicei o aplicatie este alcatuita dintr-un numar important de clase. Cu cat sunt mai multe clase intr-o aplicatie, problema comunicarii intre obiecte devine mai complexa, ceea ce face programul mai greu de citit si intretinut. Modificarea programelor, in astfel de situatii, devine mai dificila din moment ce orice modificare poate afecta codul din mai multe clase.

Cu ajutorul sablonului mediator comunicatia dintre obiecte este incapsulata in obiectul mediator. Obiectele nu mai comunica direct intre ele, ci comunica in schimb prin intermediul mediatorului. Acest lucru reduce dependenta intre obiecte, micsorand cuplarea.

In contextul acestei diagrame termenul de coleg este utilizat pentru a desemna un obiect ce doreste sa comunice cu alte obiecte din acelasi grup, un grup avand un singur moderator.



Mediator defineste interfata de comunicare intre obiecte

- ConcreteMediator - implementeaza interfata Mediator si coordoneaza comunicarea intre objecte. Stie care sunt toate obiectele ce doresc sa comunice si scopurile acestor comunicari.
- ConcreteColleague – comunica cu alte obiecte "colegi" cu ajutorul mediatorului.

Sablonul mediator se poate combina cu sablonul singleton, daca o clasa concreta mediator are sens sa fie instantiata o singura data. Intr-un astfel de caz, daca aplicatia abstractizeaza mai multe grupuri de colegi, pentru fiecare grup se poate crea cate o clasa concreta mediator care sa fie instantiata o singura data.

Sablonul mediator nu trebuie confundat cu sablonul proxy. Diferenta principala este faptul ca mediatorii si colegii nu sunt creati pe baza aceleiasi interfete (sau clase abstracte). Un proxy este „imaginea" unui alt obiect, prin urmare structura unui proxy depinde destul de mult de structura obiectului destinatie. Nu degeaba sablonul proxy este inclus in grupul sabloanelor structurale. Structura unui mediator nu este imaginea in oglinda a unui obiect coleg, ceea ce ii da posibilitatea sa transmita mesajele spre mai multe tipuri de clase concrete de colegi, altfel spus sa medieze mesajele intre colegi cu structuri diferite.

Exista asemanari intre sablonul mediator si alte sabloane la lista de avantaje:

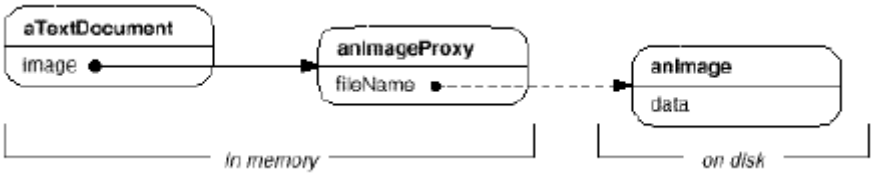
- un coleg (binevoitor) poate sa transmita „anonime" (avantaj intalnit si la proxy);
- un coleg (raspandac) poate sa transmita acelasi mesaj mai multor obiecte (avantaj intalnit si la sablonul compozit);
- un mediator (securist) poate sa foloseasca liste de control al accesului (avantaj intalnit si la proxy);
- un mediator (mai lenes) poate astepta mai multe mesaje de la colegi pentru a lansa o anumita operatiune (avantaj specific sablonului mediator).

Proxy

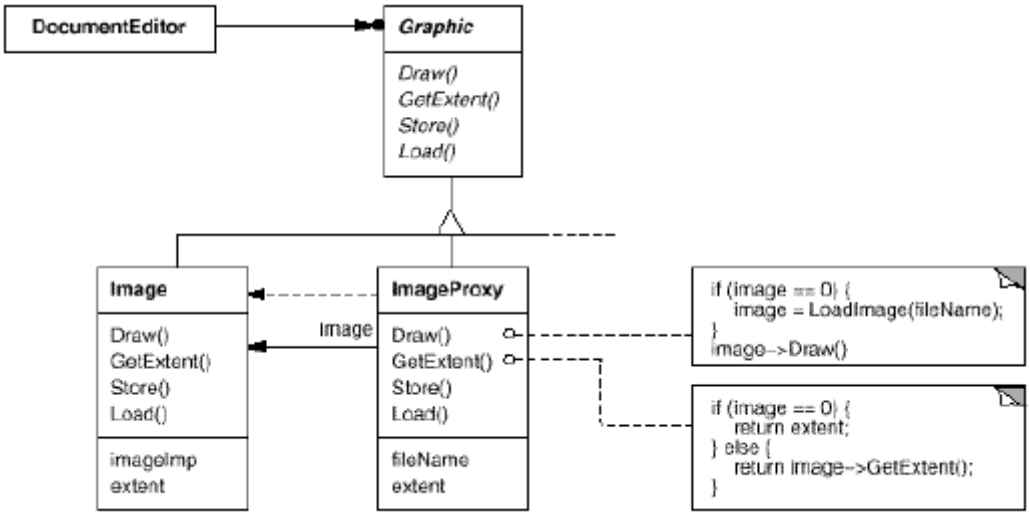
Intentia : ofera un inlocuitor pentru alt obiect de a avea acces la el.

Cunoscut ca : Surrogate

Motivatie : un motiv pentru a avea controlul asupra unui obiect este sa amane costul total al creatiei pana cand avem neaparat nevoie de el.Aceste constrangeri sugera sa cream fiecare care obiect scump doar la comanda.Solutia este sa folosim alt obiect , in cazul de fata o imagine proxy.



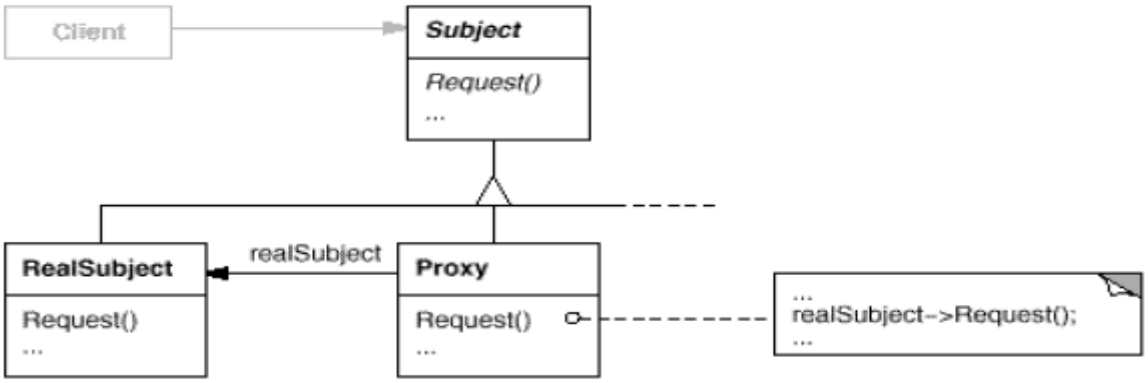
Urmatoarea diagrama ilustreaza un exemplu mai in detaliu :



Aplicabilitate – proxy este folosit atunci cand este nevoie de o referinta mai diversificata sau sofisticata a unui obiect:

- Un proxy la distanta ofera o reprezentare locala a unui obiect aflat in alt spatiu de adresa.
- Un proxy virtual creaza obiecte scumpe la comanda.
- Un proxy pentru protectie controleaza accesul la obiectul original.

Structura



Participanti:

- Proxy (ImageProxy)
- Subject (Graphic)
- RealSubject (Image)

Colaborari : Proxy trimite mai departe cererile catre subiectu real cand este cazul , depinzand de tipul de proxy.

Consecinte :

- Modelul proxy a introdus un nivel indirect cand acceseaza un obiect
- Alta optimizare pe care proxy o are pe langa faptul ca se poate ascunde de client este cea de copy-on-write , si are legatura cu crearea la comanda.

Implementarea :

Modelul proxy poate exploata urmatoarele caracteristici de limbaj.

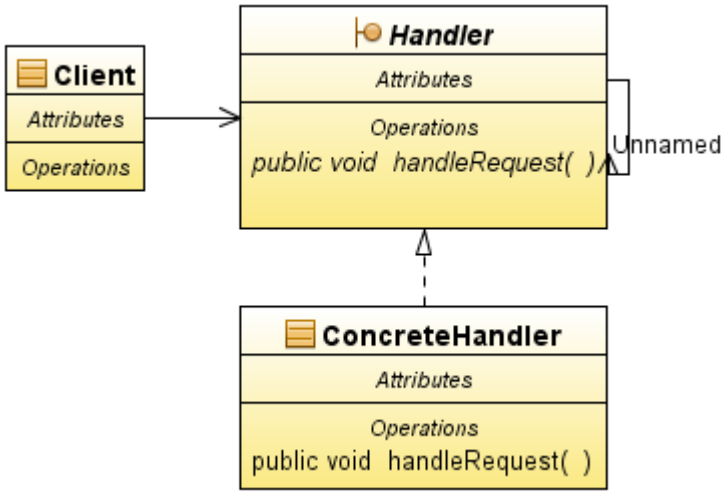
- Supraincarcarea accesului membrului operator in C++
- Folosirea “doesNotUnderstand” in Smalltalk
- Proxy nu trebuie sa stie intotdeauna tipul subiectului real

Lant de responsabilitati

Sablonul lantului de responsabilitati (chain of responsibility) este un sablon comportamental care permite evitarea cuplarii directe a expeditorului unei cereri cu un anumit destinatar, folosindu-se in acest sens clase intermediare.

Avantajele unui astfel de sablon:

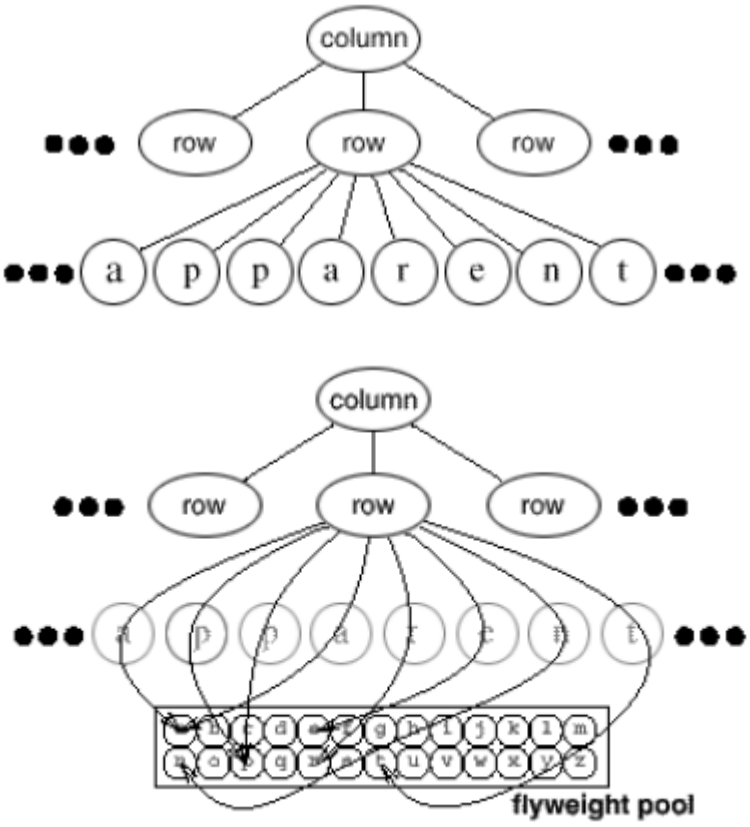
- expeditorul poate sa nu cunoasca exact care este destinatarul final al cererii sale, pe el interesandu-l doar ca respectiva sarcina sa fie indeplinita;
- clasele intermediare pot alege destinatarii, gestionand eventual si gradul de solicitare al acestora si gradul de solicitare a sistemelor de calcul pe care ruleaza acestia;
- o cerere poate fi procesata de mai multi destinatari, in acelasi timp, secvential sau respectand chiar anumite fluxuri de cereri;
- clasele intermediare pot realiza log-uri ale cererilor;
- lipsa oricarui potential destinatar poate fi aflata de catre expeditor printr-un mesaj primit de la clasele intermediare.



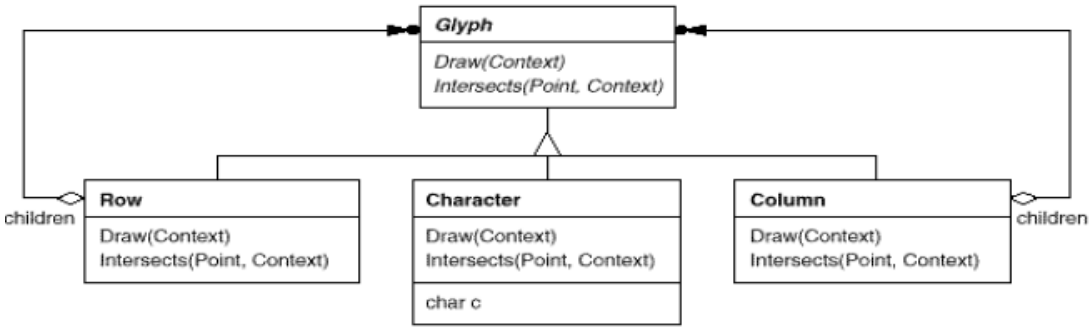
Flyweight

Intentia : Foloseste sharing pentru a sustine un numar mare de obiecte eficient.

Motivatie : un flyweight reprezinta un obiect impartit ce poate fi folosit in contexte multiple simultan.Se comporta ca un obiect independent in fiecare context , nu se distinge dintr-o stare a obiectului daca nu este impartit.



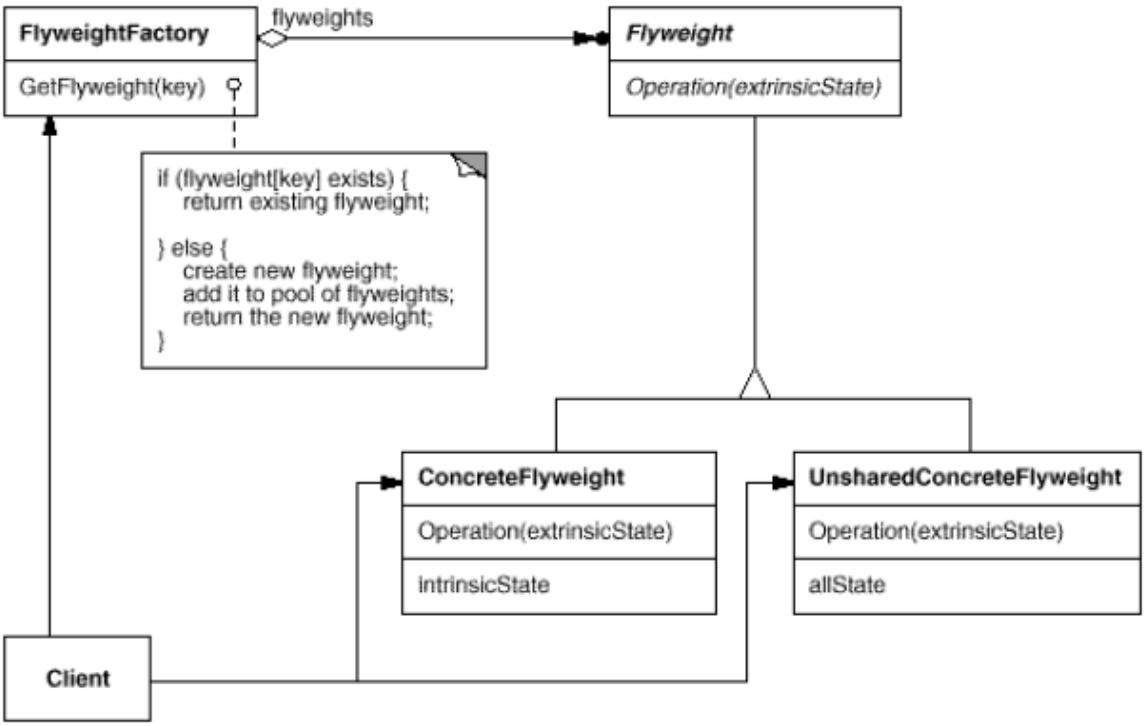
Structura claselor pentru obiectele de mai sus :



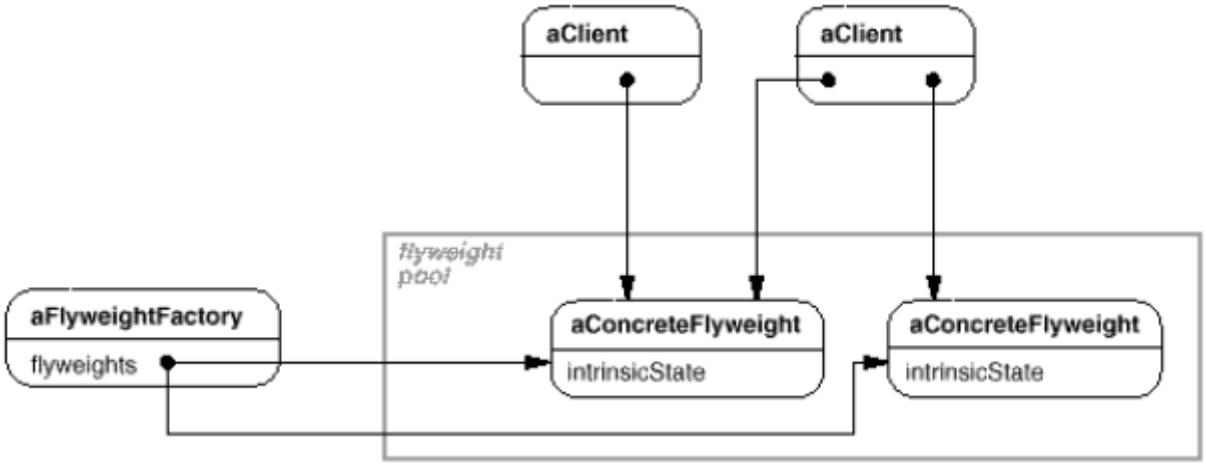
Aplicabilitate – modelul Flyweight se foloseste atunci cand urmatoarele afirmatii sunt corecte:

- O aplicatie foloseste un numar mare de obiecte
- Costul stocarii este mai mare din cauza cantitatii mari de obiecte
- Majoritatea starilor obiectelor pot fi preparate
- Multe grupuri de obiecte pot fi inlocuite de cateva obiecte impartite odata ce starea preparata este eliminata
- Aplicatia nu depinde de identitatea obiectului.

Structura



Urmatoarea digrama arata cum sunt impartite modele flyweight.



Participanti:

- Flyweight
- ConcreteFlyweight (Character)
- UnsharedConcreteFlyweight (Row , Column)
- FlyweightFactory
- Client

Colaborari:

- Clientii nu ar trebui sa instantieze ConcreteFlyweight in mod direct
- Afirmatia ca flyweight ca sa functioneze trebuie sa fie caracterizat fie ca intrinsec sau extrinsec.

Consecinte:

- Cu cat sunt mai multe flyweights impartite , castigurile de stocare sunt mai mari.
- Modelul flyweight este de obicei combinat cu modelul Composite.

Implementarea:

- Eliminarea starii extrinsece
- Managementul obiectelor impartite

Modele creationale

Uta Cosmin Alexandru
442A

Tiparele creationale ofera mecanisme de instantiere, facilitand crearea obiectelor intr-o maniera potrivita situatiei.

Abstract Factory

Scop

- Sa ofere o interfata pentru crearea de familii de obiecte inrudite sau dependente fara a le specifica clasele concrete.
- O ierarhie care incapsuleaza: multe "platforme" posibile si construirea unei suite de produse.
- Operatorul new este considerat daunator

Aplicabilitate

Se foloseste Abstract Factory cand:

- sistemul trebuie sa fie independent de cum sunt create, aduse impreuna sau reprezentate obiectele sale
- un sistem ar trebui sa fie configurat cu una din mai multe familii de produse
- o familie de obiecte-produs inrudite este creata pentru a fi folosite impreuna si este nevoie sa fie intarita aceasta constrangere
- se doreste furnizarea unei librarii de clase de produse si se doreste a se oferi doar interfetele acestora nu si implementarea lor

Problema

Daca se doreste ca o aplicatie sa fie portabila, trebuie sa incapsuleze dependentele de platforma. Aceste "platforme" s-ar putea referi la: sistemul de ferestre, sistemul de functionare, baze de date etc. Foarte des, aceasta incapsulare nu este tratata in avans si

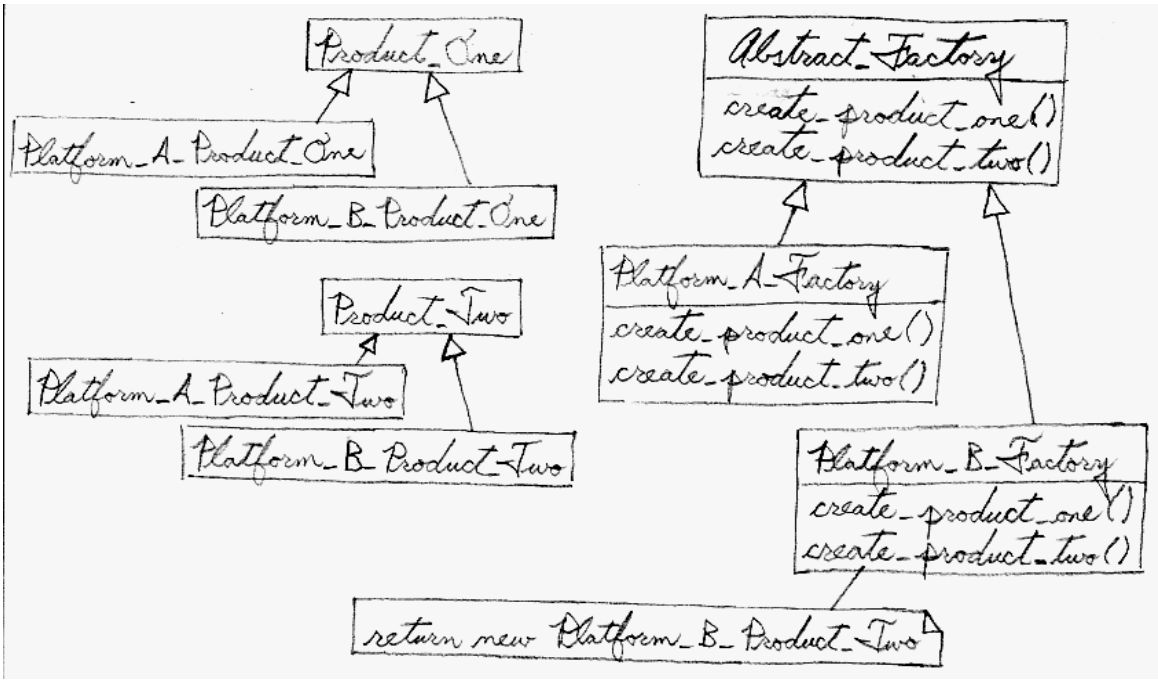
multe declaratii de genul #ifdef cu optiuni pentru diversele platforme curent suportate incep sa apara ca iepurii dupa construirea codului.

Implementarea

Cateva tehnici utile pentru implementarea modelului Abstract Factory:

1. Ca modele singleton. O aplicatie de acest fel are nevoie de obicei o singura instanta concreta a obiectului Factory pentru fiecare familie de produse. Deci cea mai buna implementare se face prin singleton.
2. Crearea produselor. AbstractFactory nu face decat sa d eclare o interfata pentru crearea produselor. Depinde de clasele derivate pentru obiectele concrete pentru a le crea in sine. Cel mai comun mod de a face acest lucru este de a defini o metoda factory pentru fiecare produs. Un factory concret va specifica produsele sale prin rescrierea metodei factory pentru fiecare.
3. Definirea de factories (fabrici) extensibile.

Structura



Modelul Abstract Factory este defineste o metoda factory pentru fiecare produs. Fiecare din aceste metode incapsuleaza (detine) operatorul new si clasele concrete, specifice platformei. Fiecare platforma este apoi modelata ca o clasa Factory derivata.

Builder

Scop

Scopul este de a separa constructia unui obiect complex de reprezentarea, infatisarea sa, astfel incat acelasi proces de construire poate crea diferite reprezentari.

Motivatie

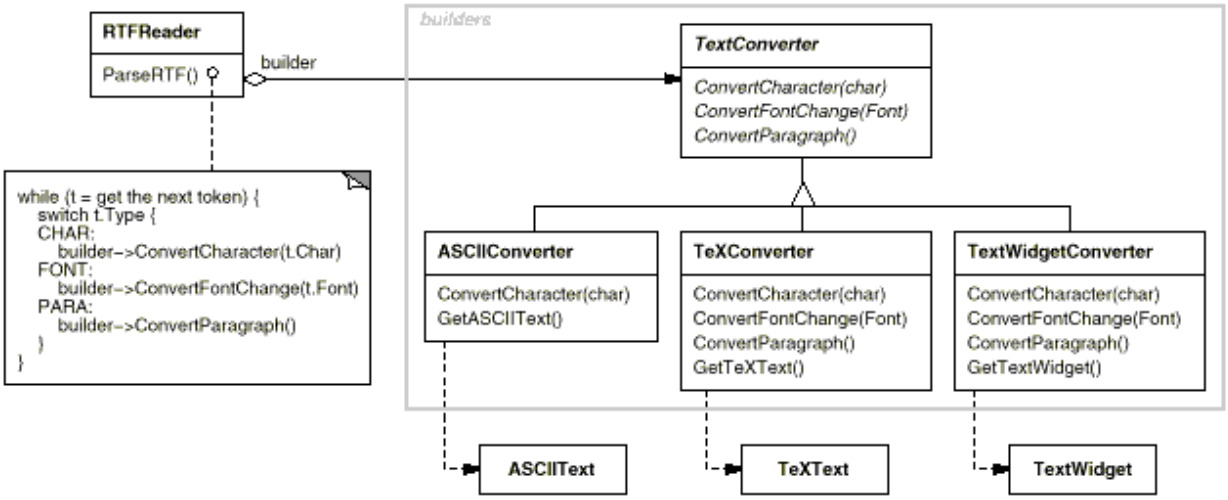
Un program care citeste fisiere (cititor) in formatul RTF (Rich Text Format) ar trebui sa fie capabil sa converteasca RTF in mai multe formate. Programul ar putea converti documentele RTF in format de text ASCII sau intr-un widget de text care sa permita editarea interactiva a acestuia. Oricum, problema este ca numarul de conversii posibile e nelimitat. Din aceasta cuza ar trebui sa fie usor sa se faca o conversie noua fara modificarea cititorului.

O solutie ar fi sa se configureze clasa RTFReader cu un obiect TextConverter care face conversia din RTF catre o alta reprezentare textuala. RTFReader parseaza documentul RTF, folosind obiectul RTFReader pentru a face conversia.

Subclase ale TextConverter se specializeaza in diferite conversii si formate. De exemplu, ASCIIConverter ignora orice cerere de conversie in orice alt tip decat text simplu (plain text). Un TeXConverter, pe de alta parte va implementa operatii pentru toate cererile de producere a unei reprezentari TeX care contine toate informatiile de stilizare a textului.

Fiecare tip de clasa de conversie contine mecanismul pentru crearea si asamblarea unui obiect complex si il ascunde in spatele unei interfete abstracte. Convertorul este separat de cititor, care este responsabil pentru parsarea documentului RTF.

Modelul Builder captureaza toate aceste relatii. Fiecare clasa de conversie este numita **builder** in acest model iar cititorul se numeste **director.** In exemplul prezentat, modelul Builder separa algoritmul de interpretare a formatului de text (acesta este parserul pentru documente RTF) de cum un format convertit este creat si reprezentat. Aceasta ne permite sa refolosim algoritmul de parsare a RTFReader pentru a crea diferite reprezentari de texte din documente RTF. Trebuie doar configurat RTFReader cu subclase diferite ale TextConverter.

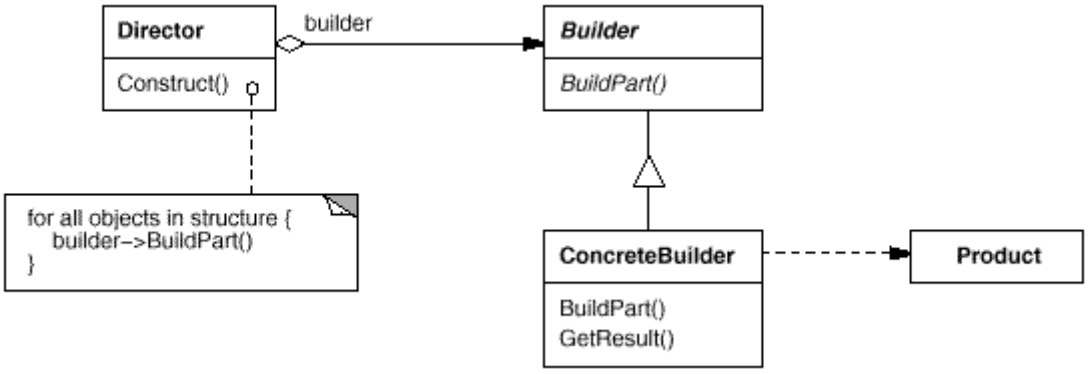


Aplicabilitate

Modelul Builder se foloseste cand:

- algoritmul pentru crearea unui obiect complex ar trebui sa fie independent de partile componente ale obiectului si de cum sunt acestea asamblate
- procesul construirii trebuie sa permita diferite reprezentari pentru obiectul ce se construieste.

Structura



Colaborari intre obiecte

- Clientul creaza obiectul Director si il configureaza cu obiectul Builder dorit.
- Directorul anunta Builder-ul de fiecare data cand o parte a produsului ar trebui construita
- Builder-ul se ocupa de cererile de la Director si adauga parti produsului
- Clientul primeste produsul de la Builder

Diagrama urmatoare arata cum Builder si Director coopereaza cu un client:

Consecinte

1. Permite varierea reprezentarii interne a produselor
2. Izoleaza codul pentru constructie fata de cel pentru reprezentare
3. Ofera un control fin asupra procesului de constructie

Implementare

De obicei exista o clasa abstracta Builder care defineste o operatie pentru fiecare componenta pe care un director ar putea-o cere pentru creare. Operatiile nu fac nimic imiplicit. O clasa concreta ConcreteBuilder rescrie operatiile pentru componentele de creat.

Factory Method

Scop

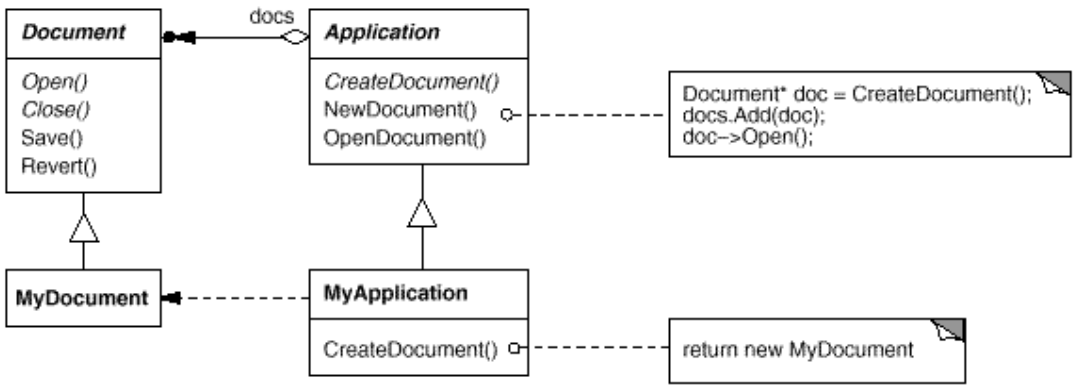
Defineste o interfata pentru crearea unui obiect, dar lasa subclase(clase derivate) sa decida ce clasa sa instantieze. Factory Method lasa o clasa sa cedeze instantierea subclaselor.

Modelul mai este cunoscut si ca : Virtual constructor.

Motivare

Frameworkurile folosesc clase abstracte pentru a defini si mentine relatii intre obiecte. Un framework este deseori responsabil de asemenea si pentru crearea acestor obiecte. Consideram un framework pentru aplicatii care pot prezenta documente multiple utilizatorului. Doua abstractizari cheie ale acestui framework sunt clasele Application si Document. Fiecare din acestea doua sunt abstracte iar clientii trebuie sa deriveze aceste clase pentru a realiza implementarile specifice aplicatiei. Pentru a crea o aplicatie de desenare, de exemplu, definim clasele DrawingApplication si DrawingDocument. Clasa Application este responsabila pentru administrarea documentelor si le va crea asa cum e necesar - de exemplu cand utilizatorul selecteaza Open sau New din meniu.

Deoarece subclasa particulara a lui Document ce trebuie instantiata este specifica aplicatiei, clasa Application nu poate prezice aceasta subclasa ce trebuie instantiata. Clasa Application nu stie decat cand un nou document ar trebui creat nu si ce tip de Document ar trebui creat. Acest lucru duce la o dilema: frameworkul trebuie sa instantieze clase dar nu stie decat de clase abstracte pe care nu le poate instantia. Modelul creational Factory Method ofera o solutie. Incapsuleaza cunostiintele asupra subclasei lui Document ce trebuie creata si "muta" aceste cunostiinte in afara frameworkului.



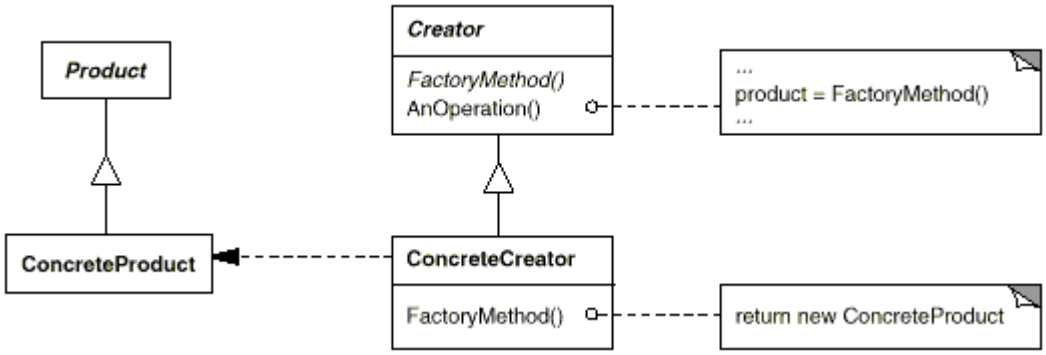
Subclasele clasei Application redefinesc operatia abstracta CreateDocument pentru a returna subclasa cea mai potrivita a clasei Document . Odata ce o subclasa a Application este instantiata pot fi instantiate Documente specifice aplicatiei fara sa se cunoasca clasele acestora. Numim CreateDocument o metoda factory deoarece este responsabila pentru "fabricarea" unui obiect.

Aplicabilitate

Se foloseste Factory Method cand:

- o clasa nu poate anticipa clasa de obiecte ce trebuie create
- o clasa doreste ca subclasele sale sa specifice obiectele pe care le creaza
- clasele delega responsabilitatea catre una din subclasele ajutatoare si se doreste sa se localizeze cunostiintele asupra carei subclasa ajutatoare este delegata.

Structura



Prototype

Scop

Specifica tipul de obiecte de creat folosind o instanta prototip si creaza obiecte noi copiind acest prototip.

Utilizare

Acest model este utilizat:

- pentru evitarea subclaselor unui obiect creator din aplicatia client, cum face modelul abstract factory
- evitarea mostenirii costului crearii unui obiect nou in modul standard cand e foarte scump pentru o aplicatie data

Structura

1) Se defineste o metoda clone() intr-o ierarhie existenta: interfata in clasa de baza si implementarea in clasele derivate

2) Fiecare implementare din clasele derivate incapsuleaza folosirea operatorului new si returneaza o instanta a lui insusi.

3)Clientul delega un obiect prototip de fiecare data cand o instanta a ierarhiei e necesara.

Modele operationale

Horsa Alexandru
442A

Template Method

Intentie
Defineste scheletul unui algoritm, lasand cativa pasi sa fie implementati de subclase. Metoda Template lasa subclasele sa redefineasca cativa pasi ai algoritmului fara a modifica structura acestuia.

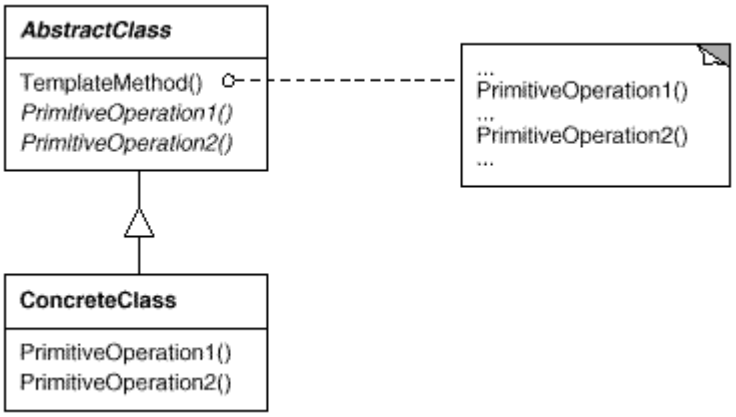
Motivare
Spre exemplu intr-un framework care prevede clasele Application si Document, clasa Application este responsabila pentru deschiderea documentelor stocate intr-un format extern, cum ar fi un fisier. Un obiect de tipul Document reprezinta informatia din document dupa citirea din fisier. Aplicatiile construite cu acest framework pot deriva Application si Document pentru a nevoi specifice. Spre exemplu o aplicatie de desenat defineste subclasele DrawApplication si DrawDocument.

Aplicabilitate
Metoda Template poate fi folosita pentru:

- Implementarea partilor invariante ale unui algoritm, lasand partile variabile subclaselor.
- Evitarea codului duplicat prin centralizarea partilor comune intr-o singura clasa.

- Controlul derivarii claselor. Se poate definii o metoda template care sa apeleze operatii de conexiune in anumite puncte cheie, permitand astfel extinderea numai acolor puncte.

Structura



Participanti

- AbstractClass (Application) – defineste operatiile abstracte primitive pe care subclasele sunt nevoite sa le defineasca. Metoda template apeleaza operatiile primitive cat si operatiile definite in AbstractClass sau cele care apartin altor obiecte
- ConcreteClass(MyApplication) – implementeaza operatiile primitive ce vor avea grija de pasii specifici algoritmului

Consecinte

- Metodele template sunt fundamentale pentru refolosirea codului. Acestea sunt importante in special in librarii.
- Metodele template conduc la o structura de control inversata. Astfel, clasa parinte apeleaza operatiile unei subclase si nu invers.

State

Definitie

Modelul State permite unui obiect sa isi schimbe comportamentul cand starea sa interna se modifica. Schimbarea are loc prin folosirea mostenirii, clasele derivatate reprezentand starile si functionalitatea.

Aplicabilitate

- In cazul in care avem nevoie de o clasa „context” care sa reprezinte o singura interfata catre exterior. Acest lucru se obtine folosind o clasa de baza abstracta State.
- Cand vrem sa reprezentam diferite stari sub forma unor clase derivate dintr-o clasa de baza State.

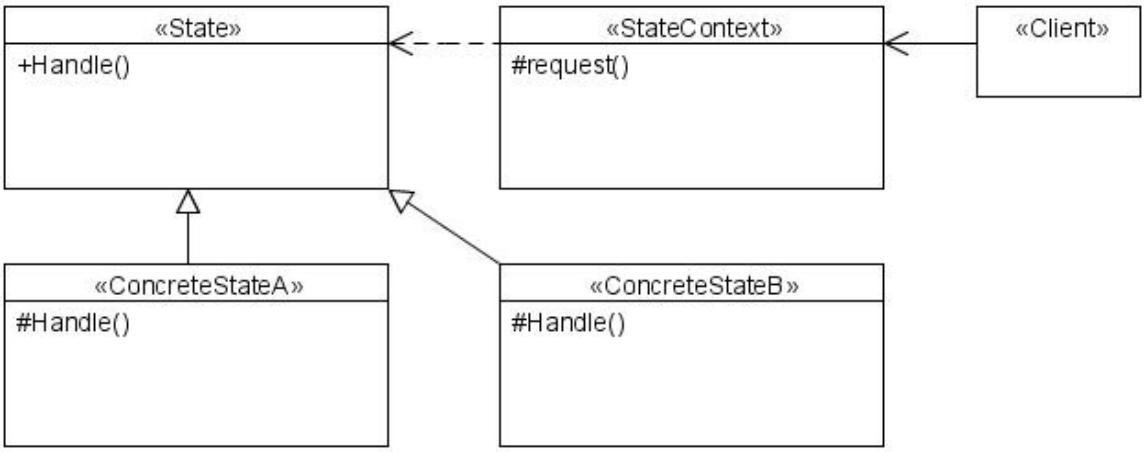
Beneficii

- Codul este mai curat cand fiecare stare este o clasa
- Folosirea claselor in loc de constante

Dezavantaje/Consecinte

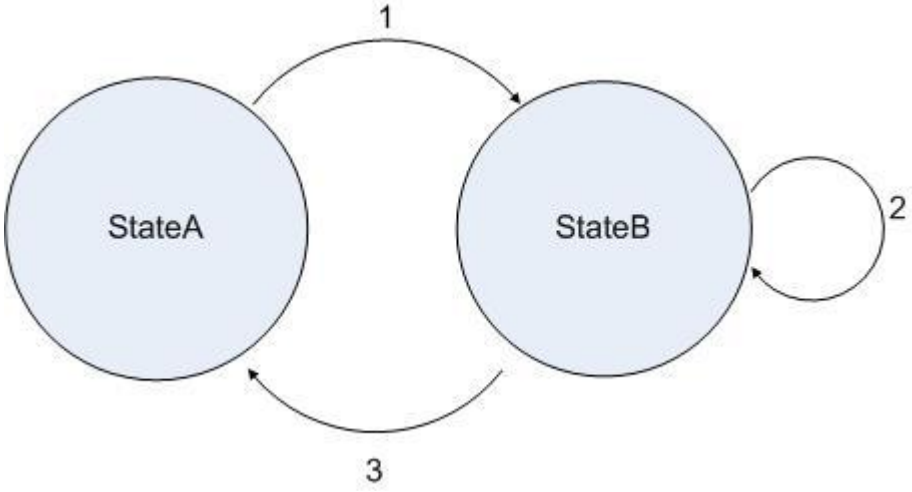
- Genereaza mai multe clase mici, dar simplifica si clarifica programul
- Elimina necesitatea instructiunilor conditionale lungi si imprastiate prin cod

Structura



Exemplul #1

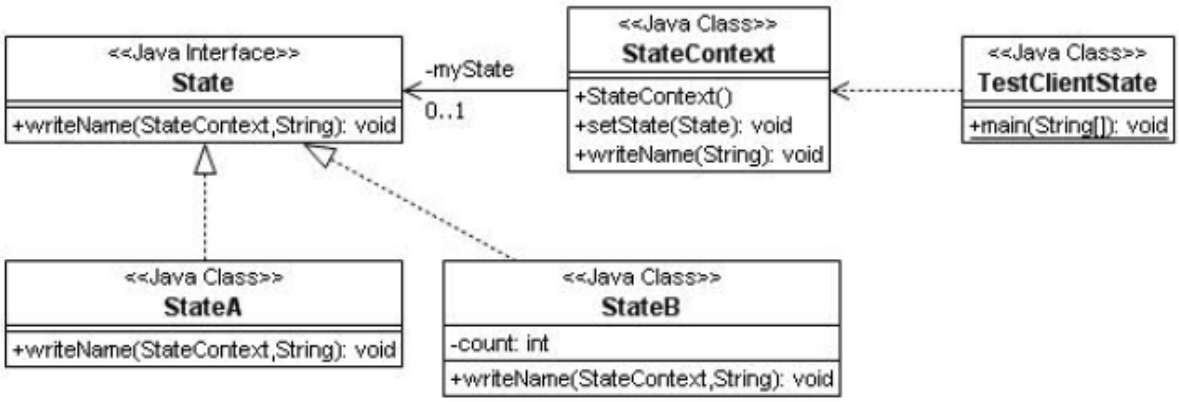
Acest exemplu are doua stari. StateA scrie numele cu caractere mici, iar StateB scrie numele cu majuscule.




```
public interface State {
    public void writeName(StateContext stateContext, String name);
}

public class StateA implements State {
    public void writeName(StateContext stateContext, String name) {
            System.out.println(name.toLowerCase());
            stateContext.setState(new StateB());
    }
}
```

```
public class StateB implements State {
      private int count=0;

      public void writeName(StateContext stateContext, String name){
            System.out.println(name.toUpperCase());
            if(++count>1) {
                  stateContext.setState(new StateA());
            }
      }
}


public class StateContext {
      private State myState;

      public StateContext() {
            setState(new StateA());
      }

      public void setState(State stateName) {
            this.myState = stateName;
      }

      public void writeName(String name) {
            this.myState.writeName(this, name);
      }
}


public class TestClientState {
      public static void main(String[] args) {
            StateContext sc = new StateContext();
            sc.writeName("Monday");
            sc.writeName("Tuesday");
            sc.writeName("Wednesday");
            sc.writeName("Thursday");
            sc.writeName("Saturday");
            sc.writeName("Sunday");
      }
}
```

Diagrama claselor

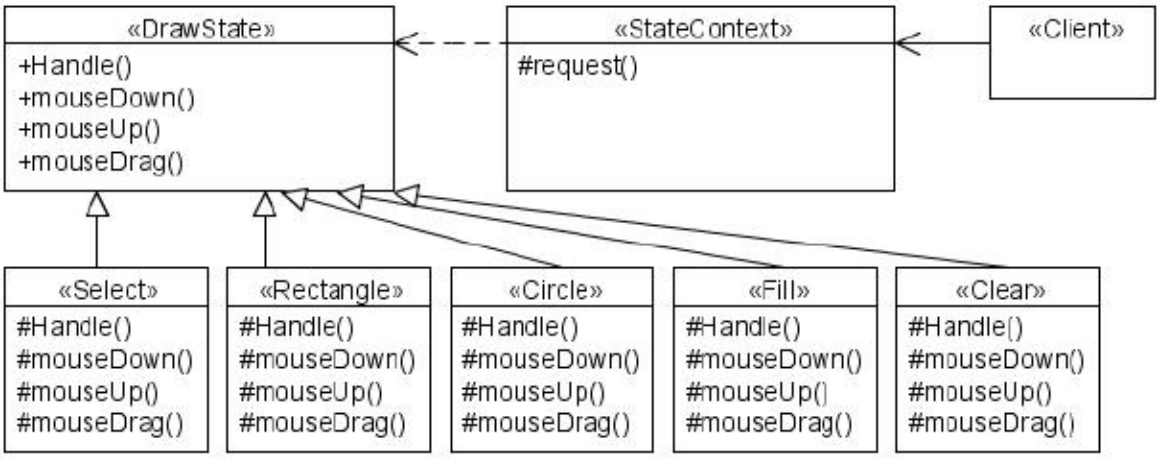


Exemplul #2

O aplicatie de desenat cu diverse unelte cum ar fi: select, dreptunghi, cerc, umplere si stergere. Fiecare unealta se comporta diferit la click sau la tragere. Starea editorului afecteaza comportamentul programului. Apare nevoie de gestionare a catorva activitati ale mouse-ului.

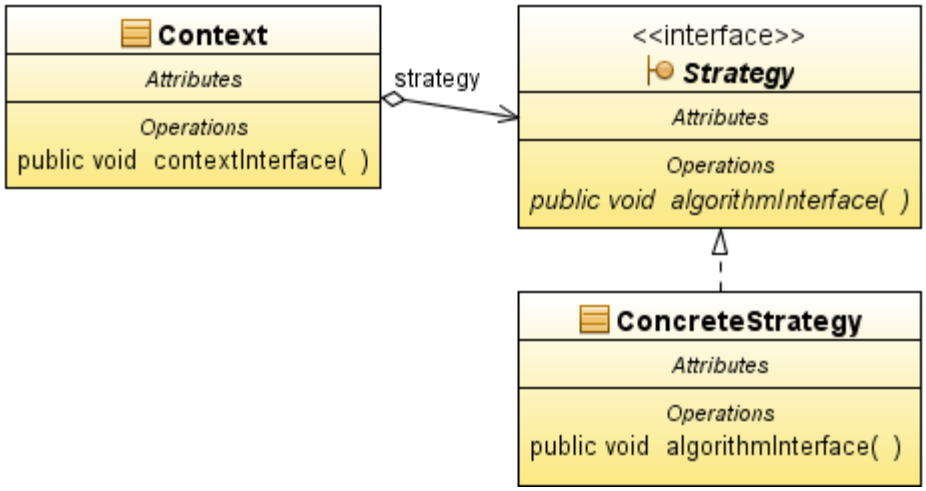
```
public class DrawState {
      public void mouseDown(int x, int y) {}
      public void mouseUp(int x, int y) {}
      public void mouseDrag(int x, int y) {}
}

public class Rectangle extends DrawState {
      //create a new Rectangle where mouse clicks
      public void mouseDown(int x, int y) {
            canvas.addDrawing(new visRectangle(x, y));
      }
}
```

Diagrama claselor



Strategy

Sablonul strategie ne ajuta sa alegem un anumit algoritm de utilizat in functie de un context. Sablonul contine un grup de algoritmi, fiecare din acestia fiind incapsulat intr-un obiect. Clientii ce folosesc algoritmii nu depind de acestia, variind in mod independent.

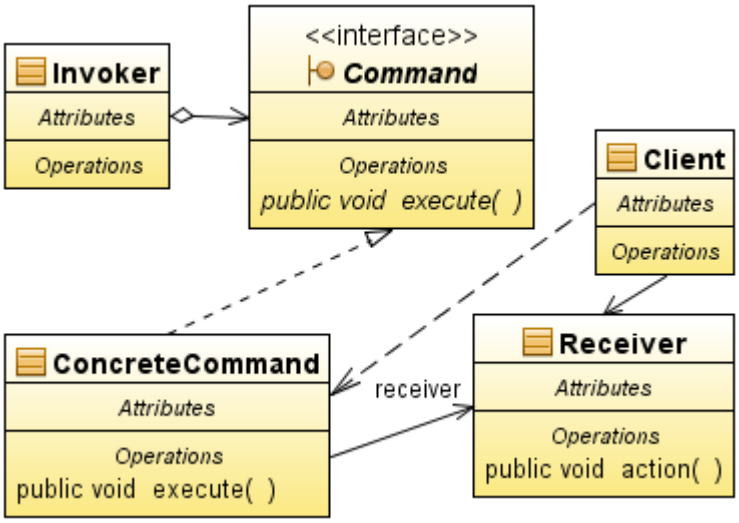


Obiectele de tip context contin cate un obiect strategy, dar alegerea unei strategii concrete se realizeaza in functie de context.

Conform sablonului strategy comportamentul unei clase nu ar trebui sa fie mostenit, ci specific contextului in care ruleaza.

Command

Conform sablonului command un obiect poate incapsula toate informatiile necesare pentru apelarea unei metode a altui obiect, cum ar fi: numele metodei de apelat, obiectul ce detine metoda si valorile de transmis parametrilor.



Clientul instantiaza obiectul de tip comanda si il pregateste pentru a fi apelat la un moment ulterior.

Obiectul invoker decide cand va fi apelat obiectul comanda.

Obiectul receiver este cel care va efectua o actiune ca urmare a lansarii in executie a comenzii.

Beneficiile acestui sablon tin de faptul ca executia unei anumite metode poate fi pregatita din timp, in asa fel incat aceasta sa fie lansata fara sa i se stie numele, obiectul de care apartine si momentul exact al executiei.

Sablonul command poate fi utilizat pentru a realiza aplicatii cu:
- functionalitati de tip „undo”;
- comportament tranzactional;
- progress bar sincronizat cu executia unui grup de comenzi;
- functionalitati de tip „wizard”;
- inregistrari de macro-uri;

Interpreter

Sablonul interpreter descrie cum se pot interpreta expresiile intr-un anumit limbaj. El isi gaseste utilitatea in aplicatiile economice in care se doreste salvarea unor formule de calcul intr-un format accesibil utilizatorilor finali si folosirea ulterioara de catre aplicatie a acestor formule.

Conform sablonului interpreter atat operanzii, cat si operatorii dintr-o expresie ar trebui sa aiba aceeasi interfata.

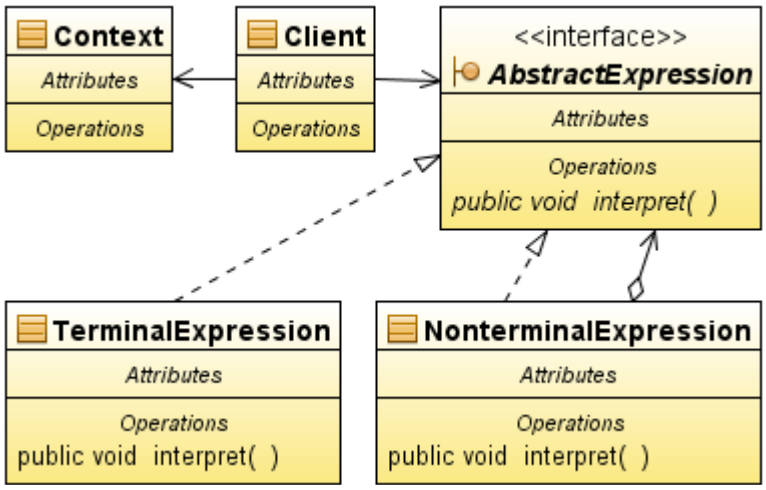


Diagrama claselor de mai sus nu surprinde nevoia existentei unui evaluator de expresii. Acest evaluator trebuie sa imparta expresia de interpretat in mai multe parti cu un parser. Tot evaluatorul trebuie sa parcurga fiecare parte a expresiei:

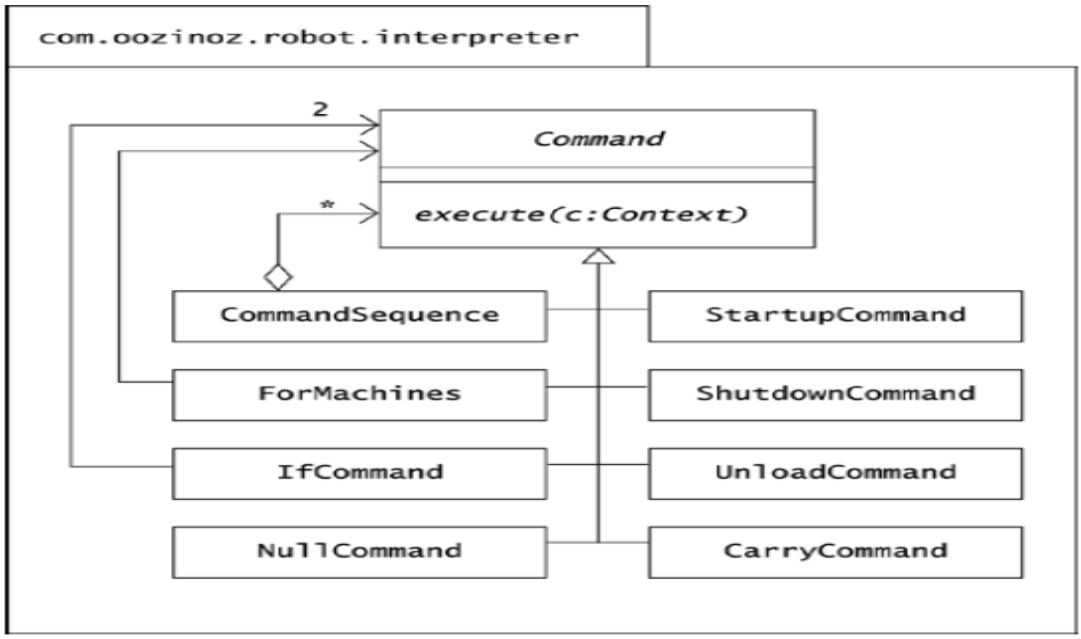
- cand intalneste un operand (o variabila pe care are voie utilizatorul sa o foloseasca intr-o anumita formula) il introduce intr-o stiva;
- cand intalneste un operator extrage din stiva valorile operanzilor afectati si introduce in loc rezultatul aplicarii operatorului asupra operanzilor.

Daca se pleaca de la o expresie corecta la sfarsitul parcurgerii acesteia in stiva utilizata va ramane o singura valoare, ea fiind chiar rezultatul interpretarii.

Exemple de aplicatii in care ar putea fi utilizat sablonul interpreter:
- aplicatii de salarizare;

- aplicatii de procesare a datelor la importul sau exportul lor din diferite sisteme;
- aplicatii de interpretare a unui dialect SQL sau de traducere dintr-un dialect SQL in altul.

In mod general, acest sablon poate fi folosit la interpretarea oricarei comenzi.



Modele de extensibilitate

Botoc Catalin

443A

Decorator

Definitie

Acest model face posibila atasarea unor responsabilitati aditionale si modificarea functionalitatii unei instante in mod dinamic. Reprezinta o alternativa la derivare pentru extinderea functionalitatii, utilizand compozitia in loc de mostenire.

Aplicabilitate

- Pentru adaugare de responsabilitati obiectelor individuale in mod dinamic si transparent
- Pentru responsabilitati ce vor avea nevoie de modificari in viitor
- Cand este imposibila extinderea cu ajutorul mostenirii.

Beneficii

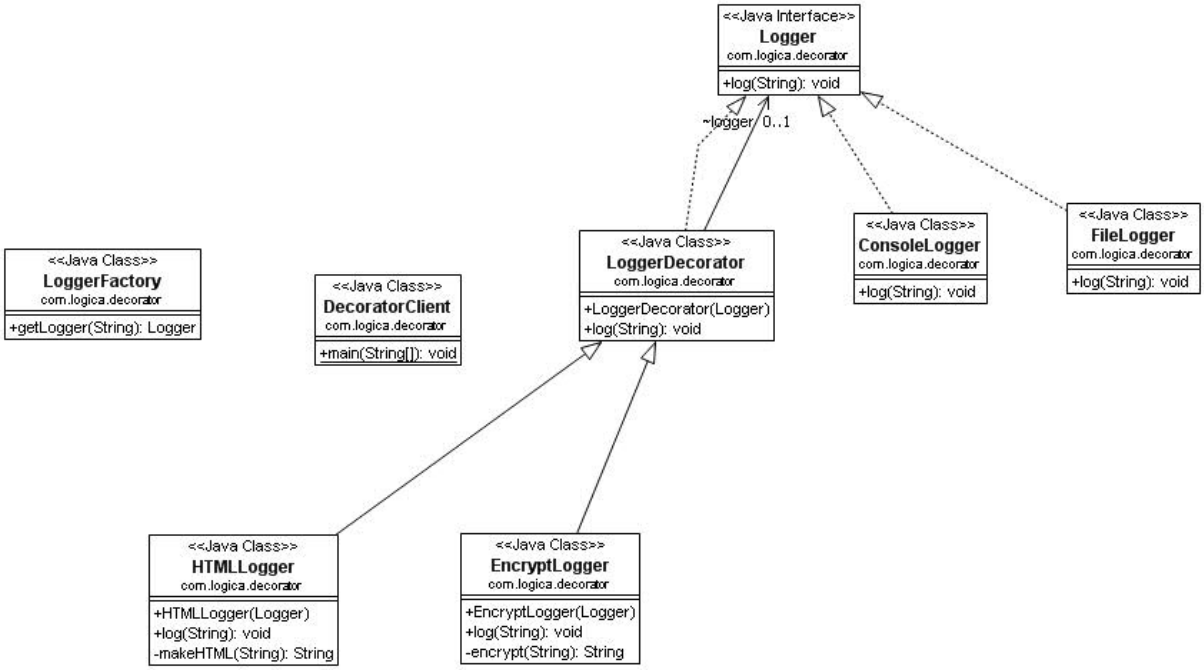
- Ofera mai multa flexibilitate decat mostenirea.

- Se evita aparitia unor clase incarcate din perspectiva functionalitatilor in partea superioara a ierarhiei.
- Simplifica programarea deoarece nu este necesara suprascrierea tuturor functionalitatilor.
- Extensibilitatea este marita deoarece modificarile au loc prin creearea unor clase noi.

Dezavantaje/Consecinte

La implementarea acestui model trebuie ca interfata componentei sa fie simpla. O interfata complexa a componentei are consecinte negative asupra decoratorului. Un alt dezavantaj al acestui model este scaderea performantei asociata cu un lant lung de decoratori.

Structura



Exemplu:

```
package com.logica.decorator;
public interface Logger {
      public void log(String msg);
}


package com.logica.decorator;
public class LoggerDecorator implements Logger {
      Logger logger;

      public LoggerDecorator(Logger logger) {
            super();
            this.logger = logger;
      }

      public void log(String msg) {
            logger.log(msg);
      }
}
```

```java
package com.logica.decorator;
public class ConsoleLogger implements Logger {
	public void log(String msg) {
		System.out.println("Detta ska skrivas till consolen! "+
			msg);
	}
}
```

```java
package com.logica.decorator;
public class EncryptLogger extends LoggerDecorator {
	public EncryptLogger(Logger logger) {
		super(logger);
	}

	public void log(String msg) {
		msg = encrypt(msg);
		logger.log(msg);
	}

	private String encrypt(String msg) {
		msg = msg.substring(msg.length()-1) + msg.substring(0,
			msg.length() -1);
		return msg;
	}
}
```

```java
package com.logica.decorator;
public class HTMLLogger extends LoggerDecorator {
	public HTMLLogger(Logger logger) {
		super(logger);
	}

	public void log(String msg) {
		msg = makeHTML(msg);
		logger.log(msg);
	}

	private String makeHTML(String msg) {
		msg = "<html><body>" + "<b>" + msg + "</b>" +
			"</body></html>";
		return msg;
	}
}
```

```java
package com.logica.decorator;
public class LoggerFactory {
	public static final String TYPE_CONSOL_LOGGER = "console";
	public static final String TYPE_FILE_LOGGER = "file";

	public Logger getLogger(String type) {
		if(TYPE_CONSOL_LOGGER.equals(type)) {
			return new ConsoleLogger();
		} else {
			return new FileLogger();
		}
	}
}
```

```
package com.logica.decorator;
public class DecoratorClient {
	public static void main(String[] args) {
		LoggerFactory factory = new LoggerFactory();
		Logger logger =
			factory.getLogger(LoggerFactory.TYPE_FILE_LOGGER);
		HTMLLogger htmlLogger = new HTMLLogger(logger);
		htmlLogger.log("A message to log");
		EncryptLogger encryptLogger = new EncryptLogger(logger);
		encryptLogger.log("A message to log");
	}

}
```

Iterator

Design-ul iteratorului produce o cale de a accesa elementele unui obiect agregat secventional fara a-i expune reprezentarile subliniate.

Sablonul iterator este folosit pentru a accesa elementele unui agregat (a unei colectii) in mod secvential, fara a ne folosi de caracteristicile acestor elemente.

Colectiile java cum ar fi ArrayList si HashMap au implementate modelul de iterator.

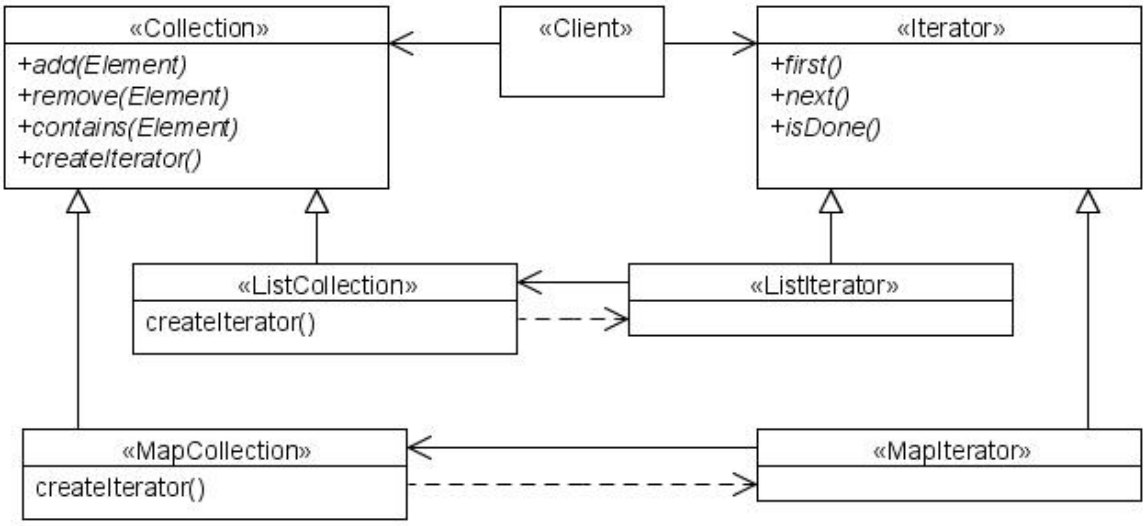
Beneficii:

- Acelas iterator poate fi utilizat la mai multe agregate
- Iti permite traversarea agregatului in diferite directii in functie de necesitatile tale
- Incapsuleaza structura interna in functie de cum apare iteratia
- Nu necesita incarcarea clasei tale cu operatori pentru diferite traversari

Consecinte:

Nu este un fir sigur de executie decat daca este un iterator robust care permite inserarea si stergerea. Se poate rezolva aceasta problema prin acceptarea utilizarii de catre iterator a unui memento pentru a captura fragmente din iteratie.

Structura:



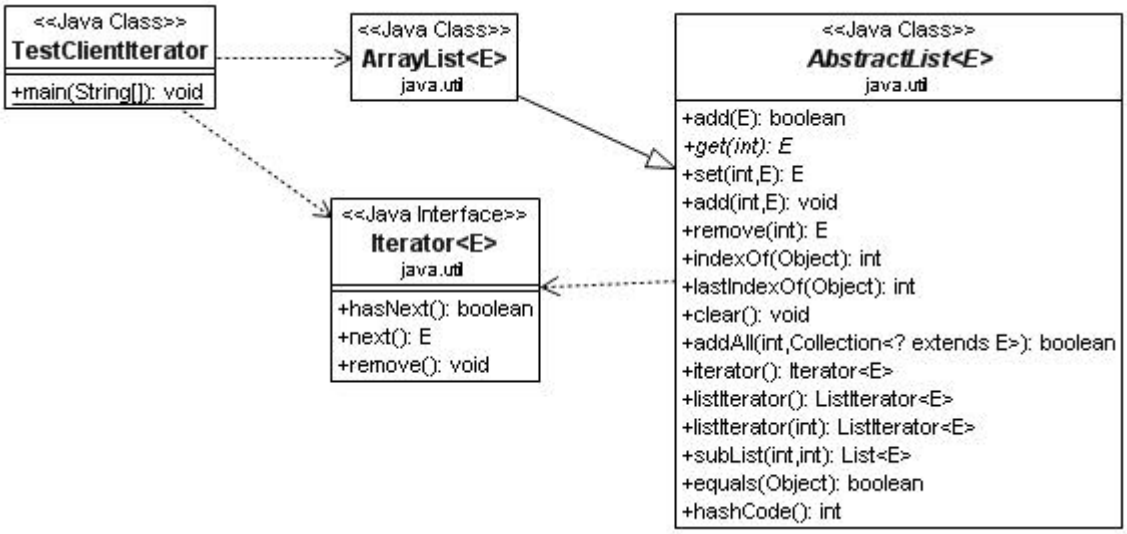
Exemplu:

Acest exemplu prezinta cum iti poti scrie propriul iterator:

```
import java.util.*;
public class BitSetIterator implements Iterator<Boolean> {
private final BitSet bitset;
private int index;
public BitSetIterator(BitSet bitset) {
this.bitset = bitset;
}
public boolean hasNext() {
return index < bitset.length();
}
public Boolean next() {
if (index >= bitset.length()) {
throw new NoSuchElementException();
}
boolean b = bitset.get(index++);
return new Boolean(b);
}
public void remove() {
throw new UnsupportedOperationException();
}
}
public class TestClientBitSet {
public static void main(String[] args) {
// create BitSet and set some bits
BitSet bitset = new BitSet();
bitset.set(1);
bitset.set(19);
bitset.set(20);
bitset.set(47);
BitSetIterator iter = new BitSetIterator(bitset);
while (iter.hasNext()) {
Boolean b = iter.next();
```

```
String tf = (b.booleanValue() ? "T" : "F");
System.out.print(tf);
}
System.out.println();
        }
}
```

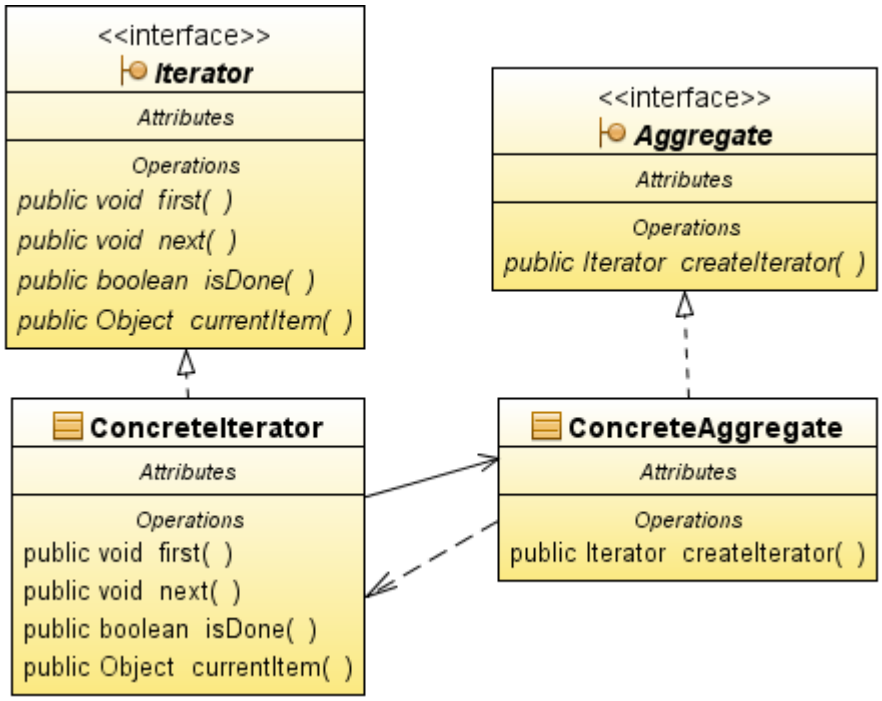
Urmatoarea diagrama-clasa ilustram structura din exemplul folosit:



Exemplu de utilizare al ArrayList:

```
import java.util.*;
public class TestClientIterator {
        public static void main(String[] args) {
                ArrayList<Object> al = new ArrayList<Object>();
                al.add(new Integer(42));
                al.add(new String("test"));
                al.add(new Double("-12.34"));
                for(Iterator<Object> iter=al.iterator();
                        iter.hasNext();)
                        System.out.println( iter.next() );
                // JEE5 syntax
                for(Object o:al)
                        System.out.println( o );
        }
}
```

In orice moment unul din elementele colectiei este considerat ca fiind elementul curent.



Operatii ale elementelor colectiei de parcurs ar putea fi:
- currentItem;
- first;
- last;
- next;
- previous;
- hasNext;
- hasPrevious;
- isFirst;
- isLast.

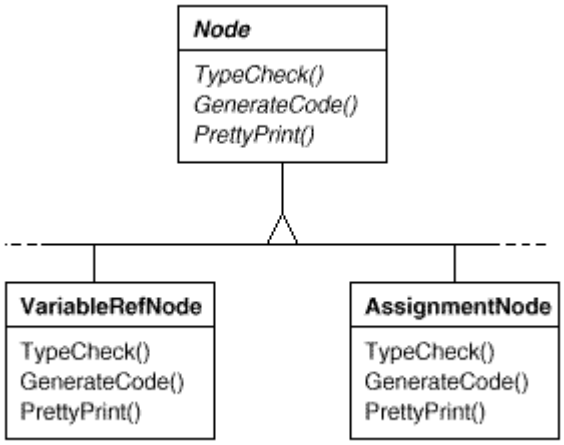
Sablonul iterator are un grad mai mare de importanta in mediile de dezvoltare in care nu sunt implementate complet tipuri de date compuse (vectori, liste, stive).

Visitor

Reprezinta o informatie ce urmeaza a zi executata asupra unui element al unei structuri-obiect. Visito-ul iti permite sa definesti o noua operatie fara sa schimbi clasele elementelor saupra carora operezi.

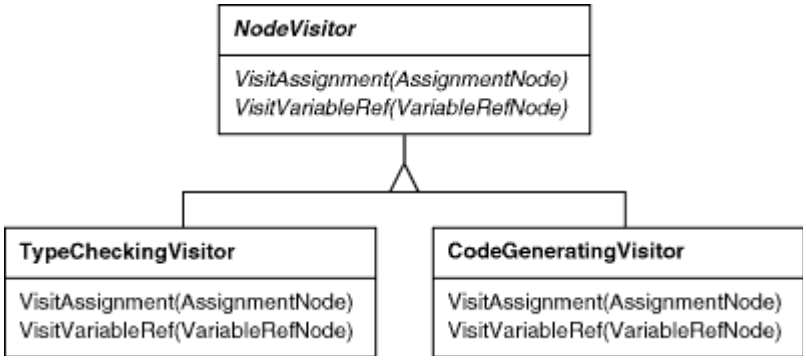
Considerand un compilator care reprezinta programele ca arbori de sintaxe abstracte. Va fi nevoie executarea de operatia asupra arborilor de sintaxe abstracte pentru analize ca verificarea daca toate variabilele sunt definite. De asemenea va fi necesara generarea de cod. Deci este posibila definirea operatiilor pentru verificarea tipurilor, optimizarea codulu, verificarea daca variabilelor le-au fost asignate valori inainte de utilizare, si asa mai departe. Mai mult decat atat , putem folosi arbori de sintaxe abstracte pentru preaty-printing, restructurarea programelor, instrumentarea codurilor.

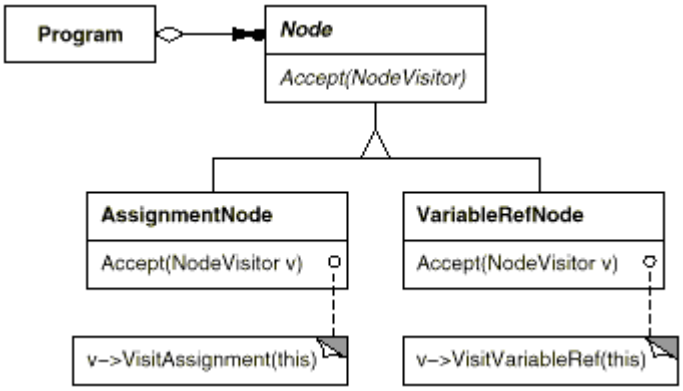
Setarea nodurilor se realizeaza astfel:



Aceasta diagrama reprezinta o parte din ierarhia clasei Node. Problema aici este ca distribuirea tuturor acestor operatii dealungul variatelor calse Node conduce la un sistem care este greu de inteles, intretinut si schimbat.

Le putem avea pe ambele prin impachetarea operatiunilor respective din fiecare clasa in obiecte separate numite „visitor”. Cand un element accepta vizitatorul trimite o cerere catre vizitator care codeaza elementele clasei. De asemenea include elementele ca un argument. Vizitatorul va executa atunci operatiile pentru acel element, operatia care obisnuia sa fie in clasa elementului.



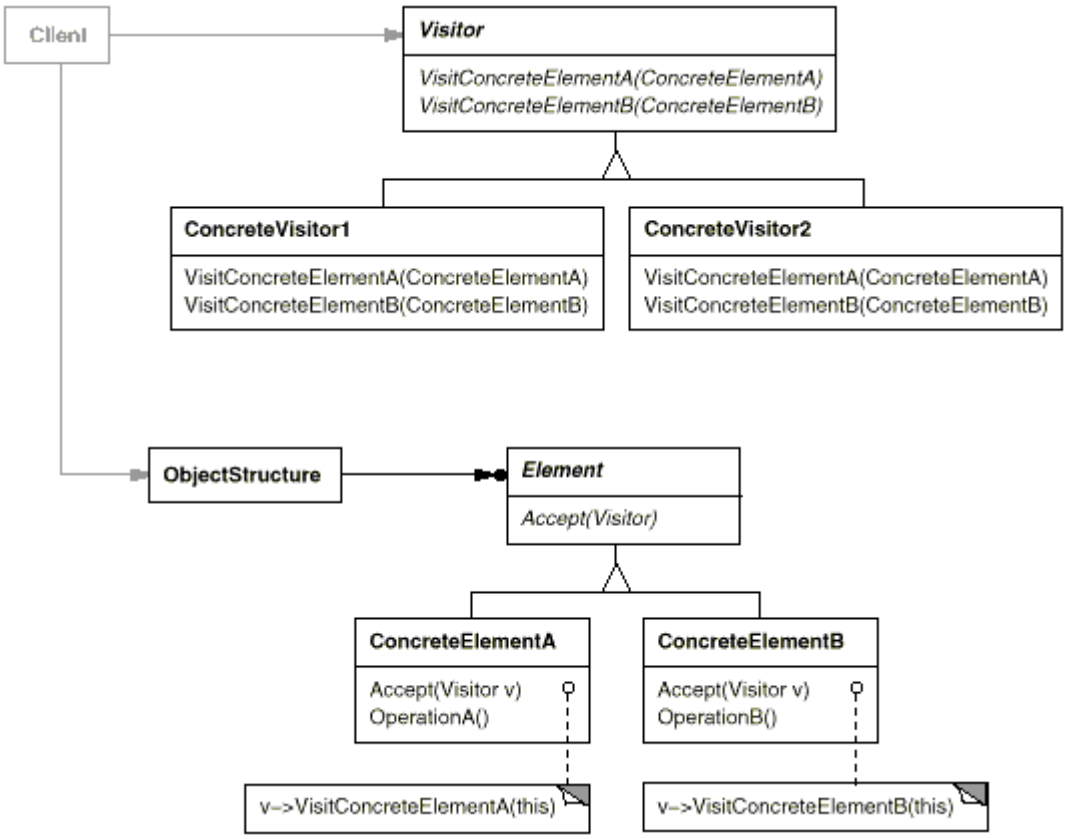


Utilizam modelul Visitor atunci cand:

- O structura obiect contine mai multe clase de obiecte cu diferite interfete, si doresti sa realizezi operatii asupra acestor obieacte care depind de clasele lor concrete.

- Multe operatii distincte au nevoie sa fie executate in structuri de obiecte, si iti doresti sa eviti poluarea claselor lor cu aceste operatii. Visitor-ii iti permit sa pastrezi operatiunile respective mai puternice prin definirea lor intr-o singura clasa.
- Clasele care definesc structura obiectelor se schimba rar , dar tu doresti des sa definesti noi operatori in structura.

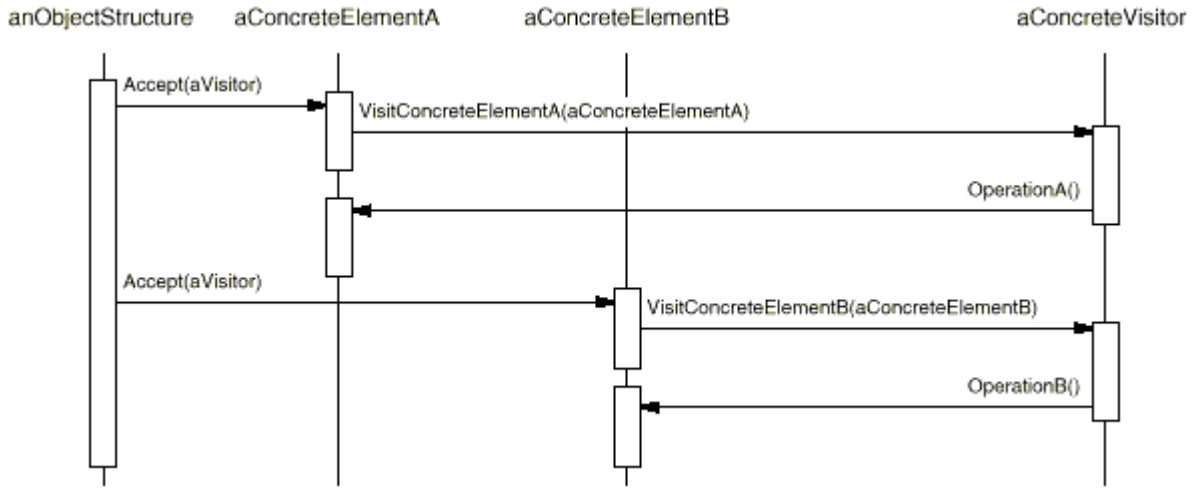
Structura:



Colaborari:

- Un client care utilizeaza modelul Visitor trebuie sa creeze un obiect ConcreteVisitor si atunci traversarea structurii obiect, vizitand fiecare element cu visitor.
- Cand un element este vizitat, se cheama operatia Vizitator care corespunde clasei sale.

Urmatoarea diagrama de interactiune ilustreaza colaborarea dintre un obiect structura , un vizitator si doua elemente.



Implementare :

Fiecare structura obiect va avea o clasa asociata vizitator. Aceasta clasa abstracta vizitator declara o operatie VisitConcretElement pentru fiecare clasa de ConcreteElement care defineste structura obiect. Fiecare operatie de vizita a Visitor-ului isi declara argumentul ca fiind un ConcreteElement particular, permitandu-i vizitatorului sa acceseze interfata ConcreteElement-ului direct.

Clasa Visitor se declara in C++ astfel:

```
class Visitor {
public:
virtual void VisitElementA(ElementA*);
virtual void VisitElementB(ElementB*);
// and so on for other concrete elements
protected:
Visitor();
};
```

Elementele concrete sunt declarate astfel:

```
class Element {
public:
virtual ~Element();
virtual void Accept(Visitor&) = 0;
protected:
Element();
};
class ElementA : public Element {
public:
ElementA();
virtual void Accept(Visitor& v) { v.VisitElementA(this); }
};
class ElementB : public Element {
public:
ElementB();
virtual void Accept(Visitor& v) { v.VisitElementB(this); }
};
```

O clasa CompositeElement poate implementa Accept astfel:

```
class CompositeElement : public Element {
public:
virtual void Accept(Visitor&);
private:
List<Element*>* _children;
};
void CompositeElement::Accept (Visitor& v) {
ListIterator<Element*> i(_children);
for (i.First(); !i.IsDone(); i.Next()) {
i.CurrentItem()->Accept(v);
}
v.VisitCompositeElement(this);
}
```

Bibliografie

1. Metsker, Steven John, Design Patterns Java Workbook

2. Benneth Christiansson, Mattias Forss, Design Patterns Explained with Java and Uml2

3. Erich Gamma, Richard Helm, Design Patterns: Elements of Reusable Object-Oriented Software