SABINA COSTACHE

INGINERIA PROGRAMELOR

SABINA COSTACHE

INGINERIA PROGRAMELOR

EDITURA FUNDAȚIEI UNIVERSITARE
„Dunărea de Jos” – GALAȚI - 2015

UNIVERSITATEA DUNĂREA DE JOS GALAȚI

Facultatea de Automatică, Calculatoare, Inginerie Electrică şi Electronică

Editura Fundației Universitare „Dunărea de Jos” din Galați
este acreditată CNCSIS

Referent ştiințific:
Prof. univ. dr. ing. LUMINIȚA DUMITRIU

Descrierea CIP a Bibliotecii Naționale a României
COSTACHE, SABINA
Ingineria Programelor / Sabina Costache. - Galați : Editura Fundației Universitare "Dunărea de Jos", 2015
Bibliogr.
ISBN 978-973-627-544-9

004.42(075)

www.editura.ugal.ro
editura@ugal.ro

Director, prof. dr. Cosma Tudose

ISBN 978-973-627-544-9

CUPRINS

PREFAȚĂ

"…Pentru a deveni un maestru în software design, trebuie să studiezi design-ul folosit de alți maeştri. În profunzimea acelui design vei descoperi şabloane ce pot fi folosite în alte design-uri. Aceste şabloane trebuie înțelese, memorate, şi aplicate până ce devin a doua natură." [Robert Martin]

Scopul ingineriei programării este scrierea unor programe de calitate. Pentru a deveni expert în programare se presupune, în primul rând, că stăpânești **regulile** de bază ale „jocului" (algoritmi, structuri de date şi limbaje de programare, exercițiu şi experiență în programare). O a doua etapă este înțelegerea acelor **criterii** şi **principii** care ne ajută să definim calitatea în general – şi de care ne vom ocupa în Capitolul 2. Acolo vom vedea care sunt scopurile design-ului, conceptele –cheie şi principiile, criteriile calității design-ului şi principiile şi regulile design-ului de calitate (importanța coeziunii, cuplajului, ascunderii informației, controlului dependențelor).

În Capitolul 3 introducem limbajul vizual UML, care ne ajută să descriem simplu şi concis structura unui program (diferitele soluții de implementare a unor probleme de design).

Capitolul 4 este dedicat regulilor, euristicilor şi principiilor de calitate specifice programelor orientate-obiect, iar Capitolele 5-8 ne vor exemplifica aplicarea practică a calității prin intermediul unor soluții contextuale denumite şabloane de proiectare (design patterns).

În Capitolul 9 vom introduce şabloanele arhitecturale (şabloane la nivel macro), exemplificând cu şablonul Model-View-Controller pentru arhitecturile orientate -obiect.

Autoarea

CAP. 1. INTRODUCERE: PROBLEME ŞI PRACTICI FRECVENTE ÎN INGINERIA SOFTWARE

Vom introduce, pentru început, câteva probleme frecvente în dezvoltarea programelor şi vom prezenta cele mai folosite practici de inginerie software (Tabel 1.1), împreună cu explicația motivației aplicării lor.

SIMPTOME	CAUZE	PRACTICI
Cerințe neîndeplinite	Cerințe incorecte	1.Dezvoltarea iterativă
Cerințe încâlcite	Comunicare ambiguă	2.Gestionarea cerințelor
Erori la integrarea modulelor	Arhitecturi fragile	3.Arhitecturi pe bază de componente
Dificil de menținut	Complexitate ridicată	
Erori descoperite târziu	Inconsistențe nedetectate	4.Modelare vizuală (UML)
Calitate slabă	Testare insuficientă	
Performanțe reduse	Evaluare subiectivă	5.Verificarea continuă a calității
Conflicte între dezvoltatori	Dezvoltarea în cascadă	
Probleme la asamblare şi livrare	Modificări necontrolate	6.Gestionarea modificărilor
	Insuficientă automatizare	

Tabel 1.1. Probleme în dezvoltarea software: simptome, cauze şi practici de eliminare

Practicile din ingineria software dirijează procesul de dezvoltare prin încercarea de a elimina cauzele problemelor.

Practica 1: dezvoltarea iterativă (vs. în cascadă)

Dezvoltarea iterativă presupune că funcționalitățile unui sistem se completează în versiuni succesive ale programului. Fiecare versiune este dezvoltată într-o perioadă fixată de timp, denumită iterație. La fiecare iterație se identifică, definesc şi analizează o parte din cerințele programului şi design-ul acestuia, iar implementarea şi testarea se concentrează pe acest set restrâns de cerințe.

Spre deosebire de dezvoltarea iterativă, dezvoltarea în cascadă (Figura 1.1) produce un singur produs livrabil. Problema fundamentală a acestei abordări este că amână riscurile în viitor, când va fi mult mai costisitor (uneori chiar imposibil) să reparăm erorile din fazele anterioare.

În cadrul dezvoltării iterative (Figura 1.2), primele iterații tratează riscurile cele mai mari, şi fiecare iterație produce un program executabil şi include faze de integrare şi testare.

Avantajele iterațiilor sunt următoarele:

• rezolvă riscurile majore înainte de a investi prea mult în proiect;

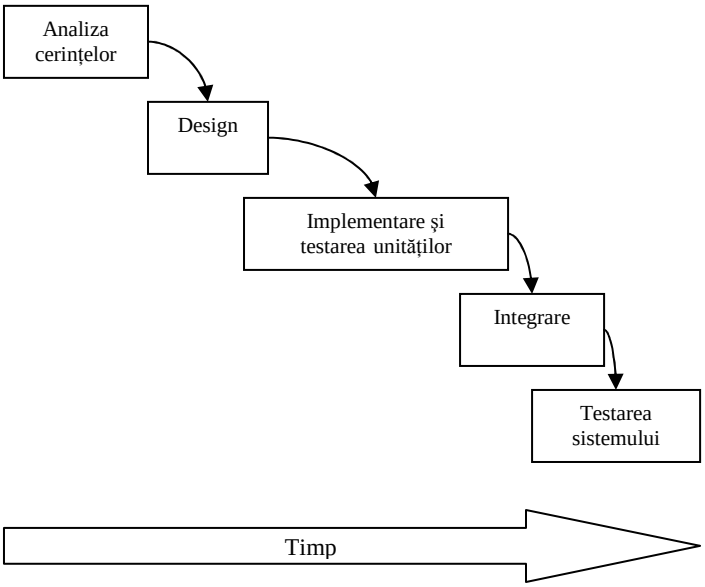
• permit feedback-ul obiectiv din timp;

• testarea şi integrarea sunt continue;

• focalizează proiectul pe obiective pe termen scurt, mai uşor de atins;

• permit realizarea unor implementări parțiale ale sistemului final complet.

Abordarea iterativă a fost creată special pentru a rezolva problemele abordării în cascadă. În loc să construim de la început tot sistemul, se creează un increment (adică o submulțime a funcționalității sistemului), apoi altul etc. Selectarea incrementelor se face în ordinea descrescătoare a riscului. Fiecare increment se construieşte pornind de la cazurile de utilizare relevante pentru riscul ales, şi care vor putea fi verificate obiectiv prin execuția unor cazuri de test.



1. Amână confirmarea rezolvării riscurilor critice
2. Conduce adesea la extensii majore, neprevăzute ale proiectului
3. Întârzie şi uneşte integrarea cu testarea

Figura 1.1.Dezvoltarea în cascadă

Dezvoltarea iterativă permite şi atingerea unor standarde superioare de calitate, întrucât caracteristicile sistemului legate de calitate devin tangibile mai devreme în procesul de dezvoltare, şi prin urmare pot fi corectate mai repede.

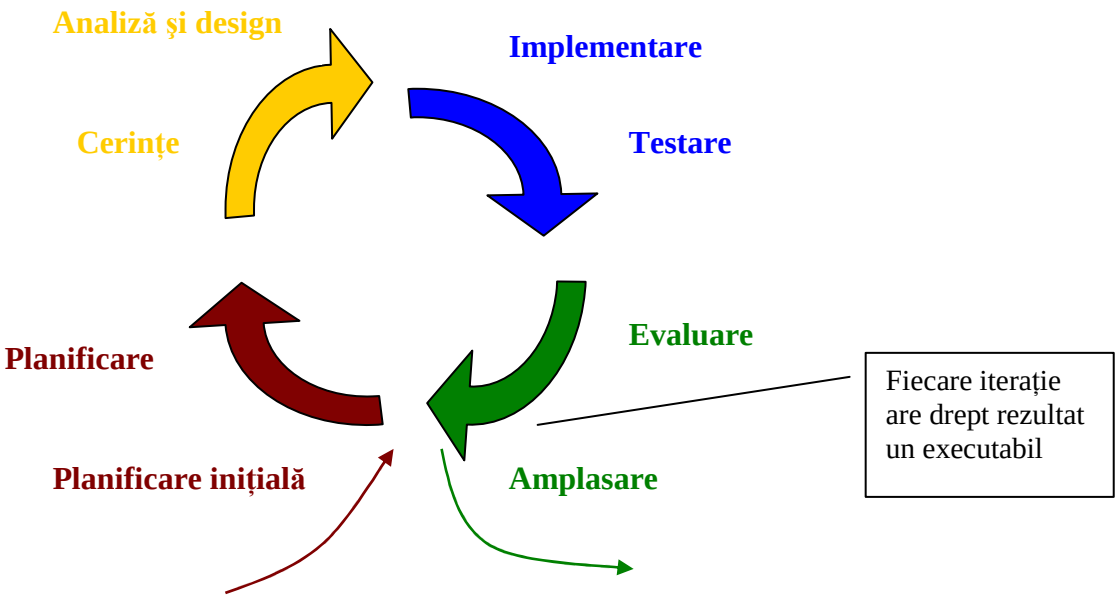


Figura 1.2. Dezvoltarea iterativă

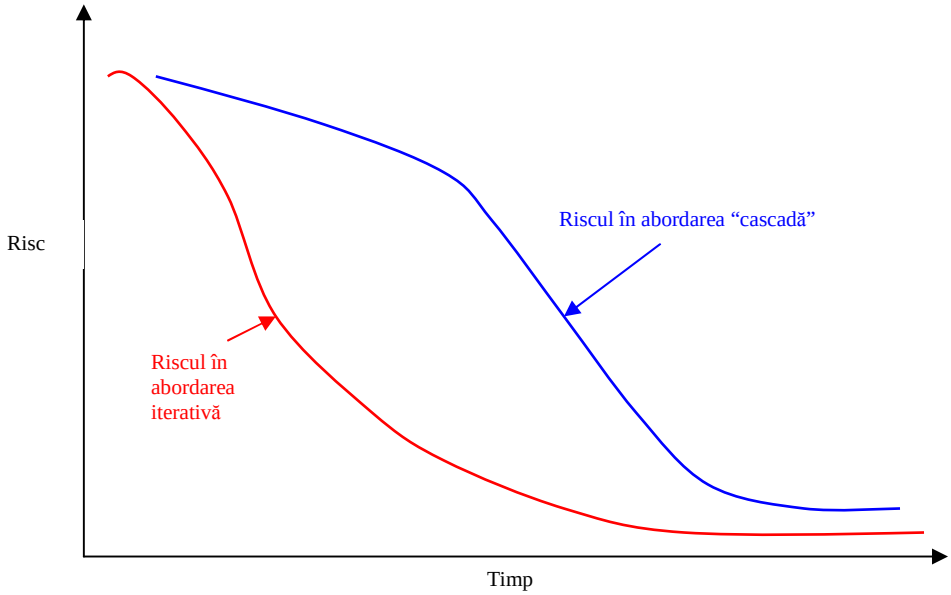


Figura 1.3. Reducerea riscului: dezvoltarea iterativă înlătură riscurile devreme

Practica 2: gestionarea cerințelor

Numărul de proiecte de dezvoltare software completate la termen şi fără depăşirea resurselor este de numai 16.2 %. Restul îl reprezintă proiectele întârziate şi cu bugetul depăşit (52.7%) sau proiectele anulate (31.1%). Aceste eşecuri sunt atribuite unui prost management al cerințelor, definirii greşite a cerințelor de la bun început sau gestionării defectuoase a cerințelor de-a lungul ciclului de dezvoltare [Chaos14].

Gestionarea cerințelor implică traducerea cererilor clienților într-un set de caracteristici şi necesități ale sistemului, detaliate apoi în specificarea cerințelor funcționale şi non-funcționale. Specificațiile detaliate se traduc într-un anumit design, o anumită documentație pentru utilizatori şi anumite teste.

Cerințele software sunt o intrare foarte importantă pentru testare. Se vor găsi întotdeauna probleme importante la granița între fiecare două secțiuni ale piramidei (Figura 1.4). Spre exemplu, sunt necesitățile reflectate în mod adecvat de caracteristici? Cerințele sunt corect reflectate în design? Cum arată un proces de testare pornind de la cerințe? Pentru a gestiona relația dintre cerințe şi testele derivate din acestea, se pot stabili relații de transparență („traceability") între aceste elemente.

Transparența codului (uşurința cu care putem urmări cauzele/ efectele unor modificări) ne ajută să realizăm următoarele:

- evaluarea impactului pe care îl va avea modificarea unei cerințe asupra proiectului;
- evaluarea impactului pe care îl va avea un test picat asupra cerințelor (ex., dacă testul pică, poate că cerința nu este satisfăcută);
- gestionarea domeniului de aplicabilitate al proiectului;
- verificarea satisfacerii integrale a cerințelor în implementare;
- verificarea faptului că aplicația nu face nimic în plus față de cerințe;
- gestionarea modificărilor (schimbărilor).

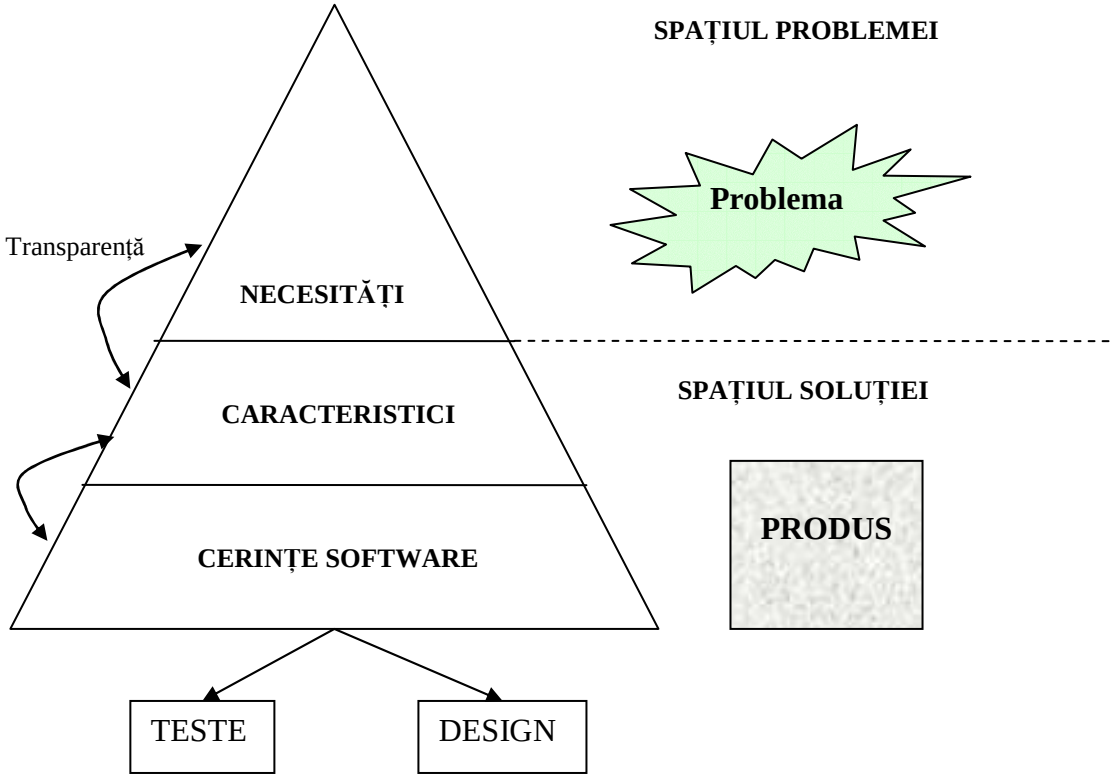


Figura 1.4.Gestionarea cerințelor

Gestionarea cerințelor : Concepte ale Cazurilor de Utilizare

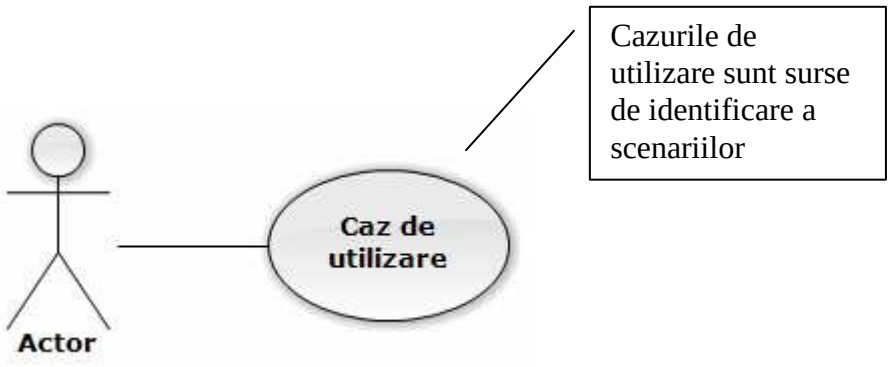


Figura 1.5.Actori şi cazuri de utilizare

Un actor reprezintă o persoană sau un alt sistem care interacționează cu sistemul analizat. Un Actor:

- nu este o parte a sistemului. Reprezintă un rol pe care un utilizator al sistemului îl va juca atunci cand va interacționa cu el;
- poate interschimba activ informație cu sistemul;
- poate fi un recipient pasiv de informație;
- poate fi un furnizor de informație;

• poate fi un om, o maşină, sau un alt sistem.

Un caz de utilizare defineşte o secvență de acțiuni realizate de sistem pentru a furniza o valoare observabilă unui actor. Un Caz de Utilizare:

• specifică un dialog între un actor şi sistem;
• este inițiat de un actor pentru a invoca o funcționalitate anume a sistemului;
• este o colecție de fluxuri de evenimente cu sens, relaționate;
• rezultatul său este o valoare observabilă.

Cazurile de utilizare (Use Cases) reprezintă o tehnică de definire a cerințelor, raportându-ne la utilizatorul final al sistemului. Ele au fost popularizate odată cu procesele de dezvoltare iterativă, deşi nu sunt specifice acestora (pot fi folosite şi în modelul cascadă).

Luate împreună, toate cazurile de utilizare reprezintă o vedere externă, de nivel înalt, asupra modalităților de utilizare a sistemului.

Practica 3: Arhitecturi construite din Componente

Arhitectura software este acel produs al procesului de dezvoltare care va răsplăti cel mai mult efortul investit, prin prisma calității, planificării şi costurilor [Bass98]. Proiectul AT&T de la SEI (The Software Engineering Institute) Architecture Tradeoff Analysis (ATA) a raportat o creştere a productivității de 10 % ca urmare a efortului de evaluare a arhitecturilor.

Definiție. *Arhitecturi rezistente bazate pe componente:*

1. Rezistent (adică îndeplineşte cerințele curente şi viitoare):
    a. Îmbunătățeşte extensibilitatea
    b. Facilitează refolosirea
    c. Încapsulează dependențele sistemului
2. Bazat pe componente
    a. Refoloseşte sau adaptează componente
    b. Selectează din componentele disponibile comercial
    c. Dezvoltă incremental soft-ul existent

Arhitectura este o parte a design-ului ce conține deciziile despre cum va fi construit sistemul, oprindu-se la abstracțiunile majore- adică elementele care vor avea o importantă influență pe termen lung asupra performanței sistemului şi asupra capacității sale de a evolua. Arhitectura este cel mai important aspect ce poate fi folosit pentru a controla sistemul într-o dezvoltare incrementală iterativă. Cea mai importantă proprietate a unei arhitecturi este rezistența, adică flexibilitatea în fața schimbării. Pentru a o atinge, arhitecții trebuie să anticipeze evoluția atât în domeniul problemei cât şi în tehnologiile de implementare, pentru a se

adapta cu uşurinţă la schimbări ale acestora. Tehnicile-cheie folosite sunt abstractizarea, încapsularea şi analiza şi design-ul orientat obiect, rezultatul fiind aplicaţii mai mentenabile şi mai extensibile.

Scopurile unei arhitecturi bazate pe Componente- (Bazele reutilizabilităţii):

- Refolosirea Componentelor
- Refolosirea Arhitecturii
- Baza pentru managementul proiectului
- Planificare
- Asignarea sarcinilor către personal
- Livrare
- Control intelectual
- Gestionarea complexităţii
- Menţinerea integrităţii

Componenta software - definiţii:

Definiţia Procesuală: Componenta este o parte înlocuibilă sistemului, non-trivială, aproape independentă, ce îndeplineşte o funcţie clară în contextul unei arhitecturi bine definite. O componentă furnizează şi se conformează realizării fizice a unui set de interfeţe.

Definiţia UML: Componenta este o parte fizică, înlocuibilă a unui sistem ce încapsulează implementarea, şi furnizează şi se conformează realizării unui set de interfeţe. O componentă reprezintă o parte fizică a implementării unui sistem, incluzând codul software (sursă, fişiere binare sau executabile) sau echivalentul acestora (cum ar fi script-urile sau fişierele de comenzi).

Structurarea sistemului pe componente se va reflecta şi în planificarea testării, şi în asignarea sarcinilor către echipă în cadrul proiectului. Componentele ajută, de asemenea, în controlul complexităţii întrucât ascund (sau încapsulează) detaliile ne-necesare, simplificând descrierea interacţiunii dintre componente la diferite nivele de detaliere (sau abstractizare), şi această simplificare este de folos şi în descoperirea testelor utile.

Practica 4: Modelarea Vizuală (UML)

Un model este o simplificare a realităţii ce furnizează o descriere completă a sistemului dintr-o anumită perspectivă, cu scopul de a înţelege mai bine sistemul modelat (complex).

Avantajele Modelarii Vizuale

- Ajută la controlul complexităţii
    - Surprinde atât structura cât şi comportamentul;
    - Arată cum se integrează elementele sistemului;

    - Ascunde/ expune detaliile după cum este convenabil;
- Menține consistența design –implementare;
- Promovează comunicarea neambiguă
    - UML este un limbaj unic pentru toți practicienii.

Modelarea ajută echipa să vizualizeze, să specifice, să construiască şi să documenteze structura şi comportamentul arhitecturii unui sistem. Mai mult, limbajul vizual UML sprijină menținerea consistenței artefactelor sistemului: cerințe, design, implementare şi teste, şi implicit ajută în controlarea complexității.

Se pot folosi diferite tehnici pentru a verifica un model înainte de scrierea codului aferent. UML-ul însuşi furnizează reguli de stabilire a faptului dacă un model este “bine format”, şi există programe speciale de depistare a anomaliilor, ce pot fi extinse cu reguli definite de utilizator. De asemenea, există instrumente de furnizare automată a unor teste plecând de la modelul UML. În plus, modelarea vizuală în UML uşurează analiza arhitecturii prin prisma portabilității, procesării paralele şi flexibilității.

Tehnicile de forward şi reverse engineering permit ca modificările din model să se reflecte automat în codul aplicației, şi viceversa, ceea ce este foarte important în cadrul unui proces iterativ care aduce astfel de modificări la fiecare iterație.

O viziune grafică de ansamblu asupra sistemului se obține cu ajutorul unei diagrame de cazuri de utilizare globale (ce include toate cazurile de utilizare, actorii şi relațiile dintre aceste elemente necesare pentru a descrie sistemul aflat în construcție). Un actor este un element extern sistemului, şi care are o interfață cu sistemul, cum ar fi utilizatorii finali. Un caz de utilizare modelează un dialog între actori şi sistem, fiind inițiat de un actor pentru a invoca o anumită funcționalitate a sistemului (Figura1.6.)- săgeata indicând direcția transmiterii mesajului în interacțiune. Cazul trebuie să descrie un flux de evenimente complet şi cu sens unitar. Împreună, cazurile de utilizare constituie toate modalitățile de utilizare a sistemului.

Practica 5: Verificarea Continuă a Calității

Noțiunea de calitate acceptă mai multe definiții. Totuşi, este corect să afirmăm că atingerea calității implică mai mult decât „satisfacerea cerințelor" sau crearea unui produs care îndeplineşte necesitățile şi aşteptările utilizatorilor. Trebuie găsite şi măsurile, şi criteriile care să verifice atingerea calității, şi trebuie implementat un proces care să asigure obținerea unui produs de un anumit nivel calitativ.

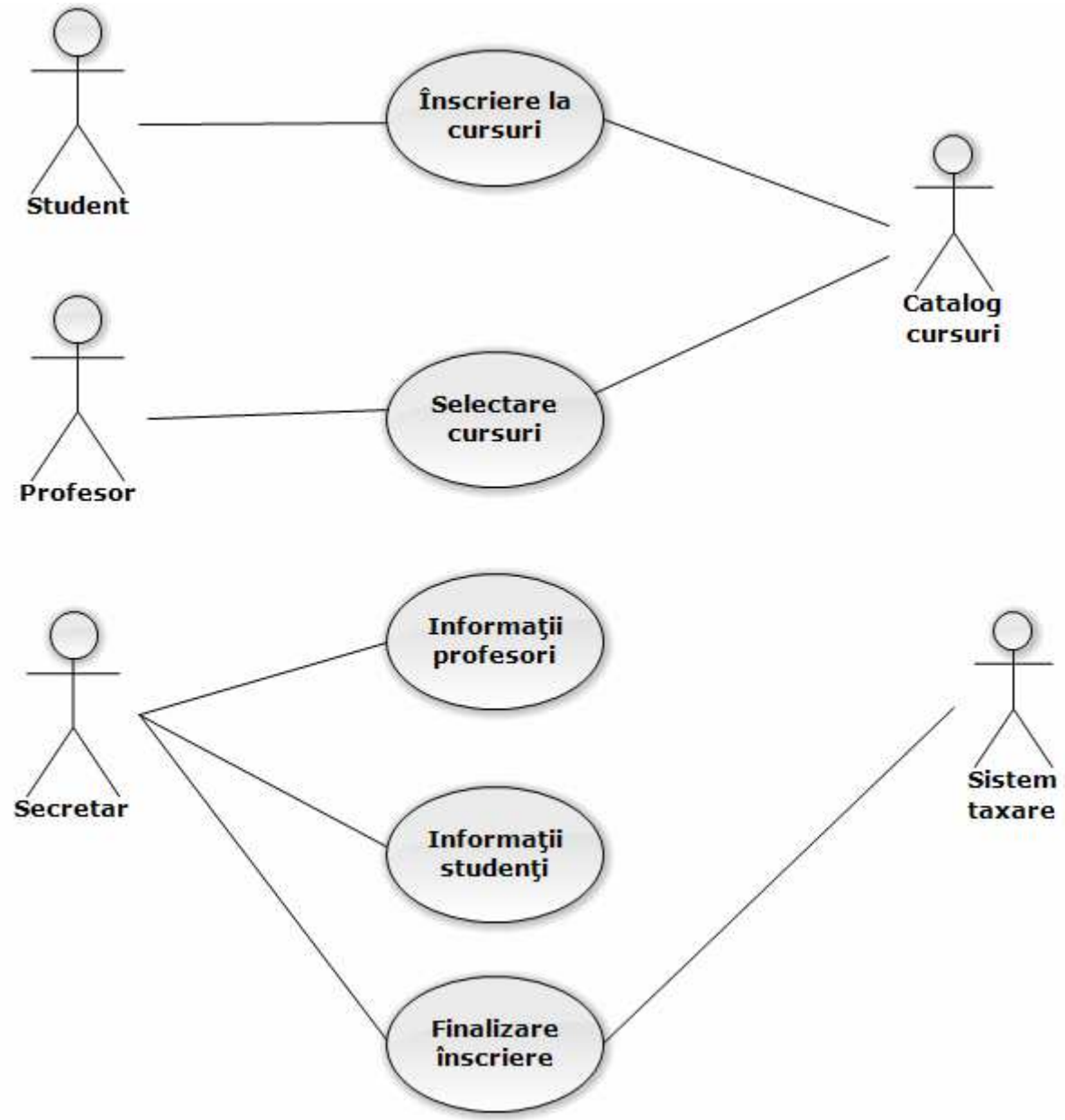


Figura 1.6.Diagrama de cazuri de utilizare pentru un sistem de înscriere la cursuri într-o universitate

Testarea este un aspect al procesului de Asigurare a Calității, şi deşi răspunde de 30% până la 50% din costurile dezvoltării software în multe organizații, percepția generală este că programele nu sunt suficient testate înaintea livrării. Această observație îşi are rădăcina în două observații interesante. Prima este că testarea software este foarte dificilă: există apropape o infinitate de moduri diferite în care se poate comporta un program. A doua este că testarea se face deseori fără o metodologie clară şi fără instrumente ajutătoare adecvate. Chiar dacă nu putem realiza o testare completă, putem totuşi folosi instrumente care să îmbunătățească productivitatea şi eficiența testării, şi putem folosi o metodologie potrivită cu contextul proiectului.

Verificați continuu calitatea – la fiecare iterație

În dezvoltarea software tradițională există tendința de a amâna anumite tipuri de evaluări -cum ar fi testarea black-box– până târziu în ciclul de dezvoltare. În cazul testarii black-box, amânarea acestor teste va întârzia descoperirea eventualelor probleme târziu în ciclul de dezvoltare, când corectarea este mult mai costisitoare. În dezvoltarea iterativă, activitățile de evaluare sunt o parte a efortului din cadrul fiecărei iterații, fiind necesare în verificarea atingerii scopului iterației respective.

UML se poate folosi pentru producerea diferitor modele, ce reprezintă varii perspective asupra programului, pe parcursul evoluției sale. Fiecare model este dezvoltat incremental de-a lungul mai multor iterații.

- Modelul Cazurilor de Utilizare reprezintă valoarea sistemului către utilizatorii externi din mediul său, şi descrie “serviciile externe” furnizate de sistem;

- Modelul de Design descrie modul în care sistemul va realiza serviciile descrise de cazurile de utilizare, servind drept model conceptual (abstractizare) pentru modelul implementării şi codul sursă asociat;

- Modelul de Implementare reprezintă elementele software fizice şi subsistemele de implementare în care acestea sunt conținute.

Evaluarea implică activități de Verificare şi de Validare: verificarea construcției corecte a programului, şi validarea faptului că este construit programul potrivit. Această distincție se referă la evaluarea adecvării procesului de construcție (verificare) şi la adecvarea produsului final livrat către client (validare).

Practica 6: Controlul modificărilor

În aproape toate proiectele semnificative sunt necesare modificări. Prin urmare, este necesar şi un control a când, cum şi cine introduce aceste modificări, şi ele trebuie sincronizate între membrii echipei şi, eventual, între locațiile diferite.

Ce dorim să controlăm?

- Modificări necesare dezvoltării iterative
    - Spațiu de lucru securizat pentru fiecare membru al echipei
    - Posibilitatea dezvoltării paralele
- Gestionarea automatizată a etapelor integrare/construcție

Asigurarea unui spațiu de lucru securizat pentru fiecare membru din echipa proiectului izolează de modificările realizate în celelalte spații de lucru şi furnizează control asupra tuturor artefactelor software: modele, cod, documentații, teste etc.

Atunci când există mai mulți lucrători, organizați în echipe diferite, posibil aflate la locații diferite, lucrând împreună pe mai multe iterații/ produse/ platforme se poate degenera uşor în haos. Trei probleme uzuale ce pot apare şi care trebuie gestionate sunt:

- Actualizarile simultane: când doi sau mai mulți membri modifică separat acelaşi artefact, ultimul care acționează distruge munca celorlalți

- Notificări limitate: unele persoane din echipă nu sunt notificate de anumite modificări din artefactele comune

- Versiuni multiple: dacă se lucrează cu multiple versiuni, fiecare aflată în altă etapă de dezvoltare, identificarea unei probleme în una dintre versiuni trebuie transmisă tuturor celorlalte versiuni

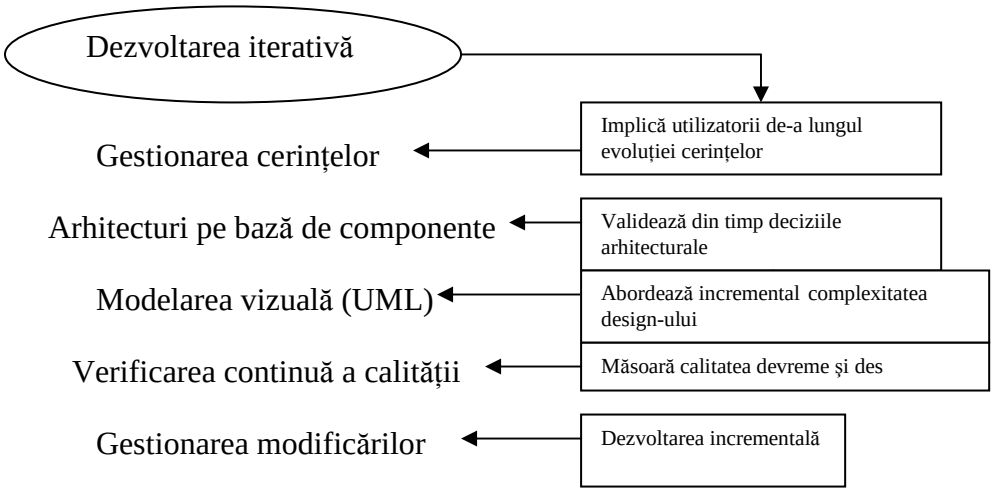


Figura 1.7. Practicile de inginerie software se sprijină reciproc

În cazul celor 6 practici de inginerie software, întregul este mult mai mult decât suma părților, întrucât fiecare practică poate consolida/ sprijini alte practici. În Figura 1.7. se observă cum dezvoltarea iterativă influențează celelalte cinci practici, reversul fiind de asemenea valabil: fiecare din cele 5 practici sprijină dezvoltarea iterativă. Spre exemplu, dezvoltarea iterativă fără un control adecvat al cerințelor nu reuşeşte să conveargă către o soluție: cerințele se schimbă după dorință, utilizatorii nu reuşesc să ajungă la un consens, iterațiile nu mai ajung la final. Controlând însă cerințele, modificările acestora devin vizibile şi impactul lor asupra procesului de dezvoltare este evaluat înainte de a le accepta (şi astfel suntem siguri că vom converge către un set stabil de cerințe). În mod similar, fiecare pereche de practici se sprijină reciproc şi deşi este posibil să le folosim independent, beneficiile maxime se obțin din utilizarea lor combinată.

Exerciții.

1. Considerați o aplicație ce va fi folosită pentru a programa sălile de conferință dintr-o clădire. Aplicația are o IUG ce permite utilizatorului să indice data, ora, şi durata (în incremente de 15 minute). Apoi afişează o listă cu sălile de conferință disponibile la ora respectivă şi permite rezervarea unei săli.

Sistemul permite şi anularea unei rezervări. Proiectul trebuie dezvoltat incremental –adică, în etape crescătoare de funcționalitate. Considerați cele două planuri din Tabelul 1.2. Care va reuşi mai bine? Ce testare se poate face la sfârşitul fiecărui increment?

Plan A	**Plan B**
Increment 1: Crearea interfeței grafice; Increment 2: Crearea subsistemului de stocare a datelor; Increment 3: Dezvoltarea subsistemului aplicație (rezervările).	Increment 1: Dezvoltarea capacității de a introduce data, ora, durata, şi afişarea disponibilității sălilor; Increment 2: Dezvoltarea capacității de a rezerva o cameră; Increment 3: Dezvoltarea capacității de a anula o rezervare.

Tabel 1.2.

2. Gândiți arhitectura jocului Brickles (o paletă se mişcă stânga-dreapta cu ajutorul mouse-ului, şi loveşte o minge care trebuie să „spargă" cât mai multe cărămizi – la atingerea mingii cărămizile se şterg- Figura 1.8).

3. Imaginați un plan incremental de dezvoltare pentru jocul **Hungry Hippau**:

Un hipopotam se deplasează în cele patru direcții principale folosind săgețile; scopul său este de a mânca fructele aduse de apă şi de a evita buştenii. Hipopotamul se poate afla în două stări distincte:

I. Este deasupra apei: poate mânca fructe, dar poate fi şi lovit de buşteni (Figura 1.9).

Figura 1.8.Jocul Brickles

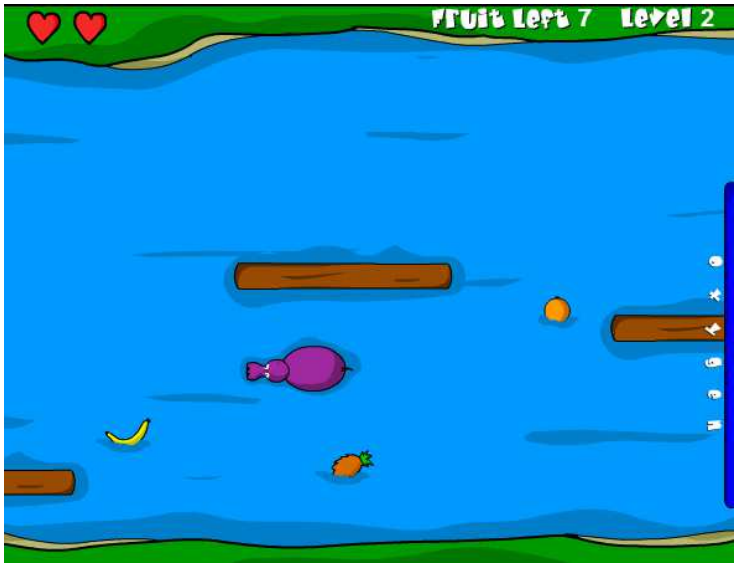


Figura 1.9. Starea I

II. La apăsarea tastei *Space* hipopotamul intră sub apă pentru a evita buştenii (dar nu mai poate mânca) (Figura 1.10).

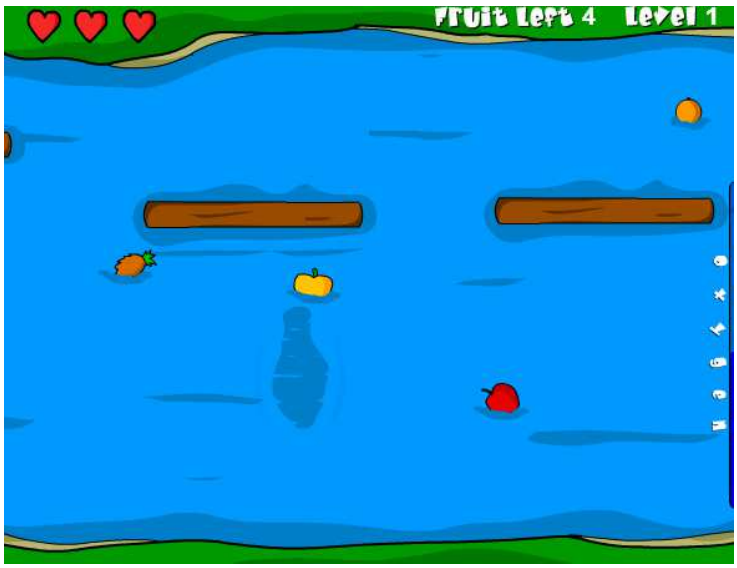


Figura 1.10. Starea II

CAP. 2. CRITERII DE CALITATE ALE DESIGN-ULUI SOFTWARE (MODULARIZAREA ŞI DESCOMPUNEREA)

> *Motto: "Un design de calitate echilibrează compromisurile necesare pentru **minimizarea costului total** al sistemului pe întreaga sa durată de viață. De asemena, evită caracteristicile ce duc la consecințe negative".* [Your91]

2.1. Introducere

Vom prezenta în acest capitol **criteriile şi principiile** care ne ajută să definim calitatea unui program, în general. Vom vedea în continuare care sunt scopurile design-ului, conceptele –cheie şi principiile, criteriile calității design-ului şi principiile şi regulile design-ului de calitate (importanța coeziunii, cuplajului, ascunderii informației, controlului dependențelor).

Scopurile design-ului. Un prim scop al design-ului este descompunerea sistemului în componente (descompunere ce ajută la identificarea şi caracterizarea arhitecturii software-ului). Al doilea scop este descrierea funcționalității fiecărei componente (într-o manieră formală sau informală). Cel de-al treilea scop constă în determinarea relațiilor dintre componente, ceea ce presupune identificarea dependențelor dintre componente şi determinarea mecanismelor de comunicare intra-componentă. În fine, cel de-al patrulea scop este specificarea interfețelor componentelor: interfețele trebuie să fie bine definite, pentru a uşura testarea fiecărei componente şi comunicarea în cadrul unei echipe de dezvoltare.

Un design de calitate minimizează costul unui sistem pe întreaga sa durată de viață. Aceasta presupune că deşi în faza de dezvoltare s-a investit mai mult (timp, resurse umane etc.) pentru a respecta criteriile de calitate, aceste investiții se vor acoperi (şi chiar vor aduce o economie) printr-un cost mai scăzut al mentenanței. În practică, nu există un design corect unic, programatorul fiind cel care decide în funcție de propria sa experiență şi de contextul aplicației. Totuşi, sunt necesare atât nişte criterii clare de evaluare a design-ului, cât şi nişte principii şi reguli pentru crearea de design-uri de calitate.

Elementele-cheie ale design-ului sunt descompunerea/ compunerea şi modularizarea, scopul principal al acestora fiind controlarea complexității sistemului software prin optimizarea factorilor de calitate şi prin facilitarea reutilizării sistematice.

2.2. Descompunerea

Vom raspunde mai întâi la întrebările de ce descompunem un sistem şi cum procedăm practic, vom continua cu definiția unei componente şi vom încheia cu principiile de descompunere ale lui Schmidt [Schm02].

Un sistem software este descompus pentru a reduce complexitatea prin împărţirea problemelor mari în componente mai mici, după principiul “divide et impera”. Practic, se aplică paşii următori:

- Selectează o parte a problemei (inițial, întreaga problemă);
- Determină componentele acestei părți folosind o paradigmă de design (funcțională, orientată-obiect, etc.);
- Descrie interacțiunile între componente;
- Repetă 1-3 până la satisfacerea unui anumit criteriu de terminare.

Definiție. O *componentă* este:

- o entitate software ce înglobează reprezentarea unei abstracțiuni;
- o metodă de a ascunde cel puțin o decizie de design;
- o sarcină de lucru (pentru un programator/ grup de programatori);
- o unitate de cod care:
  - Are un nume;
  - Are granițe identificabile;
  - Poate fi (re-)folosită de alte componente;
  - Încapsulează date;
  - Ascunde detaliile ne-necesare;
  - Poate fi compilată separat (eventual).

Definiție. *Interfața* unei componente cuprinde mai multe secțiuni:

- Importuri: servicii solicitate altor componente;
- Exporturi: servicii oferite altor componente;
- Controlul accesului (private/protected/public).

Principiile Schmidt de descompunere

- Nu creați componente care să corespundă unor paşi de execuție (deciziile de design transcend timpul execuției)[Parn72];
- Descompuneți astfel încât să limitați efectele deciziilor de design (tot ce influențează sistemul va fi costisitor de modificat);
- Componentele trebuie specificate prin toată informația necesară pentru utilizarea componentei respective (şi nimic în plus).

2.3. Modularitatea

Vom începe cu prezentarea definiției şi avantajelor modularității, vom introduce apoi criteriile de modularizare ale lui Meyer [Meye94] şi vom încheia cu principiile (regulile) care duc la respectarea acestor criterii.

Un sistem este modular dacă este structurat în abstracțiuni identificabile numite *componente*. Componentele trebuie să aibă interfețe abstracte bine specificate, şi să fie caracterizate prin coeziune ridicată şi cuplaj slab. Aşa cum menționa Bertrand Meyer, o metodă de construire software este modulară dacă ajută designerii să producă sisteme software alcătuite din *elemente autonome*, conectate printr-o *structură simplă* şi *coerentă*.

Avantajele modularității

Modularitatea facilitează factorii de calitate software, cum ar fi:

- Extensibilitatea (dată de interfețele abstracte, bine definite)
- Reutilizabilitatea (cuplaj slab, coeziune ridicată)
- Portabilitatea (independența de maşină)

Modularitatea este importantă pentru un design de calitate deoarece:

- Sporeşte gradul de separație a preocupărilor (programatorilor) într-un proiect
- Reduce complexitatea sistemului prin arhitecturi software descentralizate
- Sporeşte scalabilitatea întrucât suportă o dezvoltare concurentă şi independentă de către mai multe persoane

Criteriile Meyer de evaluare a modularității [Meye94]

A. Decompozabilitatea. Decompozabilitatea presupune descompunerea problemei în sub-probleme mai mici ce pot fi rezolvate separat (*Sunt componentele mari descompuse în componente mai mici?*)

- Exemplu: design top-down
- Contra-exemplu: modul de inițializare care inițializează toate datele pentru toate componentele

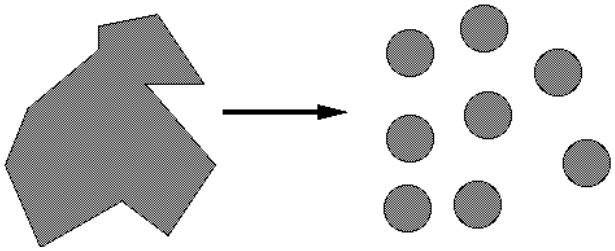


Figura 2.1. Decompozabilitatea

B. Compozabilitatea. Un sistem este compozabil dacă modulele se pot combina liber pentru a produce noi sisteme (*Sunt componentele mai mari compuse din componente mai mici?*)

- Componente- în medii diferite;
- Exemplu: bibliotecile de funcții matematice;
- Contra-exemplu: utilizarea de biblioteci non-POSIX.

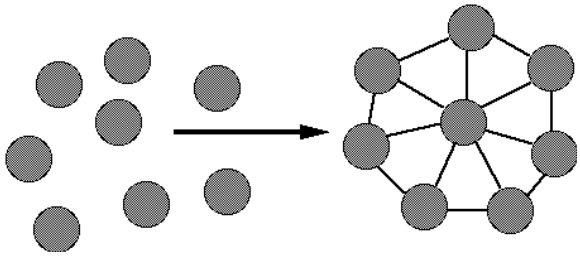


Figura 2.2. Compozabilitatea

C. Claritatea. Un program este clar dacă modulele individuale sunt inteligibile unui cititor uman (*Sunt componentele, luate separat, inteligibile?*)

- Contra-exemplu: dependențele secvențiale (A|B|C).

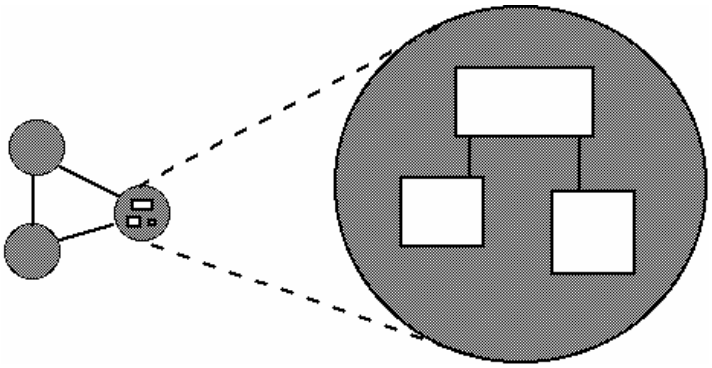


Figura 2.3. Claritatea

D. Continuitatea. Un sistem are calitatea de continuitate dacă modificări mici în specificație vor rezulta în modificări în puține module şi nu vor afecta arhitectura (*Modificări mici ale specificației afectează un număr limitat de componente, în mod limitat?).*

- Exemplu: constantele simbolice;
- Contra-exemplu: tablourile statice

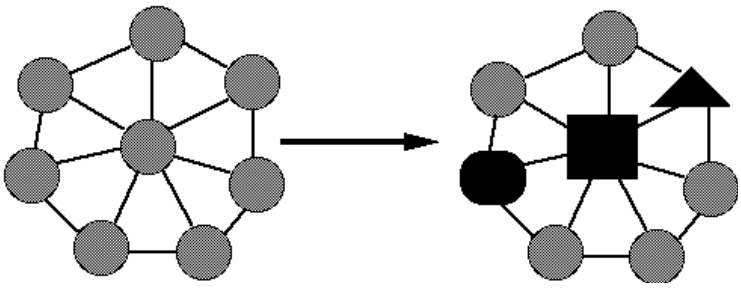


Figura 2.4. Continuitatea

E. Protecția. Un sistem este protejat dacă efectele unei condiții anormale la execuție sunt limitate doar la câteva module (*Efectele anomaliilor din timpul execuției sunt limitate la un număr mic de componente relaționate?)*

- Exemplu: validarea intrărilor la sursă;
- Contra-exemplu: excepții haotice

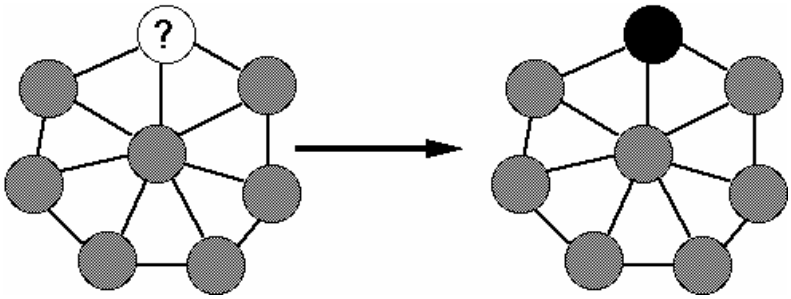


Figura 2.5. Protecția

Regulile modularității ale lui Meyer

A. Corespondența directă. Această regulă implică existența unei relații consistente între modelul problemei şi structura soluției. Structura soluției trebuie să rămână compatibilă cu structura domeniului problemei modelate, astfel încât între cele două să existe o corespondență clară.

Principiul va avea impact asupra continuității, întrucât va fi mai uşor de conştientizat şi de limitat impactul unei eventuale schimbări, dar şi asupra decompozabilității (descompunerea domeniului problemei modelate este un bun punct de plecare pentru descompunerea programului).

B. Interfețe puține. Pentru a respecta această regulă trebuie ca fiecare modul (componentă) să comunice cu un număr minim de alte module (componente). Acest principiu afectează continuitatea, protecția, claritatea şi compozabilitatea.

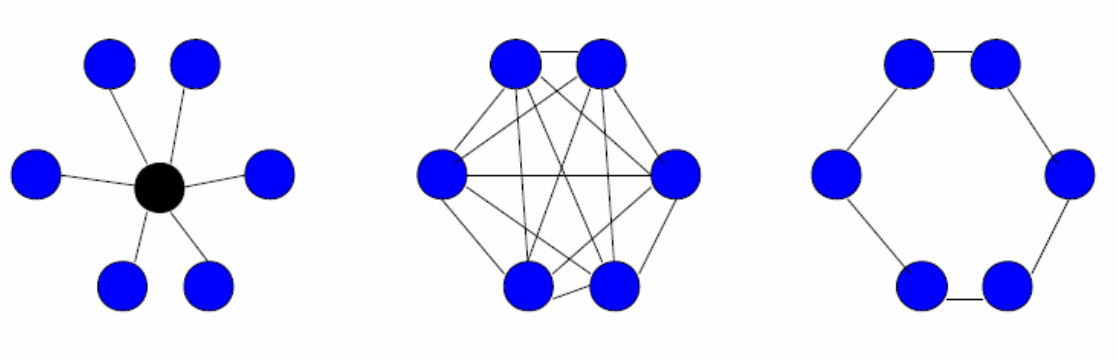


Figura 2.6. Mai bine n-1 relații de comunicare decât n(n-1)/2

C. Interfețe mici. Dacă două module (componente) comunică, ele trebuie să schimbe cât mai puțină informație. Acest aspect va influența pozitiv continuitatea şi protecția sistemului.

D. Interfețe explicite. Când două module (componente) A şi B comunică, aceasta trebuie să fie evident din textul lui A sau B sau din al ambelor. Impactul va fi atât asupra decompozabilității şi compozabilității, dar şi asupra continuității şi clarității.

- Contra-exemplu: problema cuplajului indirect (partajarea datelor):

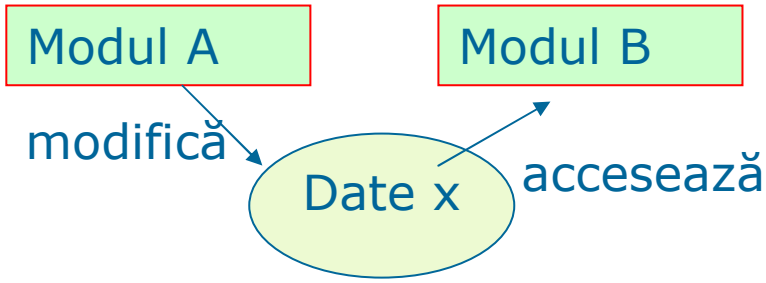


Figura 2.7. Cuplaj indirect

E. Ascunderea informației. Motivația din spatele ascunderii informației este că deciziile de design ce pot fi subiectul unei schimbări trebuie ascunse în spatele unor interfețe abstracte (i.e. componente). Ascunderea informației este o metodă de a spori/întări abstractizarea. Pentru a respecta acest principiu este necesar:

- Componentele trebuie să comunice doar prin interfețe bine-definite;
- Fiecare componentă trebuie să fie specificată de minimul de informație necesar (vezi principiile de descompunere Schmidt);
- Dacă detaliile interne se modifică, componentele –client trebuie să fie minimal afectate (astfel încât nici măcar recompilarea să nu fie necesară).

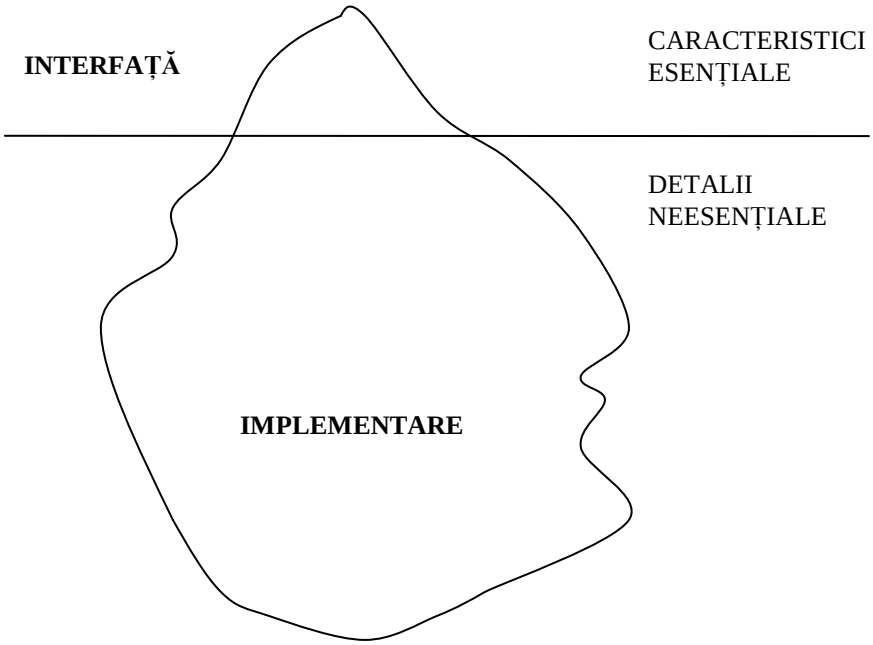


Figura 2.8. Abstractizarea vs. Ascunderea informației

CAP. 3. UML (UNIFIED MODELING LANGUAGE)

3.1. Introducere UML

UML (Unified Modeling Language- Limbaj de Modelare Unificat) este un limbaj standardizat, tehnic de modelare şi de comunicare între dezvoltatorii de software, care descrie grafic cerințele, arhitectura şi design-ul unui program sau sistem. Instrumentul este utilizat în comunicarea dintre dezvoltatori, sau dezvoltator-client, având abilitatea de a genera automat părți din sistem. El poate comunica eficient şi precis informații legate de toate aspectele sistemului: cerințe, arhitectură, design, şabloane de proiectare, implementare.

UML este compus dintr-o mulțime de **diagrame**, dezvoltate pentru facilitarea task-urilor de specificare, vizualizare, design arhitectură, construcție, simulare şi testare şi documentare.

Utilizat judicios, UML *nu* este un consum de timp, ci scade din cheltuielile şi timpul total de dezvoltare, deoarece scade costul comunicării şi creşte înțelegerea, productivitatea şi calitatea. El poate fi folosit atât de creatorii de modele şi de designeri în explorarea diferitelor soluții, cât şi de către implementatorii care construiesc aceste soluții.

3.2. Tipuri de diagrame

Diferitele tipuri de diagrame reprezintă modele ale programului. Cele mai folosite diagrame se pot clasifica astfel:

1. Diagrame **statice (sau structurale):** Arată trăsăturile statice ale sistemului.

    - **Diagrama de clase** –vizualizarea elementelor de analiză şi design, a claselor de implementare şi a relațiilor dintre clase

2. Diagrame **dinamice (funcționale sau de comportament):** Arată evoluția sistemului în timp, detaliind comportamentul şi algoritmii (diagrama de cazuri de utilizare, diagramele de interacțiuni, diagrama de stare, diagrama de activități).

    - **Diagrama cazurilor de utilizare-** vizualizarea serviciilor pe care **actorii** le pot solicita de la sistem;

    - **Diagrama de secvențe:** schimbul de mesaje într-un grup de obiecte şi ordinea mesajelor;

- **Diagrama de colaborare**: schimbul de mesaje într-un grup de obiecte şi relațiile dintre obiecte;

- **Diagrama de stare-** vizualizează ciclul de viață al unui obiect particular, sau secvențele prin care trebuie să treacă un obiect/ secvențele care trebuie suportate de o interfață;

- **Diagrama de activități-** vizualizează fluxul de date/ fluxul de control al comportamentului; fluxul de lucru dintre obiectele ce cooperează.

3.3. Construcția modelului

În construirea modelului folosind UML ar trebui ținut cont de următorii paşi:

- **Cine foloseşte sistemul?** Actori- (utilizatori ai sistemului), şi cazurile de utilizare asociate (scopurile sistemului).

- **Din ce este alcătuit sistemul?** Diagrama de clase (structura logică), diagrama de componente (structura fizică).

- **Unde sunt componentele localizate în sistem?** Unde se vor stoca/ executa diferitele componente (diagrame de implementare/ amplasare).

- **Când au loc evenimente importante în sistem?** La ce reacționează obiectele pentru a-şi îndeplini sarcinile (diagrame de stare şi de interacțiune).

- **De ce face acest sistem ceea ce face?** Identificați scopurile utilizatorilor sistemului şi reprezentați-le în cazurile de utilizare.

- **Cum va funcționa sistemul?** Părțile componente şi interacțiunile la nivel detaliat (diagramele de comunicare).

3.4. Tipuri de diagrame UML

3.4.1.Diagrama de clase

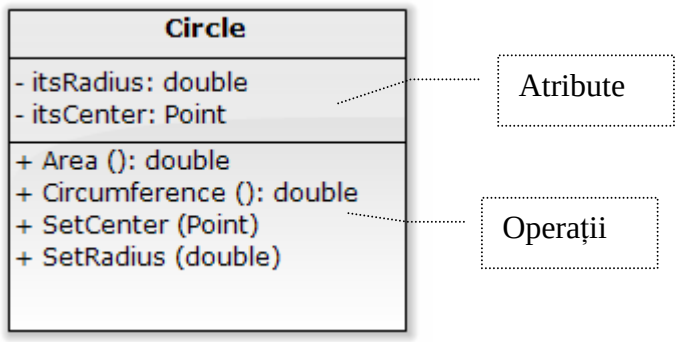


Figura 3.1. Clase în UML

O clasă se reprezintă ca un dreptunghi cu trei secțiuni: numele clasei, atribute şi operații (Figura 3.1). Specificatorii de acces, atât pentru atribute cât şi pentru operații sunt:

- + public
- - private
- # protected

O tehnică utilizată frecvent de alegere a obiectelor şi a claselor este *Sublinierea Substantivelor*:

Pasul 1. Se începe cu descrierea sistemului (sau a comportamentului său)- actori şi cazuri de utilizare

Pasul 2. Se examinează fiecare substantiv din descriere şi se verifică dacă satisface următoarele criterii:

- Este un lucru/ o familie de lucruri;
- Este o parte a problemei de rezolvat;
- Nu aparține detaliilor de implementare;
- Nu este un eveniment/ o apariție;
- Nu este o proprietate a unui lucru.

Pasul 3. Se creează o listă (generoasă) de substantive; se identifică apoi clasa pentru obiectele asociate substantivelor (substantivele ce nu se califică drept obiecte pot fi atribute, stări, operații, evenimente ş.a. ce pot fi ulterior utile).

Există cinci tipuri de relații între clase:

- Relația de compunere;

- Relația de moştenire;
- Relația de agregare;
- Relația de asociere;
- Relația de dependență.

A. Relația de compunere

Relația de compunere este o formă puternică de conținere sau agregare- o relație de tip parte-întreg. Compunerea este mai mult decât agregare, în sensul că durata de viață a părții depinde de întreg (în exemplul din Figura 3.2, dacă Circle este distrus, şi Point se va distruge).

Exemplu. Circle este întregul, Point este o parte a lui Circle: fiecare instanță a clasei Circle conține o instanță de tip Point. În UML, în lipsa săgeții, relațiile se presupun bidirecționale: dacă în Figura 3.2 nu ar fi existat săgeată atunci Point ştia de Circle (la nivel de cod: #include “circle.h” în point.h).

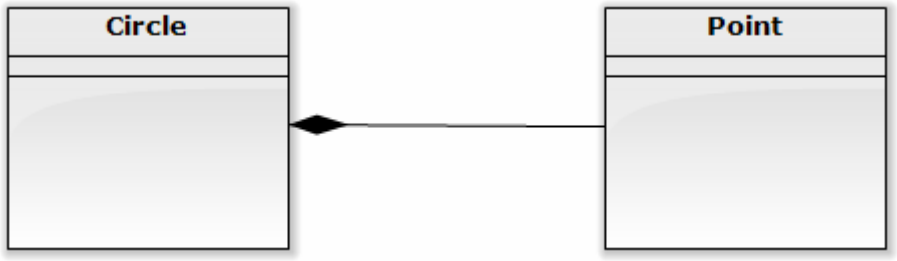


Figura 3.2. Relația de compunere

```
class Circle
{
public:
void SetCenter(const Point&);
void SetRadius(double);
double Area() const;
double Circumference() const;
private:
double itsRadius;
Point itsCenter;
};
```

…

B. Relația de moștenire

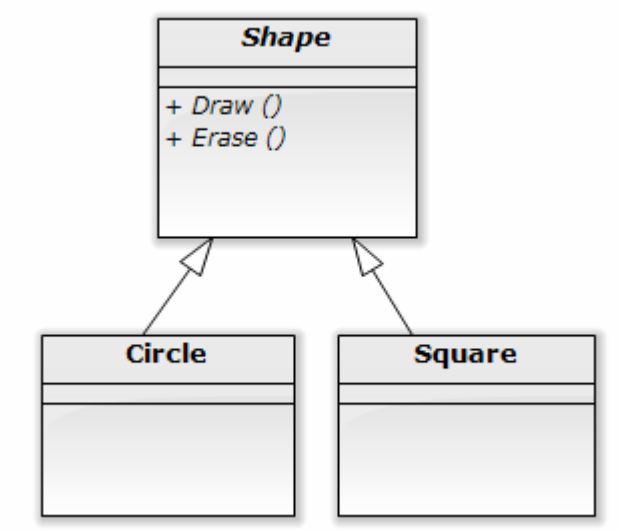


Figura 3.3. Moştenirea

C. Agregarea

În agregare, clasa agregat este într-un anumit fel "întregul", clasa conținută este într-un anumit fel o "parte" (Figura 3.4).

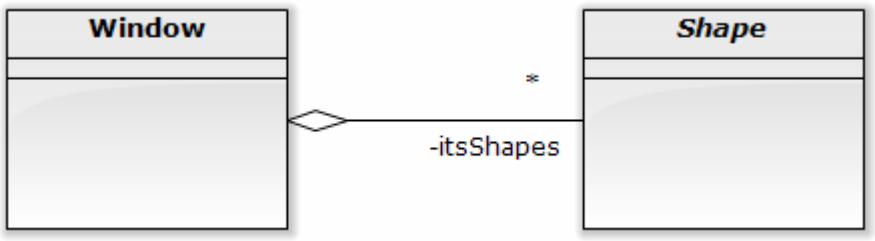


Figura 3.4. Agregarea

Exemplu. În Figura 3.4, clasa Window conține mai multe instanțe de tip Shape. Capetele unei asocieri se numesc "roluri", iar multiplicitatea "*" semnifică o colecție de elemente (în Figura 3.4- vectorul *itsShapes*)

```
class Window
{
    public:
    //...
    private:
    vector<Shape*> itsShapes;};
```

D. Relația de asociere

Asocierea se implementează de obicei cu ajutorul unui pointer. Diferența între agregare şi asociere este că agregarea denotă relații de tip parte/întreg, spre deosebire de asocieri. Implementarea este identică (în practică se poate ignora agregarea).

E. Relația de dependență

Relația de dependență este o relație foarte slabă: una dintre clase este tip al unui parametru folosit de una dintre metodele celeilalte clase.

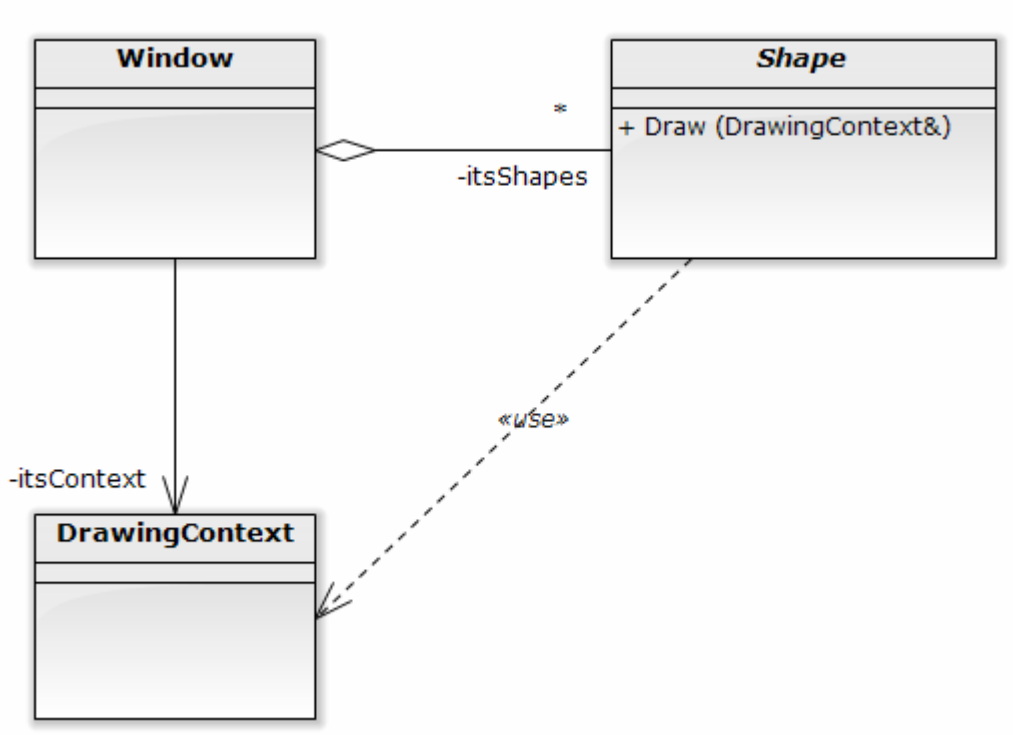


Figura 3.5. Dependența

3.4.2. Diagrama cazurilor de utilizare (use-case)

Cazurile de utilizare se folosesc în faza de analiză a unui proiect pentru identificarea şi partiționarea funcționalității sistemului. Acest tip de diagramă oferă o descriere funcțională complet separată de design-ul software, şi separă sistemul în *actori* şi *cazuri de utilizare*.

Actorii

Actorii reprezintă *roluri* jucate de utilizatorii sistemului: utilizatori umani, alte calculatoare, componente hardware sau alte sisteme software. Criteriile de definire a actorilor sunt:

- sunt părți externe sistemului partiționat,
- care *furnizează stimuli* acelei părți a sistemului de care sunt ataşați şi
- *primesc* anumite *ieşiri* de la ea

Cazurile de utilizare

Cazurile de utilizare descriu (textual) ***comportamentul*** sistemului la primirea unui stimul particular din partea unui actor şi ***ieşirile*** rezultate (utilizarea sistemului de către un actor pentru a îndeplini o sarcină). Ele descriu ***cerințele*** asupra sistemului software. Textul descrierii cuprinde şi ***problemele*** ce pot apare în timpul comportamentului specificat şi ce ***acțiuni de remediere*** a problemei execută sistemul. Cazurile de utilizare sunt organizate ierarhic folosind două tipuri de relații: "**uses**" ("**include**") şi "**extends**"**.**

Scenarii

În contextul unui caz de utilizare se pot defini mai multe **scenarii.** Un scenariu este o instanțiere a unui caz de utilizare (o aplicare a acestuia într-o situație particulară, descrierea unei interacțiuni particulare). Scenariile implică obiecte cu valori ale atributelor relevante, spre deosebire de cazurile de utilizare, care implică obiecte fără să ia în considerare **valorile atributelor.**

Exemplu[Mart03]. Să plecăm de la o aplicație folosită de casele unui hipermarket, cu doi actori principali : Client (cumpărător) şi Vânzător (Figura 3.6).

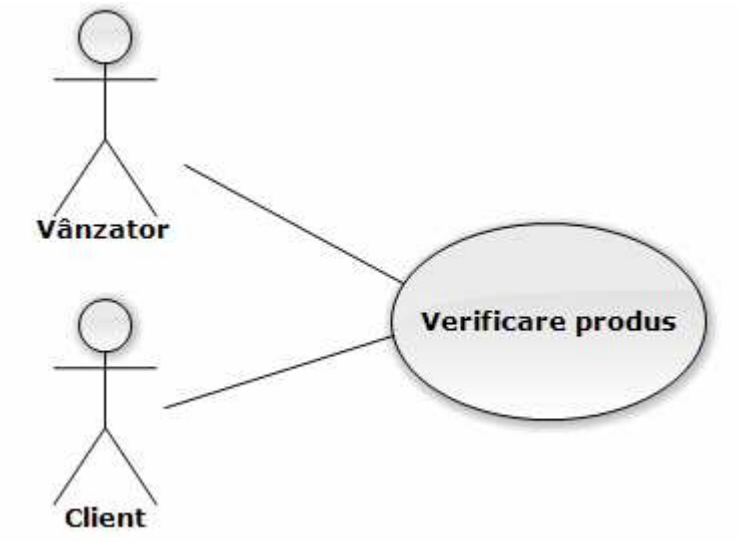


Figura 3.6.

Cazul de utilizare 1: Vânzătorul verifică un produs [Mart03]

1. Clientul prezintă produsul vânzătorului
2. Vânzătorul citeşte numărul de inventar şi îl introduce în calculator
3. Sistemul caută numărul de inventar în baza de date şi afişează descrierea şi prețul produsului
4. Sistemul afişează descrierea şi prețul produsului
5. Sistemul adaugă prețul şi tipul produsului la factura curentă

- Caz de eroare 1: Numărul de inventar nu este corect
  - Dacă după pasul 2, numărul de inventar este invalid, emite un sunet special

- Caz de eroare 2: Produsul nu se găseşte în baza de date
  - Dacă după pasul 3 nu se găseşte nici o intrare în baza de date pentru produs, se aprinde un indicator, şi se permite adăugarea de către vânzător a prețului şi descrierii produsului,

6. Mergi la pasul 4.

Relația "**uses**" ("**include**"):
Citirea şi introducerea numărului de inventar al produsului apare în descrierea cazului de utilizare "Verificare produs", dar apare şi în cazul de utilizare "Inventariere (produse)" (Figura 3.7). În loc de a face descrierea acestui comportament de două ori, putem utiliza relația «uses» pentru a arăta că el aparține ambelor cazuri de utilizare, şi cazul "Verificare produs" va arăta astfel:

Exemplu "**uses**": Cazul "Verificare produs"

1. Clientul prezintă produsul vânzătorului
2. «uses» "**citeşte număr inventar**"
3. Sistemul caută numărul de inventar în baza de date şi afişează descrierea şi prețul produsului
4. Sistemul afişează descrierea şi prețul produsului
5. Sistemul adaugă prețul şi tipul produsului la factura curentă

Relația «uses» seamană cu un apel de funcție (subrutină), iar cazul utilizat astfel se numeşte *caz de utilizare abstract* (nu poate exista de sine stătător ci trebuie să fie utilizat de alte cazuri).

Relația **"extends"**
Relatia „extends" exprimă tratarea separată a unei situații excepționale. Spre exemplu, în multe magazine vânzătorii sub 21 de ani nu au voie să introducă numărul de inventar pentru băuturi alcoolice: în aceste situații trebuie anunțat managerul, pentru a-i înlocui temporar. Astfel, în Figura 3.7, avem o relație de extindere între cazul de utilizare "Verificare produs" şi "Verificare produs 21". Această modificare a cazului de utilizare se poate exprima în două moduri:

1. Folosind "if"
2. Folosind puncte de extensie

Cazul "Verificare produs" (extins folosind **if**)

1. Clientul prezintă produsul vânzătorului
2. Dacă produsul este băutură alcoolică
   2.1. Trimite mesaj "21"
   2.2. Aşteaptă manager.
   2.3. Managerul «uses» "citire cod cu UPC"
   2.4. Mergi la pasul 4
3. «uses» "citire cod cu UPC"

4. Sistemul caută numărul de inventar în baza de date şi afişează descrierea şi preţul produsului
5. Sistemul afişează descrierea şi preţul produsului
6. Sistemul adaugă preţul şi tipul produsului la factura curentă

Aplicarea Principiului Deschis-Închis poate începe încă din etapa de definire a cerinţelor: dorim să scriem cod care să nu se modifice la modificarea cerinţelor. Într-un cod bine proiectat o modificare a cerinţelor determină *adăugarea* de cod nou, nu modificarea celui existent. Putem aplica aceeaşi regulă la specificarea funcţională a cazurilor de utilizare: când se modifică cerinţele, să adăugăm cazuri de utilizare noi. Astfel, în loc de a folosi "if", folosim relaţia «extends», ce permite specificarea unui caz nou ce suprascrie şi modifică cazul de utilizare existent (extins). Spre exemplu, în Figura 3.7, cazul de utilizare "Verificare 21" extinde cazul "Verificare produs".

Cazul "Verificare 21"

1. Înlocuieşte Pasul 2 din "Verificare produs" cu:
   1.1. Trimite mesaj "21"
   1.2. Aşteaptă manager
   1.3. Manager «uses» "citeşte număr inventar"

Puncte de extensie

În abordarea de mai sus, cazul "Verificare 21" menţionează cazul extins ("Verificare produs") direct. Putem repara această situaţie prin adăugarea de puncte de extensie cazurilor extinse. Punctele de extensie sunt nume simbolice de identificare a poziţiilor în cazul extins în care este apelat cazul care extinde.

Verificare produs

1. Clientul prezintă produsul vânzătorului
2. XP21: vânzătorul citeşte numărul de inventar
3. Sistemul caută numărul de inventar în baza de date şi afişează descrierea şi preţul produsului
4. Sistemul afişează descrierea şi preţul produsului
5. Sistemul adaugă preţul şi tipul produsului la factura curentă

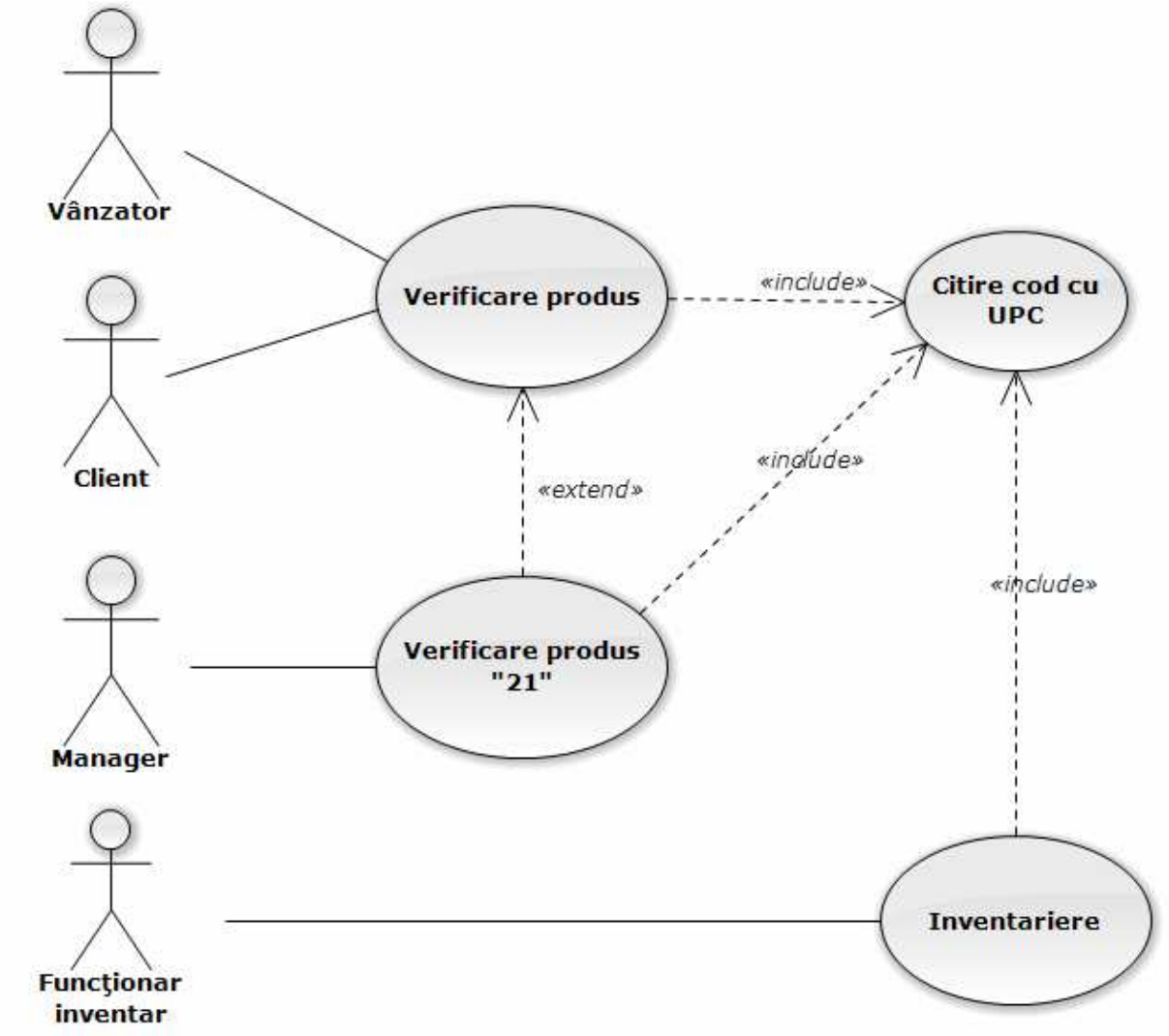


Figura 3.7. Cazuri de utilizare “Hipermarket”

Verificare 21

2.Înlocuieşte XP21 în cazul extins cu:

2.1. Trimite mesaj “21”

2.2. Aşteaptă manager

2.3. Manager «uses» “citeşte număr inventar”

3.4.3. Diagrama de colaborare şi diagrama de secvențe

Diagrama de colaborare şi diagrama de secvențe (denumite generic şi diagrame de interacțiune) sunt echivalente conceptual (ele chiar se pot transforma automat una în cealaltă), diferind doar prin dispunerea grafică a elementelor. Alături de diagrama de stare (Secțiunea 3.4.4.) ele formează *modelul dinamic*. Modelul dinamic este esențial pentru testarea modelului static (diagrama claselor): utilizat frecvent în verificarea software pentru a valida comportamentul claselor construite, modelul dinamic ne ajută la alegerea unui anumit model static dintre mai multe variante (design-ul unui program trebuie să fie determinat strict de

comportamentul aşteptat). Fiecare model dinamic explorează o anume variație a unui caz de utilizare, scenariu sau cerință, şi întotdeauna legăturile dintre obiectele din modelul dinamic implică existența unor asociații (între clase) în modelul static.

Ambele diagrame (de colaborare şi de secvențe) descriu fluxul de mesaje între obiecte (după cum s-a mai menționat, conțin aceeaşi informație).

Diagramele de secvențe pun accent pe ordinea în care sunt trimise mesajele, fiind utile în:

- Descrierea fluxului procedural într-un grup mare de obiecte (relevante pentru algoritmul folosit)
- Determinarea condițiilor de competiție în sistemele concurente

Diagramele de colaborare pun accent pe relațiile dintre obiecte, fiind utile în:

- Vizualizarea modului în care colaborează anumite obiecte pentru a îndeplini o sarcină
- Compararea modelului dinamic cu modelul static

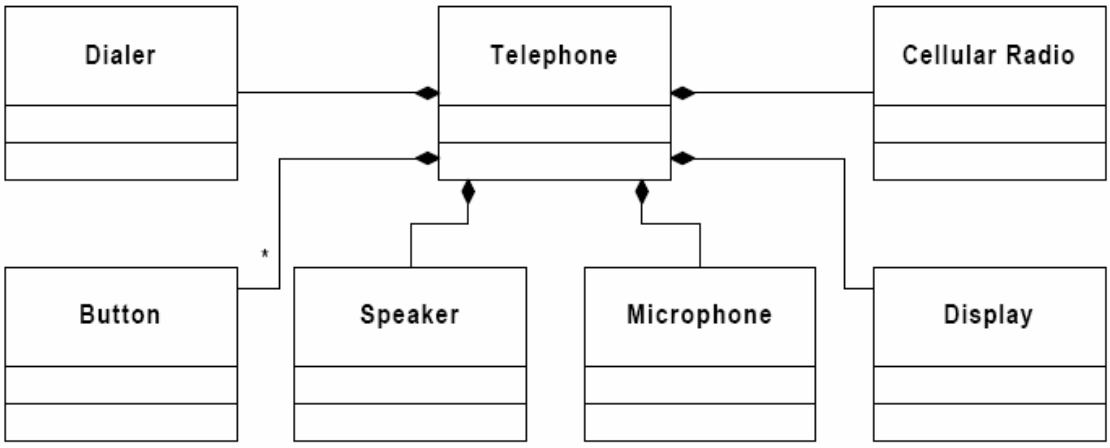


Figura 3.8. Exemplu: telefonul celular

Exemplu [Mart03]. În Figura 3.8, s-a plecat de la componentele fizice ale unui telefon celular în definirea claselor:

- Taste pentru numere,
- Un buton “Send” pentru apel
- Microfon, speaker, display

Scenariu: apel telefonic

1. Utilizatorul apasă tastele pentru a forma numărul de telefon
2. Pentru fiecare cifră, afişarea este actualizată astfel încât să apară şi noua cifră tastată
3. Pentru fiecare cifră, tastatura (dialer) generează un sunet corespunzător

4. Utilizatorul apasă “Send”
5. Indicatorul “in use” se aprinde pe ecran (display)
6. Radio-ul celular (cellular radio) stabileşte o conexiune cu rețeaua
7. Lista de cifre este trimisă rețelei
8. Se stabileşte conexiunea cu partea apelată

Scenariul de mai sus descrie procedura de realizare a unui apel. Vom urmări în continuare cum colaborează obiectele din modelul static în executarea acestei proceduri.

Apel telefonic- obiecte implicate

- Tastarea numărului – un obiect de tip Buton trimite un mesaj (cifra asociată butonului), către obiectul de tip Dialer.

- Obiectul Dialer (care formează numărul de telefon) transmite ecranului (Display) să afişeze noua cifră şi microfonului (Speaker) să emită tonul potrivit

- Obiectul Dialer trebuie să rețină într-o listă cifrele tastate (numărul de telefon)

- Operația se repetă până la apăsarea butonului “Send”, care trimite mesajul “send” către Dialer, care trimite la rândul său un mesaj Connect şi numărul de telefon către CellularRadio

- CellularRadio solicită Display să ilumineze indicatorul “In Use”

Dreptunghiurile diagramei de colaborare reprezintă obiecte (instanțe ale claselor). Legăturile sunt instanțieri ale asociațiilor (permise doar dacă au drept corespondent o asociere, agregare sau compunere în diagrama de clase). Săgețile reprezintă mesaje (etichetate cu numele lor, numerotarea în secvență şi argumente):

- Numele unui mesaj este numele funcției membre (metodă) din clasa obiectului ce recepționează mesajul;
- Numerele corespund ordinii de apariție a mesajelor;
- Numerele imbricate sunt folosite pentru a depista ce mesaje se trimit din interiorul altor mesaje;
- Message 1*:Digit(code): asteriscul reprezintă o buclă: mesajul “Digit” poate apărea de mai multe ori înainte de mesajul 2.

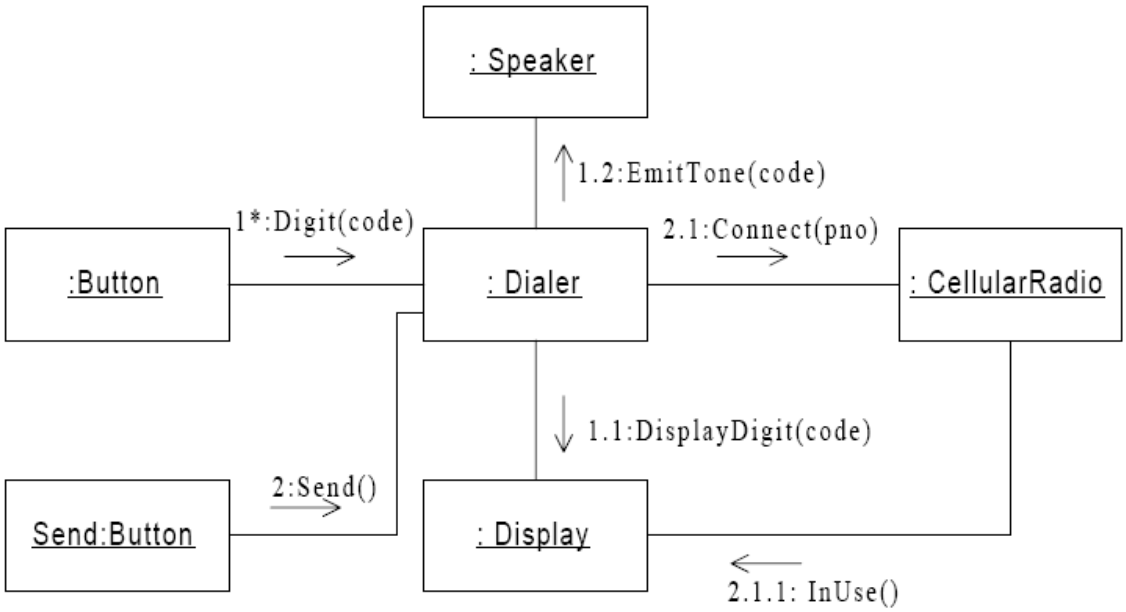


Figura 3.9. Diagrama de colaborare a cazului "Apel telefonic"

Armonizarea modelului static cu cel dinamic

Observăm că modelul static (Figura 3.8) nu seamănă cu modelul dinamic (Figura 3.9), deoarece o legătură între obiecte trebuie să corespundă unei asocieri între clase, deci trebuie corectat modelul static. Dacă am forța modelul dinamic să urmeze modelul static, am avea un obiect ce concentrează toată inteligența, multiplu conectat (cuplaj ridicat): obiectul Telephone (master controller). Dezavantajul acestei abordări este că atunci când o parte a obiectului master se schimbă, alte părți ale sale pot deveni nefuncționale, ceea ce este mai puțin probabil dacă împărțim sarcinile între mai multe obiecte. Descentralizarea inteligenței (Figura 3.9) are avantajul că modificări ale unei părți ale modelului nu se repercutează în mod necesar asupra altor părți. În Figura 3.10 observăm modelul static corectat.

Noul model static este construit după comportamentul real al telefonului, nu după componența sa fizică. Întrucât clasele telefon şi microfon nu au jucat nici un rol în modelul dinamic, au fost şterse (dacă vor fi necesare într-un alt scenariu dinamic, vor fi re-adăugate). Problemele noului model static sunt:

1. Ar fi util să putem reutiliza clasa Button în programe ce nu au "Dialers" (clasa Button nu ar trebui să aibă nevoie să ştie despre clasa Dialer). Soluția acestei probleme este şablonul Server Adaptat (Fig. 3.11).

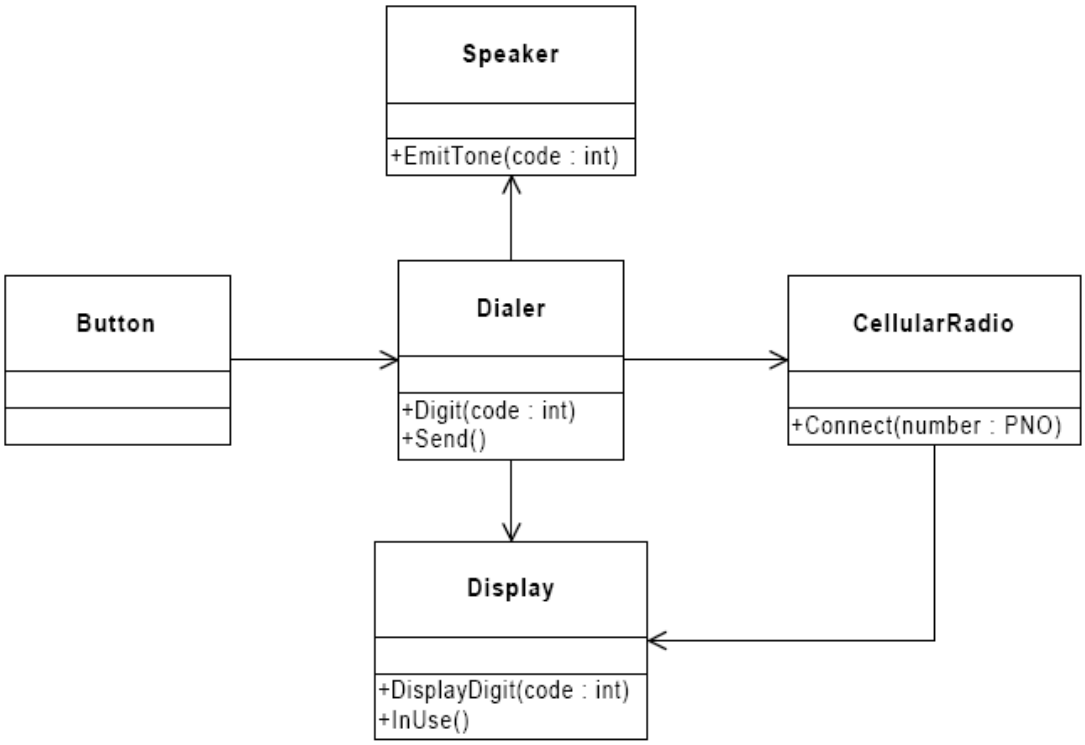


Figura 3.10. Modelul static corectat

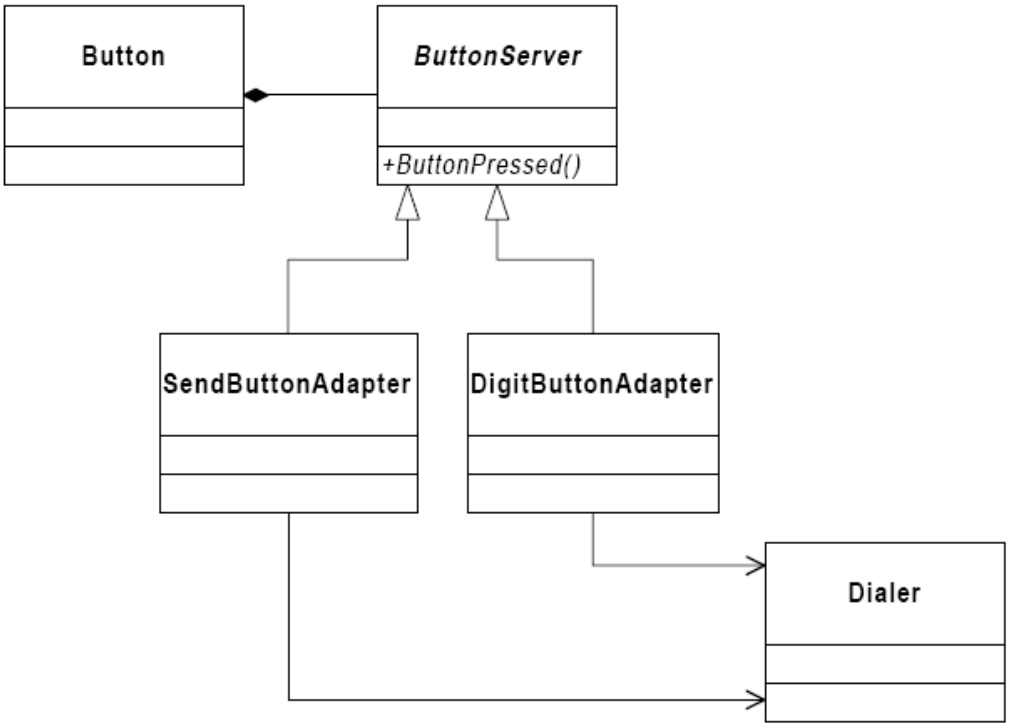


Figura 3.11. Şablonul Server Adaptat

Prin aplicarea şablonului “Server Adaptat” (Figura 3.11), clasa Button a devenit complet reutilizabilă (orice altă clasă ce necesită detectarea unui buton apăsat extinde ButtonServer şi implementează funcția virtuală ButtonPressed).

2. A doua problemă a modelului static din Figura 3.10 este cuplajul ridicat al clasei Display (ținta asocierii de la mai mulți clienți diferiți). Dacă una dintre metodele din Display trebuie modificată din cauza necesităților clasei Dialer, atunci CellularRadio va fi afectat (va trebui cel puțin recompilat).

Soluția acestei probleme constă în segregarea interfeței clasei Display (Figura 3.12).

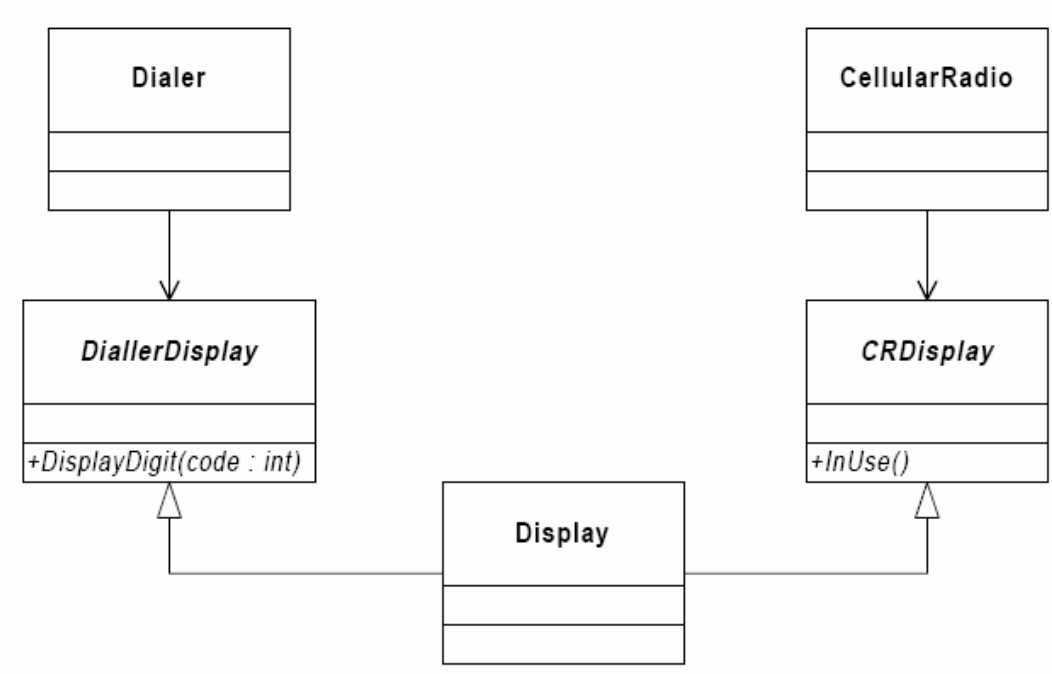


Figura 3.12. Segregarea interfeței clasei Display

Diagramele de secvențe

Diagramele de secvențe conțin aceeaşi informație ca diagramele de colaborare dar accentuează secvența de mesaje, în locul colaborării între obiecte. Ele conțin secvențe de evenimente, şi pot modela când sunt create/distruse obiectele, şi operațiile concurente (relevarea interacțiunilor concurente cu mai multe fire este dificil de vizualizat într-o diagramă de colaborare). Un algoritm poate fi descris ca o suită de interacțiuni între obiecte. În diagrama de secvențe, mesajele între obiecte ilustrează procesele domeniului modelat (cum sunt realizate sarcinile ca interacțiuni între obiecte în cadrul unui scenariu). Mesajele între obiecte pot apărea doar în direcția unei asociații navigabile (direcționată).

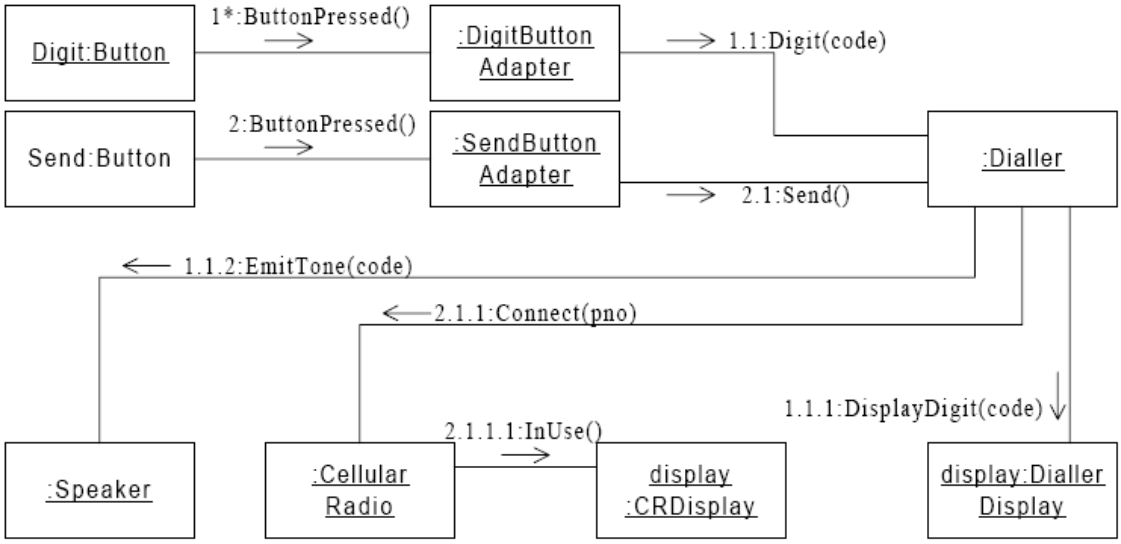


Figura 3.13. Modelul dinamic iterat

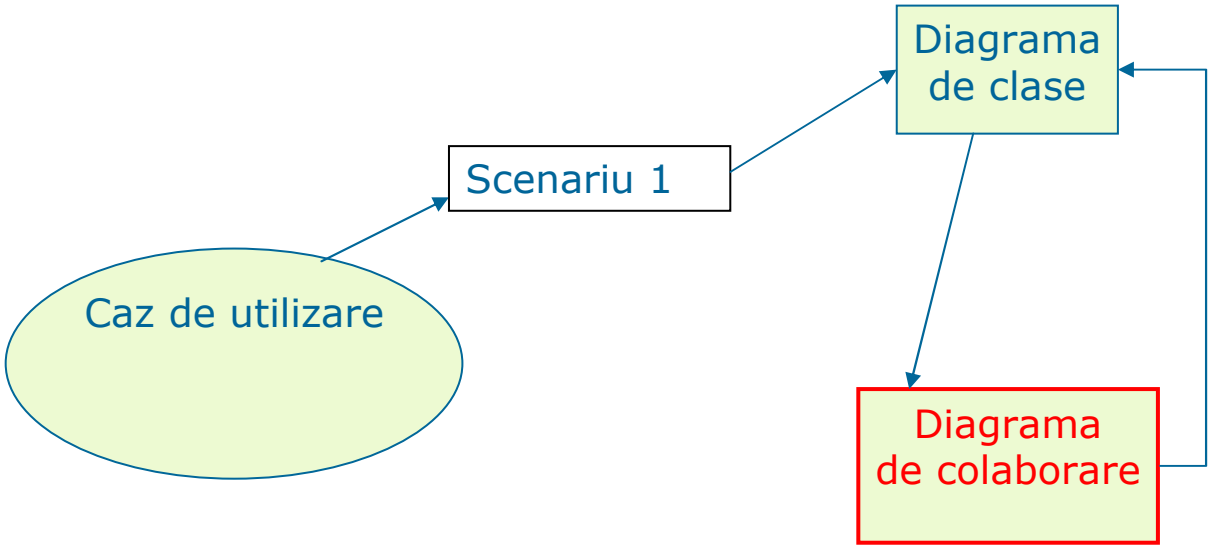


Figura 3.14. Algoritmul de optimizare a unui model

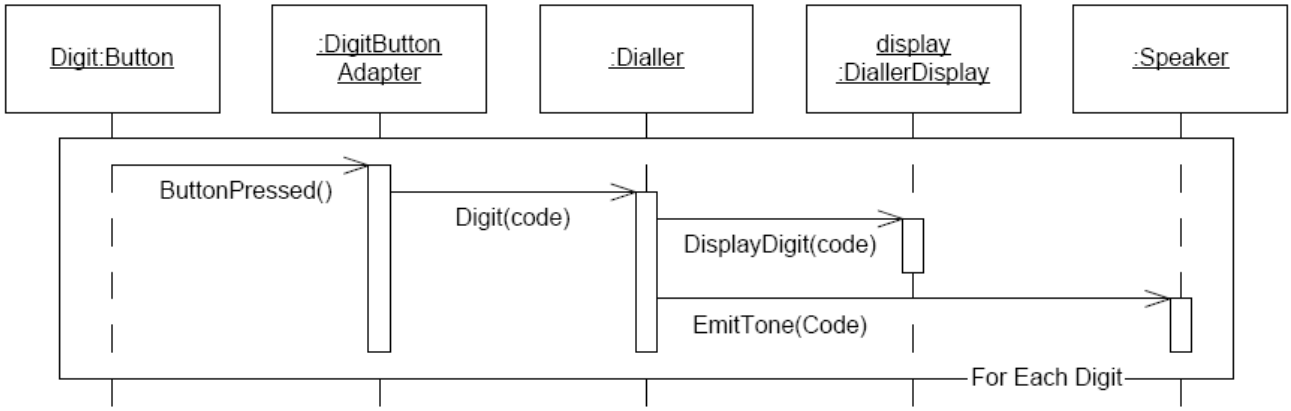


Figura 3.15. Diagrama de secvențe la apăsarea unei taste

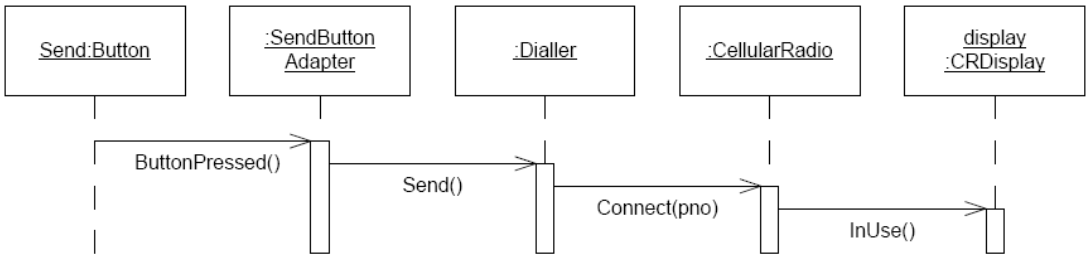


Figura 3.16. Diagrama de secvențe la apăsarea "Send"

3.4.4. Diagrama de stare

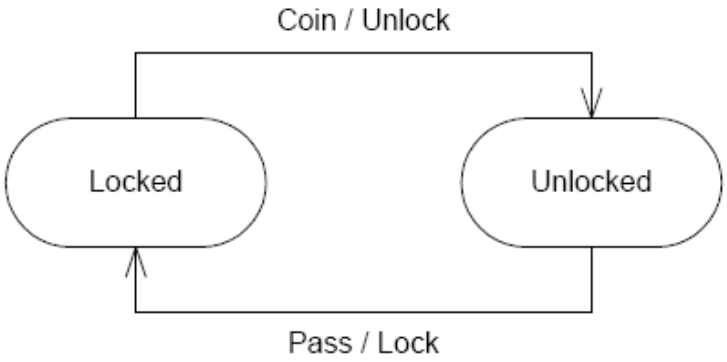


Figura 3.17. Diagrama de stare a unei intrări de metrou [Mart03]

Diagrama de stare reprezintă o clasă ca o maşină finită de stări, formată din stări şi tranziții între stări. O *stare* este o configurație particulară a valorilor atributelor obiectului iar o *tranziție* defineşte următoarea stare permisă. Fiecare stare poate specifica o activitate de executat cât timp obiectul se află în starea respectivă.

Aceste diagrame descriu comportamentul obiectelor unei clase în termeni de stări observabile şi modificări ale acestor stări în funcție de evenimente ce afectează obiectul. Faptul că modificarea de comportament de la o stare la alta este observabilă presupune că, dacă acelaşi mesaj este trimis obiectului de două ori, obiectul poate reacționa diferit, în funcție de starea în care se află la recepționarea mesajului.

Există şi super- stări, formate din mai multe sub-stări componente, şi în care fiecare substare moşteneşte toate tranzițiile ce intră/ ies din superstarea sa.

Etichetele tranzițiilor

- *eveniment [condiție] / acțiune;*
- Eveniment ce declanşează tranziția (de obicei, receptarea unui mesaj);
- Condiție (guard) ce trebuie îndeplinită înainte ca tranziția să poată avea loc. Condițiile sunt utile în reprezentarea tranzițiilor în stări diferite ca urmare a unui singur eveniment;
- O acțiune specifică procesarea ce trebuie executată la declanşarea tranziției.

Utilitatea diagramelor de stări

Diagramele de stări sunt o reprezentare foarte densă, bazată pe reguli simple şi uşor de verificat ce poate fi utilizată inclusiv în generarea de cod. Ele sunt un instrument puternic de descriere şi implementare a logicii controlului aplicațiilor, foarte utile în modelarea protocoalelor de comunicație, a controlului interacțiunilor într-o IUG etc.

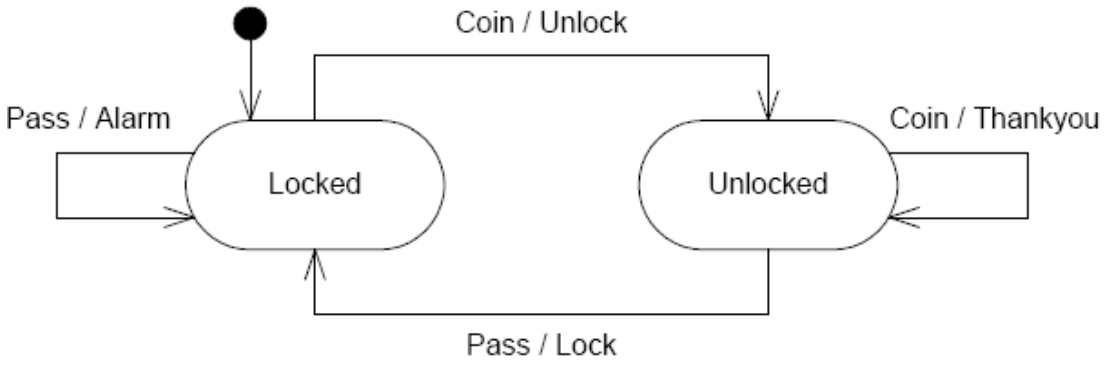


Figura 3.18. Intrare metrou: evenimente anormale [Mart03]

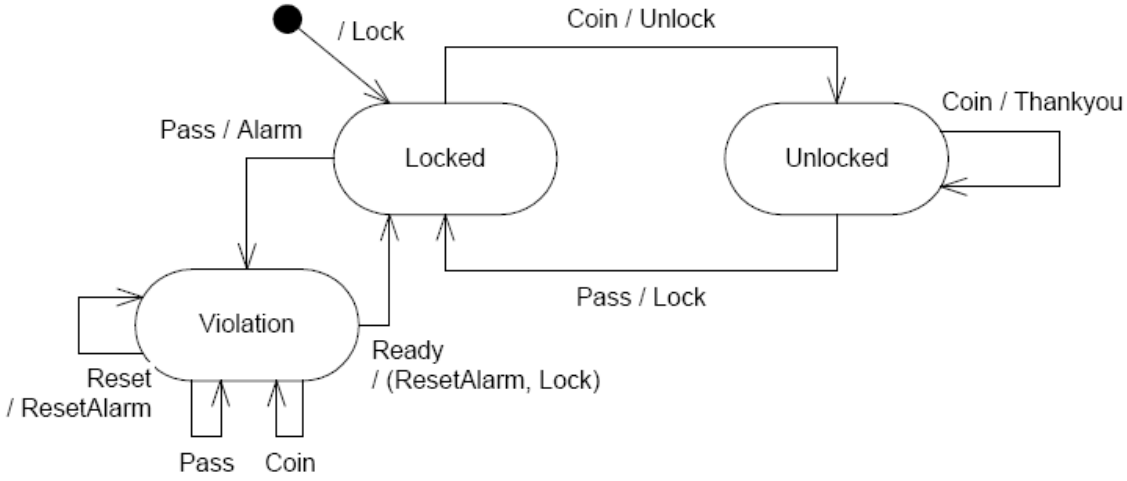


Figura 3.19. Intrare metrou: evenimente anormale- versiunea optimizată [Mart03]

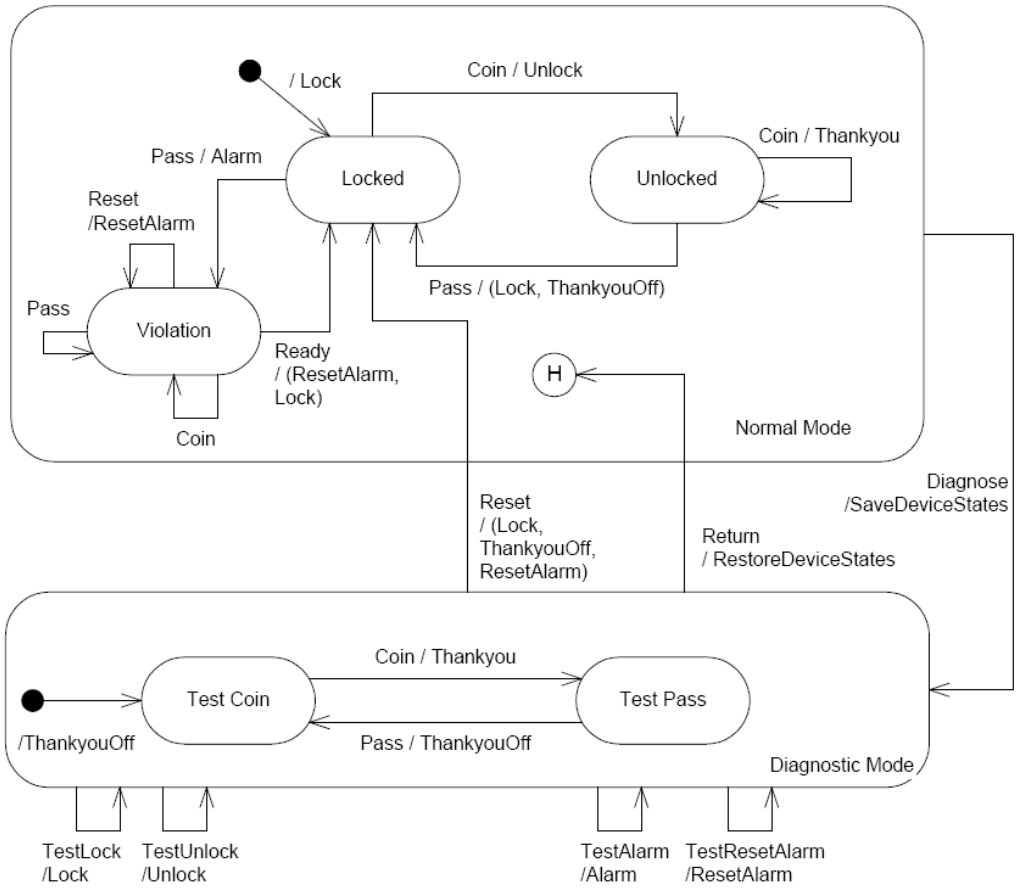


Figura 3.20. Intrare metrou cu diagnoză [Mart03]

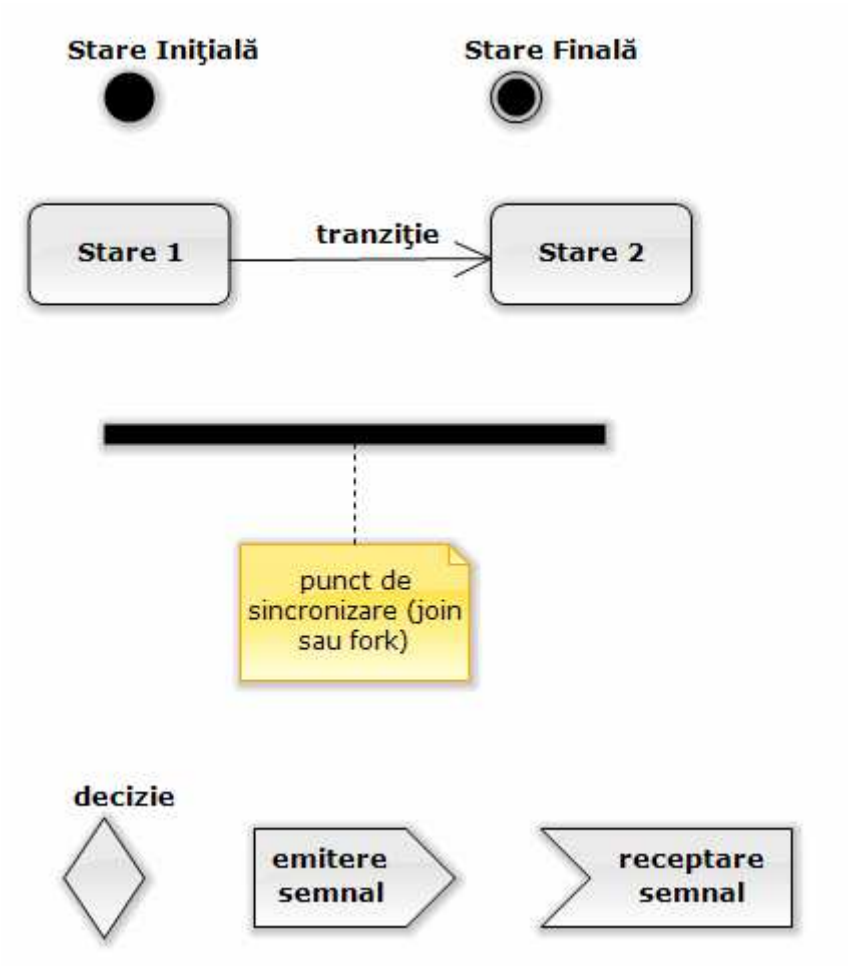


Figura 3.21. Simboluri folosite în diagrama de stare

3.4.5. Diagrama de activități

În diagramele de secvențe este dificil şi câteodată imposibil de reprezentat iterația şi concurența. Astfel au apărut maşinile de stări cu mai multe fire de execuție, în cadrul cărora tranziții complexe împart/ reunesc mai multe fire de execuție. Acestea sunt diagramele de activități, ce folosesc o combinație de notații de tip rețele Petri şi scheme logice.

Exemplu: Maşina de prăjit pâine [Mart03]

Presupunem că există două spații separate pentru două felii de pâine, şi la apăsarea unei manete, maşina primeşte evenimentul 'Start', care declanşează o *tranziție complexă.* Vom şti când pâinea este gata:

- După culoare
- După trecerea unui anumit timp (poate este o bucată de gresie albă în toaster…)
- Dacă se modifică prea repede culoarea, toaster-ul se opreşte (pentru a preveni incendiile dacă cineva pune hârtie, spre exemplu)

Controlerul trebuie să se ocupe de reglarea temperaturii, astfel încât să nu fie nici prea fierbinte, nici prea rece. De ce culoarea pâinii se schimbă, temperatura optimă de prăjit creşte (temperatura este o funcție de culoare).

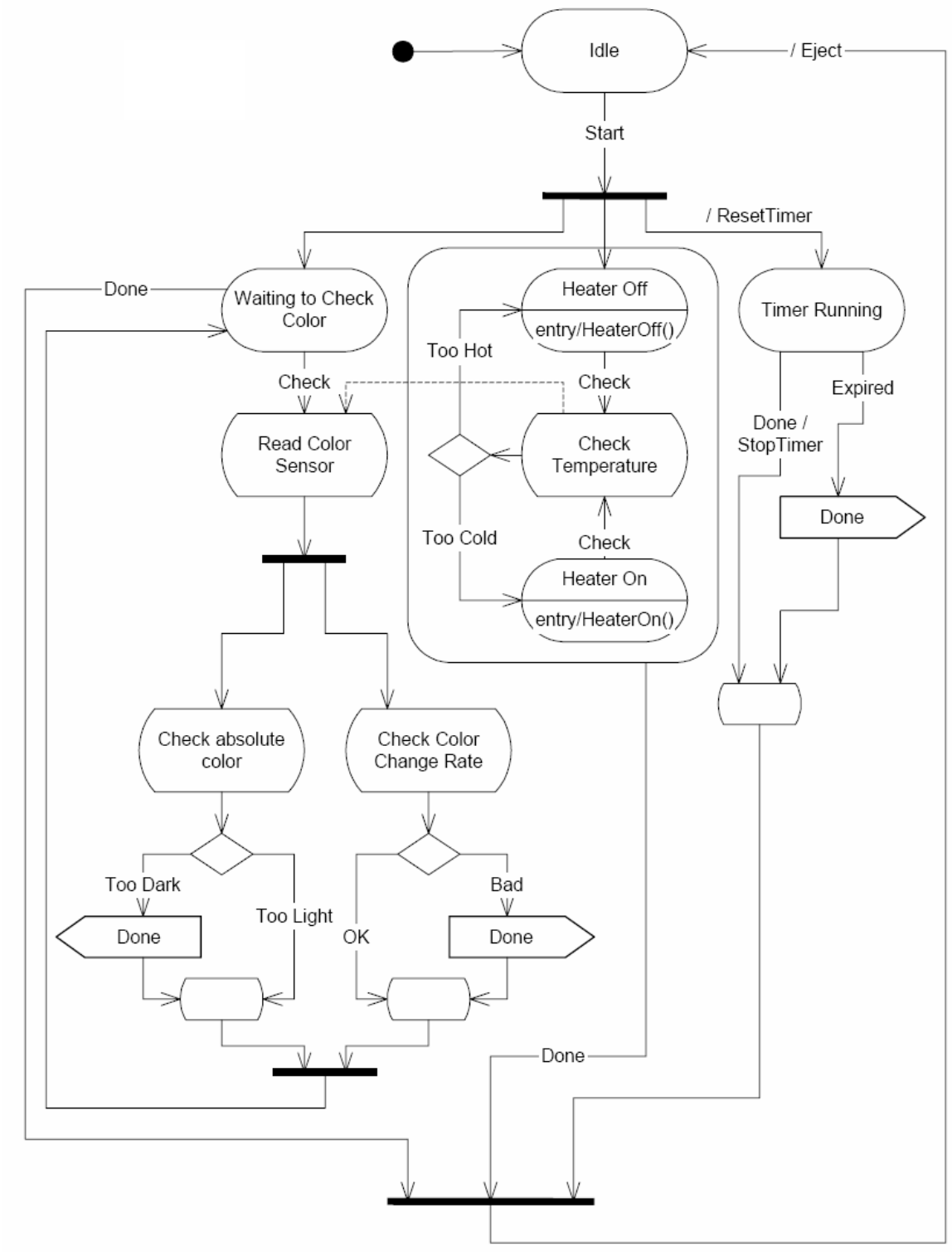


Figura 3.22. Diagrama de activități a controlerului unui prăjitor de pâine (maşină de stări cu fire de execuție multiple) [Mart03]

Concluzii

Modelul dinamic este un instrument util în verificarea şi optimizarea modelului static, întrucât îl redefineşte prin prisma funcționalității şi comportamentului său. Modelul dinamic este folositor atât în etapa de design cât şi în etapa de testare.

Exercițiu.

Realizați modelul UML complet al unei aplicații de la exercițiile Capitolelor 6-8.

CAP. 4. PRINCIPII DE DESIGN ALE PROGRAMELOR ORIENTATE OBIECT

4.1. Introducere -Programarea orientată obiect

Ne propunem în Secțiunea de față să introducem conceptele necesare pentru înțelegerea aprofundată şi de ansamblu a Programării Orientate Obiect (POO). Vom explica de ce sunt importante obiectele, cum putem să le definim, şi vom urmări câteva exemple.

Abstractizarea [Ecke09]

Toate limbajele de programare folosesc abstractizarea într-un fel sau altul (depinde doar ce abstractizează). Spre exemplu, limbajul de asamblare abstractizează maşina, în timp ce limbajul C poate fi considerat o abstractizare a limbajului de asamblare. Totuşi, chiar şi limbajul C obligă încă programatorul să gândească în funcție de **structura calculatorului**, nu a **problemei.** Programatorul este cel care trebuie să stabilească asocierea între **modelul-maşină** (în "spațiul soluției"-calculatorul) şi **modelul problemei** rezolvate (în "spațiul problemei") –programele fiind dificil de scris şi greu mentenabile.

POO reprezintă o alternativă la modelarea maşinii, întrucât ne ajută să **modelăm problema** pe care dorim să o rezolvăm. Paradigma oferă programatorului instrumente de reprezentare a spațiului problemei, anume **obiectele-** elemente în spațiul problemei şi reprezentarea lor în spațiul soluției. O provocare importantă a POO este crearea unei corespondențe bijective între elementele din spațiul problemei şi elementele din spațiul soluției. Clasele POO sunt tipuri abstracte de date.

1. Totul este un obiect	− Obiect= variabilă de calitate superioară:
	  − reține date;
	  − îi putem adresa cereri (operații asupra sa însăşi);
	− Orice componentă conceptuală a problemei poate deveni obiect în program;

2. Un program este o mulțime de obiecte care îşi trimit unele altora mesaje (cereri)	− Mesaj = apel al unei metode ce aparține unui obiect particular;
3. Fiecare obiect deține o memorie proprie (formată din alte obiecte)	− Programe complexe, aspect simplu;
4. Orice obiect are un tip	− Este o instanță a unei clase;

	– Cea mai importantă caracteristică a unei clase este răspunsul la întrebarea "ce mesaje îi pot fi trimise?"
5. Toate obiectele de un anumit tip pot primi aceleaşi mesaje	– "Circle" acceptă mesajele clasei "Shape" – substitutabilitatea (polimorfism)

Tabel 4.1. Cele 5 caracteristici de bază ale limbajelor orientate-obiect [Kay03]

*"Orice obiect are o **stare**, un **comportament** şi o **identitate**"*[Booc08] -adica un obiect poate avea **date interne** (care îi determină starea), **metode** (pentru a produce un anumit comportament), şi fiecare obiect poate fi distins în mod unic de alte obiecte (are o **adresă unică** în memorie).

Orice obiect are o interfață, cererile ce pot fi adresate unui obiect fiind definite de *interfața* sa (tipul determină interfața).

Exemplu.

Nume tip

Interfață

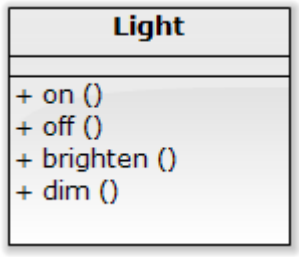


Figura 4.1. Tip şi interfață

```
Light lt = new Light(); lt.on();
```

Orice obiect este creat pentru a oferi servicii- este "prestator de servicii". Această viziune are două avantaje:

- Descompunerea (top-down) a problemei în obiecte va fi mai uşoară
- Coeziunea unui obiect se îmbunătățeşte
- Este mai clară şi îmbunătățeşte reutilizabilitatea: un obiect văzut prin prisma serviciilor oferite se poate potrivi mai bine în design

Coeziunea ridicată este o calitate fundamentală a design-ului software. Coeziunea presupune ca diferitele aspecte ale unei componente software (obiect/ librărie de obiecte etc.) să se potrivească bine împreună.

Pentru că fiecare obiect trebuie să aibă o mulțime coezivă de servicii oferite, nu trebuie aglomerate prea multe funcționalități într-un singur obiect. Într-un design orientat obiect de calitate, fiecare obiect face un lucru bine, fără să încerce să facă prea mult, ceea ce facilitează reutilizarea.

Ascunderea implementării

În cadrul programării orientate-obiect există două tipuri de dezvoltatori: **creatorii de clase** (creatori de tipuri noi de date) şi **programatorii-client** (consumatori de clase ce folosesc tipurile de date în aplicațiile lor). Scopul programatorului client este de a colecta clase ce pot fi folosite uşor şi rapid în dezvoltarea de aplicații, iar scopul creatorului de clase este să construiască o clasă ce expune doar partea absolut necesară programatorului şi ascunde restul, specificatorii de acces fiind cei cunoscuți: *public*, *private* şi *protected*. Avantajele decurg în mod firesc:

- Partea ascunsă este inaccesibilă programatorului, deci creatorul o poate modifica fără ca aceasta să aibă impact asupra cuiva;
- Se reduc bug-urile (întrucât partea ascunsă conține aspecte delicate ale obiectului, ce pot fi corupte de un programator neglijent/ neinformat).

Compunerea claselor: Reutilizarea implementării

Reutilizabilitatea codului unul este dintre cele mai mari avantaje ale programarii orientate-obiect. Relația de compunere a claselor este definită prin declararea într-o clasă de date membru de tipul altor clase (vezi şi Secțiunea 3.4.1; Figura 4.2).

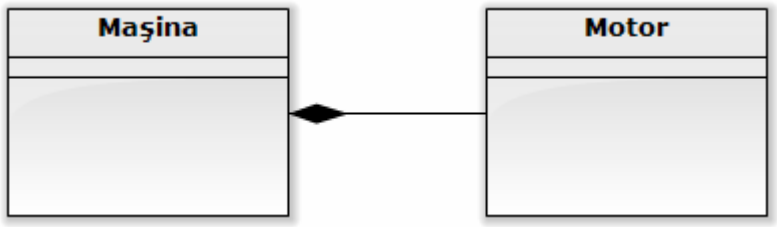


Figura 4.2. Compunerea

Compunerea este mai simplă şi mult mai flexibilă decât derivarea, şi de aceea ar trebui preferată acesteia ori de câte ori este posibil. Datele membru de tip clasă este indicat de obicei să fie private, şi deci inaccesibile programatorilor- clienți ce folosesc clasa.

Moştenirea: reutilizarea interfeței

În cazul moştenirii, clasa derivată are funcționalități similare clasei de bază, eventual adaugând metode noi, sau /şi suprascriind metode ale clasei de bază. Modificările clasei de bază se reflectă în clasa derivată (ceea ce de multe ori este un dezavantaj).

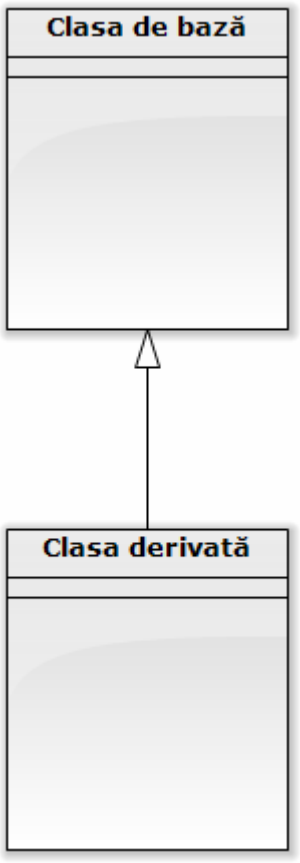


Figura 4.3. Derivarea

Moştenirea poate modela doua tipuri de relații: ***este-un*** sau ***este–ca-o.*** Relația "*este-un*" doar suprascrie metodele existente ale clasei de bază (fiind ideală întrucât respectă **principiul substituției-** vezi Secțiunea 4.3.2). Relația "*este-ca-un*" adaugă metode noi în clasa derivată, nerespectând principiul substituției, şi în această situație este de preferat utilizarea compoziției sau crearea unei clase de bază mai generale din care să derivăm mai multe clase. Există situații când adăugarea de metode noi în clasa derivată este de neevitat (Figura 4.4.).

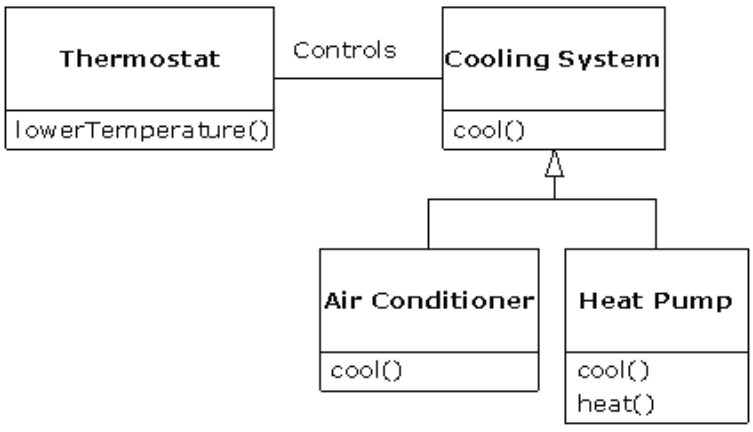


Figura 4.4. Exemplu Termostat

Polimorfismul: obiecte interschimbabile

Polimorfismul este mecanismul care permite scrierea de cod independent de un tip specific (permite tratarea obiectelor ca aparținând tipului dat de clasa de bază).

Exemplu.

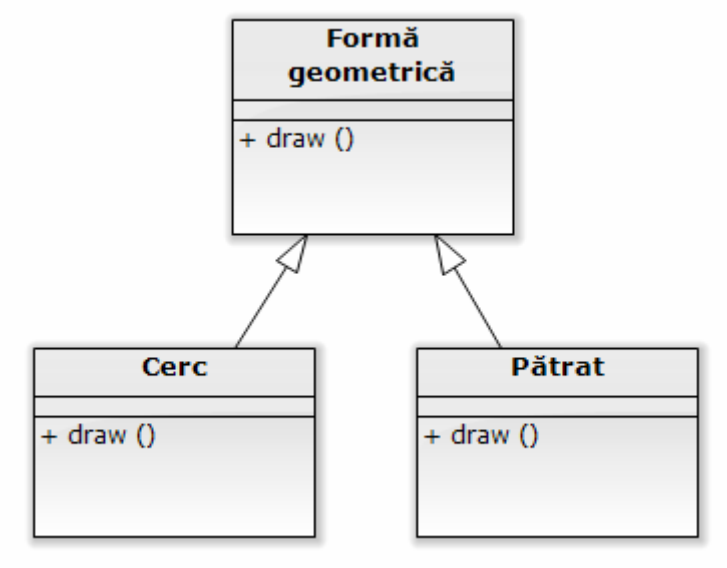


Figura 4.5. Exemplu polimorfism

```
void doStuff(Shape s)
{ s.erase();
// ...
s.draw(); }


Circle c = new Circle();
Triangle t = new Triangle();
Line l = new Line();
   doStuff(c);
   doStuff(t);
   doStuff(l);
```

În exemplul de mai sus, toate formele pot fi desenate, șterse, mutate, indiferent de tipul specific de formă geometrică. Avantajul acestui tip de cod este că el nu este afectat de adăugarea de tipuri noi (una dintre cele mai frecvente metode de a extinde un program orientat-obiect pentru a-l adapta noilor situații) – și astfel costul de mentenanță este redus. Polimorfismul folosește mecanismul legării târzii (la execuție, nu la compilare).

Interfețe şi clase de bază abstracte

O clasă de bază abstractă reprezintă doar o interfață pentru clasele sale derivate, constituind un instrument de forțare a unui anumit design. În Java, cuvântul-cheie **interface** (care înlocuieşte şi moştenirea multiplă) asigură o separație completă interfață-implementare.

Crearea, utilizarea şi distrugerea obiectelor

În limbajul C++ accentul cade pe controlul eficienței şi obiectele sunt plasate pe stiva statică **(stack).** Avantajul acestei abordări constă în viteza la alocare/ dealocare, iar dezavantajul în reducerea flexibilității (trebuie să ştim exact cantitatea, durata de viață şi tipul obiectelor în timpul scrierii programelor). Dacă abia la execuție se cunoaşte numărul de obiecte şi durata de viață, este de preferat să creem dinamic obiectele pe stiva dinamică (heap). Java foloseşte exclusiv această abordare (operatorul *new* de creare a obiectelor). Avantajul constă în gestionarea mai eficientă a memoriei (foarte important în cazul unor cantități mari de date), iar dezavantajul în timpul mărit de manevrare a memoriei (comparativ cu alocarea statică).

În privința distrugerii obiectelor, la alocarea pe **stack** compilatorul determină durata de viață şi poate şterge automat obiectele ne-necesare, iar la alocarea pe **heap** programatorul trebuie să se ocupe de ştergerea obiectelor (nu şi în Java unde mecanismul *garbage collector* detectează automat obiectele care nu mai sunt folosite şi le distruge).

Dacă până la execuție nu ştim de câte obiecte avem nevoie, nici ce durată de viață au, vom stoca obiectele folosind liste dinamice – colecții şi iteratori (în Java ArrayList, LinkedList etc.).

4.2. Introducere în design-ul orientat-obiect (OO)

O idee esențială a design-ului orientat-obiect este că această paradigmă trebuie folosită pentru *sistemele de dimensiuni mari, ce se schimbă frecvent*.

Metodele *orientate-obiect* sunt tehnici de analiză, descompunere şi modularizare a arhitecturilor sistemelor software ce structurează arhitectura sistemelor în funcție de *obiectele* sale (şi de clasele de obiecte), mai mult decât în funcție de *acțiunile* pe care le execută. Justificarea folosirii acestor metode este dată de faptul că, în general, sistemele se pot schimba şi funcționalitățile pot evolua, dar obiectele şi clasele tind să rămână stabile în timp.

Nu trebuie confundat design-ul OO cu programarea OO. Astfel, programarea OO presupune construcția sistemelor software ca o colecție structurată de tipuri Abstracte de Date (TAD), ce foloseşte moştenirea şi

polimorfismul, şi se ocupă de detaliile de implementare asociate cu diferite limbaje deprogramare specifice. Programarea OO este un support pentru design-ul orientat obiect. Acesta din urmă este o metodă de descompunere a arhitecturilor software bazată pe obiectele manevrate de fiecare sistem şi relativ independentă de limbajul de programare utilizat.

Unul din cele mai importante aspecte ale design-ului OO este polimorfismul (Figura 4.6), prin intermediul căruia comportamentul promis în interfața publică a obiectelor superclasei este implementat de obiectele subclasei (în modul specific subclasei). Importanța sa rezidă din flexibilitatea pe care o conferă design-ului (arhitecturilor): logica de nivel înalt se defineşte în termeni de interfețe abstracte, bazându-ne pe implementarea concretă din subclase, şi se pot *adăuga subclase fără a schimba* logica de nivel înalt. Rezultatul este că prin polimorfism obținem stabilitate în ceea *ce* se execută, dar flexibilitate în modul *cum* se execută.

Caracteristicile unui design slab (orientat obiect sau nu) sunt:

- *Rigiditatea* (Cod dificil de modificat)
- *Fragilitatea* (Modificări minore pot provoca probleme în cascadă)
- *Imobilitatea* (Codul este atât de amestecat şi încâlcit încât este imposibil de reutilizat ceva)

Principiile prezentate în secțiunea următoare ne vor ajuta să înțelegem cauzele unui design slab. Cerințele de modificare a unui program sunt inevitabile. Aşa cum spunea Ivar Jacobson [Jaco92] "*Toate sistemele se vor schimba pe parcursul folosirii lor. La aceasta trebuie să ne gândim când proiectăm sisteme ce dorim să dureze mai mult decât prima versiune*". Aşadar, atât design-urile bune cât şi cele slabe trebuie să facă față schimbărilor, dar un design bun va fi stabil în confruntarea cu aceste schimbări. Foarte important în această stabilitate este managementul dependențelor, strâns legat de problema cuplării şi coeziunii, şi care poate fi controlat (spre exemplu, prin Principiul Inversării Dependențelor -PID).

```
//Metoda refresh  (ScreenManager) (Fig.4.6)

for all Shapes in screen  {move (newPosition)}
```

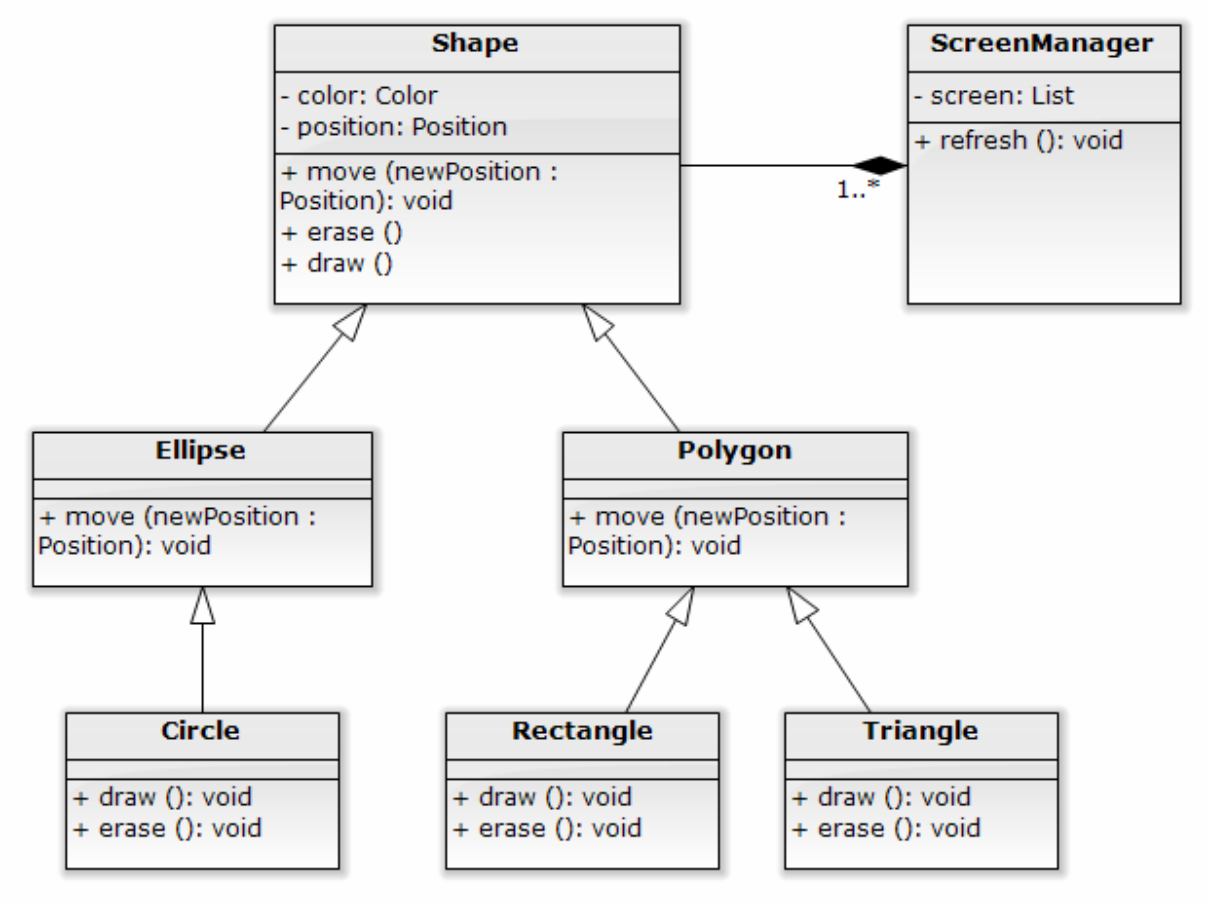


Figura 4.6.Exemplu polimorfism

4.3. Cele 3 principii fundamentale ale design-ului OO

4.3.1. Principiul Deschis-Închis (PDÎ)

"Entitățile software trebuie să fie deschise la extinderi, dar închise la modificări."

Bertrand Meyer

Deschiderea la extinderi presupune că putem extinde comportamentul modulului, iar închiderea la modificări înseamnă că nu trebuie să modificăm codul sursă al modulului. Prin urmare, *modulele trebuie scrise astfel încât să poată să fie extinse fără să fie modificate.*

Exemplu. Să considerăm cele două clase din Figura 4.7. Avem o compunere între clasa "Maşina" şi clasa "Motor cu piston". În această variantă, dacă dorim să folosim şi un motor Turbo, este evident că va trebui să modificăm codul clasei "Maşina"- deci acesta nu este stabil.

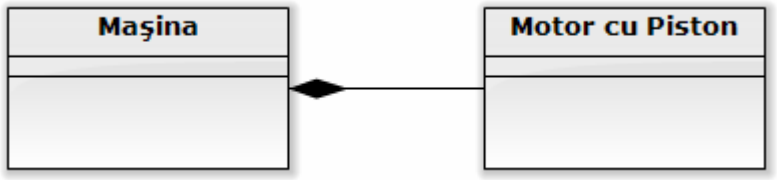


Figura 4.7. Structură rigidă, dificil de extins. Codul clasei “Maşina” este instabil.

Dacă însă facem apel la polimorfism şi introducem o clasă abstractă pentru modelarea motoarelor în general (Figura 4.8), la extinderea programului pentru a folosi şi motoare Turbo va fi suficient să scriem o nouă clasă “Motor Turbo” derivată din “Motor Abstract”. În consecință, codul clasei “Maşina” este stabil față de adăugarea de tipuri noi de motoare, el depinzând de o abstracțiune, nu de codul unei clase concrete.

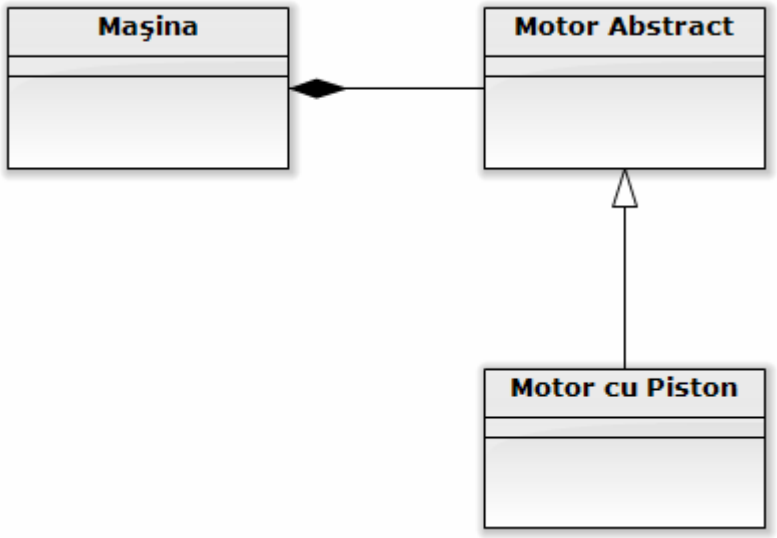


Figura 4.8. Structură flexibilă, uşor de extins. Codul clasei “Maşina” este stabil.

În concluzie, pentru respectarea principiul Deschis-Închis, o clasă nu trebuie să depindă de o clasă concretă ci de o clasă abstractă, folosind dependențe (apeluri) polimorfice.

Închiderea unui program trebuie să fie strategică, nu completă. Dupa cum afirma Robert Martin *“Nici un program semnificativ nu poate fi 100% închis.”* [Mart02]. Aceasta înseamnă că trebuie să alegem acele aspecte ce pot fi extinse cu uşurință, şi codul care dorim să rămână stabil față de ele- ceea ce depinde de aplicație şi de contextul folosirii acesteia.

Închiderea unui program se obține prin utilizarea abstractizării. Pentru a determina deciziile strategiei generale (spre exemplu, desenare de pătrate înainte de cercuri) trebuie construite metode ce pot fi invocate

dinamic. De asemenea, pentru închidere, ar trebui utilizată o abordare "condusă de date": plasarea deciziilor strategiei particulare într-o locație separată (de exemplu, fişier sau obiect separat) minimizează localizările modificărilor viitoare.

Euristici PDÎ

1. O primă euristică ce ajută la respectarea principiului este *eliminarea variabilelor globale* şi *declararea tuturor datelor membru private*. Justificarea este că, pe de o parte, modificările datelor publice aduc un risc de "deschidere" a modulului; ele pot provoca un efect de modificări în cascadă, în mai multe locații neașteptate, şi erorile pot fi dificil de corectat/ de găsit în totalitate (iar corecturile pot introduce erori în alte părți). Pe de altă parte, membrii non-privați sunt modificabili şi pot schimba starea clasei.

2. ITE (identificarea tipului la executie) este neelegantă şi periculoasă. Dacă un modul încearcă să convertească *dinamic un pointer la clasa de bază către una dintre clasele derivate*, atunci de câte ori se extinde ierarhia de clase, se va modifica şi modulul respectiv. Abordarea este uşor de recunoscut prin structurile tip *switch* sau *if-else*, (nu toate aceste situații încalcă PDI, ci doar când sunt folosite ca filtre).

4.3.2. Principiul Substituției (Liskov)- PSL

Elementele cheie ale Principiului Inversării Dependențelor (PID- Secțiunea 4.3.3) sunt abstractizarea şi polimorfismul. Acestea sunt însă implementate prin relația de moştenire. Cum putem însă măsura calitatea moştenirii? Principiul substituției răspunde la această întrebare:

"Prin moştenire, orice ***proprietate adevărată despre obiectele supertipului*** *trebuie să rămână* ***adevărată despre obiectele subtipului."*** [Lisk88]

"Funcțiile ce folosesc ***pointeri/ referințe la clasele de bază*** *trebuie* ***să poată folosi obiecte ale claselor derivate*** *fără să ştie aceasta."* [Mart02]

Moştenirea *pare* simplă:

```
class Bird {
  public: virtual void fly();   //pasarile zboara...
};

class Parrot : public Bird {
   public: virtual void mimic();  //papagalii repeta cuvinte
```

```
};

//...

Parrot mypet;
mypet.mimic();  //poate mima, fiind papagal
mypet.fly();   //poate zbura, este o pasare
```

Dacă însă dorim să modelăm o clasă de păsări care nu zboară, (cum ar fi Pinguinii), situația va fi următoarea:

```
class Penguin : public Bird {
      public: void fly() {
           error ("Pinguinii nu zboara !"); }
};

void PlayWithBird (Bird& abird) {
      abird.fly(); //Ok daca este un papagal, daca este un pinguin- EROARE!
       }
```

Abordarea de mai sus nu modelează *"pinguinii nu pot zbura"*, ci *"pinguinii au voie să zboare, dar dacă o fac, rezultă eroare!"*. Aşadar, la rulare vor apare erori dacă o instanță de "pinguin" încearcă să zboare, iar aceste situații sunt de evitat.

Exemplu. Ne propunem să verificăm dacă este corect să derivăm Square din Rectangle. Pentru aceasta, ar trebui să răspundem la întrebarea "Pătratul ESTE-UN Dreptunghi?".

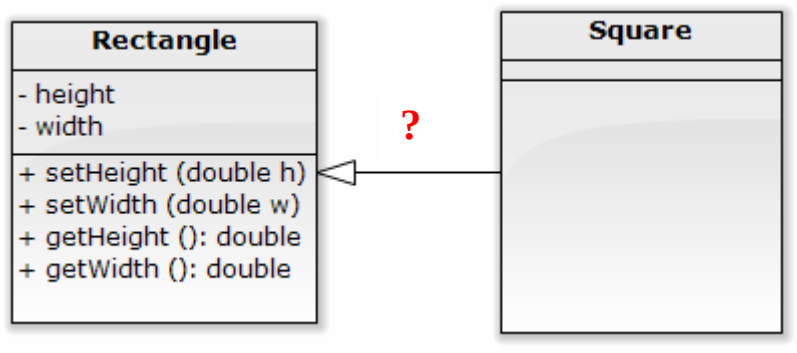


Figura 4.9. Exemplu derivare

Dacă în clasa de bază Rectangle avem metodele din Figura 4.9, acestea ar trebui suprascrise în clasa derivată cu versiuni în care la setarea lungimii, înălțimea ia aceeaşi valoare şi viceversa. Pentru a funcționa corect, aceste metode trebuie declarate virtuale în clasa de bază (pentru C++). Problema este însă următoarea: dacă un programator are nevoie de următoarea funcție:

```
void g(Rectangle& r)
```

```
{
r.SetWidth(5);
r.SetHeight(4);
assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

transmiterea unui parametru de tip Square va da o eroare de assert. Funcția a fost gândită pentru dreptunghiuri, aşadar deducem că din punctul de vedere al *comportamentului* aşteptat de aplicația respectivă, un Pătrat NU ESTE un Dreptunghi, deci derivarea nu este corectă! Concluzia este că la verificarea relației "ESTE-UN" trebuie ținut cont de comportamentul clasei în cadrul aplicației.

Aşadar, înainte de a proiecta o relație de moştenire trebuie înțelese foarte bine semantica şi scopul fiecărei metode şi clase, şi oriunde se face referire la o clasă în cod, orice subclase existente sau viitoare ale sale trebuie să fie 100% substituibile. Altfel spus, la modul ideal, orice cod ce poate chema legal metodele altei clase trebuie să poată substitui orice subclasă a clasei respective, fără modificări.

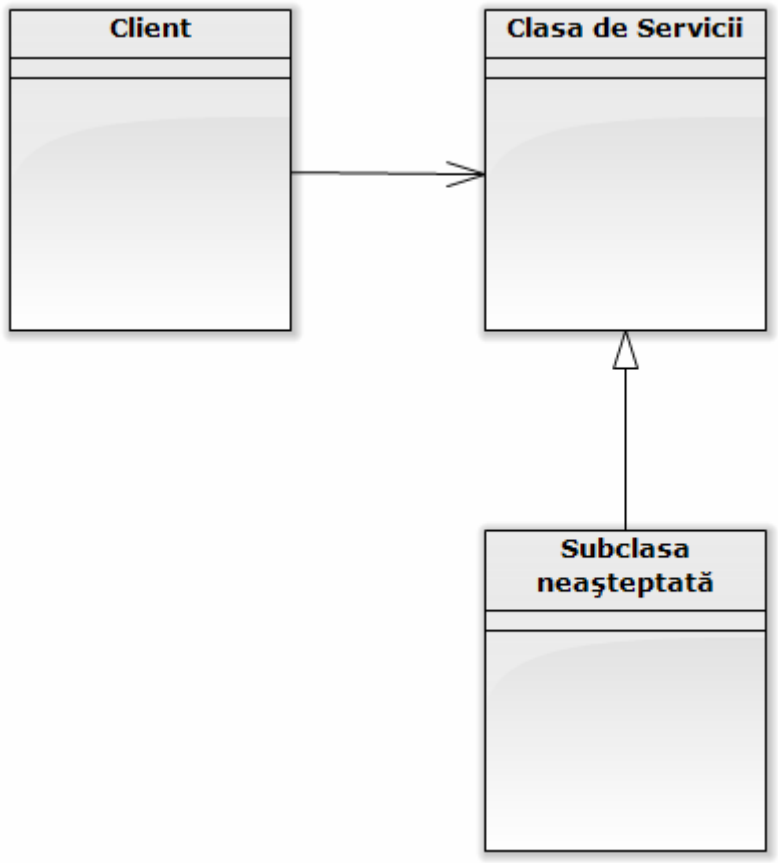


Figura 4.10. Codul Client trebuie să poată folosi metodele din Clasa de Servicii fără să fie influențat ulterior în vreun fel de apariția unei Subclase neaşteptate

Euristici PSL

1. În programarea prin contract (în care nu se fac verificări reciproce într-un apel de mesaj pentru că primitorul mesajului are încredere în transmițător că a respectat necesitățile sale -precondițiile-, iar transmițătorul are încredere că primitorul îşi respectă promisiunile anunțate –postcondițiile-) regula care ne ajută să respectăm Principiul Substituției a fost formulată de B. Meyer [Meye94]:

"*Când redefinim o metodă într-o clasă derivată, putem înlocui* precondiția *sa doar cu o* precondiție mai slabă, *şi* postcondiția *sa doar cu o* precondiție mai tare (i.e. mai restrictivă)".

Într-o formulare mai simplă, serviciile claselor derivate trebuie să nu ceară mai mult, şi să nu promită mai puțin decât versiunile lor din clasa de bază. Spre exemplu, dacă avem în clasa de bază metoda f cu următoarele pre- şi postcondiții:

```
int Base::f(int x);
//NECESITA:  x impar
//PROMITE: intoarce un rezultat intreg par
```

atunci o versiune corectă a sa într-o clasă derivată este următoarea:

```
int Derived::f(int x);
//NECESITA:  x intreg (mai putin restrictiv decat impar)
//PROMITE: intoarce valoarea 8 (o submultime a numerelor pare…)
```

2. *Este ilegal ca o clasă derivată să suprascrie o metodă a clasei de bază printr-o metodă care nu face nimic* (NOP No-OPeration). Există două soluții pentru a evita această situație: **crearea unei relații de derivare inversă** (dacă în clasa de bază inițială există doar comportament adițional: ex. cal - cal_care_nu_aleargă) sau **extragerea unei clase de bază comune**, dacă atât clasa inițială cât şi clasele derivate au comportamente diferite (spre exemplu, pentru Pinguini: Păsări, PăsăriZburătoare, Pinguini).

4.3.3. Principiul Inversării Dependențelor (PID)

"*Modulele de nivel-înalt **nu** trebuie să depindă de modulele de nivel scăzut. Ambele trebuie să depindă de abstracțiuni.*

Abstracțiunile nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstracțiuni." [Mart02]

Principiul spune că o clasă de bază într-o ierarhie nu trebuie să îşi ştie subclasele; modulele cu implementări detaliate trebuie să depindă de abstracțiuni, şi nimic nu trebuie să depindă de ele. Principiul Deschis-Închis este scopul, PID este mecanismul care realizează acest scop, iar Principiul Substituției este asigurarea că mecanismul implementat de PID va funcționa corect.

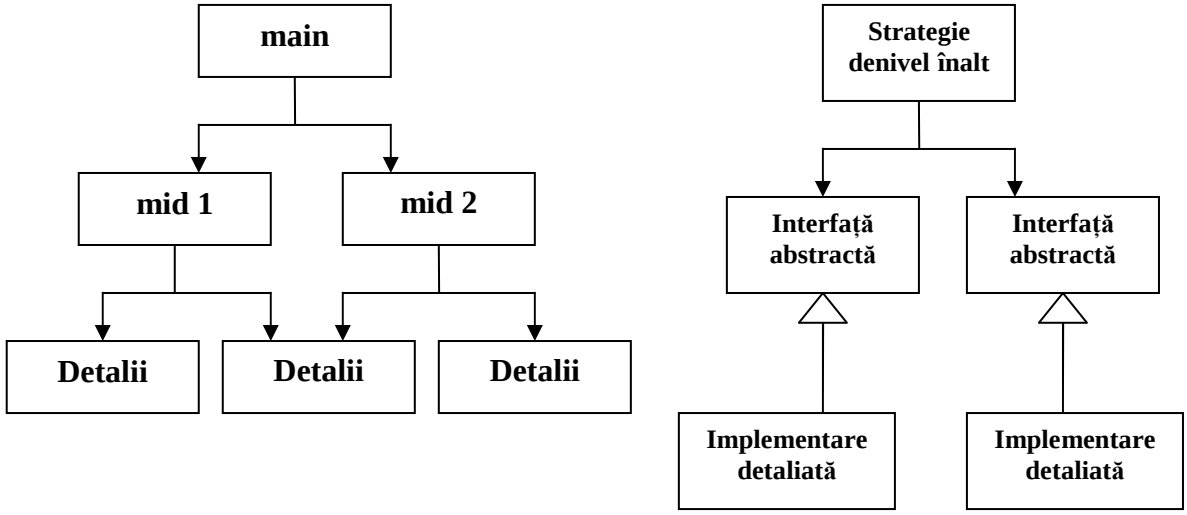


Figura 4.11. Arhitectura procedurala (stânga) versus Arhitectura OO (dreapta)

Exemplu [Mart02]. Un program citeşte de la tastatură caracter cu caracter şi scrie pe disc sau la imprimantă.

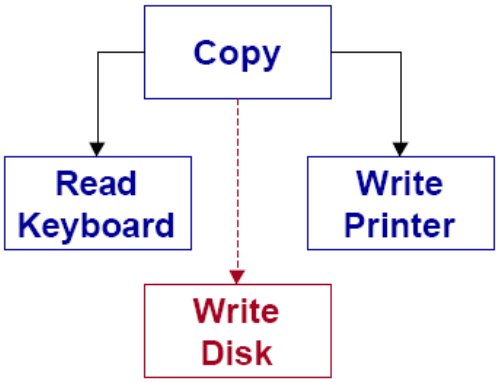


Figura 4.12. Varianta 1: Implementare imobilă şi rigidă

```
enum OutputDevice {printer, disk};
void Copy (OutputDevice dev) {
    int c;
    while ((c=ReadKeyboard())!=EOF)
           if(dev == printer)
```

```
                WritePrinter (c);
            else
                WriteDisk (c);
}

void Copy () {
      int c;
      while ((c=ReadKeyboard())!=EOF)
           WritePrinter(c);
}
```

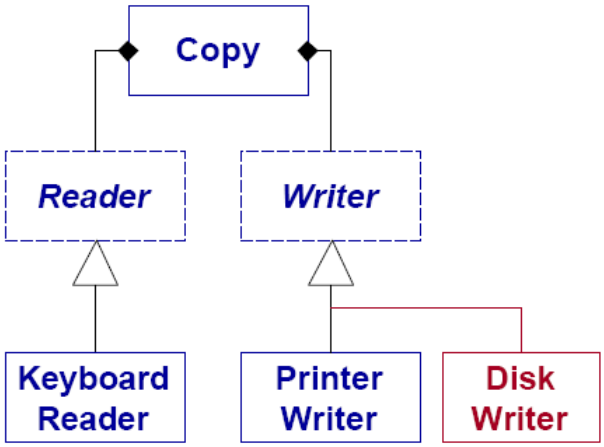


Figura 4.13. Varianta 2: Implementare cu aplicarea PID

```
class Reader {
    public:
        virtual int read()=0;
};

class Writer {
    public:
        virtual void write(int)=0;
}

void Copy(Reader& r, Writer& w) {
    int c;
    while ((c=r.read())!=EOF)
        w.write(c);
}
```

Euristici PID

A. *Proiectați centrat pe interfețe, nu pe implementare*. Utilizați moştenirea pentru a evita legarea directă de clase (Figura 4.14).

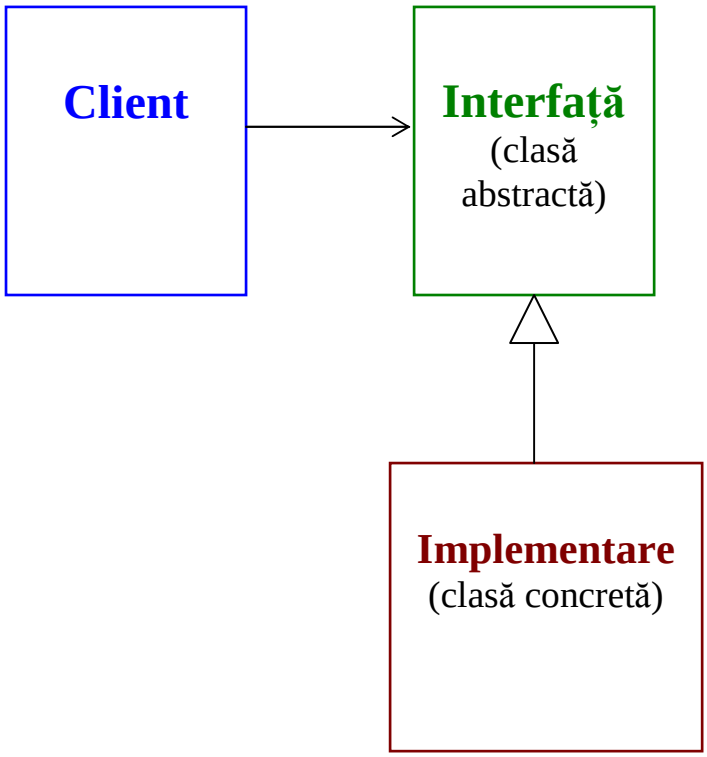


Figura 4.14. Evitarea legării directe

Avantajul proiectării centrate pe interfețe este dat de faptul că interfețele/ clasele abstracte tind să se schimbe mai rar, iar abstracțiunile sunt puncte de articulare în care este mai uşor de extins/ modificat. Teoretic, nu ar trebui să fie necesară modificarea abstracțiunilor (PDI). Există, desigur, şi excepții: unele clase este foarte puțin probabil să se schimbe, deci nu este avantajos să introducem stratul de abstractizare (spre exemplu, clasa String). În aceste situații, se foloseşte direct clasa concretă.

B. *Evitați dependențele tranzitive*. Evitați structurile în care straturi de nivel înalt depind de abstracțiuni de nivel scăzut. În exemplul de mai jos, stratul Strategie, este, în cele din urmă, dependent de stratul Servicii.

Figura 4.15. Dependențe tranzitive

Soluția pentru dependențele tranzitive este să utilizăm moştenirea şi clasele-ancestor abstracte pentru a le elimina.

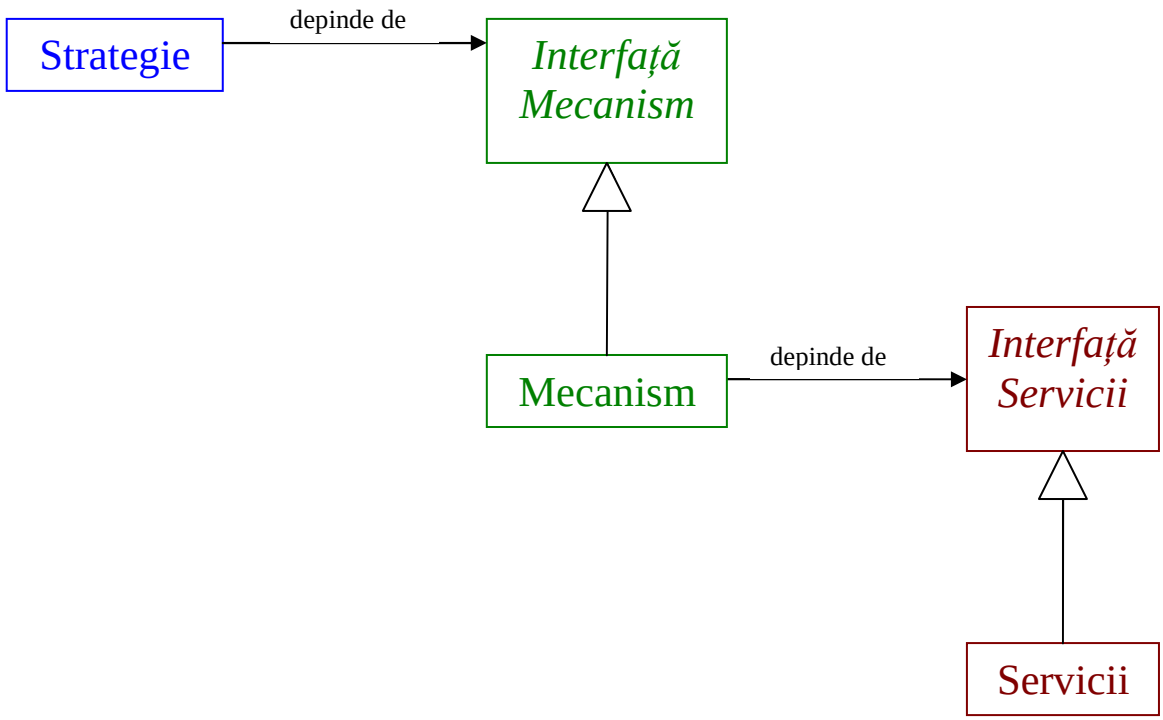


Figura 4.16. Eliminarea dependențelor tranzitive

C. *În situațiile neclare, adăugați un nivel de indirectare*. Dacă nu găsiți o soluție satisfăcătoare pentru clasa pe care o construiți, delegați responsabilitatea la una sau mai multe clase.

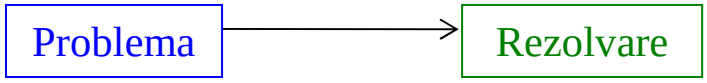


Figura 4.17

În situațiile neclare, este mai uşor să ştergem sau să evităm nivelele existente de indirectare, decât să le adăugăm mai târziu (Figurile 4.18 - 4.19). Apelurile indirecte din clasa albastră, de mesaje ale clasei roşii poate nu îndeplinesc anumite criterii (spre exemplu, constrângeri de timp real etc.) (Figura 4.18)

Figura 4.18. Dependență tranzitivă

Atunci clasa albastră poate reimplementa responsabilitățile pe care le folosește de la clasa verde, pentru a apela obiectul roşu direct (Figura 4.19).

Figura 4.19. O altă metodă de eliminare a dependenței tranzitive

***

Cele trei principii fundamentale sunt strâns relaționate. Astfel, încălcarea PSL sau PID va duce inevitabil la încălcarea PDI. Aceste principii sunt esențiale pentru a profita de avantajele dezvoltării orientate-obiect.

4.4. Principiul Segregării Interfețelor (PSI)

"Clienții nu trebuie forțați să depindă de interfețe pe care nu le folosesc." [Mart02]

Principiul Segregării Interfețelor afirmă că *este mai bine să existe mai multe interfețe specializate pe fiecare client (Figura 4.21), decât o interfață de interes general (Figura 4.20)*. Consecința va fi că impactul modificărilor unei interfețe va fi mai mic dacă interfața este mai mică.

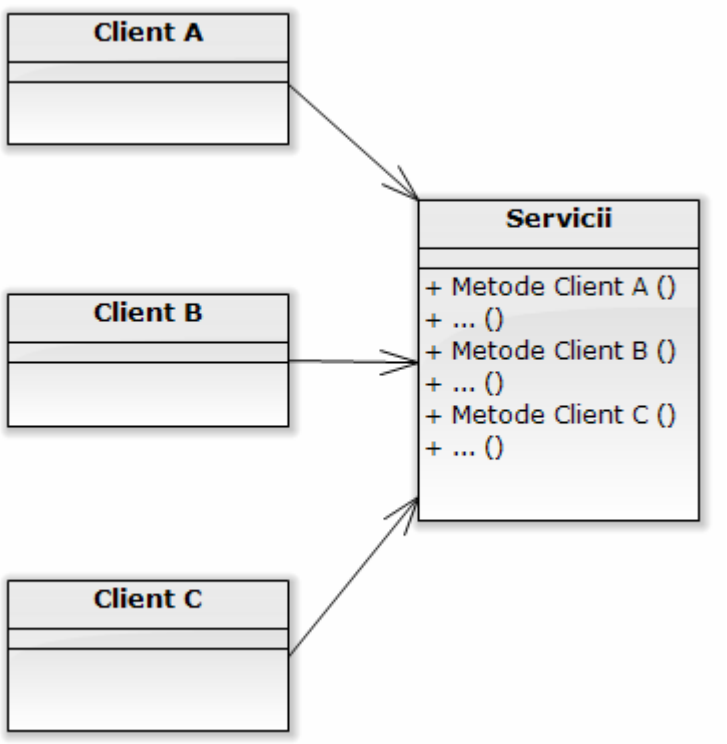


Figura 4.20. O interfață de interes general

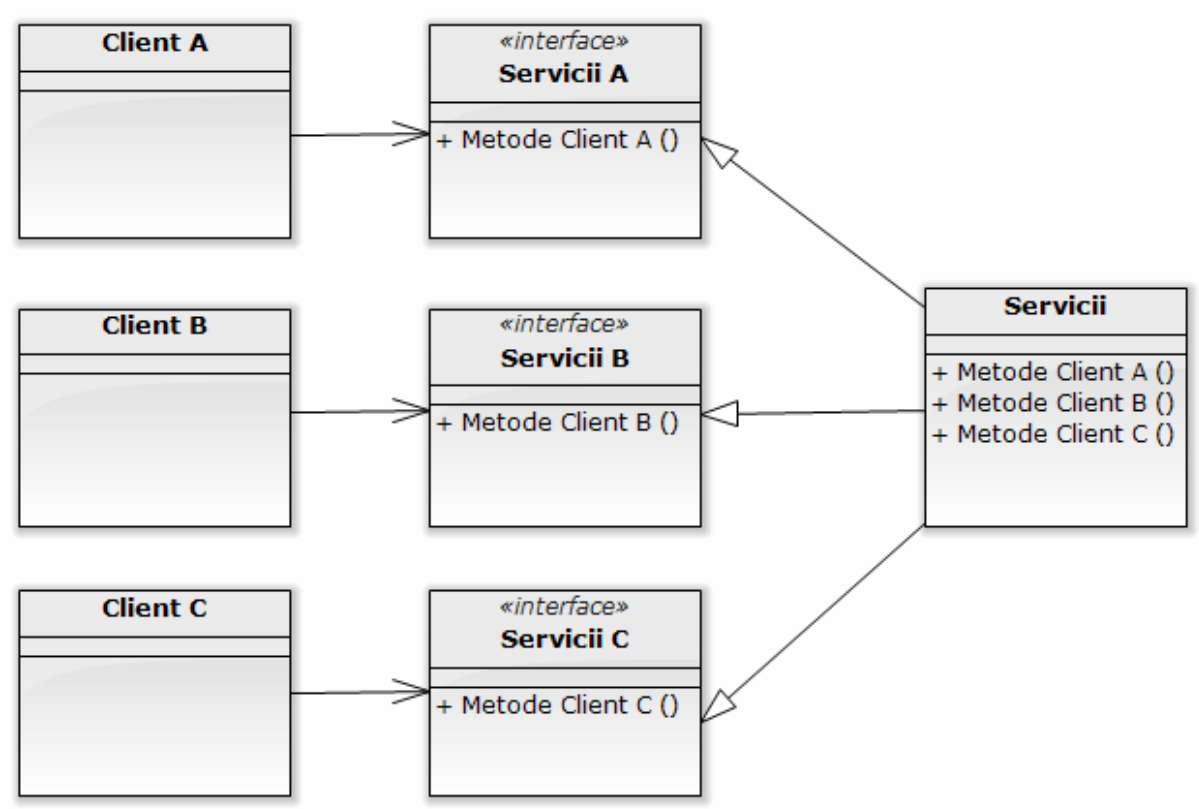


Figura 4.21. Interfețe specializate pe clienți

Exemple PSI

1.Uşa şi uşa temporizată [Mart02]

În implementarea din Figura 4.22, abordarea nu permite refolosirea caracteristicii de temporizare pentru alte elemente în afară de uşă. Rezolvarea situației este prezentată în Figura 4.23.

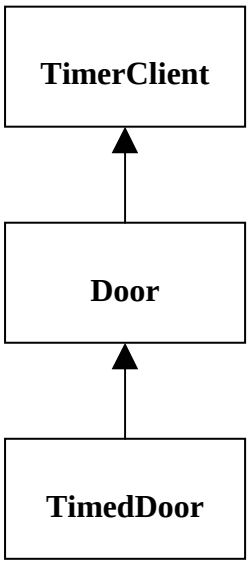


Figura 4.22. Uşa temporizată

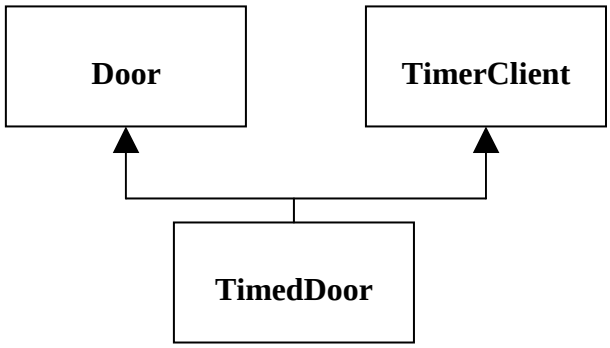


Figura 4.23. Separarea interfețelor prin moştenire multiplă

```
class TimedDoor : public Door, public TimerClient
     { public:
            virtual void TimeOut (int timeOutId); };
```

2. Ierarhia tranzacțiilor ATM [Mart02]

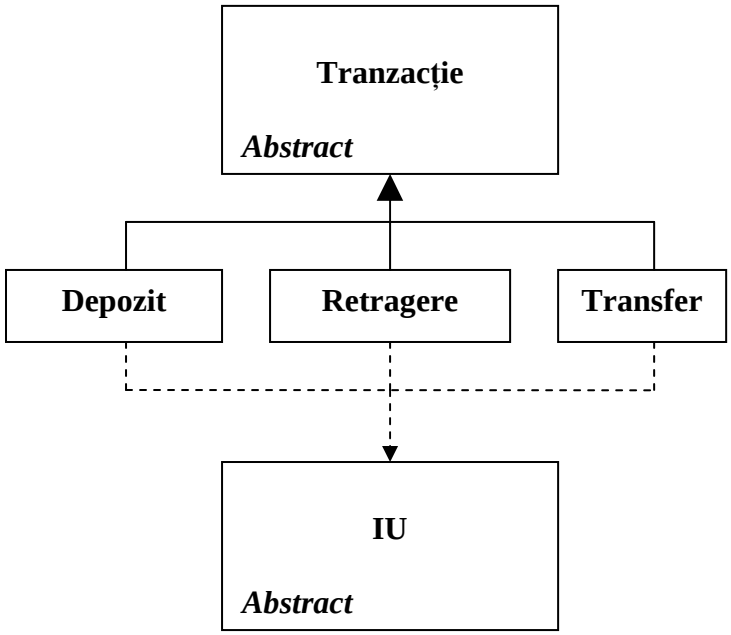


Figura 4.24.Tranzacții ATM

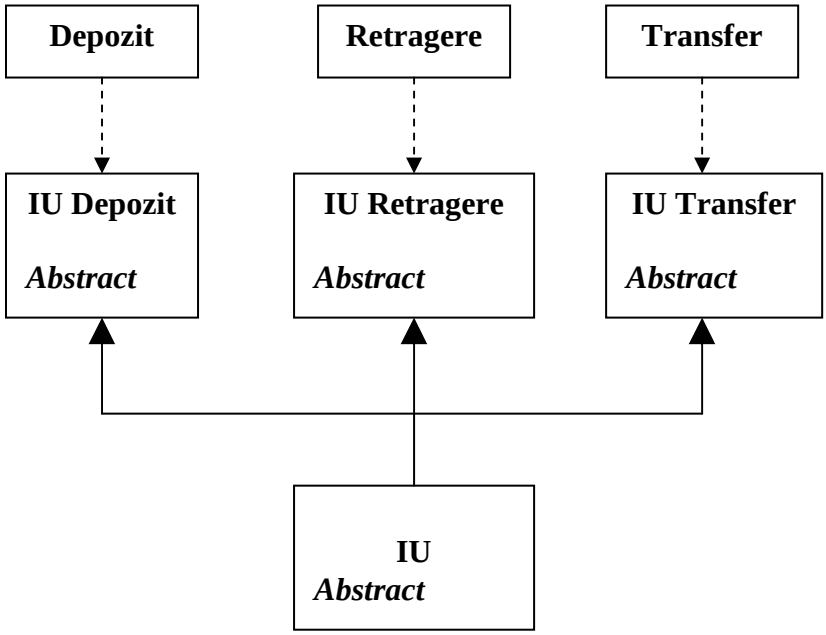


Figura 4.25. Tranzacții ATM- aplicare PSI

4.5. Definirea şi principiile design-ului de nivel înalt

Design-ul de nivel înalt priveşte *sistemele pe scară largă* (> 50000 linii de cod), construite de echipe de dezvoltatori, nu de un singur individ. Clasele sunt un mecanism de valoare, dar insuficient pentru organizarea design-ului unui sistem de dimensiuni mari, fiind prea mărunte. Este necesar un mecanism de impunere a ordinii la nivel mai înalt, şi vom folosi în continuare pachetele, dintre toate opțiunile existente. Un pachet este o grupare logică de declarații ce pot fi importate în alte programe (în Java şi Ada) sau un container pentru un grup de clase (UML), ce permite raționamentul la un nivel de abstractizare mai înalt.

Scopul design-ului de nivel înalt este *partiționarea* claselor unei aplicații conform unor *criterii* şi apoi *alocarea* acelor partiții unor pachete. Apar, astfel, următoarele întrebări:

- Care sunt criteriile optime de partiționare?
- Ce principii controlează design-ul pachetelor? (*crearea* şi *dependențele* dintre pachete)
- Trebuie să proiectăm mai întâi pachetele? Sau mai întâi clasele? (i.e.. *abordarea top-down* vs. *bottom-up*)

Vom prezenta în continuare principiile ce guvernează proiectarea pachetelor (creare şi relații între pachete). Principiile design-ului orientat-obiect de nivel înalt sunt de două tipuri: principii de coeziune şi principii de cuplaj (scopul fiind reducerea cuplajului şi sporirea coeziunii).

1. Principii de Coeziune
   - Principiul Echivalenței Refolosire/Folosire (REP)
   - Principiul Reutilizării Comune (PRC)
   - Principiul Închiderii Comune (PÎC)

2. Principii de Cuplaj
   - Principiul Dependențelor Aciclice (PDA)
   - Principiul Dependențelor Stabile (PDS)
   - Principiul Abstracțiunilor Stabile (PAS)

Un program este reutilizabil *dacă şi numai dacă nu avem nevoie să vedem codul sursă niciodată* [Mart02]. „Copy-Paste” nu înseamnă neapărat reutilizabilitate, întrucât codul respectiv trebuie depanat, eventual schimbat şi mentenanța sa va fi dificilă. Codul refolosit trebuie să poată fi tratat ca un *produs* de sine stătător, căruia nu este necesar să-i facem noi mentenanța. Astfel, clienții (cei care refolosesc) vor putea decide momentul folosirii unei versiuni mai noi pentru o anume componentă, când vor avea nevoie.

4.5.1. Principiul Echivalenței Refolosire/Folosire (PER)

„Granularitatea de refolosire este granularitatea de distribuție (livrare).
Doar componentele ce pot fi uşor urmărite (în cod, în versiunile ulterioare), pot fi refolosite eficient." [Mart02]

Principiul exprimă faptul că un element software reutilizabil nu poate fi refolosit cu adevărat în practică dacă nu este gestionat de un sistem de distribuție (*release system)* de un anumit tip (ex. numere sau nume asociate elementului refolosit). În practică, nu există "clasă reutilizabilă" fără garanția notificării, siguranței şi asistenței. Aşadar, trebuie integrat întregul modul (nu se poate reutiliza mai puțin). Deci *fie toate clasele unui pachet sunt reutilizabile, fie nici una nu este*! [R. Martin, 1996]

4.5.2. Principiul reutilizării comune (PRC)

„Toate clasele unui pachet [library] trebuie refolosite împreună. Dacă refolosim o clasă din pachet, le refolosim pe toate." [Mart02]

Principiul spune că pachetele de componente reutilizabile trebuie grupate după *utilizarea aşteptată*, nu după vreo funcționalitate comună, sau după orice altă clasificare arbitrară. De obicei, clasele sunt refolosite în grupuri pe baza colaborării între librării de clase (spre exemplu, containere şi iteratorii asociați lor). Trebuie să ținem însă cont că atunci când depindem de un pachet, depindem de fiecare clasă din acel pachet şi în consecință trebuie să grupăm într-un pachet clase ce pot fi refolosite împreună; altfel vom depinde de clase ce nu au legătură cu aplicația noastră, şi va trebui să re-compilăm aplicația la fiecare nouă versiune a pachetului de care depindem (chiar dacă versiunea modifică doar acele clase de care noi nu avem nevoie).

4.5.3. Principiul Închiderii Comune (PÎC)

„Clasele unui pachet trebuie să fie închise față de aceleaşi tipuri de modificări.
O modificare ce afectează pachetul, afectează toate clasele acelui pachet." [Mart02]

Principiul exprimă faptul că acele *clase care se modifică împreună trebuie grupate la un loc*, cu scopul limitării dispersării modificărilor între pachetele distribuite. Prin urmare, modificările trebuie să modifice un număr minim de pachete distribuite. Acest principiu este în strânsă relație cu Principiul Deschis- Închis întrucât ajută dezvoltatorul să definească ce parte a programului va rămâne stabilă, şi în raport cu ce tip de extinderi va fi stabilă. Altfel spus, clasele unui pachet trebuie să fie coezive şi pentru un anumit tip de modificare, fie se modifică toate clasele unei componente, fie nu se modifică nici una.

Refolosire vs. Mentenanță

Principiile PER şi PRC fac reutilizarea mai uşoară. Ele conduc la crearea de pachete foarte mici. Pe de altă parte, principiul PÎC face mentenanța mai uşoară şi conduce la crearea de pachete mari. Pachetele se pot însă modifica de-a lungul ciclului de viață al unui sistem, astfel că putem începe cu gruparea claselor ținând cont de PÎC (care este mai important la început) iar mai apoi, când dorim să reutilizăm putem regrupa respectând PER şi PRC.

4.5.4. Principiul dependențelor aciclice

„Structura de dependențe a componentei distribuite trebuie să fie un Graf Direcționat Aciclic (GDA). Nu trebuie să existe cicluri." [Mart02]

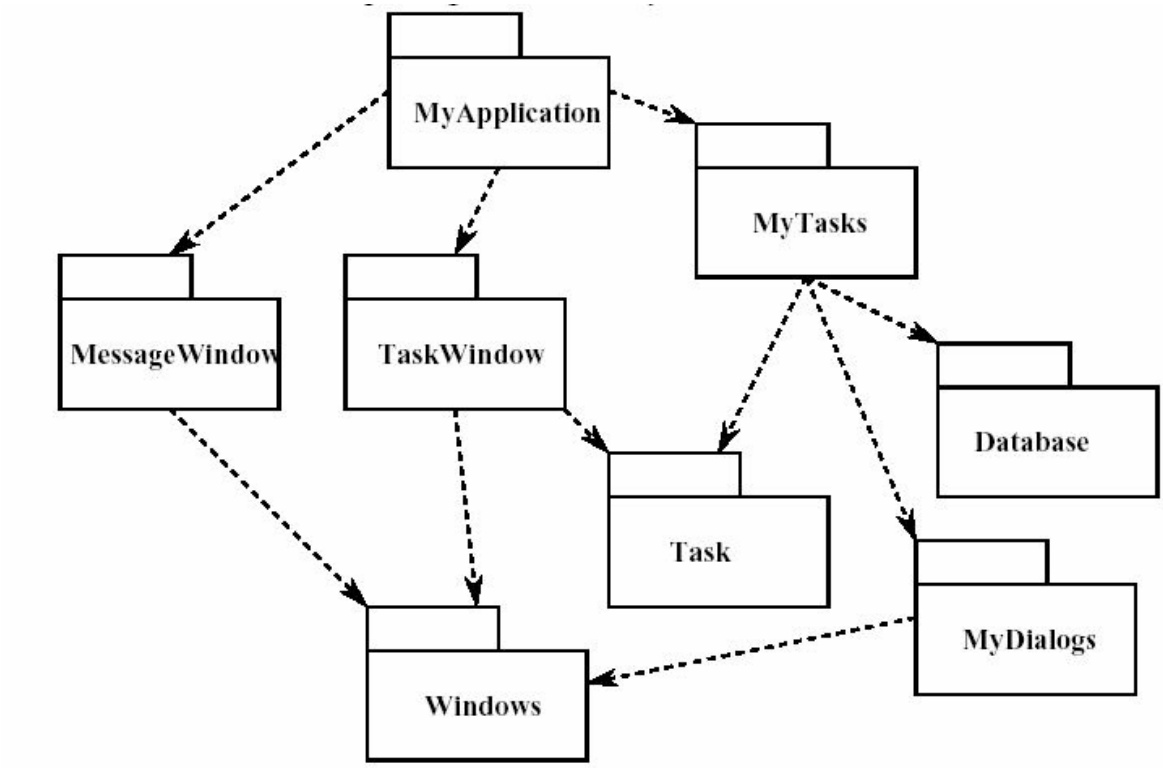


Figura 4.26. Graf aciclic de dependențe

Concluzia principală a acestui Principiu este că *structura de dependențe a pachetelor trebuie creată după scrierea codului*, nu înainte (Figura 4.26-4.28).

4.5.5. Principiul Dependențelor Stabile

„Dependențele între componente trebuie să fie în direcția stabilității.
O componentă trebuie să depindă de componente mai stabile decât ea.” [Mart02]

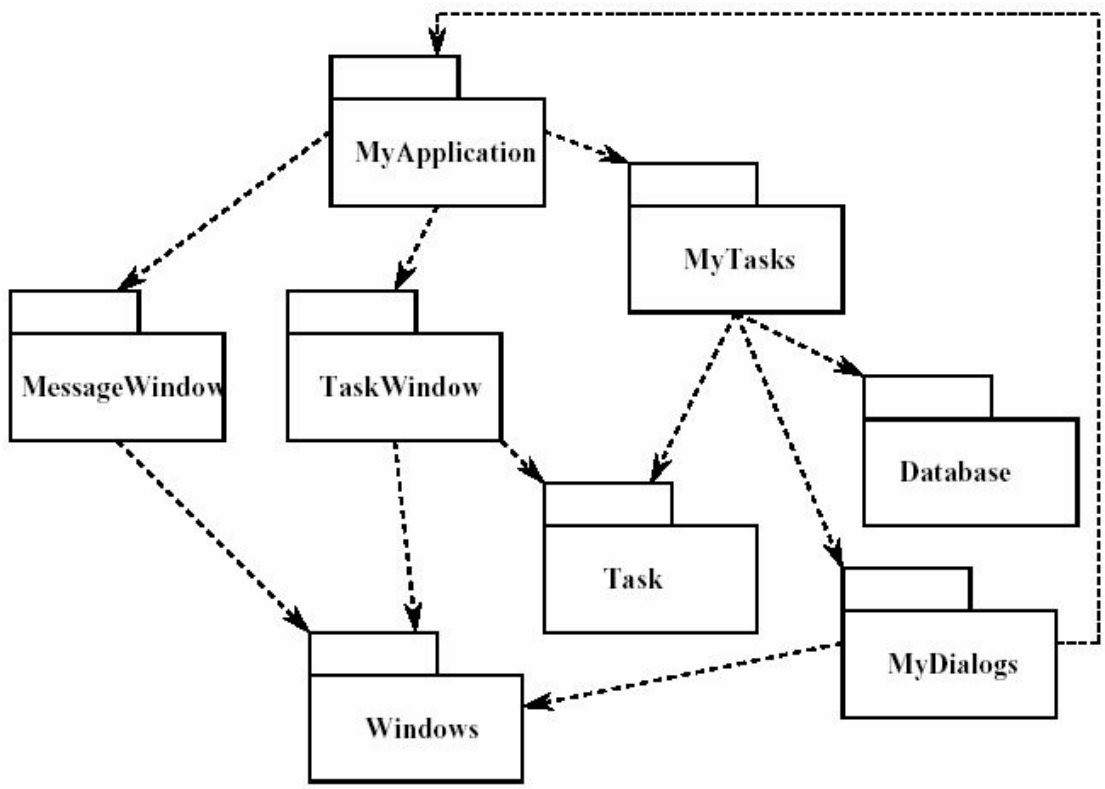


Figura 4.27. Graf ciclic de dependențe

Exemplu: Programul “Copy”

Să revenim asupra programului „Copy” de mai sus (Figura 4.29). Observăm că în programul menționat nu există interdependențe – ceea ce conferă robustețe, mentenabilitate şi reutilizabilitate codului, şi, pe de altă parte, țintele dependențelor inevitabile sunt non-volatile (este prea puțin probabil să se schimbe). Concret, clasele Reader şi Writer au volatilitate scăzută deci şi clasa Copy (care depinde de ele) este “imună” la modificări. În concluzie, o “Dependență Bună” este o dependență de o țintă cu volatilitate redusă.

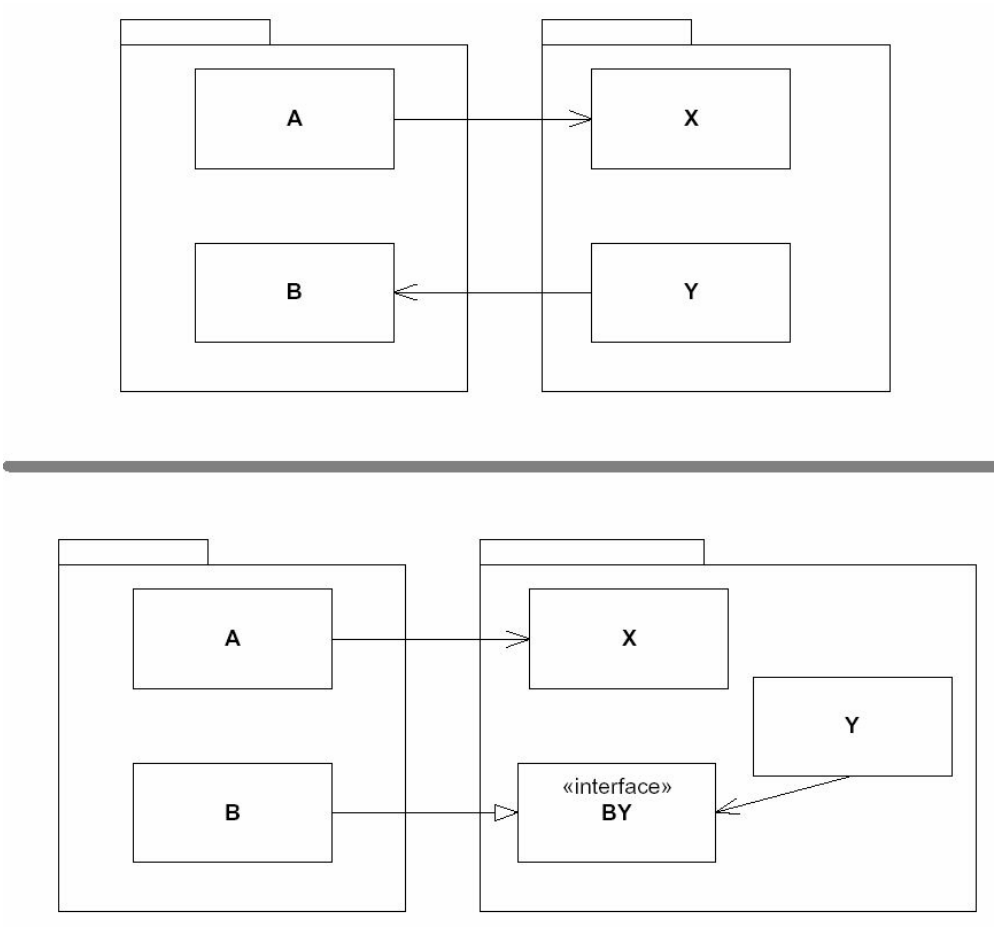
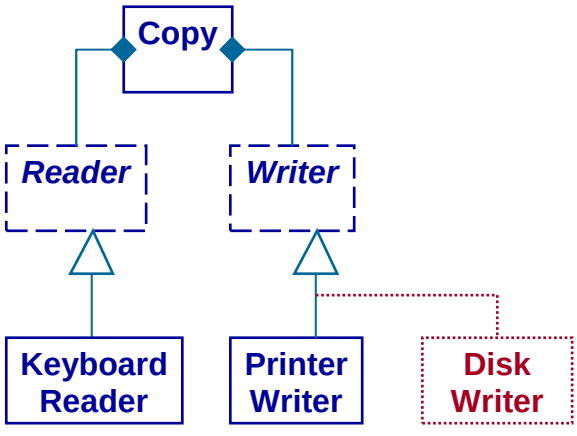


Figura 4.28. Spargerea Ciclurilor




```
class Reader {
  public:
    virtual int read()=0;
};

class Writer {
  public:
    virtual void write(int)=0;
};

void Copy(Reader& r, Writer& w){
   int c;
   while((c = r.read()) != EOF)
     w.write(c);
}
```

Figura 4.29. Programul “Copy”

Volatilitate şi Stabilitate

Factorii de Volatilitate (adesea dificil de înțeles şi de prezis) provin fie din codificarea unor informații improprii (ex. printarea numărului versiunii unui program) fie din presiunea pieței şi capriciile clienților. Opusul volatilității este *stabilitatea*, definită ca "dificil de mişcat". Stabilitatea poate fi văzută ca o măsură a dificultății schimbării unui modul (nu o măsură a probabilității unei schimbări!). Implicit, modulele stabile vor fi mai puțin volatile (ex. clasele Reader şi Writer sunt stabile). Stabilitatea poate fi măsurată prin anumite metrici dedicate, prezentate în continuare. Sintetic, putem considera formula de definiție:

Stabilitate = Responsabilitate + Independență

Metrici de stabilitate

- *Cuplajul Aferent* (**Ca**) = numărul de clase din afara acestui pachet care depind de pachetul măsurat ("*cât de responsabil sunt?*" -IN) ;

- *Cuplajul Eferent* (**Ce**) = numărul de clase din afara pachetului măsurat de care depind clase din interiorul acestui pachet ("*cât de dependent sunt?*" -OUT);

- *Factorul de Instabilitate* (**I**)
  - $I \in [0, 1] = Ce / (Ce + Ca)$;
  - 0 = total stabil; 1 = total instabil.

Exemplu: Calcularea metricilor de stabilitate. În Figura 4.30, pachetul din centru are un Ca=4 (4 clase externe depind de clase din interiorul său), şi un Ce=3 (depinde de 3 clase externe), deci I=3/(3+4)=3/7.

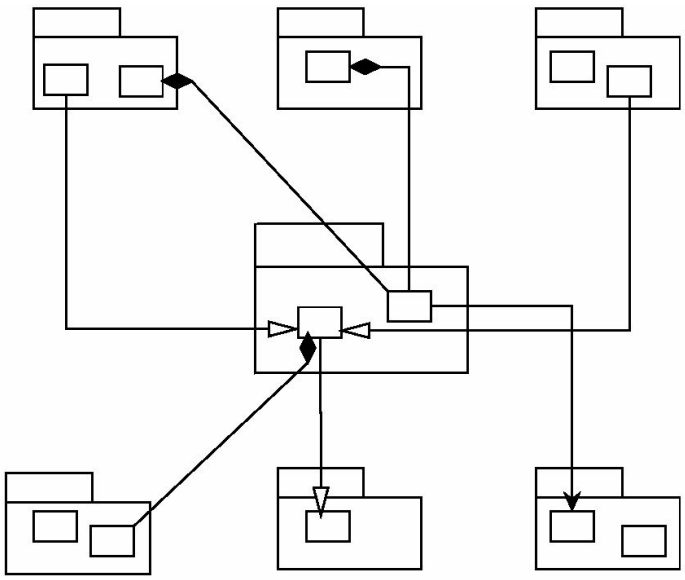


Figura 4.30. Exemplu de calcul a metricii de stabilitate

Principiul dependențelor stabile poate fi reformulat prin prisma metricii I astfel:

"Depinde numai de componente a căror metrică I este mai mică decât a ta!" [R. Martin, 1996]

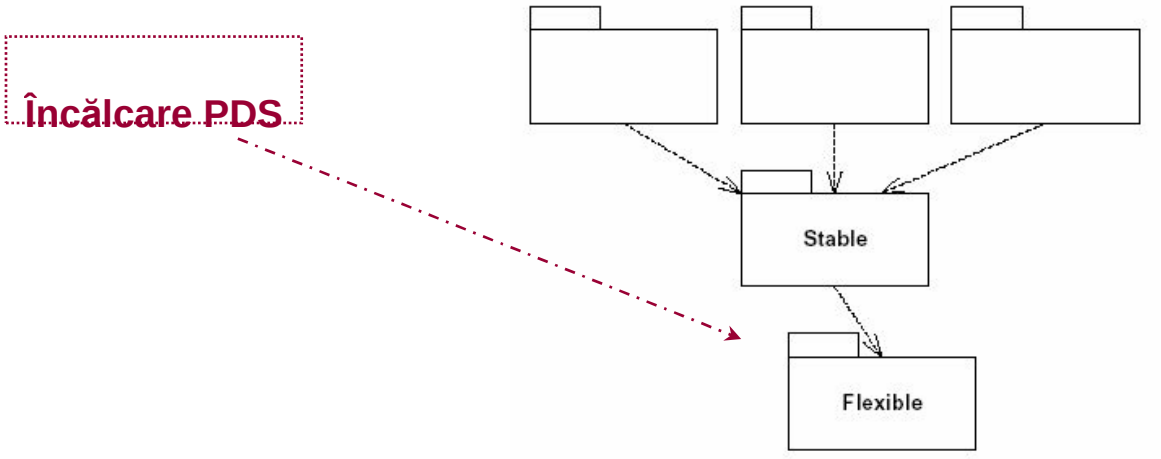


Figura 4.31. Exemplu de încălcare a PDS

Întrucât arhitectura şi deciziile de design de nivel înalt nu se modifică frecvent ar trebui plasate în pachete stabile. Însă un design greu de modificat este *inflexibil*, ceea ce nu este de dorit. Totuşi, un pachet total stabil (I = 0) poate fi suficient de flexibil pentru a face față schimbării folosind Principiul Deschis-Închis. Astfel, acest pachet ar trebui să conțină clase ce pot fi extinse fără să le modificăm- deci Clase Abstracte (dupa cum reiese şi din principiul următor).

4.5.6. Principiul Abstracțiunilor Stabile (PAS)

„Abstractizarea unui pachet trebuie să fie proporțională cu stabilitatea sa.
Pachetele maximal stabile trebuie să fie maximal abstracte.
Pachetele instabile trebuie să fie concrete." [R. Martin, 1996]

Arhitectura Ideală ar trebui să conțină pachetele instabile (modificabile) în partea superioară (ele trebuie să fie *concrete*) iar pachetele stabile (greu de modificat) în partea inferioară. Acestea din urmă trebuie să fie foarte *abstracte* (deci uşor de extins şi dificil de modificat).

Principiul Abstracțiunilor Stabile şi Principiul Dependențelor Stabile sunt, de fapt, o reformulare a Principiului Inversării Dependențelor la nivelul macro al pachetelor.

Măsurarea abstractizării unui pachet

Abstractizarea unui pachet se măsoară cu ajutorul unui număr $A \in [0, 1]$, unde valoarea 0 semnifică absența totală a claselor abstracte, iar valoarea 1 reprezintă situația în care toate clasele sunt abstracte.

$$A = \frac{NoAbstractClasse}{NoClasses}$$

Metrica *A* este imperfectă, dar utilizabilă. Clasele hibride (ce includ funcţii pur virtuale) sunt considerate abstracte (o alternativă ar fi găsirea unei metrici bazate pe fracţia de funcţii virtuale).

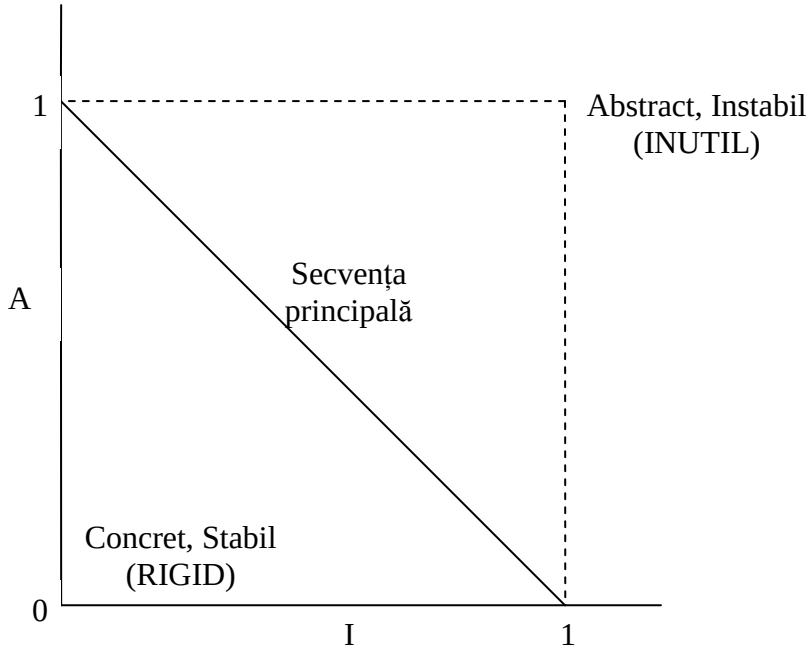


Figura 4.32. Secvența principală

"*Instabilitatea* (*I*) trebuie să crească pe măsură ce descreşte *abstractizarea* (*A*)." [Mart002]

Secvența Principală (Figura 4.32)

- Zona Rigidă
  - foarte stabil şi concret (deci rigid)
  - exemple:
    - schemele bazelor de date (volatile, se depinde foarte mult de ele)
    - librăriile de utilități concrete (instabile dar non-volatile)

- Zona de Inutilitate
  - instabil şi abstract (deci nefolositor)
    - nu depinde nimeni de clasele respective

- Secvența Principală
  - maximizează distanța între zonele de evitat

- reprezintă un echilibru între abstractizare şi stabilitate.

Măsurarea arhitecturilor orientate obiect

Pentru evaluarea arhitecturilor orientate obiect se foloseşte distanța:

$$D = \frac{|A + I - 1|}{\sqrt{2}}$$

Această metrică poate lua valori în domeniul [0, ~0.707]. În locul său se poate folosi şi distanța normalizată:

$$D = |A + I - 1|$$

Distanța normalizată ia valori în domeniul [0, 1]. Pachetele de pe secvența principală au distanța normalizată egală cu 0. Cu cât valoarea este mai apropiată de 1, cu atât pachetul este mai departe de secvența principală.

4.6. Legea Demetrei

Un principiu de design (şi în special de design orientat obiect) dezvoltat în anul 1987 şi care conduce la reducerea cuplajului şi la crearea de cod mai adaptabil şi mai uşor de menținut este Legea Demetrei [Crai05]. Legea este construită pe principiul cunoaşterii minime: fiecare modul trebuie să păstreze cât mai multe informații posibil ascunse (vezi şi regula de modularizare "Ascunderea Informației") şi să interacționeze cu un număr minim de alte module (vezi şi regula de modularizare "Interfețe Puține"- Capitolul 2), ideea fiind că eliminarea interacțiunilor protejează pe toată lumea. Principiul urmează exemplul contractor-ului general care trebuie să–şi manajerieze subcontractorii.

Legea Demetrei

- **Forma slabă**

Într-o metodă M a clasei C, datele pot fi accesate în şi mesajele pot fi trimise doar către obiectele următoare:

- this şi super
- datele membre ale clasei C
- parametrii metodei M
- obiecte create în interiorul M
  - prin apelul direct al unui constructor
  - prin apelul unei metode care creează obiectul
- variabile globale

- **Forma tare:**

*În afară de restricțiile Formei Slabă, este interzisă, în plus, accesarea membrilor moşteniți direct***.**

Exemplu. Orice metodă a unui obiect trebuie să apeleze doar metode aparținând:

```
class Demeter {
private:
  A *a;
public:
  // …
  void example(B& b);


void Demeter::example(B& b) {
  C *c;
  c = func();                  //  sieși

  b.invert();                  //parametrilor primiți

  a = new A();

  a->setActive();    //obiectelor create

  c->print();   //obiectelor componente deținute

}
```

Eliminarea încălcărilor LoD(Law of Demeter)

Exemplu Inițial

```
class Course {
Instructor boring = new Instructor();
int pay = 5;
public Instructor getInstructor() { return boring; }
public Instructor getNewInstructor() {return new Instructor(); }
public int getPay() {return pay; } }

class C {
Course test = new Course();

//metoda ce incalca Legea
public void badM() { test.getInstructor().fired();}


public void goodM() { test.getNewInstructor().hired(); }
public int goodM2() { return test.getpay() + 10;  }
}
```

Exemplu Îmbunătățit

```
class Course {
Instructor boring = new Instructor();
int pay = 5;
public Instructor fireInstructor() { boring.fired(); }
public Instructor getNewInstructor() { return new Instructor();}
public int getPay() { return pay ; }}

class C {
  Course test = new Course();

  //metoda ce nu mai incalcă Legea: nu mai apelează „fired” pentru o instanță
  //ce nu a fost creată de această clasă
  public void reformedBadM() { test.fireInstructor(); }

  public void goodM() {test.getNewInstructor().hired();  }
  public int goodM2() {return  test.getpay() + 10; }}
```

Avantajele aplicării Legii Demetrei

- *Controlul Cuplajului*: reduce cuplarea datelor;
- *Ascunderea Informațiilor*: previne accesul la părțile componente ale unui obiect;
- *Restricția Informațiilor*: restricționează utilizarea metodelor care oferă informații;
- *Interfețe Puține*: restricționează clasele care pot fi folosite într-o metodă;
- *Interfețe Explicite*: stabileşte explicit ce clase pot fi folosite într-o metodă.

Încălcări acceptabile ale Legii Demetrei

- Din motive de optimizare (restricții de viteză sau memorie)
- Dacă modulul accesat este o “cutie neagră” complet stabilizată (i.e., nu se aşteaptă, în mod rezonabil, modificări ale interfeței, datorită testării şi folosirii extensive, etc.)

În alte condiții, nu trebuie încălcată această regula, întrucât costurile pe termen lung vor fi mari.

Exerciții.

1. Calculați indicele de stabilitate ale pachetelor din exercițiile Capitolelor 6-8.

2. Verificați dacă ați respectat Legea Demetrei în exercițiile din Capitolele 6-8. Puteți optimiza codul astfel încât să o respectați?

3. Explicați ce principii de design OO sunt respectate prin aplicarea şabloanelor de design, la exercițiile Capitolelor 6-8.

CAP. 5. INTRODUCERE ÎN ŞABLOANE DE PROIECTARE

Motto: „Şabloanele te ajută să înveți din succesele altora, nu din eşecurile proprii”.(Mark Johnson)

Şabloane de învățare

Soluțiile de succes în multe arii ale cunoaşterii sunt adânc înrădăcinate în şabloane. Învățarea programării de calitate seamănă cu învățarea şahului: întâi se învață regulile de joc (în cazul programelor, este vorba de algoritmi, structuri de date şi limbaje de programare), apoi se învață principiile ce guvernează diferite paradigme de programare (structurată, modulară, orientată-obiect, etc), dar pentru a stăpâni design-ul software, trebuie studiat design-ul altor maeştri care conține *şabloane* ce trebuie înțelese, memorate şi aplicate repetat (aşa cum în şah sunt studiate partidele marilor maeştri pentru a descoperi şi a înțelege anumite scheme de gândire strategică).

Un şablon este [Alex79] *„o regulă cu trei părți ce exprimă o relație între un anumit context, o problemă şi o soluție*”: *„Fiecare şablon descrie o problemă care apare în mod repetat în mediul nostru, şi apoi descrie nucleul soluției acelei probleme, într-un mod care permite utilizarea repetată a acestei soluții, de fiecare dată însă în mod diferit”*. Un şablon introduce un nivel de abstractizare suplimentar în program, similar design-ului, care *separă lucrurile schimbătoare de cele neschimbătoare*, permițând extragerea factorilor comuni unei familii de probleme similare. Şablonul reprezintă un mod isteț şi aprofundat de rezolvare a unei *clase de probleme* particulare, soluția cea mai generală şi mai flexibilă.

Exemplu: Şablonul Observer

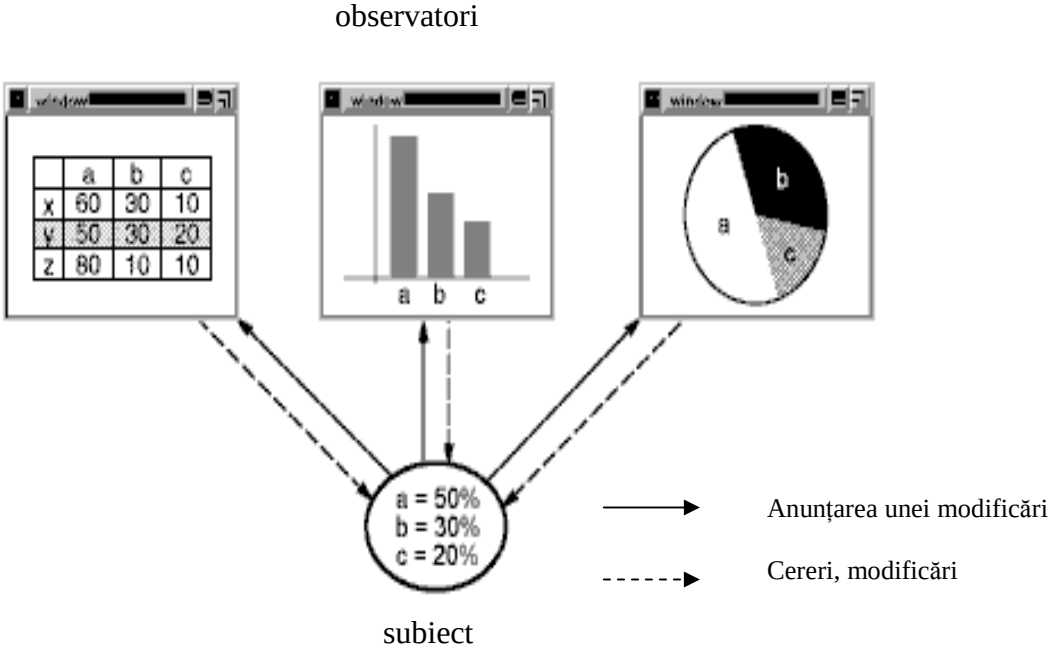


Figura 5.1. Şablonul Observer rezolvă problema consistenței Date-Vederi

- **Scop**
  - Definiți o dependență unul-la-mai-mulți între obiecte astfel încât atunci când un obiect îşi schimbă starea, toate dependențele sale sunt anunțate şi actualizate automat;

- **Forțe** (i.e. factori ce modelează specificul şablonului)
  - Pot exista mulți observatori;
  - Fiecare observator poate reacționa diferit la aceeaşi notificare;
  - Sursa de date (subiectul) trebuie să fie cât mai decuplată posibil de observator(i) (pentru a permite observatorilor să se modifice independent de subiect).

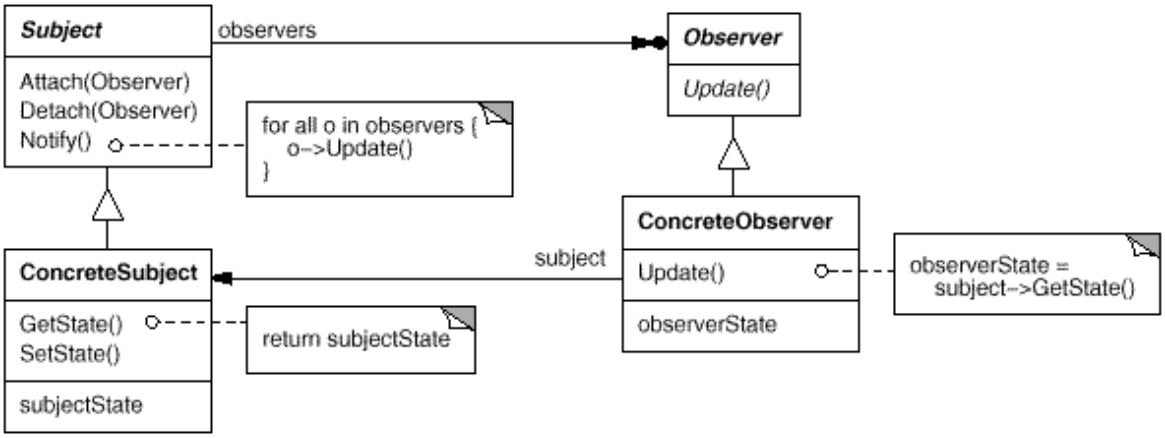


Figura 5.2. Structura şablonului Observer

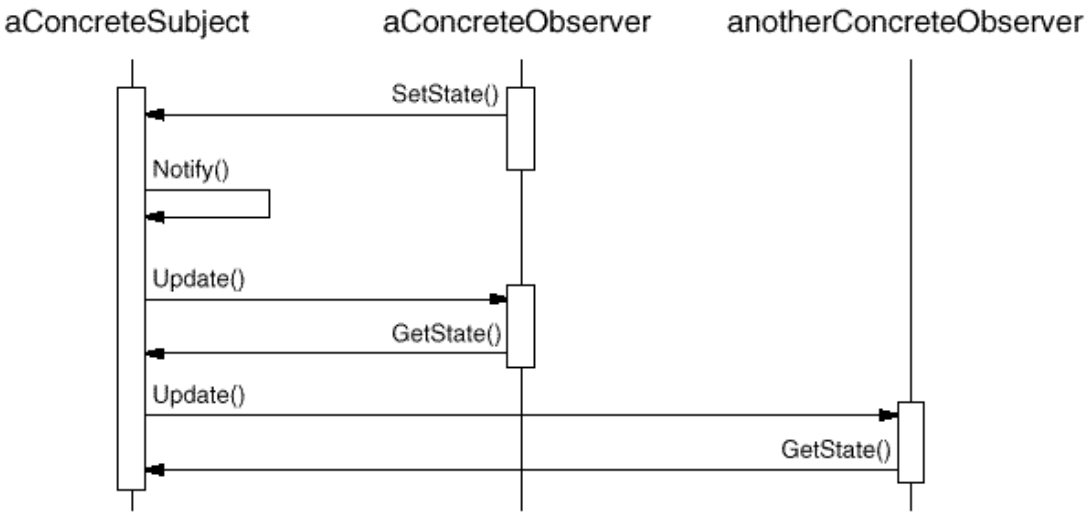


Figura 5.3. Colaborări în şablonul Observer

Pentru a verifica dacă soluția unei probleme de design reprezintă cu adevărat un şablon, avem la dispoziție o listă de verificare a trăsăturilor definitorii. Astfel, un şablon trebuie să îndeplinească următoarele criterii:

1. **să rezolve o problemă** (i.e Să fie util)
2. **să aibă un context** (i.e. Să descrie unde se poate aplica soluția)
3. **să reapară** (i.e. Trebuie să fie relevant şi în alte situații)
4. **să îndrume** (i.e. Să ofere suficientă înțelegere pentru ajustarea soluției)
5. **să aibă un nume** (Referit consistent)

Numele şablonului şi clasificarea	
Scop	ce face şablonul
Cunoscut şi ca	alte nume cunoscute ale şablonului (dacă este cazul)
Motivație	problema de design
Aplicabilitate	situații în care şablonul poate fi aplicat
Structură	o reprezentare grafică a claselor din şablon
Participanți	clasele/obiectele participante şi responsibilitățile lor
Colaborări	ale participanților pentru a rezolva responsabilitățile
Consecințe	compromisuri, preocupări
Implementare	idei, tehnici
Exemplu de cod	fragment de cod oferind o posibilă implementare
Utilizări cunoscute	şabloane din sistemele reale
Şabloane înrudite	şabloane strâns înrudite

Tabel 5.1.Forma unui Şablon de Proiectare [Gamm94]

Forma Algoritmică a Şabloanelor

```
DACĂ   eşti în CONTEXT
     de exemplu EXEMPLU,
     cu PROBLEMA,
     implicând FORŢELE
ATUNCI pentru anumite MOTIVE,
     aplică O FORMĂ ŞI/SAU REGULĂ DE DESIGN
     pentru a construi SOLUŢIA
     conducând la un CONTEXT NOU şi la
               ALTE ŞABLOANE
```

Şabloanele de proiectare a programelor (*design patterns*) sunt privite ca perechea Problemă/Soluție într-un Context (ele reprezintă soluții la probleme ce apar când dezvoltăm software într-un context particular). Acestea surprind structura statică dar şi pe cea dinamică şi colaborarea între participanții cheie în proiectarea software (participant cheie– abstracțiune majoră ce apare într-o problemă de proiectare). Un avantaj în plus al şabloanelor este acela că facilitează reutilizarea arhitecturilor software şi de design de succes.

Moştenirea Clasei vs. Moştenirea Interfeței

O clasă defineşte o anumită implementare, în timp ce un tip defineşte doar interfața (i.e. mulțimea de cereri la care poate răspunde un obiect). Clasa implică tipul, dar reciproc nu este valabil (pot exista mai multe clase cu acelaşi tip). Atunci când se moşteneste o clasă, practic se descrie o implementare în termenii altei implementări, însă atunci când se moşteneşte un tip, un obiect poate fi folosit în locul altuia.

Principiul de Design 1[Gamm94]: *Programați raportat la o interfață, nu la o implementare.*

Practic, aceasta se realizează astfel:

- Folosiți interfețe pentru a defini interfețele comune (şi/sau clase abstracte în C++)
- Declarați variabile ca instanțe ale clasei abstracte (nu instanțe ale claselor particulare)
- Folosiți *Şabloane Creaționale* pentru a asocia interfețele cu o implementare

Avantajul acestei abordări este că *reduce* masiv *dependențele de implementare*: obiectele client nu cunosc clasele ce implementează obiectele folosite, ci doar clasele abstracte (sau interfețele) ce definesc interfața.

Moştenirea claselor vs. Compunere

Există două mecanisme de refolosire în programarea orientată-obiect: refolosire white-box (moştenirea clasei) vs. refolosire black-box (compunerea obiectelor). Moştenirea claselor presupune refolosirea implementării şi are în spatele său legarea statică (deci care nu se poate schimba la execuție). În plus, amestecul de reprezentări fizice ale datelor este o încalcare a încapsulării, şi orice modificare în părinte va

determina modificări în subclasă. Pe de altă parte, în cadrul compunerii obiectelor, obiectele sunt accesate doar prin interfețele lor şi nu se încalcă încapsularea. Mai mult, orice obiect poate fi înlocuit de altul la execuție, cât timp au acelaşi tip.

Principiul de Design 2[Gamm94]: *Preferați compunerea moştenirii claselor.*

Pentru a facilita respectarea acestui principiu este recomandat să păstram clasele centrate pe o singură sarcină. În cadrul refolosirii prin moştenire se creează componente noi prin modificarea componentelor vechi, deci design-urile vor fi mai refolosibile dacă depind mai mult de compunerea obiectelor.

Clasificarea Şabloanelor de Proiectare

1. **Şabloane Creaționale:** se ocupă de inițializarea şi configurarea claselor şi obiectelor (*Cum îmi voi crea obiectele?);*

2. **Şabloane Structurale:** decuplează interfatța şi implementarea claselor şi obiectelor (*Cum sunt compuse clasele şi obiectele pentru a construi structuri mai mari?);*

3. **Şabloane de Comportament:** organizează interacțiunile dinamice între grupuri de clase şi obiecte (*Cum gestionăm fluxurile de control complexe (comunicații)?).*

	Creaționale	**Structurale**	**Comportamentale**
Clasă	▪ Metoda Factory (Constructor virtual)	▪ Adapter (Adaptor)	
Obiect	▪ Abstract Factory ▪ Builder (Constructor) ▪ Prototype (Prototip) ▪ Singleton	▪ Adapter (Adaptor) ▪ Bridge (Punte) ▪ Composite (Compozit) ▪ Decorator ▪ Flyweight ▪ Proxy	▪ Chain of Responsibility (Lanț de responsabilități) ▪ Command (Comandă) ▪ Iterator ▪ Mediator ▪ Memento ▪ Observer (Observator) ▪ State (Stare) ▪ Strategy (Strategie) ▪ Visitor (Vizitator)

Tabel 5.2. Catalogul Şabloanelor de Proiectare [Gamm94]

Beneficii ale Şabloanelor de Design	▪ Inspirație ▸ *Şabloanele nu oferă soluții, ele inspiră soluții;* ▸ Şabloanele capturează explicit cunoştinţele experţilor şi compromisurile de design şi fac această expertiză accesibilă pe scară largă; ▸ Şabloanele uşurează tranziția către tehnologia orientată-obiect; ▪ Şabloanele îmbunătățesc comunicarea între dezvoltatori ▸ Numele şabloanelor formează un vocabular; ▪ Şabloanele ajută la documentarea arhitecturii unui sistem ▸ Îmbunătățesc înțelegerea; ▪ Şabloanele permit reutilizarea pe scară largă a arhitecturilor software
Dezavantaje ale Şabloanelor de Design	▪ Şabloanele nu conduc la refolosirea directă a codului; ▪ Poate apare o supra-încărcare cu şabloane; ▪ Integrarea şabloanelor într-un proces de dezvoltare software este o activitate intens-umană.

Tabel 5.3 Avantaje şi dezavantaje ale şabloanelor

CAP. 6. ŞABLOANE CREAŢIONALE

Şabloanele creaționale abstractizează procesul de instanțiere astfel ca sistemul să devină independent de cum sunt create, compuse şi reprezentate obiectele sale. Există două tipuri de şabloane creaționale: şabloane creaționale de tip clasă, ce folosesc moştenirea pentru a varia clasa instanțiată (*Factory Method*) şi şabloane creaționale tip obiect, care delegă instanțierea către alt obiect (*Abstract Factory*, *Prototype*, *Singleton*, *Builder*).

Exemplul 1.

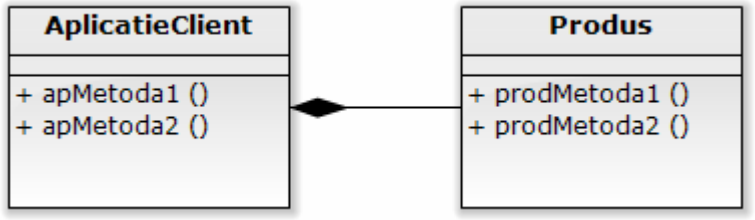


Figura 6.1

Etapa 1. Putem modifica codul intern al clasei Produs fără să modificăm AplicatieClient:

```
class AplicatieClient   {
   Produs a;
   Produs b;

    public apMetoda1()   {
      Produs d = new Produs ();
      d.prodMetoda1();
      // . . .
      Produs e = new Produs ();
      // . . .   }

    public apMetoda2()   {
      // . . .
      Produs f = new Produs ();
      f.prodMetoda1();
      // . . .
      Produs g = new Produs ();
      // . . .}
```

În varianta de mai sus vom avea serioase probleme cu modificările, dacă, spre exemplu, descoperim un nou "produs" şi dorim să îl folosim în AplicatieClient. Între Produs şi AplicatieClient avem cuplaj multiplu:

- AplicatieClient ştie interfaţa Produs;
- AplicatieClient foloseşte explicit tipul Produs (şi deci va fi dificil de schimbat pentru că Produs este o clasă concretă);
- AplicatieClient creează explicit noi Produse în multe locuri, aşadar dacă dorim să folosim noul Produs în locul celui iniţial, modificările sunt răspândite în tot codul.

Etapa 2. Vom aplica principiul "Programează orientat pe Interfeţe" şi vom avea structura din Figura 6.2.

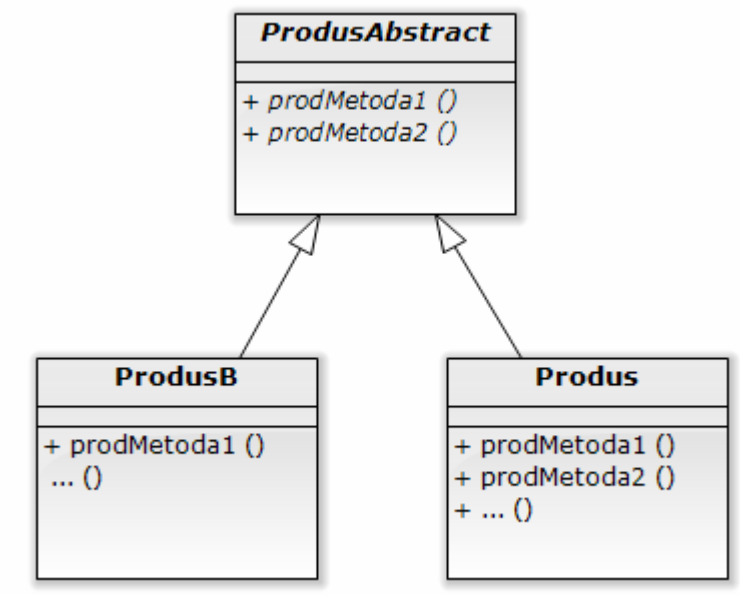


Figura 6.2

```
class AplicatieClient   {ProdusAbstract a;
   ProdusAbstract b;

   public apMetoda1()   {ProdusAbstract d = new Produs();
      d.prodMetoda1();
      // . . .
      ProdusAbstract e = new Produs();
      …
  }…}
```

AplicatieClient depinde acum de o interfaţă (abstractă) dar încă am implementat ce produs să creez.

Exemplul 2. Să considerăm programul Labirint [Gamm94]. Componentele labirintului sunt Pereţi, Uşi şi Camere, iar MapSite este o clasă abstractă comună tuturor Componentelor Labirintului (Figura 6.3):

```
enum Direction {North, South, East, West};
```

```
class MapSite {
public:
   virtual void enter() = 0;
};
```

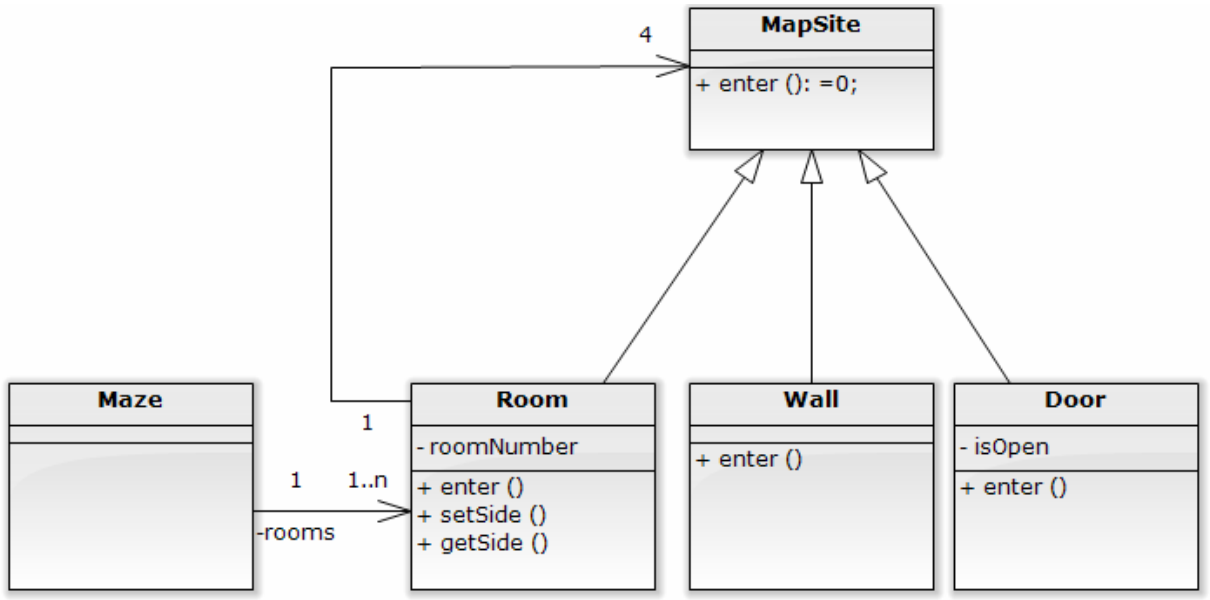


Figura 6.3. Diagrama de clase pentru Labirint

Înțelesul functiei *enter()* depinde de *unde* se intră: dacă este o cameră *enter* duce la modificarea locației, dacă este o uşă şi aceasta este deschisă se poate intra; altfel nu se întâmplă nimic.

```
class Room : public MapSite {
public:
   Room(int roomNo);
   MapSite* getSide(Direction) const;
   void setSide(Direction, MapSite*);

void enter();

private:
   MapSite* sides[4];
   int roomNumber;
}

class Wall : public MapSite {
public:
   Wall();
   virtual void enter();
};

class Door : public MapSite {
public:
```

```
    Door(Room* = 0, Room* = 0);
    virtual void enter();
    Room* otherSideFrom(Room*);

private:
    Room* room1;
    Room* room2;
    bool isOpen;
};
```

Componentele labirintului sunt asamblate în clasa Maze (Labirint):

```
class Maze {
public:
    void addRoom(Room*);
    Room * roomNo(int) const;
private:
};
```

Aşadar, un labirint este o colecţie de camere. Clasa "Maze" poate găsi o cameră anume fiind dat numărul camerei: funcţia *roomNo()* face o verificare folosind căutare lineară, o funcţie hash sau un tablou.

Desfăşurarea jocului

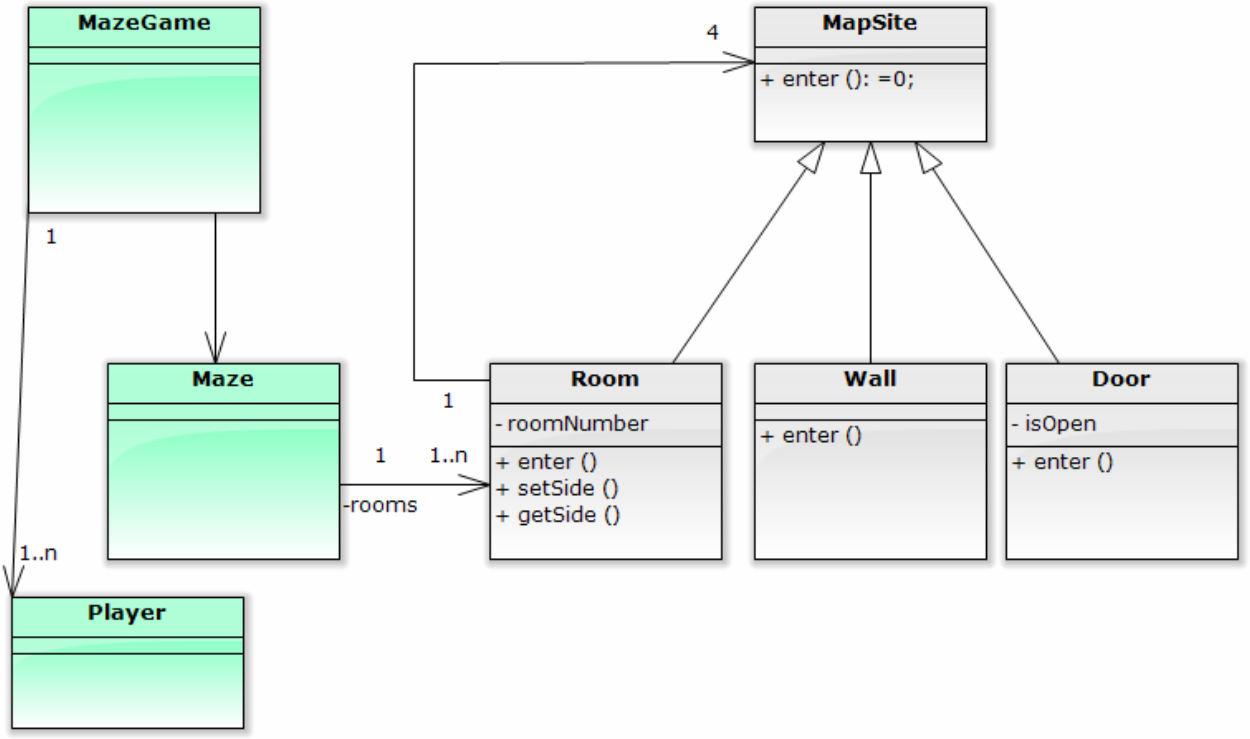


Figura 6.4. Diagrama de clase extinsă

În prezentarea şabloanelor creaționale vom studia crearea labirintului folosind funcția *createMaze()* a clasei MazeGame:

```
Maze* MazeGame::createMaze() {
   Maze* aMaze = new Maze;
   Room* r1 = new Room(1);
   Room* r2 = new Room(2);
   Door* theDoor = new Door(r1, r2);
   aMaze->addRoom(r1);
   aMaze->addRoom(r2);

   r1->setSide(North, new Wall);  r1->setSide(East, theDoor);
   r1->setSide(South, new Wall); r1->setSide(West, new Wall);

   r2->setSide(North, new Wall);  r2->setSide(East, new Wall);
   r2->setSide(South, new Wall); r2->setSide(West, theDoor);
}
```

Problema rezolvării anterioare este inflexibilitatea, care provine din codificarea explicită în *createMaze()* a aspectului labirintului. Dacă dorim flexibilitate în crearea labirintului, în sensul de a putea varia tipurile de labirinturi (camere cu bombe, camere speciale etc.), o primă soluție este programarea orientată spre interfețe (ca în Exemplul 1): derivăm MazeGame şi suprascriem *createMaze()*:

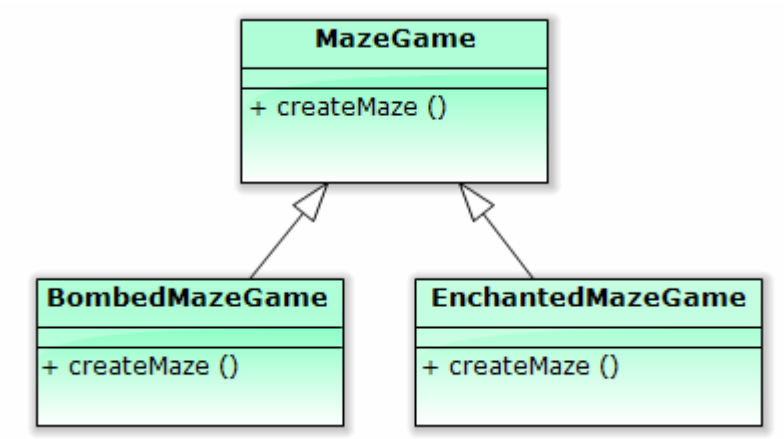


Figura 6.5.

```
Maze* BombedMazeGame::createMaze() {
   Maze* aMaze = new Maze;
   Room* r1 = new RoomWithABomb(1);
   Room* r2 = new RoomWithABomb(2);
   Door* theDoor = new Door(r1, r2);
   aMaze->addRoom(r1);
   aMaze->addRoom(r2);

   r1->setSide(North, new BombedWall);
```

```
    r1->setSide(East, theDoor);
    r1->setSide(South, new BombedWall);
    r1->setSide(West, new BombedWall);
    // etc.
}
```

Dezavantajul acestei soluții este că vom avea mult cod duplicat (versiunea de *createMaze()* din clasa BombedMazeGame diferă de cea din clasa EnchantedMazeGame numai prin numele constructorilor apelați).

6.1. Metoda FACTORY (Constructor Virtual)

Scop şi aplicabilitate

Atunci când o clasă nu poate anticipa clasa obiectelor pe care trebuie să le creeze si doreşte ca subclasele sale să specifice obiectele create, se defineşte o interfață pentru crearea unui obiect, iar instanțierea efectivă este deferită subclaselor.

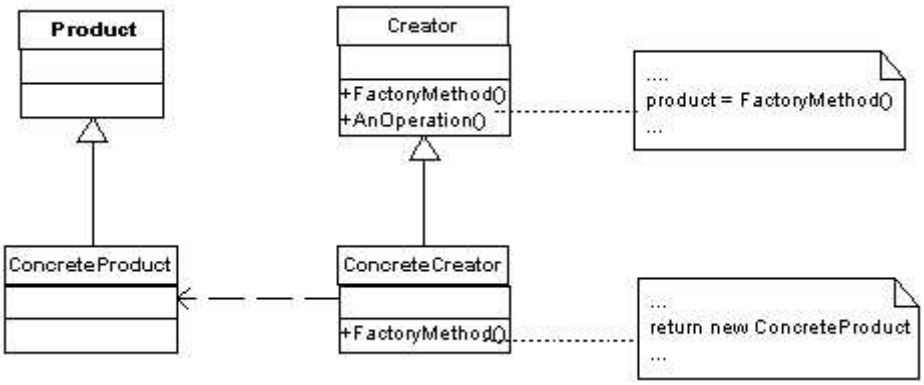


Figura 6.6. Structura şablonului Metoda Factory

Participanți & Colaborări [Gamm94- Fig. 6.6]

- Product
  - defineşte interfața obiectelor ce vor fi create de MF (Metoda Factory);
  - Produs Concret implementează interfața;
- Creator
  - declară MF, care returnează un produs de tip Product.
    - poate defini o implementare implicită a MF;
    - poate apela MF pentru crearea unui produs;
- ConcreteCreator

    - suprascrie MF pentru a furniza o instanță a ConcreteProduct;
- *Creator se bazează pe subclasele sale pentru definirea metodei factory astfel încât să returneze instanța potrivită a clasei ConcreteProduct.*

Consecințe [Gamm94]

Avantaje:

- Elimină legarea între clasele specifice aplicației din cod (codul de creare foloseşte doar interfața clasei Product);
- Facilitează derivarea (şi subclasele pot modifica produsul creat);
- Poate conecta *ierarhii paralele de clase*, lăsând clienții să apeleze MF .

Dezavantaje:

- Clienții ar trebui să deriveze din Creator doar pentru a crea un anumit obiect ConcreteProduct.

Detalii de implementare [Gamm94]

- Varietăți ale Metodelor Factory
    - Clasa Creator este abstractă
        - nu implementează MF declarată;
        - necesită subclase;
        - folosită pentru instanțierea claselor neprevăzute;
    - Clasa Creator este concretă
        - creează o implementare implicită;
        - MF folosite pentru flexibilitate;
        - creează obiecte într-o operație separată astfel încât subclasele o pot suprascrie;
- Parametrizarea Metodelor Factory
    - o variație a şablonului lasă metodele factory să creeze mai multe tipuri de produse;
    - un *parametru* identifică tipul de Product de creat;
    - toate obiectele create au aceeaşi interfață cu Product.

Caz special de Factory

Un caz special, simplificat al şablonului, care evită crearea de subclase, foloseşte următoarea abordare: nu se derivează clasa client, ci se creează o clasă Factory separată, cu o metodă de creare a diferitelor subclase în funcție de un parametru primit, iar clasa client va fi în relație de compunere cu aceasta. Spre exemplu, un program desenează 100 de forme geometrice alegând aleator între cerc şi pătrat (Figura 6.7).

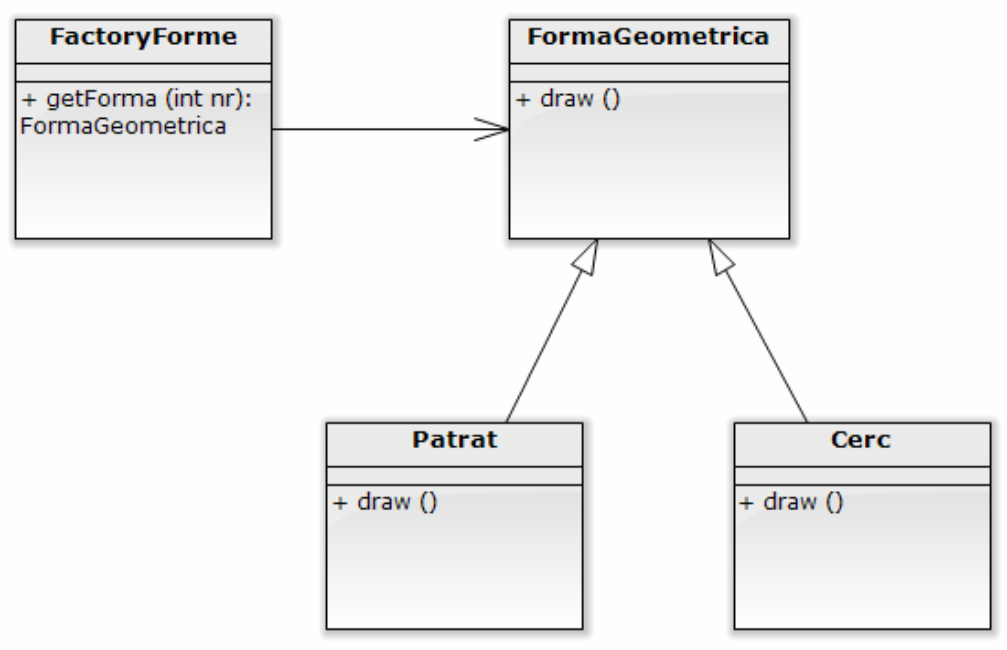


Figura 6.7

```
FormaGeometrica * getForma (int nr)
{if (nr%2 ==0) return new Cerc();
 else  return new Patrat();}
...
```

Utilizarea Factory:

//în codul clasei client:

```
FactoryForme  ff=new FactoryForme();
FormaGeometrica * fg;

for (int i =1; i<=100; i++)
       {fg=ff.getForma(i);
        s.draw();}
```

Codul client este astfel stabil faţă de adăugarea de noi forme geometrice. Extinderea ierarhiei va determina modificări numai în codul metodei *getForma* din clasa Factory (am localizat modificările într-o singură clasă).

Exemplul 1 (reluare). Folosim şablonul creaţional Metoda Factory. Se creează clasa AplicatieClientB în care se copiază codul din AplicatieClient, cu diferenţa că în loc de apelul constructorului *Produs* se va apela *ProdusB()* (codul duplicat este un dezavantaj al acestei metode).

```
class AplicatieClient   {
   ProdusAbstract a;
   ProdusAbstract b;
```

În C++ funcţia următoare se declară virtuală şi se folosesc pointeri la obiectul tip AplicatieClient

```
public ProdusAbstract creazaProdus() {return new Produs();}

public apMetoda1()   {
         ProdusAbstract d = creazaProdus();
         d.prodMetoda1();
         // . . .
         ProdusAbstract e = creazaProdus();
      // . . .}
   // . . .}

class AplicatieClientB extends AplicatieClient {
public ProdusAbstract creazaProdus() {return new ProdusB();}}
```

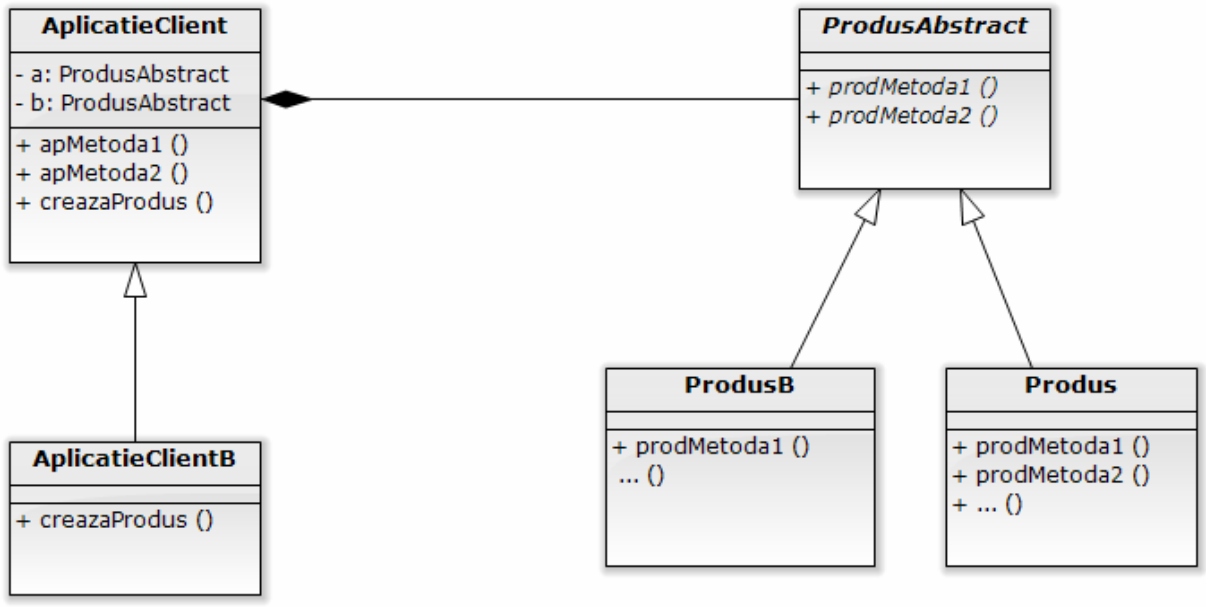


Figura 6.8.

În altă parte a codului se poate folosi secvența:

```
AplicatieClient test = new AplicatieClientB();
test.apMetoda1();
ProdusAbstract aw=test.creazaProdus();  aw.prodMetoda();…
```

Utilizând Metoda Factory, crearea explicită de obiecte Produs nu mai este dispersată (deci codul este mai uşor de schimbat). În plus, metodele funcționale din AplicatieClient sunt decuplate de diversele implementări concrete ale mecanismelor, şi se evită duplicarea inestetică de cod din AplicatieClientB:

subclasele refolosesc metodele funcționale, doar implementând metoda *Factory* concretă necesară. Dezavantajele acestei abordari sunt că trebuie sa creem o subclasă doar pentru a suprascrie Metoda Factory şi că nu putem modifica Produs la execuție.

Exemplul 2. Soluția Problemei Labirintului (reluare). Folosim Metoda Factory, după structura din Figura 6.9.

Clasa MazeGame şi metoda *createMaze()* devin:

```
class MazeGame {
public: Maze* createMaze();        // metode factory:

virtual Maze* makeMaze() const
{ return new Maze; }

virtual Room* makeRoom(int n) const  { return new Room(n); }

virtual Wall* makeWall() const
{ return new Wall; }

virtual Door* makeDoor(Room* r1, Room* r2) const
{ return new Door(r1, r2); }};

Maze* MazeGame::createMaze ()
    {Maze* aMaze = makeMaze();
      Room* r1 = makeRoom(1); Room* r2 = makeRoom(2);
      Door* theDoor =makeDoor(r1, r2);
       aMaze->addRoom(r1);  aMaze->addRoom(r2);
       r1->SetSide(North, makeWall());  r1->SetSide(East, theDoor);
       r1->SetSide(South, makeWall()); r1->SetSide(West, makeWall());
       r2->SetSide(North, makeWall()); r2->SetSide(East, makeWall());
       r2->SetSide(South, makeWall()); r2->SetSide(West, theDoor);
       return aMaze;
}
```

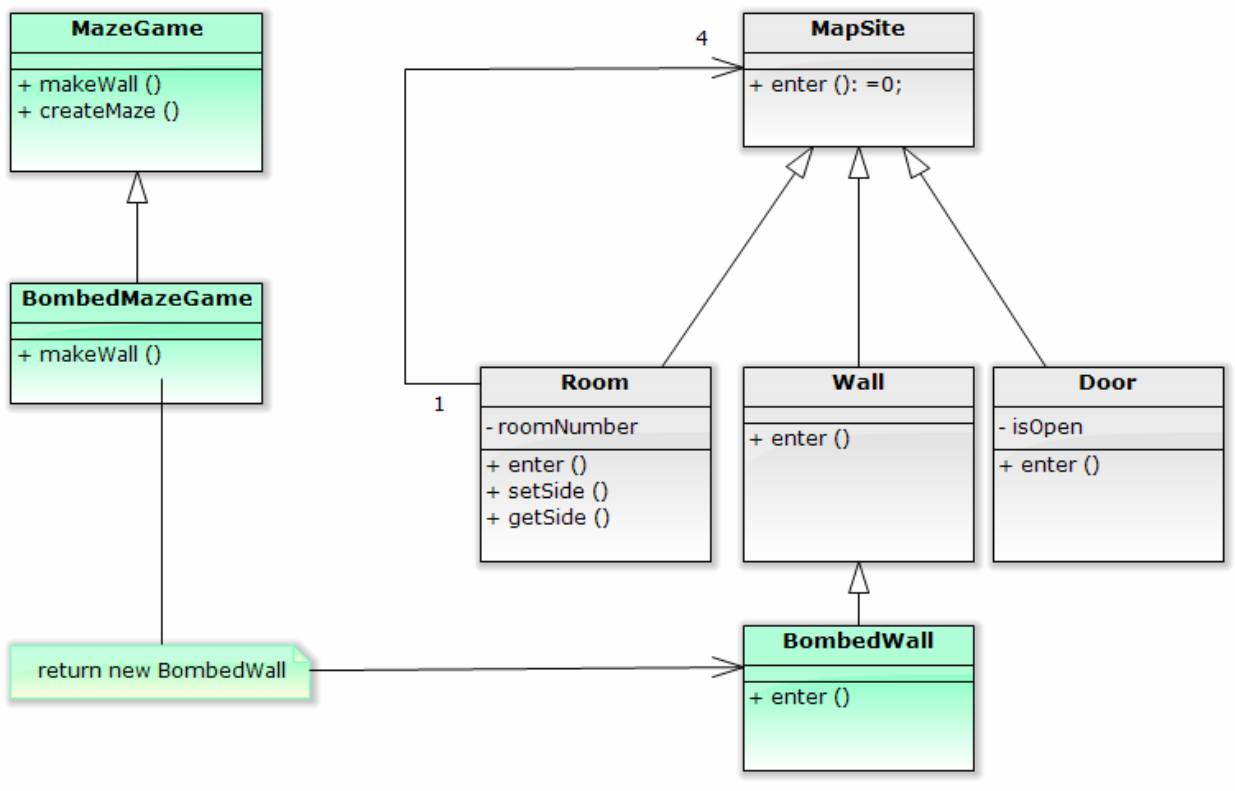


Figura 6.19. Labirint cu Metoda Factory

O altă abordare ar fi să creem o Metodă Factory în Product (produsul devine responsabil de propria creare). Spre exemplu, putem să lăsăm clasa Door să ştie cum să-şi construiască instanțele, în locul clasei MazeGame. Clientul produsului va necesita o referință la "creator" (specificată în constructor, după cum se observă în codul următor).

```
class Room : public MapSite  {
  public:
   virtual Room* makeRoom(int no)
{ return new Room(no); }
   // ...};

class RoomWithBomb : public Room {
  public:
   Room* makeRoom(int no)
{ return new RoomWithBomb(); }
   // ...
};
// ...

class MazeGame {
  protected:
   Room* roomMaker;
   // ...
```

```
  public:
   MazeGame(Room* rfactory) {
     roomMaker = rfactory;}

public Maze* createMaze() {
     Maze aMaze = new Maze();
  Room r1 = roomMaker->makeRoom(1);
     // ...
};
```

Exemplul 3 [Gamm94]. Conectarea Ierarhiilor Paralele de Clase

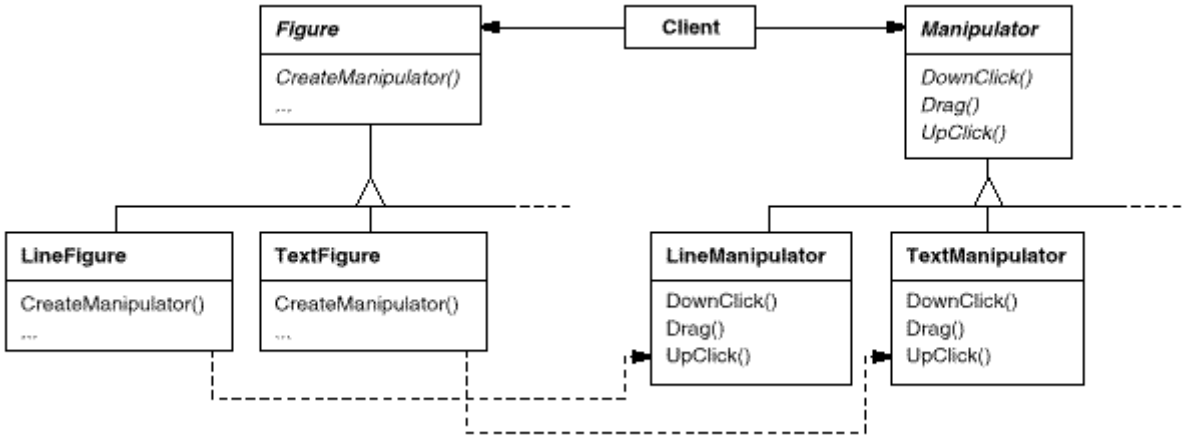


Figura 6.10

Observăm că informaţia despre clasele ce aparţin împreună este localizată.

Exerciţii

1. Tetris. Creaţi cele 5 forme necesare jocului Tetris folosind şablonul Metoda Factory (Figura 6.11). Formele sunt create în funcţie de un parametru care ia valori aleatoare pe parcursul jocului.

Figura 6.11. Tetris

2. În jocul Hungry Hippau (Cap. 1, exercițiul 3), creați 5 feluri de fructe folosind Factory.

6.2.Şablonul PROTOTYPE (Prototip)

Scop şi aplicabilitate

Şablonul Prototip este util atunci când instanțierea unei clase este consumatoare de timp sau complexă într-un anumit fel: în loc de a crea mai multe instanțe, se creează obiecte noi prin copierea unei instanțe originale (prototipul), şi aceste copii se modifică după necesități. În cadrul acestui şablon clasele de instanțiat sunt *specificate la execuție.*

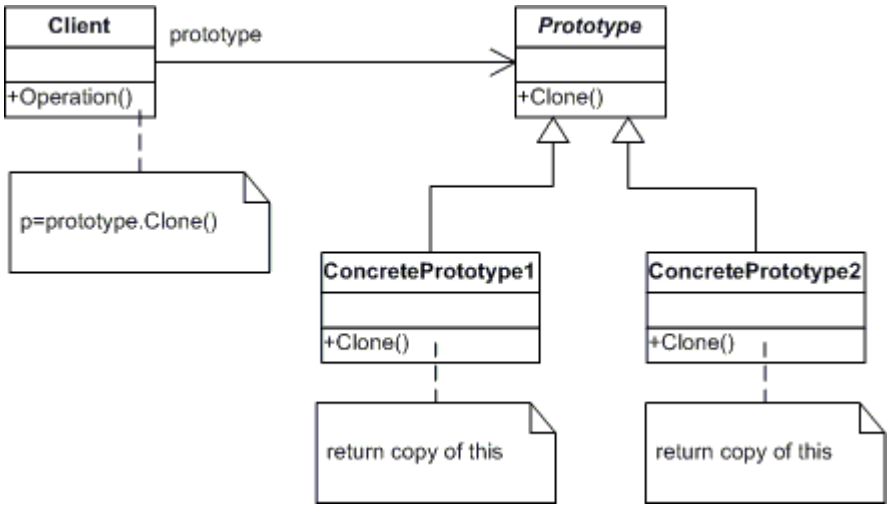


Figura 6.12.Structura şablonului Prototip

Participanți şi colaborări [Gamm94- Fig.6.12]

- Prototype
  - declară o interfață pentru a se clona;
- ConcretePrototype
  - implementează o operație pentru a se clona;
- Client
  - creează un obiect nou cerând prototipului să se cloneze;
- *Un client cere unui prototip să se cloneze;*
- *Clasa client trebuie să se inițializeze în constructor cu prototipul concret adecvat.*

Consecințe [Gamm94]

Avantaje:

- *Adăugarea şi ştergerea produselor la execuție.* Clasele de instanțiat pot fi specificate dinamic (clientul poate instala şi şterge prototipuri la execuție): prototipul se creează dinamic cu ajutorul unui parametru ce poate lua valori la execuție. În Metoda Factory, se instanțiază un anumit tip de aplicație la început, care va crea un tip specific de produse;
- Mai puțină derivare
  - evită ierarhia *de clase-factory paralelă cu ierarhia claselor de produse;*
  - în acord cu principiul care recomandă să *Preferăm Compunerea Moştenirii;*
- Specificarea de obiecte noi prin varierea *valorilor* prototipurilor
  - clientul se comportă diferit prin delegarea către prototip;
- Specificarea de obiecte noi prin varierea *structurii* prototipurilor
  - produse compuse;
- Abordarea ascunde complet clasele produs concret de clienți (se reduc dependențele de implementare). Astfel, se foloseşte doar interfața abstractă în codul client, devenit stabil față de extinderea cu tipuri noi de produse.

Dezavantaje:

- Fiecare subclasă a Prototype trebuie să implementeze *clone (deepClone)*
  - dificil când clasele există deja sau
  - obiectele interne nu permit copierea/ nu se pot serializa pentru clonarea de profunzime sau au referințe circulare.

Detalii de implementare [Gamm94]

- Folosirea unui gestionar de Prototipuri
  - numărul de prototipuri nu e fixat
    - ținеți un registru, care va fi manager de prototipuri;

- Clienții vor cunoaşte doar managerul (nu prototipul)
  - stocare asociativă;
- Inițializarea clonelor
  - Diferite metode de inițializare in functie de datele necesare;
- Implementarea operației clone
  - copie de suprafață vs. copie de profunzime.

Clonarea în Java

Copierea unui obiect în Java se face cu ajutorul metodei **clone:**

```
Obj ob1 = (Obj)ob0.clone();
```

Metoda creează o clonă a obiectului: alocă o nouă instanță şi plasează o *clonă bit cu bit* a obiectului curent în noul obiect. Metoda clone întoarce un obiect de tip Object, care trebuie convertit la tipul real al obiectului clonat. Mai există încă trei restricții ale acestei metode:

- Este o metodă *protected* şi poate fi apelată numai din interiorul clasei;
- Se pot clona doar obiectele care implementează interfața Cloneable;
- Obiectele ce nu se pot clona ridică CloneNotSupportedException.

```
protected Object clone() throws CloneNotSupportedException
```

Aceste aspecte sugerează următoarea implementare:

```
public class Produs implements Cloneable
{
public Object clone()  //versiunea definita de noi
{
try{
return super.clone();  //metoda interna protected
}
catch(Exception e)
{System.out.println(e.getMessage());
return null;
}}}
```

Exemplu.

```
class Door implements Cloneable  {
  public void Initialize( Room a, Room b) {
    room1 = a; room2 = b;
  }
```

```
  public Object clone() throws CloneNotSupportedException {
          return super.clone();
  }
  Room room1, room2;
}
```

Se pot construi şi proceduri particulare de clonare care modifică datele sau metodele de procesare din clasa clonată, în funcţie de parametrii transmişi metodei *clone*. *Clone* realizează doar o copie de suprafaţă a clasei originale, astfel încât dacă se operează modificări asupra datelor acestea se vor reflecta şi în obiectul original al clasei Prototip (Figura 6.13 a şi b)

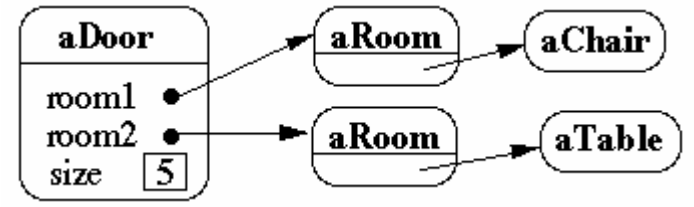


Figura 6.13.a. Obiect original

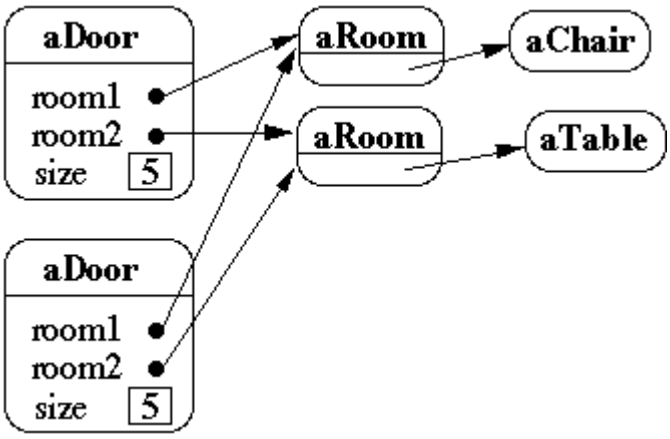


Figura 6.13.b. Copie de suprafaţă

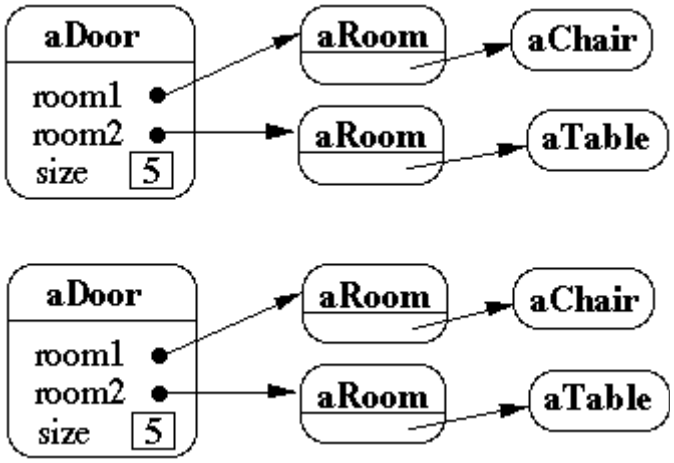


Figura 6.13.c. Copie de profunzime

Altfel spus, referințele la date sunt copii care se referă la aceeaşi locație în memorie. Dacă avem nevoie de o copie de profunzime trebuie să folosim interfața Serializable: o clasă este serializabilă dacă se poate scrie ca un stream de octeți şi aceşti octeți pot fi recitiți pentru reconstrucția clasei. Spre exemplu, dacă avem clasa Pacient şi clasa DatePacienti ce conține un vector de Pacienti, DatePacienti fiind Prototipul pe care dorim să îl clonăm din varii motive, le-am putea declara pe ambele serializabile:

```
public class DatePacienti implements Cloneable, Serializable
class Pacient implements Serializable
```

şi putem scrie octeții într-un stream de ieşire şi îi putem reciti pentru a crea o copie completă a datelor instanței clasei:

```
public Object deepClone()
{
try{
ByteArrayOutputStream b = new ByteArrayOutputStream();
ObjectOutputStream out = new ObjectOutputStream(b);
out.writeObject(this);
ByteArrayInputStream bIn = new
ByteArrayInputStream(b.toByteArray());
ObjectInputStream oi = new ObjectInputStream(bIn);
return (oi.readObject());
}
catch (Exception e)
{ System.out.println("exception:"+e.getMessage());
return null;
}
}
```

Metoda *deepClone* permite copierea unei instanțe a unei clase de orice complexitate, între original şi copie datele fiind complet independente.

Exemplul 1 (Soluția 2- Prototip). O alternativă la Metoda Factory este clonarea unui Prototip: se construiește în Produse o metodă de clonare (clone) cu ajutorul căreia se creează o copie a unui obiect Produs existent (Figura 6.14).

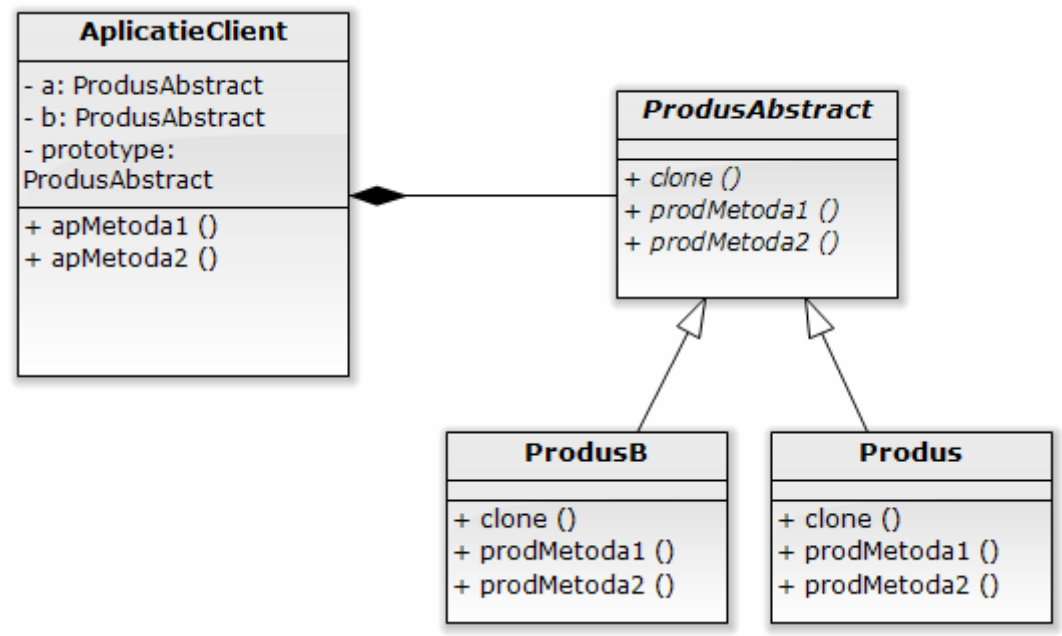


Figura 6.14

```
class Produs {
   int data;

   public Produs clone() {
      Produs aCopy = new Produs();
      aCopy.data = data;
      return aCopy; }
}
```

Folosirea Clonei **:**

```
class AplicatieClient   {
   ProdusAbstract a;
   ProdusAbstract b;
   ProdusAbstract prototype;

   public AplicatieClient(ProdusAbstract cloneMe ) {prototype = cloneMe;}

   public apMetoda1()   {
```

```
        ProdusAbstract d = prototype.clone();
        d.prodMetoda1();
        // . . .
        ProdusAbstract e = prototype.clone();
        // . . .    }
     // ...etc. etc...}
```

În altă parte:

```
AplicatieClient test = new AplicatieClient(new Produs() );
AplicatieClient testB = new AplicatieClient(new ProdusB() );
```

Observăm că prin aplicarea şablonului Prototip am evitat derivarea clasei ApplicationClass.

Exemplul 2. Problema labirintului

```
class MazePrototypeFactory {
public:
  MazePrototypeFactory(Maze*, Wall*, Room*, Door*);

  virtual Maze* MakeMaze() const;
  virtual Room* MakeRoom(int) const;
  virtual Wall* MakeWall() const;
  virtual Door* MakeDoor(Room*, Room*) const;

private:
  Maze* _prototypeMaze;
  Room* _prototypeRoom;
  Wall* _prototypeWall;
  Door* _prototypeDoor;};

MazePrototypeFactory::MazePrototypeFactory (
  Maze* m, Wall* w, Room* r, Door* d)
{ _prototypeMaze = m;
   _prototypeWall = w;
   _prototypeRoom = r;
  _prototypeDoor = d;}

Wall* MazePrototypeFactory::MakeWall () const
{return _prototypeWall->Clone();}

Door* MazePrototypeFactory::MakeDoor (
  Room* r1, Room *r2) const {
     Door* door = _prototypeDoor->Clone();  //creare prototip
     door->Initialize(r1, r2);  //initializare prototip
     return door;}
```

Crearea unui labirint pentru un joc:

```
MazePrototypeFactory simpleMazeFactory (new Maze, new Wall, new Room, new Door);
MazeGame game;
Maze* maze = game.CreateMaze(simpleMazeFactory);
```

Alte modificări posibile. Dacă ApplicationClass (Exemplul 1) mai foloseşte şi alte "produse" (ex. Roți, Angrenaje, etc.), fiecare aplicație este o familie de obiecte (şi toate acestea au subclase). Să presupunem că există restricții asupra tipului de Widget ce poate fi folosit cu un anume tip de Roată sau Angrenaj. În acest caz, metodele Factory sau Prototipurile pot manevra fiecare tip de produs dar devine dificil de asigurat că tipurile de produse nu se vor combina în mod greşit.

Exerciții.

1. Din tabela “Pacienți” a unei BD MySQL se preiau câmpurile CNP, Nume şi Diagnostic (folosiți JDBC). Datele se afişează sortate după nume într-o listă în partea din stânga a unei ferestre Swing. La acționarea unui buton “Clone” din cadrul interfeței, datele se afişează într-o listă din partea dreaptă, sortate crescator după diagnostic şi crescător după nume pentru persoanele cu acelaşi diagnostic. Folosiți şablonul Prototip. (Indicație: creați clasa Pacient şi clasa DatePacienti care conține un vector de pacienți şi implementează interfața Cloneable). Scrieți două implementări: în cadrul primeia sortarea se reflectă şi asupra datelor din lista din stânga, în cea de-a doua datele din cele două liste sunt independente.

2. Folosiți Prototype în crearea buştenilor şi a fructelor din jocul Hungry Hippau (Cap. 1, Ex.3).

6.3. Şablonul ABSTRACT FACTORY (Constructor Virtual Abstract)

Scop şi aplicabilitate

Acest şablon oferă o interfață pentru crearea de familii de obiecte înrudite sau dependente fără specificarea claselor lor concrete. Este util când sistemul trebuie configurat cu una din mai multe familii de produse şi trebuie forțat ca o familie de obiecte produs să fie folosite împreună. Clasele concrete generate vor fi izolate de codul client, numele lor fiind ascunse în Factory- astfel, familiile de clase se pot schimba/ interschimba cu uşurință. Şablonul necesită atenție la definirea condițiilor noi, clare (neambigue) ce trebuie să determine returnarea unei noi familii de clase, atunci când această familie este adaugată la program.

Participanți şi colaborări [Gamm94- Fig.6.15]

- Abstract Factory

- declară o interfață pentru operații pentru crearea de produse abstracte;

- ConcreteFactory
  - implementează operațiile de creare a produselor;
- AbstractProduct
  - declară o interfață pentru un tip de obiecte produs;
- ConcreteProduct
  - implementează o interfață pentru un tip de obiecte produs;
- Client
  - foloseşte doar interfețe declarate de AbstractFactory şi AbstractProduct;
- *O singură instanță a ConcreteFactory creată, creează produse cu o implementare particulară.*

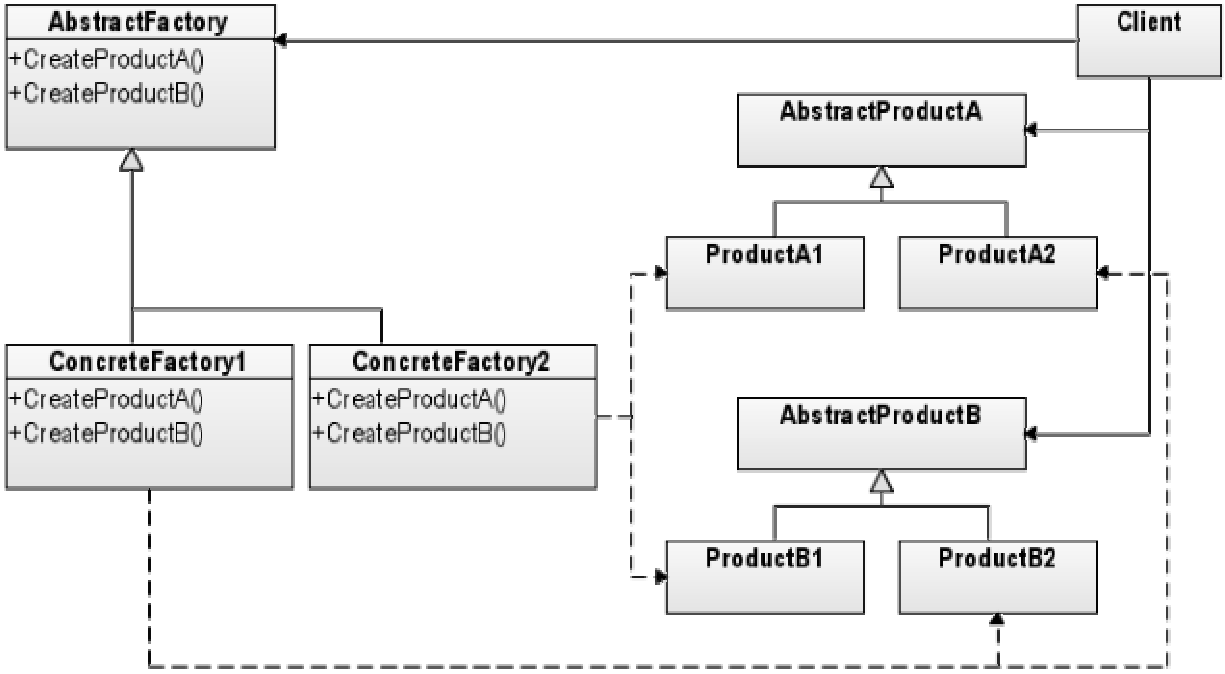


Figura 6.15.Structura şablonului Abstract Factory

Consecințe [Gamm94]

- Izolarea claselor concrete (acestea apar în clasele ConcreteFactory, nu în codul clientului);
- Facilitează schimbul de familii de produse
  - un ConcreteFactory apare într-un singur loc – deci va fi uşor de modificat;
- Promovează consistența între produse
  - toate produsele unei familii se modifică *împreună, în acelaşi timp;*
- Este dificilă susținerea de tipuri noi de produse
  - necesită o modificare în interfața AbstractFactory (şi, în consecință, a tuturor subclaselor sale).

Detalii de implementare [Gamm94]

- Fabrici ca Singleton-uri
  - pentru a asigura că doar un singur ConcreteFactory este creat pentru fiecare familie de produse;
- Crearea Produselor
  - colecţie de *Metode Factory;*
  - poate fi implementat şi folosind *Prototype*
    - definiţi o instanţă prototipică pentru fiecare produs din ConcreteFactory;
- Definirea de “Fabrici” Extensibile
  - o singură metodă factory cu parametri;
  - mai flexibil, dar mai nesigur.

Exemplul 1-Etapa 5. Şablonul Abstract Factory (Fig. 6.16)

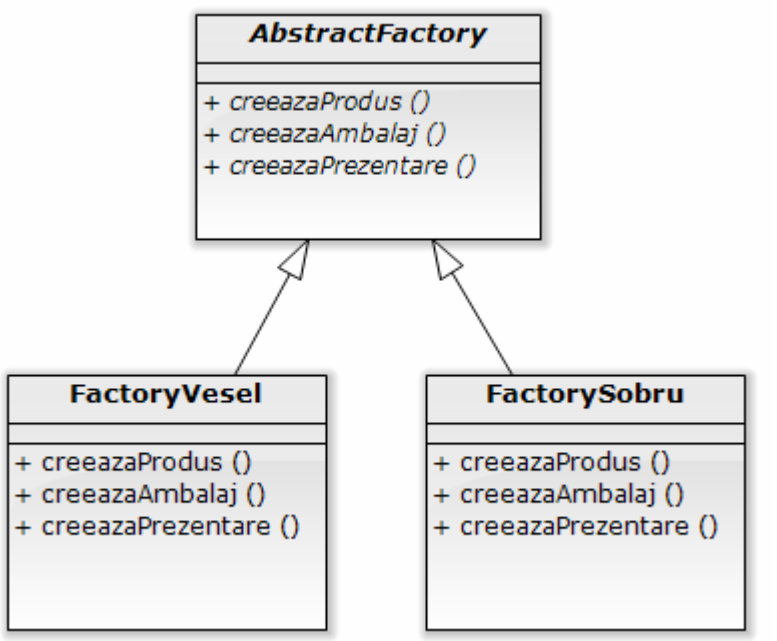


Figura 6.16

```
class AplicatieClient   {
ProdusAbstract a;
AmbalajAbstract b;
AbstractFactory partFactory;

public AplicatieClient(AbstractFactory aFactory)   {partFactory = aFactory;   }

public apMetoda1()      {
          ProdusAbstract d = partFactory.creeazaProdus();
                        d.prodMetoda1();      // ...
```

```
AmbalajAbstract e = partFactory.creeazaAmbalaj();   } // ....
```

Exemplul 2. Rezolvarea Problemei Labirintului [Gamm94]

```
class MazeFactory {
  public:
    MazeFactory();

    virtual Maze* MakeMaze() const
       { return new Maze; }
    virtual Wall* MakeWall() const
       { return new Wall; }
    virtual Room* MakeRoom(int n) const
       { return new Room(n); }
    virtual Door* MakeDoor(Room* r1, Room* r2) const
       { return new Door(r1, r2); }
};
Maze* MazeGame::CreateMaze (MazeFactory& factory) {
     Maze* aMaze = factory.MakeMaze();
     Room* r1 = factory.MakeRoom(1);
     Room* r2 = factory.MakeRoom(2);
     Door* aDoor = factory.MakeDoor(r1, r2);

     aMaze->AddRoom(r1);
     aMaze->AddRoom(r2);

     r1->SetSide(North, factory.MakeWall());
     r1->SetSide(East, aDoor);
     r1->SetSide(South, factory.MakeWall());
     r1->SetSide(West, factory.MakeWall());

     r2->SetSide(North, factory.MakeWall());
     r2->SetSide(East, factory.MakeWall());
     r2->SetSide(South, factory.MakeWall());
     r2->SetSide(West, aDoor);

     return aMaze;
}
```

Exemplul 3. Crearea produselor

3.1. Crearea produselor folosind propriile metode factory :

```
abstract class ProdusFactory {
   public Window createWindow();
```

```
   public Menu createMenu();
   public Button createButton();
}

class MacProdusFactory extends ProdusFactory {
     public Window createWindow()
       { return new MacWidow() }
     public Menu createMenu()
       { return new MacMenu() }
     public Button createButton()
       { return new MacButton()  }
}
```

3.2. Crearea produselor folosind metodele factory ale produselor:

- subclasa doar furnizează produsele concrete în constructor;
- evită reimplementarea MF în subclase;

```
abstract class ProdusFactory {
   private Window windowFactory;
   private Menu menuFactory;
   private Button buttonFactory;
   public Window createWindow()
      { return windowFactory.createWindow() }
   public Menu createMenu();
      { return menuFactory.createWindow() }
   public Button createButton()
      { return buttonFactory.createWindow() }
}

class MacProdusFactory extends ProdusFactory {
   public MacWidgetFactory() {
      windowFactory = new MacWindow();
      menuFactory = new MacMenu();
      buttonFactory = new MacButton();
   }
}
```

Exemplul 4 [Gamm94].

O aplicaţie clasică a Abstract Factory este situaţia în care sistemul trebuie să poată afişa interfeţe utilizator diverse: Windows-9x, Motif sau Macintosh. Factory primeşte cererea de interfeţe Windows şi va returna componente vizuale tip Windows (butoane, check-box, ferestre etc.)

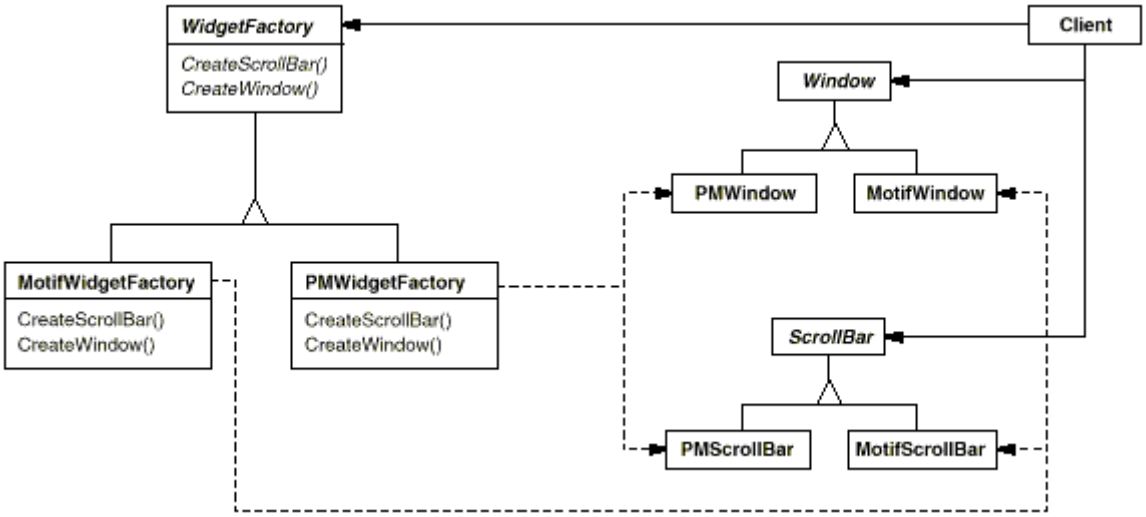


Figura 6.17. Exemplul 4

Exemplul 5 [Coop98]. Grădini

Să presupunem că dorim să creem trei tipuri de grădini: perene, de legume şi anuale. Plantele trebuie combinate corespunzător, ținând cont de trei criterii:

1. Ce plante se potrivesc cel mai bine în marginea grădinii?
2. Ce plante se potrivesc cel mai bine în centrul grădinii?
3. Ce plante nu sunt afectate de umbra parțială?

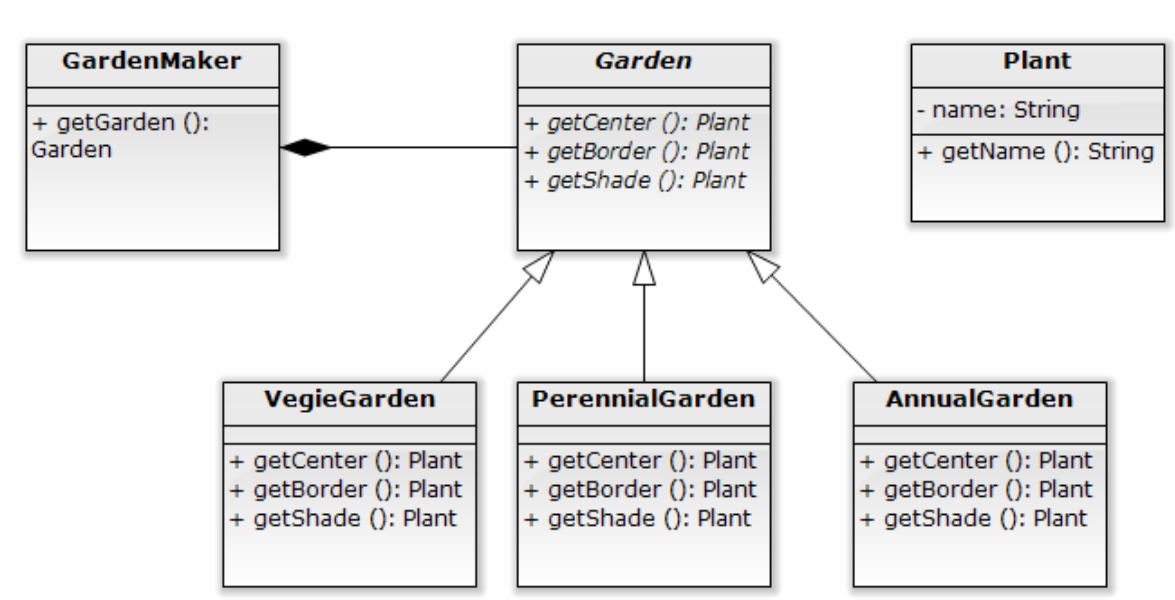


Figura 6.18. Exemplul 5

Clasa de bază Garden este abstractă şi defineşte trei întrebări:

```
public abstract class Garden {
public abstract Plant getCenter();
public abstract Plant getBorder();
public abstract Plant getShade();
}
```

Clasa Plant nu face altceva decât să returneze sub formă de String numele plantei:

```
public class Plant {
String name;
public Plant(String pname) {
name = pname; //save name
}
public String getName() {
return name;
}
}
```

Într-un sistem real, ar trebui să consultăm o bază de date elaborată, cu informaţii despre plante. În acest exemplu simplu ne limităm să returnăm câte un tip din fiecare plantă. Astfel, spre exemplu, pentru grădina de legume vom avea:

```
public class VegieGarden extends Garden {
public Plant getShade() {
return new Plant("Broccoli");
}
public Plant getCenter() {
return new Plant("Corn");
}
public Plant getBorder() {
return new Plant("Peas");
}
}
```

Obţinem în acest mod mai multe obiecte Garden, fiecare dintre ele returnând mai multe obiecte de tip Plante. Vom construi clasa GardenMaker să returneze un astfel de obiect Garden în funcţie de parametrul String primit:

```
class GardenMaker
{
//Abstract Factory ce returneaza un tip de gradina din trei
private Garden gd;
```

```
public Garden getGarden(String gtype)
{
gd = new VegieGarden(); //implicit
if(gtype.equals("Perennial"))
gd = new PerennialGarden();
if(gtype.equals("Annual"))
gd = new AnnualGarden();
return gd;
}
}
```

Utilizarea GardenFactory

Dacă s este un String ce a luat valori prin diverse metode în cadrul unei IUG,

```
garden = new GardenMaker().getGarden(s);
```

tipurile concrete de plantă ale acestei grădini sunt:

```
centerPlant = garden.getCenter().getName();
borderPlant = garden.getBorder().getName();
shadePlant = garden.getShade().getName();

//…
```

Exerciții.

1. Realizați meniuri pentru un restaurant prin alegerea combinațiilor potrivite din următoarele feluri: *Aperitiv*, *Fel principal*, *Desert*, *Bautură*. Construiți următoarele variante: Meniu Vegan, Meniu Peşte, Meniu Pasăre, Meniu Vită.

2. Realizați două tipuri de formular de introducere date personale (câmpuri text pentru nume, o listă cu scrollbar pentru vârstă (18-65 ani), un buton radio pentru vegetarian/ nevegetarian, un buton OK de preluare a datelor): o versiune mai veselă, în culori deschise, chenare mai originale etc, şi o versiune mai simplă şi mai sobră. Combinați şabloanele Abstract Factory şi Builder (Secțiunea 6.4.).

6.4. Şablonul BUILDER (Constructor)

Scop şi aplicabilitate

Acest şablon separă construirea unui obiect complex de reprezentarea sa, astfel încât acelaşi proces de construcție poate genera reprezentări diferite. Spre deosebire de Abstract Factory şi de Metoda Factory care

au drept scop folosirea polimorfismului pentru un cod mai flexibil şi mai stabil, şablonul Builder găseşte o soluţie la apelurile telescopice de constructori (un constructor apelează alţ(i) constructori care apelează alţ(i) constructor(i) etc. în combinaţii exponenţiale). În loc de a folosi constructori numeroşi, şablonul Builder foloseşte un alt obiect (Constructorul) care primeşte parametrii de iniţializare pas cu pas şi returnează obiectul construit la finalizarea sa. Şablonul este util în crearea obiectelor ce conţin date de tip atribut – valoare (ca interogările SQL)- date ce nu se pot edita uşor şi care trebuie construite deodată (nu pas cu pas).

Adesea se începe în design cu Factory Method (mai uşoară, dar în care proliferează subclasele) şi se evoluează spre Abstract Factory, Prototype, sau Builder (mai flexibil şi mai complex). Şabloanele creaţionale se pot folosi şi complementar: Builder poate utiliza un alt şablon pentru a implementa ce componente se construiesc.

Folosiţi Builder atunci când:

- algoritmul pentru crearea obiectului complex trebuie să fie independent de părţile ce compun obiectul şi de modul lor de asamblare;
- procesul de construcţie trebuie să permită reprezentări diferite pentru obiectul construit.

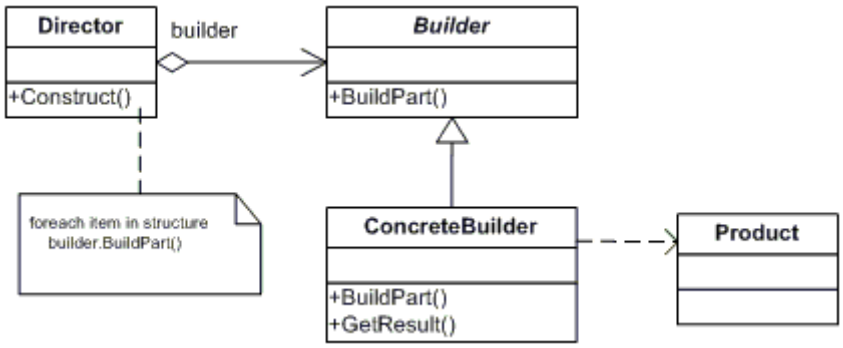


Figura 6.19. Structura şablonului Builder

Participanţi [Gamm94, Fig. 6.19]

- Builder
  - Specifică o interfaţă abstractă pentru crearea părţilor unui obiect Product;
- ConcreteBuilder
  - Construieşte şi asamblează părţile produsului prin implementarea interfeţei Builder;
  - Defineşte şi ţine evidenţa reprezentărilor pe care le creează;
  - Furnizează o interfaţă pentru preluarea produsului;
- Director
  - Construieşte un obiect folosind interfaţa Builder;
- Product

- Reprezintă obiectul complex care se construieşte. ConcreteBuilder construieşte reprezentarea internă a produsului şi defineşte procesul prin care el este asamblat;
- Include clasele ce definesc părțile constituente, inclusiv intrerfețele pentru asamblarea părților în rezultatul final.

Colaborări [Gamm94]

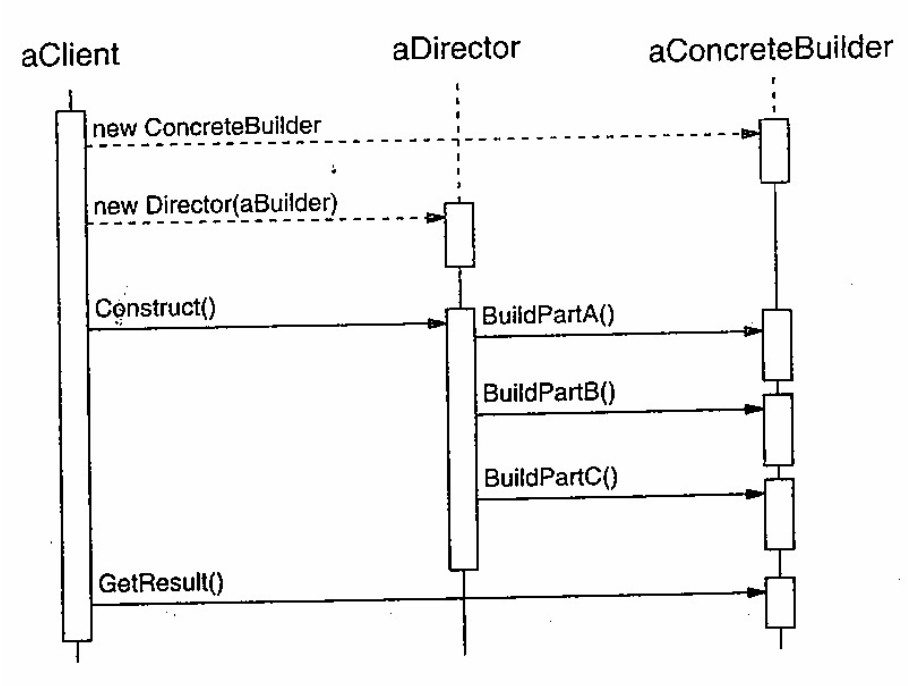


Figura 6.20.Colaborări în şablonul Builder

- Clientul creează obiectul Director şi îl configurează cu obiectul Builder dorit (versiunea particulară de subclasă);
- Directorul notifică Builder-ul când trebuie construită o nouă parte a obiectului;
- Builder-ul preia cererile de la Director şi adaugă părțile la produs (pe rând);
- Clientul preia produsul de la Builder.

Consecințe [Gamm94]

- Permite varierea reprezentării interne a unui produs
  - Directorul foloseşte o interfață abstractă (Builder) pentru a face referire la produse;
  - Dacă se doreşte modificarea reprezentării interne a produsului nu trebuie decât să scriem o nouă clasă care să implementeze interfața Builder;
- Izolează codul pentru construcție şi reprezentare
  - Îmbunătățeşte modularitatea prin încapsularea modului de construire şi reprezentare a unui obiect complex;
  - Fiecare ConcreteBuilder conține tot codul necesar pentru crearea şi asamblarea unui tip particular de produs;

- Rafinează controlul asupra procesului de construire
  - Produsul este construit pas cu pas sub controlul Directorului (nu se creează totul odată, ca la celelalte şabloane creaționale).

Implementare [Gamm94]

- Modelul procesului de construire şi asamblare:
  - Rezultatele cererilor de construcție să se poată adăuga produsului final (construit pas cu pas);
- Metode vide în Builder
  - Metodele de construire nu sunt declarate ca funcții membre virtuale, ci ca metode vide, lăsând astfel clienții să suprascrie doar operațiile care îi interesează.

Exemplu. Funcția de creare a labirintului *CreateMaze():*

```
class Maze {
public:
    Maze();
    void AddRoom(Room*);
    Room* RoomNo(int) const;
private:
    //….};

Maze* MazeGame::CreateMaze() {
           Maze* aMaze=new Maze;
           Room* r1=new Room(1);
           Room* r2=new Room(2);
           Door* theDoor=new Door(r1,r2);
           aMaze->AddRoom(r1);
           aMaze->AddRoom(r2);

           r1->SetSide(North, new Wall);
           r1->SetSide(East, theDoor);
           r1->SetSide(South, new Wall);
           r1->SetSide(West, theDoor);
           r2->SetSide(North, new Wall);
           r2->SetSide(East, new Wall);
           r2->SetSide(South, new Wall);
           r2->SetSide(West, theDoor);
          return aMaze; }
```

Aplicarea şablonului Builder

Interfața de definire a labirinturilor (Builder):

```
class MazeBuilder{
public:
     virtual void BuildMaze(){}
     virtual void BuildRoom(int room){}
     virtual void BuildDoor(int roomFrom, int roomTo){}
    virtual Maze*  GetMaze() {return 0;}
protected:
    MazeBuilder();
};
```

Funcția membru *CreateMaze()* poate avea un Builder ca parametru:

```
Maze* MazeGame::CreateMaze(MazeBuilder& builder) {
           builder.BuildMaze();

           builder.BuildRoom(1);
           builder.BuildRoom(2);
           builder.BuildDoor(1,2);
          return builder.GetMaze(); }
```

Builder-ul ascunde reprezentarea internă a labirintului (clasele ce definesc uşi, camere etc.), şi felul în care ele se asamblează pentru a forma labirintul final. Putem refolosi Builder-ul pentru a crea tipuri diferite de labirinturi:

```
Maze* MazeGame::CreateComplexMaze(MazeBuilder& builder) {
           builder.BuildRoom(1);
           …
           builder.BuildRoom(1001);
          return builder.GetMaze(); }
```

Subclasa StandardMazeBuilder este o implementare ce creează labirinturi simple. Labirintul construit este reținut în variabila _currentMaze.

```
class StandardMazeBuilder :  public MazeBuilder {
public:
      StandardMazeBuilder();
      virtual void BuildMaze();
      virtual void BuildRoom(int );
      virtual void BuildDoor(int , int );

virtual Maze* GetMaze();
private:
     Direction CommonWall(Room*, Room*); // determină direcția peretelui  comun
dintre două camere
```

```
        Maze*   _currentMaze;
};
```

//Constructorul:

```
StandardMazeBuilder::StandardMazeBuilder(){
      _currentMaze=0;}
void StandardMazeBuilder::BuildMaze(){
      _currentMaze=new Maze; }
Maze* StandardMazeBuilder::GetMaze(){
       return _currentMaze;}
```

//Crearea unei camere şi a pereţilor săi:

```
void StandardMazeBuilder::BuildRoom(int n) {
       if (!_currentMaze->RoomNo(n)) {
       Room* room=new Room(n);
       _currentMaze->AddRoom(room);
       room->SetSide(North, new Wall);
       room->SetSide(South, new Wall);
       room->SetSide(East, new Wall);
       room->SetSide(West, new Wall); } }
```

//crearea uşii dintre camerele n1 şi n2:

```
void StandardMazeBuilder::BuildDoor (int n1, int n2) {
       Room*  r1=_currentMaze->RoomNo(n1);
       Room*  r2=_currentMaze->RoomNo(n2);
       Door* d=new Door(r1, r2);
       r1->SetSide(CommonWall(r1,r2), d);
       r2->SetSide(CommonWall(r2,r1), d); }
```

Clienţii pot crea acum labirinturi folosind CreateMaze în conjuncţie cu StandardMazeBuilder:

```
Maze* maze;
MazeGame game;
StandardMazeBuilder builder;
game.CreateMaze(builder);
Maze=builder.GetMaze();
…
```

Exemplu[Gamm94].

Un cititor din formatul RTF trebuie să poată converti RTF în mai multe formate de text (text ASCII, editor interactiv de text etc.). Configurăm clasa RTFReader cu un obiect TextConverter care efectuează conversia pe măsură ce RTFReader parsează documentul RTF (Figura 6.21). Obiectele TextConverter convertesc datele şi reprezintă token-ul într-un anumit format. Prin folosirea şablonului, se separă algoritmul de parsare

a documentelor RTF de modul de creare şi reprezentare a diferitelor formate -deci se poate refolosi algoritmul de parsare în alte programe (RTFReader este Director, extensiile TextConverter sunt Builderi).

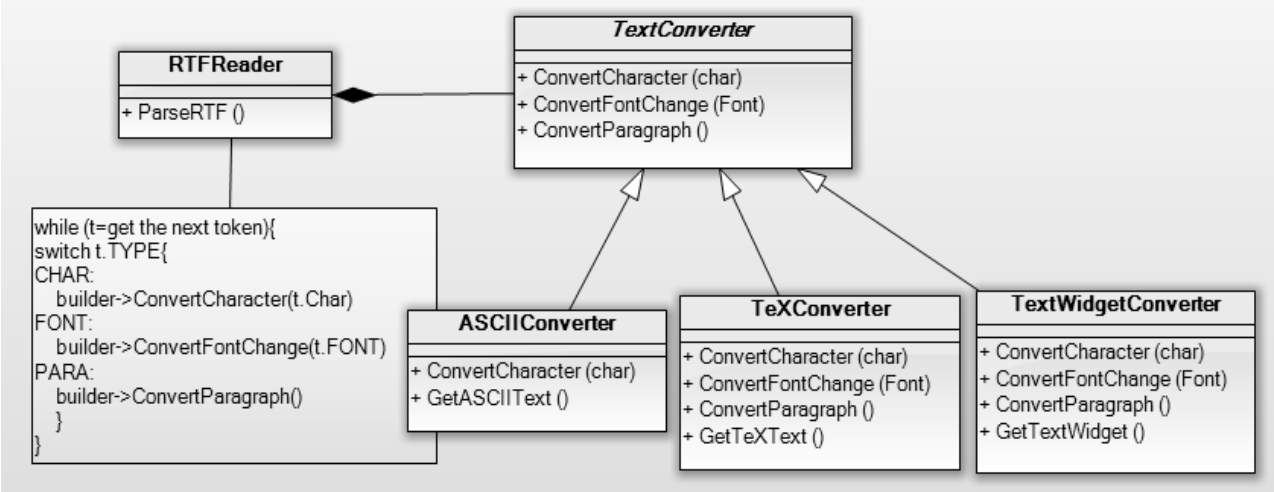


Figura 6.21.Convertor de text

Şabloane înrudite

- Abstract Factory este înrudit cu Builder prin faptul că poate construi obiecte complexe, dar există şi diferențe:

Builder	**Abstract Factory**
se concentrează pe crearea obiectelor pas cu pas	pune accentul pe construirea de familii de produse (simple sau complexe)
întoarce produsul într-un final, după ce a asamblat toate părțile	întoarce produsul imediat

- Deseori Composite se construieşte cu Builder.

Exerciții

1. Într-o fereastră Java Swing, secțiunea din partea stângă conține câteva câmpuri de înregistrare ale datelor personale ale unui pacient: Nume, Adresă, CNP, şi o listă cu diverse specializări medicale. La selectarea unei specializări, în secțiunea din dreapta a ferestrei se afişează două liste: prima cu analizele de sânge specifice specializării, şi a doua cu observațiile clinice specifice specializării. Alcatuiți perechi de fişiere cu aceste

informaţii, pentru cel puţin 3 specializări, şi folosiţi şablonul Builder pentru a actualiza partea din dreapta a ferestrei.

2. Creaţi o interfaţă în Java Swing: într-o fereastră există două panel-uri dispuse pe orizontală. În panel-ul din stânga, intitulat “Informaţii spital” plasaţi o listă cu diverse specializări medicale: cardiologie, neurologie, dermatologie, pediatrie etc. La selectarea unei specializări din listă, în panoul din dreapta apar numele tuturor medicilor cu specializarea respectivă, şi orarul lor de lucru (Ionescu Luni: 15-17, Marti: 14-16, etc). Folosiţi tehnologia JDBC pentru a interoga o tabelă MySQL cu următoarele câmpuri: Nume_medic, Specializare, Program_luni, Program_marti, …, Program_sambata. Organizaţi codul cu ajutorul şablonului Builder pentru a crea conţinutul panoului din dreapta şi notaţi ce avantaje are această abordare.

3. Folosiţi Builder în crearea celor trei nivele diferite de dificultate ale jocului Brickles (Cap. 1, Ex. 2).

6.5. Şablonul SINGLETON

Scop şi aplicabilitate

Şablonul Singleton asigură ca o clasă să aibă doar o singură instanţă şi furnizează un punct unic de acces global la ea. Instanţa poate fi extensibilă. O variantă a şablonului permite crearea unui set numărabil de instanţe.

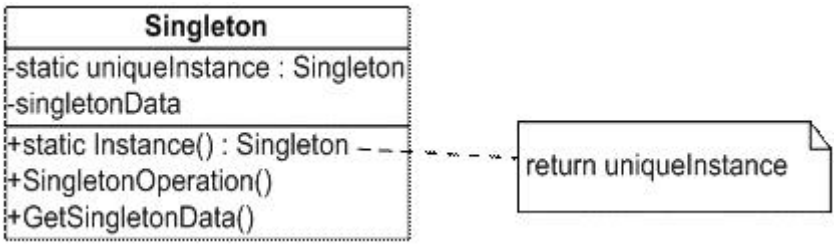


Figura 6.22.Structura şablonului Singleton

Participanţi şi Colaborări [Gamm94- Fig.6.22]

- Singleton
  - defineşte o metodă Instance care devine singura “poartă" prin care clienţii pot accesa instanţa sa unică
    - Instance este o metodă a clasei (funcţie membră statică în C++);
  - Poate fi responsabil cu crearea propriei sale instanţe unice;
- *Clienţii accesează instanţele Singleton doar prin metoda Instance.*

Consecințe [Gamm94]

Avantaje:

- Acces controlat la instanța unică;
- Permite rafinarea operațiilor şi reprezentării;
- Permite un număr variabil (dar precis) de instanțe;
- Reduce vizibilitatea globală.

O idee de implementare este să plasați constructorul în secțiunea de date private/protected (ca în Exemplul 1).

Exemplul 1 [Coop98]. Derulatorul de imprimare (printer spooler): Crearea unui Singleton folosind o metodă statică (Java).

O abordare posibilă este crearea Singleton-urilor folosind o metodă statică pentru a controla şi urmări instanțele. Pentru a preveni instanțierea clasei mai mult decât o dată, constructorul este declarat *private* astfel că o instanță se poate crea numai din interiorul metodei statice a clasei:

```
class iSpooler
{
//prototipul unei clase de derulator-imprimare  ce poate avea doar o instanta

static boolean instance_flag = false; //true daca exista o instanta

// constructorul este private- nu e necesar sa aiba vreun continut

private iSpooler() { }
//metoda statică Instance returnează o instanţă sau null
static public iSpooler Instance()
{
if (! instance_flag)
{
instance_flag = true;
return new iSpooler(); //apelabil doar din interiorul Instance
}
else
return null; //nu mai returnează alte instanţe
}

public void finalize()
{
instance_flag = false;
}
}
```

Un avantaj al acestei metode este că nu trebuie tratate excepţiile dacă Singleton-ul există deja: pur şi simplu metoda *Instance* întoarce *null*:

```
iSpooler pr1, pr2;

//deschide un derulator—aceasta trebuie sa functioneze:
System.out.println("Deschiderea unui derulator");
pr1 = iSpooler.Instance();
if(pr1 != null) System.out.println("exista deja 1 derulator");

//incercarea de a deschide inca un derulator trebuie sa esueze
System.out.println("Deschiderea a doi derulatori");
pr2 = iSpooler.Instance();
if(pr2 == null)
System.out.println("nu există instanţe");
```

Dacă veţi încerca să creaţi direct instanţe ale clasei iSpooler, eşecul va fi vizibil de la compilare întrucât constructorul a fost declarat private:

```
//eşuează la compilare întrucât constructorul este private
iSpooler pr3 = new iSpooler();
```

Exemplul 2. Asigurarea Instanţei Unice- implementare în C++

```
class Singleton {
public:
    static Singleton* Instance();
private:
    Singleton();
    static Singleton * _instance; };

Singleton * Singleton::_instance=0;

Singleton * Singleton::Instance()  {
     if (_instance==0) {
          _instance= new Singleton;}
     return _instance;
}
```

Exemplul 3. Crearea unei singure MazeFactory

```
class MazeFactory {
public:
```

```
    static MazeFactory* Instance();
private:
    MazeFactory();
    static MazeFactory * _instance; };

MazeFactory * MazeFactory::_instance=0;

MazeFactory * MazeFactory::Instance()  {
     if (_instance==0) {
          _instance= new MazeFactory;}
     return _instance;
}
```

Exerciții.

1. Aplicați şablonul Singleton pe clasa ce implementează hipopotamul din jocul Hungry Hippau (Cap 1, ex.3).
2. Aplicați şablonul Singleton pe clasa Factory care creează formele din jocul Tetris (Ex. 1, Secțiunea 6.1.)

O concluzie de ansamblu a şabloanelor creaționale este că acestea pot face crearea unui joc (şi a oricărui tip de obiecte, în general) mai *flexibilă*, *nu mai scurtă*.

CAP.7. ŞABLOANE COMPORTAMENTALE

7.1. Şablonul ITERATOR

„Algoritmii (orice cod) pe care îl scriem folosind iteratori este decuplat de structurile de date pe care le manevrează astfel încât codul este mai reutilizabil şi mai general (Alexandr Stepanov).”

Scop şi aplicabilitate

Şablonul Iterator furnizează o modalitate de accesare secvențială a elementelor unui agregat, fără a expune reprezentarea care stă la baza acestor elemente (accesează conținutul unui obiect agregat fără să îi expună reprezentarea internă). Iteratorul susține traversări multiple ale obiectelor agregat şi furnizează o interfață uniformă de traversare a diferitelor structuri agregat (i.e. pentru a susține iterații polimorfice).

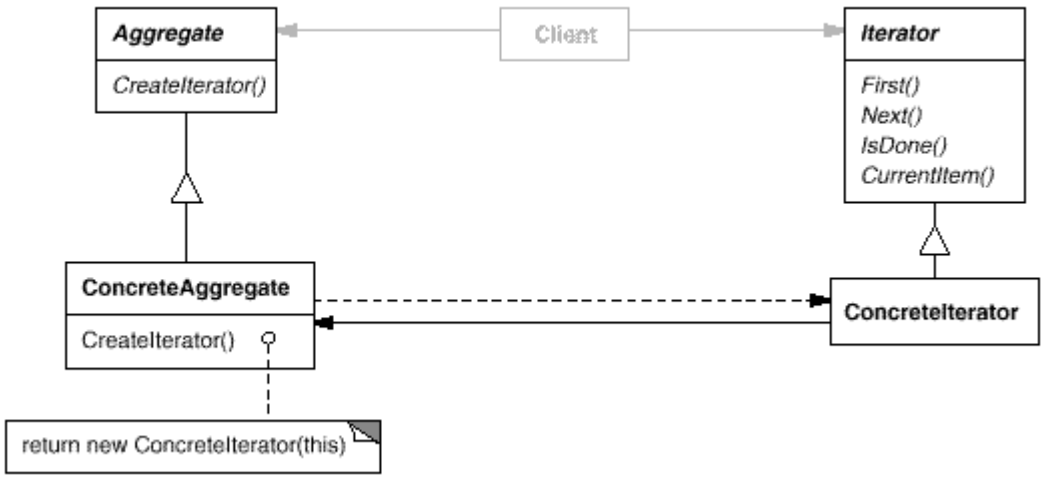


Figura 7.1. Structura şablonului Iterator

Participanți [Gamm94, Fig. 7.1]

- Iterator
  - Defineşte o interfață pentru traversarea şi accesarea elementelor;
- ConcreteIterator
  - Implementează interfața Iterator;
  - Ține evidența poziției curente în traversarea agregatului;
- Aggregate
  - Defineşte o interfață pentru crearea unui obiect Iterator;
- ConcreteAggregate
  - Implementează interfața de creare Iterator, pentru a întoarce o instanță potrivită de ConcreteIterator.

Colaborări [Gamm94]

- Un ConcreteIterator ține evidența obiectului curent din agregat, şi poate calcula următorul obiect din traversare.

Consecințe [Gamm94]

Sprijină variațiunile în traversarea unui agregat (ex. în generarea de cod- arborele de analiză poate fi traversat în inordine sau preordine; folosind iteratori, se schimbă doar o instanță de iterator cu alta);

1. Iteratorii simplifică interfața agregatului (din care se elimină operațiile specifice de acces şi traversare);
2. Pot exista mai multe traversări simultan;
   - Fiecare iterator ține evidența stării propriei iterații.

Implementare [Gamm94]

Iteratorii trebuie să fie robuşti:

- Trebuie să ne asigurăm că inserțiile şi ştergerile nu vor interfera cu parcurgerea agregatului, fără a crea un duplicat al acestuia;
- De obicei robustețea se obține prin înregistrarea iteratorilor la agregatul care îi foloseşte: agregatul actualizează starea iteratorilor asociați la fiecare ştergere sau adăugare de elemente.

Exemplul 1. Iterator pentru liste.

Folosim Iterator când dorim ca un obiect agregat tip listă să furnizeze acces la elementele sale fără să cunoaştem structura sa internă şi/sau avem nevoie de tipuri diferite de traversare asupra listei, şi nu dorim să încărcăm interfața listei cu operațiile de traversare. Un obiect iterator preia responsabilitatea pentru accesarea şi traversarea listei (Figura 7.2).

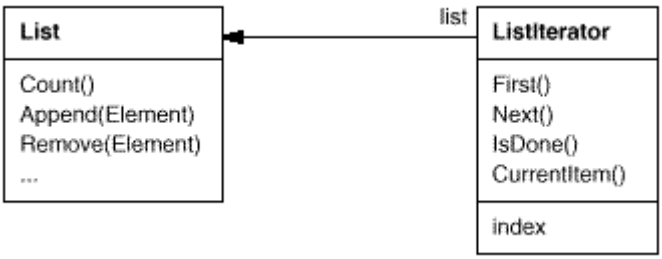


Figura 7.2. Iteratorul unei liste

Separarea mecanismului de traversare de obiectul List permite definirea de iteratori pentru diferite tipuri de traversare (fără a-i enumera în interfața listei). Putem generaliza conceptul de iterator să susțină iterații polimorfice- astfel putem schimba clasa agregat fără a modifica şi codul client şi mecanismul de iterație devine independent de clasa agregat concretă.

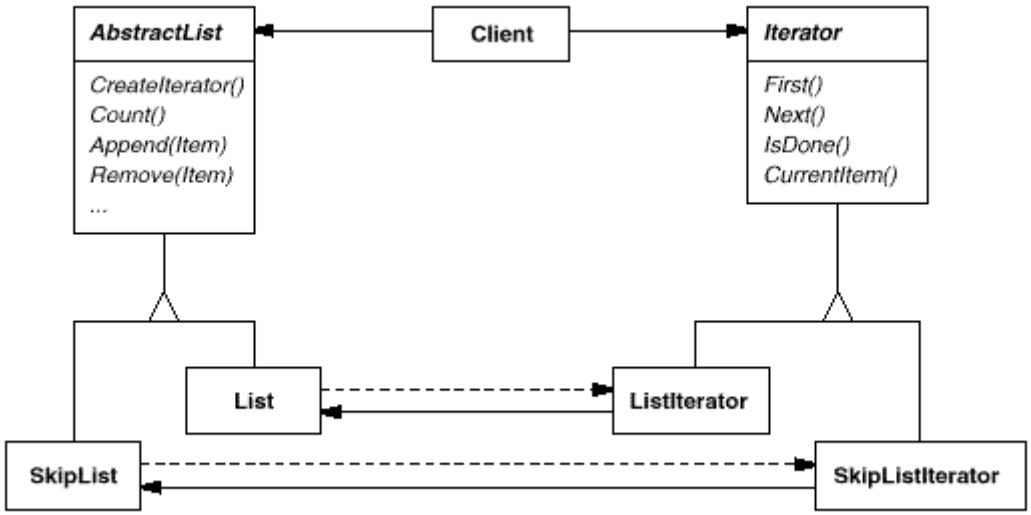


Figura 7.3. Iterator pentru liste

Obiectele listă îşi creează propriul iterator prin metoda *CreateIterator()*; iar *CreateIterator()* este o metodă Factory, care conectează cele două ierarhii [Fig. 7.3].

Exemplul 2. În Java şablonul este implementat de clasa Iterator -cu metodele *next()*, *hasNext()*, *remove()*. Metoda *next()* întoarce un tip generic Object. Dacă dorim ca Iteratorul să lucreze numai cu un anumit tip putem folosi abordarea următoare:

```
import java.util.*;

public class TypedIterator implements Iterator {
  private Iterator imp;
  private Class type;
  public TypedIterator(Iterator it, Class type) {
    imp = it;
    this.type = type;
  }
  public boolean hasNext() {
    return imp.hasNext();
  }

public void remove() { imp.remove(); }

public Object next() {
    Object obj = imp.next();
    if(!type.isInstance(obj))
      throw new ClassCastException("TypedIterator for type " + type +
        " encountered type: " + obj.getClass());
    return obj;
  }
```

```
}
```

Exerciții.

1. Folosiți şablonul Iterator pentru parcurgerea în inordine, preordine şi postordine a arborilor de sortare-căutare.
2. Construiți Iteratori pentru parcurgerea listelor dinamice în C++.

7.2. Şablonul COMMAND (Comandă- Tratarea cererilor)

Scop şi aplicabilitate

Şablonul Command încapsulează cereri ca obiecte, şi permite:

- parametrizarea clienților cu diferite cereri (şi alegerea cererii la momentul execuției);
- înlănțuirea şi/sau înregistrarea cererilor (într-un fişier log, spre exemplu; sprijină specificarea, înlănțuirea, şi executarea cererilor la momente diferite);
- suport pentru operații anulabile (UNDO);
- modelarea tranzacțiilor
    - structurarea sistemelor construite pe operații primitive, în jurul operațiilor de nivel înalt;
    - interfața comună permite invocarea identică a tuturor tranzacțiilor.

Spre deosebire de şablonul Lanț de Responsabilități (Chain of Responsibility), care transmite o cerere de-a lungul unui lanț de clase, şablonul Command transmite o cerere doar către un anumit obiect. Şablonul încapsulează cererea pentru o acțiune specifică în interiorul unui obiect, conferindu-i o interfață publică cunoscută, iar clientul va apela la această interfață, fără să fie nevoit să cunoască acțiunea propriu-zisă pe care o cere. Acțiunea se va putea astfel modifica fără să fie necesar să afectăm în vreun fel codul client.

Obiectele Command se mai pot folosi şi pentru a spune programului să execute comanda în momentul în care resursele necesare devin disponibile, situație în care comenzile sunt înlănțuite într-o coadă de aşteptare pentru a fi executate mai târziu. În fine, obiectele de tip Command pot reține operații pentru a realiza cereri de anulare (Undo).

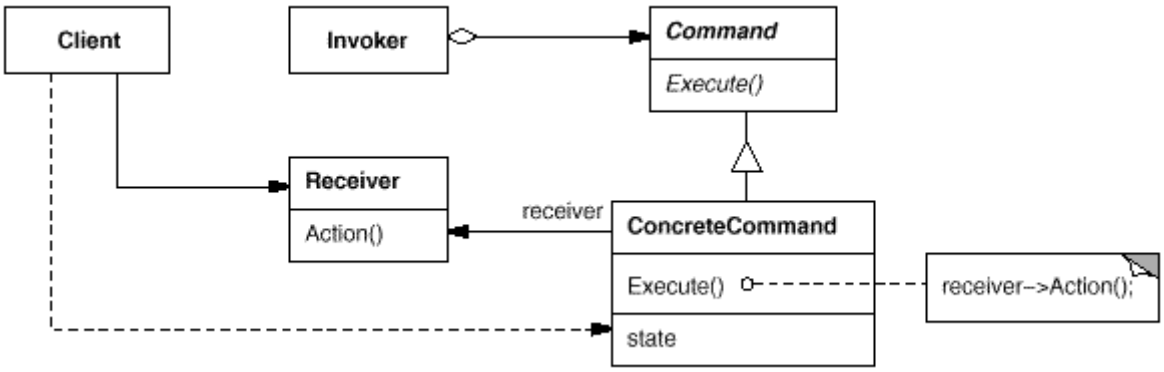


Figura 7.4. Structura şablonului Command

Participanți [Gamm94, Fig.7.4]

- Command
  - declară interfața pentru executarea operației;
- ConcreteCommand
  - asociază o acțiune concretă unei cereri;
- Invoker
  - solicită comenzii să execute cererea;
- Receiver
  - ştie cum să execute operațiile asociate cu îndeplinirea unei cereri;
- Client
  - creează o comandă concretă (ConcreteCommand) şi setează primitorul său.

Colaborări [Gamm94, Fig. 7.5]

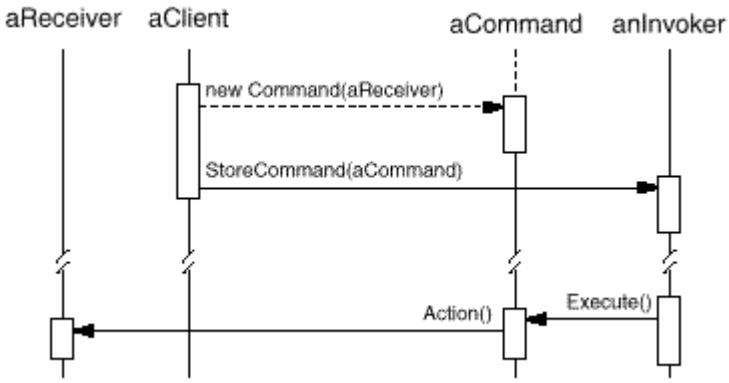


Figura 7.5. Colaborări în şablonul Command

- Client → ConcreteCommand
  - creează şi specifică primitorul;

- Invoker → ConcreteCommand (inițiază executarea comenzii);
- ConcreteCommand → Receiver (realizează acțiunea corespunzătoare comenzii).

Consecințe [Gamm94]

Avantaje:

- Decuplează Invoker de Receiver;
- Comenzile sunt obiecte ale unor clase
  - pot fi manevrate şi **extinse;**
- Comenzi compuse (Composite Commands)
  - a se vedea şi şablonul *Composite;*
- Uşor de adăugat comenzi noi
  - Invoker nu se modifică;
  - Codul este stabil (respectă Principiul Deschis-Închis).

Dezavantaje:

- Potențial pentru un exces de clase de tip Command.

Inteligența obiectelor Command [Gamm94]

- “Idioate"
  - delegă totul către Receiver;
  - folosite doar pentru a decupla Sender de Receiver;
- "Genii"
  - fac totul singure fără să delege nimic;
  - utile dacă nu există primitor (Receiver);
  - lasă ConcreteCommand să fie independentă de alte clase;
- “Istețe"
  - găsesc dinamic primitorul (Receiver).

Comenzi Anulabile (Undo-able) [Gamm94]

- Necesită memorarea unei stări adiționale pentru a face execuția reversibilă
  - obiectul primitor;
  - parametrii operației executate asupra primitorului;
  - valori originale în primitor ce se pot modifica în urma cererii
    - primitorul trebuie să ofere operații care să permită obiectului Command să se întoarcă în starea precedentă;
- Listă istorică
  - secvență de comenzi ce au fost executate
    - folosită ca LIFO cu execuție-inversă ⇒ undo;

- folosită ca FIFO cu execuție ⇒ redo;

- Comenzile pot necesita copiere
  - când starea comenzilor se schimbă prin execuție.

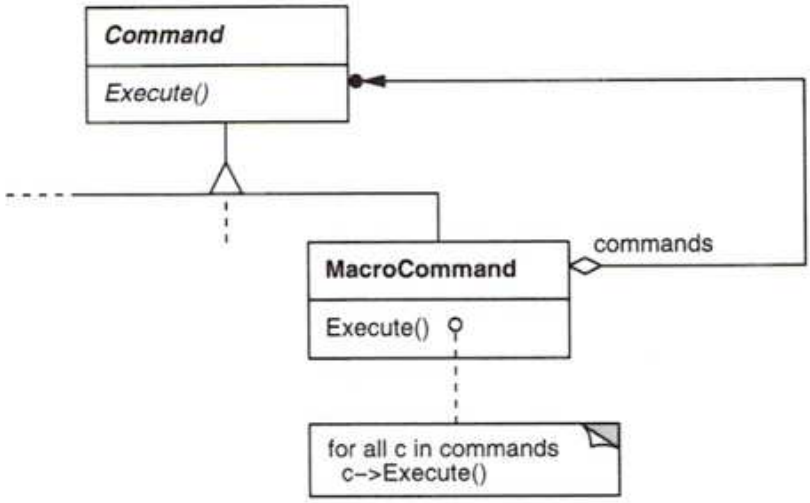


Figura 7.6. Comenzi compuse

Exemplul 1. Apeluri de Meniu

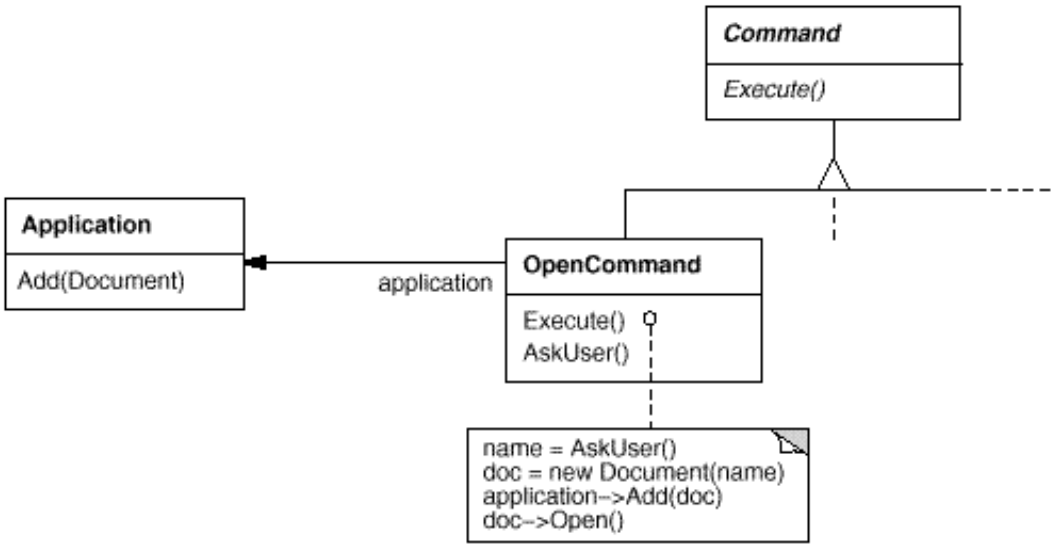


Figura 7.7. Apeluri de meniu

Exemplul 2[Coop98]. O interfață utilizator în Java poate conține diverse controale grafice care permit utilizatorului să trimită comenzi programului. La selectarea unui astfel de control grafic, programul primeşte un eveniment *ActionEvent*, pe care îl tratează implementând metoda *actionPerformed*. Să presupunem că

avem un program simplu care permite selectarea opțiunilor de meniu **File | Open** şi **File | Exit**, şi a unui buton Red care modifică fundalul în roşu (Figura 7.8).

Figura 7.8. Un meniu şi un buton

Programul conține obiectul de tip File Menu cu opțiunile (MenuItem) *mnuOpen* şi *mnuExit*, şi un buton numit *btnRed*. Un clic pe oricare dintre acestea determină un eveniment *actionPerformed* pe care îl putem capta cu următorul cod:

```
public void actionPerformed(ActionEvent e) {
Object obj = e.getSource();
if(obj == mnuOpen)
fileOpen(); //deschidere fisier
if (obj == mnuExit)
exitClicked(); //iesire din program
if (obj == btnRed)

redClicked(); //schimba in rosu
}
```

Cele trei metode *private* apelate de această metodă sunt:

```
private void exitClicked() {
System.exit(0);
}

private void fileOpen() {
FileDialog fDlg = new FileDialog(this, "Deschide fisier",
FileDialog.LOAD);
fDlg.show();
}

private void redClicked() {
p.setBackground(Color.red);
}
```

Dacă lucrăm cu zeci de opțiuni de mediu şi butoane, abordarea de mai sus va crea o structură decizională extinsă în interiorul *actionPerformed*, ce va fi greu de întreținut la extinderea interfeței. Folosirea şablonului Command va permite ca fiecare obiect să îşi primească comenzile direct. Un obiect Command are întotdeauna (minim) o metodă *Execute()* care este apelată la apariția unei acțiuni asupra obiectului:

```
public interface Command {
public void Execute();
}
```

Astfel, metoda *actionPerformed* se reduce la:

```
public void actionPerformed(ActionEvent e) {
Command cmd = (Command)e.getSource();
cmd.Execute();
}
```

Putem defini câte o metodă *Execute()* pentru fiecare obiect ce realizează o acțiune, păstrând astfel ceea ce se execută chiar în interiorul clasei respective. În plus, am separat interfața grafică de acțiunile pe care aceasta le inițiază, şi componentele grafice sunt de asemenea separate între ele. Cel mai simplu mod de a construi obiecte de tip Command este să derivăm clase noi din clasele MenuItem şi Button, care să implementeze fiecare interfața Command:

```
class btnRedCommand extends Button
implements Command {
public btnRedCommand(String caption) {
super(caption); //initialize the button
}
public void Execute() {
p.setBackground(Color.red);
}
}

class fileExitCommand extends MenuItem
implements Command {
public fileExitCommand(String caption) {
super(caption); //initialize the Menu
}
public void Execute() {
System.exit(0);
}
}
```

Metoda *actionPerformed* se va simplifica dar va fi necesar să instanțiem câte o clasă pentru fiecare acțiune pe care dorim să o executăm:

```
        mnuOpen.addActionListener(new fileOpen());
        mnuExit.addActionListener(new fileExit());
        btnRed.addActionListener(new btnRed());
...
```

Exemplul 3. Macrocomenzi [Ecke02]

În exemplul următor, şablonul Command permite înlănțuirea într-o coadă a unui set de acțiuni, pentru a fi executate împreună (macrocomenzi).

```
        import java.util.*;
        import junit.framework.*;

        interface Command {
          void execute();
        }

        class Hello implements Command {
          public void execute() {
            System.out.print("Hello ");
          }
        }

        class World implements Command {
          public void execute() {
            System.out.print("World! ");
          }
        }

        class IAm implements Command {
          public void execute() {
            System.out.print("Sunt şablonul command!");
          }
        }

        // Un obiect ce reține comenzile:
        class Macro {
          private List commands = new ArrayList();
          public void add(Command c) { commands.add(c); }
           public void run() {
            Iterator it = commands.iterator();
```

```
    while(it.hasNext())
      ((Command)it.next()).execute();
  }
}
```

Observați cum putem crea dinamic comportamente noi, folosind obiectele-comenzi pe post de parametri:

```
public class CommandPattern   {
  Macro macro = new Macro();

    macro.add(new Hello());
    macro.add(new World());
    macro.add(new IAm());
    macro.run();
  }
  public static void main(String args[]) {
    //…
  }}
```

Exerciții.

1. O interfață conține o listă cu nume de persoane în stânga unei ferestre Swing, şi trei butoane de tip JButton în dreapta listei: **Date personale** (Vârstă, Adresă, Tel) (afişează informațiile citite dintr-o tabelă MySQL), **CV** şi **Eseu** (afişează CV-ul şi respectiv, eseul compus de candidat citite din fişiere). Folosiți şablonul Command în implementarea comenzilor asociate componentelor grafice active.

2. Extindeți Exercițiul 1 prin crearea unei macrocomenzi "**Informații candidat**" care cumulează afişarea tuturor informațiilor asociate cu cele trei butoane (Date personale, CV, Comenzi compuse), şi care se declanşează la apăsarea unui buton cu acelaşi nume.

7.3. Şablonul MEMENTO

Scop şi aplicabilitate

Şablonul Memento captează şi externalizează starea internă a unui obiect astfel încât obiectul poate fi restaurat în această stare mai târziu, fără a sparge încapsularea. În sistemele ce trebuie să suporte operații *undo* este necesar să reținem starea integrală a unui obiect fără să spargem încapsularea altor obiecte şi fără să-i expunem public interfața: şablonul Memento rezolvă această problemă printr-un acces privilegiat la starea obiectului (ce trebuie salvată).

Folosiți şablonul Memento atunci când:

- (o parte din) starea unui obiect trebuie salvată pentru a readuce mai târziu obiectul în această stare;
- o interfață directă pentru obținerea stării ar expune detalii de implementare şi ar sparge încapsularea obiectului.

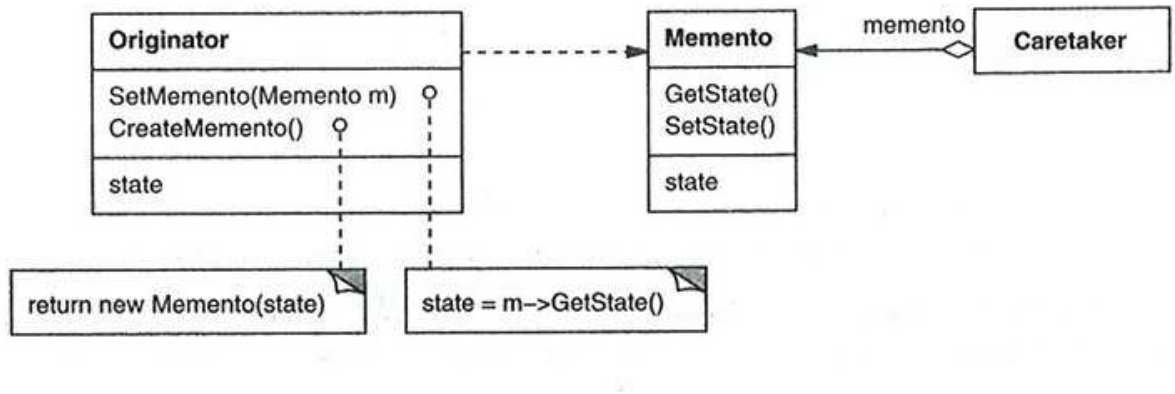


Figura 7.9. Structura şablonului Memento

Participanți [Gamm94, Fig. 7.9]

- Memento
  - Stochează starea internă a obiectului Originator (oricât este necesar din aceasta);
  - Protejează împotriva accesului din partea altor obiecte decât Originator (Originator vede o interfață extinsă din Memento, pentru a-şi reface starea; Caretaker vede doar o interfață restrânsă- cât să poate trece un obiect Memento obiectelor care au nevoie de el);
- Originator
  - Reprezintă obiectul a cărui stare dorim să o salvăm;
  - Creează un Memento conținând un extras al stării sale interne;
  - Foloseşte Memento pentru a-şi reface starea internă;
- Caretaker (mecanism de *undo*)
  - Răspunde de păstrarea în siguranță a Memento-ului;
  - Nu examinează şi nu operează niciodată cu conținutul Memento-ului;
  - Gestionează momentul salvării stărilor, salvează Memento, şi dacă este necesar, foloseşte Memento pentru a restabili starea obiectului Originator.

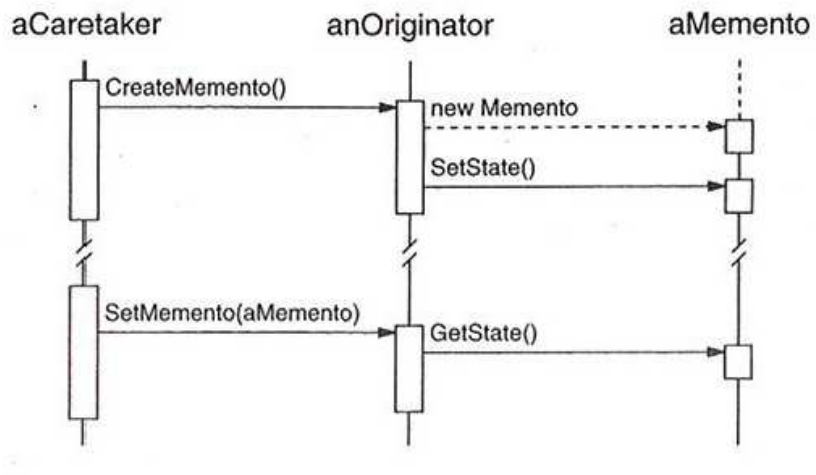


Figura 7.10. Colaborări în şablonul Memento

- Un Caretaker cere un Memento de la un Originator, îl păstrează un timp, şi apoi îl dă înapoi la Originator (vezi diagrama de interacțiune din Fig. 7.10)- dacă Originatorul trebuie să revină într-o stare anterioară;
- Memento-urile sunt pasive. Doar Originatorul care a creat un Memento îi va asigna o stare sau va prelua o stare de la el.

Consecințe [Gamm94]

- Protejează granițele încapsulării;
- Simplifică Originatorul
  - Acesta nu e nevoit să rețină stări anterioare la care ar fi necesar să revină;
- Uneori folosirea este costisitoare
  - Dacă este multă informație de copiat sau atunci când clienții creează des Memento-uri;
- Costuri ascunse în manevrarea Memento-urilor
  - Un Caretaker nu ştie exact cât să aloce pentru stocarea unui Memento (pentru că el nu are acces la informațiile din interior).

Implementare [Gamm94]

- Suport în limbajul de programare:
  - În C++ Originator se declară *friend* pentru Memento, iar interfața Memento este *private* (cu excepția părții restrânse care trebuie ştiută şi de Caretaker şi care este declarată *public*);
- Memorarea schimbărilor incrementale
  - Când Memento-urile se creează şi se întorc spre Originator într-o secvență predictibilă, Memento poate reține doar schimbările incrementale în starea internă a Originatorului.

Şabloane relaționate [Gamm94]

- Command

- Poate folosi Memento pentru a menține starea pentru operații ce se pot revoca (undo-able);

- Iterator
  - Memento se poate folosi pentru iterații.

Exemplul 1. [Coop98]

Acest exemplu prezintă un program grafic simplu ce creează dreptunghiuri şi permite selectarea şi mutarea lor cu mouse-ul. Interfața are 3 butoane: Rectangle, Undo şi Clear (Figura 7.11).

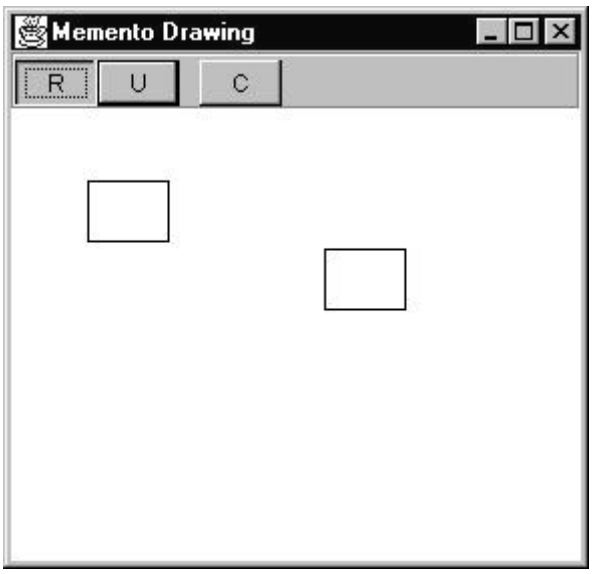


Figura 7.11. Exemplul 1

Butonul Rectangle este un JToggleButton ce stă selectat până dăm click cu mouse-ul să desenăm un nou dreptunghi. Odată desenat dreptunghiul, poate fi selectat cu click; odată selectat poate fi mutat cu mouse-ul (Figura 7.12).

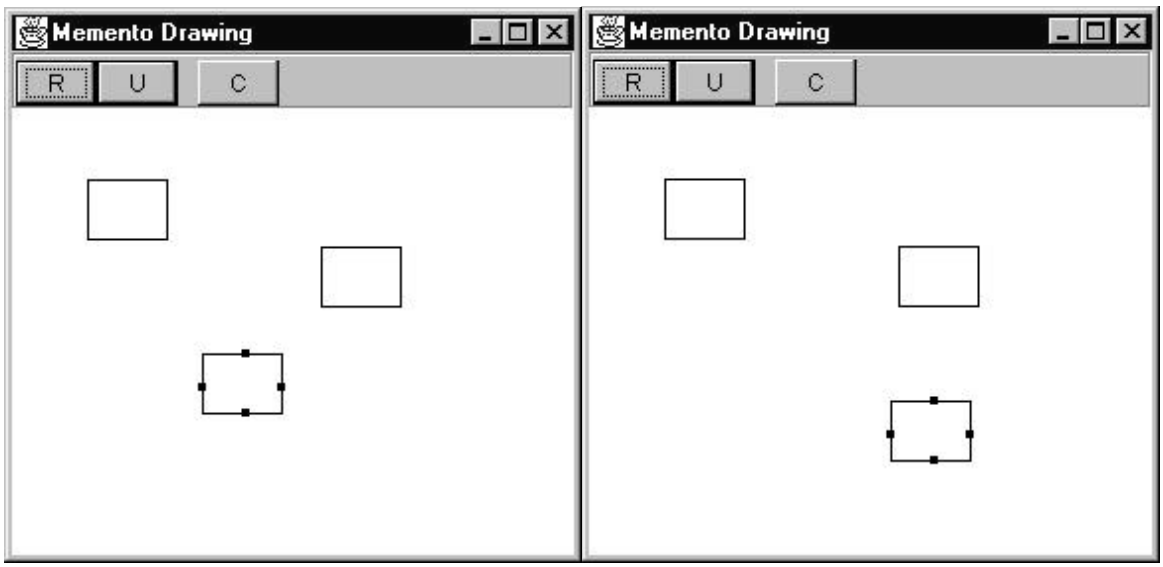


Figura 7.12. Exemplul 1 (cont.)

Butonul Undo poate reface o succesiune de operații: mutarea şi crearea de dreptunghiuri. Sunt 5 acțiuni la care trebuie să răspundem:

1. Rectangle button click
2. Undo button click
3. Clear button click
4. Mouse click
5. Mouse drag

Cele 3 butoane se construiesc ca obiecte Command, operațiile de click şi drag cu mouse-ul pot fi tratate tot ca nişte comenzi –putem folosi şablonul Mediator- (Secțiunea 7.5). În Mediator se va trata şi lista acțiunilor Undo (reține lista ultimelor n operații, pentru a fi anulate) deci Mediatorul funcționează şi ca obiect Caretaker (descris anterior). În acest exemplu salvăm şi anulăm (undo) doar 2 acțiuni: crearea de dreptunghiuri noi şi schimbarea poziției dreptunghiurilor.

```
public class visRectangle
{int x, y, w, h;
Rectangle rect;
boolean selected;

public visRectangle(int xpt, int ypt) {
x = xpt; y = ypt; //salvează locația
w = 40; h = 30; //folosește dimensiunile implicite
saveAsRect();}


public void setSelected(boolean b) {
```

```
selected = b;}


private void saveAsRect() {
//converteşte în dreptunghi pentru a putea folosi metoda contains
rect = new Rectangle(x-w/2, y-h/2, w, h);}

public void draw(Graphics g) {
g.drawRect(x, y, w, h);
if (selected) { //desenează “handles”
g.fillRect(x+w/2, y-2, 4, 4);
g.fillRect(x-2, y+h/2, 4, 4);
g.fillRect(x+w/2, y+h-2, 4, 4);
g.fillRect(x+w-2, y+h/2, 4, 4);}}
//-----------------------------------------

public boolean contains(int x, int y) {
return rect.contains(x, y);}
//-----------------------------------------

public void move(int xpt, int ypt) {
x = xpt; y = ypt;
saveAsRect();}}
```

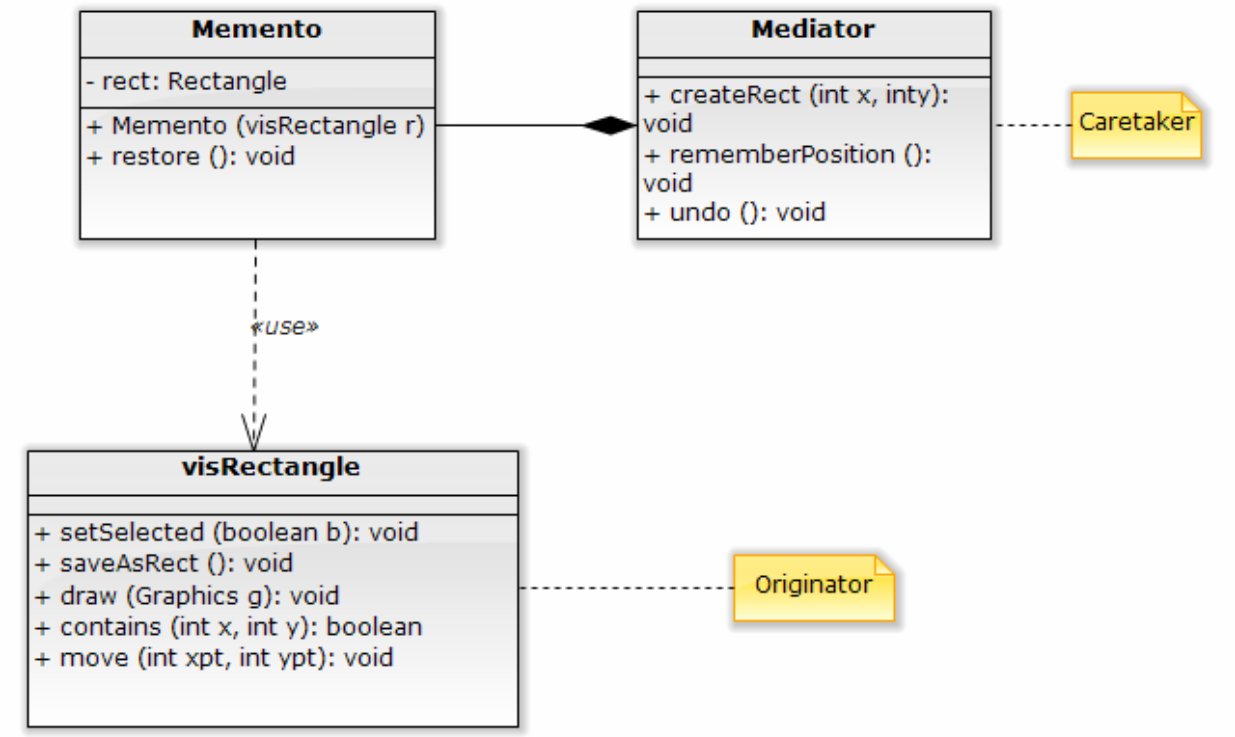


Figura 7.13. Exemplul 1

Clasa Memento:

```
class Memento
{visRectangle rect;
//câmpuri salvate- memorează câmpuri interne pentru dreptunghiul specificat

int x, y, w, h;
public Memento(visRectangle r) {
rect = r; //salvează o copie a instanței in data membra rect
x = rect.x; y = rect.y; //salvează poziția
w = rect.w; h = rect.h; //și dimensiunea
}

public void restore() {
//restaurează starea internă a dreptunghiului specificat
rect.x = x; rect.y = y; //restaurare poziție
rect.h = h; rect.w = w; //restaurare dimensiune
}}
```

Când instanțiem clasa Memento, îi transmitem instanța de visRectangle pe care dorim s-o salvăm. Memento copiază dimensiunea şi poziția şi salvează o copie a instanței visRectangle. Când vom dori să restaurăm aceşti parametri mai târziu, Memento-ul ştie ce instanță trebuie să restaureze şi poate face aceasta direct (vezi metoda *restore()*). Restul activităților sunt repartizate în clasa Mediator, unde salvăm starea precedentă a listei desenelor ca Integer (în lista undo, conform codului următor).

```
public void createRect(int x, int y)
{unpick(); //asigurare ca nici un dreptunghi nu este selectat
if(startRect) //butonul Rect apăsat
     {Integer count = new Integer(drawings.size());
       undoList.addElement(count); //Salvează dimensiunea precedentă a listei
       visRectangle v = new visRectangle(x, y);
       drawings.addElement(v); //adaugă un element nou în listă
       startRect = false; //gata cu acest dreptunghi
       rect.setSelected(false); //un-click buton
       canvas.repaint();}
else
pickRect(x, y); //dacă butonul Rect nu este apăsat caută un dreptunghi de selectat
}
```

Salvăm poziția precedentă a unui dreptunghi înainte de a-l muta într-un Memento:

```
public void rememberPosition()
{if(rectSelected){
Memento m = new Memento(selectedRectangle);
undoList.addElement(m);
```

```
}}
```

Metoda *undo* decide dacă să reducă lista de desenare cu 1 sau să invoce metoda *restore* a unui Memento:

```
public void undo()
{if(undoList.size()>0)
{
//obține ultimul element din lista undo
Object obj = undoList.lastElement();
undoList.removeElement(obj); //șterge

//dacă este Integer, ultima acțiune a fost un nou dreptunghi
if (obj instanceof Integer)
{
//șterge ultimul dreptunghi creat
Object drawObj = drawings.lastElement();
drawings.removeElement(drawObj);
}
//dacă acesta este un Memento, ultima acțiune a fost o mutare
if(obj instanceof Memento)
{
//obține  Memento-ul
Memento m = (Memento)obj;
m.restore(); //și restaurează poziția veche
}
repaint();}
}
```

Exerciții.

1. Finalizați programul din Exemplul 1 şi extindeți-l astfel încât să deseneze şi cercuri în afară de dreptunghiuri.

2. Într-un joc de tip găsirea unui cuvânt în interiorul unei matrici de litere, se dă o matrice de butoane marcate cu litere şi trebuie să formați cuvinte folosind aceste butoane (cuvintele se pot forma pe verticală, orizontală sau diagonală, înainte sau înapoi). Pe măsură ce apăsați butoanele, acestea rămân apăsate, iar cuvântul care se formează este afişat într-o casetă sub grila de butoane. Dacă în final (când considerați că ați ghicit cuvântul şi apăsați un buton OK), cuvântul este găsit în dicționarul jocului, el este afişat într-o listă în dreapta şi vă creşte punctajul. Implementați posibilitatea de revocare a ultimelor litere tastate –ultima, ante-penultima (atenție, acestea trebuie şterse şi din caseta de sub butoane).

7.4. Şablonul OBSERVER

Scop şi aplicabilitate

Şablonul Observer defineşte o dependență de tipul unul-la-mai-mulți între obiecte astfel încât atunci când un obiect îşi schimbă starea, toate dependențele sale sunt anunțate şi actualizate automat. Şablonul se aplică atunci când:

- O abstracțiune are două aspecte, una dependentă de cealaltă; încapsularea acestor aspecte în obiecte diferite permite varierea şi refolosirea lor independentă;

- Modificarea unui obiect solicită modificarea altora, şi nu ştim exact câte obiecte trebuie modificate (modificări dinamice, la execuție);

- Un obiect trebuie să poată notifica alte obiecte fără să presupună cine sunt acestea (va rezulta cuplaj redus între aceste obiecte).

Partiționarea sistemului în colecții de clase cooperante are ca efect secundar necesitatea menținerii consistenței între obiectele relaționate. Pot exista mai mulți observatori, şi fiecare observator poate reacționa diferit la aceeaşi notificare. Sursa de date (subiectul) trebuie să fie cât mai decuplat posibil de observator(i), tocmai pentru a permite observatorilor să se modifice independent de subiect

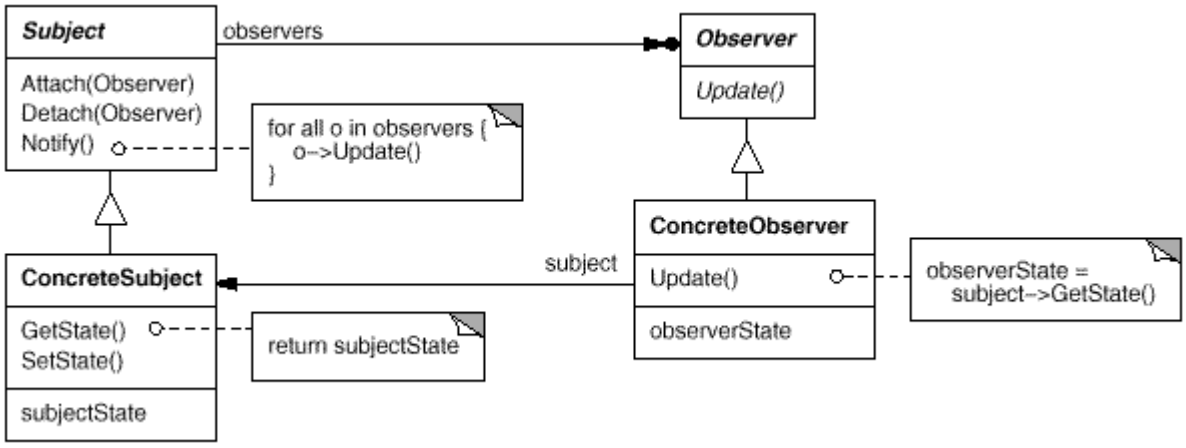


Figura 7.14. Structura şablonului Observer

Participanti [Gamm94, Fig. 7.14]

- Subject
  - îşi cunoaşte observatorii (în număr oarecare);
  - defineşte o interfață pentru ataşarea şi înlăturarea de obiecte Observer;

- Observer
  - defineşte o interfață de actualizare pentru obiectele ce trebuie notificate de modificările din subiect;
- ConcreteSubject
  - stochează starea de interes pentru obiectele ConcreteObserver;
  - trimite o notificare observatorilor săi când îşi modifică starea;
- ConcreteObserver
  - menține o referință către un obiect ConcreteSubject;
  - stochează starea ce trebuie să rămână consistentă cu a subiectului;
  - implementează interfața de actualizare din Observer pentru a-şi menține starea consistentă cu a subiectului.

Colaborări [Gamm94, Fig.7.15]

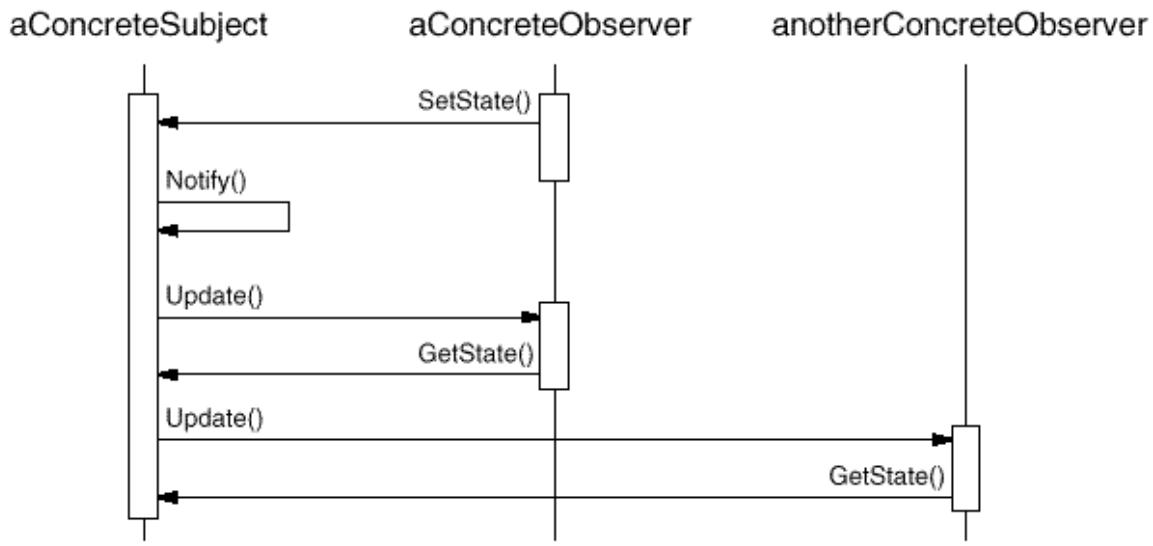


Figura 7.15. Colaborări în şablonul Observer

- ConcreteSubject îşi notifică observatorii de câte ori apare o modificare ce poate face starea observatorilor inconsistentă cu a sa;
- după ce a fost notificat de modificarea unui subiect, un ConcreteObserver interoghează subiectul pentru informații, pentru a-şi reconcilia starea cu a subiectului.

Consecințe [Gamm94]

Avantaje

- Cuplaj minimal şi abstract între subiect şi observatori (pot fi refolosite independent mai uşor în alte aplicații);
- Suport pentru comunicare de tip difuzare (“broadcast”)

- Notificarea trimisă de subiect nu trebuie să-şi specifice primitorul, singura responsabilitate a subiectului fiind să-şi notifice observatorii, indiferent de numărul lor. Observerul decide dacă tratează sau ignoră notificarea.

Dezavantaje

- Întrucât observerii nu ştiu unul de existența celuilalt, pot apare modificări incorecte, în cascadă (modificări neaşteptate).

Implementare [Gamm94]

- Maparea subiectelor cu observatorii
  - Referințe la observatori stocate în subiect;
  - Multe subiecte, puțini observatori: căutare asociativă cu tabele hash;
- Mai multe subiecte observate simultan de un observator
  - Parametru al metodei *update()*, pentru precizarea subiectului;
- Starea Subiectului trebuie să fie consistentă înainte de notificare;
- Cine declanşează actualizarea? (Ce obiect apelează “notify”?)
  - Operațiile de modificare a stării din Subiect, după schimbarea stării
    - Clienții nu trebuie să-şi amintească să apeleze “notify”;
    - Ineficient: Operații consecutive vor duce la actualizări consecutive;
  - Clienții apelează Notify
    - Evită actualizările intermediare ne-necesare;
    - Responsabilități suplimentare pentru clienți-pot uita să apeleze Notify;
- Un subiect trebuie să-şi informeze observatorii când se şterge;
- Încapsularea unei semantici complexe de actualizare:
  - Când relațiile de dependență între subiecte şi observatori sunt foarte complexe, poate fi necesar un obiect pentru menținerea acestor relații: **ChangeManager;**
  - Exemplu: dacă o operație implică schimbări în mai multe subiecte interdependente, trebuie ca observatorii să fie notificați doar **o singură dată**, **după** ce s-au modificat **toate** subiectele;
  - Se poate implementa folosind şablonul Mediator;
- ChangeManager are 3 responsabilități:
  - Asociază un subiect cu observatorii săi şi oferă o interfață de menținere a acestei asocieri, astfel că observerii nu necesită referințe către subiect, nici viceversa;
  - Defineşte o strategie particulară de actualizare;
  - Actualizează toți observerii dependenți, la cererea unui subiect.

ChangeManager poate fi de două tipuri**:**

- SimpleChangeManager
  - Naiv: actualizează toți observerii unui subiect;

- Util când nu avem actualizări multiple;

- DAGChangeManager
  - Tratează grafuri direcționate aciclice (GDA) de dependențe între subiecte şi observatorii lor;
  - Preferabil când un observator observă mai mult de un subiect, pentru a ne asigura că observatorul primeşte o singură actualizare (o modificare în două sau mai multe subiecte poate provoca actualizări redundante).

Exemplul 1. Exemplul prezentat în continuare rezolvă problema consistenței date-vederi – Figura 5.1, Capitolul 5).

Să presupunem că avem în tabela unei BD informații despre notele unor studenți, pe care le putem vizualiza grafic sub forma tabelară (clasa SpreadSheetView).

Nume şi prenume	**Materia 1**	**Materia 2**	**Materia 3**
Ionescu Ion	9	10	7
Popescu Petru	6	8	7
Vasilache Dumitru	5	6	6

Tabel 7.1

Să mai presupunem că implementarea care realizează comunicarea între datele propriu zise şi afişarea lor foloseşte interfața:

```
interface GradeDBViewer{
void update(String course, String name, real grade); }
```

La introducerea de date noi sau la modificarea celor existente (spre exemplu, măriri de note), informațiile trebuie transmise către vizualizarea grafică:

```
SpreadSheetView ssv= new SpreaSheetView();
ssv.update(“Materia 1”, “Popescu Petru”, 8);
```

Dacă ulterior dorim şi o vizualizare a mediilor notelor pe materii sub forma unui BarGraf – atunci codul de actualizare a afişărilor va arăta astfel:

```
SpreadSheetView ssv= new SpreaSheetView();
BarGraphView bgv= new BarGraphView();
…
ssv.update(“Materia 1”, “Popescu Petru”, 8);
bgv.update(“Materia 1”, “Popescu Petru”, 8);
```

Adăugarea altor tipuri de vizualizări va atrage după sine alte modificări în cod; dacă dorim să le evităm, baza de date ar trebui să mențină o listă de observatori pe care sa-i notifice de modificarea stării sale:

```
Vector observers= new Vector();
…
for (int i=0; i<observers.size();i++){
GradeDBViewer v= (GradeDBViewer) observers[i];
v.update(“Materia 1”, “Popescu Petru”, 8);}
```

Pentru a inițializa vectorul de observatori, baza de date ar trebui să furnizeze două metode adiționale: *register* pentru adăugarea unui observator, şi *remove* pentru a şterge un observator:

```
void register(GradeDBViewer observer){
observers.add(observer);}

boolean remove(GradeDBViewer observer) {
return observers.remove(observer);
}
```

Şablonul Observer permite ca astfel de ştergeri/ adăugări să se efectueze chiar la momentul execuției. Baza de date poate transmite toate informațiile relevante către observatorii interesați la momentul apariției (structura *push*), sau poate notifica doar că a apărut o modificare lăsând observatorii să-şi extragă singuri informațiile de interes (structura *pull*), rezultând mai multe mesaje de comunicare, dar un volum mai mic de date transferate. Folosind Observer, putem refolosi prezentarea grafică şi datele independent unele de altele în aplicații noi.

Exerciții

1. Personajul unui joc de tip arcade trebuie să evite să fie atins de proiectilele lansate la intervale egale de un tun cu poziție fixă. Personajul se deplasează în cele patru direcții principale folosind tastele cu săgeți. Tunul urmăreşte în pemanență poziția personajului în câmpul de joc pentru a redirecționa tirul. Folosiți Observer, considerând că personajul este subiectul observat.

2. In jocul Hungry Hippau (Cap1, Ex. 3), coliziunea unui buştean cu hipopotamul are drept efect scăderea numărului de vieți (din cele trei existente inițial), respectiv pierderea jocului dacă se ajunge la 0 vieți. Implementați Observer considerând că buştenii sunt observatorii hipopotamului.

3. Un personaj al unui joc de război (un invadator) lansează proiectile odată la 500ms, cu ajutorul unui obiect de tip Timer (clasa java Timer). Considerați că această clasă de tip Timer este subiectul, iar personajul este Observatorul său.

4. În jocul Tetris, implementați panoul de joc ca observator al formelor în mişcare, pentru a reactualiza ecranul la completarea unei linii.

7.5. Şablonul MEDIATOR

Scop şi aplicabilitate

Şablonul Mediator defineşte un obiect ce încapsulează cum interacționează o mulțime de obiecte. Şablonul promovează cuplajul redus prin faptul că obiectele nu mai fac referință direct unele la altele, permițând varierea independentă a interacțiunii lor.

Programarea orientată- obiect încurajează distribuirea comportamentului între obiecte şi astfel se creează multe conexiuni între obiecte. Deşi partiționarea sistemului în obiecte măreşte reutilizabilitatea, proliferarea interconexiunilor tinde să o reducă. Spre exemplu, există de multe ori dependențe între componentele grafice ale unei casete de dialog; deşi unele ferestre pot arăta identic, comportamentul diferă, şi nu putem refolosi interfața. Soluția constă în încapsularea comportamentului colectiv într-un obiect mediator separat. Mediatorul controlează şi coordonează interacțiunile grupului de obiecte, fiind un intermediar ce previne ca obiectele să se refere explicit unele la altele (ele cunosc doar mediatorul şi comunică indirect prin intermediul lui).

Folosiți Mediator atunci când:

- O mulțime de obiecte comunică într-un mod bine definit dar complex;
  - Interdependențele sunt nestructurate şi dificil de înțeles;
- Refolosirea unui obiect este dificilă deoarece comunică cu prea multe alte obiecte;
- Un comportament distribuit între mai multe clase ar trebui să fie readaptabil fără derivare;

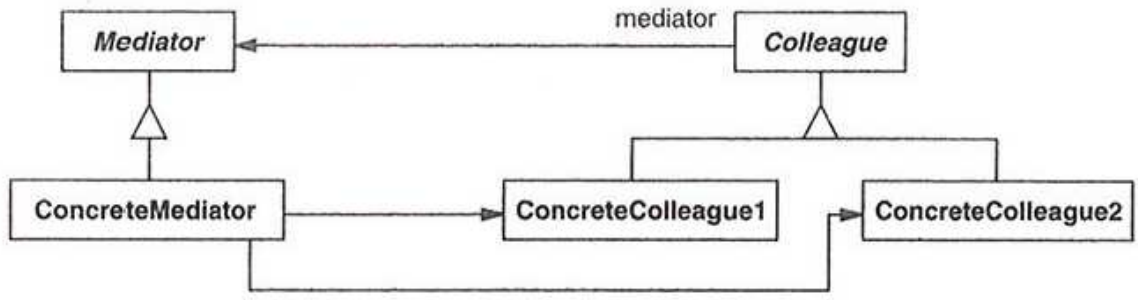


Figura 7.16. Structura şablonului Mediator

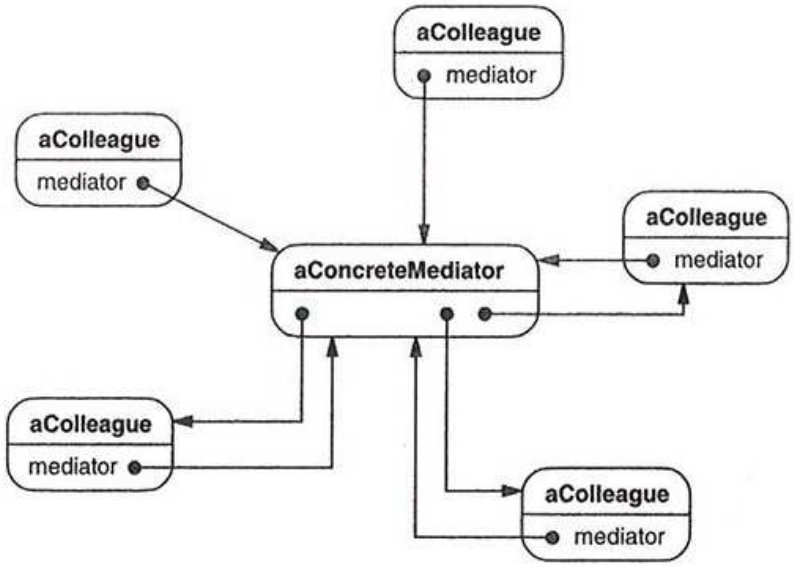


Figura 7.17. Structură tipică de obiecte

Participanți & Colaborări [Gamm94, Fig.7.16]

- Mediator
  - Defineşte o interfață pentru comunicarea cu obiectele Colleague;
- ConcreteMediator
  - Implementează comportamentul cooperativ prin coordonarea obiectelor Colleague;
  - Îşi cunoaşte şi îşi menține “colegii”;
- Clasele Colleague
  - Fiecare clasă Colleague îşi cunoaşte obiectul Mediator;
  - Fiecare coleg comunică cu mediatorul său oricând ar fi comunicat cu un coleg.

Consecințe [Gamm94]

Avantaje

- Limitează derivarea
  - Modificarea comportamentului necesită doar derivarea Mediatorului, clasele Colleague pot fi refolosite ca atare;
- Decuplează colegii
  - Putem varia şi refolosi clasele Colleague şi Mediator independent;
- Simplifică protocoalele obiectelor
  - Înlocuieşte interacțiunile de tip „mai-mulți-la-mai-mulți” cu interacțiuni de tip „unul-la-mai-mulți” (mai uşor de înțeles, extins, menținut);
- Abstractizează cooperarea obiectelor
  - Ajută la clarificarea interacțiunilor independent de comportamentul individual.

Dezavantaje

- Centralizează controlul
  - Mediatorul se poate transforma într-un monolit dificil de întreținut.

Implementare [Gamm94]

- Când există un singur mediator, putem omite clasa abstractă Mediator;
- Comunicarea Colleague-Mediator:
  1. Se poate folosi şablonul Observer: clasele colegi sunt subiecte, mediatorul este observator care este notificat de modificări (la apariția unor evenimente în colegi);
  2. Se poate defini o interfață specializată de notificare în Mediator, ce permite colegilor să fie mai direcți în comunicare (un coleg se transmite pe sine ca argument în comunicarea cu mediatorul pentru a permite mediatorului să-l identifice).

Exemplul 1 [James Cooper]

În interfața din Figura 7.18, selectarea unei opțiuni din listă copiază textul în caseta TextBox (şi viceversa: scrierea unui text valid în TextField selectează opțiunea corespunzătoare din listă). Dacă există text în câmpul de text, alte butoane devin active permițând formatarea sau ştergerea de text etc. Mediatorul este un „centru de comunicare” pentru componentele grafice (Figura 7.19).

Figura 7.18. Exemplul 1

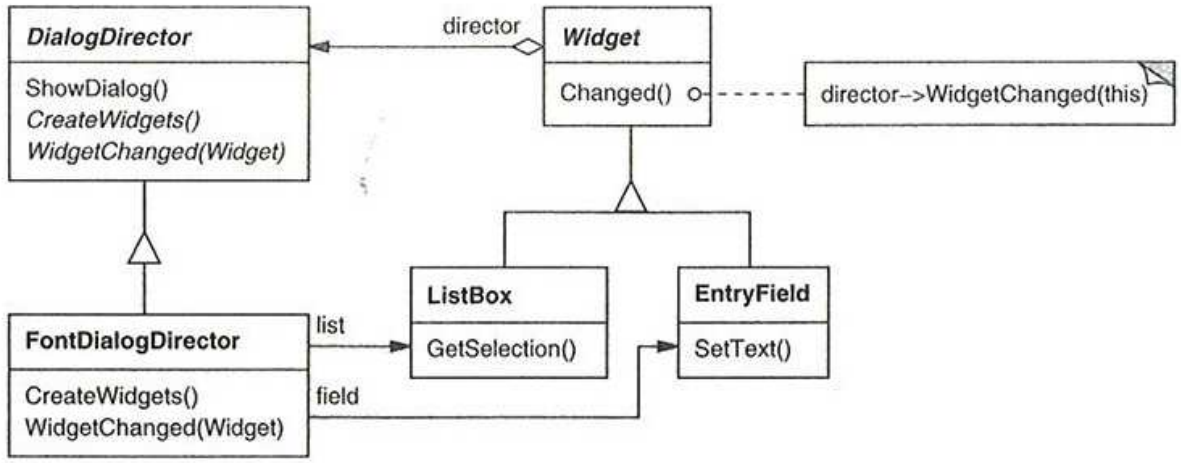


Figura 7.19. Diagrama de clase- Exemplul 1

În Figura 7.20, clasa DialogDirector este Mediatorul, FontDialogDirector este ConcreteMediator, iar ListBox şi EntryField sunt „colegii”.

O colaborare

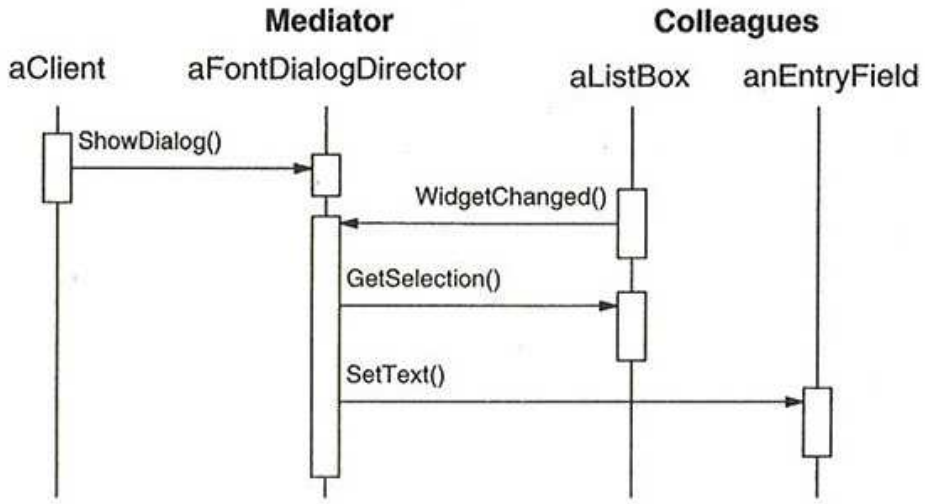


Figura 7.20. O colaborare în Exemplul 1

```
class DialogDirector {
public:
   virtual ~DialogDirector();
   virtual void ShowDialog();
   virtual void WidgetChanged(Widget *)=0;
```

```
protected:
   DialogDirector();
   virtual void CreateWidgets()=0;

};

class Widget{
public:
   Widget (DialogDirector*);
   virtual void Changed();

   virtual void HandleMouse(MouseEvent& event);
//…
private:
    DialogDirector* _director;};

void Widget::Changed(){
    _director->WidgetChanged(this); }
```

Subclasele lui DialogDirector suprascriu *WidgetChanged()* pentru a modifica componentele corespunzătoare. Widget transmite o referință către sine ca argument în *WidgetChanged()*, pentru a permite Directorului să identifice ce componentă s-a modificat.

Elementele specializate ale interfeței

```
class ListBox: public Widget{
public:
  ListBox(DialogDirector*);

 virtual const char* GetSelection();
 virtual void SetList(List<char*>* listItems);
 virtual void HandleMouse(MouseEvent& event);
   //…
};

class EntryField:  public Widget{
public:
  EntryField(DialogDirector*);

  virtual void SetText(const char* text);
  virtual const char* GetText();
  virtual void HandleMouse(MouseEvent& event);
//…};

class Button: public Widget{
public:
```

```
  Button(DialogDirector*);

virtual void SetText(const char* text);
virtual void HandleMouse(MouseEvent& event);
//…};


void Button::HandleMouse(MouseEvent& event){
//…
Changed();
}
```

Clasa de mediere: FontDialogDirector

```
class FontDialogDirector: public DialogDirector{
public:
 FontDialogDirector();
 virtual ~FontDialogDirector();
 virtual void WidgetChanged(Widget*);

protected:
   virtual void CreateWidgets();

private:
   Button* _ok;
   Button* _cancel;
   ListBox* _fontList;
   EntryField* _fontName;}

void FontDialogDirector::CreateWidgets() {
_ok=new Button(this);
_cancel=new Button(this);
_fontList=new ListBox(this);
_fontName=new EntryField(this);
//umplere lista cu nume
//asamblează componentele în dialog
}

void fontDialogDirector::WidgetChanged (Widget * theChangedWidget) {
if (theChangedWidget==_fontList)
{_fontName->SetText(_fontList->GetSelection());}
else if (theChangedWidget==_ok){
//aplică schimbarea de font şi închide  dialogul
}
else if (theChangedWidget==_cancel){
//închide dialogul
}}
```

Exerciții.

1. Folosiți şablonul Mediator pentru a coordona utilizarea Memento-ului în Exercițiile din Secțiunea 7.3.

2. Implementați o aplicație de chat robustă folosind şablonul Mediator şi structura din Figura 7.21.

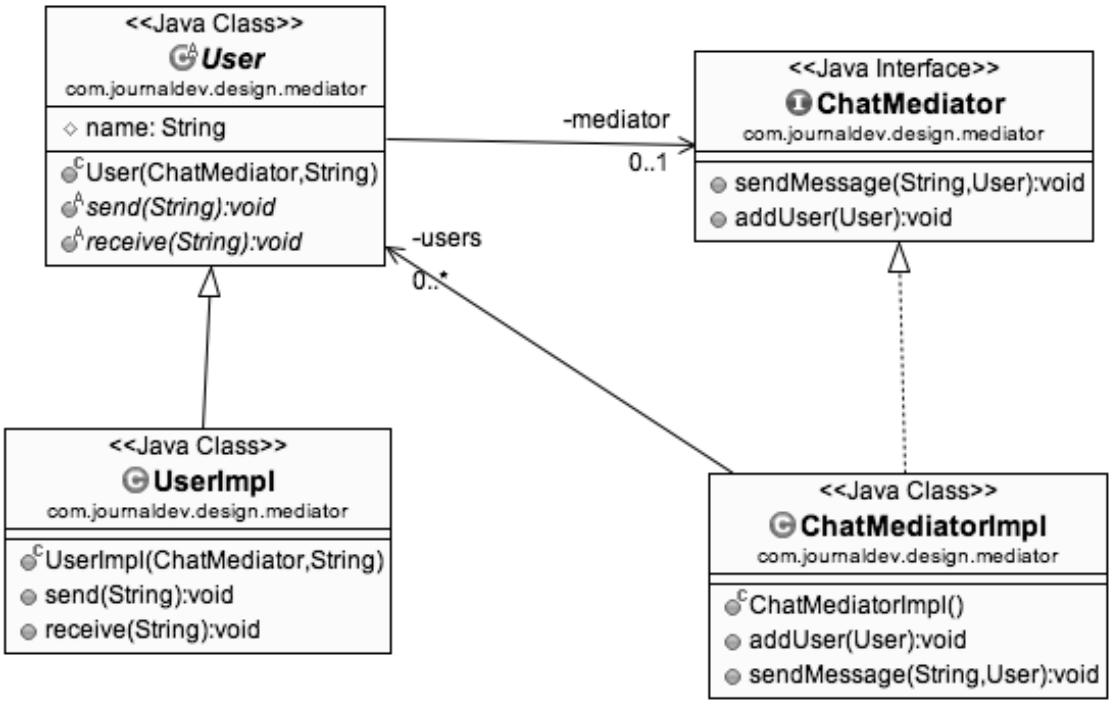


Figura 7.21. Ex.2: aplicație de chat robustă

3. Folosiți Mediator în implementarea Observatorilor din Exercițiile 1 şi 4 de la Secțiunea 7.4.

7.6. Şablonul STRATEGY

Scop şi aplicabilitate

Şablonul Strategy defineşte o familie de algoritmi, încapsulând fiecare algoritm şi făcându-i astfel interschimbabili. Şablonul permite algoritmului să varieze independent de clienții ce îl folosesc.

Folosiți şablonul Strategy atunci când:

- Sunt necesare variante diferite ale unui algoritm;
- Un algoritm foloseşte date despre care clienții nu trebuie să ştie
  - evită expunerea structurilor de date complexe, specifice algoritmului;
- Multe clase relaționate diferă doar prin comportament

- configurează o clasă cu un comportament particular;

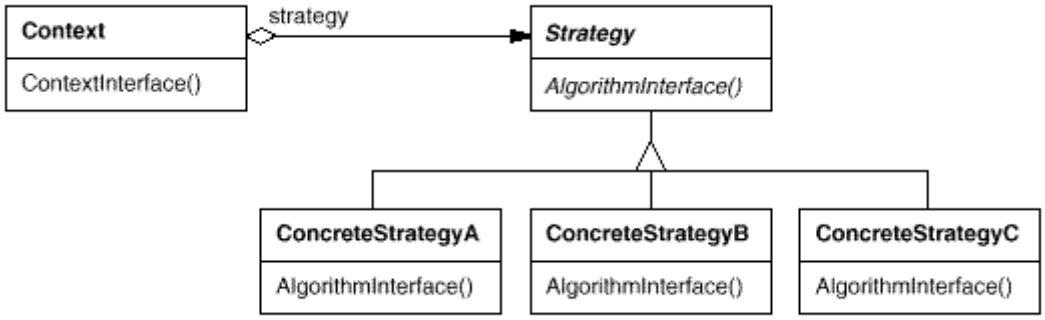


Figura 7.22. Structura şablonului Strategy

Participanți [Gamm94, Fig. 7.22]

- Strategy
  - declară o interfață comună tuturor algoritmilor;
  - Context foloseşte această interfață pentru a apela algoritmul definit de o clasă ConcreteStrategy;
- ConcreteStrategy
  - implementează algoritmul folosind interfața Strategy;
- Context
  - configurat cu un obiect ConcreteStrategy;
  - poate defini o interfață ce permite obiectelor Strategy să îi acceseze datele.

Consecințe pozitive [Gamm94]

- Familii de algoritmi înrudiți
  - de obicei oferă implementări diferite ale aceluiaşi comportament;
  - alegerea se face ținând cont de un compromis timp-spațiu;
- Alternativă la derivare;
- Elimină instrucțiunile condiționale.

ÎNAINTE	DUPĂ

`switch(flag){`
`  case A:doA();`
`break;`
`  case B:doB();`
`break;`
`  case C:doC();`
`break;}`
	`strategy.do();`

Figura 7.23. Eliminarea instrucțiunilor condiționale prin aplicarea Strategy

Consecințe negative [Gamm94]

- Exces de comunicare între Strategy şi Context
  - o parte din clasele ConcreteStrategy nu necesită informațiile trimise de Context;
- Clienții trebuie să fie conştienți de diferitele strategii
  - clienții trebuie să înțeleagă diferitele strategii;
- Număr sporit de obiecte
  - fiecare Context foloseşte obiectele sale concrete de strategie;
  - se poate reduce prin păstrarea strategiilor fără stare (împărțirea lor între mai multe obiecte).

Implementare [Gamm94]

- Cum circulă datele între Context şi Strategii?
  - Abordarea 1: datele sunt trimise către Strategy
    - decuplat, dar poate fi ineficient;
  - Abordarea 2: trimite Contextul însuşi şi lasă strategiile să îşi preia datele
    - Context trebuie să ofere acces mai cuprinzător la datele sale (deci este mai cuplat);
  - În Java ierarhia de strategii poate fi formată din clase interioare;
- Obiectul Strategy opțional
  - comportament implicit în Context
    - dacă este folosit comportamentul implicit nu este nevoie să creem obiectul Strategy;
  - folosim Strategy doar dacă nu ne place comportamentul implicit.

Exemplul 1. Java Layout Managers

Clasele container ale IUG din Java sunt de nivel înalt (ferestre, dialoguri, applet-uri) şi intermediare (panouri -Panels). Fiecare clasă container are un manager de poziționare (Layout Manager) ce determină dimensiunea şi poziția componentelor (Figura 7.24). Există aproximativ 20 tipuri de poziționări, şi aproximativ 40 de tipuri de containere: combinarea lor liberă prin moştenire ar duce la foarte multe subclase.

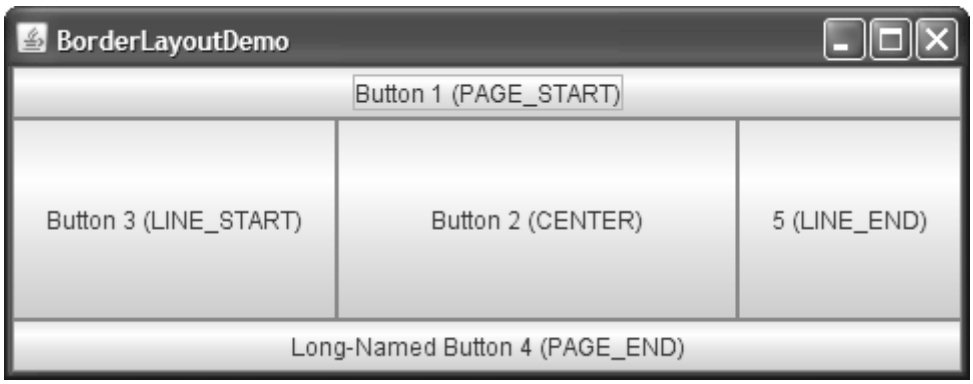


Figura 7.24. Diferiți manageri de poziționare în Java

Soluția Strategy

```
import java.awt.*;

class  BoxExample  extends Frame  {

public BoxExample(int  width, int height) {
setTitle( "Box Example" );
setSize( width, height );
setLayout(new BoxLayout()); //aplicarea managerului de poziționare
 for ( int label = 1; label < 10; label++ )
 add(new Button(String.valueOf( label )));
 show();  }

public  static  void  main(String  args[]) {
new  BoxExample( 185, 100 );
new  BoxExample( 165, 130 );  }}
```

Exemplul 2 [Ecke02]

Strategy mai poate adăuga un “Context”, care poate fi o clasă surogat care controlează selecția şi utilizarea unui obiect Strategy particular – ca în *State*. În continuare este prezentat un exemplu (fig. 7.25).

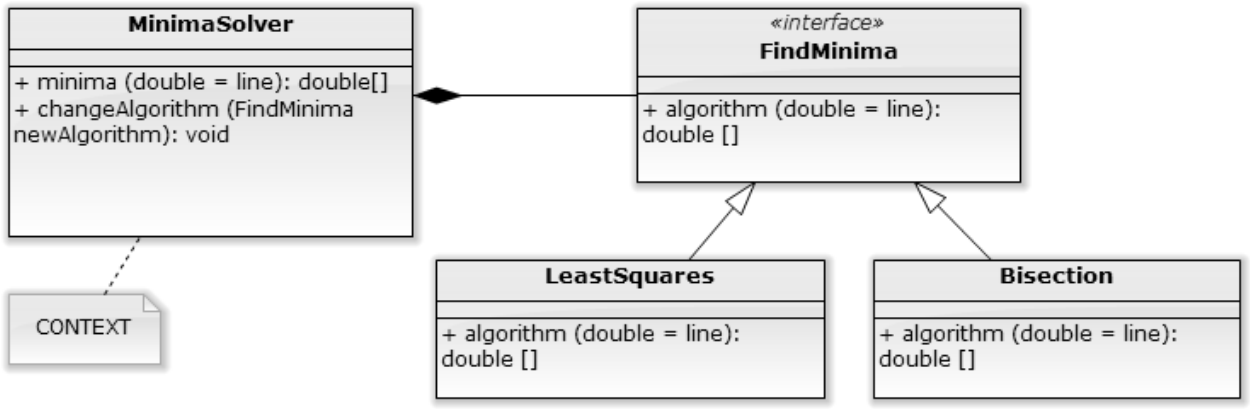


Figura 7.25. Exemplul 2: Algoritmi de calcul al minimului într-o funcție

```
// Interfața strategy :
interface FindMinima {
  // Linia e o secventa de puncte:
  double[] algorithm(double[] line);
}

// Diferitele strategii:
class LeastSquares implements FindMinima {
  public double[] algorithm(double[] line) {
    return new double[] { 1.1, 2.2 };
  }}


class Bisection implements FindMinima {
  public double[] algorithm(double[] line) {
    return new double[] { 5.5, 6.6 };
  }}


// "Contextul" controlează strategia:
class MinimaSolver {
  private FindMinima strategy;
  public MinimaSolver(FindMinima strat) {strategy = strat;}

  double[] minima(double[] line) {
    return strategy.algorithm(line);
  }
   void changeAlgorithm(FindMinima newAlgorithm) { strategy = newAlgorithm; }
}

public class StrategyPattern {
   MinimaSolver solver =new MinimaSolver(new LeastSquares());
  double[] line = { 1.0, 2.0, 1.0, 2.0, -1.0, 3.0, 4.0, 5.0, 4.0 };
```

```
 public void test() {
    System.out.println(Arrays2.toString(solver.minima(line)));
    solver.changeAlgorithm(new Bisection());
    System.out.println(Arrays2.toString(solver.minima(line)));
  }
  public static void main(String args[]) {
  //…
  }
}
```

Exerciții.

1. Sortați datele dintr-o listă de elemente de tip Pacient (nume, prenume, diagnostic) crescător după nume. Folosiți BubbleSort pentru liste sub 1500 înregistrări şi Quicksort pentru liste cu lungime >=1500.

2. O aplicație primeşte în linie de comandă numele unor fişiere cu următorul conținut: Nume …, Adresa…, Simptome…, şi le poate prelucra, (în funcție de un alt parametru citit) astfel încât să creeze pe baza lor fişiere în care rămân Numele, Adresa şi fie doar simptomele care sunt sugestive pentru specializarea Cardiologie, fie doar simptomele care sunt sugestive pentru specializarea Reumatologie (aplicația are memorate separat listele complete de simptome sugestive pentru aceste două specializari). Folosiți Strategy.

7.7. Şablonul STATE

Scop şi aplicabilitate

Şablonul permite unui obiect să îşi modifice comportamentul când i se modifică starea internă, astfel încât obiectul va părea că îşi schimbă clasa. Este aplicabil atunci când comportamentul obiectului depinde de starea sa, acesta trebuind să îşi modifice comportamentul la execuție în funcție de stare. Înlocuieşte operațiile cu instrucțiuni condiționale multiple, depinzând de starea obiectului (starea fiind reprezentată de una sau mai multe constante enumerate). Existența mai multor operații cu aceeaşi structură condițională este un indiciu de aplicabilitate/ necesitate a şablonului.

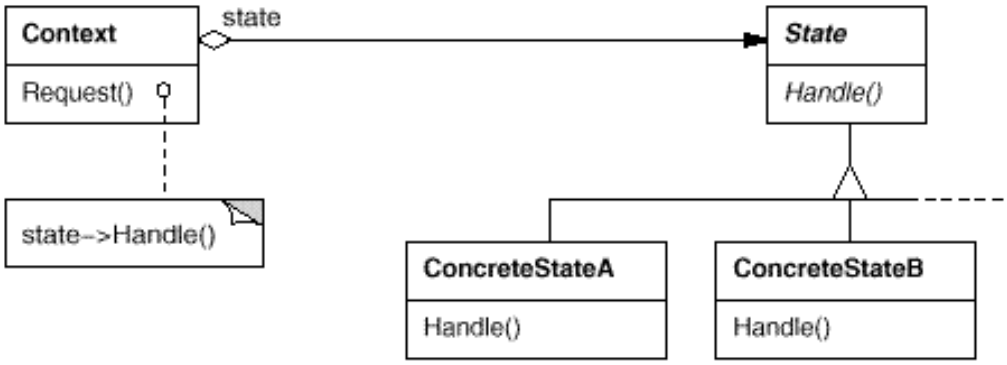


Figura 7.26. Structura şablonului State

Participanți [Gamm94, Fig.7.26]

- Context
  - definește interfața de interes pentru clienți;
  - menține o instanță a unei subclase ConcreteState;
- State
  - definește o interfață pentru încapsularea comportamentului asociat cu o stare particulară a Contextului;
- ConcreteState
  - fiecare subclasă implementează un comportament asociat cu o stare a Contextului.

Colaborări [Gamm94]

- Context deleagă cereri specifice stărilor către obiectele State
  - Contextul se poate transmite pe sine către obiectul State -dacă State trebuie să îl acceseze pentru a trata cererea;
- Tranzițiile stărilor sunt gestionate fie de Context, fie de State (vezi exemplele de mai jos) ;
- Clienții interacționează exclusiv cu Contextul, dar pot configura contextele cu stări (spre exemplu, definesc starea inițială).

Consecințe pozitive [Gamm94]

- Localizează comportamentul specific stărilor și partiționează comportamentul pentru stări diferite
  - pune tot comportamentul asociat unei stări într-un obiect-stare;
  - este uşor de adăugat stări şi tranziții noi
    - Contextul devine Deschis-Închis;
  - comportamentul este împrăştiat între mai multe subclase ale State
    - numărul de clase creşte, mai puțin compact decât o singură clasă;
    - este foarte util dacă sunt multe stări;

- Face explicită tranziția între stări
  - tranziția este mai mult decât o simplă modificare a unei valori interne;
  - stările primesc statutul unui obiect de sine stătător ;
  - protejează Contextul de stări interne inconsistente.

Şabloane relaționate: State versus Strategy

Ambele şabloane vizează modificarea conținutului unui obiect. Strategy **optimizează**, folosind un algoritm alternativ pentru implementarea comportamentului iar State **modifică** efectiv comportamentul la schimbarea stării obiectului.

- Rata de Modificare
  - Strategy
    - Obiectul Context de obicei conține unul din mai multe obiecte ConcreteStrategy posibile;
  - State
    - Obiectul Context îşi schimbă des obiectul ConcreteState în timpul duratei de viață;

- Expunerea Modificării
  - Strategy
    - Toate obiectele ConcreteStrategy fac acelaşi lucru, dar diferit;
    - Clienții nu văd nici o modificare de comportament în Context;
  - State
    - ConcreteState acționează diferit;
    - Clienții văd comportament diferit în Context.

Exemplul 1. Automat trecere metrou

Un automat la intrarea în metrou poate fi definit cu ajutorul maşinii de stări din Figura 7.27.

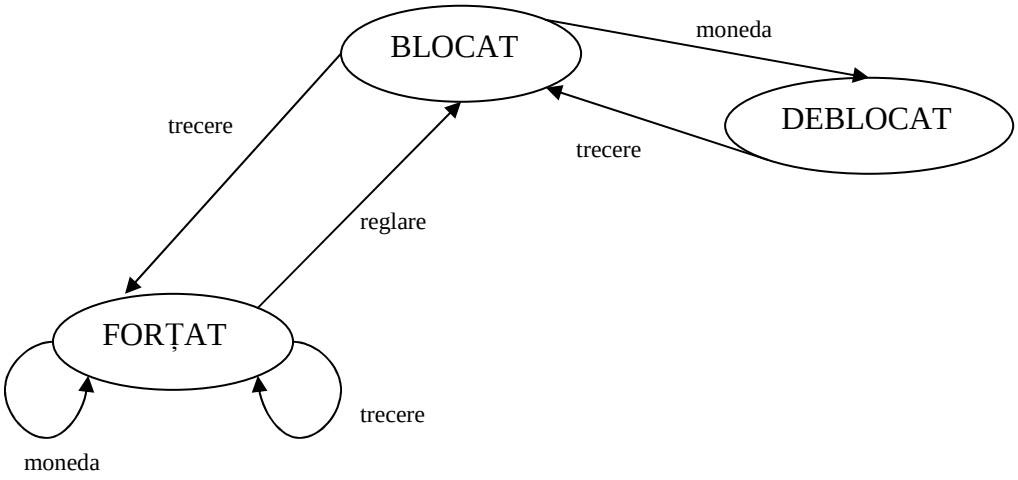


Figura 7.27. Maşina de stări a unui automat de metrou

În lipsa şablonului State, va trebui să folosim switch-uri complexe, funcţii lungi, şi aceleaşi switch-uri vor apare repetat în funcţii diferite.

```
class Automat_Metrou {
static final int BLOCAT = 2;
static final int DEBLOCAT= 3;
static final int FORTAT = 4;

private int state = BLOCAT;   //starea iniţială


public void moneda() {
switch (state) {
                        case BLOCAT:  state = DEBLOCAT; break;
                        case DEBLOCAT: afiseazaMultumesc(); break;
                        case FORTAT: apeleazaPersonalTehnic();   }   }

public void trecere () {
switch (state) {
                        case BLOCAT: state=FORTAT;  pornesteAlarma();  break;
                        case DEBLOCAT: state=BLOCAT; break;
                        case FORTAT: apeleazaPersonalTehnic();  }      }

public void reglare() {
if (state==FORTAT) {opresteAlarma(); state=BLOCAT;   }   }

public void pornesteAlarma() {…};
public void opresteAlarma() {…};
public void apeleazaPersonalTehnic() {…};
```

```
// . . .    }
```

Adăugarea unei noi stări automatului (de exemplu de diagnoză) va conduce la modificări în tot codul (comportamentul obiectului depinde de starea sa şi abordarea nu respectă Principiul Deschis-Închis).

Aplicarea State la Automat metrou

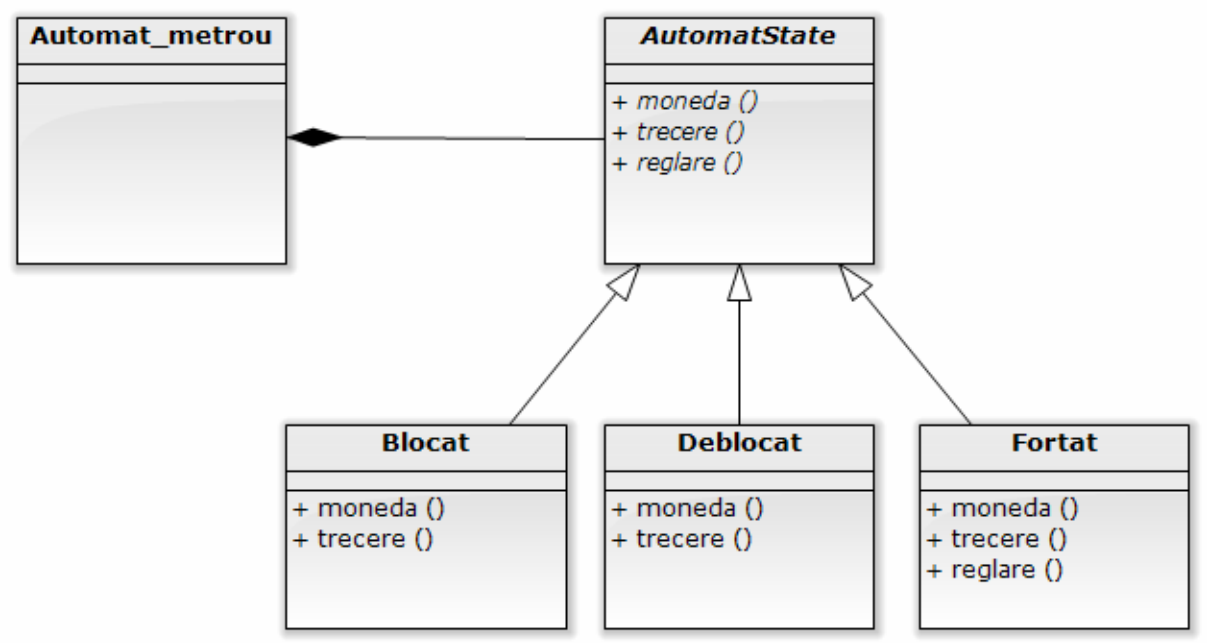


Figura 7.28. Stările Automat metrou

Clasele din Figura 7.28 au următoarele roluri (ale şablonului State):

- Automat_metrou este Context
- AutomatState este AbstractState
- Blocat, Deblocat şi Forţat sunt ConcreteStates

Observăm că toate stările şi comportamentul *real* sunt în AbstractState şi în subclase; acesta este un exemplu extrem. În general, Contextul are şi alte date şi metode, pe lîngă stările şi metodele State dar aceste date/ metode nu vor modifica stări. Doar unele aspecte ale Contextului îi vor modifica comportamentul.

```
class Automat_metrou {    //clasa Context
     private AutomatState state = new Blocat();

 public void moneda() {
       state = state.moneda();
      }
```

```
public void trecere() {
       state = state.trecere();
     }

public void reglare() {
       state = state.reglare();
     }

     }
   //  .  . .
```

Tranziția între Stări poate fi definită de către:

- Stările însele:
  - Este mai flexibil să lăsăm subclasele State să specifice următoarea stare (aşa cum am văzut şi în codul anterior: State specifică starea următoare);

- Contextul dacă:
  - Dorim să refolosim stările în maşini de stări diferite, cu tranziții diferite;
  - Criteriile de schimbare a stărilor sunt fixate.

Contextul modifică starea următoare

```
class Automat_metrou {private AutomatState state = new Blocat();

public void moneda( ) {
state.moneda();
state = new Deblocat();   }

public void trecere() {
if ( state.trecere() )        // intoarce true daca s-a trecut
    state = new Blocat( );
else
    state = new Fortat(); }

public void reglare( ) {
if (state.reglare())     //intoarce true daca a fost apelata in starea FORTAT
state = new Blocat();   }

}
```

Exemplul 2. Exemplul de Swing LayoutManager (Exemplul 1, Secțiunea 7.6) este un exemplu care ilustrează atât comportamentul şablonului State, cât şi al şablonului Strategy. Diferența dintre *Strategy* şi *State* stă în problemele rezolvate.

Exerciții.

1. Modelați cele două stări ale hipopotamului din jocul „Hungry Hippau” (Exercițiul 3, Capitolul 1) folosind şablonul State.

2. State este util în modelarea protocoalelor de orice tip. Simulați protocolul TCP/IP plecând de la maşina sa de stări (Figura 7.29).

3. Creați un mini-editor grafic cu trei butoane: radieră, creion, pensulă, ce implementează corespunzător metodele unei clase abstracte AbstractTool, ce modelează răspunsul aplicației la comenzi de mouse:

- **`moveTo`**`(point)`
  - input: locația *point* la care a fost deplasat mouse-ul

- **`mouseDown`**`(point)`
  - input: locația *point* la care se găseşte mouse-ul

- **`mouseUp`**`(point)`
  - input: locația *point* la care se găseşte mouse-ul

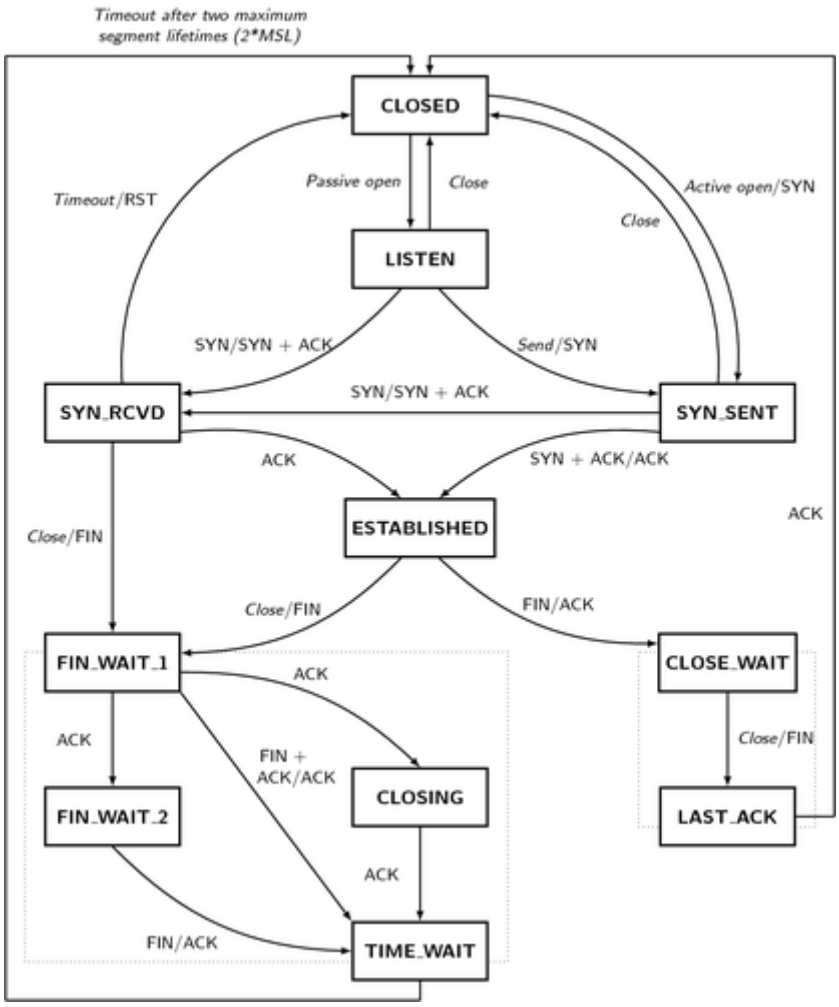


Figura 7.29. Protocolul TCP/IP

7.8. Şablonul VISITOR (Vizitator)

Scop şi aplicabilitate

Şablonul Visitor reprezintă o operație ce trebuie executată asupra elementelor din structura unui obiect, şi permite definirea unei noi operații fără schimbarea claselor elementelor asupra cărora acționează.

Folosiți Visitor atunci când:

- Structura unui obiect conține multe clase de obiecte cu interfețe diferite, şi dorim să efectuăm operații asupra acestor obiecte ce depind de clasele lor concrete;

- Mai multe operații distincte şi independente trebuie efectuate asupra obiectelor din structura unui obiect şi nu dorim "poluarea" claselor lor cu aceste operații. Visitor menține la un loc operațiile înrudite prin definirea lor în aceeaşi clasă;

- Clasele ce definesc structura obiectului se schimbă arareori, dar dorim destul de des definirea de noi operații asupra structurii. (Dacă clasele din structura obiectului se schimbă des, este mai puțin costisitoare definirea operațiilor în cadrul lor; altfel ar trebui redefinită interfața către toți Visitorii).

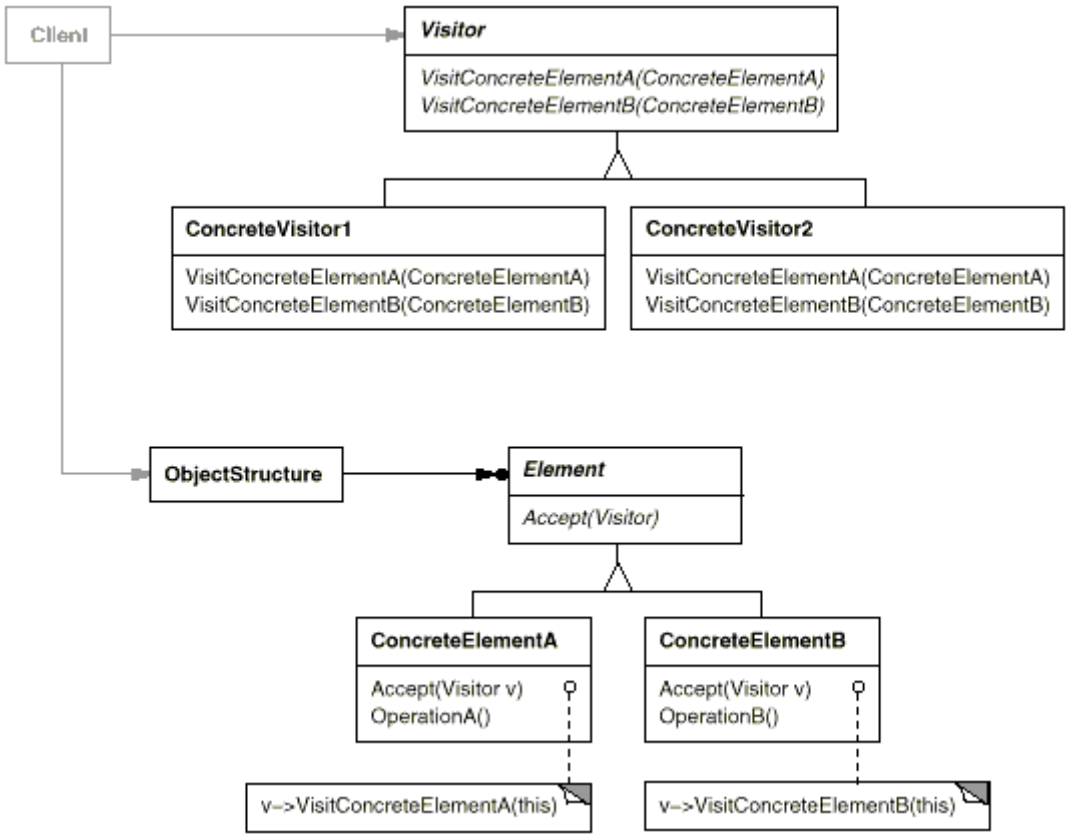


Figura 7.30. Structura şablonului Visitor

Participanți [Gamm94, Fig. 7.30]

- Visitor
  - Declară o operație Visit pentru fiecare clasă ConcreteElement din structura de obiecte. Numele şi signatura operației identifică clasa ce trimite cererea Visit către vizitator, ceea ce permite Visitor-ului să determine clasa concretă a elementului vizitat. Apoi Visitor-ul poate accesa elementul direct prin interfața sa specifică;
- ConcreteVisitor
  - Implementează fiecare operație declarată de Visitor;
- Element
  - Defineşte o operație Accept care primeşte un Visitor drept argument;
- ConcreteElement
  - Implementează o operație Accept, care primeşte un Visitor drept argument;

- ObjectStructure
  - Îşi poate enumera elementele;
  - Poate furniza o interfață de nivel înalt pentru a permite Visitor-ului să-şi viziteze elementele;
  - Poate fi un Composite sau o colecție (list, set).

Colaborări [Gamm94, Fig. 7.31]

- Un client ce foloseşte şablonul Visitor trebuie să creeze un obiect ConcreteVisitor şi apoi să traverseze structura de obiecte, vizitând fiecare element cu Visitor-ul;
- Când un element este vizitat, apelează operația Visitor care corespunde clasei sale. Elementul se transmite pe sine ca argument acestei operații pentru a-i permite Visitor-ului să-i acceseze starea, dacă este necesar.

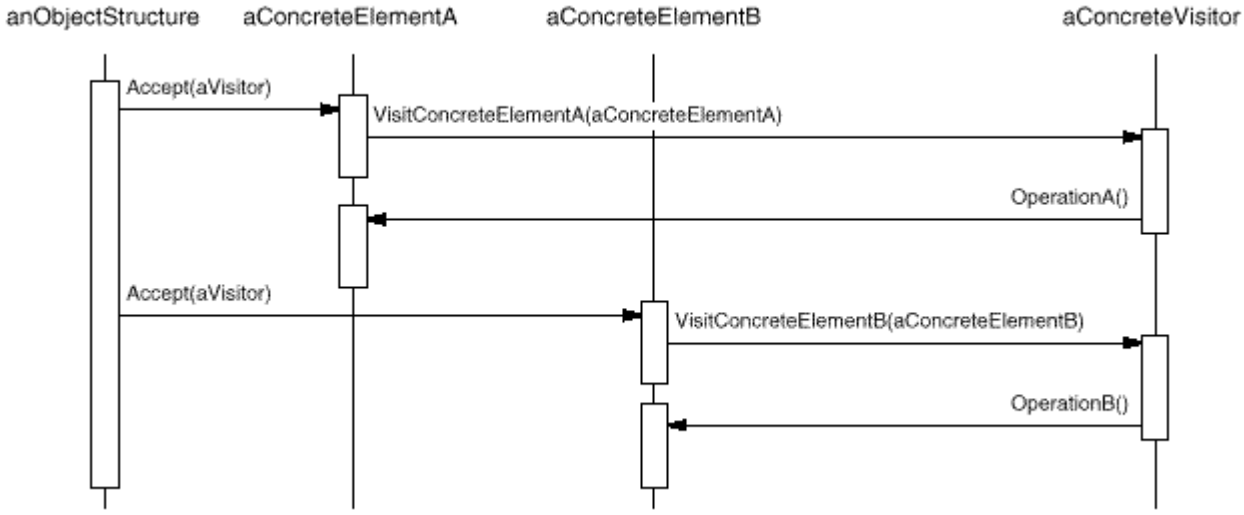


Figura 7.31. Colaborări în şablonul Visitor

Consecințe pozitive [Gamm94]

- Visitor facilitează adăugarea de operații noi (operații ce depind de componentele unor obiecte complexe). Se poate defini o operație nouă pe o structură de obiecte prin simpla adăugare a unui nou Visitor. Prin contrast, dacă distribuim funcționalitatea în mai multe clase, fiecare dintre aceste clase va trebui modificată la definirea unei operații noi.
- Un Visitor reuneşte operațiile înrudite şi le separă pe cele nerelaționate. Comportamentele înrudite sunt localizate în cadrul unui Visitor, nu sunt împrăştiate în clasele ce definesc structura obiectelor. Seturile nerelaționate de comportamente sunt separate în cadrul subclaselor propriului Visitor. Aceasta simplifică atât clasele ce definesc elementele cât şi algoritmii definiți de Visitori. Orice structuri de date specifice algoritmului pot fi ascunse în Visitor.

Exemplul 1.[Ecke02] Programul BeeAndFlowers.java.

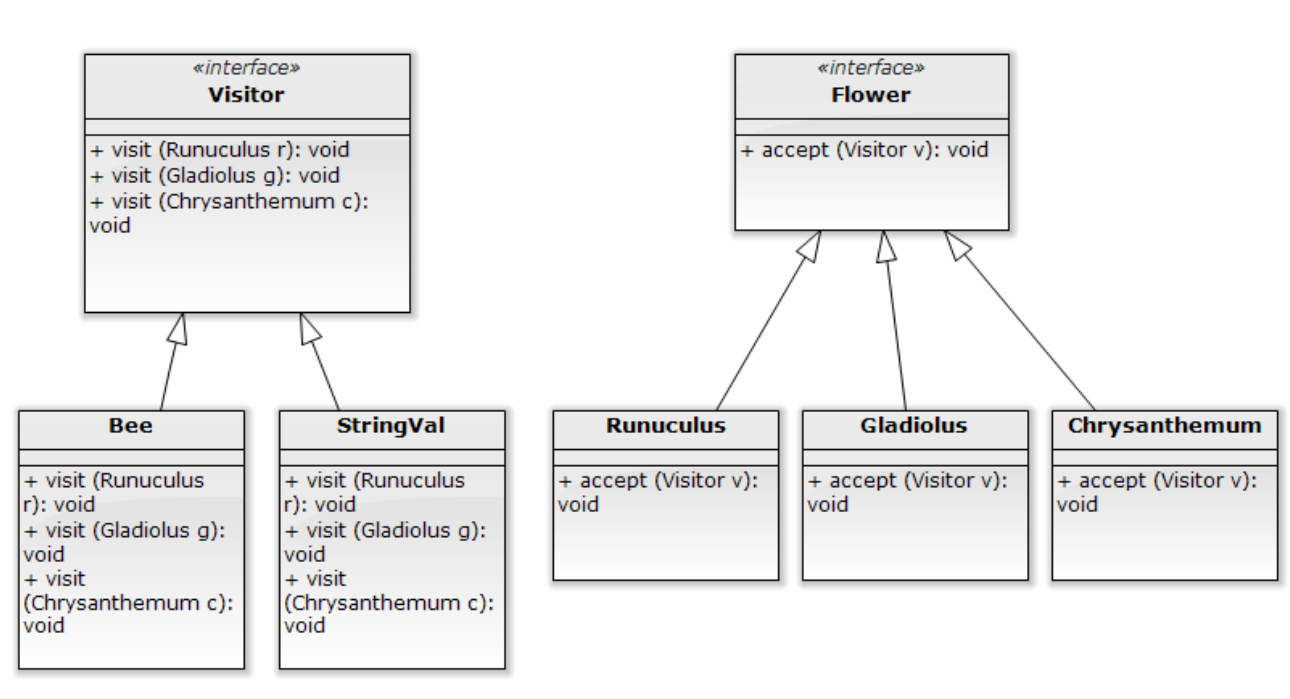


Figura 7.32. Exemplul 1

```
import java.util.*;
import junit.framework.*;

interface Visitor {
  void visit(Gladiolus g);
  void visit(Runuculus r);
  void visit(Chrysanthemum c);
}

// Ierarhia Flower – nu poate fi schimbata:
interface Flower {
  void accept(Visitor v);
}

class Gladiolus implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

class Runuculus implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}

class Chrysanthemum implements Flower {
  public void accept(Visitor v) { v.visit(this);}
}
```

```
// Adaugarea abilitatilor de a produce un string:
class StringVal implements Visitor {
  String s;
  public String toString() { return s; }
  public void visit(Gladiolus g) {
    s = "Gladiolus";
  }
  public void visit(Runuculus r) {
    s = "Runuculus";
  }
  public void visit(Chrysanthemum c) {
    s = "Chrysanthemum";
  }
}

// Adaugarea abilităţilor de a executa activităţi tip "Bee":
class Bee implements Visitor {
  public void visit(Gladiolus g) {
    System.out.println("Bee and Gladiolus");
  }
  public void visit(Runuculus r) {
    System.out.println("Bee and Runuculus");
  }
  public void visit(Chrysanthemum c) {
    System.out.println("Bee and Chrysanthemum");
  }
}

class FlowerGenerator {
  private static Random rand = new Random();
  public static Flower newFlower() {
    switch(rand.nextInt(3)) {
      default:
      case 0: return new Gladiolus();
      case 1: return new Runuculus();
      case 2: return new Chrysanthemum();
    }
  }
}

public class BeeAndFlowers {
  List flowers = new ArrayList();
  public BeeAndFlowers() {
    for(int i = 0; i < 10; i++)
      flowers.add(FlowerGenerator.newFlower());
  }
  public void test() {
```

```
    // reprezentarea florii ca string:
     StringVal sval = new StringVal();
     Iterator it = flowers.iterator();
     while(it.hasNext()) {
       ((Flower)it.next()).accept(sval);
       System.out.println(sval);
     }

   // Aplicarea operatiei "Bee" asupra tuturor florilor:
     Bee bee = new Bee();
     it = flowers.iterator();
     while(it.hasNext())
       ((Flower)it.next()).accept(bee);
   }
   public static void main(String args[]) {
    //...
   }
}
```

Exerciții.

1. Folosiți Visitor pentru a traversa structurile Composite de la Exercițiile 1 şi 2, Secțiunea 7.4.

CAP. 8. ŞABLOANE STRUCTURALE

8.1. Şablonul PROXY

***"proxy** = Agent pentru o persoană care se comportă ca substitut pentru aceasta, autoritatea de a acționa în locul altcuiva."*

Scop şi aplicabilitate

Şablonul Proxy furnizează un surogat (substitut) pentru alt obiect, pentru a controla accesul la acesta. Şablonul este util oricând este nevoie de o referință către un obiect mai *flexibilă* sau mai *sofisticată* decât un simplu pointer.

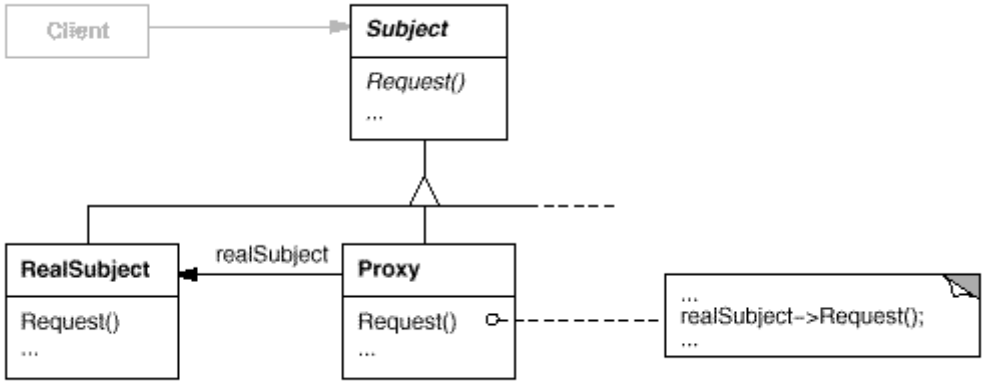


Figura 8.1. Structura şablonului Proxy

Participanți [Gamm94, Fig. 8.1]

- Proxy
  - Menține o referință ce permite Proxy-ului să acceseze subiectul real;
  - Furnizează o interfață identică cu a lui Subject
    - Astfel că Proxy poate fi substituit subiectului real;
  - Controlează accesul la subiectul real
    - Poate fi responsabil de crearea şi ştergerea acestuia;
- Subject
  - Defineşte interfața comună pentru RealSubject şi Proxy;
- RealSubject
  - Defineşte obiectul real pe care îl înlocuieşte Proxy.

Colaborări [Gamm94, Fig.8.2]

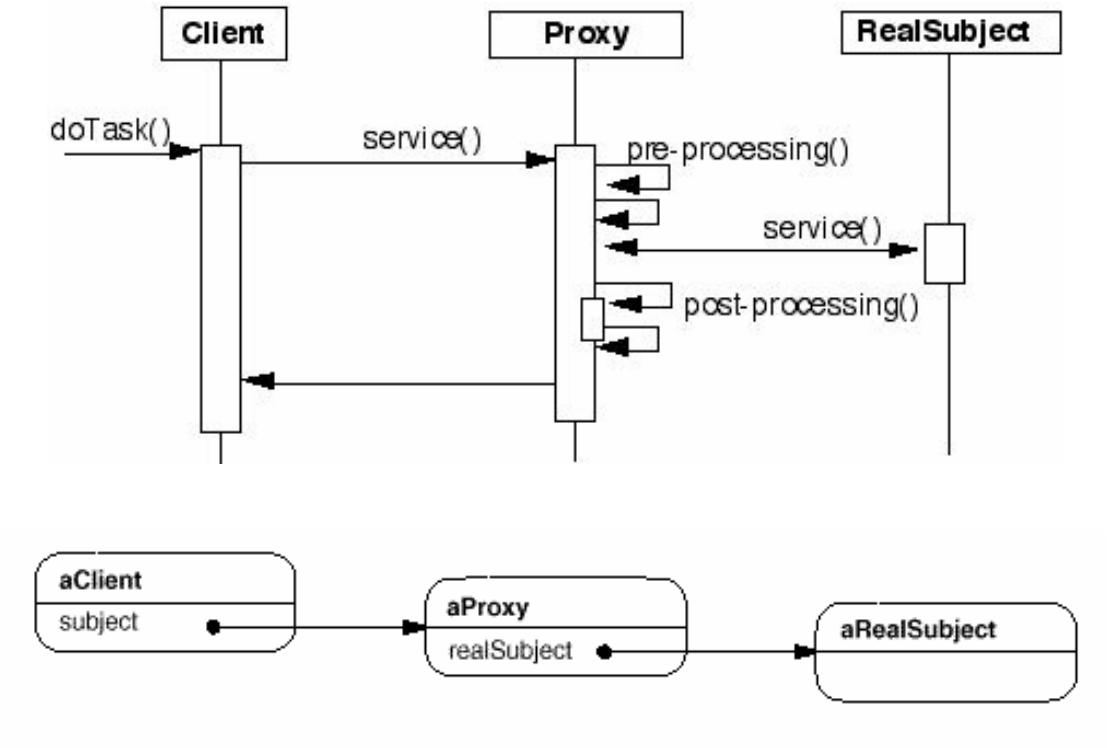


Figura 8.2. Colaborări în şablonul Proxy

Tipuri de Proxy

1. Proxy la distanță

- Ascunde detaliile reale ale accesării unui obiect
  - Obiectul real este pe o maşină la distanță (spațiu de adresă la distanță);
  - Folosit în RMI şi CORBA;

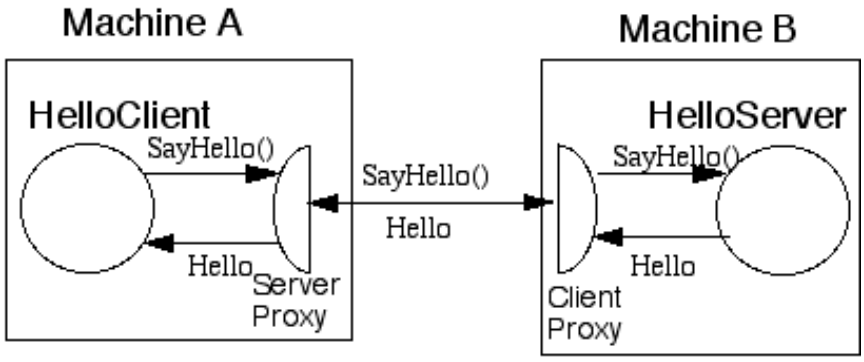


Figura 8.3. Proxy la distanță

2. Proxy de Sincronizare

- Sincronizează accesările multiple ale subiectului real;

3. Proxy Virtual

- Creează/accesează obiecte costisitoare la cerere
  - Poate se doreşte amânarea creării unui obiect costisitor până ce este accesat efectiv;
- Poate fi prea costisitor să păstrăm întreaga stare a obiectului în memorie odată;

4. Proxy de Protecție

- Furnizează diferite nivele de acces la obiectul original;

5. Proxy de Cache (Proxy Server)

- Clienți locali multipli pot partaja rezultatele operațiilor costisitoare
  - Accesări la distanță sau calcule lungi;

6. Proxy -accesorii (pointeri inteligenți) –

- Previn ştergerea accidentală a obiectelor (numără referințele)- vezi Exemplul1.

Consecințe pozitive

- Proxy-urile introduc un nivel de indirectare (ocolire)
  - folosit diferit în funcție de tipul de Proxy:
    - Ascunderea spațiului adresă (Proxy la distanță);
    - Crearea la cerere (Proxy virtual);
    - Permite activități auxiliare adiționale (protecție, pointeri inteligenți);
- Copiază-în-timp-ce-scrii [vezi Exemplul 1]
  - copierea obiectelor mari şi complicate este costisitoare;
  - folosiți Proxy pentru copiere numai când obiectul nou este modificat;
  - Subiectul trebuie numărat pe baza referinței
    - Proxy incrementează contorul în timpul copierii;
    - Când apar modificări, copiază şi decrementează contorul;
    - if (counter = = 0) delete Subject.

Exemplul 1. Clasa de manevră

- String conține un obiect StringRep
  - StringRep reține textul şi numărul de referințe;
- String transferă operațiile cu şiruri către obiectul StringRep;
- String tratează operațiile cu pointeri şi ştergerea obiectului StringRep când numărul de referințe ajunge la zero (Fig. 8.4).

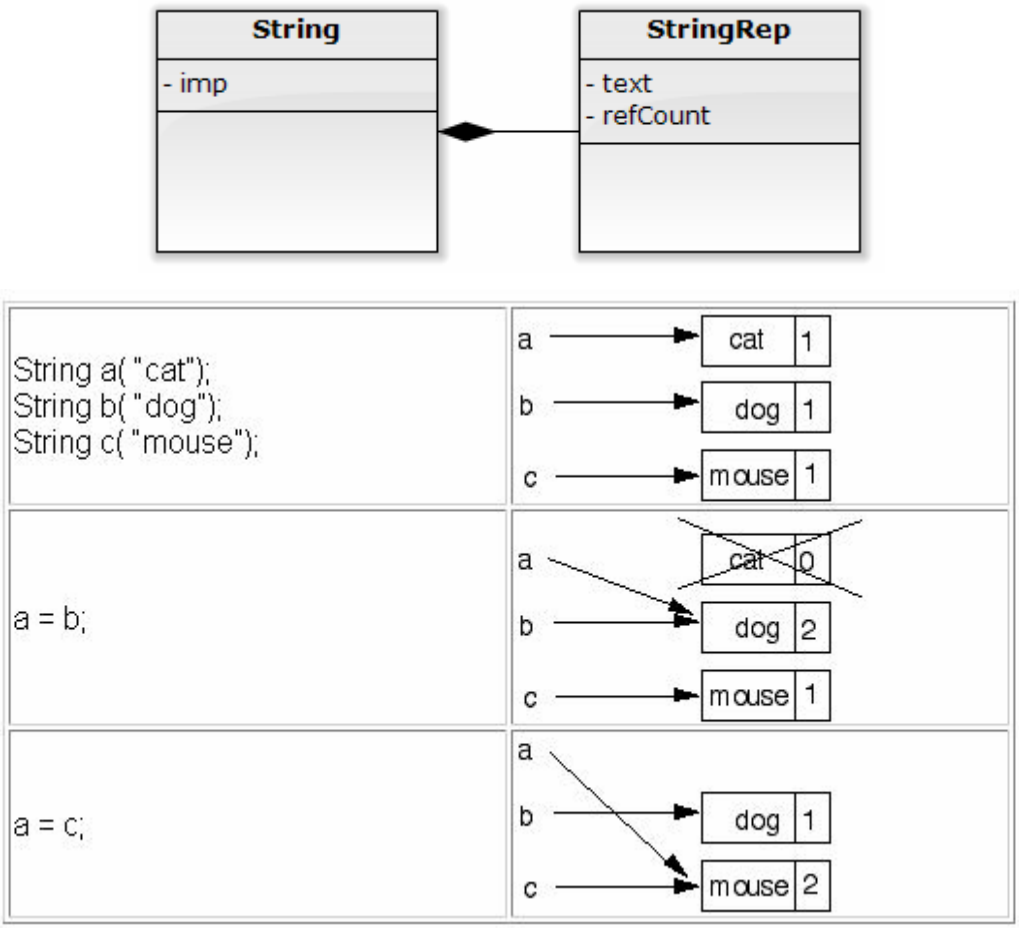


Figura 8.4. Un şir şi reprezentarea sa

Exemplul 2. O demonstrație simplă a şablonului Proxy [Ecke02].

```
interface ProxyBase {
  void f();
  void g();
  void h();
}

class Proxy implements ProxyBase {
  private ProxyBase implementation;
  public Proxy() {
    implementation = new Implementation();
  }

  // se transmite apelul metodelor catre implementare:
  public void f() { implementation.f(); }
  public void g() { implementation.g(); }
  public void h() { implementation.h(); }
}
```

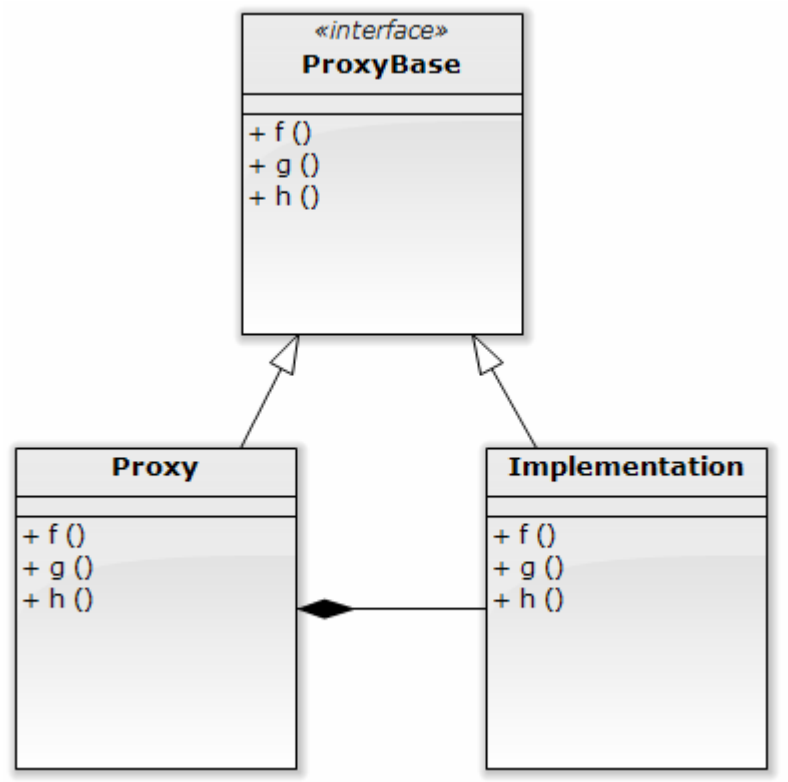


Figura 8.5

```
class Implementation implements ProxyBase {
  public void f() { System.out.println("Implementation.f()"); }
  public void g() { System.out.println("Implementation.g()"); }
  public void h() { System.out.println("Implementation.h()"); }
}

public class ProxyDemo   {
  Proxy p = new Proxy();
  public void test() {
                        p.f();
                        p.g();
                        p.h();
  }
//...
public static void main(String args[]) {test(); }
}
```

Exemplul 3. Încărcarea obiectelor de dimensiuni mari (Proxy virtual) [Coop98].

Există editoare de documente ce pot încorpora obiecte multimedia. Aceste obiecte sunt costisitor de creat, şi vor determina deschiderea lentă a documentului. Putem însă evita crearea de obiecte costisitoare, pentru că acestea nu sunt necesare în totalitate, aşa cum nu sunt nici vizibile în totalitate la un anumit moment. Fiecare obiect costisitor poate fi creat *la cerere* (spre exemplu, în momentul când imaginea trebuie afişată). În locul

obiectului real ar trebui pus un substitut care să nu complice editorul de documente, dar ceva mai elegant decât un mesaj care să explice situația. Ideea este să folosim un substitut (Proxy), creat doar când este necesar pentru desenare (acesta va reține informații despre dimensiune) (Figura 8.6).

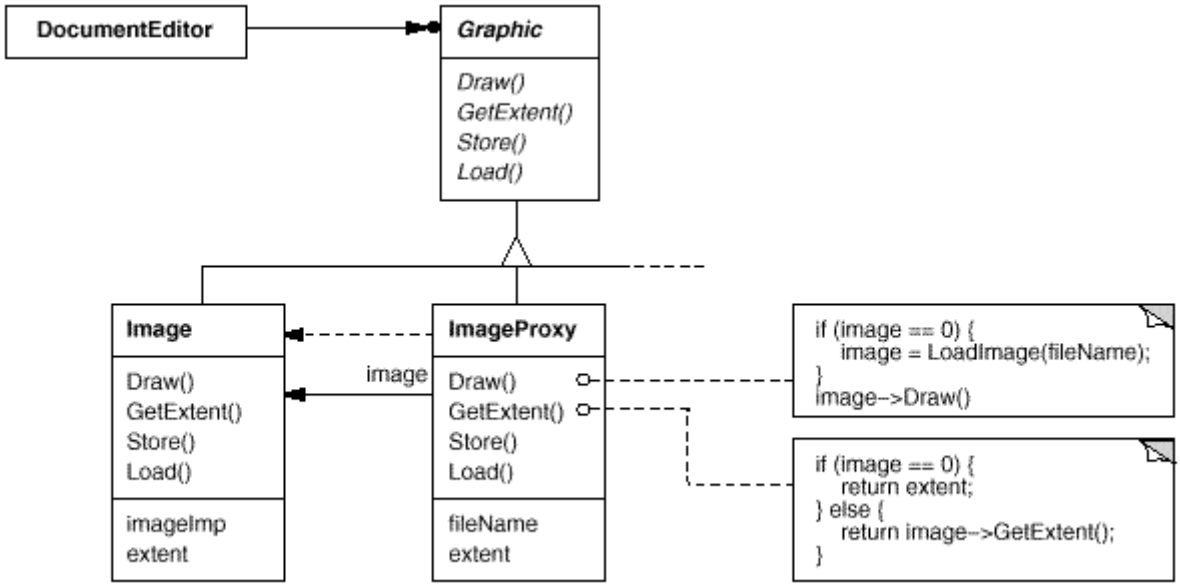


Figura 8.6. Substitut pentru crearea obiectelor de dimensiuni mari

Exerciții.

1.Proxy la distanță (RMI)

În java RMI, un obiect aflat pe o mașină (executat de o JVM) şi denumit *client* poate invoca metode ale altui obiect de pe o altă maşină (altă JVM)- obiect "remote" (la distanță). Proxy-ul (denumit şi stub) se găseşte pe maşina client şi este invocat de către client ca şi cum ar invoca obiectul însuşi (proxy-ul implementează aceeaşi interfață ca şi RealSubject). Proxy-ul se va ocupa de comunicarea cu obiectul la distanță, va invoca metodele acelui obiect, şi va returna un eventual rezultat clientului. Scrieți un program ce foloseşte RMI: programul client (RmiClient.class) trimite un mesaj programului server (RmiServer.class), care îl afişează la consolă.

(http://docs.oracle.com/javase/8/docs/technotes/guides/rmi/hello/hello-world.html)

2.Proxy de protecție

Într-un director dat se pot efectua operații de *cut*, *copy* şi *paste*, şi dorim să specificăm că doar utilizatorii autorizați pot avea acces şi pot efectua aceste operații (Figura 8.7). Implementați folosind Proxy.

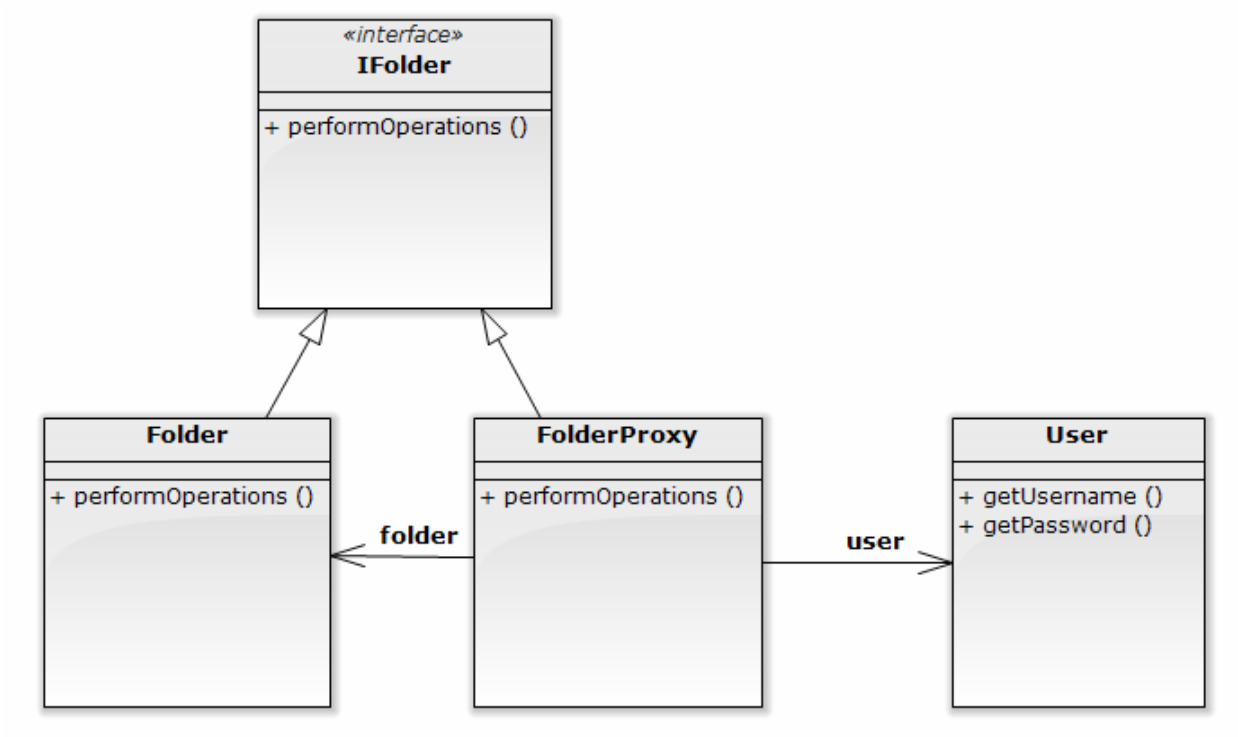


Figura 8.7. Exercițiul 2: Proxy de protecție

3. Folosiți Proxy pentru a implementa tehnica de “double buffering”, folosită pentru a reduce licăririle (flickering) din jocurile cu animație (în jocul Hungry Hippau- Cap.1 , Ex. 3).

8.2. Şablonul ADAPTER (Adaptor)

Scop şi aplicabilitate

Şablonul Adapter converteşte interfața unei clase într-o altă interfață aşteptată de client, permițând unor clase cu interfețe diferite să lucreze împreună (prin “traducerea” interfețelor). Şablonul este util atunci când dorim să folosim o clasă pre-existentă, şi interfața acesteia nu se potriveşte cu interfața de care avem nevoie, sau când vrem să creem o clasă reutilizabilă ce cooperează cu clase nerelaționate şi neprevăzute (i.e. clase ce pot avea interfețe incompatibile). *Adaptorul obiect* este util şi atunci când avem nevoie să folosim mai multe subclase existente, dar nu este practic să derivăm din fiecare pentru a le adapta interfața. Un obiect adaptor poate adapta interfața clasei sale părinte.

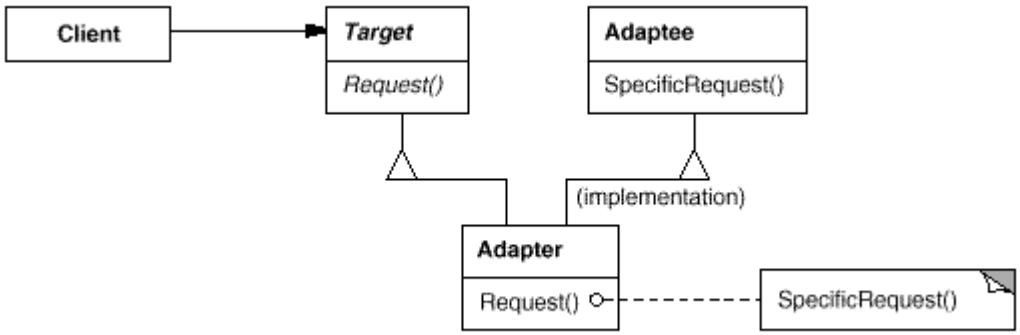


Figura 8.8. Structură – Adapter clasă

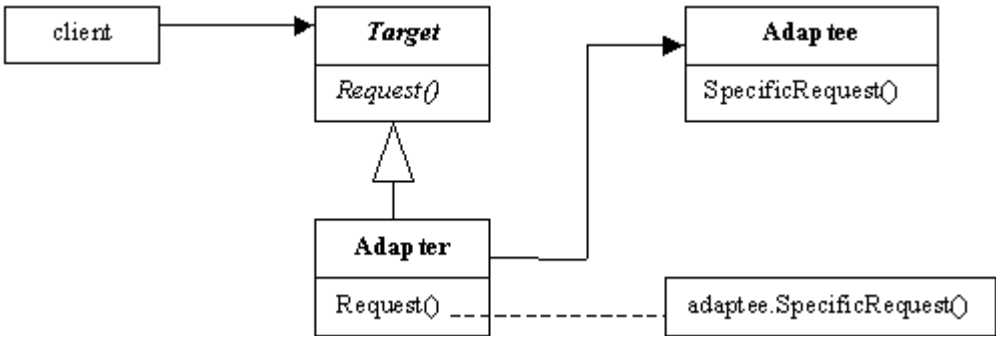


Figura 8.9. Structură – Adapter obiect

Participanți [Gamm94, Fig.8.8-8.9]

- Target
  - Definește interfața specifică domeniului pe care o folosește Clientul;
- Client
  - Colaborează cu obiecte în conformitate cu interfața Target;
- Adaptee
  - Definește o interfață existentă ce necesită adaptare;
- Adapter
  - Adaptează interfața lui Adaptee la interfața Target.

Colaborări [Gamm94]

- Clienții apelează operații ale unei instanțe Adapter, iar adaptorul apelează operațiile Adaptee (adaptatului), ce pot trata cererile.

Consecințe adaptor clasă [Gamm94]

- Adaptează Adaptee la Target ținând cont de o clasă Adapter concretă;
- Permite clasei Adapter să suprascrie o parte din comportamentul Adaptee, întrucât Adapter este o subclasă a lui Adaptee;
- Introduce doar un singur obiect, şi nu este nevoie de indirectare suplimentară cu pointeri pentru a obține obiectul adaptat.

Consecințe adaptor obiect [Gamm94]

- Permite unui singur Adapter să lucreze cu mai multe clase Adaptee- i.e. clasa Adaptee şi toate subclesele sale, dacă acestea există; Adapter poate adăuga funcționalități la toate clasele Adaptee odată;
- Este mai dificil de suprascris comportamentul lui Adaptee; pentru aceasta va fi necesar să derivăm Adaptee şi să facem Adapter să se refere la subclasă direct, nu la Adaptee;
- Cantitatea de muncă necesară în Adapter depinde de cât de similare sunt Adaptee şi Target (poate merge de la redenumirea unor operații până la scrierea unui set complet diferit de operații);
- Adaptori conectabili (plugg-able): clase mai uşor de refolosit întrucât au fost construite cu interfețe adaptabile din start (vezi la Implementare).

Implementare [Gamm94]

- În C++, varianta adaptor clasă, Adapter moşteneşte public Target şi privat Adaptee (deci Adapter este un subtip al lui Target, dar nu şi al lui Adaptee);
- Adaptori conectabili
  - Găsirea unei interfețe minimale (cel mai mic subset de operații ce permite realizarea adaptării);
  - Folosirea operațiilor abstracte (pentru interfața minimală)- subclasele trebuie să implementeze operațiile abstracte.

Şabloane relaționate [Gamm94]

- Bridge
  - Au structuri similare, dar Bridge are un scop diferit: să separe o implementare de interfața sa, astfel încât cele două să poată fi variate mai uşor, independent una de alta. În schimb, adaptorul are scopul de a modifica interfața unui obiect *existent;*
- Decorator
  - Îmbogățeşte un obiect fără să-i schimbe interfața- deci este mai transparent pentru aplicație, decât un adaptor; în consecință, decoratorul suportă compoziție recursivă, ceea ce nu este posibil cu adaptorii puri;
- Proxy
  - Defineşte un reprezentant (surogat) pentru un obiect, fără să-i schimbe interfața.

Exemplul 1[Gamm94]

Un editor de desenare lucrează cu obiecte grafice ce implementează clasa abstractă Shape (Figura 8.10). Dacă LineShape sau PolygonShape sunt uşor de implementat (au capacităţi limitate de desenare şi editare), o clasă TextShape ce poate afişa şi edita text este mai dificilă (actualizare ecran, buffering…). Dacă dorim să utilizăm o clasă predefinită TextView care tratează textul corect, definim clasa TextShape care adaptează interfaţa TextView la cea a lui Shape (TextShape- clasa Adapter), astfel ca editorul de text poate refolosi clasa incompatibilă TextView.

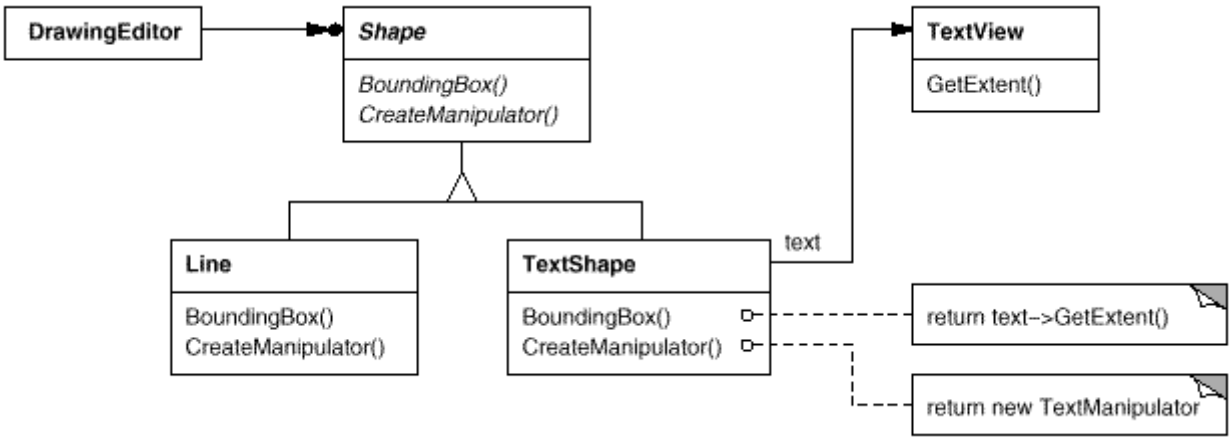


Figura 8.10. Exemplul 1: editor grafic

Roluri în cadrul şablonului: Shape este Target, DrawingEditor este Client, TextView este Adaptee (adaptat), iar TextShape este Adapter (adaptor).

Exemplul 2. [Ecke02] Adaptor obiect simplu

```
import junit.framework.*;

class Target {public void request() {}}

class Adaptee {
  public void specificRequest() {
    System.out.println("Adaptee: SpecificRequest");}
}

class Adapter extends Target {
  private Adaptee adaptee;
  public Adapter(Adaptee a) {
    adaptee = a;}

  public void request() {
    adaptee.specificRequest();}}
```

```
public class SimpleAdapter {
  Adaptee a = new Adaptee();
  Target t = new Adapter(a);
  public void test() {
    t.request();
  }
  public static void main(String args[]) {
    //...
  }
}
```

Exerciții.

1. Presupuneți că aveți clasa ProdusVechi (Adaptee), cu metodele g() şi h() de tip void, şi aveți nevoie de clasa ProdusNou (Target) cu metoda f() de tip void (în interiorul căreia se vor apela g() şi h()). Construiți un adaptor clasă şi clasa client care foloseşte Produsul nou şi demonstrați printr-o diagramă de clase că ați aplicat Adapter-clasă. Modificați astfel încât să aveți un Adaptor obiect. Ce variantă este mai avantajoasă şi de ce?

2. Un program de gestionare a informațiilor despre pacienții unui spital foloseşte clasa „Fişa” cu următoarele date membru: Nume, Adresă, CNP (String-uri). Clasa are o metodă publică *simptome*, ce primeşte ca intrare CNP-ul şi întoarce o listă de simptome, citite dintr-un fişier „*CNP*.doc”. Dorim să folosim această metodă în cadrul unei clase noi „Fişa_nouă”, cu diferența că trebuie să primească un parametru întreg *Cod*. Codul se poate găsi pe bază de CNP, într-o tabelă specială. Folosiți Adapter.

8.3. Şablonul BRIDGE (Punte)

Scop şi aplicabilitate

Şablonul Bridge decuplează o abstracțiune de implementarea sa, permițând implementării să varieze independent de abstracțiune. Abstracțiunea defineşte şi implementează o interfață; toate operațiile din abstracțiune apelează metode din obiectul de implementare.

În şablonul Bridge:

- o abstracțiune poate folosi implementări diferite;
- o implementare poate fi folosită în diferite abstracțiuni.

Şablonul este util atunci când:

- Trebuie evitată legarea permanentă între o abstracțiune şi implementarea sa;
- Abstracțiunile şi implementările lor trebuie să fie *independent extensibile* prin derivare;
- Ascunde implementarea unei abstracțiuni complet de clienți
  - Codul clienților nu mai trebuie recompilat atunci când se modifică implementarea;
- Partajează o implementare între mai multe obiecte
  - Şi acest lucru trebuie ascuns de client.

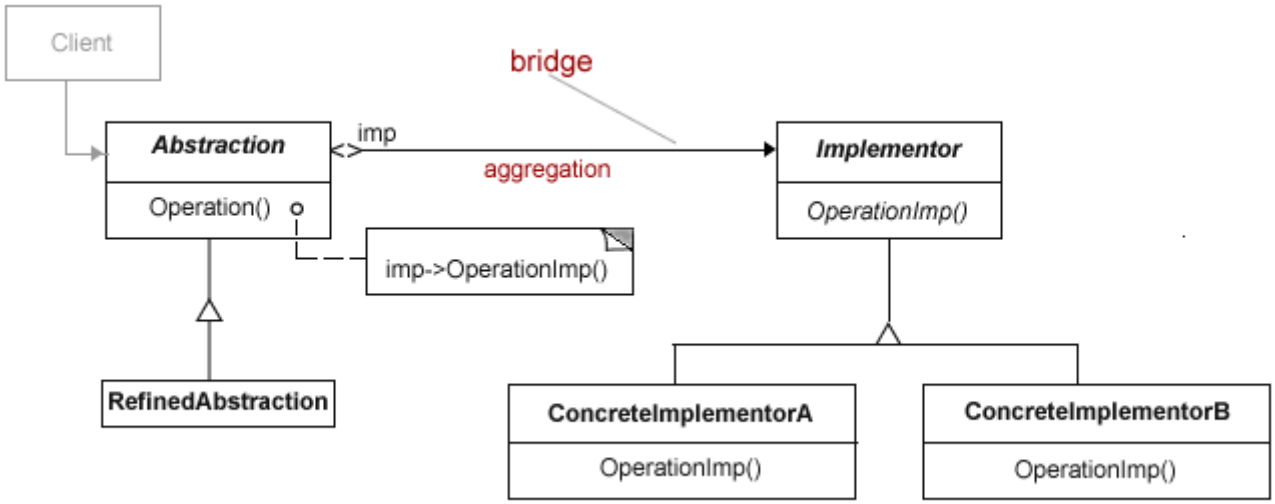


Figura 8.11. Structura şablonului Bridge

Participanți [Gamm94, Fig. 8.11]

- Abstraction
  - Defineşte interfața abstracției;
  - Menține o referință la un obiect de tip Implementor;
- Implementor
  - defineşte interfața pentru clasele de implementare
    - Nu corespunde neapărat interfeței Abstraction;
    - Implementor conține operații primitive;
    - Abstraction defineşte operațiile de nivel înalt bazate pe aceste primitive;
- RefinedAbstraction
  - Extinde interfața definită de Abstraction;
- ConcreteImplementor
  - implementează interfața Implementor interface, definind o implementare concretă.

Consecințe [Gamm94]

- Decuplează interfața şi implementarea

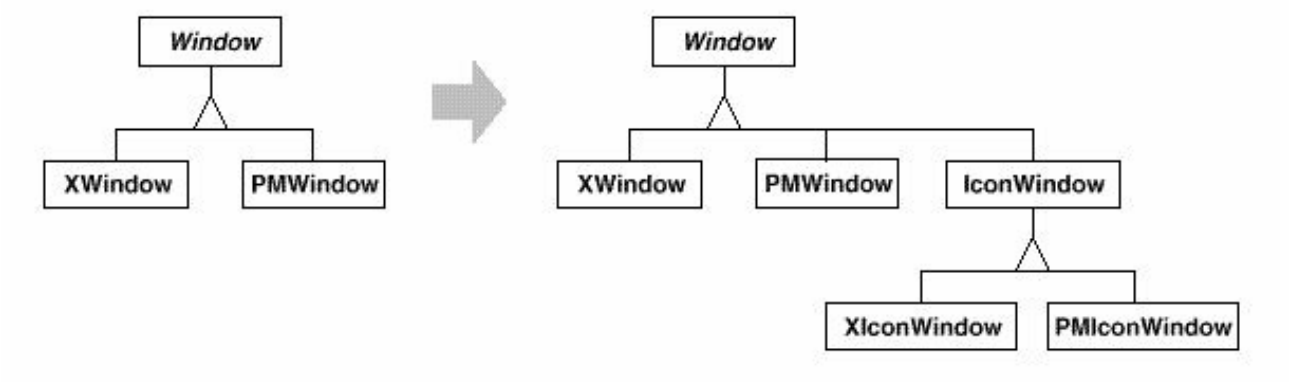
- implementarea este configurabilă şi modificabilă la execuție;
- reduce dependențele de la momentul compilării
  - implementarea modificărilor nu necesită recompilarea Abstracțiunii;

- Extensibilitate îmbunătățită
  - extinde prin derivarea independentă a Abstracțiunilor şi Implementărilor;
- Ascunderea detaliilor de implementare de clienți
  - ascunde clienților detaliile de implementare
    - ex. partajarea obiectelor implementor împreună cu numărarea referințelor;
  - "clasa de manevră" (Handle Class) respectă principiile unui Bridge şi ale unui Proxy în acelaşi timp.

Implementare [Gamm94]

- Un singur Implementor
  - Nu este necesar să creăm o clasă implementor abstractă;
  - primitiv, dar util datorită decuplării;
- Ce Implementor trebuie să folosim?
  - Varianta 1: lasă Abstracțiunea să cunoasca toate clasele de implementare concrete şi să aleagă între ele;
  - Varianta 2: alege inițial implementarea implicită şi modifică mai târziu;
  - Varianta 3: folosiți Abstract Factory
    - Nu există cuplaj între Abstracțiune şi clasele de implementare concrete;
- Partajarea Implementorilor
  - Folosiți "Clasa de manevră" .

Exemplul 1. În Windows, ferestre cu aspecte diverse trebuie să conțină componente cu aspect adaptat. Abordarea problemei prin moştenire conduce la complexitate ridicată şi la multe subclase (Figura 8.12). Figura 8.13 prezintă soluția Bridge.



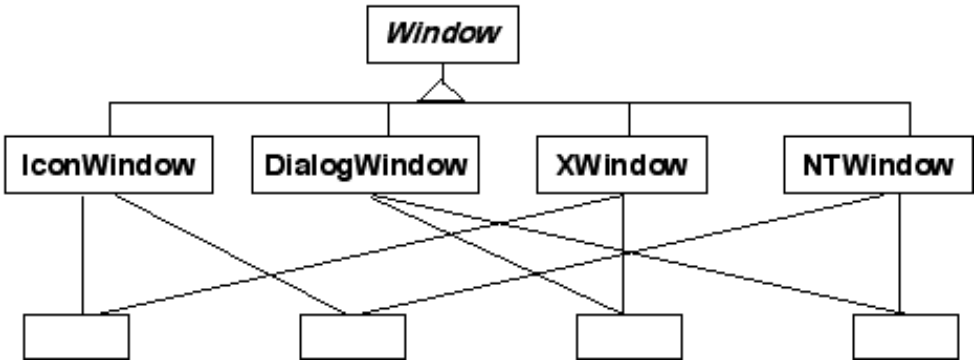


Figura 8.12. Abordarea complexă

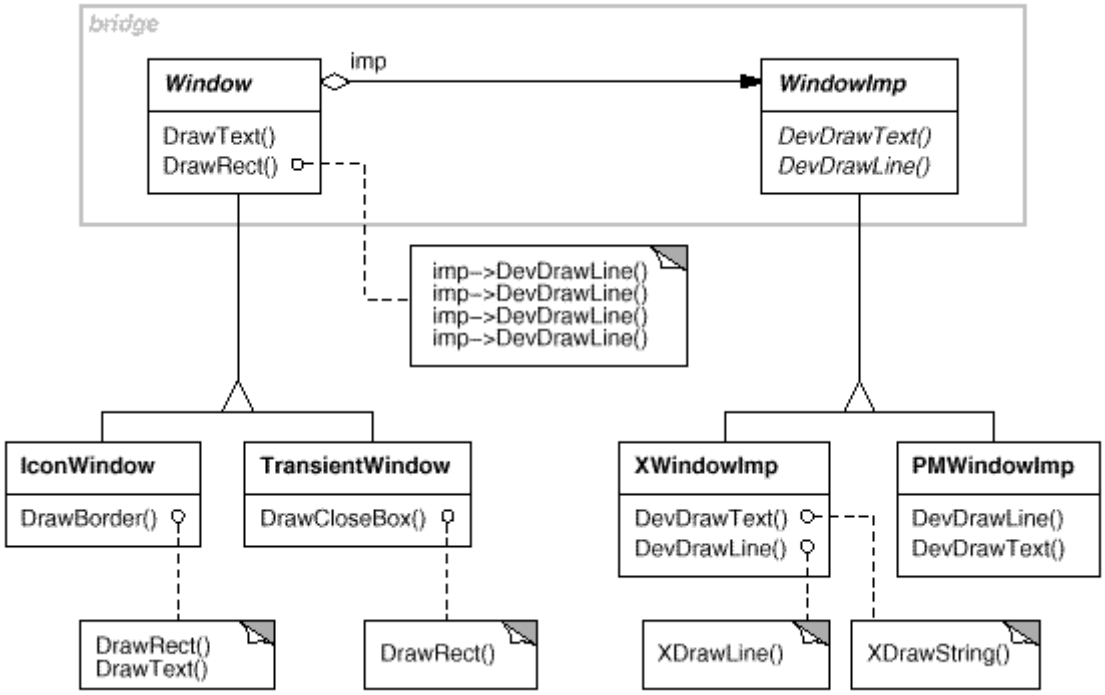


Figura 8.13. Soluția Bridge

Exemplul 2 [Coop98]. Un program afişează o listă de produse într-o fereastră (un JList simplu). După vinderea unui număr semnificativ de produse, dorim afişarea lor într-un tabel împreună cu numărul de exemplare vândute (Figura 8.14). Într-un program simplu, ar fi de ajuns să creem un adaptor-clasă care să adapteze interfața elaborată a JList la necesitățile noastre. Acesta însă nu ar funcționa la cerințe mai complexe, după cum vom vedea în continuare.

Spre exemplu, să presupunem că dorim să producem două tipuri de afişări pentru produse: o afişare pentru clienți (customer view) care doar listează produsele şi o afişare pentru director (executive view) care afişează şi numărul de unități vândute. Lista simplă de produse se va afişa într-un JList, iar lista pentru director într-un tabel JTable (ambele în aceeaşi fereastră, ca în Figura 8.14).

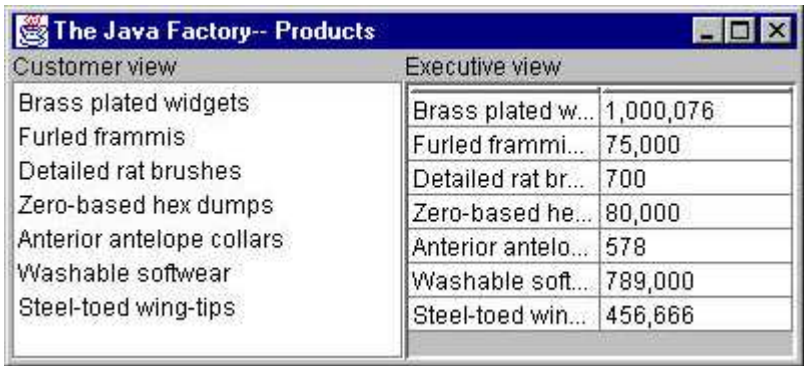


Figura 8.14. Tabel Exemplul 2

Vom crea instanțe ale unui tabel şi ale unei liste din clase derivate din JList şi JTable şi care pot să separe numele de cantitățile corespunzătoare:

```
pleft.setLayout(new BorderLayout());
pright.setLayout(new BorderLayout());
//adauga lista simpla
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center", new productList(prod));
//adauga lista cu numere ca tabel
pright.add("North", new JLabel("Executive view"));
pright.add("Center", new productTable(prod));
```

Derivăm clasa *productList* direct din clasa JawtList, astfel că Vectorul ce conține lista de produse este singura intrare către clasă (JawtList preia un vector de elemente şi le afişează într-un JList).

```
public class productList extends JawtList
{
public productList(Vector products)
{
super(products.size()); //pentru compatibilitate
for (int i = 0; i < products.size(); i++)
{
//seia fiecare şir şi se păstrează doar numele prodselor, fără cantităţi
String s = (String)products.elementAt(i);

//separare cantitate de nume
int index = s.indexOf("--");
```

```
if(index > 0)
add(s.substring(0, index));
else
add(s);}}}
…
```

Construcția unui Bridge

Dacă dorim să afişăm produsele în ordine alfabetică, şi păstrăm abordarea de până acum, va trebui să modificăm sau să derivăm ambele clase pentru liste- deci dacă în timp vom avea nevoie de mai multe tipuri de afişări, mentenanța devine dificilă. Astfel că vom construi o singură clasă (bridge) care se va ocupa de modificările necesare în eventualitatea unor extinderi.

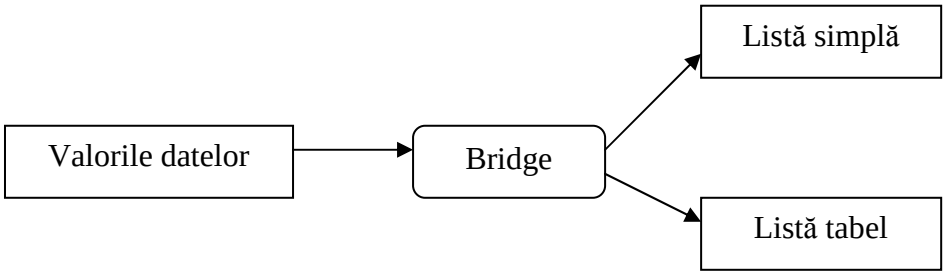


Figura 8.15. Clasa Bridge

Pentru a determina clasa Bridge să returneze o componenta vizuală potrivită o vom deriva dintr-o clasă ce reprezintă un panou cu bare de derulare:

```
public class listBridge extends JscrollPane
```

Când construim o clasă Bridge, trebuie să alegem cum va determina aceasta ce clasă să instanțieze între mai multe posibile: pe baza unor constante simple, sau pe baza unor valori/ cantități de date de afişat. Aici definim două constante în clasa *listBridge*:

```
static public final int TABLE = 1, LIST = 2;
```

Constructorul din programul principal rămâne în mare acelaşi, doar înlocuim cu apeluri la clasa *listBridge*:

```
pleft.add("North", new JLabel("Customer view"));
pleft.add("Center",new listBridge(prod, listBridge.LIST));
//adauga “execute view” ca tabel
pright.add("North", new JLabel("Executive view"));
pright.add("Center",
new listBridge(prod, listBridge.TABLE));
```

Constructorul pentru clasa *listBridge* devine:

```
public listBridge(Vector v, int table_type)
{
Vector sort = sortVector(v); //sortare vector
if (table_type == LIST)
getViewport().add(makeList(sort)); //creează tabel
if (table_type == TABLE)
getViewport().add(makeTable(sort)); //creează lista
}
```

Avantajul pe care îl aduce clasa Bridge este că putem utiliza clasele JTable şi JList direct, fără modificări, şi deci putem plasa orice calcule ce țin de interfața adaptatoare în modelele de date ce construiesc datele pentru listă şi respectiv pentru tabel:

```
private JList makeList(Vector v) {
return new JList(new BridgeListData(v));
}

private JTable makeTable(Vector v) {
return new JTable(new prodModel(v));}
```

Rezultatul sortat este prezentat în Figura 8.16.

Figura 8.16. Tabelul sortat

La prima vedere şablonul seamănă cu Adapter, prin faptul că o clasă este folosită să convertească un tip de interfață la alta. Totuşi, scopul şablonului Adapter este să facă interfața unei (unor) clase identică cu a unei clase date, în timp ce scopul şablonului Bridge este să separe interfața unei clase de implementare, pentru aputea varia /înlocui implementarea fără a modifica şi codul client.

Exerciții.

1. Folosiți şablonul Bridge în jocul Hungry Hippau (Cap. 1, Exercițiul 3), pentru a crea o abstracțiune unică (*Obiect mobil*) pe care codul client o foloseşte pentru a trata uniform buştenii şi fructele. Clasa *Obiect_mobil* este în relație de compunere cu interfața „Implementare_comportament”, implementată de „Comportament_obstacol” şi „Comportament_bonus”.

2. Completați clasele JawtList şi JscrollPane din Exemplul 2, şi extindeți interfața astfel încât aceasta să mai afişeze şi graficul produselor ordonate crescător după numărul de exemplare vândute (într-o histogramă, spre exemplu).

8.4. Şablonul COMPOSITE (Compozit)

Scop şi aplicabilitate

Şablonul Composite tratează uniform obiectele individuale şi compunerile acestor obiecte. Poate compune obiectele în structuri arborescente pentru a reprezenta agregări recursive (Figura). Şablonul este util atunci când trebuie să reprezentăm ierarhii de obiecte de tip parte-întreg, oferind abilitatea de a ignora diferența dintre compuneri de obiecte şi obiecte individuale (clientul nu trebuie să ştie dacă primeşte un argument atomic sau format din mai multe părți).

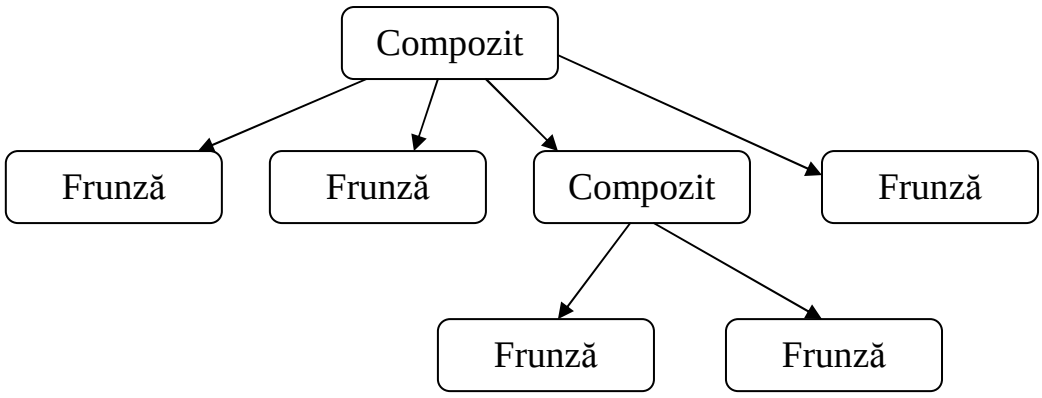


Figura 8.17. Structura ierarhică arborescentă

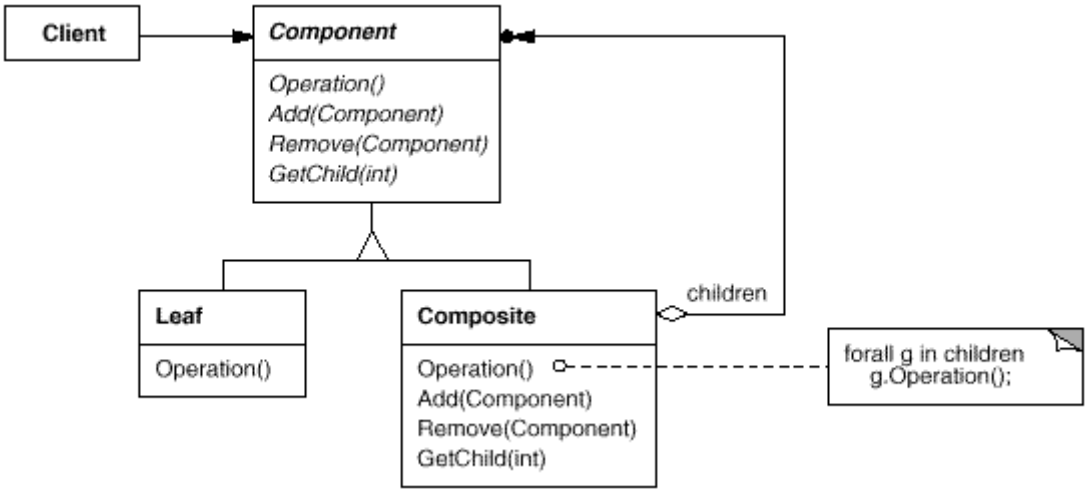


Figura 8.18. Structura şablonului Composite

Participanți & Colaborări [Gamm94, Fig. 8.18]

- Component
  - declară interfața pentru obiectele din compunere;
  - implementează comportamentul implicit pentru componente, când aceasta este posibil;
- Composite
  - defineşte comportamentul pentru componentele cu copii;
  - stochează componentele-copil
    - implementează operații specifice copiilor;
- Leaf
  - defineşte comportamentul pentru obiectele primitive din compunere;
- Client
  - manipulează obiectele din compunere prin interfața Componentei;

Consecințe [Gamm94]

Avantaje

- Defineşte ierarhii uniforme de clase
  - compunere recursivă de obiecte;
- Face clienții simpli
  - nu trebuie să ştie dacă este o frunză sau un compus;
  - simplifică codul deoarece evită tratarea diferită a fiecărei clase;
- Mai uşor de extins
  - uşor de adăugat noi clase Composite sau Leaf;
  - aplicație a Principiului Deschis-Închis.

Dezavantaje

- Design prea general
  - sunt necesare verificări de tip pentru a restricționa tipurile admise într-o anumită structură compusă.

Detalii de Implementare [Gamm94]

- Referințe explicite la părinți
  - acestea simplifică traversarea, şi sunt plasate în Component;
  - problema consistenței: referința la părinte trebuie schimbată numai când se adaugă sau se şterge copilul;
- Ordonarea copiilor se poate face folosind Iterator;
- Composite trebuie să îşi şteargă proprii copii;
- Caching pentru îmbunătățirea performanțelor: putem stoca temporar informația despre copii în părinți.

Exemplul 1 [MIT90]. O bicicletă se poate descompune în părți astfel:

Bicicleta

Roată

Butuc de roată

Spițe

Jantă

Cameră

Anvelopă etc...

Ghidon

Şa

...

Fiind dată o componentă a bicicletei, dorim să determinăm greutatea şi pretul său indiferent dacă aceasta este formată din subcomponente sau nu. Codul client trebuie să trateze uniform o roată şi un reflector, spre exemplu. Soluția este ca toate componentele bicicletei să implementeze o interfață comună:

```
class BicycleComponent {
      int weight();
      float cost();  }
```

Implementarea *Wheel.weight()* poate apela *weight* asupra părților sale, dar aceasta nu interesează clientul.

Exemplul 2 [Gamm94]. Să presupunem că avem o interfață în Swing, în care diferite componente (elemente IUG: butoane, meniuri, arii de text etc.) sunt grupate cu ajutorul componentelor-container şi a ferestrelor. Ar

fi util ca diferitele operații de manevrare a componentelor să trateze uniform componentele individuale (Widget) şi componentele -container (WidgetContainer) ce conțin alte componente.

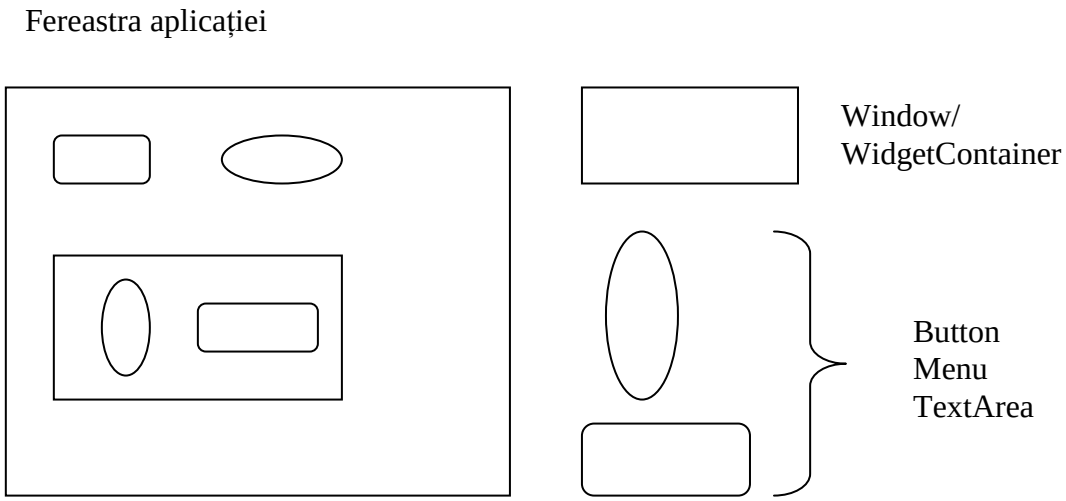


Figura 8.19. Exemplul 2.

Idei de Implementare

- Implementare greşită (greoaie)
  - pentru fiecare operație se tratează individual cu fiecare categorie de obiecte; codul va fi complex, neuniform, şi vom avea mult cod duplicat;
- Programare orientată spre interfețe
  - se tratează uniform operațiile asociate fiecărui Widget, totuşi, container-ele sunt tratate diferit .

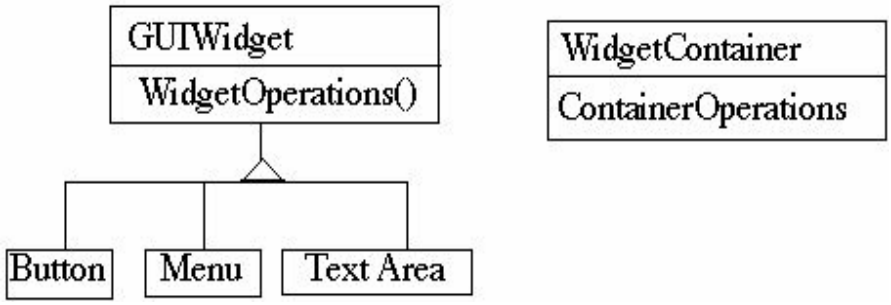


Figura 8.20. Programare orientată spre interfețe

Implementare greşită

```
class Window {   Buttons[] myButtons;
Menus[] myMenus;
```

```
TextAreas[] myTextAreas;
WidgetContainer[] myContainers;

public void update() {
if ( myButtons != null )
  for ( int k = 0; k < myButtons.length(); k++ )
                              myButtons[k].refresh();
 if ( myMenus != null )
   for ( int k = 0; k < myMenus.length(); k++ )
                              myMenus[k].display();

 if ( myTextAreas != null )
   for ( int k = 0; k < myButtons.length(); k++ )
                              myTextAreas[k].refresh();

 if ( myContainers != null )
   for (int k = 0; k < myContainers.length();k++)
                              myContainers[k].updateElements(); // ...etc.   }

public void fooOperation()      {if ( conditii ) etc.   // iar.. . .       } }
```

Programare orientată spre Interfețe

```
class Window {
      GUIWidgets[] myWidgets;
      WidgetContainer[] myContainers;

      public void update() {
                if(myWidgets != null)
                     for (int k = 0; k < myWidgets.length(); k++)
                                    myWidgets[k].update();
               if(myContainers != null)
                     for (int k = 0; k < myContainers.length(); k++)
                                    myContainers[k].updateElements();
// .. .. etc.   }}
```

Aplicarea Composite la Problema Widget

După cum se observă în codul următor, Component implementează comportamentul implicit când este posibil (Button, Menu, etc. suprascriu metodele Component când este necesar), iar WidgetContainer va trebui să suprascrie toate operațiile Widget.

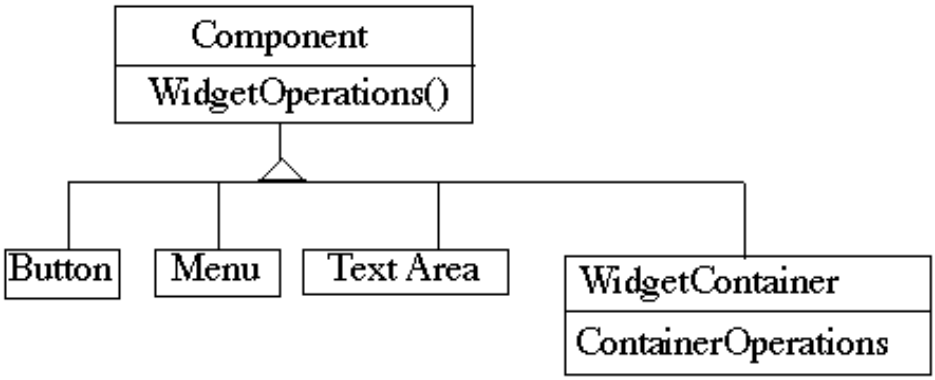


Figura 8.21. Soluția Composite

```
class WidgetContainer {
Component[] myComponents;

public void update() {
if (myComponents != null)
for (int k = 0; k < myComponents.length(); k++)    myComponents[k].update();  }}
```

Operațiile Container (adăugarea, ştergerea şi gestionarea componentelor în compus) pot fi plasate în Component (pentru transparență) sau în Composite (pentru siguranță).

Abordarea Pro-Transparență

Declararea operațiilor container în Component conferă tuturor subclaselor aceeaşi interfață (toate subclasele pot fi tratate la fel), dar costă siguranța, întrucât clienții pot face lucruri absurde (spre exemplu, adăugarea de obiecte la frunze). Pentru a îmbunătăți siguranța metodei *GetComposite()* în această situație se poate folosi abordarea următoare.

Funcția *GetComposite()*

```
class Composite;

class Component {
    public:
        //...
    virtual Composite* GetComposite() { return 0; }
};


class Composite : public Component {
    public:
    void Add(Component*);
```

```
    // ...
    virtual Composite* GetComposite() { return this; }
};


class Leaf : public Component {
        // ...
};

Composite* aComposite = new Composite;
Leaf* aLeaf = new Leaf;

Component* aComponent;
Composite* test;

aComponent = aComposite;
if (test = aComponent->GetComposite()) {
  test->Add(new Leaf);
}

aComponent = aLeaf;
if (test = aComponent->GetComposite()) {
  test->Add(new Leaf); // will not add leaf
}
```

Abordarea Pro-Siguranță

Declararea operațiilor container în WidgetContainer este mai sigură (adăugarea sau ştergerea Widget la non-WidgetContainer este o eroare). Răspunsul potrivit la adăugarea unei TextArea într-un Button (spre exemplu), ar fi o excepție sau să nu se întâmple nimic, dar astfel se va încălca Principiul Substituției al lui Liskov.

Exerciții

1. Completați implementarea Exemplului 1.

2. Folosiți Composite pentru reprezentarea şi parcurgerea unui arbore binar folosit în evaluarea expresiilor matematice (scrierea postfixată /forma poloneză inversă).

8.5. Şablonul FLYWEIGHT

Scop şi aplicabilitate

Şablonul Flyweight foloseşte partajarea pentru a susține implementarea eficientă a unui număr mare de obiecte cu granulație fină (structură complexă)- de exemplu caractere individuale sau iconițe pe un ecran. Un Flyweight este un obiect partajat care poate fi folosit simultan în contexte diferite, astfel că numărul de instanțe se reduce semnificativ, prin recunoaşterea faptului că de fapt clasele sunt la fel cu excepția câtorva *parametri*. Conceptul cheie al şablonului este distincția între *stare intrinsecă* (păstrată în Flyweight, independentă de context) şi *stare extrinsecă* (care depinde şi variază în funcție de context, deci nu poate fi partajată).

Folosiți acest şablon când toate condițiile de mai jos sunt îndeplinite simultan:

- O aplicație foloseşte un număr mare de obiecte;
- Costurile de stocare sunt ridicate din cauza numărului mare de obiecte;
- Cea mai mare parte din starea unui obiect poate fi făcută extrinsecă;
- Multe grupuri de obiecte se pot înlocui de un număr relativ mic de obiecte partajate, odată ce am înlăturat starea extrinsecă;
- Aplicația nu depinde de identitatea obiectelor. Întrucât obiectele Flyweight pot fi partajate, testele de identitate vor întoarce *true* pentru obiecte conceptual distincte.

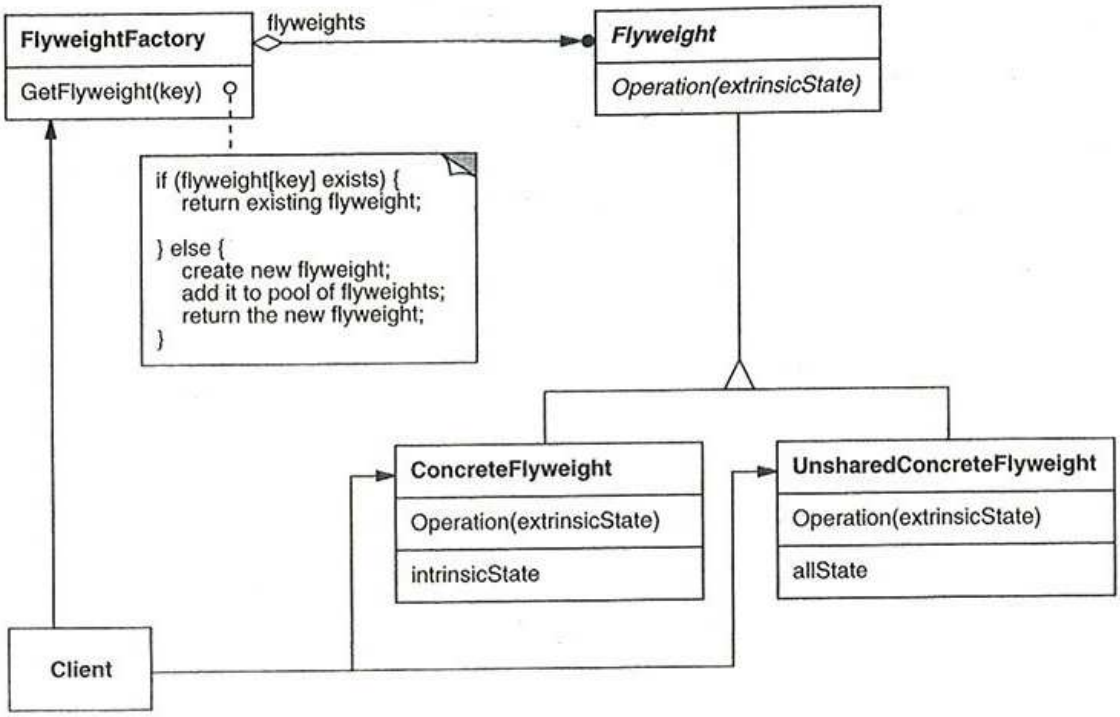


Figura 8.22. Structura şablonului Flyweight

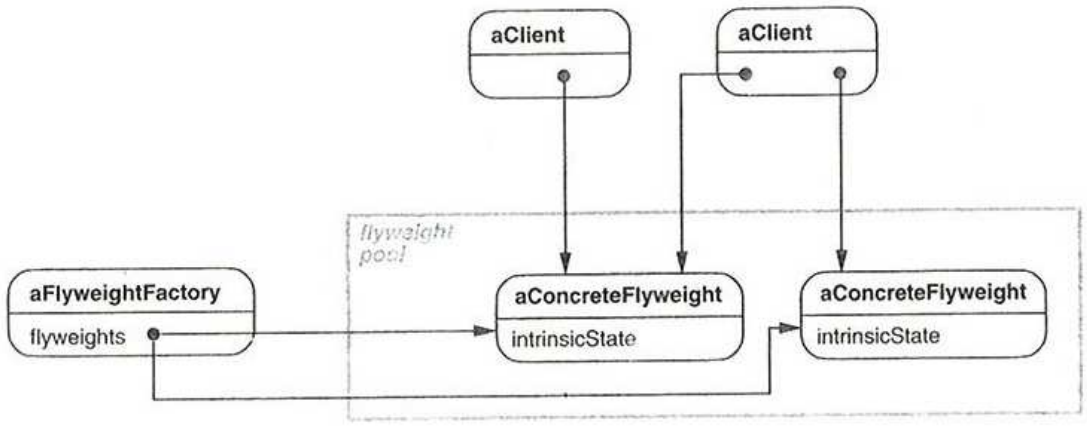


Figura 8.23. Structură- diagramă de obiecte (partajarea)

Participanți [Gamm94, Fig.8.23]

- Flyweight
  - Declară o interfață prin care flyweight primesc starea extrinsecă şi pot acționa asupra ei;
- ConcreteFlyweight
  - Implementează interfața Flyweight şi adaugă stocarea stării intrinseci- independentă de contextul obiectului; trebuie să fie partajabil;
- UnsharedConcreteFlyweight
  - Nu *toate* subclasele Flyweight trebuie să fie partajate, deseori obiectele UnsharedConcreteFlyweight au obiecte ConcreteFlyweight drept copii la un anumit nivel al structurii de obiecte Flyweight;
- FlyweightFactory
  - Creează şi administrează obiectele Flyweight;
  - Se asigură că flyweight-urile sunt partajate corect; când un client necesită un flyweight, FlyweightFactory furnizează o instanță existentă sau creează una dacă astfel de instanță nu există;
- Client
  - Menține o referință la flyweight(-uri);
  - Calculează sau stochează starea extrinsecă a flyweight(-urilor).

Colaborări [Gamm94]

- Starea de care flyweight are nevoie pentru a funcționa trebuie să fie ori intrinsecă (stocată în obiectul ConcreteFlyweight), ori extrinsecă (stocată sau calculată de obiectele Client, şi transmisă când Clienții invocă operațiile obiectelor flyweight);
- Clienții nu trebuie să instanțieze ConcreteFlyweight direct, ci trebuie să obțină aceste obiecte doar de la FlyweightFactory, pentru a fi siguri că ele sunt partajate corect.

Consecințe [Gamm94]

- Flyweight-urile adaugă costuri la momentul execuției (transferul, găsirea şi/sau calcularea stării extrinseci), dar aceste costuri sunt compensate de spațiul eliberat prin fiolosirea şablonului;
- Cantitatea de spațiu economisit depinde de mai mulți factori:
  - Reducerea numărului total de instanțe prin partajare;
  - Dimensiunea starii intrinseci pentru fiecare obiect;
  - Dacă starea extrinsecă este calculată sau stocată;
- Cel mai mult spațiu se economiseşte dacă obiectele folosesc cantități susbstanțiale de stare intrinsecă şi extrinsecă, şi dacă starea extrinsecă se poate calcula- în loc să fie stocată.

Implementare [Gamm94]

- Ştergerea stării extrinseci
  - Aplicabilitatea şablonului depinde de cât este de uşor să identificăm starea externă şi să o scoatem în afara obiectelor partajate (ideal ar fi ca starea externă să poată fi calculată, nu stocată);
- Gestionarea obiectelor partajate
  - Întrucât obiectele sunt partajate, clienții nu trebuie să le instanțieze direct; deseori obiectele FlyweightFactory folosesc un cadru asociativ ce permite clienților să caute flyweight-urile de interes (de exemplu o tabelă de flyweight indexată după codul caracterului);
- Numărul de instanțe ce se alocă pentru flyweight se decide pe măsură ce aceste instanțe devin necesare în program, prin intermediul unei clase FlyweightFactory;
- Această clasă este un Singleton, deoarece trebuie să țină cont dacă o instanță particulară a fost deja creată sau nu (şi întoarce o referință sau o instanță nouă).

Şabloane relaționate [Gamm94]

- Flyweight se combină adesea cu Composite pentru a implementa o structură ierarhică din punct de vedere logic sub forma unui graf direcționat aciclic cu noduri-frunză partajate;
- Este deseori potrivită implementarea obiectelor State şi Strategy ca Flyweight-uri.

Exemplul 1 [Coop98]

Dacă avem de desenat multe iconițe în interiorul unui folder, fiecare reprezentând o persoană sau un fişier de date, nu are sens să avem câte o instanță individuală a unei clase pentru fiecare dintre ele, care să memoreze numele persoanei şi poziția iconiței pe ecran. De obicei aceste iconițe sunt una sau câteva imagini similare, iar poziția se calculează dinamic în funcție de dimensiunea ferestrei. Să presupunem că dorim să desenăm iconița unui director, cu un nume sub ea, pentru fiecare persoană a unei organizații. Directoarele pot avea două aspecte: Selectat şi Deselectat. Creem un FolderFactory care întoarce versiunea selectată sau pe cea neselectată (doar două instanțe în program):

```
class FolderFactory
{Folder unSelected, Selected;
  public FolderFactory()
  {Color brown = new Color(0x5f5f1c);
   Selected = new Folder(brown);
   unSelected = new Folder(Color.yellow);}

public Folder getFolder(boolean isSelected)
{if (isSelected) return Selected;
else  return unSelected;}}
```

Vom folosi Flyweight pentru a transmite coordonatele şi numele de scris (datele extrinseci) sub fiecare folder când îl desenăm. Clasa Folder completă este:

```
class Folder extends JPanel
{private Color color;
final int W = 50, H = 30;
public Folder(Color c)
{color = c;}

public void Draw(Graphics g, int tx, int ty, String name)
{g.setColor(Color.black); //outline
g.drawRect(tx, ty, W, H);
g.drawString(name, tx, ty + H+15); //titlu
g.setColor(color); //fill
g.fillRect(tx+1, ty+1, W-1, H-1);
g.setColor(Color.lightGray); //linie
g.drawLine(tx+1, ty+H-5, tx+W-1, ty+H-5);
g.setColor(Color.black); //linii umbră
g.drawLine(tx, ty+H+1, tx+W-1, ty+H+1);
g.drawLine(tx+W+1, ty, tx+W+1, ty+H);
g.setColor(Color.white); //linii de evidenţiere
g.drawLine(tx+1, ty+1, tx+W-1, ty+1);
g.drawLine(tx+1, ty+1, tx+1, ty+H-1);}
}
```

Coordonatele de poziționare se calculează dinamic în timpul funcției *paint*, şi se foloseşte FolderFactory să ne întoarcă instanța corectă de fiecare dată:

```
public void paint(Graphics g)
{Folder f;
String name;
int j = 0; //numărul în linie
int row = Top; //începe stânga sus
int x = Left;
//parcurge toate numele şi folder-ele
```

```
for (int i = 0; i< names.size(); i++)
{name = (String)names.elementAt(i);
if(name.equals(selectedName))
   f = fact.getFolder(true);
else
   f = fact.getFolder(false);

//folder-ul se desenează în acest punct
f.Draw(g, x, row, name);
x = x + HSpace; //următoarea poziție
j++;
if (j >= HCount) //următoarea linie
{j = 0;
row += VSpace;
x = Left;}
}}
```

La selectarea unui folder (când se trece cu mouse-ul pe deasupra) reținem numele folder-ului care a fost selectat şi cerem Factory să întoarcă un folder “selectat” pentru el. Întrucât folder-ele nu sunt instanțe individuale, nu putem asculta mouse-ul din cadrul lor, şi nu putem spune celorlalte instanțe să se deselecteze decât dacă ascultăm mouse-ul la nivelul ferestrei: dacă mouse-ul este în interiorul unui Rectangle, facem ca numele corespunzător să fie numele selectat:

```
public void mouseMoved(MouseEvent e)
{int j = 0; //numărul în linie
int row = Top; //începe stânga sus
int x = Left;
//toate numele şi folder-ele
for (int i = 0; i< names.size(); i++)
{
//vezi dacă folder-ul conține mouse-ul
Rectangle r = new Rectangle(x,row,W,H);
if (r.contains(e.getX(), e.getY()))
{
selectedName=(String)names.elementAt(i);
repaint();}
x = x + HSpace; //următoarea poziție
j++;
if (j >= HCount) //următorul rând
{j = 0;
row += VSpace;
x = Left;}
}
```

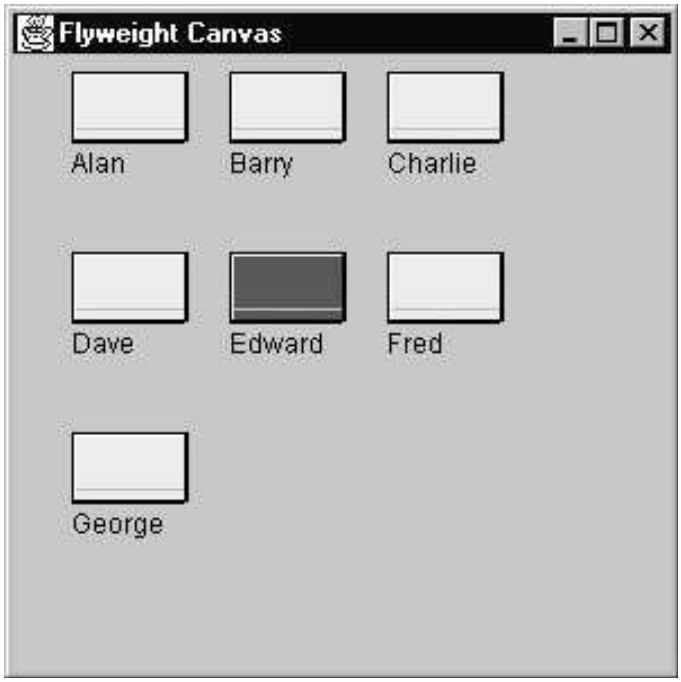


Figura 8.24. Exemplul 1

Exemplul 2 [Gamm94]

Fiecare caracter al unui font este reprezentat ca o singură instanță a unei clase Caracter, iar pozițiile unde se desenează caracterul sunt reținute ca date externe astfel încât există o singură instanță a fiecărui caracter care apare în text (Figurile 5.82, 5.83)

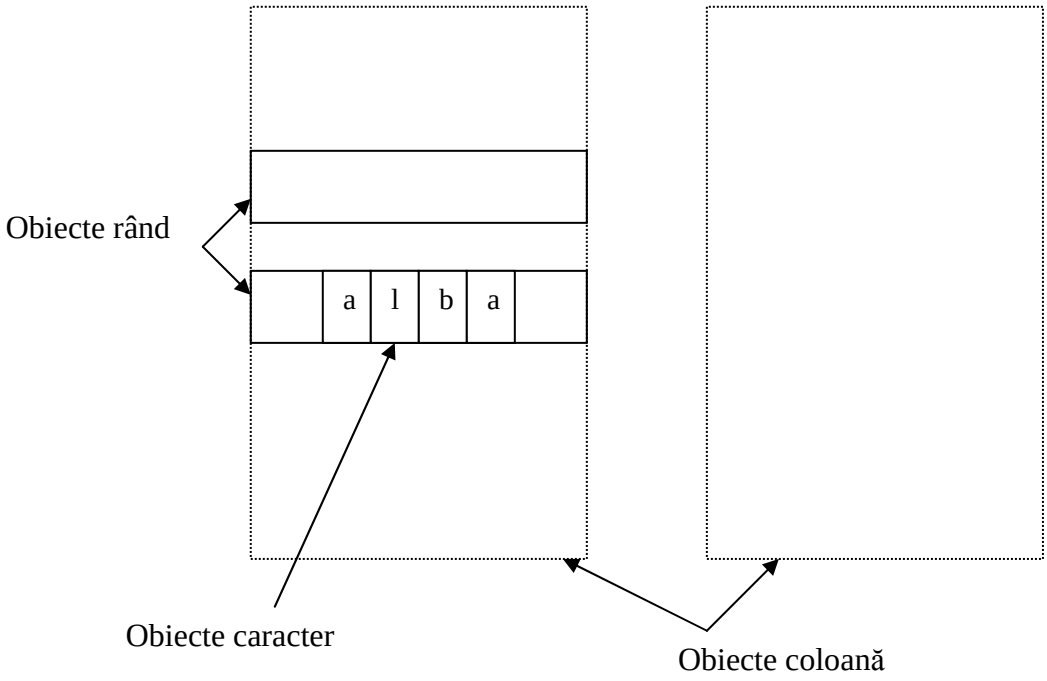


Figura 8.25. Exemplul 2

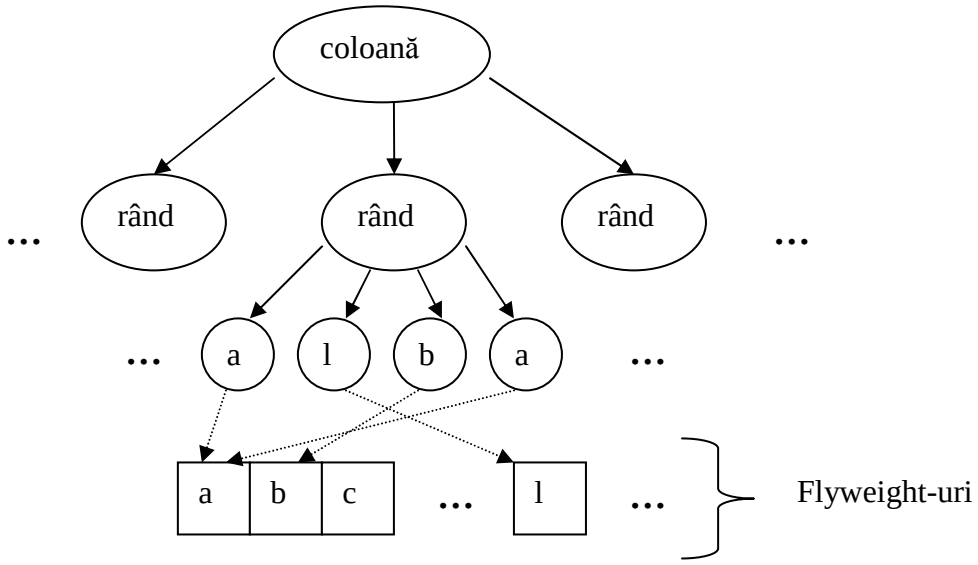


Figura 8.26. Exemplul 2 (cont.)

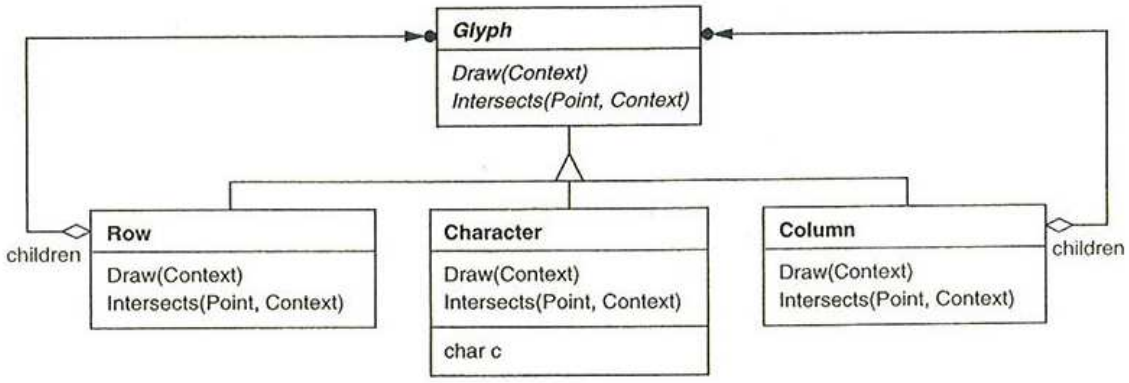


Figura 8.27. Exemplul 2 (cont.)

(*Glyph*[1] este Flyweight, *Character* este ConcreteFlyweight iar *Row* şi *Column* sunt UnsharedConcreteFlyweight).

Exerciții.

1. Implementați un joc ce desenează pe ecran cercuri, timp de trei minute. Centrul cercului este generat aleator, la fel şi culoarea sa (galben, roşu sau albastru) şi raza (un întreg între 1 şi 5). Dacă se execută click cu mouse-ul în interiorul unui cerc, cercul se şterge şi punctajul jucătorului creşte cu o unitate (dacă punctul

[1] Literă incizată, hieroglif

este „acoperit” de mai multe cercuri, se şterg toate, şi punctajul creşte cu un număr de puncte egal cu numărul de cercuri şterse). Folosiți Flyweight.

2. Un personaj se mişcă în cele 4 direcții principale, folosind săgeţile, printr-un labirint. În labirint există 3 tipuri de poziții (o poziție fiind desenată pe un pătrat de 10*10 pixeli): obstacole (personajul nu poate trece prin ele), capcane (personajul pierde o viață la trecerea prin ele) şi poziții libere. Inițial personajul are trei vieți, la 0 vieți jocul este pierdut. Scopul personajului este să ajungă într-un punct final cu un număr minim de mutări. Folosiți Flyweight în reprezentarea labirintului.

3.Folosiți Flyweight în implementarea cărămizilor din jocul Brickles (Cap. 1, Ex.2).

8.6. Şablonul CHAIN OF RESPONSIBILITY (Lanț de Responsabilități)

Scop şi aplicabilitate

Şablonul Lanț de Responsabilități decuplează transmițătorul cererii de primitorul acesteia, oferind mai multor obiecte şansa de a trata cererea. Primitorii sunt ordonați într-un lanț şi cererea este transmisă de-a lungul lanțului, până ce un obiect o tratează.

Şablonul se foloseşte atunci cand:

- mai mult decât un obiect poate trata o cerere -şi nu cunoaştem apriori cine o va trata;
- mulțimea de obiecte ce pot trata cererea trebuie să fie specificabilă dinamic;
- trimitem o cerere la mai multe obiecte fără să specificăm primitorul.

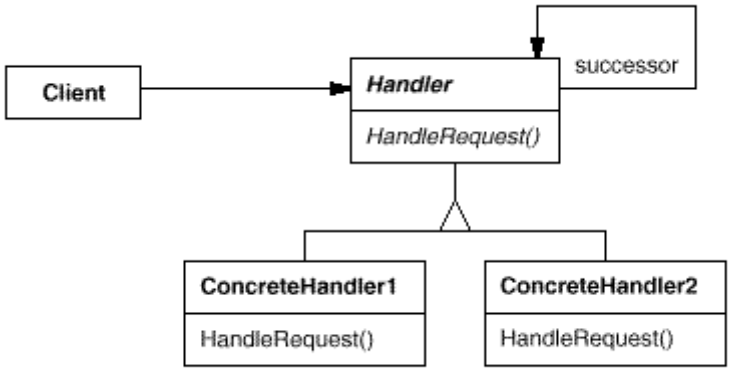


Figura 8.28. Structura şablonului Lanț de Responsabilități

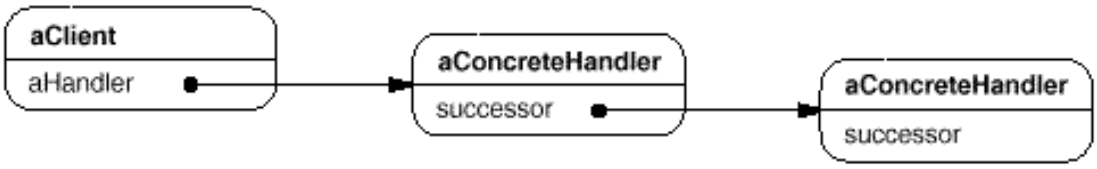


Figura 8.29. O structura de obiecte în şablonul Lanț de Responsabilități

Participanți si Colaborări [Gamm94, Fig. 8.28]

- Handler
  - defineşte interfața pentru tratarea cererilor;
  - poate implementa link-ul către successor;
- ConcreteHandler
  - fie tratează cererea pentru care este răspunzător, dacă este posibil…
  - fie trimite cererea mai departe succesorului său;
- Client
  - inițiază transmiterea cererii către un obiect ConcreteHandler din lanț.

Consecințe [Gamm94]

Avantaje

- Cuplaj redus
  - clientul (trimițătorul) nu trebuie să ştie cine îi va trata cererea;
  - trimițătorul şi primitorul nu se cunosc unul pe celălalt;
  - în loc ca trimițătorul să cunoască toți potențialii primitori, se păstrează doar o referință la următorul Handler din lanț (deci se simplifică interconexiunile dintre obiecte);
- Flexibilitate în asignarea responsabilităților către obiecte
  - responsabilitățile se pot adăuga sau schimba;
  - lanțul se poate modifica la execuție.

Dezavantaje

- Cererile pot rămâne netratate, dacă lanțul nu a fost configurat corect.

Lanț de Comenzi [Gamm94]

- Lanțul este configurat ierarhic, ca în armată:
  - se efectuează o cerere;
  - se transmite în partea superioara a lanțului până când cineva are autoritatea să răspundă cererii (Fig. 8.30)

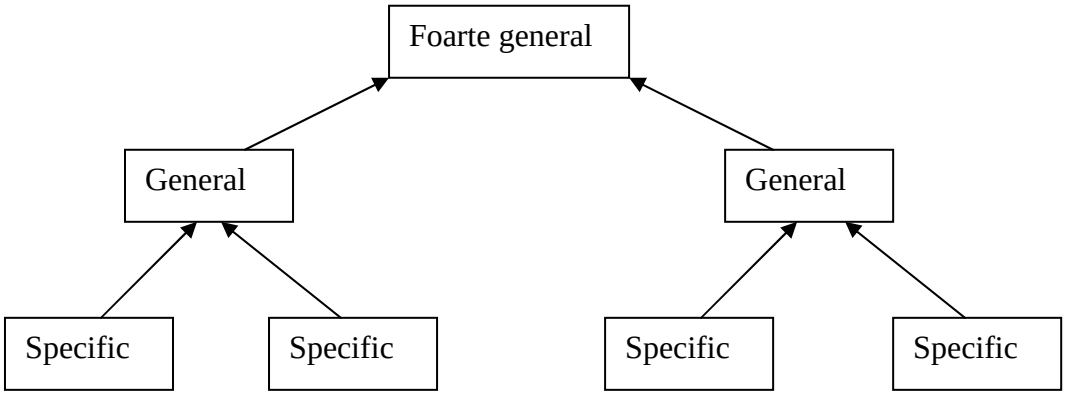


Figura 8.30. Lanț ierarhic

Implementarea Lanțului de Succesori [Gamm94]

- Definirea unei legături noi
  - fiecare handler are un link către succesorul său;
- Folosirea legăturilor existente
  - handler-ii concreți pot avea deja pointeri către succesorii lor (pot fi folosiți);
  - referințe către părinți într-o ierarhie parte-întreg
    - pot defini succesorul unei părți;
  - economiseşte muncă şi spațiu ...
  - ... dar trebuie să reflecte lanțul de responsabilități necesar.

Conectarea Succesorilor [Gamm94]

Dacă nu există referințe pre-existente pentru construirea lanțului, link-ul către succesor este de obicei gestionat de Handler. O implementare implicită doar trimite cererea succesorului, eliberând clasele ConcreteHandler neinteresate de tratarea cererii.

Exemplu de Implementare (C++)

```
class HelpHandler {
  public:
        HelpHandler(HelpHandler* s) : successor(s) { }
     virtual void HandleHelp();
  private: HelpHandler* _successor;
};
void HelpHandler::HandleHelp () {
  if (_successor) _successor->HandleHelp(); }
```

Reprezentarea Cererilor [Gamm94]

- Fiecare cerere este codificată explicit
  - convenabil şi sigur;
  - inflexibil
    - limitat la mulțimea fixată de cereri definite de Handler;
- Handler unic cu parametri
  - mai flexibil;
  - dar necesită instrucțiuni condiționale pentru distribuirea cererilor;
  - transmiterea parametrilor este mai nesigură din punctul de vedere al parametrilor;
- Handler unic cu Parametru de tip cerere-obiect
  - subclasele mai degrabă extind, decât suprascriu metoda din Handler.

Exemplul 1. Sistemul de Help dependent de Context [Coop98]

- Help senzitiv la context
  - o cerere de help este tratată de unul din mai multe obiecte din IU (interfața utilizator);
- Contextul decide care obiect anume tratează cererea;
- Obiectul care inițiază cererea nu cunoaşte obiectul care va oferi (eventual) ajutor.

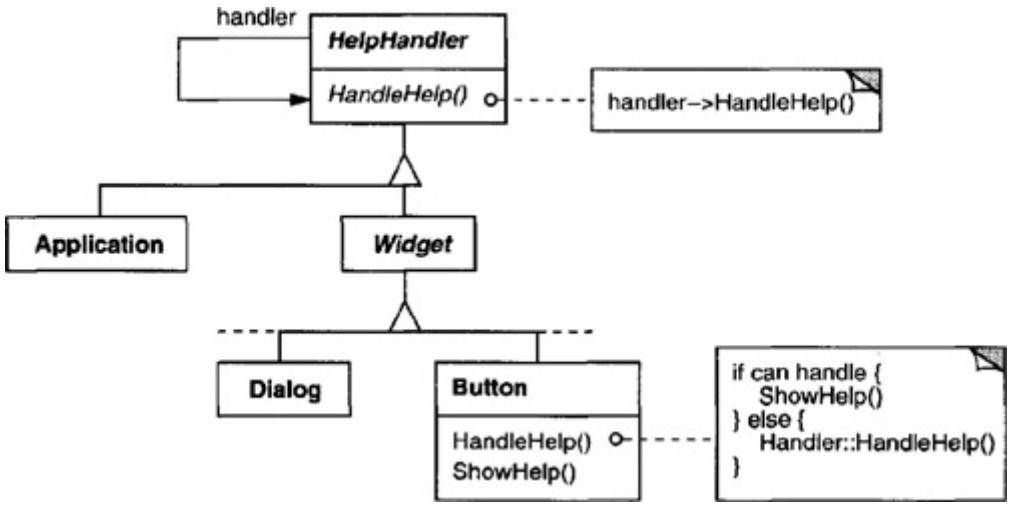


Figura 8.31. Help senzitiv la context

Codificare explicită

```
abstract class HardCodedHandler {
private HardCodedHandler successor;
public HardCodedHandler(HardCodedHandler aSuccessor){ successor = aSuccessor; }
```

```
public void handleOpen()
{ successor.handleOpen(); }
public void handleClose()
{ successor.handleClose(); }
public void handleNew( String fileName)   { successor.handleNew( fileName ); }}
```

Handler unic cu parametri

```
abstract class SingleHandler {
private SingleHandler successor;
public SingleHandler( SingleHandler aSuccessor) {
     successor = aSuccessor; }

public void handle( String request) { successor.handle( request );   }}

class ConcreteOpenHandler extends SingleHandler {
public void handle( String request) {
switch ( request ) {
       case "Open" : // executa ce trebuie
       case "Close" : // …
       case "New" : // …
       default: successor.handle( request );      }   }}
```

Parametru de tip cerere-obiect

```
void Handler::HandleRequest (Request* theRequest) {
   switch (theRequest->GetKind())
      {case Open:
            HandleOpen((OpenRequest*) theRequest); break;
       case New:
            HandleNew((NewRequest*) theRequest);
      // ...
       break;
       default:
     // ...
       break;    }}
```

Exerciții.

1. Aprobarea cheltuielilor într-o întreprindere ține cont de următoarele criterii: cheltuielile zilnice de până la 5000 de lei sunt aprobate de un contabil, reparațiile şi investițiile de până la 15000 de lei sunt aprobate de contabilul şef, iar tot ce depăşeşte 15000 de lei poate fi aprobat numai de directorul întreprinderii. Simulați circulația cererilor în sistem folosind Lanț de Responsabilități: cererile aprobate sunt trecute într-o tabelă –

Cheltuieli_Aprobate, iar restul se şterg din sistem. O cerere conține informațiile: Nume, Scop, Valoare, Termen limită.

2. La Secția de Urgențe a unui Spital de Pediatrie pacienții sunt triați după următoarele criterii: fracturile şi traumatismele majore cu hemoragie sunt trimise imediat la blocul operator, stările infecțioase cu febră >39 şi stările ce implică deshidratarea sunt internate, iar indigestiile, traumatismele minore şi stările infecțioase cu febra <=39 sunt tratate ambulator. Programul ține evidența a trei tabele: Bloc_Operator, Internați, si Tratament_Ambulator, în care adaugă pacienții, după caz, cu următoarele informații: Nume, Diagnostic, Vârstă, Data_şi_ora_prezentării, Medic. Medicul care răspunde de caz este ales în funcție de diagnostic, de orarul său şi de modul de tratare (chirurgie, internare sau ambulator), pe baza informațiilor dintr-o tabelă de intrare. Folosiți Lanț de Responsabilități.

8.7. Şablonul DECORATOR (Wrapper)- Schimbarea învelişului unui obiect

Scop şi aplicabilitate

Şablonul Decorator *adaugă responsabilități unui obiect particular, mai degrabă decât clasei sale*. Şablonul ataşează dinamic responsabilități adiționale unui obiect, fiind o alternativă flexibilă la derivare. Adăugarea de responsabilități obiectelor se face transparent şi dinamic (i.e. fără să afecteze alte obiecte) şi se foloseşte atunci când extinderea prin derivare nu este practică (pentru că poate duce la prea multe subclase).

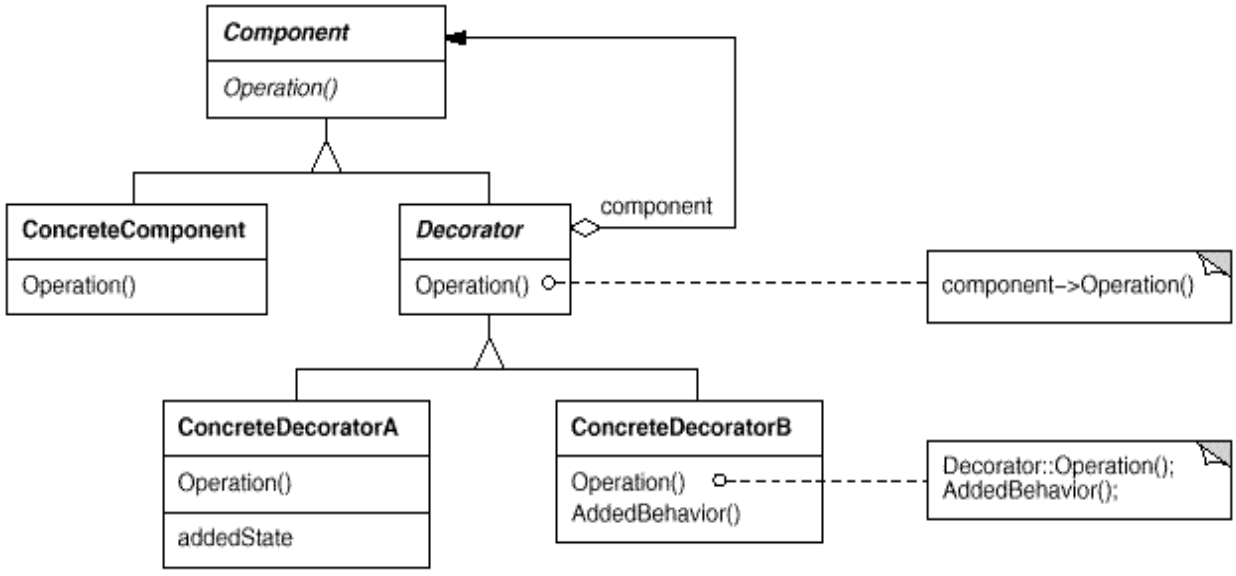


Figura 8.32. Structura şablonului Decorator

Participanți şi colaborări [Gamm94, Fig. 8.32]

- Component
  - defineşte interfața obiectelor cărora li se pot adăuga dinamic responsabilități;
- ConcreteComponent
  - obiectele “de bază" la care se pot adăuga responsabilități;
- Decorator
  - defineşte o interfață conformă cu interfața clasei Component (pentru transparență);
  - menține o referință către un obiect Component;
- ConcreteDecorator
  - adaugă responsabilități componentei.

Consecințe [Gamm94]

Avantaje

- Mai flexibil decât moştenirea statică
  - permite amestecarea şi potrivirea responsabilităților;
  - permite aplicarea unei proprietăți de două ori;
- Evită clasele încărcate de prea multe cracteristici specifice în partea de sus a ierarhiei
  - abordarea "*plăteşte-pe-măsură-ce-consumi*";
  - uşor de definit noi tipuri de decorațiuni.

Dezavantaje

- O mulțime de obiecte mici
  - uşor de adaptat, dar dificil de învățat şi depanat;
- Un decorator şi componenta sa nu sunt identice
  - verificarea identității obiectelor poate genera probleme
    - ex. if (aComponent instanceof TextView) …

Detalii de Implementare [Gamm94]

- Păstrați Decoratorii cât mai lejeri
  - Nu plasați date membru în VisualComponent;
  - Folosiți-i pentru modelarea interfeței;
- Clasa abstractă Decorator se poate omite
  - dacă este necesară doar o decorațiune;
  - subclasele ar putea plăti pentru lucruri ne-necesare.

Şabloane relaționate

Lanț de Responsabilități	**Decorator**
Comparabil cu arhitecturile "orientate pe evenimente"	Comparabil cu arhitecturile stratificate (foile unei cepe)
Obiectele "filtru" sunt de rang egal	Se presupune un obiect "nucleu", toate obiectele "strat" sunt opționale
Utilizatorul vede lanțul ca pe o linie "lansează şi abandonează"	Utilizatorul vede obiectul decorat ca pe un obiect îmbunătățit
O cerere este transmisă până ce un singur obiect filtru o tratează. Mai multe (toate) obiectele filtru *pot* contrib. la tratarea fiecărei cereri.	Un obiect strat întotdeauna execută pre sau post procesare la delegarea cererii.
Toate obiectele ce tratează sunt ca nodurile unei liste înlănțuite – "este necesară condiția de "sfârşit de listă".	Toate obiectele strat în final delegă către un singur obiect nucleu – "nu este necesară tratarea condiției "sfârşit de listă".

Tabel 8.1. Decorator vs. Lanț de Responsabilități [Gamm94]

Spre deosebire de Adapter, Decorator îmbogățeşte un obiect fără să-i schimbe interfața.

Exemplul 1 [Gamm94]

Într-o interfață utilizator Swing, un TextView poate avea 2 caracteristici:

- chenare (3 opțiuni: fără, flat, 3D);
- bare de derulare (4 opțiuni: fără, lateral, jos, ambele).

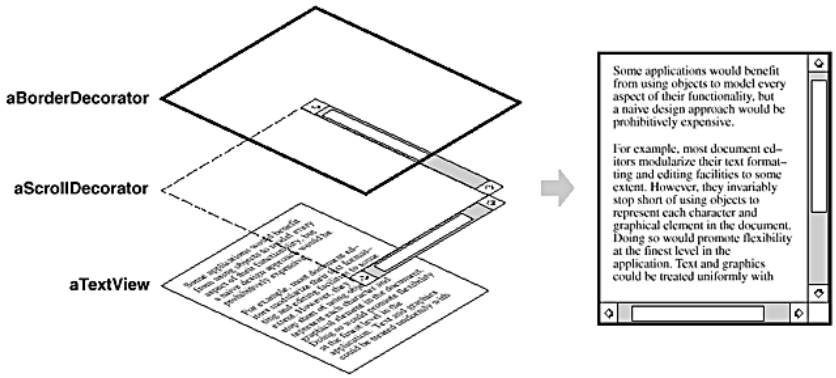


Figura 8.33. Un TextView şi decorațiunile sale

Dacă am folosi derivarea pentru a modela toate combinatiile de opțiuni, am ajunge la 3 x 4 = 12 clase şi subclase (TextView, TextViewWithNoBorder & SideScrollbar, TextViewWithNoBorder & BottomScrollbar, TextViewWithNoBorder & Bottom & SideScrollbar, TextViewWith3DBorder, TextViewWith3DBorder & SideScrollbar, TextViewWith3DBorder&BottomScrollbar, TextViewWith3DBorder&Bottom&SideScrollbar, ...)

Soluția 1. O primă abordare ar fi să folosim compunerea pentru a adăuga componente unui TextView (Figura 8.34). Însă prin compunere, codul clasei TextView nu va mai respecta Principul Deschis-Închis (va trebui modificat la apariția de noi alternative de decorare).

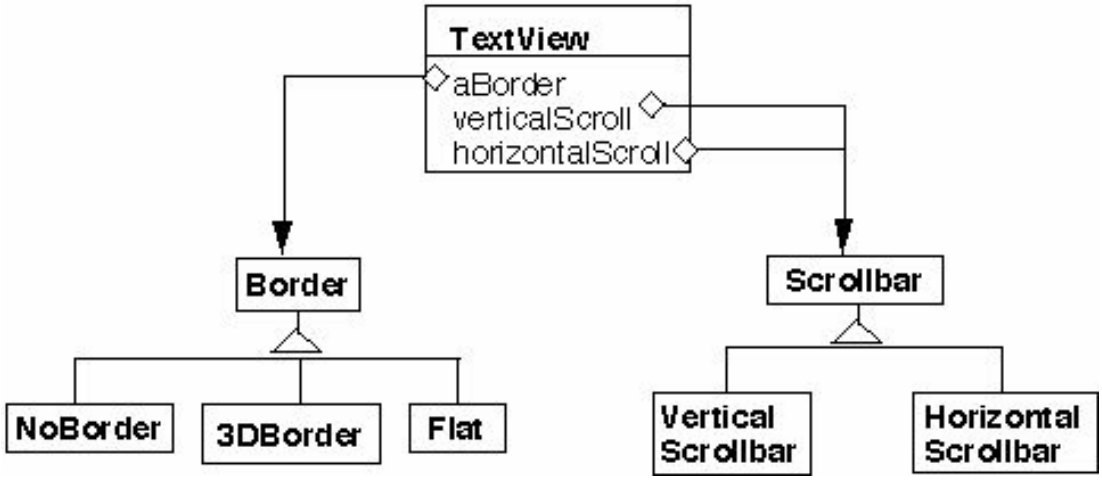


Figura 8.34. Exemplul 1

Soluția 2: O soluție ce implică şablonul Decorator presupune să schimbăm doar învelişul, nu şi conținutul clasei (Figura 8.35), şi conduce la respectarea Principiului Deschis- Închis.

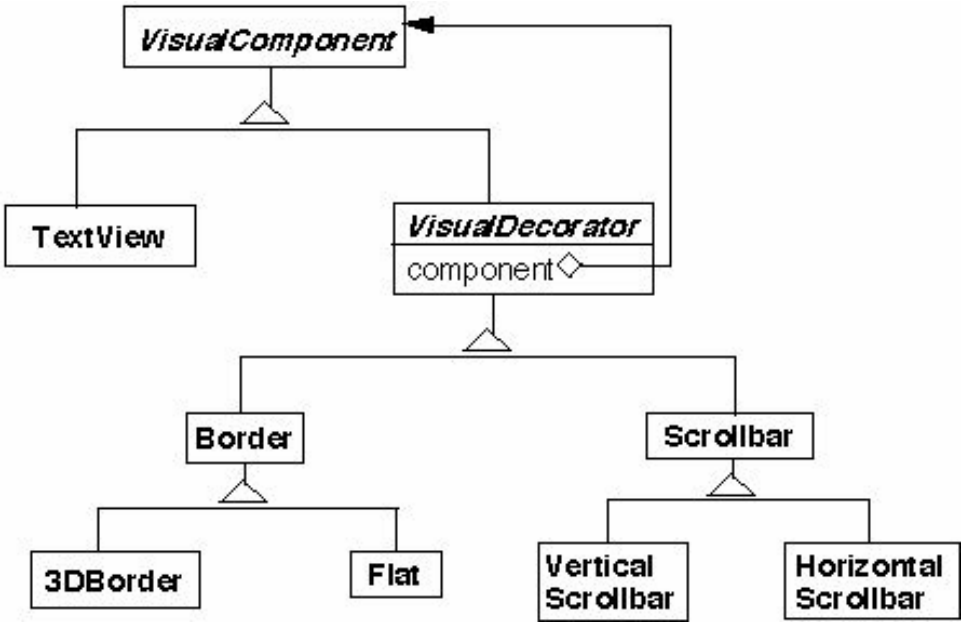


Figura 8.35. Exemplul 1

În această abordare, clasa TextView nu are chenare sau bare de derulare, adăugarea acestora făcându-se *deasupra* unui TextView**.**

Exemplul 2 [Ecke02]

Se poate crea un meniu rezonabil al selecțiilor de bază pentru un automat de cafea, ce pot fi modificate (cu lapte, decafeinizată etc.), folosind decoratori.

```
// compromis între combinațiile de bază și decoratori
import junit.framework.*;

interface DrinkComponent {
  float getTotalCost();
  String getDescription();
}

class Espresso implements DrinkComponent {
  private String description = "Espresso";
  private float cost = 0.75f;
  public float getTotalCost() {return cost;}

  public String getDescription() {return description; }
}
```

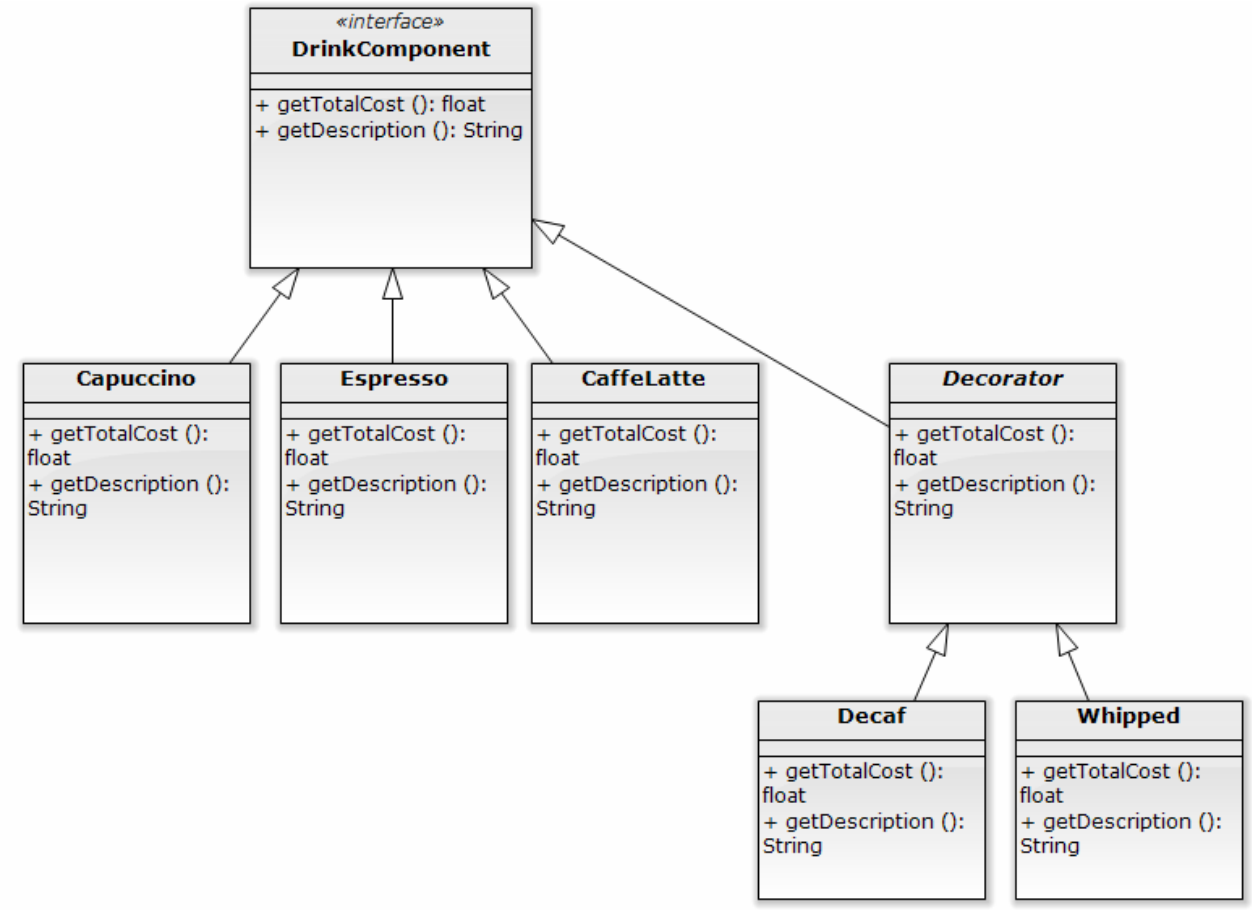


Figura 8.36.Exemplul 2.

```
class Cappuccino implements DrinkComponent {
   private float cost = 1;
   private String description = "Cappuccino";
   public float getTotalCost() { return cost;}

   public String getDescription() { return description; }}

 class CafeLatte implements DrinkComponent {
   private float cost = 1;
   private String description = "Cafe Late";
   public float getTotalCost() { return cost;  }
   public String getDescription() {return description;}
 }

 abstract class Decorator implements DrinkComponent {
   protected DrinkComponent component;
   public Decorator(DrinkComponent component) {
     this.component = component;}

   public float getTotalCost() { return component.getTotalCost(); }
```

```
  public String getDescription() { return component.getDescription(); }
}


class Whipped extends Decorator {
  private float cost = 0.50f;
  public Whipped(DrinkComponent component) {super(component); }

  public float getTotalCost() { return cost + component.getTotalCost(); }

public String getDescription() {
    return component.getDescription() +" whipped cream"; }}


class Decaf extends Decorator{
  public Decaf(DrinkComponent component) {super(component); }

 public String getDescription() {
    return component.getDescription() + "decaf";
  }
}

public class CoffeeShop {
  public void testCappuccino() {
        // Cappucino simplu
    DrinkComponent cappuccino = new Cappuccino();
   System.out.println(cappuccino.getDescription()+":$"+
                                          cappuccino.getTotalCost());
  }

public void testEspresso() {
    // decaf espresso cu lapte
    DrinkComponent espresso = new Whipped(new Decaf(new Espresso()));
    System.out.println(espresso.getDescription()+":$"+
                                          espresso.getTotalCost());
  }
  public static void main(String[] args) {
   //...
  }
}
```

În abordarea de mai sus, dacă dorim să modificăm meniul mai târziu (prin adăugarea unui tip nou de băutură) se creează o singură clasă suplimentară. Dacă se modifică costul cafelei cu lapte când prețul laptelui crește, prin folosirea decoratorilor nu trebuie să modificăm o metodă în fiecare clasă- ci o singură clasă (ar exista câte o clasă pentru fiecare combinație dacă n-am folosi decoratori, şi fiecare metodă de cost ar trebui modificată).

Exerciții.

1. Ornați butoanele Start, Pause şi Restart ale unui joc (spre exemplu, Brickles, Cap 1, Ex. 2) cu chenare şi diverse culori folosind şablonul Decorator, şi două butoane auxiliare Colorează şi Încadrează.

2. Un angajat al unei întreprinderi poate fi membru într-o echipă, şef de echipă sau poate deveni director. Gândiți o schemă adecvată de adăugare de responsabilități, folosind Decorator.

3. “Decorați” buştenii din jocul Hungry Hippau astfel ca în momentul coliziunii cu hipopotamul să îşi schimbe culoarea.

CAP. 9. ARHITECTURI SOFTWARE ORIENTATE SPRE ŞABLOANE: MODEL-VIEW-CONTROLLER

9.1. Introducere în Arhitecturi Software

În sistemele pe scară mare, în care dimensiunea şi complexitatea cresc, predomină *problemele structurale* (şi nu problemele legate de algoritmi şi structuri de date). Problemele structurale cuprind:

- Compunerea componentelor;
- Asignarea de funcționalități elementelor de design;
- Protocoale pentru comunicare, sincronizare, acces la date;
- Scalarea şi performanța;
- Dimensiunile evoluției.

Descrierile tipice ale arhitecturilor software sunt de obicei informale, completate de diagrame dreptunghiuri-şi-linii, care fac apel la intuiție. Precizia acestor descrieri este scăzută, formalizările sunt rare (spre exemplu, *“Am ales o abordare distribuită, orientată obiect pentru tratarea informației.”*)

Deşi descrierile arhitecturilor software sunt informale, ele reuşesc să comunice efectiv datorită vocabularului bogat (ce transmite un *conținut cu o semantică apreciabilă:* şabloane şi stiluri *de organizare a sistemelor software*), şi datoriă faptului că oferă un cadru de înțelegere a preocupărilor la nivel de sistem (proprietățile sistemului şi abilitatea acestuia de a îndeplini cerințele utilizatorilor).

Definiția Arhitecturii Software. “Arhitectura Software implică descrierea elementelor din care sistemul este construit, a interacțiunilor între aceste elemente, a şabloanelor care guvernează compunerea lor şi a constrângerilor asupra acestor şabloane“ [Shaw93].

Arhitectura software este design-ul de nivel înalt ce cuprinde:

- Componente (acestea sunt construite plecând de la alocarea funcționalității)
  - Ex. filtre, obiecte, clienți, servere, tipuri abstracte de date;
- Conexiuni (comunicații)
  - Apel de proceduri, canale de comunicație, difuzarea evenimentelor;
- Constrângeri
  - Cerințe non-funcționale asupra sistemului;
  - Pre/post –condiții.

Impactul Arhitectural al Cerințelor Non-Funcționale

- Performanța
  - Performanța este îmbunătățită prin reducerea comunicațiilor şi prin construirea unui număr mic de sub-sisteme (alcătuirea de componente de granularitate mare)

- Securitatea
  - În sistemele în care trebuie să predomine securitatea se preferă structura stratificată, cu stratul cel mai critic la interior

- Mentenabilitatea
  - Dacă se doreşte ca programul să fie uşor de modificat se construiesc componente separate (i.e. care nu se conțin) şi de granularitate mică. În plus, trebuie evitate structurile de date partajate, şi producătorii de date trebuie separați de consumatori.

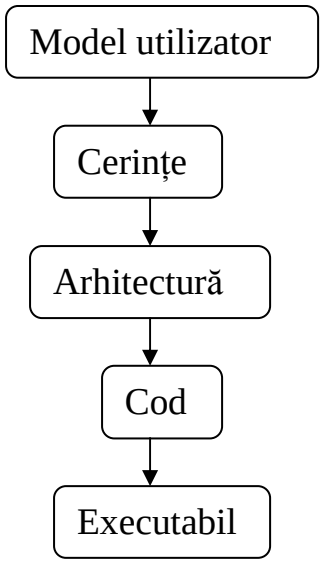


Figura 9.1. Locul Arhitecturii Software în procesul de dezvoltare

Arhitecturi Software Orientate spre Şabloane

După cum s-a definit în Capitolul 5, şabloanele tratează o *problemă recurentă* ce apare într-un *context* specific şi prezintă o *soluție* a ei. Şabloanele oferă un vocabular şi o înțelegere comune pentru principiile de design, fiind utile în scrierea de documentații şi facilitând *refolosirea.* Şabloanele identifică şi specifică abstracțiunile de nivel înalt, ajutând la construirea unor sisteme mai complexe şi eterogene. De asemenea, acestea sprijină construirea de software cu proprietăți definite, prin referirea explicită a cerințelor non-funcționale.

Aşa cum şabloanele de design reprezintă o structură frecventă de componente ce comunică şi rezolvă o problemă generală de design într-un context special, există şabloane şi la nivelul arhitecturii. Acestea din urmă propun o schemă fundamentală de organizare structurală, construită dintr-o mulțime predefinită de componente-conexiuni-constrângeri.

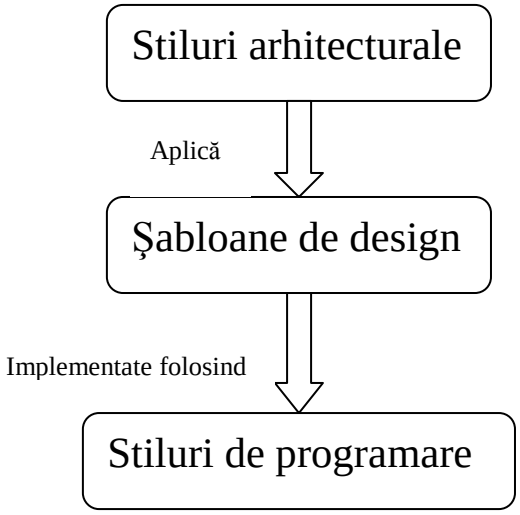


Figura 9.2. Sistemul de Şabloane

Activități ale Design-ului Arhitectural

- Structurarea Sistemului
  - identifică principalele *subsisteme*
  - identifică *comunicațiile* între subsisteme
  - stiluri: blackboard, arhitectura pe nivele, client-server

- Descompunerea Modulară
  - descompune subsistemele în *module*
  - stiluri: orientate-obiect, flux de date (canale şi filtre)

- Modelarea Controlului
  - stabileşte *modelul de control* între părțile sistemului
  - stiluri: apel şi revenire din funcție, sisteme dirijate de evenimente

9.2. Arhitecturi Software Orientate spre Şabloane: Sistemele Orientate-Obiect

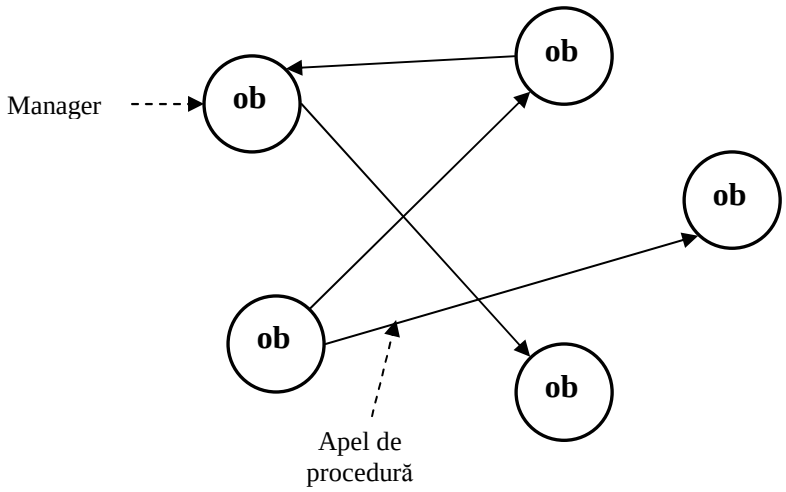


Figura 9.3. Obiectele ca manageri independenți ce comunică prin invocarea de proceduri

În sistemele orientate-obiect, obiectele sunt un fel de manageri ce conservă integritatea resurselor. Încapsularea ascunde detaliile de reprezentare de alte obiecte, iar moştenirea stabileşte o ierarhie între abstracțiuni înrudite. Prin legarea dinamică se determină ce operație se apelează la momentul execuției programului.

Avantajele sistemelor orientate-obiect

- Schimbă implementarea fără să afecteze clienții;
- Descompune problemele în colecții de agenți care colaborează;
- Obiectele sunt reprezentări ale entităților din lumea reală
    - Structura sistemului este uşor de înțeles;

Dezavantajele sistemelor orientate-obiect

- Dependența de identitatea şi interfața altor obiecte;
- Modificările de interfață afectează toți clienții unui anumit serviciu;
- Fluxul de control este dificil de analizat
    - Diagramele de secvențe sunt foarte utile.

9.3. Şablonul arhitectural Model-View-Controller

O provocare a sistemelor interactive este să separăm *nucleul funcțional* de *interfața utilizator*: nucleul funcțional este de obicei stabil, fiind bazat pe cerințe funcționale, în timp ce interfața utilizator este subiect permanent de *adaptare* şi *modificare*.

IU sunt predispuse la modificarea cerințelor (trebuie să fie portabile, să se adapteze, şi să ofere aspecte diferite pentru aceeaşi funcționalitate). În sistemele cu durată lungă de viață, interfețele utilizator sunt ținte aflate mereu în mişcare. Un nucleu funcțional *strâns cuplat* cu IU face flexibilitatea acestor interfețe atât costisitoare cât şi expusă erorilor.

Soluția problemei enunțate mai sus trebuie să țină cont de următoarele aspecte:

- Avem aceeaşi informație, dar prezentări diferite (ex. diverse diagrame);
- Manipularea datelor trebuie să se reflecte instantaneu în afişarea lor şi în comportamentul aplicației;
- Schimbările IU trebuie să fie facile (inclusiv posibile şi la momenul execuției - “look-and-feel”);
- Standardele “look-and-feel" nu ar trebui să afecteze codul funcționalităților de bază.

“Vederile” sunt folosite de relativ mult timp în programare. Ele reprezintă, de fapt, obiecte distincte ce oferă tipuri diferite de acces la o structură comună de date. Spre exemplu, în Java, Iteratorii sunt vederi diferite asupra listei pe care o traversează. Similar, implementările interfeței Map, trebuie să scrie metoda *keySet* ce returnează o mulțime de chei din map. Această mulțime este o vedere (un “*view*”) asupra lui *map*. De asemenea, clasa List are o metodă *subList* ce poate fi privită ca o vedere a sa. La modul ideal, atât vederea cât şi obiectul subiacent ar trebui să fie modificabile (ceea ce nu este întotdeauna posibil: un iterator devine invalid dacă lista asociată se modifică în timpul iterației). Soluția acestor probleme este şablonul MVC (Model-View-Controller), şi constă în divizarea aplicației în trei părți [Busc95]:

- Model (Procesarea)
  - încapsulează date şi funcționalități (nucleu);
  - este independent de *reprezentarea ieşirilor* şi/sau *comportamentul intrărilor*
  - notifică toate vederile (Views) când se modifică datele

- View (Output)
  - afişează informația către utilizator, preluând datele din model;
  - permite vederi multiple pentru acelaşi set de date;
  - creează şi îşi inițializează controlerul asociat;

- Controller (Input)
  - este asociat unei anume vederi (View);

- primeşte intrările sub formă de evenimente;
- traduce evenimentele în *cereri de servicii* pentru Model sau View.

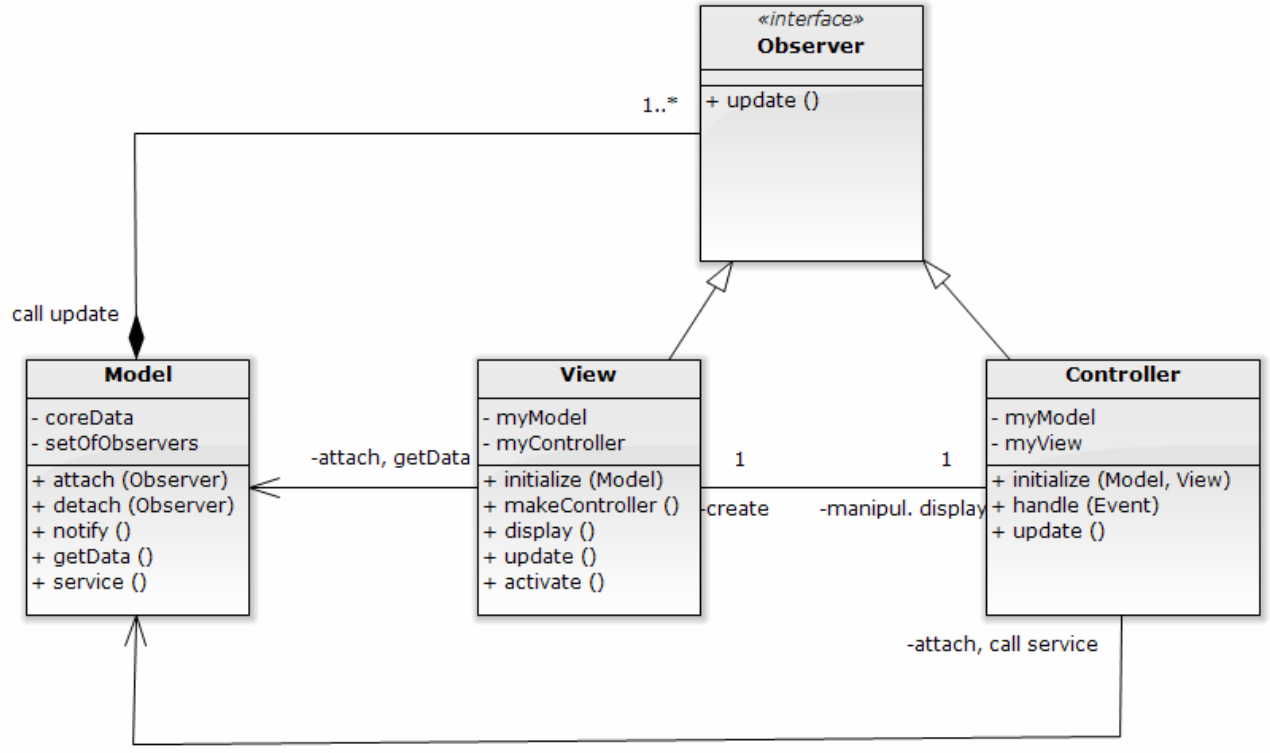


Figura 9.4. Structura MVC

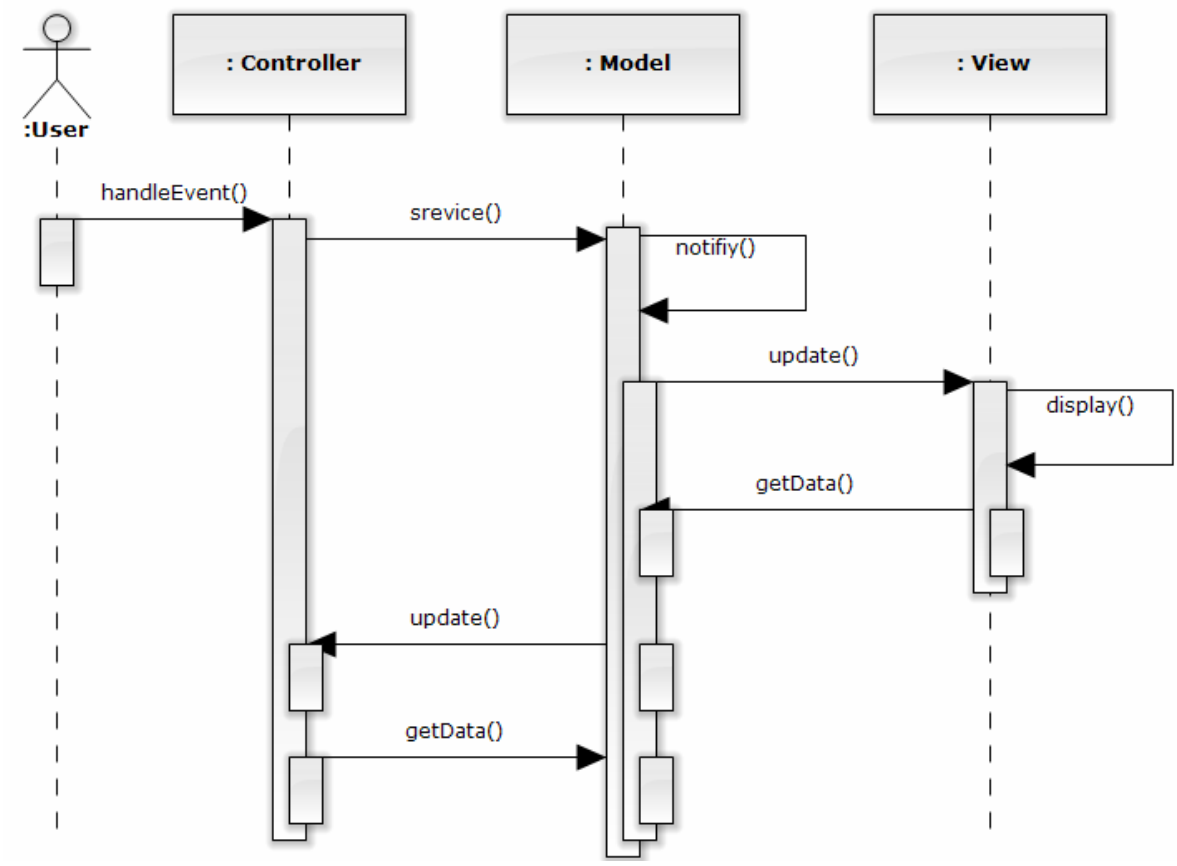


Figura 9.5. Mecanismul de Propagare a Modificărilor

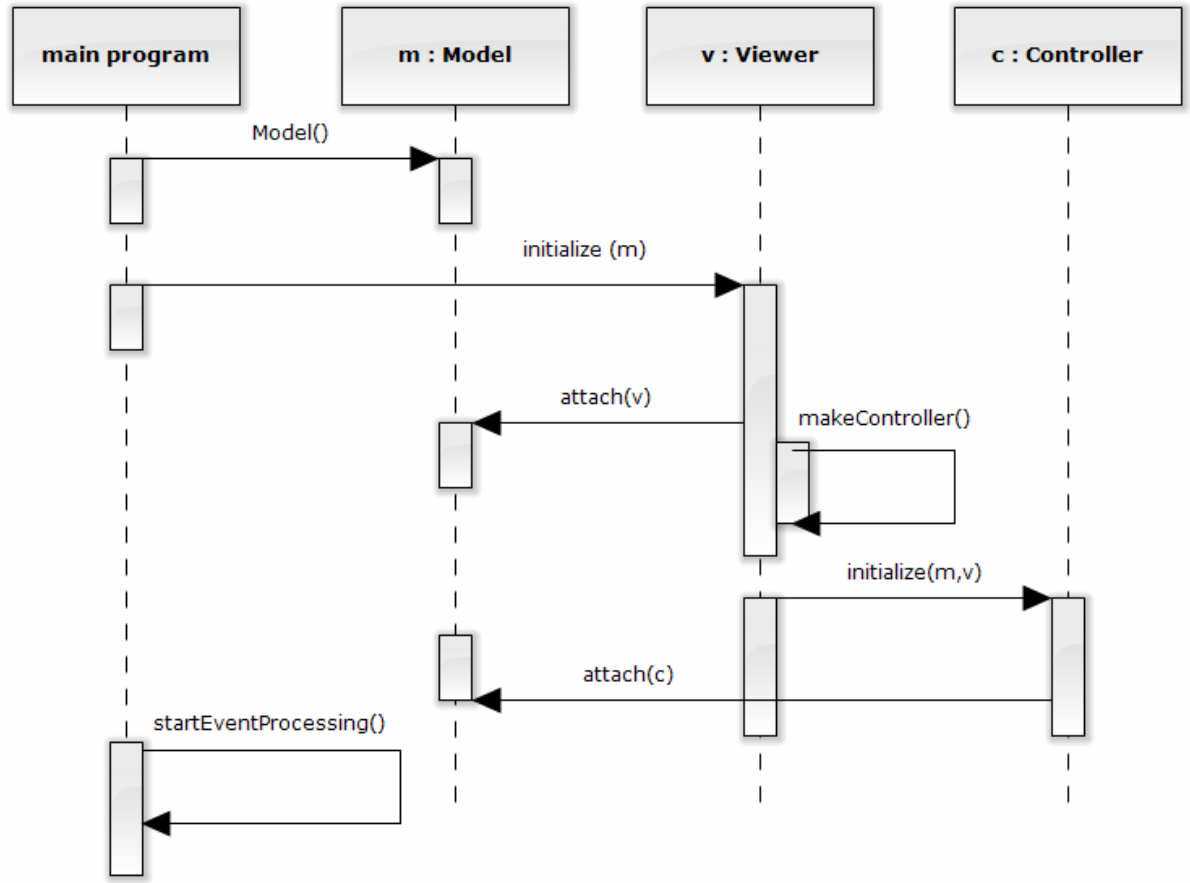


Figura 9.6. Inițializarea

Detalii de Implementare [Shaw93]

Pasul 1. Creează Design-ul pentru componenta Model

- Separă interacțiunea cu utilizatorul de funcționalitatea de bază;

Pasul 2. Implementează mecanismul de propagare a modificărilor

- Pe baza şablonului Observer;

Pasul 3. Creează design-ul şi implementarea Vederilor

- Clasa de bază comună "*View*";
- Redefineşte *initialize()* şi *draw()* în subclase;
- *Draw()* este apelată din *update()*
    - Adu datele din Model (optimizând preluările frecvente: controlerii vederilor nu redesenează/ funcționează la evenimente succesive);
  - *Initialize()*
    - Abonează la mecanismul de propagare a modificărilor;
    - Stabileşte relația cu Controller-ul prin functia *makeController() ;*

Pasul 4. Creează Design-ul şi implementează Controlerii

- intrările utilizatorilor sunt primite ca evenimente cu *handleEvent(Event);*
- dacă nu se procesează evenimente, *handleEvent* este o metodă NOP (no operation);
- cuplarea strânsă între Model şi Controller;
- decuplează folosind şablonul Command (Model este furnizor de comenzi, Controller este Invoker-ul comenzilor);

Pasul 5. Creează Design-ul relației View-Controller (se poate folosi şablonul de proiectare Factory Method).

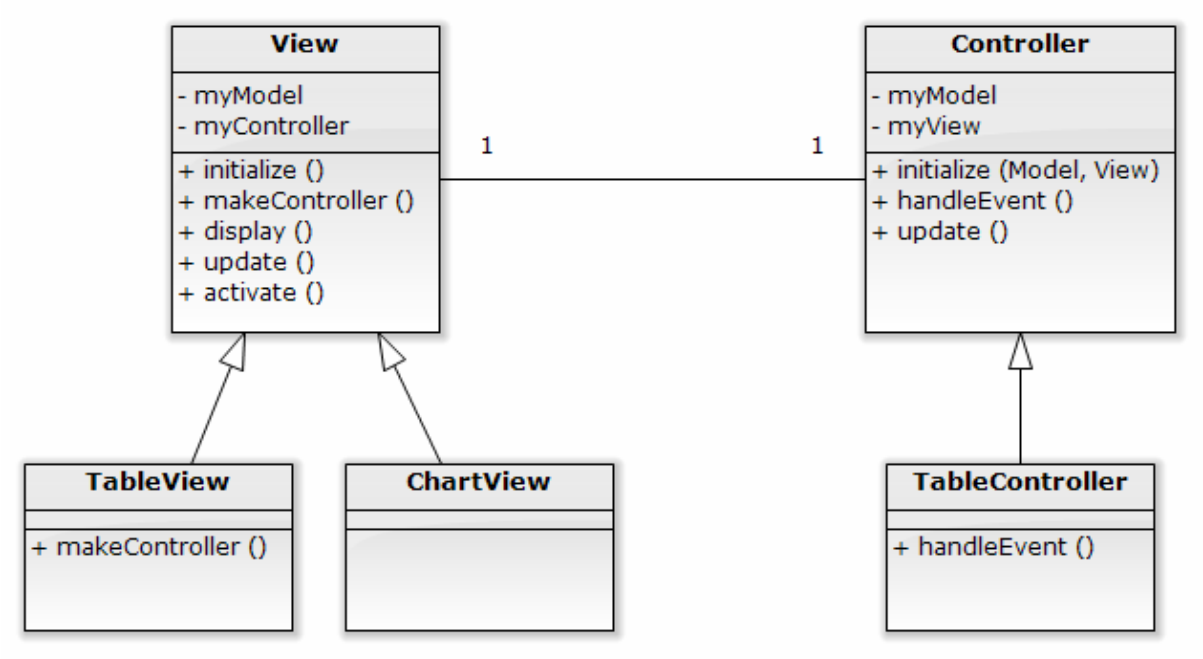


Figura 9.7.

Avantaje

- Vederi multiple (şi sincronizate) asupra aceluiaşi model;
- Vederi şi controleri uşor de conectat;
- "look-and-feel" uşor de schimbat;
- Adaptarea la un context dinamic.

Dezavantaje

- Posibilitatea unor actualizări excesive (există situații în care nu toate vederile necesită actualizări);
- Accesul la date prin intermediul vederii este ineficient (vor fi cerute şi date nemodificate);
- Cuplarea strânsă a View şi Controller de Model (schimbări în Model vor modifica implementarea View şi Controller);
- Complexitate sporită.

Exerciții.

1. Folosiți şablonul Command pentru decuplarea Modelului de View şi de Controller.
2. Folosiți MVC pentru implementarea completă a Exercițiului 1, Secțiunea 7.4.

BIBLIOGRAFIE

[Alex79] Christopher Alexander, *The Timeless Way of Building*, 1979;

[Bass98] Len Bass, Paul Clements & Rick Kazman, *Software Architecture in Practice*, 1998, Addison-Wesley;

[Booc08] Grady Booch, „*Handbook of software architecture*" , 2008;
http://handbookofsoftwarearchitecture.com/

[Busc95] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, *Pattern-Oriented Software Architecture*, Wiley, 1995;

[Chaos14] Chaos Report, http://www.standishgroup.com, 2014;

[Coop98] James W. Cooper , *The Design Patterns Java Companion* 1998 ;
http://www.e-booksdirectory.com/details.php?ebook=2245

[Crai05] Craig Larman, *Applying UML and Patterns*, Prentice Hall, 2005 ;

[Ecke02] Bruce Eckel, *Thinking in Patterns*, 2002;
http://www.mindview.net/Books/ TIPatterns/

[Ecke09]Bruce Eckel, *Thinking in C++* , 2009;
http://mindview.net/Books/TICPP/ThinkingInCPP2e.html

[Gamm94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison- Wesley, 1994;

[Jaco92] Ivar Jacobson, Marcus Christerson, Patrik Jonsson & Gunnar Overgaard 1992. *Object-Oriented Software Engineering: A Use Case Driven Approach (ACM Press)* . Addison-Wesley, 1992;

[Kay03] Alan Kay „*On OO Programming*", 2003;
http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/ doc_kay_oop_en

[Lisk88] Liskov, B., *"Keynote address - data abstraction and hierarchy"*. ACM SIGPLAN Notices 23, 1988;

[Mart02] Robert Martin, *Agile Software Development: Principles, Patterns and Practices*, Prentice Hall, 2002;

[Mart03] Robert Martin, *UML for java programmers*, Prentice Hall, 2003;

[Meye94] Bertrand Meyer, *An Object-Oriented Environment: Principles and Application*, Prentice-Hall, 1994;

[MIT90] MIT Laboratory in software engineering;
http://ocw.mit.edu/6/6.170/f01/lecture-notes/index.html

[Parn72] D.L. Parnas. "*On the Criteria to Be Used in Decomposing Systems into Modules*," Communications of the ACM 15(12), Decembrie 1972;

[Schm02] Douglas C. Schmidt , *Software Design Principles and Guidelines* ;
http://www.cs.wustl.edu/schmidt/

[Shaw93] M.Shaw , D.Garlan, *"An Introduction to Software Architecture,"* Advances in Software Engineering and Knowledge Engineering, World Scientific Publishing Company, New Jersey, 1993;

[Your91] Peter Coad, Edward Yourdon, *Object Oriented Design*, Prentice Hall, 1991.