

Cursuri 7-8

Proiectarea obiectuală: Reutilizare

*Suport de curs bazat pe B. Bruegge and A.H. Dutoit
"Object-Oriented Software Engineering using UML, Patterns, and Java"*

Proiectarea obiectuală

- Activitățile tehnice ale ingineriei software reduc, în manieră graduală, discrepanța problemă - mașină fizică / calculator, prin identificarea / definirea obiectelor care compun viitorul sistem
 - *Analiză*
 - identificarea obiectelor aferente conceptelor din domeniul problemei
 - *Proiectare de sistem*
 - definirea mașinii virtuale, prin selectarea de componente predefinite pentru servicii standard (sisteme de operare, medii de execuție, cadre de aplicație, kituri de instrumente pentru interfața cu utilizatorul, biblioteci de clase de uz general)
 - identificarea unor componente predefinite aferente obiectelor din domeniul problemei (ex. o bibliotecă reutilizabilă de clase reprezentând concepte caracteristice domeniului bancar)
 - *Proiectare obiectuală*
 - identificarea de noi obiecte din domeniul soluției, rafinarea celor existente, adaptarea componentelor reutilizate, specificarea riguroasă a interfețelor subsistemelor și claselor componente

Proiectarea obiectuală (cont.)

- Activități ale proiectării obiectuale

- *Reutilizare*

- reutilizarea unor componente de bibliotecă pentru structuri de date / servicii de bază
 - reutilizarea șabloanelor de proiectare pentru rezolvarea unor probleme recurente și asigurarea modificabilității / extensibilității sistemului
 - adaptarea componentelor reutilizate prin încapsulare / specializare

- *Specificarea interfețelor*

- specificarea completă a interfețelor și claselor ce compun subsistemele

- *Restructurarea modelului obiectual*

- transformarea modelului obiectual în scopul creșterii inteligibilității și mentenabilității acestuia

- *Optimizarea modelului obiectual*

- transformarea modelului obiectual în scopul asigurării criteriilor de performanță (în termeni de timp de execuție / memorie utilizată)

Concepte legate de reutilizare

- Obiecte din domeniul problemei vs. obiecte din domeniul soluției
- Moștenirea specificării vs. moștenirea implementării
- Delegare
- Principiul Liskov al substituției
- Principiile SOLID de proiectare a claselor

Obiecte din domeniul problemei / soluției

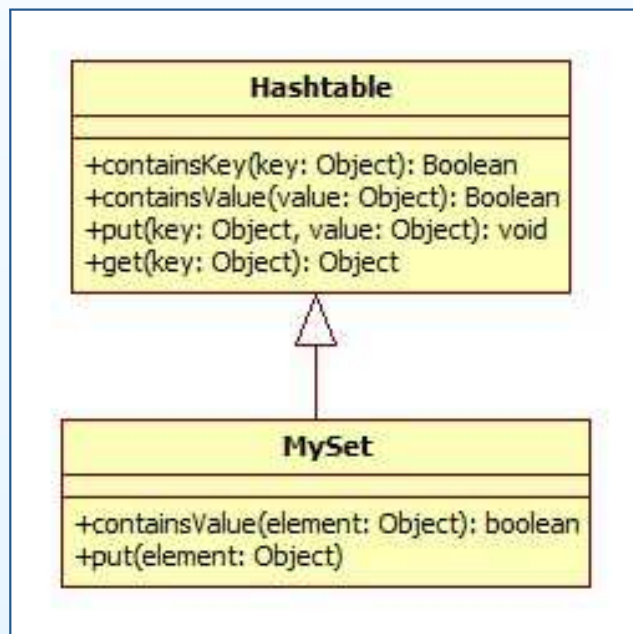
- Diagramele UML de clase pot fi utilizate pentru modelarea atât a domeniului problemei, cât și a domeniului soluției
- Dualitatea *domeniul problemei / domeniul soluției*
 - *Obiectele din domeniul problemei* (eng. *domain objects, application objects*) corespund unor concepte ale domeniului relevante pentru sistemul în cauză
 - *Obiectele din domeniul soluției* (eng. *solution objects*) corespund unor concepte fără corespondent la nivelul domeniului problemei (ex. depozite de date persistente, obiecte ale interfeței grafice utilizator)
- Etapele identificării obiectelor din domeniul problemei / soluției
 - *În analiză* - se identifică obiectele din domeniul problemei + acele obiecte din domeniul soluției ce sunt vizibile utilizatorului (obiecte *boundary* și *control* ce corespund interfețelor și tranzacțiilor definite de sistem)
 - *În proiectarea de sistem* - se identifică obiecte din domeniul soluției în termenii platformei hardware/ software
 - *În proiectarea obiectuală* - se rafinează și detaliază obiectele din domeniul problemei / soluției identificate anterior și se identifică restul obiectelor din domeniul soluției

Moștenirea specificării vs. moștenirea implementării

- În etapa de analiză, moștenirea este utilizată pentru clasificarea obiectelor în taxonomii
 - Rolul generalizării (identificarea unei superclase comune pentru un număr de clase existente) și al specializării (identificarea unor subclase ale unei clase existente) este acela de organiza obiectele din domeniul problemei într-o ierarhie ușor de înțeles și de a diferenția comportamentul comun unei mulțimi de obiecte (expus de clasa de bază sau superclasă) de comportamentul specific caracteristic claselor derivate (sau subclaselor)
- În etapa de proiectare obiectuală, rolul moștenirii este acela de a elimina redundanțele din modelul obiectual și de a asigura modificabilitatea/extensibilitatea sistemului
 - Factorizarea comportamentului comun la nivelul clasei de bază reduce riscul apariției unor inconsistențe în eventualitatea efectuării unor modificări la nivelul implementării respectivului comportament, prin localizarea modificărilor într-o singură clasă
 - Definirea de clase abstracte / interfețe asigură extensibilitatea, permițând specializarea comportamentului prin definirea de noi subclase, conforme interfețelor abstracte

Moștenirea specificării / implementării (cont.)

- *Ex.:* Se dorește implementarea unei clase *MySet*, reutilizând funcționalitatea oferită de clasa existentă *Hashtable*
- *Soluția 1:* Clasa *MySet* e o specializare a lui *Hashtable*



```
public class MySet extends Hashtable {

    public MySet() {}

    public void put(Object element)
    {
        if (!containsKey(element))
            put(element, this);
    }

    @Override
    public boolean containsValue(Object element)
    {
        return containsKey(element);
    }

    /* ... */
}
```

Moștenirea specificării / implementării (cont.)

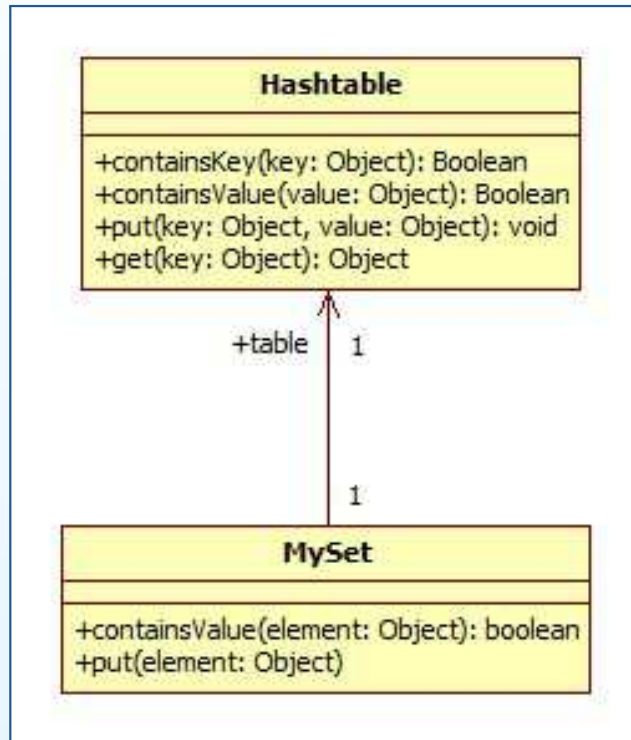
- *Avantaje*
 - Funcționalitatea dorită
 - Reutilizarea codului
- *Probleme*
 - Clasa *MySet* moștenește *containsKey()* și suprascrie *containsValue()*, oferind același comportament - contraintuitiv
 - Clienții *MySet* pot utiliza *containsKey()*, ceea ce face dificilă modificarea reprezentării *MySet*
 - Substituirea unui obiect *Hashtable* cu unul *MySet* conduce la un comportament *containsValue()* nedorit
- *Moștenirea implementării* (eng. *implementation inheritance*) = utilizarea moștenirii având drept unic scop reutilizarea codului
- *Moștenirea specificării* sau *moștenirea interfeței* (eng. *specification inheritance, interface inheritance*) = clasificarea conceptelor în ierarhii

Delegare

- *Delegarea* reprezintă o alternativă la moștenirea implementării, atunci când se dorește implementarea unei funcționalități prin reutilizare
- Se spune că o clasă *delegă* unei alte clase în cazul în care implementează o operație retrimițând mesajul aferent către cea din urmă
- Implementarea clasei *MySet* prin delegare către *Hashtable* rezolvă problemele menționate anterior
 - *Modificabilitate* - Reprezentarea internă a clasei *MySet* poate fi schimbată fără a-i afecta clienții
 - *Subtipizare* - un obiect *MySet* nu poate fi substituit unui obiect *Hashtable* => codul client care utilizează *Hashtable* nu va fi afectat
- Delegarea este de preferat moștenirii implementării, din rațiuni de modificabilitate și neafectare a codului existent; moștenirea specificării este de preferat delegării, în situații de subtipizare, din rațiuni de extensibilitate

Delegare (cont.)

- *Soluția 2: Clasa MySet delegă către Hashtable*



```
public class MySet {

    private Hashtable table;

    public MySet()
    {
        table = new Hashtable();
    }

    public void put(Object element)
    {
        if (!table.containsKey(element))
            table.put(element, this);
    }

    public boolean containsValue(Object element)
    {
        return table.containsKey(element);
    }

    /* ... */
}
```

Principiul Liskov al substituției

- Oferă o definiție formală conceptului de *moștenire a specificării*
- (B. Liskov, 1988) Dacă pentru fiecare obiect $o1$ de tipul S există un obiect $o2$ de tipul T , astfel încât orice program P definit în termenii lui T își păstrează comportamentul atunci când $o2$ este substituit cu $o1$, atunci S este un subtip al lui T .
- \Leftrightarrow (B. Bruegge) Dacă un obiect de tip S poate fi utilizat oriunde este așteptat un obiect de tip T , atunci S este un subtip al lui T
- \Leftrightarrow (R. Martin) Subtipurile trebuie să poată substitui tipurile lor de bază (eng. *Subtypes must be substitutable for their base types*)



Principiile SOLID de proiectare a claselor

- **SOLID =**
 - **S**ingle Responsibility Principle (Principiul responsabilității unice) +
 - **O**pen Closed Principle (Principiul deschis/închis) +
 - **L**iskov Substitution Principle (Principiul Liskov al substituției) +
 - **I**nterface Segregation Principle (Principiul separării interfețelor) +
 - **D**ependency Inversion Principle (Principiul inversării dependențelor)



Principiul responsabilității unice

- Nu trebuie să existe niciodată mai mult de un motiv pentru ca o clasă să sufere modificări (eng. *There should never be more than one reason for a class to change*)
- \Leftrightarrow Nu trebuie să existe mai mult de o responsabilitate per clasă



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Principiul deschis/închis

- Entitățile software (clase, module, funcții) trebuie să fie deschise pentru extindere, dar închise pentru modificare (eng. *Software entities (classes, modules, functions) should be open for extension, but closed for modification*)



Principiul separării interfețelor

- Clienții nu trebuie constrânși să depindă de interfețe pe care nu le utilizează (eng. *Clients should not be forced to depend upon interfaces that they do not use*)



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

Principiul inversării dependențelor

- Modulele de nivel înalt nu ar trebui să depindă de module de nivel jos, ambele ar trebui să depindă de abstractizări (eng *High level modules should not depend upon low level modules, both should depend upon abstractions*)
- Abstractizările nu ar trebui să depindă de detalii, detaliile ar trebui să depindă de abstractizări (eng. *Abstractions should not depend upon details, details should depend upon abstractions*)



Șabloane de proiectare [Gamma et al., 1994]

- Proiectarea de sistem vs. proiectarea obiectuală - paradox generat de obiective conflictuale
 - Obiectivul proiectării de sistem: gestionarea complexității prin crearea unei arhitecturi stabile, descompunând sistemul în subsisteme cu interfețe bine stabilite și slab cuplate => un anumit grad de rigiditate
 - Obiectiv al proiectării obiectuale: flexibilitate - proiectarea unui soft modificabil și extensibil, în vederea minimizării costurilor unor viitoare schimbări în sistem
- Soluție: anticiparea schimbării și proiectarea pentru schimbare (eng. *anticipate change and design for change*)
- Surse comune ale schimbărilor în sistemele soft
 - Furnizor nou / tehnologie nouă
 - unele dintre componentele achiziționate pentru reutilizare în cadrul sistemului sunt înlocuite cu versiuni oferite de alți furnizori
 - Implementare nouă
 - după integrare, unele structuri de date / unii algoritmi sunt înlocuiți cu versiuni mai eficiente, pentru satisfacerea constrângerilor legate de performanță

Șabloane de proiectare (cont.)

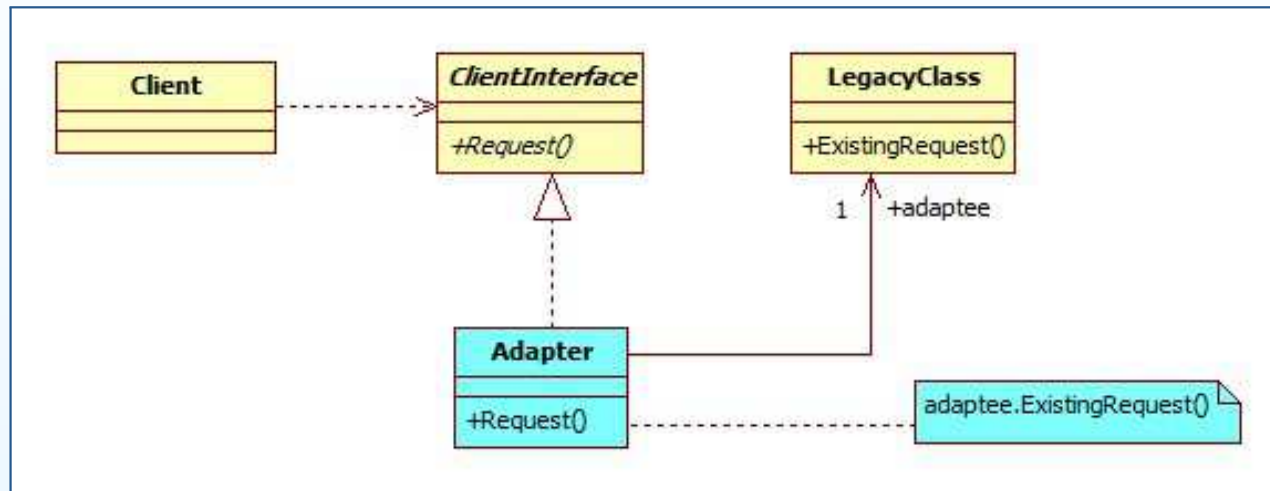
- Funcționalitate nouă
 - testele de validare pot dezvălui absența unor funcționalități din sistem
- O nouă complexitate a domeniului aplicației
 - identificarea unor generalizări posibile ale sistemului în cadrul domeniului de aplicație sau modificarea regulilor de business ale domeniului
- Erori la nivelul cerințelor sau erori de proiectare/implementare
- Șabloane de proiectare adecvate acestor tipuri de schimbări
 - Furnizor nou / tehnologie nouă / implementare nouă
 - *Adapter*
 - *Bridge*
 - *Strategy*
 - Furnizor nou / tehnologie nouă
 - *Abstract Factory*
 - Funcționalitate nouă
 - *Command*
 - O nouă complexitate a domeniului aplicației
 - *Composite*

Încapsularea componentelor existente cu *Adapter*

- Creșterea complexității sistemelor cerute și scurtarea termenelor de realizare conduce la necesitatea reutilizării unor componente existente, fie din sisteme mai vechi (eng. *legacy systems*), fie achiziționate de la terți
 - În ambele cazuri, este vorba de cod care nu a fost scris pentru sistemul în cauză și care, în general, nu poate fi modificat
 - Arhitectura sistemului curent considerându-se a fi stabilă, nici interfețele acestuia nu ar trebui modificate
 - Gestionare unor astfel de componente presupune încapsularea lor, folosind șablonul *Adapter*
- Șablonul *Adapter*
 - *Scop*: Transformă interfața unei clase existente (eng. *legacy class*) într-o altă interfață care este așteptată de client, astfel încât clientul și clasa să poată colabora fără nici o modificare la nivelul acestora.
 - *Soluție*: O clasă *Adapter* implementează interfața *ClientInterface* așteptată de client. Obiectele *Adapter* delegă cererile primite de la client obiectelor *LegacyClass* și efectuează conversiile necesare.

Șablonul *Adapter* (cont.)

◦ *Structură:*



◦ *Consecințe:*

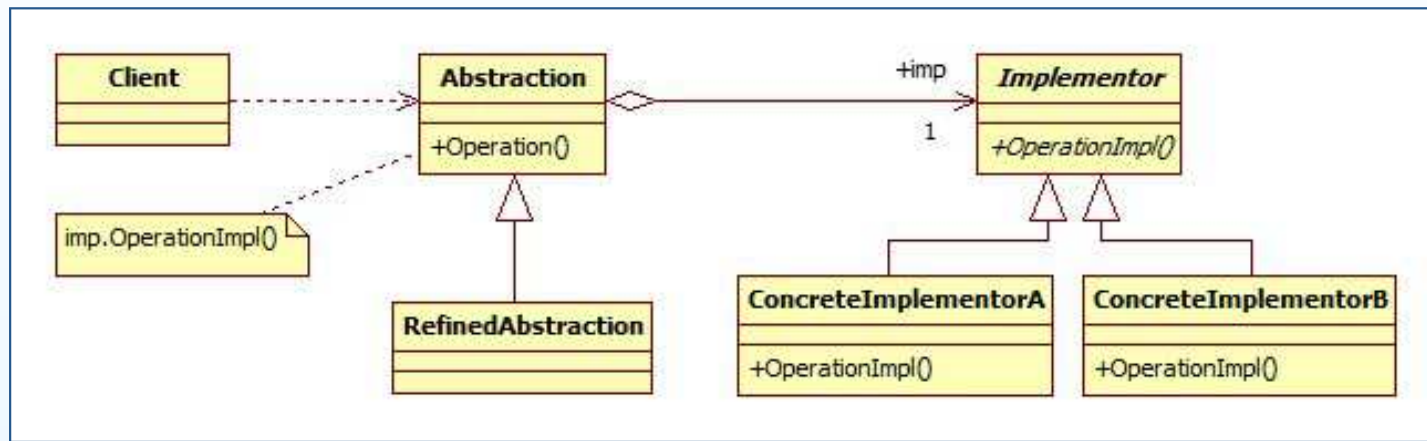
- Clasele *Client* și *LegacyClass* pot colabora fără nici un fel de modificare la nivelul acestora
- *Adapter* lucrează cu *LegacyClass* și oricare dintre subclasele acesteia
- Pentru fiecare specializare a *ClientInterface* trebuie scris un nou *Adapter*

Decuplarea abstractizărilor de implementări cu *Bridge*

- Considerăm problema dezvoltării, integrării (din subsistemele componente) și testării unui sistem
 - Subsistemele pot fi finalizate la momente de timp diferite; pentru a nu întârzia integrarea și testarea sistemului, subsistemele nefinalizate pot fi înlocuite pe moment cu implementări *stub*
 - Pot exista implementări diferite ale aceluiași subsistem (ex.: o implementare de referință, care realizează funcționalitatea dorită folosind un algoritm simplu și o implementare mai eficientă, dar cu un grad sporit de complexitate)
 - Este necesară o soluție care să permită substituirea dinamică a implementărilor posibile ale unei aceleiași interfețe, în funcție de necesități
- Șablonul *Bridge*
 - *Scop*: Decuplează o abstractizare de implementarea ei, astfel încât cele două să poată varia independent. Implementările posibile vor putea fi astfel substituite la execuție.

Șablonul *Bridge* (cont.)

- *Structură:*



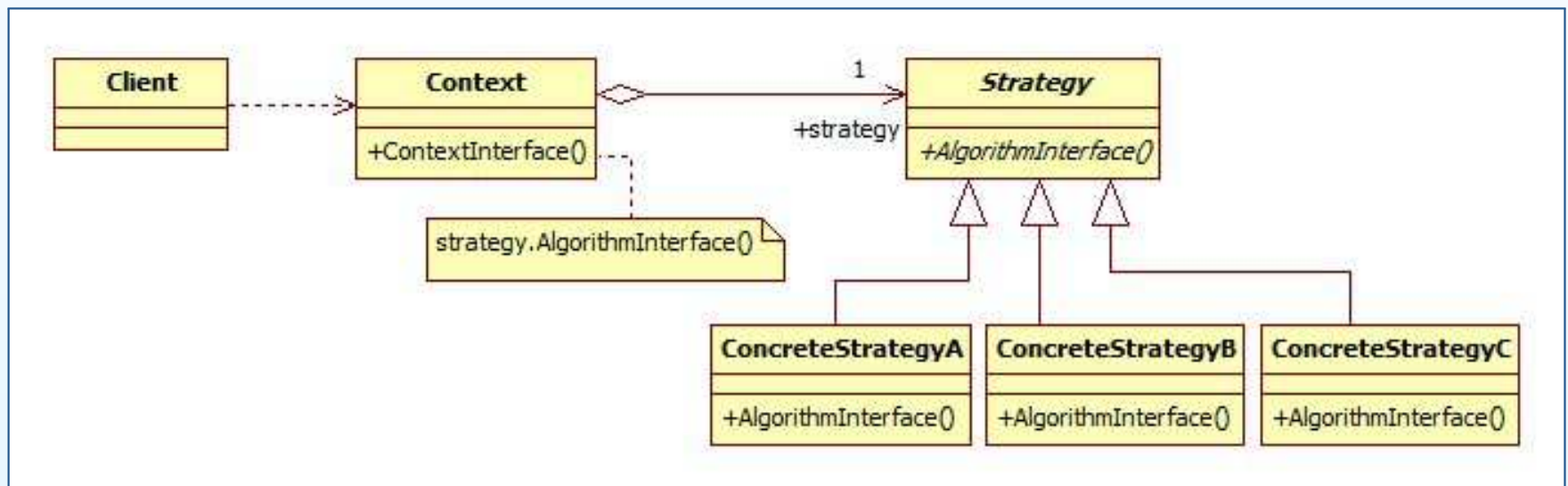
- *Soluție:* Clasa *Abstraction* definește interfața vizibilă clientului. *Implementor* este o clasă abstractă, care declară metode de nivel jos disponibile clasei *Abstraction*. O instanță *Abstraction* reține o referință către instanța *Implementor* curentă. Clasele *Abstraction* și *Implementor* pot fi rafinate independent.
- *Consecințe:*
 - Clientul este protejat de implementările abstracte și concrete
 - Abstractizarea și implementările pot fi rafinate independent

Încapsularea algoritmilor cu *Strategy*

- Presupunem existența unui număr de algoritmi diferiți pentru rezolvarea aceleiași probleme și necesitatea de a interschimba în mod dinamic algoritmul utilizat
 - Ex.:
 - Un editor de documente poate folosi algoritmi diferiți pentru împărțirea textului pe rânduri, funcție de preferințe (rânduri de lungime fixă, împărțire în silabe sau nu, etc.)
 - într-un joc de șah, calculatorul poate folosi strategii diferite de alegere a următoarei mutări, funcție de nivelul selectat (începător, expert, etc.)
 - Abordări posibile
 - Includerea tuturor algoritmilor la nivelul clasei care îi utilizează și folosirea de instrucțiuni condiționale pentru a-i interschimba => algoritmii pot fi interschimbați dinamic, însă clasa respectivă va avea o complexitate sporită și va fi incomod de modificat în condițiile adăugării unui algoritm nou
 - Clasa care utilizează algoritmii implementează o versiune (varianta default), iar pentru celelalte versiuni se definesc clase derivate aferente, care suprascriu doar algoritmul din clasa de bază => un număr mare de clase înrudite care diferă doar prin comportarea aferentă aceluiași algoritm, iar algoritmul nu va putea fi variat dinamic

Șablonul *Strategy* (cont.)

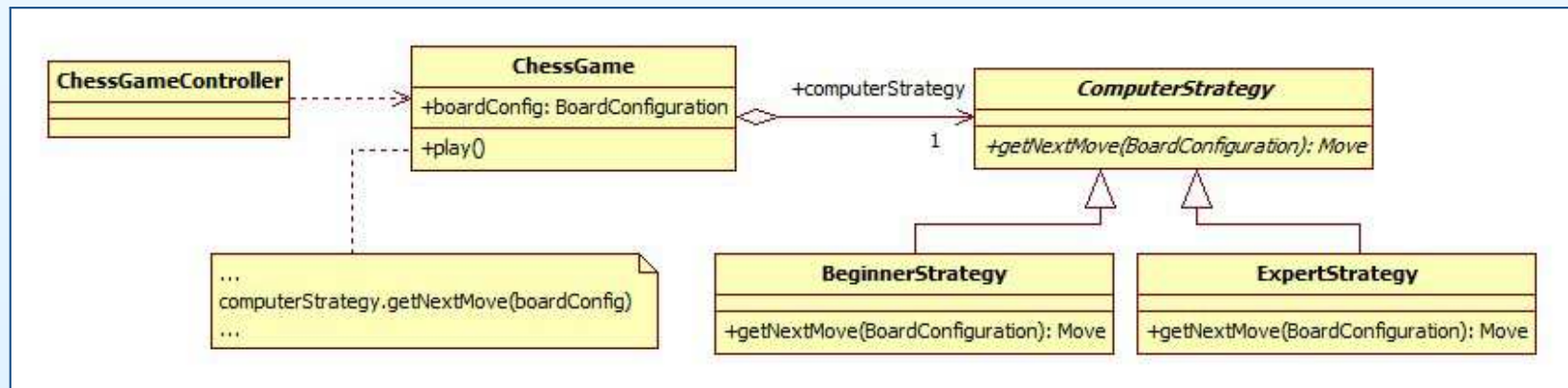
- *Scop*: Definește o familie de algoritmi, încapsulează fiecare algoritm și îi face interschimbabili. Permite algoritmului să varieze independent de clienții care îl utilizează.
- *Structură*:



- *Soluție*:
 - Fiecare dintre algoritmi este încapsulat/implementat de o clasă *ConcreteStrategy*. *Strategy* definește interfața comună a tuturor algoritmilor suportați.

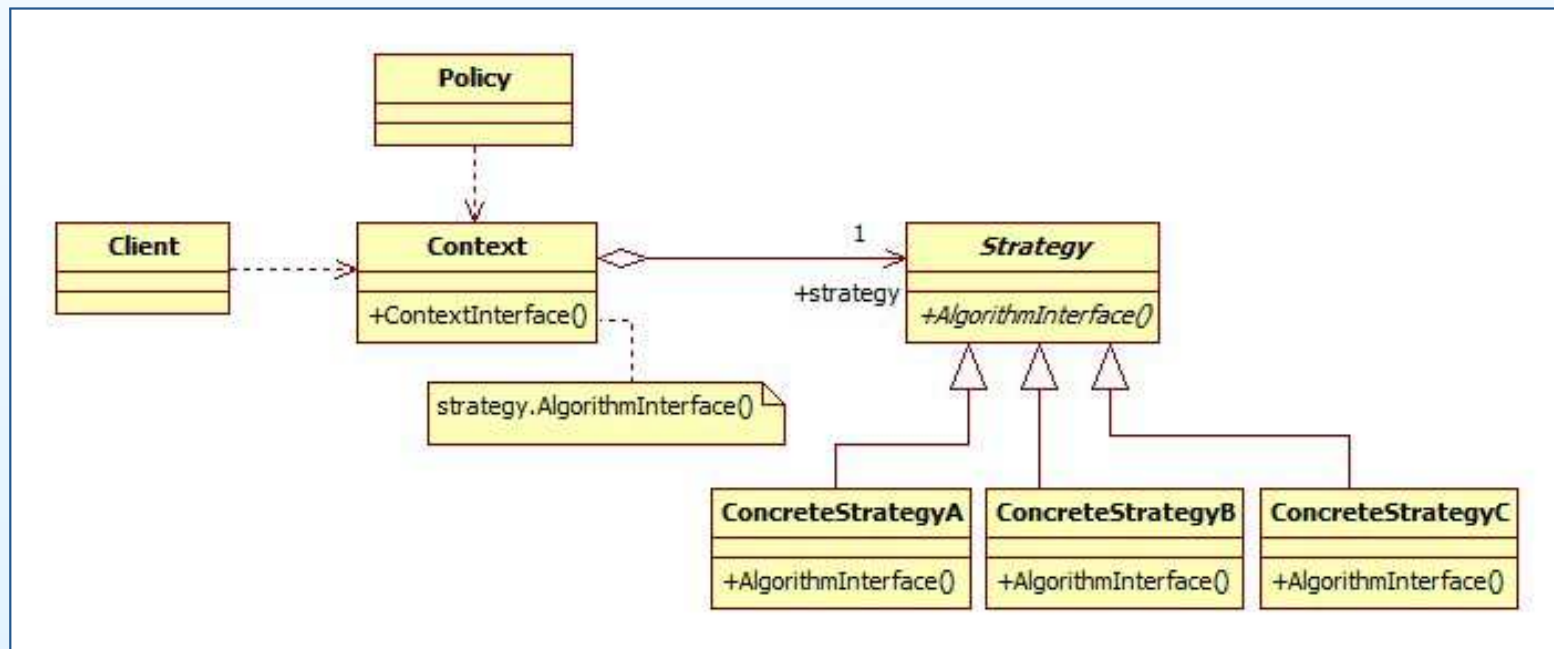
Șablonul *Strategy* (cont.)

- Obiectele *Context* utilizează această interfață pentru a apela algoritmul definit de o clasă *ConcreteStrategy*. Contextul păstrează o referință către un obiect *Strategy* și îi delegă acestuia responsabilitatea execuției algoritmului, atunci când este cazul.
 - Obiectul strategie este creat și plasat în clasa *Context* de către *Client*.
 - Contextul poate trece către strategie toate date necesare algoritmului, atunci când acesta este apelat. Ca și alternativă, contextul se poate trece pe sine ca și argument în operațiile interfeței *Strategy* și poate oferi o interfață care să-i permită obiectului *Strategy* să-i acceseze datele.
- Ex.: Instanțierea șablonului *Strategy* pentru un joc de sah cu calculatorul



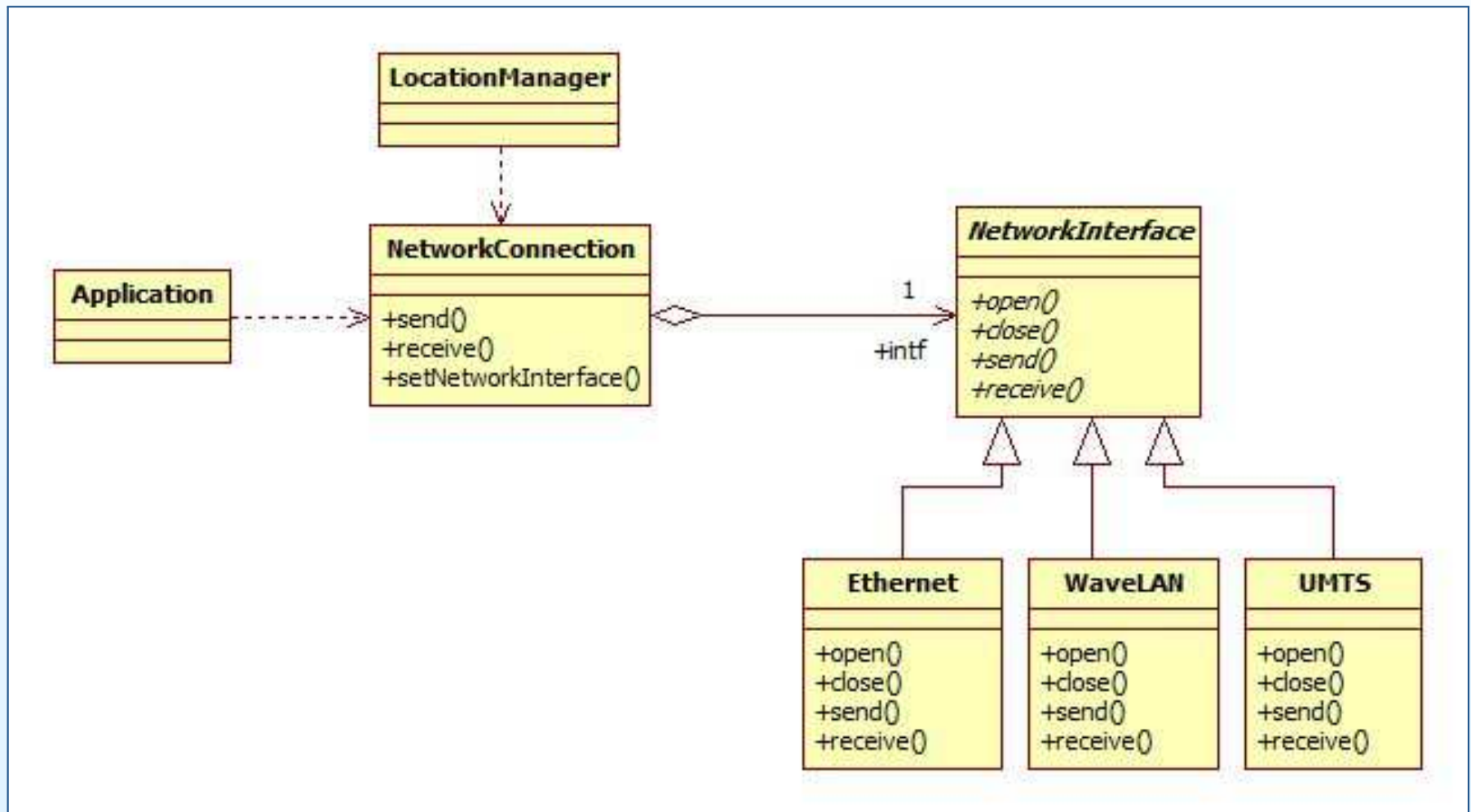
Șablonul *Strategy* (cont.)

- *Consecințe*
 - Strategiile concrete pot fi substituite în mod transparent relativ la context
 - Pt fi adăugați algoritmi noi fără a modifica contextul sau clientul
- *Variație*
 - Responsabilitatea configurării contextului cu o strategie concretă este atribuită unei clase specializate *Policy*



Șablonul *Strategy* (cont.)

- Ex.: Schimbarea rețelei în aplicații pentru dispozitive mobile (instanțiere a variației șablonului *Strategy*)



Şablonul *Strategy* (cont.)

```
/** The NetworkConnection object represents a single abstract connection
 * used by the Client. This is the Context object in Strategy pattern. */
public class NetworkConnection {
    private String destination;
    private NetworkInterface intf;
    private StringBuffer queue;

    public NetworkConnect(String destination, NetworkInterface intf) {
        this.destination = destination;
        this.intf = intf;
        this.intf.open(destination);
        this.queue = new StringBuffer();
    }

    public void send(byte msg[]) {
        // queue the message to be send in case the network is not ready.
        queue.concat(msg);
        if (intf.isReady()) {
            intf.send(queue);
            queue.setLength(0);
        }
    }

    public byte [] receive() {
        return intf.receive();
    }

    public void setNetworkInterface(NetworkInterface newIntf) {
        intf.close();
        newIntf.open(destination);
        intf = newIntf;
    }
}
```

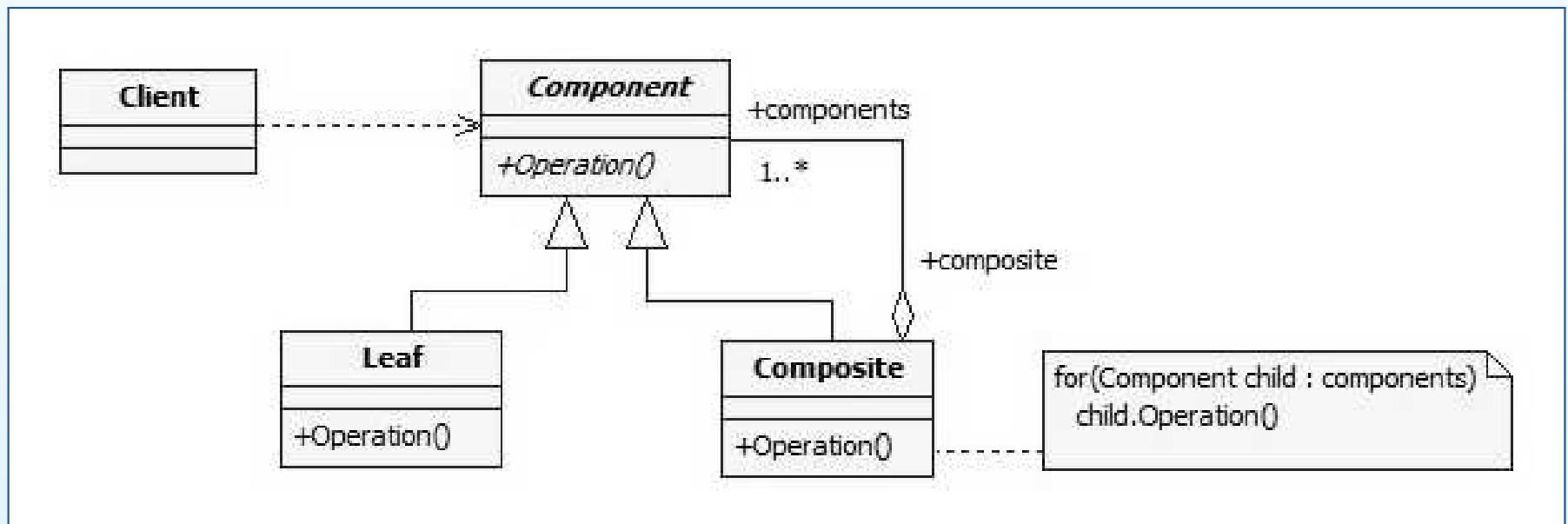
Şablonul *Strategy* (cont.)

```
/** The LocationManager decides on which NetworkInterface to use based on
 * availability and cost. */
public class LocationManager {
    private NetworkInterface networkIntf;
    private NetworkConnection networkConn;
    /* ... */

    // This method is invoked by the event handler when the location
    // may have changed
    public void doLocation() {
        if (isEthernetAvailable()) {
            networkIntf = new EthernetNetwork();
        } else if (isWaveLANAvailable()) {
            networkIntf = new WaveLANNetwork();
        } else if (isUMTSAvailable()) {
            networkIntf = new UMTSNetwork();
        } else {
            networkIntf = new QueueNetwork();
        }
        networkConn.setNetworkInterface(networkIntf);
    }
}
```

Încapsularea ierarhiilor cu *Composite*

- Șablonul *Composite*
 - *Tip*: șablon structural
 - *Scop*: Permite reprezentarea unor ierarhii de lățime și adâncime variabilă (recursive), astfel încât frunzele și agregatele să fie tratate uniform, prin intermediul unei interfețe comune.
 - *Structură*:



Șablonul *Composite* (cont.)

- *Soluție*: Interfața *Component* specifică serviciile partajate de *Leaf* și *Composite* (ex. *move(x,y)*, pentru un obiect grafic). Clasa *Composite* agregă obiecte *Component* și implementează aceste servicii iterând peste componentele conținute și delegându-le serviciul în cauză (ex. *move(x,y)* din *Composite* invocă iterativ *move(x,y)* pentru fiecare obiect *Component* conținut). Funcționalitatea concretă este asigurată de implementările serviciilor din *Leaf* (implementarea *move(x,y)* din *Leaf* modifică coordonatele unei primitive grafice și o redesenează).
- *Exemple*:
 - Ierarhii de componente grafice: componentele grafice pot fi organizate în containere, ce pot fi scalate și re poziționate uniform. Un container poate conține alte containere
 - Ierarhii de fișiere și directoare: directoarele pot conține fișiere și alte directoare. Aceleași operații sunt folosite pentru copierea/ ștergerea amândurora
 - Descompunerea unui sistem: un subsistem constă din clase și alte subsisteme

Șablonul *Composite* (cont.)

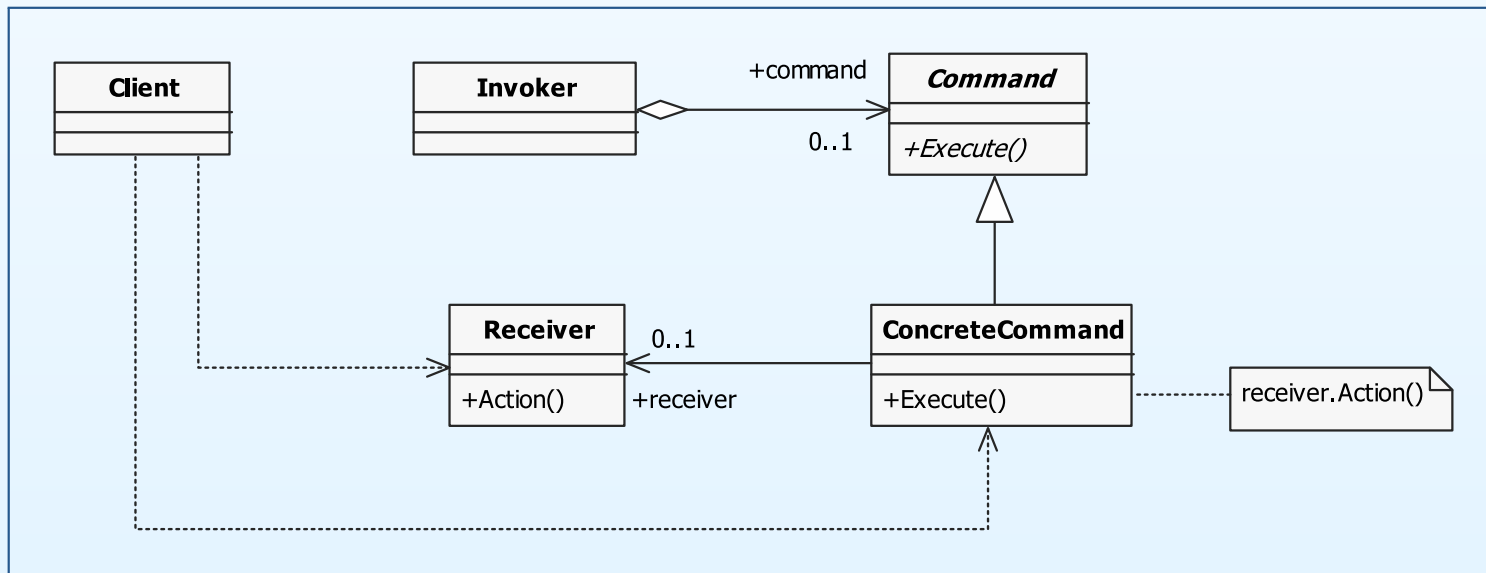
- *Consecințe:*

- Un client utilizează același cod pentru a lucra cu obiecte *Leaf* și *Composite*
- Noi clase *Leaf* pot fi adăugate fără a schimba ierarhia
- Comportamentele specifice *Leaf* pot fi modificate fără a schimba ierarhia

Încapsularea fluxului de control cu *Command*

- Șablonul *Command*

- *Tip*: șablon comportamental
- *Scop*: Încapsulează o cerere ca și un obiect, permițând parametrizarea clienților cu diferite cereri, precum și formarea unei cozi sau a unui registru de cereri și asigurarea suportului pentru operațiile ce pot fi anulate (*facilități undo*).
- *Structură*:



Șablonul *Command* (cont.)

- *Soluție*: O clasă abstractă *Command* declară interfața suportată de clasele *ConcreteCommand*. O clasă *ConcreteCommand* încapsulează un serviciu ce poate fi aplicat unui *Receiver*. Clasa *Client* creează obiecte *ConcreteCommand* și le asociază obiectelor *Receiver* adecvate. Un obiect *Invoker* execută sau anulează (*undo*) o comandă.
- *Consecințe*:
 - Comanda decuplează obiectul care invocă operația, de cel care știe cum să o efectueze.
 - Comenzile sunt obiecte de prima clasă. Ele pot fi manipulate și extinse, la fel ca și oricare alt obiect.
 - Comenzile pot fi asamblate într-o comandă compusă (instanță a șablonului *Composite*).

Șabloanele *State* și *Singleton*

- Vezi Seminar 7 și [Gamma et al., 1994]

Referințe

- [Bruegge, 2010] Berndt Bruegge and Allen H. Dutoit, *Object-Oriented Software Engineering Using UML, Patterns and Java*, Prentice Hall, 2010.
- [Gamma et al., 1994] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, Reading, MA, 1994.