Ingineria sistemelor soft

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

- Activităţile tehnice ale ingineriei software reduc, în manieră graduală, discrepanţa problemă - maşină fizică / calculator, prin identificarea / definirea obiectelor care compun viitorul sistem
  - *Analiză*
    - identificarea obiectelor aferente conceptelor din domeniul problemei
  - *Proiectare de sistem*
    - definirea maşinii virtuale, prin selectarea de componente predefinite pentru servicii standard (sisteme de operare, medii de execuţie, cadre de aplicaţie, kituri de instrumente pentru interfaţa 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 soluţiei, rafinarea celor existente, adaptarea componentelor reutilizate, specificarea riguroasă a interfeţelor subsistemelor şi claselor componente

Proiectarea obiectuală (cont.)

- Activităţi ale proiectării obiectuale
  - *Reutilizare*
    - reutilizarea unor componente de bibliotecă pentru structuri de date / servicii de bază
    - reutilizarea şabloanelor de proiectare pentru rezolvarea unor probleme recurente şi asigurarea modificabilităţii / extensibilităţii sistemului
    - adaptarea componentelor reutilizate prin încapsulare / specializare
  - *Specificarea interfeţelor*
    - specificarea completă a interfeţelor şi claselor ce compun subsistemele
  - *Restructurarea modelului obiectual*
    - transformarea modelului obiectual în scopul creşterii inteligibilităţii şi mentenabilităţii acestuia
  - *Optimizarea modelului obiectual*
    - transformarea modelului obiectual în scopul asigurării criteriilor de performanţă (în termeni de timp de execuţie / memorie utilizată)

Concepte legate de reutilizare

- Obiecte din domeniul problemei vs. obiecte din domeniul soluţiei
- Moştenirea specificării vs. moştenirea implementării
- Delegare
- Principiul Liskov al substituţiei
- Principiile SOLID de proiectare a claselor

Obiecte din domeniul problemei / soluţiei

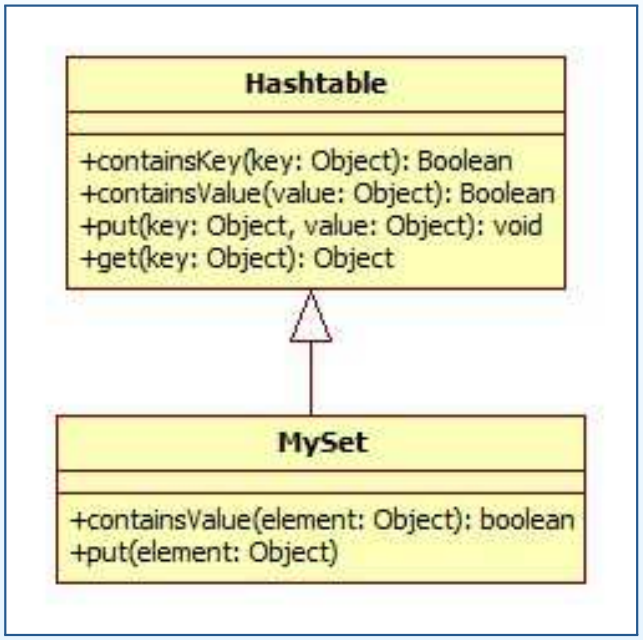
- Diagramele UML de clase pot fi utilizate pentru modelarea atât a domeniului problemei, cât şi a domeniului soluţiei
- Dualitatea *domeniul problemei* / *domeniul soluţiei*
  - *Obiectele din domeniul problemei* (eng. *domain objects*, *application objects*) corespund unor concepte ale domeniului relevante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

Moştenirea specificării vs. moştenirea implementării

- În etapa de analiză, moştenirea este utilizată pentru clasificarea obiectelor în taxonomii
  - Rolul generalizării (identificarea unei superclase comune pentru un număr de clase existente) şi al specializării (identificarea unor subclase ale unei clase existente) este acela de organiza obiectele din domeniul problemei într-o ierarhie uşor de înţeles şi de a diferenţia comportamentul comun unei muţimi de obiecte (expus de clasa de bază sau superclasă) de comportamentul specific caracteristic claselor derivate (sau subclaselor)
- În etapa de proiectare obiectuală, rolul moştenirii este acela de a elimina redundanţele din modelul obiectual şi de a asigura modificabilitatea/extensibilitatea sistemului
  - Factorizarea comportamentului comun la nivelul clasei de bază reduce riscul apariţiei unor inconsistenţe în eventualitatea efectuării unor modificări la nivelul implementării respectivului comportament, prin localizarea modificărilor într-o singură clasă
  - Definirea de clase abstracte / interfeţe asigură extensibilitatea, permiţând specializarea comportamentului prin definirea de noi subclase, conforme interfeţelor abstracte

Moştenirea specificării / implementării (cont.)

- *Ex.*: Se doreşte implementarea unei clase *MySet*, reutilizând funcţionalitatea oferită de clasa existentă *Hashtable*
- *Soluţia 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);

    }

    /* ... */
}
```

Moştenirea specificării / implementării (cont.)

- *Avantaje*
  - Funcţionalitatea dorită
  - Reutilizarea codului
- *Probleme*
  - Clasa *MySet* moşteneşte *containsKey()* şi suprascrie *containsValue()*, oferind acelaşi comportament - contraintuitiv
  - Clienţii *MySet* pot utiliza *containsKey()*, ceea ce face dificilă modificarea reprezentării *MySet*
  - Substituirea unui obiect *Hashtable* cu unul *MySet* conduce la un comportament *containsValue()* nedorit

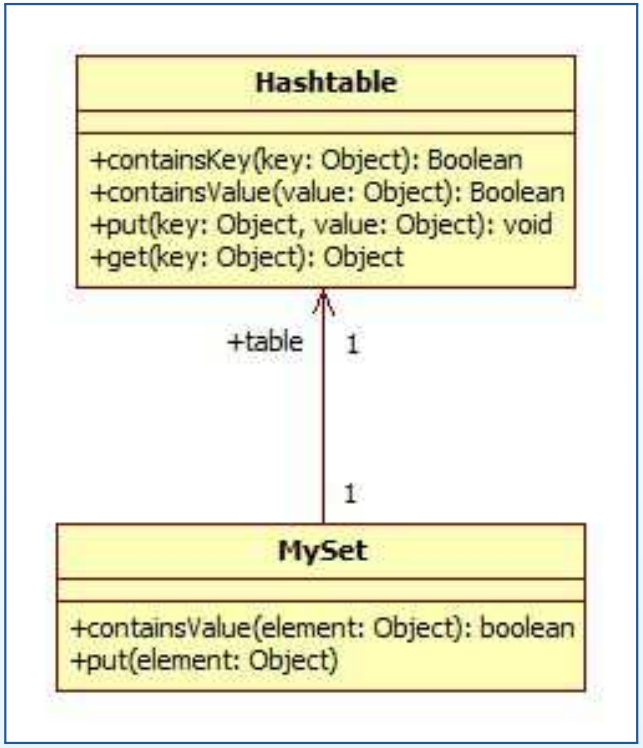
- *Moştenirea implementării* (eng. *implementation inheritance*) = utilizarea moştenirii având drept unic scop reutilizarea codului
- *Moştenirea specificării* sau *moştenirea interfeţei* (eng. *specification inheritance*, *interface inheritance*) = clasificarea conceptelor în ierarhii

Delegare

- *Delegarea* reprezintă o alternativă la moştenirea implementării, atunci când se doreşte implementarea unei funcţionalităţi prin reutilizare
- Se spune că o clasă *delegă* unei alte clase în cazul în care implementează o operaţie retrimiţând mesajul aferent către cea din urmă
- Implementarea clasei *MySet* prin delegare către *Hashtable* rezolvă problemele menţionate anterior
  - *Modificabilitate* - Reprezentarea internă a clasei *MySet* poate fi schimbată fără a-i afecta clienţii
  - *Subtipizare* - un obiect *MySet* nu poate fi substituit unui obiect *Hashtable* => codul client care utilizează *Hashtable* nu va fi afectat
- Delegarea este de preferat moştenirii implementării, din raţiuni de modificabilitate şi neafectare a codului existent; moştenirea specificării este de preferat delegării, în situaţii de subtipizare, din raţiuni de extensibilitate

Delegare (cont.)

- *Soluţia 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 substituţiei

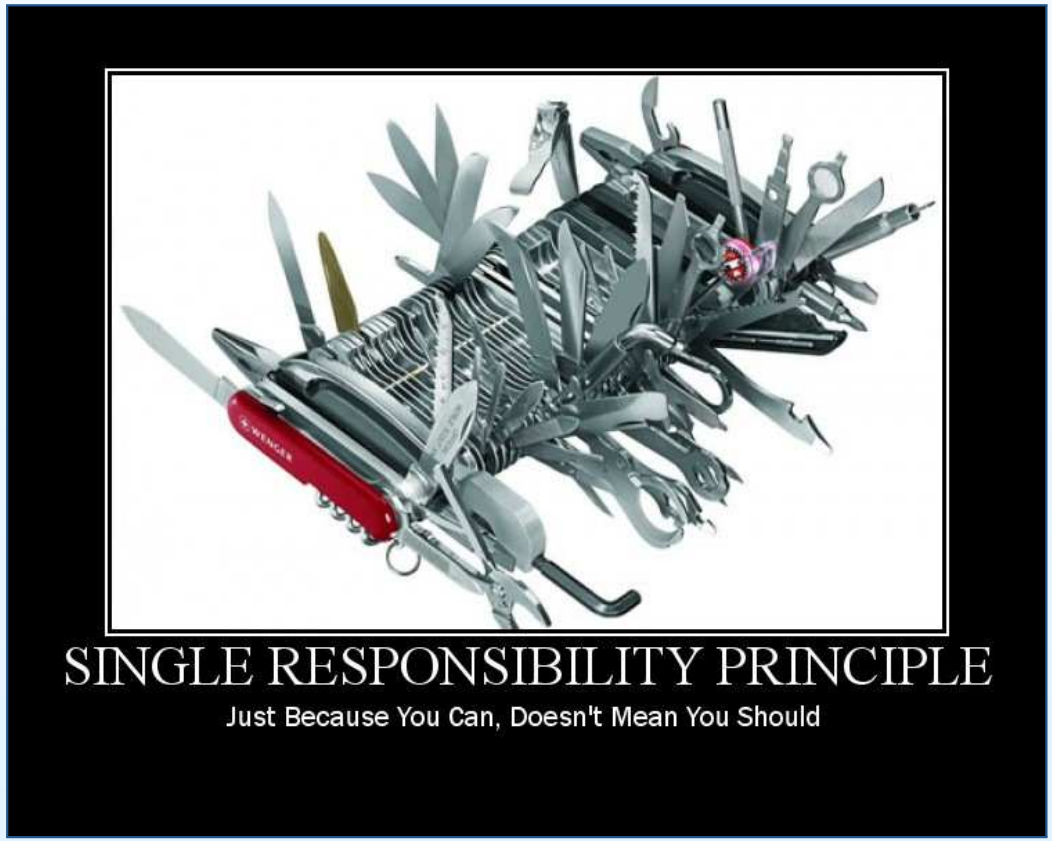
- Oferă o definiţie formală conceptului de *moştenire a specificării*
- (B. Liskov, 1988) Dacă pentru fiecare obiect *o1* de tipul *S* există un obiect *o2* de tipul *T*, astfel încât orice program *P* definit în termenii lui *T* îşi păstrează comportamentul atunci când *o2* este substituit cu *o1*, atunci *S* este un subtip al lui *T*.
- <=> (B. Bruegge) Dacă un obiect de tip *S* poate fi utilizat oriunde este aşteptat un obiect de tip *T*, atunci *S* este un subtip al lui *T*
- <=> (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 responsabilităţii unice) +
  - **O**pen Closed Principle (Principiul deschis/închis) +
  - **L**iskov Substitution Principle (Principiul Liskov al substituţiei) +
  - **I**nterface Segregation Principle (Principiul separării interfeţelor) +
  - **D**ependency Inversion Principle (Principiul inversării dependenţelor)

Principiul responsabilităţii 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*)
- <=> Nu trebuie să existe mai mult de o responsabilitate per clasă

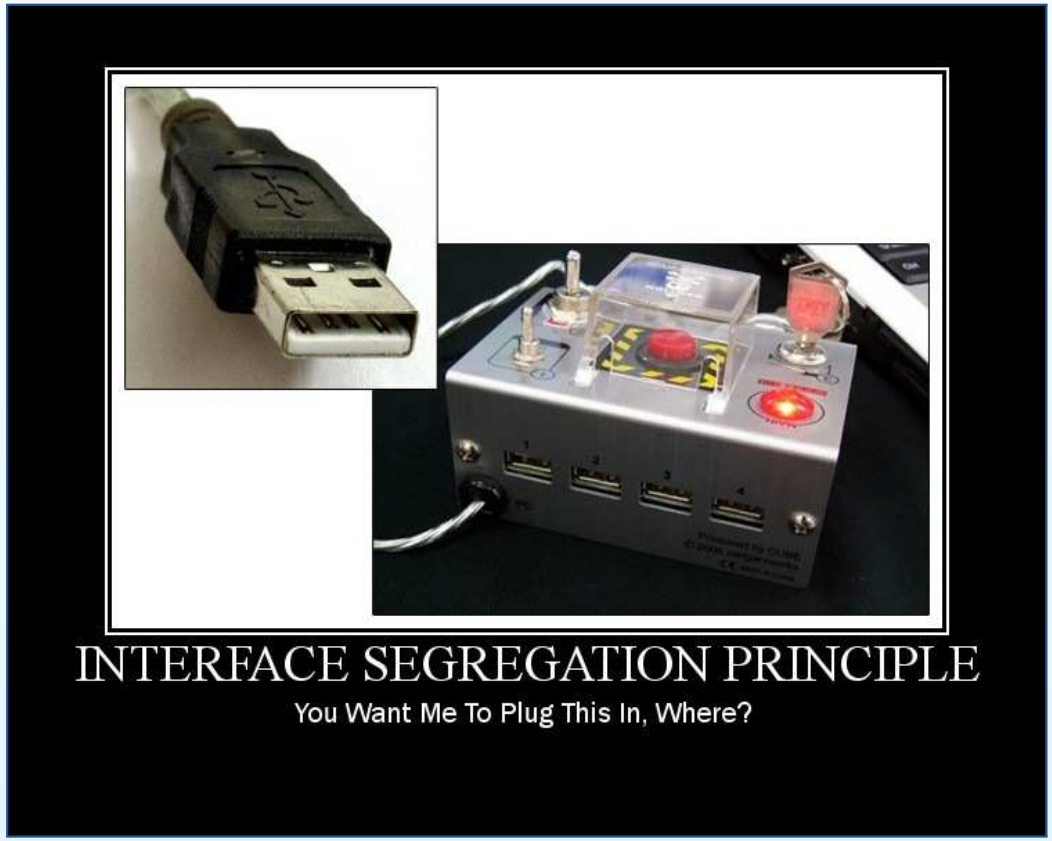


Principiul deschis/închis

- Entităţile software (clase, module, funcţii) 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*)

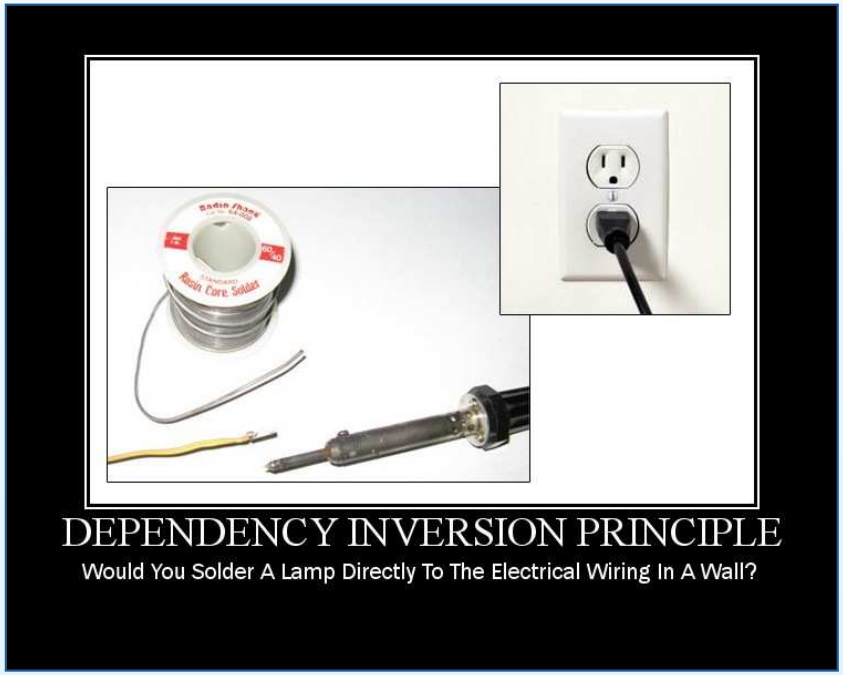
Principiul separării interfeţelor

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



Principiul inversării dependenţelor

- 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*)



Şabloane de proiectare [Gamma et al., 1994]

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

Şabloane de proiectare (cont.)

- Funcţionalitate nouă
  - testele de validare pot dezvălui absenţa unor funcţionalităţi din sistem
- O nouă complexitate a domeniului aplicaţiei
  - identificarea unor generalizări posibile ale sistemului în cadrul domeniului de aplicaţie sau modificarea regulilor de business ale domeniului
- Erori la nivelul cerinţelor sau erori de proiectare/implementare

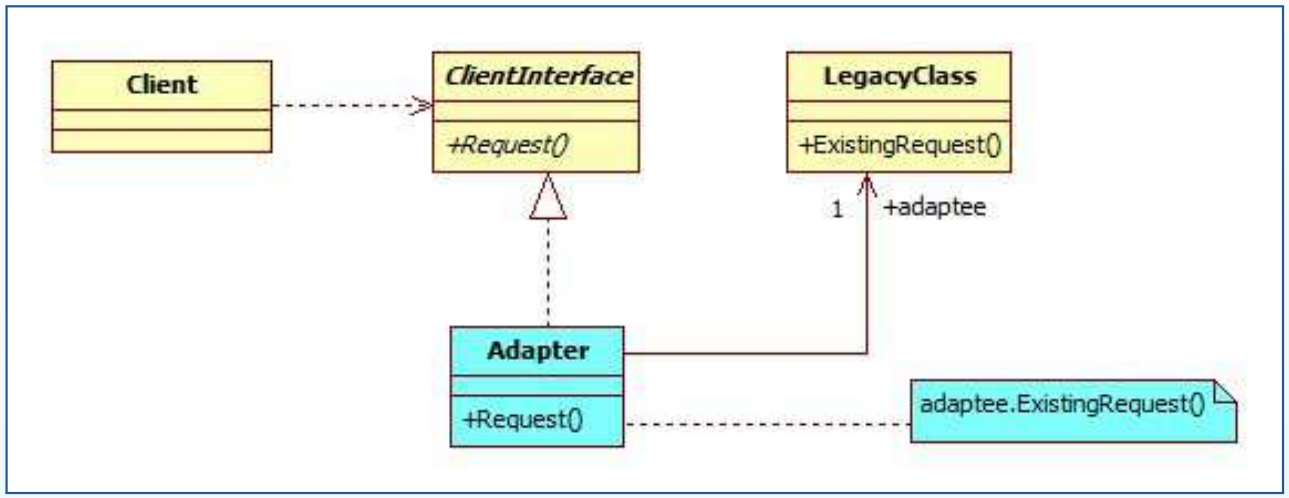
- Şabloane de proiectare adecvate acestor tipuri de schimbări
  - Furnizor nou / tehnologie nouă / implementare nouă
    - *Adapter*
    - *Bridge*
    - *Strategy*
  - Furnizor nou / tehnologie nouă
    - *Abstract Factory*
  - Funcţionalitate nouă
    - *Command*
  - O nouă complexitate a domeniului aplicaţiei
    - *Composite*

Încapsularea componentelor existente cu *Adapter*

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

Şablonul *Adapter* (cont.)

- *Structură*:



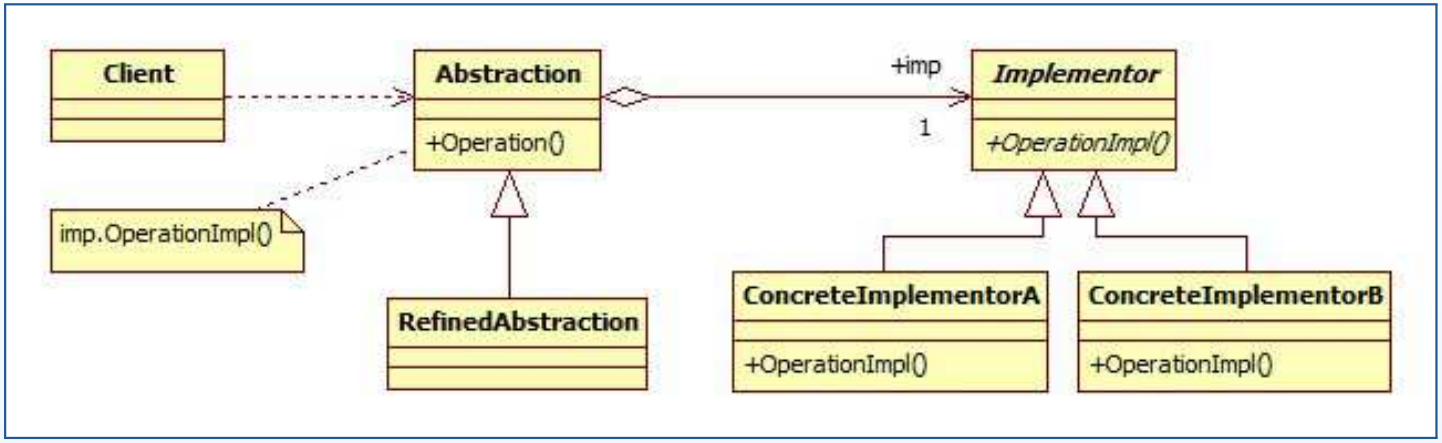
- *Consecinţe*:
  - Clasele *Client* şi *LegacyClass* pot colabora fără nici un fel de modificare la nivelul acestora
  - *Adapter* lucrează cu *LegacyClass* şi oricare dintre subclasele acesteia
  - Pentru fiecare specializare a *ClientInterface* trebuie scris un nou *Adapter*

Decuplarea abstractizărilor de implementări cu *Bridge*

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

Şablonul *Bridge* (cont.)

- *Structură*:



- *Soluţie*: Clasa *Abstraction* defineşte interfaţa vizibilă clientului. *Implementor* este o clasă abstractă, care declară metode de nivel jos disponibile clasei *Abstraction*. O instanţă *Abstraction* reţine o referinţă către instanţa *Implementor* curentă. Clasele *Abstraction* şi *Implementor* pot fi rafinate independent.

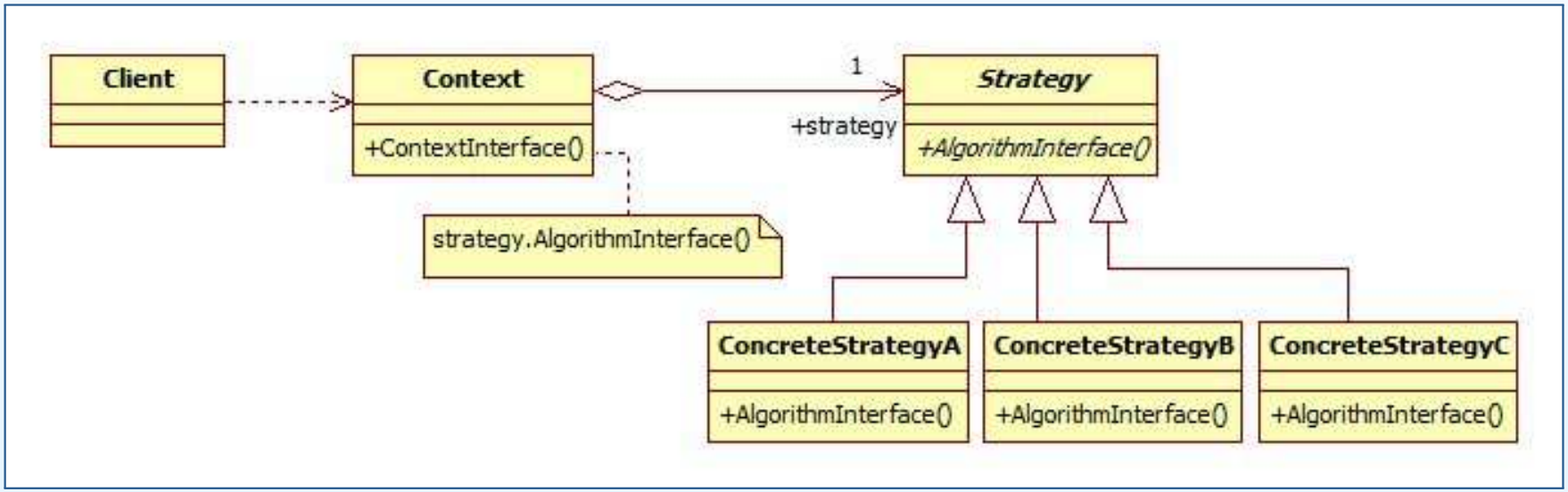
- *Consecinţe*:
  - Clientul este protejat de implementările abstracte şi concrete
  - Abstractizarea şi implementările pot fi rafinate independent

Încapsularea algoritmilor cu *Strategy*

- Presupunem existenţa unui număr de algoritmi diferiţi pentru rezolvarea aceleiaşi probleme şi necesitatea de a interschimba în mod dinamic algoritmul utilizat
  - Ex.:
    - Un editor de documente poate folosi algoritmi diferiţi pentru împărţirea textului pe rânduri, funcţie de preferinţe (rânduri de lungime fixă, împărţire în silabe sau nu, etc.)
    - într-un joc de şah, calculatorul poate folosi strategii diferite de alegere a următoarei mutări, funcţie de nivelul selectat (începător, expert, etc.)
  - Abordări posibile
    - Includerea tuturor algoritmilor la nivelul clasei care îi utilizează şi folosirea de instrucţiuni condiţionale pentru a-i interschimba => algoritmii pot fi interschimbaţi dinamic, însă clasa respectivă va avea o complexitate sporită şi va fi incomod de modificat în condiţiile 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 acelui algoritm, iar algoritmul nu va putea fi variat dinamic

Şablonul *Strategy* (cont.)

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

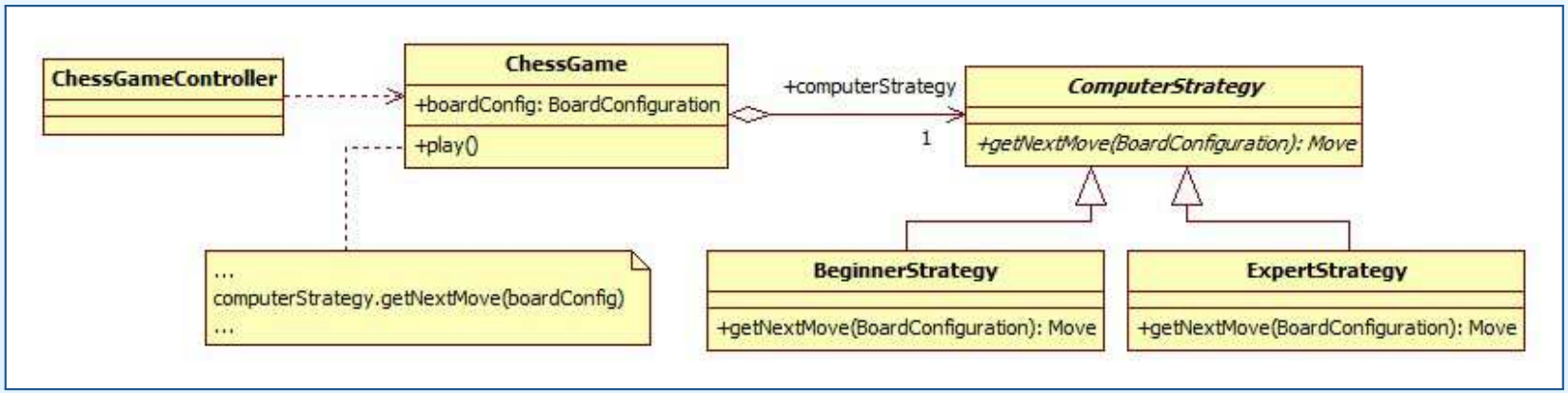


- *Soluţie*:
  - Fiecare dintre algoritmi este încapsulat/implementat de o clasă *ConcreteStrategy*. *Strategy* defineşte interfaţa comună a tuturor algoritmilor suportaţi.

Şablonul *Strategy* (cont.)

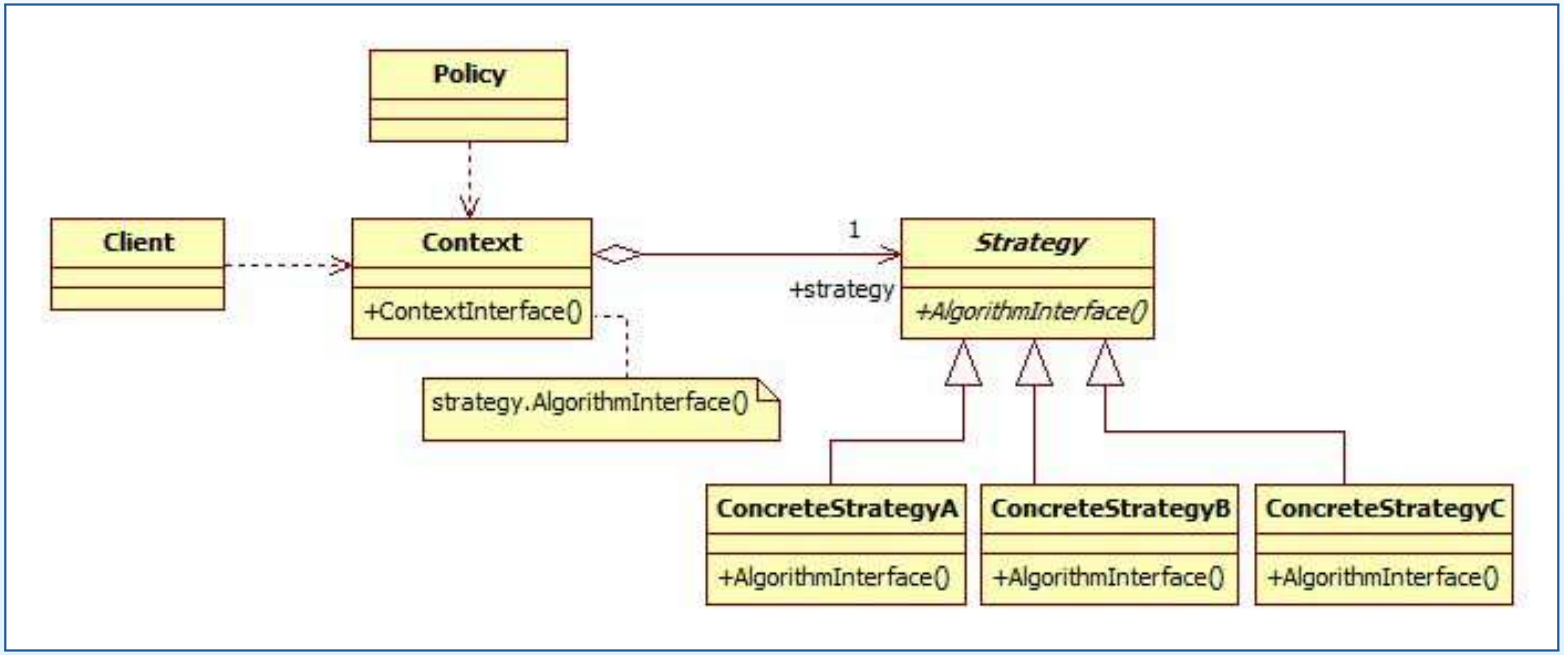
- Obiectele *Context* utilizează această interfaţă pentru a apela algoritmul definit de o clasă *ConcreteStrategy*. Contextul păstrează o referinţă către un obiect *Strategy* şi îi deleagă acestuia responsabilitatea execuţiei algoritmului, atunci când este cazul.
- Obiectul strategie este creat şi plasat în clasa *Context* de către *Client*.
- Contextul poate trece către strategie toate date necesare algoritmului, atunci când acesta este apelat. Ca şi alternativă, contextul se poate trece pe sine ca şi argument în operaţiile interfeţei *Strategy* şi poate oferi o interfaţă care sa-i permită obiectului *Strategy* sa-i acceseze datele.

- Ex.: Instanţierea şablonului *Strategy* pentru un joc de sah cu calculatorul



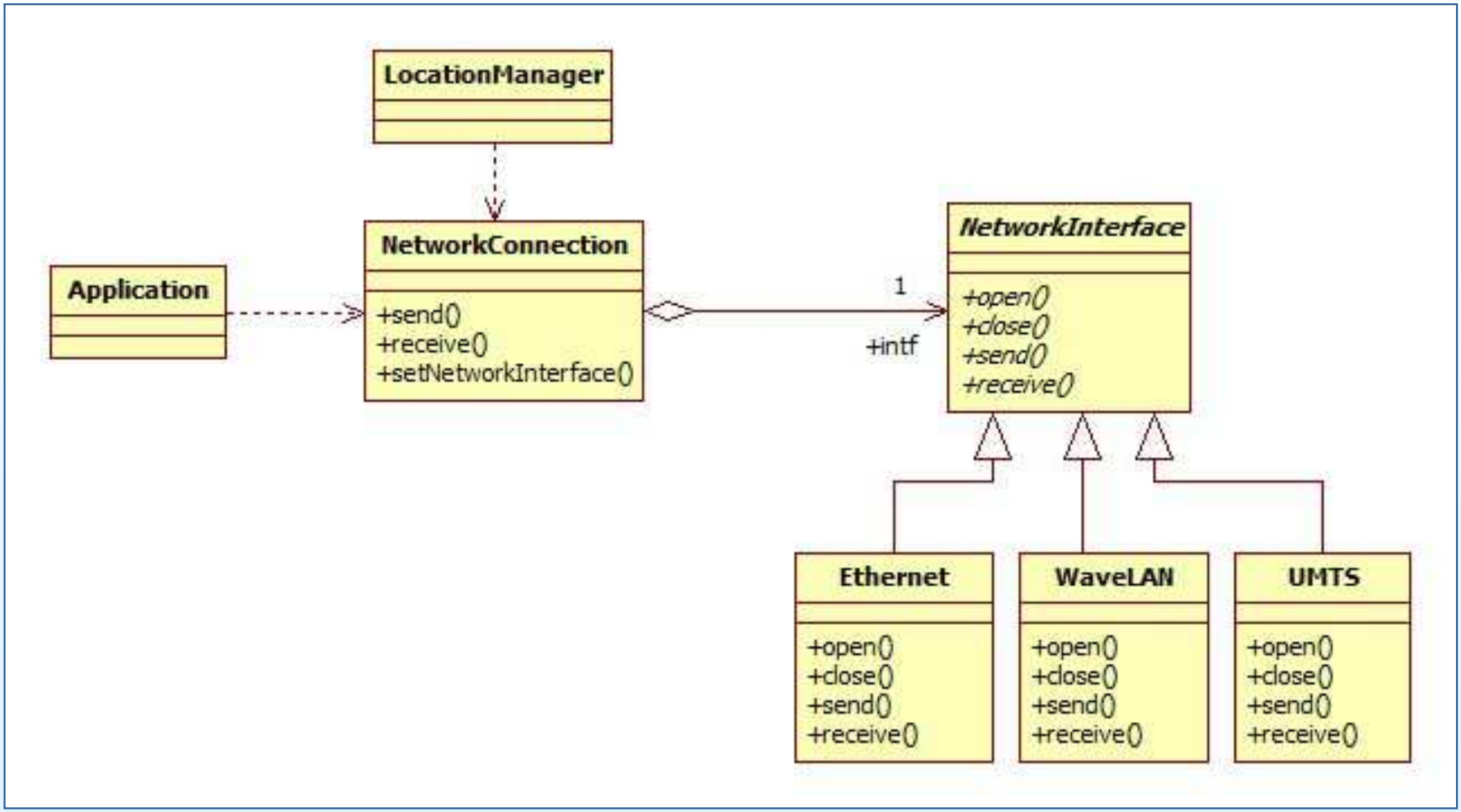
Şablonul *Strategy* (cont.)

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



Şablonul *Strategy* (cont.)

- Ex.: Schimbarea reţelei în aplicaţii pentru dispozitive mobile (instanţiere a variaţiei şablonului *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;
    }
}
```

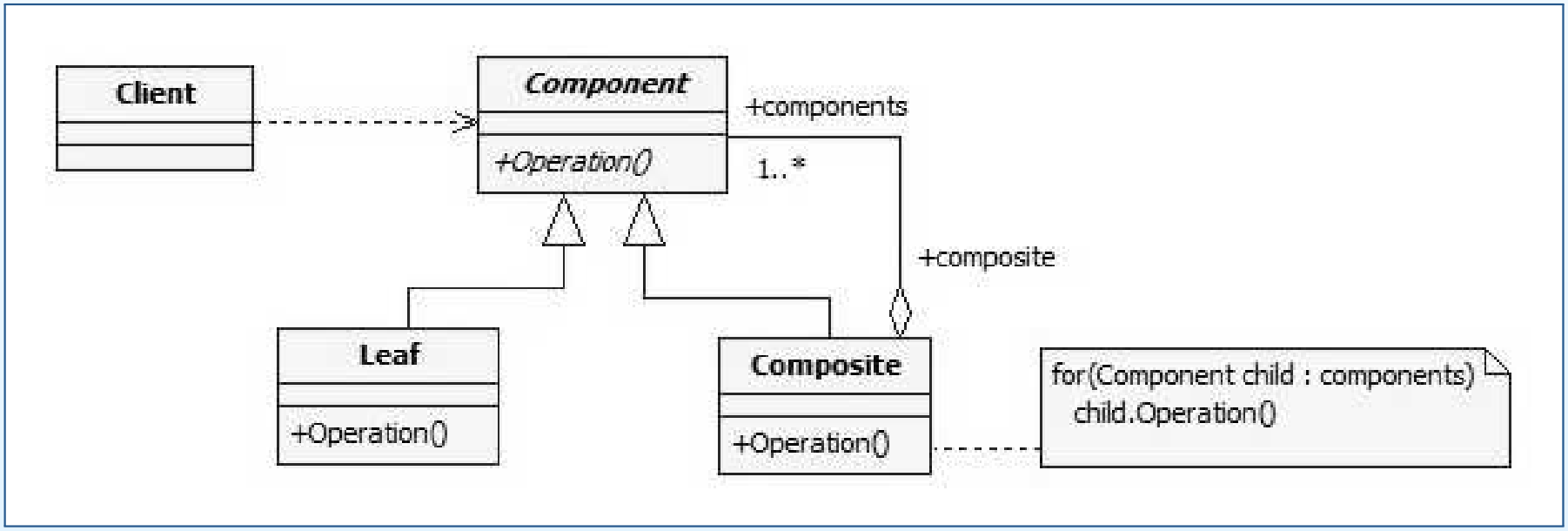
Şablonul *Strategy* (cont.)

```
/** 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*

- Şablonul *Composite*
  - *Tip*: şablon structural
  - *Scop*: Permite reprezentarea unor ierarhii de lăţime şi adâncime variabilă (recursive), astfel încât frunzele şi agregatele să fie tratate uniform, prin intermediul unei interfeţe comune.
  - *Structură*:



Şablonul *Composite* (cont.)

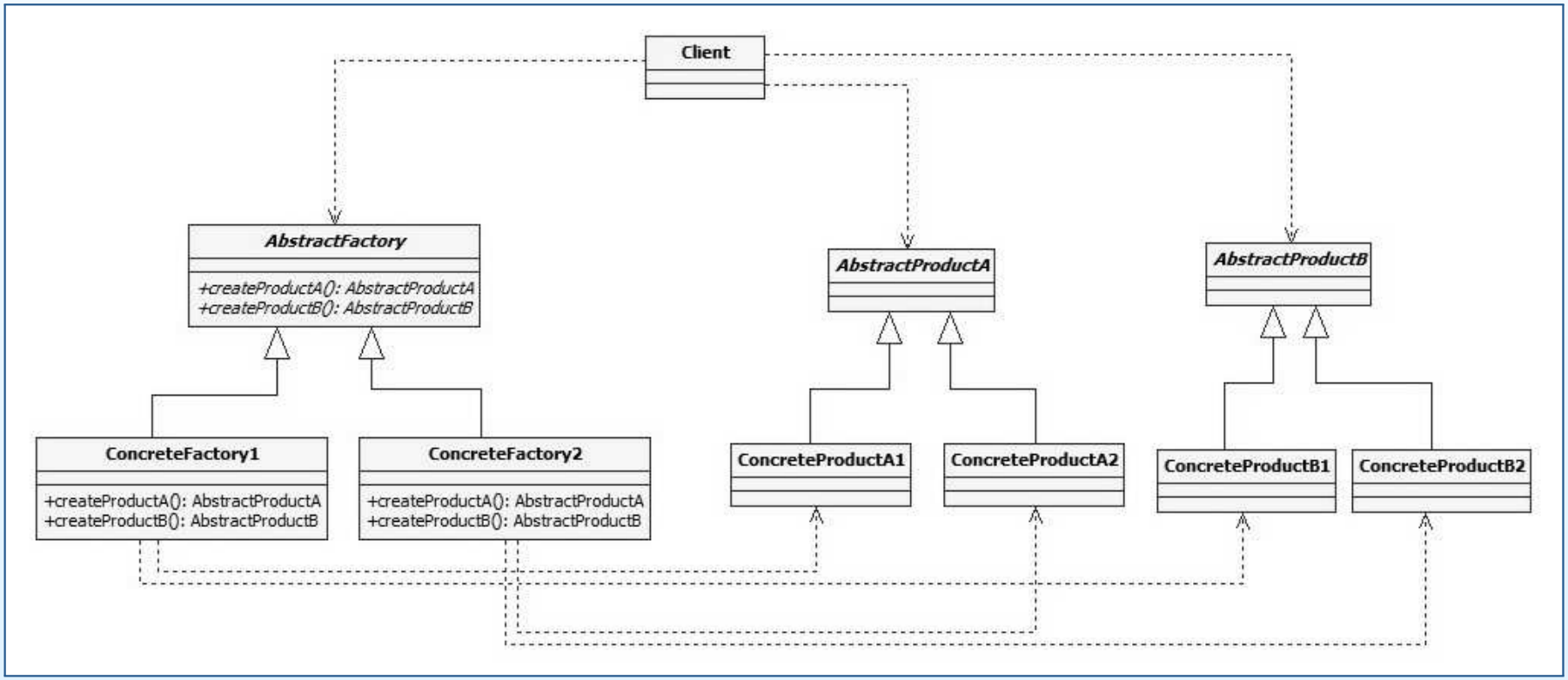
- *Soluţie*: Interfaţa *Component* specifică serviciile partajate de *Leaf* şi *Composite* (ex. *move*(*x*,*y*), pentru un obiect grafic). Clasa *Composite* agregă obiecte *Component* şi implementează aceste servicii iterând peste componentele conţinute şi delegându-le serviciul în cauză (ex. *move*(*x*,*y*) din *Composite* invocă iterativ *move*(*x*,*y*) pentru fiecare obiect *Component* conţinut). Funcţionalitatea concretă este asigurată de implementările serviciilor din *Leaf* (implementarea *move*(*x*,*y*) din *Leaf* modifică coordonatele unei primitive grafice şi o redesenează).
- *Exemple*:
  - Ierarhii de componente grafice: componentele grafice pot fi organizate în containere, ce pot fi scalate şi repoziţionate uniform. Un container poate conţine alte containere
  - Ierarhii de fişiere şi directoare: directoarele pot conţine fişiere şi alte directoare. Aceleaşi operaţii sunt folosite pentru copierea/ ştergerea amândurora
  - Descompunerea unui sistem: un subsistem constă din clase şi alte subsisteme

Şablonul *Composite* (cont.)

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

Încapsularea platformelor cu *Abstract Factory*

- Şablonul *Abstract Factory*
  - *Tip*: şablon creaţional
  - *Scop*: Furnizează o interfaţă pentru crearea familiilor de obiecte înrudite sau dependente, fără specificarea claselor lor concrete.
  - *Structură*:



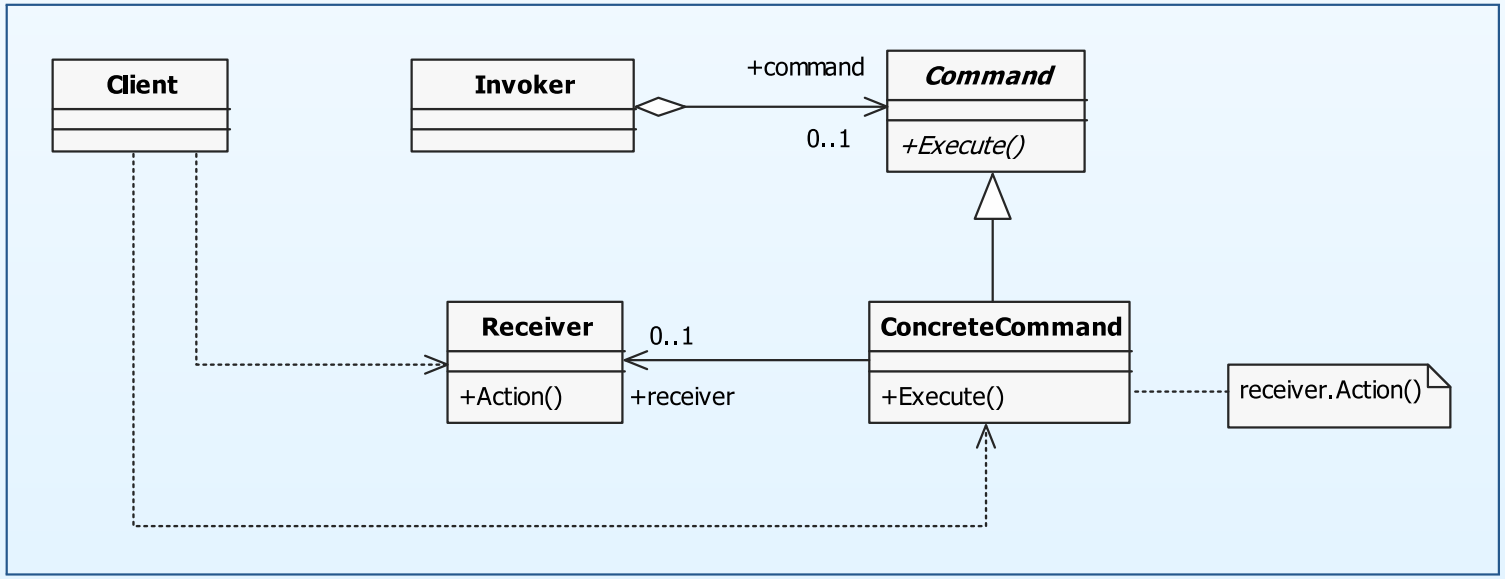
Şablonul *AbstractFactory* (cont.)

- *Soluţie*: O platformă (ex. kit pentru interfaţa grafică cu utilizatorul) constă dintr-o mulţime de produse (de tip *AbstractProduct*), fiecare reprezentând un anumit concept (ex. buton), suportat de către toate platformele. O clasă *AbstractFactory* declară o interfaţă cu operaţii pentru crearea ficărui tip de produs. O platformă specifică e reprezentată de o mulţime de produse concrete (câte unul pentru fiecare produs abstract), instanţiate de un *ConcreteFactory* (acesta din urmă depinde doar de produsele concrete pe care le instanţiază). Clientul depinde doar de produsele abstracte şi de clasa *AbstractFactory*, fapt ce facilitează substituirea platformelor.

- *Consecinţe*:
  - Clientul este decuplat de clasele produs concrete.
  - Este posibilă substituirea familiilor de produse la runtime, prin înlocuirea clasei *ConcreteFactory* utilizate.
  - Adăugarea unor noi tipuri de produse este relativ dificilă, întrucât presupune modificarea interfeţei *AbstractFactory* şi a claselor *ConcreteFactory* existente.

Încapsularea fluxului de control cu *Command*

- Şablonul *Command*
  - *Tip*: şablon comportamental
  - *Scop*: Încapsulează o cerere ca şi un obiect, permiţând parametrizarea clienţilor cu diferite cereri, precum şi formarea unei cozi sau a unui registru de cereri şi asigurarea suportului pentru operaţiile ce pot fi anulate (facilităţi *undo*).
  - *Structură*:



Şablonul *Command* (cont.)

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

Şabloanele *State* şi *Singleton*

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

Referinţe

- [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.